computing@computingonline.net
www.computingonline.net

ISSN 1727-6209
International Journal of Computing

BIG DATA TRANSFER FOR TABLET-CLASS MACHINES

Tevaganthan Veluppillai [1], Brandon Ortiz [2], Robert E. Hiromoto [3]

[1), 3)] University of Idaho, Idaho Falls, Idaho 83402, USA, {hiromoto@uidaho.edu, veluteva@isu.edu}
[2)] University of Idaho, Moscow, Idaho 83844-1010, USA, brandon.ortiz@vandals.uidaho.edu

Abstract: Several well-known data transfer protocols are presented in a comparative study to address the issue of big data transfer for tablet-class machines. The data transfer protocols include standard Java and C++, and block-data transfers protocols that use both the Java New IO (NIO) and the Zerocopy libraries, and a block-data C++ transfer protocol. Several experiments are described and results compared against the standard Java IO and C++ (stream-based file transport protocols). The motivation for this study is the development of a client/server big data file transport protocol for tablet-class client machines that rely on the Java Remote Method Invocation (RMI) package for distributed computing. *Copyright © Research Institute for Intelligent Computer Systems, 2013. All rights reserved.*

Keywords: Blocked file transport; stream-based; block-oriented; tablet machines; client/server configuration.

1. INTRODUCTION

Big data transfer across distributed client/server systems has become a major concern in all fields of information processing. Numerous big data transport utilities have been developed to address the demands in area such as genomics and cancer research [1, 2], high-energy particle physics [3], and GIS analysis [4].

Today's big data file transfer utilities rely on streams of either a single striped file or multiple individual files, whose parallel send/receive client-server strategies are the basis for the increase in big data throughput. The GridFTP File Transfer Protocol [5] is one such. GridFTP is characterized by a fast file transfer protocol that supports large files, secure file transfers, capabilities for multiple destination points for file transfers, and an API that allows various file transfer capabilities. GridFTP is part of the Globus Toolkit.

Current big data transfer tools are designed to make optimal use of hardware parallelism on both the client and server sides of a distributed cluster system. File transport, in such an environment, can be organized into parallel file transfer streams. BBFTP [6], BBCP [7] and the Fast Data Transfer (FDT) [8] utilities are conceptually similar to the GridFTP parallel file streaming approach. FDT, for example, is written in Java and has the capability to run on all major computer platforms. In addition, FDT uses the Java New IO (NIO) [9], where files are processed in blocks-of-bytes rather than the byte-by-byte data stream as performed by standard Java IO.

The introduction of tablet-class machines such as the Apple iPad and Android tablets extends the paradigm of a client-server and opens up the speculation of how big data transfer capabilities can be provided. The challenges posed by tablet-class machines are underscored by their limited hardware resources: the lack of a disk array storage facility, the limited number of communication ports, underclocked processors to reduce heat production, and flash-based memory that are more susceptible to failure. However, one advantage offered by off-the-self tablets is its processor configuration that supports at least a dual-core processing unit. Although limited, this processor parallelism provides a means to implement producer/consumer data transfer strategies. The availability of a producer/consumer data transfer capability and the use of a high-throughput data transfer protocol would provide big data solutions to tablet-class machines. In this paper, we limit the focus to the design of a data transfer utility that can support a tablet-class, client-server environment for big data transfers.

As mentioned above, parallel transfer of large striped-data files provide high throughput by utilities such as GridFTP and FDT. Unfortunately, these approaches require high-end peripheral hardware to capture, coordinate and merge the multiple concurrent streams of a single large data file that arrives at the client-end. Tablet machines are at a

significant disadvantage and as such a more modest yet robust data transfer strategy is required.

This paper is an extension of our presentation made at the IAACS 2013 workshop held in Berlin [10], we limit the focus to the design of data transfer utilities that support a tablet-class, client-server environment for big data transfers. This paper provides a comparative analysis of several Java and C++ approaches that introduces optimizations to reduce the standard Java and C++ IO overheads in filling the socket buffer. Java NIO and Zerocopy [11] are well known techniques described and compared in this paper. A blocked (buffered) Zerocopy is also described and presented as an analog to the NIO method. It should be pointed out that Zerocopy requires a Linux or Unix OS but that should not be an issue on the server side of the network. In addition comparisons with C++ byte-by-byte streaming and a C++ blocked data transfers are used in comparing the different strategies.

In the next sections, assumptions regarding the use of Java New IO (NIO) and Java Zerocopy are described. The proposed approach describes the integration of Zerocopy and NIO with data transfer timing results provided. To complete the comparison, file transfer times between a standard C++ and a block-data transfer are also provided.

2. BACKGROUND

Tablet-class client machines represent a collection of window-oriented technologies for which no unique operating system dominates the industry. In such an environment, the design of a generic client-server data transfer tool must rely on programming languages that are compatible on any and all computer platforms. Java is one such language that bridges this gap but also provides a Remote Method Invocation (RMI) capability that supports coordination between distributed computer platforms seamlessly. The RMI mode can be relatively slow since the instructions are interpreted. However, the use of RMI as a coordinator of a distributed client-server architecture is not a computationally intensive task; for that reason the application of RMI should not incur substantial overhead delays. On the other hand, the internal Java IO stream can lead to TPC/IP overhead delays.

Java is an object-oriented language that employs a byte-by-byte streaming IO process in preparing the socket buffer for data transfer into the network interface card (NIC) buffer and transferred to its destination. At such a fine level of data granularity, Java IO is inefficient and does not scale well.

Block IO data transfers are a more efficient alternative. As such, Java NIO was developed as a block-oriented approach. Java NIO provides the flexibility to adjust the IO block size, which can potentially affect the TPC/IP bandwidth-delay product (BDP) and enhance its throughput performance [12].

In addition to Java NIO, an optimized treatment of internal data copying, known as Zerocopy, was developed and made available in the NIO library. The next two sections will describe the NIO and Zerocopy approaches. The integration of these two methods forms the basis of the desires tablet client-server big data file transfer mechanism.

2.1. JAVA NIO

Java NIO is an open source library developed and maintained by Sun (Oracle). NIO provides block-oriented IO transfers of a file. The strategy of sending a file in a block-wise fashion reduces the software management needs for packetized byte information. The Channel and the Buffer are the principle NIO objects. Channels are analogous to the original Java IO but are bidirectional. In this sense, a channel can be opened for read or for write or for both. Data that is written into a channel must first be written into a buffer, and data that is read from a channel is read into a buffer. A buffer is an object that holds the data that has been read from the channel or holds the data that is to be written into the channel. In the NIO library, all data is handled with buffers. The interplay between the Channel object and the Buffer object marks the operational difference between Java IO and Java NIO. In Java IO, data is written and read directly from Stream objects. NIO allows, therefore, a pipeline between the Channel object and the Buffer object to hide access latencies.

We give two coding examples of NIO that illustrates reading from and writing to a file. A more complete description of NIO can be found in the introductory tutorial [13].

Reading a file requires three steps. First, a channel is acquired by creating a FileInputStream using the original Java IO library:

```
FileInputStream fin = new FileInputStream( "r.txt" );
FileChannel fc = fin.getChannel();
```

Second, a Buffer object is created:

```
ByteBuffer buffer = ByteBuffer.allocate( buff_Size );
```

Third, the data is read from the Channel into the Buffer:

```
fc.read( buffer );
```

This example points out an important aspect of the Channel and Buffer objects. Notice that the

coding of the channel does not explicitly indicate the amount of data that needs to be read into the buffer. As a consequence, Buffer objects are endowed with an internal accounting system that keeps track of the amount of data that has been read and the amount of buffer space remaining for additional data. This capability will be useful in our future implementation of a producer/consumer IO strategy.

The second example is writing to a file. First, a channel is created:

```
FileOutputStream fout = new FileOutputStream( "w.txt" );
FileChannel fc = fout.getChannel();
```

Second, a buffer is created and then data written into it:

```
ByteBuffer buffer = ByteBuffer.allocate(buff_Size );
for (int i=0; i<message.length; ++i) {
    buffer.put( message[i] );
}
buffer.flip();
```

Third, write into the buffer:

```
fc.write( buffer );
```

As in the previous example, the internal accounting system of the buffer automatically tracks the amount of data written into the buffer and the remaining buffer space for which additional data can still be written. The *buffer.put*() method fills the buffer with data, and the *buffer.flip*()method readies the newly filled buffer data to be written to another channel.

Notice that the allocate() method defines the block-oriented IO size. This block size parameter can be dynamically adjusted to affect TCP/IP performance in combination with algorithms such as Fast TCP [14].

NIO supports memory-mapped file IO. This method when applied to reading and writing file data can greatly improve channel-based IO as well as the original Java IO. Memory mapping is an OS capability where the file system maps portions of a file into portions of the memory on demand.

The following code is an example of how a FileChannel() or portions of it can be mapped into memory:

```
MappedByteBuffer mbb =
fc.map(FileChannel.MapMode.READ_WRITE, 0,
buff_Size );
```

The *map*() method returns a *MappedByteBuffer* as a subclass of *ByteBuffer*. Any manipulations using this buffer are automatically mapped to memory on demand by the operating system.

It should be pointed out that the Fast Data Transfer (FDT) IO tools is based in part on NIO.

2.2. ZEROCOPY

Zerocopy is a stream-based file transfer library that differs from Java IO in that it reduces the number of internal data copying and associated context switches. Fig. 1 illustrates the Java IO data copying behavior when a request to send a file from the server to a remote client. The following code represents the data flow encountered in Fig. 1:

```
File.read(fileDesc, buf, len);
Socket.send(socket, buf, len);
```

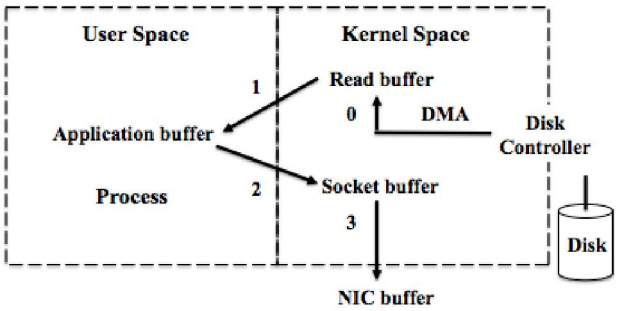

Fig. 1 – Java IO internal data movement & copying [11].

Table 1. Java IO context switching [11].

Time Sequence	User Context	Kernel Context
	Before Read	
T_0		Syscall Read
T_1	Before Send	
T_3		Syscall Write
T_3	Next Cycle	

We see in this illustration that the file's data flow copies the file elements into the Read buffer that is then copied into the application buffer then into the Socket buffer, and finally into the NIC buffer. In order to handle these internal data transfers, the OS intervenes with a corresponding number of context switches. Table 1 lists the temporal order and number of context switches incurred by Java IO. Zerocopy mitigates the number of copying required by Java IO by copying the content of the Read buffer directly into the Socket buffer. The *transferTo*() method in Zerocopy bypasses the Application buffer and copies the Read buffer directly to the Socket buffer. UNIX and various flavors of Linux operating systems support *transferTo*() by routing the method invocation to the *sendfile*() system call. Rather than relying on the two methods File.read() and Socket.send(), Zerocopy is expressed by a single call:

```
transferTo(position, count, writableChannel);
```

Fig. 2 illustrates the Zerocopy approach. From Table 2, the associated number of context switching is reduced from four to two.

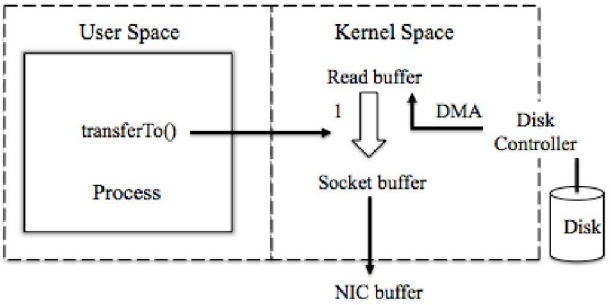

Fig. 2 – Zerocopy internal data movement & copying [11].

Table 2. Zerocopy context switching [11].

Time Sequence	User Context	Kernel Context
	Before transferTo()	
T_0		Syscall Read and Send
T_1	Next Cycle	

2.3. ZEROBUFFER

Unlike NIO, Zerocopy does not rely upon the application buffer to assist in file transfers under the TCP protocol. As a consequence Zerocopy lacks the block-oriented IO structure required to support parallel file transfers, as does NIO. A closer examination of the *transferTo*() method, however, finds that a position index can be set to a beginning location of the file to be transferred. This capability can be used to transform Zerocopy into a block-oriented IO implementation, which is referred to as ZeroBuffer. The following code fragment details the block-oriented ZeroBuffer implementation:

```
fc = new FileInputStream(input).getChannel();

while (position != fc.size())
{
position += fc.transferTo(position, bufferSize, sc);
}
```

The file channel *fc* is initialized and set to the new *FileInputStream*. Within the *while* loop, the *transferTo*() method is invoked with a position indicator that points to a location within the file, a block *bufferSize*, and the socket channel descriptor *sc*.

The complete programs for all the file transfer tests are available at [15].

3. FILE TRANSFER COMPARISIONS

Table 3 provides the detail of the Client/Server machine properties. Table 4 summarizes the file transfer times (milliseconds) between Java IO and Zerocopy for which both use stream-based (byte-by-byte) data transfer methods. The results of these tests indicate a 50 times performance increase of Zerocopy over Java IO. Similar results are reported elsewhere in the literature [11]. The test used file sizes ranging from 0.3 MB to 33 MB. Each file is run ten times to determine a "best" file transfer time. An average transfer time is not reported since the network utilization can distort the significance of average values.

Table 3. Machine functions and properties.

Machine Function	Machine Types	Processor Configuration	Memory System	OS
Server	MacBook	2.8 GHz Intel Core i7	8 GB 1333 MHz DDR3	OS X Lion
Client	MacPro	2 x 2.4 GHz Quad-Core Intel Xeon	16 GB 1066 MHz DDR3 ECC	OS X Lion

NIO and ZeroBuffer are characterized by explicit block (buffer)-size assignments, which from Fig. 1 is referred to as the application buffer. Table 5 lists file transfer times measured for Zerocopy, NIO and ZeroBuffer. Although, Zerocopy is independent of the application buffer size its transfer time is listed for each buffer for comparison purposes only. The best NIO and ZeroBuffer results are listed as a function of their corresponding application buffer sizes, which range from 1 KB to 16 MB. The results presented in this paper are for a 1.03 GB file. Other results for smaller files produced similar results.

The experiments also varied the TCP Send/Recv buffer sizes. The Mac OS X sets the default size of the Send/Recv buffer to 64 KB. The TCP Sendbuffer (on the server-side) is manually adjusted and is coordinated with a reciprocal assignment (on the client-side) for the TCP Receive buffer using the same size.

Table 4. Data transfer times.

Files (MB)	Java IO (ms)	Zerocopy (ms)
File1 (0.3)	7	3
File2 (0.6)	13	7
File3 (1.2)	3687	47
File4 (2.5)	6103	107
File5 (4.9)	14969	298
File6 (9.9)	29307	564

Fig. 3 is a plot of the corresponding file transfer times listed in Table 5. The TCP Send/RecvSpace buffer size is manually set to 64 KB. NIO, Zerocopy

and ZeroBuffer deliver comparable transfer times. There are several items to point out. First notices that ZeroBuffer suffers noticeable overheads for small application buffer sizes (1 KB to 2 KB); whereas, the while-loop introduced to create its block-oriented IO structure shows little difference in comparison with Zerocopy. This is quite surprising given the somewhat "artificial" block-oriented approach taken. The second item to note is the behavior of the NIO method. NIO exhibits file transfer slowdowns for small application buffer sizes (1 KB to 2 KB); as well as for larger buffer sizes (2 MB to 16 MB). Overall, NIO and ZeroBuffer methods are comparable to the file transfer time of Zerocopy, between the application buffer sizes ranging from 4 KB to 1 MB.

Table 5. File size = 1.03 GB (TCP Send/RecvSpace = 64 KB).

Application Buffer Size	NIO (ms)	ZeroBuffer (ms)	Zerocopy (ms)
1KB	109.88	135.15	88.2
2KB	96.62	100.8	88.2
4KB	88.96	88.4	88.2
8KB	89.69	89.05	88.2
16KB	90.29	89.11	88.2
32KB	89.23	89.62	88.2
64KB	90.26	89.12	88.2
128KB	89.79	88.98	88.2
256KB	88.46	88.81	88.2
512KB	88.45	88.36	88.2
1MB	89.97	88.28	88.2
2MB	92.22	88.55	88.2
4MB	94.06	88.45	88.2
8MB	94.55	89.03	88.2
16MB	94.7	88.51	88.2

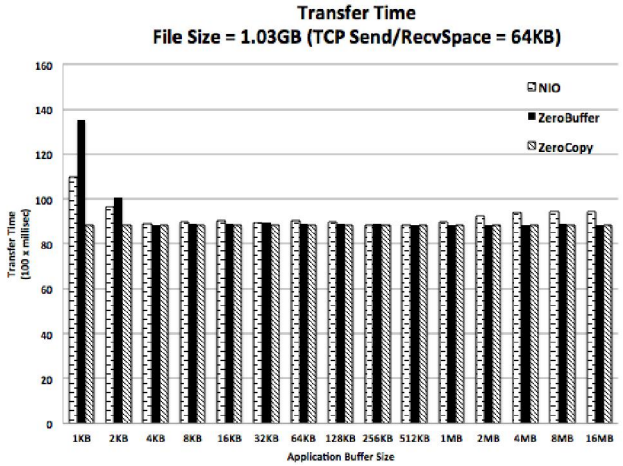

Fig. 3 – A plot of the Table 5.

The TCP Send/RecvSpace buffer settings are considerations based on work by several authors investigating [16, 17, 18] the optimal tuning of TCP file transfers. Starting with the TCP Send/RecvSpace buffer sizes of 64 KB, 156 KB, 256 KB, 512 KB, and 1 MB, we find little difference in file transfer performance. What we observed, however, is that the file transfer performance for NIO improves with increasing TCP Send/RecvSpace buffer size. Fig. 4 and Fig. 5 are presented to illustrate this observation.

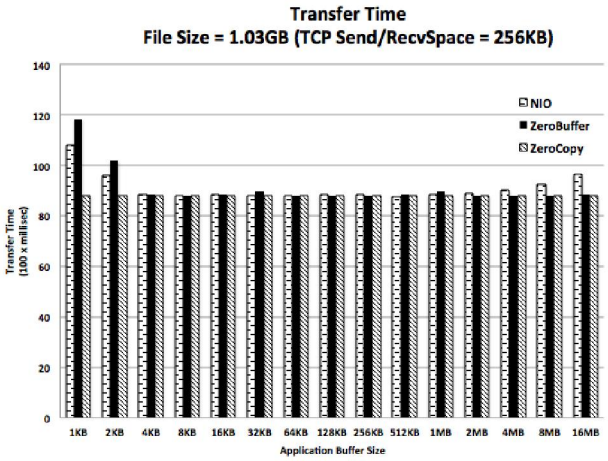

Fig. 4 – Results for TCP Send/RecvSpace = 256KB.

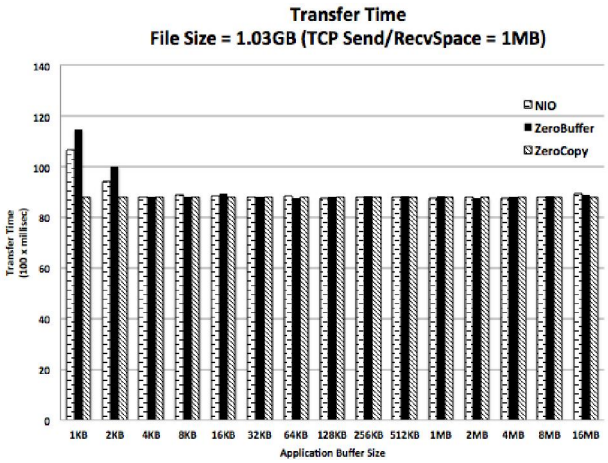

Fig. 5 – Results for TCP Send/RecvSpace = 1MB.

As a final set of comparisons, we examine the transfer speeds that can be obtained with a byte-by-byte and block or buffered data transfer approaches using C++. The two methods are referred to as C++(1) and C++(2), respectively. These methods are analogous to standard Java and Java NIO, whereas a C++ version that is comparable to Zerocopy was not readily available for this study. Fragments of the socket program for each method are provided below [19, 20]. C++(1) uses a simple *while* construct:

```
int ch;
char toSEND[1];

while((ch=getc(file))!=EOF) {
        toSEND[0] = ch;
        send(socket, toSEND, 1, 0);
}
```

C++(2) uses a buffered send size of 1024 bytes as depicted in the following code fragment:

```
char buf[1024];

while (!feof(file)) {
int rval = (int)fread(buf, 1, sizeof(buf), file);
int sent = (int)send(sock, &buf[off], rval - off, 0);
off += sent;
}
```

The TCP Send/RecvSpace for the server processor is maintained at a size of 131,072. On the client side, the TCP Send/RecvSpace is fixed at a size of 65,536. No attempts are made to study the transfer times as a function of the TCP Send/RecvSpace. The sole purpose of this experiment is to compare the relative performance of the various C++ file transfer methods with Zerocopy and ZeroBuffer. The client/server configuration is hosted on a local area network (LAN) within the Center for Advance Energy Studies CAES) in Idaho Falls. We are currently working towards the assessment of file transfer performances over a wide area network (WAN).

Table 6 displays the data transfer times for the different Java and C++ programs. The time is given in milliseconds. The files range in size from 0.256 MB up to 16.4 MB. The buffer size used for ZeroBuffer and C++(2) is set at 1024 bytes.

Table 6. Data transfer times.

File Size (MB)	ZeroBuffer (ms)	Zero (ms)	C++(2) (ms)	C++(1) (ms)
0.256	9	8	1	333
0.512	16	18	2	596
1.0	31	31	4	1158
2.0	55	54	7	2267
4.1	83	85	16	4236
8.2	143	151	34	8273
16.4	161	164	67	16636

Notice that the data transfer rates for the C++(2) implementation is far superior to its byte-by-byte transfer counterpart. This is consistent with results between Java and Java NIO. In either case the block-data transfers utilizes the I/O subsystems more efficiently. This of course is a well-known result. The transfer times of C++(2) varies by a factor of 9 to 2.5 times faster in comparison to the corresponding Zero{Buffer, copy} transfer times; however, notice that the speedup factor decreases with increases in file size. In other words, it appears that for much larger file sizes C++(2) may reach parity with Zero{Buffer, copy} transfer rates. To test this conjecture, we ran two test cases using file sizes of 0.734 GB and 1.47 GB. Table 7 presents the results for C++2, Zero copy and Zero copy + buffer. Again a buffer size of 1024 bytes is used.

Table 7. Data transfer times.

File Size (GB)	ZeroBuffer (ms)	Zero (ms)	C++(2) (ms)
.734	7726	6092	5068
1.47	15194	13332	10098

Comparing the ratios between Zero (copy) and C++(2), the transfer rates of C++(2) now appears to be given by factors of 1.2 (0.734 GB) and 1.32 (1.47 GB), respectively. These results are in good agreement with our prior experiments.

We make one final observation with regards to the transfer of large files. In Figs. 3, 4, and 5 ZeroBuffer consistently underperforms Zero (copy) for buffer sizes less than 4 K bytes. For buffer sizes of 4 K bytes and larger, ZeroBuffer is comparable in file transfer times to Zero (copy). To test the consistency of this behavior, Table 8 list the results of performing the same file transfer experiments but using a buffer size of 5K bytes. For both the 0.734 GB and 1.47 GB files, ZeroBuffer and Zero are comparable in transfer time as argued. What is not expected is the reduction in transfer times for C++(2) by more than 50 %, and between 2.8 to 3.25 speedup over ZeroBuffer.

Table 8. Data transfer times (Buffer = 5 KB).

File Size (GB)	ZeroBuffer (ms)	Zero (ms)	C++(2) (ms)
0.734	6537	(6092)	2010
1.47	12865	(13332)	4583

In Table 8, the times for Zero (copy) are inserted only for comparison purposes (recall that Zero (copy) is buffer size independent).

Determining a more optimal buffer size for C++(2) requires further experiments; although, we suspect that further experiments with larger files and larger buffer sizes will exhibit behavior similar to those illustrated in Figs. 3, 4, and 5.

4. SUMMARY AND CONCLUSION

We developed data transfer programs using commonly available socket libraries. The socket programs for NIO, Zerocopy, and C++ are straightforward and required no special programming considerations. The timer resolution of the C++ programs is based on the Apple LLVM compiler and developed under the XCode Integrated Development Environment (IDE) [21].

The comparative collections of I/O results are presented. The programs build upon standard Java and C++ techniques and libraries available to the general users. More specialized routines might be found at advanced application websites or through proprietary sources.

The data streaming strategy of Java IO and C++ are known to be inefficient for big data transfers. In this study, block-oriented IO methods are compared to an efficient non-blocked IO method known as Zerocopy. In addition, this paper provides the first direct comparison between Zerocopy and NIO.

A block-oriented IO method (ZeroBuffer) is introduced that supports Zerocopy efficiencies over a large range of application and TCP buffer sizes. Although the Zerocopy method can be reliably used to address the appetite of tablet-class client/server file transfers, ZeroBuffer has the potential to support non-blocking, concurrent IO threads, which is an important feature of high-end grid-IO.

Overall, the buffered C++ implementation proved to be the fastest of the file transfer codes, but required a buffer size greater than 4 K bytes. ZeroBuffer is shown to have similar behavior; although, not as fast.

A further advantage may be gained when clients and server are hosted on a WAN where the bandwidth utilization for the transfer of large data files is strongly influenced by the behavior of the TCP network protocol under extreme external demands. In this regard, the block-data transfer mode can provide a self-throttling mechanism to reduce the TCP overhead latencies experienced by large data transfers initiated in a single-continuous send-operation mode over the network.

Finally, this paper represents the initial stages of a much more ambitious effort to provide serious information discovery and analysis on tablet-class machines. To achieve this goal, a new paradigm of big data transfer should be considered and realized: raw data should never be transferred across wide area networks. Instead raw data should be localized in persistent data servers where stored data should only be marshaled into a form that is information-dense; that is, a form that can be easily visualized, virtualized, and orders of magnitude smaller in size than its original raw data footprint. Optimal data transfer time or data compression is not enough to achieve this goal. A big data system will likely require block-algorithms to sustain a producer/consumer-like pipelined data transfer protocol, where blocks of data are computed and transferred across the network in an overlapping pipeline fashion. In order to sustain this operation, a distributed server farm that coordinates the in-situ placement of raw data and its access will prove to be advantageous.

5. ACKNOWLEGMENTS

This research is being performed using funding received from the DOE Office of Nuclear Energy's Nuclear Energy University Programs.

6. REFERENCES

[1] C. Wang, D. Zhang. A novel compression tool for efficient storage of genome resequencing data, *Nucleic Acids Research*, Vol. 39, Issue 7, April 2011, e45.

[2] Lynda Chin, William C. Hahn, Gad Getz, et al. Making sense of cancer genomic data, *Genes & Development*, Vol. 25, Issue 6, 2011, pp. 534-555.

[3] http://supercomputing.caltech.edu (last accessed 15, Feb., 2013.)

[4] M. J. de Smith, M. F. Goodchild, P. A. Longley. *Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools*, 2nd edition, Troubador, UK, 2007.

[5] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster, The globus striped GridFTP framework and server, *Proceedings of the ACM/IEEE conference on Supercomputing SC'05*, ACM Press, November 2005, pp. 54.

[6] http://doc.in2p3.fr/bbftp/

[7] http://www.slac.stanford.edu/~abh/bbcp/

[8] R. S. Prasad, M. Jain and C. Dovrolis, Socket Buffer Auto-Sizing for High-Performance Data Transfers, *Journal of Grid Computing*, Vol. 1, Issue 1, 2003, pp. 361-376.

[9] Getting started with NIO, https://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio-pdf.pdf.

[10] V. Tevaganthan, B. Ortiz, R. E. Hiromoto, A big data file transfer tool for tablet-class machines, *Proceedings of the IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS'2013)*, Berlin, Germany, September 12-14, 2013, pp. 676-680.

[11] S. K. Palaniappan and P. B. Nagaraja, Efficient data transfer through zero copy: Zero copy, zero overhead, 2 Sep 2008, http://www.ibm.com/developerworks/library/j-zerocopy/

[12] M. Jain, R. S. Prasad and C. Dovrolis, The TCP bandwidth-delay product revisited: network buffering, cross traffic, and socket buffer auto-sizing, http://hdl.handle.net/1853/5920.

[13] Greg Travis, Getting started with new I/O (NIO) skill level: introductory,

http://www.ibm.com/developerworks/java/tutorials/j-nio/j-nio-pdf.pdf.
[14] C. Jin et al., FAST TCP: from theory to experiments, *IEEE Network*, Vol. 19, Issue 1, 2005, pp. 4-11.
[15] http://datatransfer.codeplex.com
[16] S. Thulasidasan, W. Feng, M. K. Gardner. Optimizing GridFTP through dynamic right-sizing, *Proceedings of IEEE International Symposium on High Performance Distributed Computing*, 2003, pp. 14-23.
[17] J. Semke, J. Mahdavi, M. Mathis, Automatic TCP buffer tuning, *Computer Communication Review*, Vol. 28, 1998, pp. 315-322.
[18] R. S. Prasad, M. Jain, C. Dovrolis, Socket buffer auto-sizing for high-performance data transfers, *Journal of Grid Computing*, Vol. 1, Issue 4, 2003, pp. 361-376.
[19] *Beej's Guide to Network Programming*, Jorgensen Publishing, October 20, 2011.
[20] http://www.linuxhowtos.org/C_C++/socket.htm (last accessed 16 Dec., 2013).
[21] https://developer.apple.com/technologies/tools/ (last accessed 31 Dec., 2013).

Tevaganthan Veluppillai *is a Master's student at the University of Idaho. He is developing of client/server system for tablet-class client machines.*

Brandon Ortiz *is a PhD student at the University of Idaho. His interests are in the development of wireless communication protocols for autonomous vehicles.*

Robert E. Hiromoto*, received his Ph.D. degree in Physics from University of Texas at Dallas. He is professor of computer science at the University of Idaho. His areas of research include information-based design of computational algorithms, and information processing applied to decryption techniques and secure wireless communication protocols.*