Windows Programming

Dr. Dobb's Journal

March 2013

# New Socket APIs in Windows 8

## Important extensions to the venerable Microsoft Winsock stack

Next

ALSO INSIDE



UBM Tech

www.drdobbs.com

Dr. Dobb's Journal

# CONTENTS

March 2013

## FEATURES



## GUEST EDITORIAL



## More on DrDobbs.com

**Jolt Awards: Coding Tools**

The best IDEs and coding tools of the past year.

http://www.drdobbs.com/240147989

**A Strategy for Defining Order Relations**

If you want to compare two complicated objects, the strategy is to define a function that computes a single number from each object and then compares those numbers.

http://www.drdobbs.com/240147625

**Why NoSQL Is Here To Stay**

While they won't displace traditional RDBMSs, the easy scalability and programmability of NoSQL databases guarantees them a permanent place in the data center.

http://www.drdobbs.com/240146171

**Mobile as the Driver of Desktop Software Design**

The metaphors and conventions of mobile apps on phones and tablets are driving the design of desktop software. The tail is definitely wagging the dog. Good dog!

http://www.drdobbs.com/240146757

**Gesture-Based Computing for the Desktop**

The Intel Perceptual Computing SDK works with a webcam to add motion/gesture sensing and facial recognition to desktop apps. This first release shows considerable promise, with some important limitations.

http://www.drdobbs.com/240146740

# Win8 Development: The Lessons Learned

Know what combination of technologies to use, design your architecture correctly, and differentiate your app from others — these are the fundamental practices of coding in the age of Windows 8.

**By Chris Sells**

I started my Windows 8 programmer career in September 2010, when I moved into my new office at Microsoft, down the hall from the Visual Studio 2012 team and the Windows Library for JavaScript team (WinJS). Having now written programs for two years on an operating system that's been generally available for only four months means that I'm often asked how programmers should approach this new platform.

To start, I'm going to point you to the tools and resources you'll find at Windows Dev Center (http://is.gd/mhNq1X). After that, I recommend the videos from Microsoft's last BUILD conference (http://is.gd/3K8MqA). Beyond the basics, I can also give you some advice that might not be immediately obvious to you from reading the docs. That's what the rest of this article is about.

## Know Your Options

Before anything else, you should understand your options. To build first-class Win8 apps, you can use .NET and XAML with C# or Visual Basic. You can also write apps with HTML and JavaScript, along with the rest of the Web platform, (CSS, Canvas, SVG, and the like). And you can write native apps using C++ and either XAML or DirectX.

However, if you use .NET or JavaScript, remember that you're bringing in the overhead of the accompanying runtime (the .NET runtime or the Web platform), whereas if you use C++, not only do you have native access without a runtime, you also have access to parts of the Windows API that aren't projected into the Windows Runtime (the latter being the set of object APIs that are shared between JavaScript, C#, Visual Basic, and C++). If you need access to these other APIs or Direct3D or if you're looking for the best performance, then you'll definitely want to go native.

If you do go native, remember that Windows 8 is no longer exclusively x86 — you'll need native binaries for both x86 and ARM if you want to cover the full range of available Win8 devices. On the other hand, .NET and JavaScript apps run just fine across x86 and ARM. If you're not building an app that needs the performance of a twitch game, but

rather a front-end to an Internet data source, a media player, or even a casual game, you'll find the hardware-accelerated performance of XAML or HTML to be excellent, not to mention the huge amount of existing code provided by the community. You'll also enjoy the higher productivity that a managed environment provides C#, VB, and JavaScript programmers.

In fact, both HTML and XAML have been optimized so thoroughly that you won't be able to tell the difference between an app written with XAML and .NET from one written in HTML and JavaScript. So which managed environment do you choose? The one you're the most comfortable with. Do you have a lot of experience with .NET and some libraries to bring forward to a new Win8-style app? Use .NET. Do you have a bunch of Web developers who like jQuery plug-ins and CSS? Use JavaScript. Follow your preference — both choices are extremely capable.

### Architecture of the App

Picking a technology is only one piece of the technical puzzle, however. You also need to be careful about how you structure your app. As an industry, we've moved from client-only to client-server to n-tier to our current favorite: hub-and-spoke. The hub-and-spoke architecture provides the basic structure for most modern mobile apps: At the hub, you've got your server, which provides data and logic synchronized between multiple client spokes, such as Windows Phone, Win8, Web, iOS, Android, etc.

Each spoke is designed and optimized for the specific host OS and the hub is the authoritative holder of the data shared between all clients. This kind of architecture calls for several considerations, including communication channel decisions (pull over HTTP, push over SMS,

etc.), offline caching support, per-client data projection and filtering, and so on. As each spoke is added, you'll find that the hub often needs to change to support the particulars, so you'll need to think about the user experience you want and make those hub design decisions accordingly.

Specifically, when it comes to Windows 8, you'll want to tailor a hub-and-spoke client for things you've undoubtedly already heard about: a touch-centric UX, Win8 UI style, fast and fluid animations, and so forth. What you might not have heard about is the three primary rules for a high-quality Win8 app: integrate, integrate, integrate! In prior versions of Windows, we had the Clipboard, which allowed each app to share whatever data it wanted in whatever formats it used. In Windows 8, there is still a Clipboard, of course, but it is only one of many "contracts" for sharing data between apps.

For example, the SkyDrive app not only allows you to browse the files in your SkyDrive account, but it's also a File and Folder "provider," so when you want to open a file from a Win8 app, the file and folder dialogs know to let you choose something in SkyDrive as well as on your local computer. The app that's requesting the file doesn't need to know or care that the file is coming from SkyDrive — the operating system takes care of that, in the same way it maintains the Clipboard. Other contracts include Search, Settings, Share, and Contact Picker, just to name a few.

In addition to the contracts, which allow you to integrate your app with other installed apps, the Win8 Start Screen also enables a great deal of integration with Live Tiles, Badges, Notifications, and the Lock Screen. Live Tiles are especially important, as they represent a key differentiator of the "Metro style" UI shared between Windows Phone, Windows 8, and the XBOX.

**Differentiate**

And speaking of differentiators, another key thing you should do in your app is just that: differentiate. When you read the guidelines that describe Win8 UI style (http://is.gd/jhZ4Yk), you'll hear a great deal about the Windows 8 way of arranging data, using the contracts, providing the right animations, consuming the right gestures, using the right fonts, and so on. However, once you've gotten your head around the Win8 UI style, you'll need to bend or even break some of the rules so that your app stands out from the crowd. Your app must have personality as well as functionality, if it's going to be featured as a top app in the Store and get you the reputation and revenue it deserves.

With Windows 8 and the Windows Store, we're at the beginning of a new era of Windows, which brings devices, touch, and UX design to the foregroud — while keeping desktop apps running well, too. The future is going to bring great changes as we learn how to design apps for this new world. Stick around. We're just getting started!

***— Chris Sells (http://www.sellsbrothers.com) is the VP of the Developer Tools Division at Telerik. He's written several books, including* Building Windows 8 Apps in JavaScript (http://is.gd/W3kHkV), Programming WPF (http://is.gd/8nqWm6), Windows Forms 2.0 Programming (http://is.gd/kCnE8h), *and* ATL Internals (http://is.gd/etS6i4). *In his free time, Chris makes a pest of himself on Microsoft forums and mailing lists.***

Comment

# The New Socket APIs in Windows 8

Windows 8 and Windows Server 2012 introduce important extensions to the venerable Microsoft Winsock stack. For some apps, the new APIs are your only option; for others, they might be your best performing option.

**By Stephen Baker**

Microsoft Windows 8 and Windows Server 2012, the latest operating system and Windows makeover, were released in fall 2012. For programmers, many new APIs were included in the features added. Several of the new features focus on the venerable sockets interface, the basic network paradigm first introduced in Berkeley Software Distribution (BSD) UNIX in the early 1980s. Sockets still form the basis for networking on all of the major platforms including Windows, Mac OS, Linux, iOS, and Android.

You might want to consider these new Windows socket APIs for your next project when writing new apps or refactoring older applications for Windows. New Windows 8 features for sockets programming are found at both ends of the network spectrum:

- Window Runtime (WinRT) sockets: Used for low-level networking in Windows Store apps.
- High-performance sockets: Used in writing network servers for desktop apps that need low network latency and the highest performance.
- WebSockets: The other new networking APIs are socket-related, at least in name; these are extensions to the HTTP protocols for the creation of a special Web socket that is fully bidirectional — once the connection is established, either endpoint can initiate or send packets.

Herein I provide a brief introduction to these new networking socket features available on Windows 8 and Windows Server 2012. The focus will be more on WinRT sockets, since these will appeal to a larger set of developers.

## Sockets and Windows

The original Berkeley socket implementation developed in 1983 was very simple, based on the design of UNIX file I/O. A network socket became a special type of file handle. BSD networking functions were basic: socket (`open` or `create`), `connect` (for TCP), `send`, `recv`, and `close`. Several types of sockets were supported including TCP (stream), UDP (datagram), and later UNIX domain sockets. For TCP servers and appli-

cations receiving UDP packets, the `bind` and `listen` functions would bind to a network address and listen for incoming packets. For use with UDP, there were the `sendto` and `recvfrom` functions. A few other functions were provided for getting and setting socket options, getting a host by name or address; and there were the `select` and `poll` functions to check on the state of a socket.

With the adoption of Windows Sockets (Winsock) in 1993, Microsoft added support for sockets to Windows. The initial Winsock 1.1 API was relatively modest with calls similar to BSD UNIX plus a number of Windows-specific extension functions. Over time, the Winsock API has grown larger and larger. The current reference documentation for Windows Sockets for desktop apps (http://goo.gl/amGqs) lists more than 140 functions and 80 structures along with several hundred IOCTLS and socket options. This doesn't include functions used by the Winsock Service Provider Interface (SPI). The existing Winsock API has become very large, complex, and challenging for new users.

The new socket APIs introduced with Windows 8/Windows Server 2012 provide much simpler, stripped-down APIs for sockets programming — closer to the spirit of the original BSD sockets.

## Windows 8: Two for the Money

Windows 8 and Windows Server 2012 introduce a new graphical user interface (GUI) that represents a major redesign of the Windows user experience. The changes are targeted primarily for touch-enabled devices, with tiles used to represent apps and new system navigation features (charms, settings, etc.). On traditional Intel/AMD x86/x64 hardware, Windows 8 can be thought of as an operating system for two different types of apps:

- Windows Store apps: New apps that run on Windows 8 only. These apps are limited to using the WinRT APIs and a few other APIs (some classes from the .NET framework along with a few desktop functions) that are exposed for use by Windows Store apps.
- Windows desktop apps: Traditional Windows apps that provide developers access to all of the Windows APIs except the WinRT APIs (unless the WinRT class has a special `DualApiPartitionAttribute`). These represent the traditional applications that ran on Windows 7 and older versions of the OS. If you use new Windows 8 desktop APIs, then the app will only be able to run on Windows 8 and Windows Server 2012 or later versions.

On ARM-based hardware (in the original Microsoft Surface tablet, for example), Windows 8 has been stripped down so only Windows Store apps can be installed and used. Windows 8 apps using WinRT are installed by purchasing and downloading them from the Microsoft App Store (many apps are free). Windows and app upgrades are also downloaded and installed from the Microsoft Store. Sideloading apps on these WinRT-based devices is mostly restricted, except when Visual Studio 2012 is installed. Sideloading allows developers to test apps they write before submitting them for publication on the Windows App Store.

(Note: The Windows App Store (http://www.windowsstore.com/) is not the Microsoft.com online store that sells versions of Windows, Office, other Microsoft software, and some hardware. There is also a separate Windows Phone App Store (http://www.windowsphone.com/store) for purchasing and downloading apps for Windows phones.)

## Windows Runtime and Sockets

The WinRT APIs used by Windows Store apps provide a set of managed APIs that are designed for several different languages and presentation schemes:

- JavaScript with HTML: Designed to appeal to traditional Web developers.
- C#/VB.NET with XAML: Designed for existing C#/VB.NET developers and others using managed code (Java developers).
- C++ with XAML: Designed for core C++ desktop and COM developers as well as others using similar languages (Objective-C developers).

Developers are free to choose whatever language and presentation they prefer. In fact, the same app can be written in any of these languages. Microsoft provides downloadable samples for Windows Store apps, many of which are implemented in multiple languages for illustration. For example, the downloadable `StreamSocket` sample (http://goo.gl/Q7mz7) and the `DatagramSocket` sample (http://goo.gl/ROndB) are implemented in all three language combinations.

WinRT sockets provide a low-level socket interface for use with TCP and UDP sockets. The features are exposed in the `Windows.Networking.Sockets` namespace (http://goo.gl/1iJAh"). WinRT sockets also use classes in the `Windows.Networking` namespace (http://goo.gl/KCuLh) to provide access to managed `Hostname` and `EndPoint` classes. WinRT sockets are needed if other higher-level WinRT networking APIs (web access, Atompub, JSON, and background HTTP/FTP transfer) don't meet the requirements of your app. WinRT sockets are the building blocks a developer can use to build apps that use TCP or UDP protocols to access services or peers. Some typical examples might be Voice over IP (VoIP), instant messaging, mail clients (IMAP, POP, SMTP), and database clients. A very simple example using sockets might be an app that sends status messages to an SMTP server or an app that logs status to a SYSLOG server.

The WinRT API for sockets provides a simple API for sockets programming with three primary classes:

- `StreamSocket`: A TCP stream socket (http://goo.gl/d4Vto).
- `DatagramSocket`: A UDP datagram socket (http://goo.gl/hte8M).
- `StreamSocketListener`: A listener for incoming network connections using a TCP stream socket (http://goo.gl/ylmNQ).

The `StreamSocket` and `DatagramSocket` classes implement the core TCP and UDP sockets. The `StreamSocketListener` class is a helper class that implements a TCP listener. Once a connection request is received and accepted, a `StreamSocket` object is created.

Associated with each of these classes are related `Control` and `Information` classes that provide socket control data and information for each of the primary classes. All WinRT socket methods are designed to be asynchronous and not block the UI thread. They even have `Async` appended in the method name to make this clear.

Developers should be aware of some limitations placed on Windows Store Apps using sockets in Windows 8:

- TCP and UDP sockets are supported. Raw sockets and other types of sockets are not supported. So, for example, you can't write a ping tool that needs access to ICMP or ICMPv6.
- Sandboxing prevents a Windows Store app that uses sockets from accessing IPv4 or IPv6 loopback addresses on the device.

This limitation is removed for apps running under the Windows debugger in Visual Studio to allow testing of client/server apps on the same device.

- The `StreamSocket` class has built-in support for SSL for client apps, but not for servers. It would be much more difficult to implement a TCP server that uses SSL because the app would need to implement all the SSL protocols.
- When a networking app using WinRT sockets loses focus, the app can't continue to send or receive packets. There are some special features (`Windows.Networking.Sockets.ControlChannelTrigger`) that can be used to support background networking with a `StreamSocket`, but they are complex (http://goo.gl/kUyaA).

**Windows Store Apps and Lifetimes**

There are some important implications when developing apps using WinRT. Each Windows Store app is run in sandbox by the OS for security and protection. This prevents a rogue app from accessing data or information from another app except through controlled mechanisms that require approval of the apps. The sandbox prevents apps from sharing sockets.

Windows Store apps must declare in a manifest what capabilities they plan to use. Several of these capabilities deal specifically with network access (Internet client, Internet client/server, or Intranet client/server). When the app is installed, the user must approve the use of these capabilities.

**Windows Store apps operate under a very different app lifecycle model than Windows 8 desktop apps**

Windows Store apps operate under a very different app lifecycle model than Windows 8 desktop apps (and applications on Windows 7 and earlier). When a Windows Store app loses focus (that is, gets dragged away from the foreground and replaced by another app), the app is not multitasked as on previous versions of Windows. The app doesn't get any more CPU cycles and may be purged from memory. Before the OS switches to the new app, the existing app gets notified

and has a short time to save state before it is cut off from further execution. Many network operations often take time to complete. So this app lifecycle model for Windows Store apps has important implications for the design and implementation of network apps. Also, when used on mobile devices, network connectivity may be lost. Caching network content becomes an important feature to include.

### High Performance Winsock and Registered I/O

At the other end of the Winsock continuum, Windows 8 and Windows Server 2012 introduced new APIs for Winsock registered I/O (RIO) extensions (http://goo.gl/JvNyU). These APIs, which are usable only by desktop apps, are a set of Winsock extensions designed to lower latency and jitter and improve the network performance of network servers. While these APIs can be used by both clients and servers, network performance improvements, when compared with using traditional Winsock APIs, are likely to occur only on heavily-loaded network servers. The greatest improvements would be on servers that handle a large number of small network packets (~1K), which is typical of database servers and some applications used in the financial services industries.

The goal of Winsock and network stacks on other operating systems has always been to minimize data copying. For sends and receives, the regular Winsock APIs and the core TCP/IP stack use network buffers that must be pinned and unpinned by the system. The OS management of these buffers used by Winsock requires CPU cycles and user-mode to kernel-mode transitions to lock and unlock the buffers in memory. There are also CPU cycles associated with the methods used for I/O completion.

The design model for the RIO extensions uses pre-registered data buffers and completion queues to increase performance. The pre-registered data buffers are locked into memory and stay locked until the data buffers are released by the app. The data buffers can be used over and over again. It is assumed that a server app would maintain a pool of these pre-registered data buffers sized appropriately. RIO uses completion queues (RIO_CQ structures) and request queues (RIO_RQ structures) that are managed by the application. The completion queues are used to indicate when an I/O operation has been completed. A RIO request queue represents a RIO socket descriptor with associated completion queues, configuration data, and a Windows socket.

### Using RIO

All RIO functions are extensions to Winsock. An application must call the `WSAIoctl` function with a special GUID to retrieve a set of pointers to the RIO functions at runtime.

The RIO API is small and divided into two types of functions:

- Buffer, queue, and notification operations: Used for registering and deregistering buffers, creating, resizing, and closing completion queues, creating, resizing, and closing request queues, and notification.
- Send and receive operations: Used to send or receive network data on a connected RIO TCP socket or a bound RIO UDP socket.

For an application using RIO, the normal Winsock functions are used for connect, accept, and listen operations. This does restrict an application using RIO from sending or receiving data as part of a connect or accept operation. Sending or receiving data on a connect or accept call is uncommon, so this limitation doesn't usually pose a hardship. RIO extension functions are used for send and receive operations.

An application creates a standard Windows socket using the socket, `WSASocket`, `accept`, `AcceptEx`, or `WSAAccept` functions. The `RIOCreateRequestQueue` function creates a RIO socket descriptor associated with the standard Windows socket. The `RIOCreateRequestQueue` function also assigns completion queues for send and receive operations to the RIO socket descriptor and sets other configuration data on the RIO socket descriptor.

Two options are available for completion queue notification:

- Event handles
- I/O completion ports

RIO does not provide any access synchronization to the completion queues and request queues. If multiple threads are needed by an app to access the same queue, then the developer must implement any needed synchronization or locking.

## Fast Loopback

Another enhancement for performance was also made to Winsock on Windows 8/Windows Server 2012. A new `SIO_LOOPBACK_FAST_PATH` IOCTL was added to lower the latency and improve performance of TCP over loopback. On Windows, packets using the IPv4 or IPv6 loopback addresses use the network layer. This IOCTL changes the behavior so that TCP packets over loopback use the transport layer instead. This shortens the code path a loopback packet must traverse in the OS, resulting in lower latency and higher performance. This IOCTL can be used with the RIO extensions when using loopback.

To use this enhancement, both sockets using loopback (the client and server) must set this IOCTL. The socket that plans to issue the connect must set this IOCTL before making the connect call. The listening socket must apply this IOCTL before accepting an incoming socket. A limitation of this enhancement is that it can't be used with Windows Filtering Platform (WFP) filters. Another minor limitation is that only a limited number of socket options can be used when this IOCTL is set.

## Socket Connected

We've taken a brief look at two new socket programming APIs added with Windows 8. Upcoming articles will dig into details on how to develop apps with these new socket APIs and improve your connected performance.

## Further Information

These links provide additional information and explanation.

- Windows Runtime Sockets: Connecting with sockets (http://goo.gl/Rz1R1)
- `Windows.Networking.Sockets` (http://goo.gl/Xvajt)
- `StreamSocket` Sample (http://goo.gl/0xBLl)
- `DatagramSocket` Sample (http://goo.gl/itkQW)
- Winsock RIO Extensions and fast loopback: What's New for Windows Sockets (http://goo.gl/vkaWU)
- `SIO_LOOPBACK_FAST_PATH` control code (http://goo.gl/rzlKu)

***— Steven Baker works for Microsoft as a senior programming writer on networking. Previously, he worked for many years at the Oregon Department of Energy. Earlier, he wrote articles for several print computer magazines including the networking column for* UNIX Review.**

Comment

# Prevent Cross-Site Scripting in ASP.NET Web Apps

Cross-site scripting threats can be greatly minimized by proper encoding. On ASP.NET apps, the Microsoft AntiXSS Library is one of the easiest ways to do the encoding correctly.

**By David Ross**

Cross-Site Scripting (XSS) is the most pervasive vulnerability present in Web applications today (http://is.gd/t564Jp). That being said, it is possible to build Web apps that are impervious to XSS by arming yourself with an understanding of the threat and a basic toolbox of encoding functions.

## XSS Review

The attack occurs in a variety of scenarios where data is taken in by your website and then replayed to the user as an executable script. For example, imagine navigating to the following URL:

```
http://www.contoso.com/shopping?name=
```

If the website were to replay the query string parameter into its HTML markup verbatim, malicious script would execute on the page. Given the same-origin policy security model of the browser (http://is.gd/tvsvFT), this script could perform actions or access data on behalf of the user behind the keyboard.

There are numerous other ways that an XSS vulnerability might arise. For example, imagine your Web application presents a page with a list of users. If one of the users managed to set their visible name to a `SCRIPT` element, we then have XSS, though this scenario does not involve query string parameters per se.

Alternatively, consider a situation where an `onerror` attribute results in malicious script execution (as opposed to a `SCRIPT` element). How many mechanisms like this exist within HTML/JavaScript that enable script execution? It turns out that there are a lot (http://html5sec.org/). Fortunately you don't need to be an XSS expert to prevent XSS vulnerabilities from being introduced in application logic.

## Best Practice

Generally speaking, the strategy to pursue in building application code is to encode potentially untrusted content appropriately for the context in which it's being output on the page.

It's worthwhile to define these terms. Potentially untrusted content could be input from the user to the website, or even information stored in a database. If your Web application takes input from the URL, that data is potentially untrusted content because the data could have been supplied by an attacker. Information from cookies is not generally di-

rectly suspect because of restrictions enforced by the same origin policy; however, if the data originally came from the URL or an HTTP `POST`, you should consider it suspect. Perhaps the easiest way to define potentially untrusted content would be to say that it's any content that the application did not itself define statically in its own business logic.

The context on the page into which the output is placed is also very important to consider as it dictates how you must encode output. Consider the following example in ASP.NET:

```
<a href=
"http://contoso.com/app.aspx?var=
        <%:Server.UrlEncode(UntrustedVar)%> ">
<%: UntrustedTitle %>
</a>
```

Notice that the `UrlEncode` function is used to encode query string data (http://is.gd/6gvi1x). It's an IIS 6.0 function that converts spaces to + signs and `non-alphanumeric` characters to their hex equivalents. The default `HtmlEncode`-based encoding is used in the context of HTML (http://is.gd/WYU4tx). To understand why different encoding mechanisms are necessary, consider what malicious input might look like if encoding were not in place. In the HREF case, the output might close off the attribute and append a new attribute that would run script, for example:

```
" onload=[Malicious script]
```

Whereas in the second case, an effective attack would be:

```
<script>[Malicious script]</script>
```

So the various encoding mechanisms must encode different sets of characters to offer effective protection. (In addition, URLs are percent-encoded so that they may be properly parsed by browsers and Web servers, whereas HTML markup is encoded into HTML entities. Using the wrong encoding would create URLs or markup that can't be properly parsed.)

It is important to understand that each distinct output context requires a different encoding method. Other notable contexts include XML (attributes and markup), CSS, and JavaScript strings.

There is one issue worthy of note: Your application might hand off data to an external control or API to render on the page. In such a case, what encoding should be applied? To find out, you may need to evaluate the security guarantee provided by the external code. It seems reasonable to assume API input is encoded appropriately for the output context, although any particular API might, in fact, push that responsibility to its caller. The documentation for any good API should specify any required encoding necessary to ensure that output is rendered securely on the page. The <%: %> syntax in ASP.NET 4 and later provides a clever solution to this problem, utilizing a new `HtmlString` type (http://is.gd/kiuZ0i).

### The Microsoft AntiXSS Library

All major Web platforms provide some sort of API for output encoding. Microsoft's implementation for ASP.NET is a library of encoding functions referred to as the Microsoft AntiXSS Library (http://is.gd/Z8Ozve). This library has been available since ASP.NET 4.5.

The first thing you'll want to do to leverage the AntiXSS Library on ASP.NET 4.5 is to enable it as the default encoder by adding the `encoderType` attribute to your Web.config file:

```
<httpRuntime ...
    encoderType=
       "System.Web.Security.AntiXss.AntiXssEncoder,
          System.Web,
```

```
        Version=4.0.0.0, Culture=neutral,
           PublicKeyToken=b03f5f7f11d50a3a"
/>
```

This entry will cause default output encoding functionality in ASP.NET to use the conservative AntiXSS Library encoding. In addition, you may then begin to utilize APIs in the `AntiXssEncoder` class:

- `HtmlEncode` (leveraged by the <%: %> syntax), `HtmlFormUrlEncode`, and `HtmlAttributeEncode`
- `XmlAttributeEncode` and `XmlEncode`
- `UrlEncode` and `UrlPathEncode`
- `CssEncode` and `JavaScriptStringEncode`

Each of these APIs encode data for different contexts. As described previously, it is very important to make use of the proper function for a given context. Examine the surrounding markup on the page to determine context appropriately and choose the right function or combination of functions. In cases where it's necessary to encode more than once, be aware that order is important. For example:

```
<a href=
"http://contoso.com/app.aspx?var=
        <%:Server.UrlEncode(UntrustedVar)%>">
<%: UntrustedTitle %>
</a>
<script>
var x = "<%=HttpUtility.JavascriptStringEncode(
         UntrustedVariable)%>";
. . .
</script>
```

In this example, only a single query string variable is encoded, using the `UrlEncode` function. `UrlEncode` and `UrlPathEncode` are not appropriate for encoding entire URLs. If you need to encode a full URL, it is necessary to, at minimum, validate its URL scheme to avoid allowing URLs with the `JavaScript:` or `VBScript:` URL schemes. To do this, construct a new `Uri` object and then validate the URL scheme as acceptable (http://is.gd/Z8Ozve).

That's really all there is to it. While there are other XSS-related security techniques to evaluate when building your Web technology (such as sandboxing, HTML sanitization, and the like), you will find that proper encoding is what's necessary to prevent the most prevalent XSS bugs.

## Conclusion

While XSS remains a pervasive Web threat, a good understanding of Web encoding techniques and their supporting APIs enables you to secure your Web applications. While all modern application platforms provide the necessary APIs to enable output encoding, it's up to individual developers to effectively apply the proper functions in the appropriate context.

The AntiXSS Library is available for download at no cost (http://is.gd/Z8Ozve). Special thanks to Levi Broderick and Barry Dorrans for contributing to this article.

*— David Ross (http://blogs.msdn.com/dross) is a Principal Security Software Engineer on the MSRC Engineering team at Microsoft. Prior to joining MSRC Engineering in 2002, Ross spent his formative years on the Internet Explorer Security Team.*

Comment

## From the Vault

# Boosting Performance with Atomic Operations in .NET 4

These simple examples in C# and Visual Basic with .NET Framework 4 let you understand and measure the performance improvements achieved when replacing locks with atomic operations.

— *DDJ*

**By Gaston Hillar**

To update some shared variables, you can replace mutual exclusion with a more efficient atomic operation. This simple change boosts performance and reduces contention. When you want to achieve the best performance for a parallelized algorithm, you can follow James Reinders' 8 Rules for Parallel Programming for Multicore (http://is.gd/rlaqoL). Here I focus my attention on the importance of Rule #5 that suggests the usage of atomic operations instead of locks, whenever possible.

I know it is difficult to write lock-free code. In fact, writing lock-free code is error prone. The key message you should get from this simple example is that locks aren't always the best option when you want to maximize both performance and scalability with multicore hardware. These simple examples in C# and Visual Basic with .NET Framework 4 let you understand and measure the performance improvements achieved when replacing locks with atomic operations. You need Visual Studio 2010 in order to run the examples on your multicore hardware and test the results by yourself. However, you have to understand that an atomic operation replaces a lock and an increment operation in this example, but it doesn't mean that you should replace all your locks with

atomic operations. You have to analyze each scenario to determine whether an atomic operation is the best alternative.

The next code snippet doesn't represent a best practice. There are simpler and more efficient ways of coding this application. For example, you can use PLINQ to achieve the same goal. However, this example is focused on the comparison of locks and atomic operations. The following C# code snippet is a console application that generates 150 million words with a serial execution and then it runs a parallelized loop to count the number of words that contain the "a" and the "e" characters, in two independent counters.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// Added
using System.Diagnostics;
using System.Threading;
using System.Threading.Tasks;

namespace CSharpTest
{
    class Program
    {
        // 150 million words
        private const int NUM_WORDS = 150000000;
        private static int _counterA;
        private static int _counterE;
        private static object _syncA = new Object();
        private static object _syncE = new Object();
        private static List<string> _wordsList;

        private static void GenerateData()
        {
            string[] words = {
                "visual",
                "studio",
                "2010",
                "adds",
                "parallel",
                "extensions",
                "but",
                "provides",
                "backwards",
                "compatibility",
                "with",
                "the",
                "previous",
                "threading",
                "model" };

            _wordsList = new List<string>(NUM_WORDS);
            for (int i = 0; i < NUM_WORDS; i++)
            {
                _wordsList.Add(words[i % words.Length]);
            }
        }
        static void Main(string[] args)
        {
            // Generate data (serial execution)
            GenerateData();
            // Measure the parallel execution time
            var sw = Stopwatch.StartNew();

            Parallel.For(0, NUM_WORDS, (i) =>
                {
                    // Locks version
                    if (_wordsList[i].Contains("a"))
                    {
                        lock (_syncA)
                        {
                            _counterA++;
                        }
                    }
                    if (_wordsList[i].Contains("e"))
                    {
                        lock (_syncE)
                        {
                            _counterE++;
                        }
                    }
                });
```

```
            Console.WriteLine("A Counter value: {0}", _counterA);
            Console.WriteLine("E Counter value: {0}", _counterE);
            Console.WriteLine(sw.Elapsed.TotalMilliseconds);
            Console.ReadLine();
        }
    }
}
```

The `Main` method calls the `GenerateData` method. This method is very simple and it just adds 150 million words (`NUM_WORDS`) to the `List _wordsList`. The method generates the same list of words each time it runs in order to make it simple to compare execution times for the same algorithm with the same input data. The code runs with a serial execution. It is possible to parallelize the code that generates the list of words but the main focus isn't on this part of the code.

Then, the `Main` method starts a `Stopwatch` to measure the execution time for the section of the code that is going to run with a parallelized execution. A parallelized `for` loop analyzes each string in `_wordsList` using a `Parallel.For` method. `Parallel.For` launches the necessary number of tasks in parallel to take advantage of the available logical cores. Each parallelized iteration runs a very simple piece of code that checks whether one of the strings in `_wordsList` contains the "a" and the "e" characters.

If the string contains "a," the code uses the `lock` keyword to acquire a mutual-exclusion lock on the `_syncA` synchronization object, increments `_counterA` by 1 and then releases the lock. This increment operation is the only line of code that runs while `_syncA` is locked, as in the following code snippet:

```
if (_wordsList[i].Contains("a"))
{
    lock (_syncA)
    {
        _counterA++;
    }
}
```

If the string contains "e," the code uses the `lock` keyword to acquire a mutual-exclusion lock on the `_syncE` synchronization object, increments `_counterE` by 1, then releases the lock, as in the following code snippet:

```
if (_wordsList[i].Contains("e"))
{
    lock (_syncE)
    {
        _counterE++;
    }
}
```

As happened when the string contained an "a," this increment operation is the only line of code that runs while `_syncE` is locked. Some words can have both an "a" and an "e," and therefore, the code doesn't use an `else` statement and evaluates both characters with an independent `if` statement.

Again, you can use PLINQ to query `_wordsList` and you will avoid these locks. However, I want to focus on the locks and the increment operations. Locks are expensive, and you have a scenario with too many locks. `Parallel.For` runs the iterations in parallel and you will have many tasks acquiring and releasing a mutual-exclusive lock on `_counterA` and `_counterE`. Sometimes, tasks will block until they can acquire the mutual-exclusive lock because there is one task that acquired the lock and is running the increment operation. The code that

runs with a serialized execution is really simple, because it just increments the value of the shared variable but the necessary code to acquire and release the lock decreases the overall performance for this simple algorithm and reduces scalability. The code with the locks guarantees correctness. The following are the final values for the counters displayed on the console:

```
A Counter value: 60000000
E Counter value: 70000000
```

I ran this example many times in a computer with a quad-core microprocessor without turbo modes and with a fixed clock speed. The parallelized `for` loop required an average time of 34,000 milliseconds to run. Windows Task manager shows all the cores with a 100% CPU usage while the parallelized loop runs. However, a sustained high value for the CPU usage doesn't mean that the computer is running efficient parallelized code. If you can achieve the same results in less time and the code still guarantees correctness, you will have an improved algorithm. In a previous post, I explained some techniques to avoid problems when measuring speedup on microprocessors with Turbo Boost. Be sure to disable Turbo Boost when you run the examples explained in this post to avoid drawing wrong conclusions.

The following C# code snippet shows a new version of the `Main` method that provides the same results but it uses atomic operations instead of locks to update the shared counters.

```
static void Main(string[] args)
{
    // Generate data (serial execution)
    GenerateData();

    // Measure the parallel execution time
    var sw = Stopwatch.StartNew();

    Parallel.For(0, NUM_WORDS, (i) =>
        {
            // Atomic operations version
            if (_wordsList[i].Contains("a"))
                Interlocked.Increment(ref _counterA);
            if (_wordsList[i].Contains("e"))
                Interlocked.Increment(ref _counterE);
        });
```

In this new version, if the string contains "a," the code calls the `Interlocked.Increment` method and passes `_counterA` by reference. `System.Threading.Interlocked.Increment` increments the value for the variable as an atomic operation. This method is thread-safe, and therefore, it is also task-safe. Remember that tasks are assigned to threads. This increment operation was the only line of code that ran while the synchronization object was locked and now there is no need to acquire a mutual-exclusion lock. The following code snippet shows the code that increments `_counterA` as an atomic operation:

```
Interlocked.Increment(ref _counterA);
```

If the string contains "e," the code calls the `Interlocked.Increment` method and passes `_counterE` by reference, as in the next line:

```
Interlocked.Increment(ref _counterE);
```

There is no need to acquire mutual-exclusion locks because the increment operation was the only line that needed to run with a serialized execution. The code with the atomic operations also guarantees correctness. The following are the final values for the counters displayed on the console:

```
A Counter value: 60000000
E Counter value: 70000000
```

I ran this example many times on a computer with a quad-core microprocessor without turbo modes and with a fixed clock speed. The parallelized `for` loop required an average time of 10,000 milliseconds to run. Windows Task manager shows all the cores with a 100% CPU usage while the parallelized loop runs. However, this new version requires less time to run than the version that uses locks. The speedup achieved is 34,000/10,000 = 3.40x over the version that used locks. The `System.Threading.Interlocked` class provides methods that perform atomic operations on shared variables. Remember that it is possible to replace a lock and an update on a shared variable with an atomic operation when the update operation is the only statement that runs in the critical section. However, don't replace locks and update statements with atomic operations if you aren't sure of what you're doing.

The following Visual Basic code snippet is the equivalent for the previously explained console application that used locks written in C#. In Visual Basic, instead of the C# lock keyword, the code uses the `SyncLock` equivalent to acquire the mutual-exclusion lock. I ran this example many times in a computer with a quad-core microprocessor without turbo modes and with a fixed clock speed. The parallelized `for` loop required an average time of 35,000 milliseconds to run. The average time required to run the application is almost the same than its C# equivalent.

```
Imports System.Diagnostics
Imports System.Threading
Imports System.Threading.Tasks

Module Module1

    ' 150 million words
    Const NUM_WORDS As Integer = 150000000
    Private _counterA As Integer = 0
    Private _counterE As Integer = 0
    Private _syncA As Object = New Object()
    Private _syncE As Object = New Object()
    Private _wordsList As List(Of String)

    Private Sub GenerateData()
        Dim words As String() = New String() {
            "visual",
            "studio",
            "2010",
            "adds",
            "parallel",
            "extensions",
            "but",
            "provides",
            "backwards",
            "compatibility",
            "with",
            "the",
            "previous",
            "threading",
            "model"}
        _wordsList = New List(Of String)(NUM_WORDS)
        For i As Integer = 0 To (NUM_WORDS - 1)
            _wordsList.Add(words(i Mod words.Length))
        Next
    End Sub

    Sub Main()
        ' Generate data (serial execution)
        GenerateData()

        ' Measure the parallel execution time
        Dim sw = Stopwatch.StartNew()

        Parallel.For(
```

```
            0,
            NUM_WORDS,
            Sub(i)
                ' Locks version
                If (_wordsList(i).Contains("a")) Then
                    SyncLock (_syncA)
                        _counterA += 1
                    End SyncLock
                End If
                If (_wordsList(i).Contains("e")) Then
                    SyncLock (_syncE)
                        _counterE += 1
                    End SyncLock
                End If
            End Sub)

        Console.WriteLine(
            "A Counter value: {0}",
            _counterA)
        Console.WriteLine(
            "E Counter value: {0}",
            _counterE)
        Console.WriteLine(
            sw.Elapsed.TotalMilliseconds)
        Console.ReadLine()

    End Sub
End Module
```

The following Visual Basic code snippet shows a new version of the `Main` procedure that provides the same results but it uses atomic operations instead of locks to update the shared counters.

```
Sub Main()
    ' Generate data (serial execution)
    GenerateData()

    ' Measure the parallel execution time
    Dim sw = Stopwatch.StartNew()

    Parallel.For(
        0,
        NUM_WORDS,
        Sub(i)
            ' Atomic operations version
            If (_wordsList(i).Contains("a")) Then
                Interlocked.Increment(_counterA)
            End If
            If (_wordsList(i).Contains("e")) Then
                 Interlocked.Increment(_counterE)
            End If
        End Sub)

    Console.WriteLine(
        "A Counter value: {0}",
```

```
            _counterA)
        Console.WriteLine(
            "E Counter value: {0}",
            _counterE)
        Console.WriteLine(
            sw.Elapsed.TotalMilliseconds)
        Console.ReadLine()

    End Sub
End Module
```

I ran this example many times in a computer with a quad-core microprocessor without turbo modes and with a fixed clock speed. The parallelized `for` loop required an average time of 10,400 milliseconds to run. Windows Task manager shows all the cores with a 100% CPU usage while the parallelized loop runs. However, as happened with the C# counterpart, this new version requires less time to run than the Visual Basic version that uses locks. The speedup achieved is 35,000/10,400 = 3.37x over the version that used locks. The speedup achieved is almost the same than the speedup for the C# version.

This simple example demonstrates the importance of considering atomic operations when you want to achieve the best performance for a parallelized algorithm. You can use profiling tools to check the differences between the lock version and the atomic operations counterpart. `System.Threading.Interlocked` is an old .NET Framework member and you should take it into account in your parallelized code with .NET Framework 4.

*— Gaston Hillar is a frequent contributor to* **Dr. Dobb's** *on the topic of Microsoft development.*

Comment

# This Month on DrDobbs.com

Items of special interest posted on www.drdobbs.com over the past month that you may have missed

## IMPROVING TESTS FOR RELIABILITY

Netflix's CEO on achieving software reliability by use of quality gates that ensure correct operation.

**http://www.drdobbs.com/240146939**

## THE GROOVY CONUNDRUM

Groovy is one of the most interesting JVM languages, but its long-time performance issues kept it confined to narrow niches. However, a series of important upgrades look like they might push the language into the mainstream. There's the conundrum.

**http://www.drdobbs.com/240147731**

## IMPLEMENTING HALF FLOATS IN D

D offers a set of features, the confluence of which enables the creation of user-defined types that work well enough to take the pressure off of adding more built-in types.

**http://www.drdobbs.com/240146674**

## A LIGHTWEIGHT LOGGER FOR C++

Any software can be developed in a way that allows the running code to provide a trace of its execution in a log file. This article walks you through building a lightweight, thread-safe logger that can be redirected to different I/O channels.

**http://www.drdobbs.com/240147505**

## TESTING COMPLEX C++ SYSTEMS

Some systems are so complex that they require non-traditional approaches to thoroughly test large code bases.

**http://www.drdobbs.com/240147275**

## ONCE AGAIN: JAVA VULNERABILITY

A recent set of vulnerabilities has been found within the Java SE 7 browser plugin. Perception often beats reality, and Java is getting a big black eye from this one.

**http://www.drdobbs.com/240146438**

## IN THIS ISSUE



Dr. Dobb's

**Andrew Binstock** Editor in Chief, Dr. Dobb's
andrew.binstock@ubm.com

**Deirdre Blake** Managing Editor, Dr. Dobb's
deirdre.blake@ubm.com

**Amy Stephens** Copyeditor, Dr. Dobb's
amy.stephens@ubm.com

**Sean Coady** Webmaster, Dr. Dobb's
sean.coady@ubm.com

**Jon Erickson** Editor in Chief Emeritus, Dr. Dobb's

**CONTRIBUTING EDITORS**
**Scott Ambler**
**Mike Riley**
**Herb Sutter**

**DR DOBB'S EDITORIAL**
751 Laurel Street #614
San Carlos, CA
94070
USA

**UBM TECH**
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

**INFORMATIONWEEK**
**Rob Preston** VP and Editor In Chief, InformationWeek
rob.preston@ubm.com 516-562-5692

**John Foley** Editor, InformationWeek
john.foley@ubm.com 516-562-7189

**Chris Murphy** Editor, InformationWeek
chris.murphy@ubm.com 414-906-5331

**Alexander Wolfe** Editor In Chief, InformationWeek.com
alexander.wolfe@ubm.com 516-562-7821

**Stacey Peterson** Executive Editor, Quality, InformationWeek stacey.peterson@ubm.com 516-562-5933

**Lorna Garey** Executive Editor, Analytics, InformationWeek
lorna.garey@ubm.com 978-694-1681

**Stephanie Stahl** Executive Editor, InformationWeek
stephanie.stahl@ubm.com 703-266-6030

**Fritz Nelson** VP and Editorial Director
fritz.nelson@ubm.com 949-223-3608

**David Berlind** Chief Content Officer, UBM Tech
david.berlind@ubm.com 978-462-5315

**ART/DESIGN**
**Mary Ellen Forte** Senior Art Director
maryellen.forte@ubm.com

**Sek Leung** Senior Designer
sek.leung@ubm.com

**INFORMATIONWEEK.COM**
**Benjamin Tomkins** Managing Editor
benjamin.tomkins@ubm.com 516-562-5336

**Roma Nowak** Senior Director, Online Operations and Production
roma.nowak@ubm.com 516-562-5274

**Tom LaSusa** Managing Editor, Newsletters
tom.lasusa@ubm.com

**Jeanette Hafke** Web Production Manager
jeanette.hafke@ubm.com

**Joy Culbertson** Web Producer
joy.culbertson@ubm.com

**Nevin Berger** Senior Director, User Experience
nevin.berger@ubm.com

**Atif Malik** Director, Web Development
atif.malik@ubm.com

**INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK**
**EVP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz**
(212) 600-3015, martha.schwartz@ubm.com

**Sales Assistant, Salvatore Silletti**
(212) 600-3327, salvatore.silletti@ubm.com

**SALES CONTACTS—WEST**
Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

**Sales Director, Michele Hurabiell**
(415) 378-3540, michele.hurabiell@ubm.com

Strategic Accounts

**Account Director, Sandra Kupiec**
(415) 947-6922, sandra.kupiec@ubm.com

**Account Manager, Vesna Beso**
(415) 947-6104, vesna.beso@ubm.com

**Account Executive, Matthew Cohen-Meyer**
(415) 947-6214, matthew.meyer@ubm.com

**MARKETING**
**VP, Marketing, Winnie Ng-Schuchman**
(631) 406-6507, winnie.ng@ubm.com

**Marketing Director, Angela Lee-Moll**
(516) 562-5803, angele.leemoll@ubm.com
**Marketing Manager, Monique Kakegawa**
(949) 223-3609, monique.luttrell@ubm.com

**Program Manager, Diane Scala**
516-562-5476, diane.scala@ubm.com

**SALES CONTACTS—EAST**
Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

**District Manager, Steven Sorhaindo**
(212) 600-3092, steven.sorhaindo@ubm.com

Strategic Accounts

**District Manager, Mary Hyland**
(516) 562-5120, mary.hyland@ubm.com

**Account Manager, Tara Bradeen**
(212) 600-3387, tara.bradeen@ubm.com

**Account Manager, Jennifer Gambino**
(516) 562-5651, jennifer.gambino@ubm.com

**Account Manager, Elyse Cowen**
(212) 600-3051, elyse.cowen@ubm.com

**Sales Assistant, Kathleen Jurina**
(212) 600-3170, kathleen.jurina@ubm.com

**AUDIENCE DEVELOPMENT**
**Director, Karen McAleer**
(516) 562-7833, karen.mcaleer@ubm.com

**BUSINESS OFFICE**
**General Manager, Marian Dujmovits**
**United Business Media LLC**
600 Community Drive
Manhasset, N.Y. 11030
(516) 562-5000




**UBM TECH**
**Paul Miller, CEO**
**Kathy Astromoff, CEO, Electronics**
**Robert Faletra, CEO, Channel**
**Edward Grossman, President, Business Technology Media**
**Marco Pardi, President, Business Technology Events**
**David Berlind, Chief Content Officer**
**John Dennehy, Chief Financial Officer**
**David Michael, Chief Information Officer**
**Martha Schwartz, Chief Sales Officer, Business Technology Media**
**Scott Vaughan, Chief Marketing Officer**
**Simon Carless, EVP, Game & App Development and Black Hat**
**Lenny Heymann, EVP, New Markets**
**Angela Scalpello, SVP, People & Culture**

**UNITED BUSINESS MEDIA LLC**
**Pat Nohilly** Sr. VP, Strategic Development and Business Administration
**Marie Myers** Sr. VP, Manufacturing

**INFORMATIONWEEK VIDEO**
informationweek.com/tv
**Fritz Nelson** Executive Producer
fritz.nelson@ubm.com

**INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK**

**DarkReading.com**
Security
**Tim Wilson**, Site Editor
tim.wilson@ubm.com
**IntelligentEnterprise.com**
App Architecture
**Doug Henschen**, Editor in Chief
doug.henschen@ubm.com
**NetworkComputing.com**
Networking , Communications, and Storage
**Andrew Conry-Murray**, Editor
andrew.murray@ubm.com
**PlugIntoTheCloud.com**
Cloud Computing
**John Foley**, Site Editor
john.foley@ubm.com
**InformationWeek SMB**
Technology for Small and Midsize Business
**Benjamin Tomkins**, Site Editor
benjamin.tomkins@ubm.com
**Byte.com**
**Larry Seltzer**
Editorial Director
larry.seltzer@ubm.com
**Dr. Dobb's**
Good Stuff for Serious Developers
**Andrew Binstock**
Editor in Chief
andrew.binstock@ubm.com