DRAFT

Java 7 *Sockets Direct Protocol* – Write Once, Run Everywhere …. and Run (Some Places) *Blazingly*.

Ben Cotton
August 10, 2013

This article is will survey the new Java™ Sockets Direct Protocol (SDP) technology that was recently introduced in the Java 7 SDK. This article assumes that you have at least some familiarity with the Java programming language, especially with regard to the java.net.* API package. The Java 7 Sockets Direct Protocol (SDP) is a very exciting breakthrough for the Java™ platform. It empowers the Ultra High Performance Computing (UHPC) community to use Java's ubiquitously common features and merits for a very uncommon use case: native access to the Infiniband RDMA capability. RDMA stands for Remote Direct Memory Access. We will survey RDMA as a central theme in this article. RDMA is incredibly cool. The UHPC community has the most stringent non-compromising low-latency and high throughput demands imaginable. The UHPC needs the very best RDMA capability availability. The UHPC community needs the Infiniband RDMA capability. With the introduction of the Java 7 Sockets Direct Protocol offering, the UHPC community is now provided a Java platform that empowers them to write Java application code that directly joins to the full power of the native Infiniband RDMA capability. Wow, right?

Before we dive deeply into the new Java™ Sockets Direct Protocol , let's briefly review Java's networking and sockets API history.

In 1995 Sun Microsystems introduced the world to Java™ and immediately began trumpeting the platform with the universally recognizable catch phrase 'Java – Write Once, Run Everywhere'. As we all know, the idea behind this was simple: instead of writing applications in C++ code (which was devilishly difficult to build/deploy with anything even close to a "run everywhere" portability confidence) you could now write application code in something called Java™ code which would build/deploy to a virtual machine (not the underlying OS execution environment). This greatly liberated the Java application programmer from having to worry about anything re: portability. The Java virtual machine would now have sole custody of the guaranteeing portability obligation. The JVM made this promise: If you could build/deploy Java code that worked on *one* Java VM (for some *specific* underlying OS), the platform guaranteed that the exact same code would work on *any* OS (for which a compliant Java VM was available). In other words, the programmer was immediately and fully unburdened from the obligation to write portable code. No more conditional compilation and pre-processor macros were needed. Anybody remember C++ and #ifdef hell? The JVM would now relieve application programmers of that cruel burden.

This was all extremely helpful and very well received by the applications programming community. As we all know, Java caught on quickly and intensely – globally embraced at a pace unmatched in the history of Computing's many, many programming language platforms.

In the beginning, Sun offered the Java VM to run on exactly 3 operating systems: (i) Solaris (ii) Linux (iii) Windows. Because Microsoft had just a few years earlier (1993) delivered the WinSOCK protocol stack with Windows, Windows could now do TCP/IP networking (and through an API fully supported by Microsoft). The various *nix systems (of course) had been doing TCP/IP since the 1970s. Microsoft's introduction of WinSOCK was absolutely essential to Java becoming what it became. Without WinSOCK you could not deliver a Windows VM that supported the java.net.* and the java.io.* APIs. Without it, Java could not build a complete network capable VM that would run on the OS that monopolized the world's desktops. Instead of Java™ reaching "maybe a million" desktops, with Windows now doing full TCP/IP, Java™ could now reach "maybe a billion" desktops.

Enough already with this "brief" review of the Java™ history of delivering a completely poprtable Networking and Sockets API to the masses. Enough with emphasizing the importance of Microsoft delivering WinSOCK to being so absolutely crucial to Java exploding onto the world wide web as *the* internet programming language.

Things have changed.

Sure, Java is still "write once, run everywhere". Portability is still a central priority. But, now with Java 7 and *Sockets Direct Protocol* a whole lot more can be done with the Java VM. Portability is not the only priority. Accommodating ultra-high performance use cases is now a very big priority for the Java VM. With *Sockets Direct Protocol* the Java VM can now deliver the same Networking and Sockets APIs that *directly* access the *native* power of **Infiniband**. Infiniband is much faster than Ethernet. Infiniband is the physical network layer provider of choice to the UHP computing community.

We'll talk about exactly what Infiniband is and how the Java 7 VM empowers applications to use native Infiniband capabilities. We'll talk about that real soon.

One interesting thing to note (especially from an historical perspective) re: Java's decision to deliver the Sockets Direct Protocol: the capability is only supported on 2 operating systems. Microsoft is not one of those operating systems. The 2 operating systems on which Java 7 SDP is supported is Solaris and Linux. Solaris SDP support is delivered standard for all versions since (and including) Solaris 10. As long as you have a physical inifiniband NIC, Java 7 SDP will work out-of-the box. For Linux, SDP support is delivered via the Open Fabrics Enterprise Distribution package. To check if your Linux version is configured with the OFED device drivers, and that you indeed have a physical Infiniband NIC adapter, simply type

```
bcotton@LRI04LD044.ny.jpm.com> egrep "^[ \t]+ib" /proc/net/dev
```

If you get any output from this command, you are all set to use Java 7 SDP on this operating system.

It is important to note that all java.net.* and java.io.* application code will – of course – still run on Microsoft Windows using the Java 7 VM … but it will run without Sockets Direct Protocol (and thus it will run with physical layer provider=Ethernet). This will be true even if you are running on a Windows Server version that provides Infiniband support (via WinSOCK Direct). Again everything will still run on Microsoft, it will just not run as fast as "some places" that are not Microsoft (i.e. *nix).

Things have, indeed, changed.

Let's now talk about Java™'s API bridge to the operating system's Network protocol stack. First, the OSI model of Network layers is as follows.

#	Layer	Protocol	Java SDK core APIs
7.	Application Layer	HTTP,FTP,SSL,etc.	`java.net.HttpURLConnection , javax.servlet.HttpServle`
6.	Presentation Layer		#no real distinction in Java between Application/Presentation OSI layers
5.	Session Layer	NetBios, RCP	#no Java SDK core support for OSI session layer
4.	Transport Layer	TCP, UDP	`java.net.Socket, java.net.ServerSocket, java.net.Datagram`
3.	Network Layer	IP	`Java.net.InetAddress`
2.	Data Link Layer	PPP	#no Java SDK core support for OSI data link layer
1.	Physical Layer	Ethernet, Infiniband	#no Java SDK core support for OSI physical layer However … ***Java 7 Sockets Direct Protocol*** (VM bridge from Infiniband to java.net.* and java.io.* core APIs)

For the OSI Network Layers view, the Java 7 Socket Direct Protocol capability takes Java application code as "close to the metal" as is possible. Java SDP provides a *direct* (the 'D' in SDP) bridge from the application code, through the VM, to *native*, *physical* Infiniband. The Java 7 SDP capability does this without the application having to change its use of the core java.net.* and java.io.* APIs. More than just that, by configuring the Java VM's specific join-point to the Infiniband OS device driver and libraries (aka Infiniband's VERBs layer API) the application code's use of java.net.* and java.io.* - which is Java's API for Transport Layer OS resources (OSI layer 4) can by-pass the traditional network protocol stack (i.e. it can by-pass OSI layer 3 and bypass OSI layer 2) and go *directly* to Infiniband (OSI layer 1). The performance implications and payoffs are very significant.

With Java 7 and Sockets Direct Protocol, Java now does RDMA (Remote Direct Memory Access)

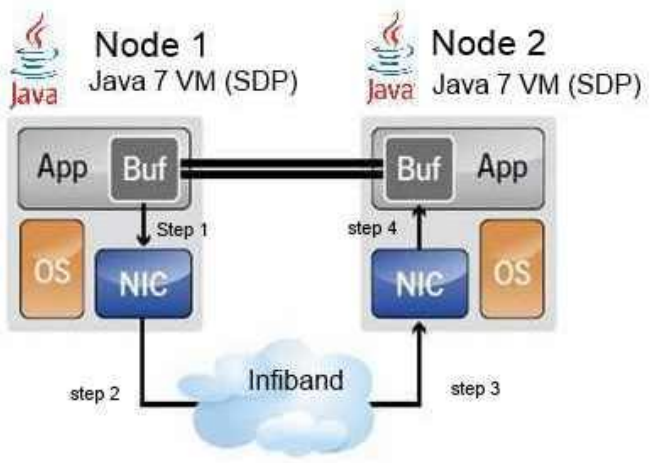
RDMA is Remote Dynamic Memory Access -- which is a way of moving application buffers between two Java VM processes' (executing in *nix user address space) across a network. RDMA differs from traditional network interfaces because it bypasses the operating system. This allows Java SDP over RDMA to deliver: (i) The absolute lowest latency (ii) The highest throughput (iii) Smallest CPU footprint.

DRAFT

By empowering a Java join point to RDMA, SDP implicitly also empowers Java to realize a very compelling "**Zero-copy**" capability. "**Zero-copy**" describes computer operations in which the CPU does not perform the task of copying data from one memory area to another. Zero-copy versions of network protocol stacks greatly increase the performance of certain application programs and more efficiently utilize system resources. Performance is enhanced by allowing the CPU to move on to other tasks while data copies proceed in parallel in another part of the machine. Also, zero-copy operations reduce the number of time-consuming mode switches between user space and kernel space. System resources are utilized more efficiently since using a sophisticated CPU to perform extensive copy operations, which is a relatively simple task, is wasteful if other simpler system components can do the copying. It is important to note that the Zero-copy capability we're talking about here is not the Zero-Copy capability you can achieve by using java.nio.channels.FileChannel's transferTo() API. It is much, much higher performing. With Java 7 SDP, you directly use the native Infiniband Zero-copy protocol implementation.

Let's start to visually depict exactly what Sockets Direct Protocol capability looks like within the context of some typical Java deployment views.

The following diagram pictures how Node 1 (a `java.net.Socket` *writer*) and Node 2 (a `java.net.ServerSocket` *listener*) can be deployed onto a Java 7 VM configured and booted to support SDP in such a way that the 2 JVMs can exchange application data buffers from one VM to the other, across an Infiniband network, without any OS system ca lls or services being invoked. The Java data transfer completely by-passes both Operating Systems. Wow, right?



1. Java 7 application=**Node 1** (JVM booted to use SDP) uses the java.net.Socket API to *write* a block of application data across the network to a java.net.ServerSocket *listener*.

2. Because the JVM was booted to use SDP the Operating System TCP/IP stack is completely bypassed – the application data is written directly to Infiniband's RDMA capability (requires Infiniband to be the physical provider of Network Interface Card).

3. Java 7 application=**Node 2** (JVM also booted to use SDP) uses the `java.net.ServerSocket` API to listen for a block of application data to arrive via RDMA across the network from a `java.net.Socket` writer. (requires Infiniband to be the physical provider of Network Interface Card).

4. Data delivered *directly* to the Java 7 VM application buffer! No OS system or service calls involved – neither from Node 1's OS nor Node 2's OS. *That* is the power of Java 7 Sockets Direct Protocol.

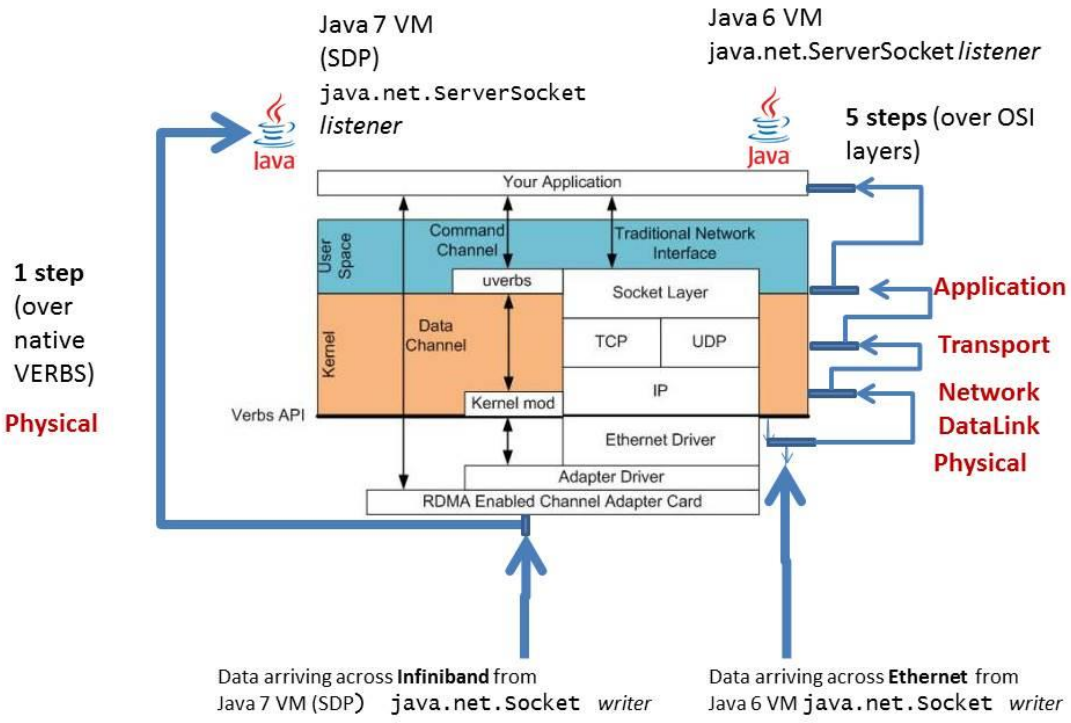
DRAFT

What is the logical performance difference between the same application running on Java 7 with SDP vs. Java 6?

The following "deep-dive" diagram details Node 2's view (from the diagram on the previos page) in two different scenarios:

1. Using Java 7 with SDP configured (shown below left) How does Node 2's reception of Node 1's transmitted data travel up the OSI Network layer protocol stack and into the Java application? How many steps does it take? It take sonly one step! (Take a look below – this is great news for UHPC Java apps . UHPC community can use Java 7 to do what needs to be done).

2. Using Java 6 (no SDP - shown below right) How does Node 2's reception of Node 1's transmitted data travel up the OSI Network layer protocol stack and into the Java application? How many steps does it take? It takes five steps (Take a look below – this is the familiar TCP/IP protocol stack – not SDP. It works for most, but does not work for the UHPC community. UHPC community just can't use Java 6 to do what needs to be done.).



DRAFT

How do you administer and configure a Java 7 VM to use Sockets Direct Protocol?

The following is taken from the configuration section of the Oracle tutorial page that introduces Java 7 SDP (see http://docs.oracle.com/javase/tutorial/sdp/sockets/index.html)

An SDP configuration file is a text file that the Java VM reads from the local file system at boot time. The SDP configuration file has exactly 2 different types of entries. Each type of entry is expressed exactly once per line:

1. An SDP configuration **comment line**
2. An SDP configuration **rule** line

A comment is indicated by the hash character (#) at the beginning of the line, and everything following the hash character will be ignored.

For configuration **rule** lines, there are exactly two types of rules:

1. BIND rules
2. CONNECT rules

A "bind" rule indicates that the SDP protocol transport should be used whenever a TCP socket binds to an address and port that match the rule. A "connect" rule indicates that the SDP protocol transport should be used when an unbound TCP socket attempts to connect to an address and port that match the rule.

Through the rules specified in the SDP configuration file, the Java VM can know exactly when to replace the normal TCP/IP protocol stack with Infiniband's VERBS/RDMA protocol stack.

The first keyword indicates whether the rule is a **bind** or a **connect** rule. The next token specifies either a host name or a literal IP address. When you specify a literal IP address, you can also specify a prefix, which indicates an IP address range. The third and final token is a port number or a range of port numbers.

Consider the following notation in this sample configuration file:

```
# Use SDP when binding to 192.0.2.1
bind 192.0.2.1 *

# Use SDP when connecting to all application services on 192.0.2.*
connect 192.0.2.0/24     1024-*
```

The first rule in the sample file specifies that SDP is used for any port (*) on the local IP address 192.0.2.1. You would add a bind rule for each local address assigned to an InfiniBand adaptor. (An *InfiniBand adaptor* is the equivalent of a network interface card (NIC) for InfiniBand.) If you had several IB adaptors, you would use a bind rule for each address that is assigned to those adaptors.

The second rule in the sample file specifies that whenever connecting to 192.0.2.* and the target port is 1024 or greater, SDP is used. The prefix on the IP address /24 indicates that the first 24 bits of the 32-bit IP address should match the specified address. Each portion of the IP address uses 8 bits, so 24 bits indicates that the IP address should match 192.0.2and the final byte can be any value. The -* notation on the port token specifies "and above." A range of ports, such as 1024—2056, would also be valid and would include the end points of the specified range.

DRAFT

How do you boot a Java 7 VM to use Sockets Direct Protocol?

```
bcotton@LRI04LD044.ny.jpm.com> java \
                                -Dcom.sun.sdp.conf=sdp.conf \
                                -Djava.net.preferIPv4Stack=true  \
                                Application.class
```

Note the use of the IPv4Stack as the network format to be booted. Even though both Java 7 and Infiniband use the more modern IPv6 network format, the mapping between the 2 on both Solaris and Linux is not supported. Thus, always use the cornerstone familiar (and decades reliable) IPv4 network format when booting a Java 7 VM that will use SDP.

How much performance *improvement* should be expected when running apps on a Java 7 VM booted with SDP support?

That of course is the ultimate question! What *exactly* do I gain from using Java 7 SDP? The answer to this question, of course, cannot be determined from the context of this article. Performance gains will vary depending on many numerous factors. As this article comes to a close, do know that the following will always be true:

Infiniband is considerably faster that Ethernet. The typical ISP delivers an Ethernet solution that is less than 1Gb/sec. Often much less. Infiniband can deliver (btw, at a steep price) speeds as fast 40Gb/sec. That's more than an order of magnitude faster than a typical Ethernet solution.

Also, Java 7 SDP uses RDMA and the very best Zero-Copy implementation. The data transfers 100% by-pass the OS network TCP/IP stack and all context switching that comes when transporting data between system calls in kernel address-space and application code buffers in user address-space.

All this to be provided with 100% Java SDK API transparency. Not 1 single line of Java application code's use of java.net.* or java.io.* needs to be changed.

In closing, though things have changed a lot, the core Java spirit remains the same. Today, just like in Java's earliest days (when the JVM took on the burden to isolate the application code from hellish portability complications) *the JVM once again takes on the entire burden* of delivering a priority capability: this time it is Sockets Direct Protocol capability. Indeed, the original Java slogan can remain almost exactly familiar (with a tiny new phrase to reflect these exciting modern times):

Java 7 SDP – write once, run everywhere and run (some places) blazingly!

Thanks,
Ben

References:

Java 7 SDP http://docs.oracle.com/javase/tutorial/sdp/sockets/

INFINIBAND http://www.infinibandta.org/

RDMA http://en.wikipedia.org/wiki/Remote_Direct_Memory_Access

DRAFT