Architecture and Implementation of Sockets Direct Protocol in Windows

Dror Goldenberg Tzachi Dar Gilad Shainer
Mellanox Technologies Inc.
{gdror , tzachid} at mellanox.co.il; shainer at mellanox.com

Abstract

Sockets Direct Protocol (SDP) is a byte stream protocol that utilizes the capabilities of the InfiniBand fabric to transparently achieve performance gains for existing socket-based networked applications. We implemented SDP stack for the Windows operating system that is fully interoperable with Linux. The paper describes the early experience with the implementation of the protocol stack. We go through the motivation, the main implementation architectural aspects and challenges. We present preliminary performance results. Running over 20Gb/s InfiniBand double data rate (DDR) links we observed bandwidth record of 1316MB/s.

1. Introduction

As network speeds and I/O demand increase, it becomes a challenge to leverage innovative networking technology in a transparent way, preserving the existing application base. Sockets Direct Protocol (SDP) answers this need, enabling socket based applications to transparently utilize the RDMA and transport offload capabilities of the InfiniBand fabric. Under the hood, SDP supports Buffer Copy (BCopy) and Zero Copy (ZCopy) operation modes. Both modes utilize the InfiniBand transport offload capability. ZCopy further utilizes RDMA to transfer data directly between application buffers.

Previous SDP research [6][7][9][19] mainly focused on development of SDP for the Linux operating system, performance analysis, tuning and addition of Zero Copy path to the SDP implementations. The SDP implementation for Linux has become part of IBG2 [3], the Mellanox InfiniBand software stack distribution for Linux, and of the Linux InfiniBand open source development distribution – OpenFabrics [2].

There is a growing interest in InfiniBand support for the Windows operating systems. Microsoft recently announced the forthcoming release of Windows Compute Cluster Server 2003 [5] which supports clustered applications running over InfiniBand and includes a certification program for the InfiniBand drivers.

We implemented SDP for the Windows operating system. The implementation includes basic support for a selected set of Winsock2 functions, and it is fully interoperable with the Linux SDP implementation. We describe the implementation of the SDP protocol stack, the interface with the operating system and present preliminary performance results.

2. Background

2.1. InfiniBand

InfiniBand[1] is a standard switched fabric that supports high bandwidth (up to 120Gb/s) and low-latency. Processor nodes and I/O devices connect to the fabric through Host Channel Adapters (HCA) and Target Channel Adapters (TCA) respectively.

The InfiniBand architecture is based on Channel I/O and offers a rich feature set including: RDMA, quality of service, multiple virtual lanes, partitioning and congestion control. This feature set is essential for fabric consolidation, management and performance.

An HCA device usually plugs into the I/O subsystem of a processor node. It contains a sophisticated DMA engine, a transport engine and management capabilities. Its key contributions to system performance are the following:

- Transport Offload (segmentation, reassembly, acknowledgement, retransmission, etc.)
- Zero Copy (exposure of local memory buffers to the fabric)
- Kernel Bypass (enables user mode applications to directly access the HCA hardware)

The InfiniBand specification defines an HCA interface called Verbs. Upper layer protocols are implemented on top of Verbs. This interface is asynchronous: a consumer posts Work Requests (Send, Receive, RDMA Write, RDMA Read and Atomic operations) to the HCA. The HCA optionally signals their completion and can schedule a completion

1-4244-0328-6/06/$20.00

notification (through event and interrupt).

2.2. Sockets Direct Protocol (SDP)

Traditional implementation of TCP socket requires two major CPU consuming tasks: (1) data copy between application buffers and NIC buffers and (2) TCP transport handling (segmentation, reassembly, timers, acknowledgments, etc.). In particular, data copying overhead was identified in the 1990s as a significant CPU consumer in TCP stacks [14]. These operations turn more severely into performance bottlenecks as I/O interconnect speed increases. Sockets Direct Protocol (SDP), defined in the InfiniBand Architecture Specification [1], eliminates the protocol stack bottlenecks.

SDP is a byte stream protocol that mimics TCP SOCK_STREAM semantics. Existing socket-based applications can use the SDP protocol without any code modification or recompilation. An implementation of SDP resides between the software socket interface and the InfiniBand Verbs interface.

SDP is implemented on top of InfiniBand Reliable Connection (RC) transport service, leveraging the HCA transport offload. It maps the socket Send and Receive calls to InfiniBand operations. Control messages are transferred by Send messages. Data messages are transferred through one of the two SDP transfer types: BCopy and ZCopy.

BCopy (Figure 1) uses dedicated, pre-registered private SDP buffers. Data is copied from application buffers into SDP private buffers; data transfer is then performed using Send messages. On the receive side, the data lands in SDP private buffers and is then copied into application buffers when they are made available.

ZCopy (Figure 2), transfers data directly between application buffers through RDMA operations. In order to enable RDMA operations, the SDP stack must pin and register the application buffers. In the example (Read ZCopy), the data source gets an application buffer to send. If the buffer is large enough, the data source registers and advertises it by sending a SrcAvail message to the data sink. Later, a buffer is posted on the data sink. The buffer is registered and an RDMA Read is performed, copying the data source buffers into data sink application buffers. Once the RDMA Read is completed, the data sink indicates this by sending an RdmaRdCompl message to the data source.

Similarly, Write ZCopy uses an RDMA Write operation for data transfer. The data sink advertises a buffer by sending a SinkAvail message. The data source then performs RDMA Write of the buffer and indicates completion of the transaction by sending an RdmaWrCompl message.

Both BCopy and ZCopy modes incur some communication overhead. BCopy mode incurs the overhead of a local data copy. ZCopy mode incurs the overhead of locking the application buffers in physical memory, registering them with the HCA, and additional communication overhead associated with buffer advertisement handshake. The SDP implementation uses a ZCopy threshold to determine which path to take: ZCopy or BCopy.

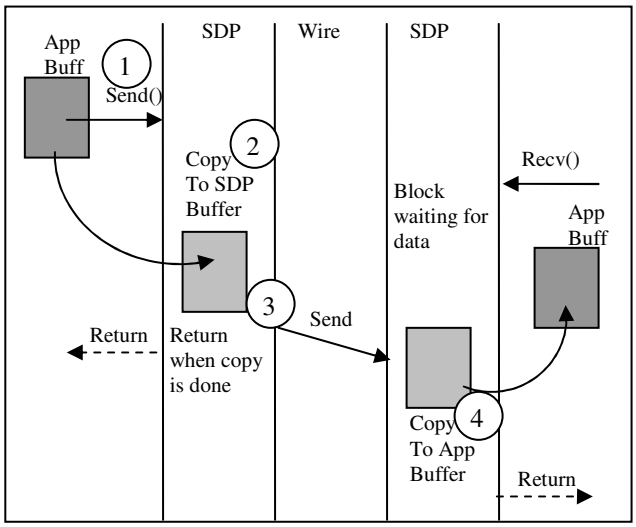

Figure 1: BCopy Flow

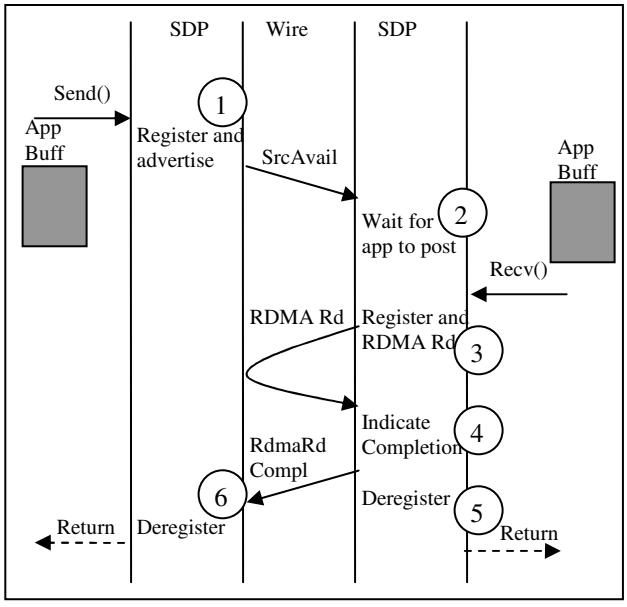

Figure 2: Read ZCopy Flow

2.3. Sockets APIs in Windows

Windows supports the Windows Socket 2

(Winsock) API [17]. It is based on the Berkeley Software Distribution (BSD) and includes multiple enhancements. The API includes a rich set of functions (more than 100) compared with just about a handful of functions in Linux.

On the data path, Winsock supports both synchronous and overlapped operations. Synchronous operations can be blocking or non-blocking. Blocking synchronous calls return after delivering the data to the remote peer or after locally buffering it. Non-blocking synchronous calls return after locally buffering the data. If buffering is not available, the call returns immediately and indicates that the data has not been sent.

Overlapped operations return immediately. The I/O operation may be internally queued for later processing. The data buffers should not be modified by the application until the overlapped operation completes. Multiple notification mechanisms exist to report the completion of an overlapped I/O operation including event objects, polling and callbacks.

Recently, the Open Group has released the Extended Socket API (ES-API) [11], which makes memory registration explicit to the application and enables overlapped mode of operation on both data and control path. We are not aware of any plans to support the new API in Windows.

From SDP implementation perspective, it is required to preserve the exact socket semantics for the various APIs. It is most challenging to add ZCopy to the synchronous calls [6][7]. The Overlapped operations provide semantics suitable for ZCopy, with the registration left to be dealt with by the SDP stack. The ES-API is the most suitable API for implementation of ZCopy SDP as registration is explicit.

2.4. Socket Offload Alternatives in Windows

Multiple alternatives for sockets protocol offload exist in the Windows operating system[1]. Some are TCP/IP wire compatible. Others provide sockets semantics to the application while under the hood they may use different transport and wire protocol.

Note that SDP in this paper refers to Mellanox implementation of SDP over InfiniBand for Windows. Similarly to WSD, Microsoft will offer an option for running SDP stack in Windows Server codename Longhorn over a SAN provider that includes several extensions. The wire protocol is based on [8].

We chose to implement SDP in order to enable full offload and ZCopy in all Windows derivatives including interoperability with Linux. It also enables further tuning and research of the protocol stack.

Table 1: Socket Offload Options in Windows

Offload Option	Wire Protocol	Transport Offload	Buffer Copy	IHV SW Component
Stateless Offloads (Checksum, LSO, GSO, RSS)	TCP/IP	Basic	Required	NDIS Miniport
I/OAT	TCP/IP	Enhanced	DMA assisted	NetDMA NDIS Miniport
TCP Chimney	TCP/IP	Full	Offloaded	Chimney NDIS Miniport
WSD	Microsoft Proprietary	Full	RDMA	SAN Provider
SDP	InfiniBand SDP	Full	RDMA	Full SDP Stack (Socket SPI and SDP Module)

2.5. WinIB

WinIB [4] is the Mellanox InfiniBand software stack distribution for Windows operating systems. WinIB is based on an open source development (OpenFabrics [2]). The software stack is designed, packaged and supported to enable OEMs to meet the High Performance Computing and Enterprise Data Center markets requirements. The package includes all necessary components for achieving full performance advantages of the InfiniBand interconnect:

- InfiniBand HCA Verbs Driver and Access Layer
- InfiniBand Subnet Management
- IPoverIB NDIS Miniport
- SDP
- WinSock Direct SAN Provider
- SCSI RDMA Protocol (SRP) StorPort Miniport

WinIB is supported in Windows Server 2003, Windows Compute Cluster Server 2003 and Windows Server codename Longhorn. Mellanox has added support for Windows XP and Windows XP embedded.

3. SDP Implementation

This section provides a high level technical overview of the SDP protocol stack.

3.1. Interfacing the Windows Operating System

The key challenge in protocol stack implementation is to cleanly interface the OS. Basic support for intercepting all socket calls is essential for an SDP

[1] Not all transport offload options are available in all Windows derivatives. For example WSD is only available on Windows Server 2003, Compute Cluster Server and Windows Server codename Longhorn.

implementation. The Linux implementation [6] for example, implemented a new address family (AF_INET_SDP) in kernel. The main challenge was to dynamically change the application address family into AF_INET_SDP. It was accomplished by using a preloaded library.

Luckily, the Windows operating system has a built in infrastructure for sockets providers. Based on Windows Open System Architecture, it is possible to register a library that implements a Windows Socket Service Provider Interface (SPI) with the socket library (WS2_32.dll). All socket calls map into WSPxxx calls in the provider. In some cases multiple sockets calls map into the same WSPxxx call with appropriate flags. Due to the rich feature set of Winsock, the provider has to support a long list of entry points.

Previous research showed different approaches to interface the operating system. WSDLite[15] over VIPL used Detours to intercept socket function calls. It requires modifying the applications binaries and it only supports x86 architecture. We took an approach similar to [16] and implemented a Winsock Service Provider Interface (SPI). We believe it is a better alternative as it is portable and supported on the various Windows derivatives.

Figure 3 shows the SDP protocol components in the Windows protocol stack. The SDP provider is implemented in user mode. It intercepts all sockets calls and takes two major roles: socket switch and proxy to the SDP module.

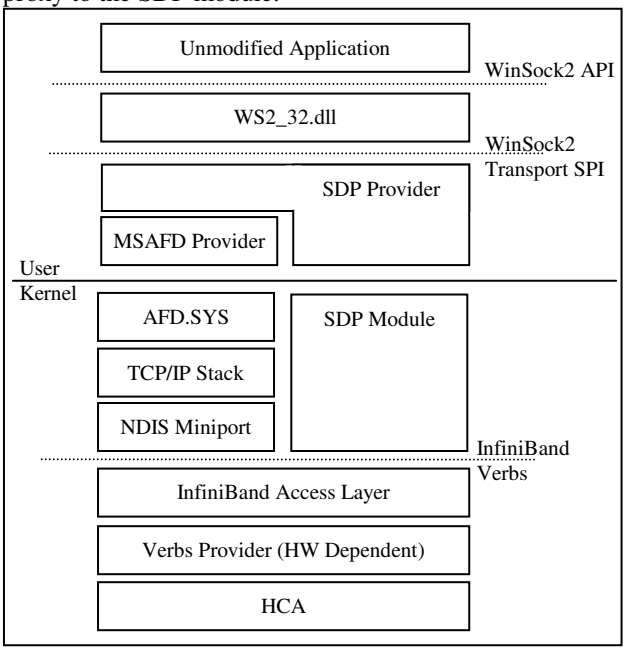

Figure 3: SDP in Windows Protocol Stack

For the connection establishment calls, the provider takes the routing and policy decision and decides whether a TCP or an SDP socket should be created. If a TCP socket is required, then all calls on the socket are redirected to the next Winsock SPI in the chain. If an SDP socket is required, then the calls are redirected to the kernel SDP module. The current implementation selects SDP vs TCP according to an environment variable and the application name.

For operations other than connection establishment the provider mainly forwards the calls down to the kernel SDP module.

3.2. Overlapped vs. Synchronous I/O

Winsock supports two major I/O types: synchronous and overlapped. Synchronous I/O operations take a single call per I/O operation. On the other hand, overlapped I/O requires two calls per operation: one to submit it and one to reap its completion. There is further an extra overhead to prepare the overlapped structure and to destroy.

The overlapped I/O architecture can improve application scalability when many sockets are used concurrently. Instead of creating a thread per socket, a thread per CPU can be used. Also, applications that use large I/Os can perform computation while an I/O is being processed by the networking stack.

For small size or latency-sensitive I/O operations, it may be desirable for certain applications to use synchronous I/Os and thus reduce the overhead incurred with overlapped I/O. Applications that are ported from BSD/POSIX socket API [10] most likely do not use overlapped communication.

In an SDP implementation special attention should be paid to the number of system calls and context switches a socket call is translated into. This is essential in order keep low CPU overhead and to obtain better latency. The current cost of a system call, implemented as an IOCTL, is in the order of 2us. We further look at Fast I/O Dispatch Routines implementation that can reduce this overhead to around 0.7us. The cost of a context switch is an order of magnitude higher.

Alternatively, in order to simplify the SDP design it is possible to implement only overlapped I/Os in the SDP module and convert each synchronous I/O into two calls in the provider library. The first call creates an overlapped structure with an event and submits the I/O. The second call blocks waiting for the I/O to complete.

Implementing BCopy for overlapped I/O in the SDP module is further complicated. The main issue has to do with data copy. As we neither pin the application buffers nor map them to kernel space, we should

therefore perform data copy at the process execution context.

Figure 4 shows an example of overlapped send. In this example there are available SDP private buffers. We copy the application buffer into SDP private buffer and send it. The call then returns. To indicate the overlapped completion, SDP wakes up a helper thread. We use a helper thread per process. This thread uses an up-call to WS2_32.dll to indicate the I/O completion. Similar flow exists for WSARecv when data is already available in SDP private buffers. In this case, we copy the data to user inside the WSARecv call.

We note that the flow can be significantly simplified in an implementation of a synchronous send call, where a helper thread is not required.

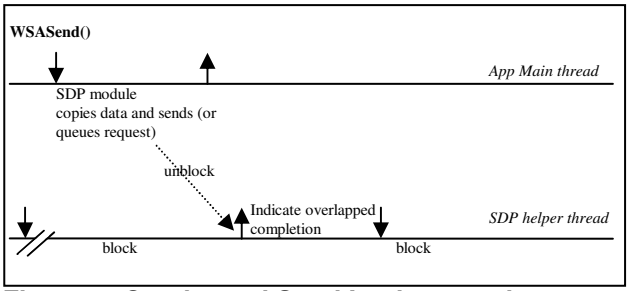

Figure 4: Overlapped Send Implementation

Figure 5 shows an example of overlapped receive where data is not yet available in SDP private buffers when WSARecv is called. Hence we queue the overlapped I/O and return. When data arrives, an interrupt occurs and a Deferred Procedure Call (DPC) is scheduled. In order to copy the data we wake up a helper thread. In turn, the helper thread unblocks, copies the data into the application buffer and indicates the overlapped completion. Similar flow exists on WSASend when there are no available SDP private buffers or when SDP can not send due to flow control (remote buffers are not available). SDP queues the overlapped structure and when SDP buffers are made available, it wakes up the helper thread to copy the data into SDP private buffers. The helper thread then posts send, complete the I/O and blocks again.

Similarly, in an implementation of a synchronous receive call a helper thread is not required.

The current implementation has a single helper thread per process. For scalability reasons the architecture should allow a helper thread per CPU. This provides more CPU resources for data copy.

We started off with an implementation that only supported synchronous I/O. We then moved to support overlapped architecture with the provider converting each non-overlapped I/O into two calls. This change in flow caused a major increase in latency (on synchronous calls) in the order of 15 microseconds. It is related to the extra context switch into the helper thread on the receive path, which is required for copying the data.

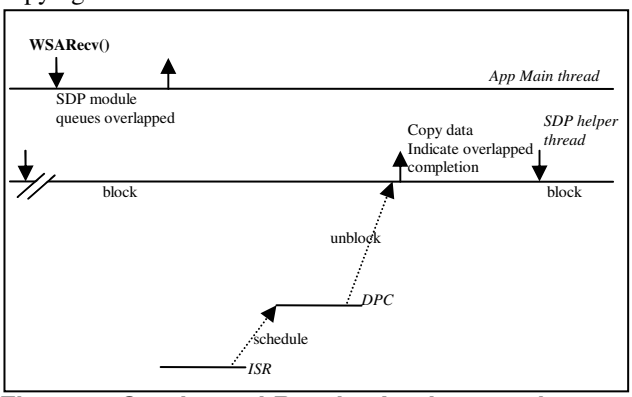

Figure 5: Overlapped Receive Implementation

A few alternatives to keep implementation simple exist, however they are not optimal. One is to directly wake up the application thread from the DPC. The application thread will have to go once again into kernel, copy the buffer and return. This saves a context switch, but is not fully optimal as it incurs an extra system call. Another alternative is to map the SDP private buffers into user. This approach is less flexible with buffer management (dynamic allocation, pooling, etc) as SDP module needs to map each buffer to the applications.

Our conclusion is that for best latency it is required to directly implement both paths: the synchronous and overlapped. It worth mentioning that the helper thread is not needed when ZCopy is used because there is no data copy.

3.3. SDP Implementation Domain

One of the tough design decisions to make was whether to implement SDP code in user mode or in kernel. The answer is not trivial, and as one can notice, there are different implementations in both domains. The Linux SDP implementation [6] is in kernel. Microsoft WSD [18] and AZ-SDP[19] are implemented in user mode. We summarize below the key factors that drove our decision to implement SDP in kernel.

Memory management is easier to handle in kernel. This includes memory pinning, memory registration with the HCA, fast registration, etc. To our knowledge it is not possible to implement registration caching in Windows except than for WSD SAN provider. The main reason is the inability to hook the VM calls. Virtual memory can change its mapping into physical address without the registration cache knowing it. As

we further look into adding zero copy to SDP, we find the kernel implementation much more suitable and effective for managing memory pinning and registration.

Interrupt overhead can be lowered if handled mostly in kernel. Interrupt overhead typically involves running an ISR in kernel and then perform the processing of the event in a DPC. Then, if a process should be woken up, the process should be unblocked and wait to be scheduled (context switch). For pure user mode SDP implementations it is required to wake up the process for every SDP control message which involves another scheduling action compared with a kernel SDP implementation. The latter can handle some of the control messages inside the DPC and thus save an extra context switch. For the case of multiple processes running concurrently, servicing events in kernel can save multiple processes scheduling for handling of control messages. It therefore expected to scale better and to reduce CPU overhead when implementing SDP is kernel.

Secondary issues are the ability to utilize resource pooling and isolation. Kernel implementation can better utilize shared resources such as SDP private buffers amongst multiple connections. It is also possible to reuse HCA resources such as MRs, QPs, etc. across connections and to report completions from multiple connections into the same CQ. Those scalability issues can be significant if many processes are using sockets concurrently.

Isolation deals with the ability to analyze failures in the field. The SDP provider dynamically runs in the same memory space with the application. Buggy application can overrun the memory used by SDP provider and cause misbehaviors that will be hard to attribute to an application or an SDP bug. By keeping SDP provider to minimum and we reduce the possible impact of application bugs on the behavior of SDP. Isolation is most likely a non-issue in production environments when applications are well written, debugged and tested.

Since we chose to implement SDP in kernel we are only able to leverage InfiniBand RDMA and Transport Offload capabilities on the data path. Kernel Bypass can not be utilized in this scheme.

3.4. API Selection

Winsock2 offers a very rich set API. At the Winsock SPI level socket calls are translated into a set of WSPxxx() calls. It is quite challenging to implement all the calls and flavors. We therefore tried to focus on a common feature set that we believe is widely used. We plan on adding more functionality in the future.

Following is the list of functions that we support: Socket, WSASocket, Connect, WSAConnect, Bind, Listen, Accept, AcceptEx, Close, Send, WSASend, Recv, WSARecv (including synchronous and overlapped), getsockname, getpeername, getsockopt, setsockopt (partial support), WSAIoctl and select. This is pretty much in line with the WSDLite[15] selection of socket calls.

We note that the Linux implementation only needed to implement a set of about 15 functions to cover full socket family support.

3.5. BCopy Implementation

We implemented BCopy path for the data transfer. The SDP module uses 16KB private buffers. Data is copied from the user buffers into SDP private buffers and then sent using IB Send operation. At the data sink, the data lands in SDP private buffers and is then copied into the application buffers.

SDP private buffers are allocated in kernel by the SDP module. They are physically contiguous. We use 1:1 mapped physical Memory Region for the HCA to access them on local scatter gather DMA.

For small messages we copy the data into the SDP private buffer and immediately post it to the HCA. We do consider adding an enhancement similar to the Nagle algorithm [20]: as long as there are outstanding messages on the wire, try to accumulate data in a local buffer instead of sending it. When reached full buffer or when wire is depleted, send the buffer. By doing that, we can effectively increase the message size on the wire and thus reduce the per user message overhead. We expect this to increase the bandwidth on small to mid-sized messages.

When we run out of SDP private buffers, we queue the overlapped requests for future processing. Once buffers are released and SDP buffers become available remotely, we copy and post the buffers to the HCA for sending. Copying is performed in this case by a helper thread on the same process. On synchronous I/Os, we just block the calling process (blocking I/O) or complete the I/O with no bytes transferred (non-blocking I/O).

On the receive side, SDP private buffers are pre-posted on the QP. When message arrives an ISR is executed and schedules a DPC. If receive application buffers are waiting, then a helper thread is scheduled to copy the data from the SDP buffers to the application buffers and complete the overlapped I/O. If application buffers are not available then the data is queued.

When the application calls WSARecv, if data is queued, it is copied into the application buffer and the helper thread is scheduled in order to complete the

overlapped I/O. If no data is queued, then the application buffer is queued until data arrives on the QP. See section 3.2 for more details.

3.6. ZCopy Implementation

ZCopy path is not yet implemented at the time of writing this paper. We definitely look into extending SDP with ZCopy support. Previous research [19][7][6] demonstrated significant performance gain from using ZCopy in Linux.

Since Winsock 2 is commonly used in Windows, we expect more applications to use overlapped I/O compared with the Linux installation base where most applications use synchronous I/O. It is much more efficient to perform RDMA ZCopy on overlapped I/Os mainly because of the ability to pipeline messages on the socket and because of the asynchronous semantics.

3.7. Socket Protocol Switch

The SDP Provider determines whether to redirect connections to SDP or to forward them to the networking stack (TCU, UDP, etc). The current policy is based on an environment variable that lists application names. For an application that appears in the list, all TCP sockets will be redirected to SDP.

For SDP candidates, the SDP Provider creates two sockets: a TCP socket and an SDP socket. All socket options (e.g. WSAIoctl) are forwarded to both sockets. At the time of resolution, WSAConnect or WSAAccept, the unneeded socket is closed.

This component can be easily extended to support administrable policy that can switch to SDP according to IP address range and port range. It is also possible to fail back to TCP in case an SDP connection establishment fails.

4. Performance Results and Analysis

We run basic performance benchmarks for latency and bandwidth. The results are listed in the following section.

4.1. Benchmarks

We measured bandwidth using ntttcp rev 2.5 benchmark. Ntttcp is a ttcp [13] version optimized by Microsoft for the Windows operating systems. The utility measures bandwidth by repeatedly sending data over a number of sockets. It supports running multiple threads simultaneously driving multiple sockets as well as overlapped and synchronous I/O.

For latency benchmarking we used netpipe 3.2.6 benchmark [12]. The utility bounces messages between two processes multiple times. The latency is calculated as the elapsed time divided by number of bounces. It represents the one way latency between the two sockets.

4.2. Benchmarking Platform

The benchmarking platform we used consists of two servers connected back to back through InfiniBand DDR (20Gb/s) link.

Servers: HP Proliant DL145 G2 (dual AMD Opteron 64 bit, 2.2GHz, 1MB Cache, 4GB RAM, NVIDIA nForce 2200 MCP chipset).

InfiniBand Host Channel Adapter: Mellanox InfiniHost III Ex DDR HCA, 4.7.600 firmware, PCI Express x8.

Software Stack: Windows XP x64. WinIB 1.3.0 (pre-release).

4.3. Unidirectional Bandwidth

Figure 6 and Figure 7 show the bandwidth test results for SDP running over one and two simultaneous connections. The CPU utilization is shown in bars.

The bandwidth peaks at 1224MB/s for a single connection (128KB messages) and at 1316MB/s for two simultaneous connections (at 128KB messages). We observe that as we add more connections the CPU utilization increase, which is related to buffer copying. We expect ZCopy to significantly release CPU resources as we add more connections. There is a bandwidth drop that begins at about 0.5MB messages. We attribute that to the CPU data cache trashing due to data copying (trashing happens at half cache size when source and destination buffers fill the entire cache).

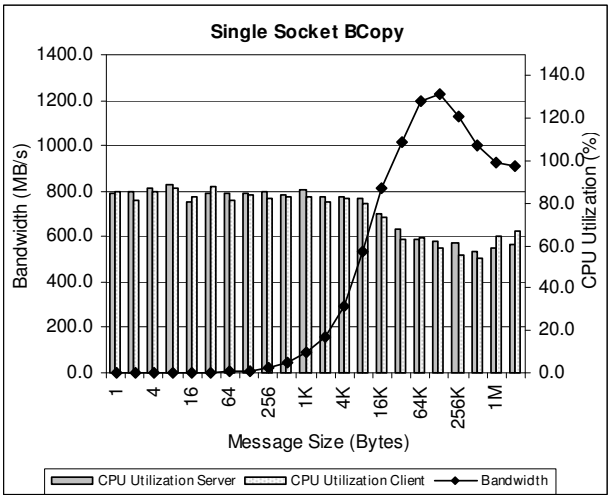

Figure 6: Single Socket Performance

For benchmarking the bandwidth we used synchronous I/Os (ntttcp without the –a option). Internally we converted each I/O operation into an overlapped operation and a wait call as explained in 3.2. Running ntttcp with asynchronous I/O showed certain performance degradation. At the time of writing the paper we still haven't fully analyzed the cause.

4.4. Latency

We measured 17.9 microseconds latency for single byte transfer. The data is shown in Figure 8. The experiment used synchronous I/O. To achieve best results, we used an SDP version that directly supports synchronous I/O. It is a bit different than the version that performs the conversion of each synchronous I/O into an overlapped operation and a blocking operation. The extra conversion significantly increases the latency as discussed in section 3.2.

5. Future Work

Adding ZCopy path to SDP is obviously the most important item for future work. Research on Linux clearly demonstrated that ZCopy is beneficial, both for throughput, CPU overhead and scalability with number of concurrent connections. Furthermore, we believe Windows overlapped I/O is commonly used and therefore applications can better benefit ZCopy.

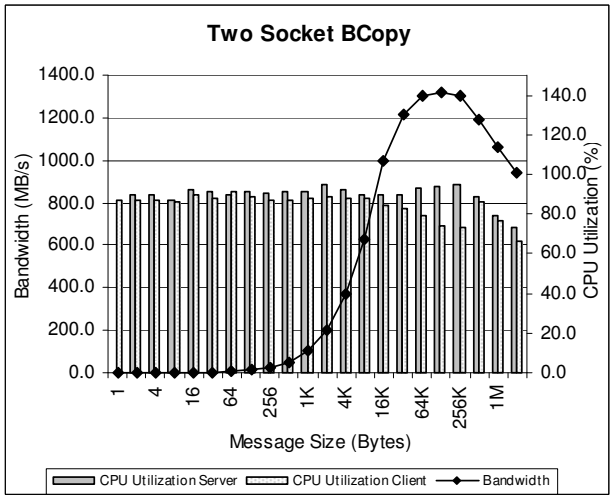

Figure 7: Two Sockets Performance

We further look into latency and bandwidth tuning. We look for alternatives to reduce the latency, possibly through polling. Improving the small to mid-sized messages bandwidth can be achieved by implementing mechanisms to accumulate multiple application messages into the same SDP buffer.

More rigorous performance analysis is encouraged, for example: performance evaluation with the scaling of the number of connections and buffer sizes, many nodes to one workloads analysis, benchmarking data center applications, comparison between Linux and Windows, comparison between WSD and SDP, comparison between SDP and alternative technologies such as iWARP and TOE, etc.

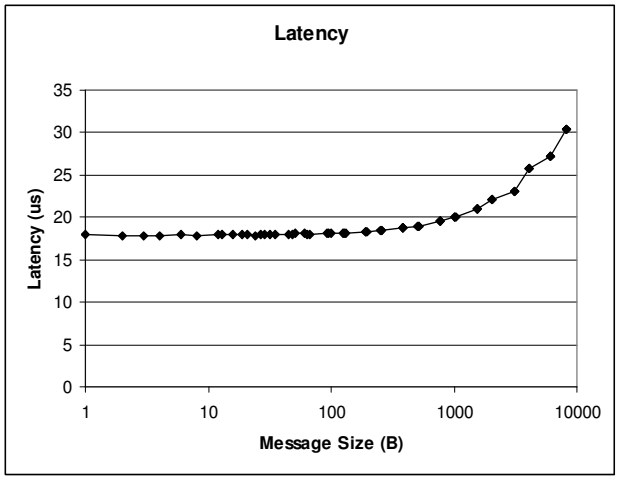

Figure 8: SDP Latency

We also look into productizing SDP. That requires an extra effort of stabilization, support for administrable policy for socket selection (TCP vs. SDP), ability to monitor SDP socket status, extend the Winsock API support, etc.

6. Conclusion

Existing socket based applications can use Sockets Direct Protocol (SDP) to transparently leverage the InfiniBand offloading capabilities. We implemented SDP protocol stack for Windows operating systems. The operating system has the right infrastructure required to interface the new SDP stack. Our implementation mainly resides in kernel and supports overlapped and synchronous I/O. We currently support the BCopy data path. We measured a record throughput of 1316MB/s at 128KB messages, which clearly demonstrates a performance advantage for using SDP. We plan on further tuning SDP performance and adding ZCopy path. The SDP stack enables interoperability with other operating systems and is fully supported on all Windows derivatives including the clients.

References

[1] The InfiniBand Architecture Specification, Release 1.2 - www.infinibandta.org/specs

[2] Open Fabrics Alliance – www.openfabrics.org.
[3] IBG2 – www.mellanox.com.
[4] WinIB – www.mellanox.com
[5] Windows Compute Cluster Server 2003 - http://www.microsoft.com/windowsserver2003/ccs/overview.mspx
[6] D. Goldenberg, M. Kagan, R. Ravid, M. S. Tsirkin, "Zero Copy Sockets Direct Protocol over InfiniBand – Preliminary Implementation and Performance Analysis", *in proc. of the 13th Symposium on High Performance Interconnects (HOTI'05)*, Aug. 2005, pp. 128-137.
[7] D. Goldenberg, M. Kagan, R. Ravid, M. S. Tsirkin, "Transparently Achieving Superior Socket Performance Using Zero Copy Socket Direct Protocol over 20Gb/s InfiniBand Links", *in proc. of the 2nd Workshop on Remote Direct Memory Access (RDMA): Applications, Implementations, and Technologies (RAIT 2005) in conjunction with Cluster 2005*, Sep 2005.
[8] RDMA Consortium SDP Specification - http://www.rdmaconsortium.org/home/draft-pinkerton-iwarp-sdp-v1.0.pdf
[9] P. Balaji, S. Narravula, K. Vaidyanathan, S. Krishnamoorthy, J. Wu, and D. K. Panda, "Sockets Direct Protocol over InfiniBand in Clusters: Is it Beneficial?", *in proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 04)*, Mar. 2004, pp. 28-35.
[10] IEEE Std 1003.1, 2004 Edition
[11] Extended Socket API (ES-API), Issue 1.0 – www.opengroup.org/icsc
[12] netpipe - http://www.scl.ameslab.gov/netpipe/
[13] ttcp - http://ftp.arl.mil/~mike/ttcp.html
[14] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overheads", *IEEE Communication Magazine*, Vol. 27, No. 2, June 1989, pp. 23 – 29.
[15] E. Speight, H. Shafi, and J.K. Bennett, "WSDLite: A Lightweight Alternative to Windows Sockets Direct Path", *in proc. of the 4th USENIX Windows Systems Symposium*, August 3-4, 2000.
[16] H. V. Shah, C. Pu, and R. S. Madukkarumukumana, "High Performance Sockets and RPC over Virtual Interface (VI) Architecture", *in proc. of CANPC workshop (in conjunction with HPCA)*, 1999, pp. 91-107.
[17] Winsock2 - http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/windows_sockets_start_page_2.asp
[18] WSD - https://www.microsoft.com/whdc/device/network/san/WSD-SAN.mspx
[19] P. Balaji, S. Bhagvat, H.W. Jin, and D. K. Panda "Asynchronous Zero-copy Communication for Synchronous Sockets in the Sockets Direct Protocol (SDP) over InfiniBand" *in proc. of Communication Architecture for Clusters (CAC'06)*.
[20] Nagle Algorithm – RFC 896