Efficiency Comparison of Blocking and Non-blocking TCP/IP Sockets using Java in Single and Multi-core Systems

Dmitry Viktorov, Julius Dichter
Computer Science and Engineering Department, University of Bridgeport, Bridgeport, CT
{dviktoro, dichter}@bridgeport.edu

Abstract **– Blocking sockets had been the default method of TCP/IP communications in Java before the non-blocking sockets were introduced in Java 1.4. Although programmers can now use any of these sockets, they may have doubts about what kind is more efficient. This issue becomes especially complicated if they use sockets in multi-core systems which have lately started to be commonly used. This paper is aimed to test both kinds of sockets in Java and compare their efficiency in single and multi-core systems. We run experiments and conclude how to get the best performance form client server applications for varying numbers of clients and application buffer sizes using both types of sockets in 1, 2, and 4-core architectures.**

Keywords **– Java, Blocking Sockets, Non-blocking Sockets, Multi-core Systems, Efficiency Comparison.**

I. INTRODUCTION

First of all, we mean the Berkeley socket interface when we talk about sockets in this paper. It was first developed at the University of California, Berkeley for use on Unix systems.

The Berkeley socket interface allows communications between hosts or between processes on one computer, using the concept of an Internet socket. It can work with many different I/O devices and drivers, although support for these depends on the operating-system implementation. This interface implementation is implicit for TCP/IP, and it is therefore one of the fundamental technologies underlying the Internet. All modern operating systems now have some implementation of the Berkeley socket interface, as it became the standard interface for connecting to the Internet.

Berkeley sockets can operate in one of two modes: *blocking* or *non-blocking* [1]. Both modes are now supported by Java and we describe them in the next section. In this paper we analyze the performance of both modes in Java and compare their efficiency using computers with different number of cores. After the tests are done, we make the conclusion about their usage.

II. BACKGROUND

Figure 1 from Calvert [2] shows the structure of a TCP/IP connection and relationships among the protocols, applications, and the sockets API (Application Programming Interface) in the hosts and routers, as well as the flow of data from one application to another. As it is depicted, the socket is the entity that is placed between the transport layer and the application layer. Operating with the socket, one of the first issues that a programmer encounters is the difference between blocking and non-blocking sockets [7].

Whenever some operation is performed on a socket, it may not be able to complete immediately and return control back to the program. For example, a read on a socket cannot complete until some data has been sent by the remote host. If there is no data waiting to be read, one of two things can happen: the function can wait until some data has been written on the socket, or it can return immediately with an error that indicates that there is no data to be read.

The first case is called a *blocking socket*. In other words, the program is "blocked" until the request for data has been satisfied. When the remote system does write some data on the socket, the read operation will complete and execution of the program will resume.

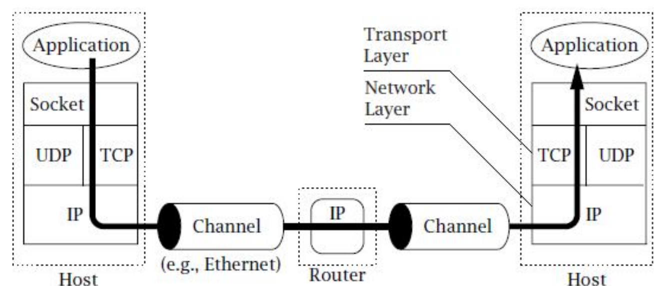

Figure 1. A TCP/IP Connection.

To sum up, a *blocking* socket will not return control until it has sent or received all data specified for the operation. Using the blocking sockets we are bounded providing service for only one client. If a multi-user interface must be supported, we need to start a separate service thread with the blocking socket for each client.

The second case is called a *non-blocking socket*, and requires that the application handle the situation appropriately. Programs that use non-blocking sockets typically use one of two methods when sending and receiving data. The first method, called polling, is when the program periodically attempts to read or write data from the socket using a timer. The second, and preferred method, is to use what is called *asynchronous notification*. This means that the program is notified whenever a socket event takes place, and in turn can respond to that event. For example, if the remote program writes some data to the socket, a "read event" is generated so that program knows it can read the data from the socket at that point.

For historical reasons, the default behavior is for socket functions to "block" and not return until the operation has completed. Under Windows this can introduce some special problems because when the program blocks, it enters what is called a "message loop", where it continues to process messages sent to it by Windows and other applications. Since messages are being processed, this means that the program can be re-entered at a different point, with the blocked operation "parked" on the program's stack. For example, consider a program that attempts to read some data from the socket when a button is pressed. Because no data has been written yet, thread blocks, and the program hangs at the current message loop. If the thread didn't block, the user could press a different GUI button, causing code to be executed, which might attempt to read data from the socket, and so on. Obviously, this could pose a big problem. To resolve it, the Windows Sockets standard states that there may only be one outstanding blocked call per thread of execution. This means that, in the example given above, the second read is not possible. While blocking sockets may be convenient in some situations, their use is generally discouraged because of the complications that they can introduce into the program.

A *non-blocking* socket gives us the power to monitor several sockets at the same time. It will tell us which ones are ready for reading, which are ready for writing, and which sockets have raised exceptions. So, for the case of non-blocking sockets we can serve many clients in one thread.

The blocking sockets were available in Java since its first version but the non-blocking sockets were added only in the version 1.4 as NIO ("New I/O") packages [2], [5]. The main reason for this addition is that the thread-per-client approach for blocking sockets is limited in terms of scalability because of the overhead associated with creating, maintaining, and switching between threads. Some modern applications have to serve so many clients that the blocking approach becomes unmanageable. The detailed description of this issue may be found in [4]. This article considers two web-servers one of which is blocking and another one is unblocking. As the article describes, the large number of Java threads in the blocking approach will cause the JVM and OS busy with handling scheduling and maintenance work of these threads, instead of processing business logic. Moreover, more threads will consume more JVM heap memory (each thread stack will occupy some memory), and will cause more frequent garbage collection.

III. TESTING MODEL

The system model is based on the application that sends data from the server to the client over a LAN with TCP/IP. Its functioning scenario (Figure 2) is very simple. The client connects and sends a data set to the server. This data set is randomly generated once as part of the initialization of the client, and then placed into the client's buffer. The server simply responds to the client by sending back the same data.

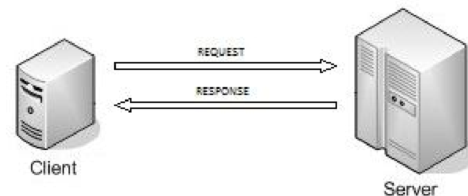

Figure 2. Functioning scenario.

When the client sends data to the server, it doesn't send the whole buffer at once but instead sends as many bytes from its buffer as possible at the present moment calling the "write" method only once. After that, the client knows that it must read the same amount of bytes that it sent before, so it will keep calling the "read" method until the bytes are read.

The server does the same as the client but inversely. After the connection is established, it reads as many bytes as possible from the incoming stream into its buffer calling the "read" method only once. After it is done, it tries to send all the read bytes back calling the "write" method until all is written.

This above-described process applies to non-blocking sockets. For blocking sockets, it differs in the following important way: The "write" method immediately blocks and remains blocked until the whole buffer is sent.

The server and the client are implemented for both kinds of sockets, blocking and non-blocking. They use the same algorithms and are absolutely compatible with each other. In other words, the blocking server can work with the non-

blocking client and the non-blocking server with the blocking client.

Both types of servers support multi-connections, i.e. several clients may connect at the same time. For this testing model we use only one computer to start several clients in separate threads and another equivalent computer to run the server. This allows us to compare the efficiency of the sockets using multi-threading, a natural efficient programming technique which takes advantage of multi-core systems. We use single-threaded, four-threaded, eight-threaded, and sixteen-threaded client applications for the tests.

Another important issue of the testing model is the size of the receiving and sending buffers. As it is described in [3] and the section 4.4.2 of [2], there are two socket options (SO) called SO_SNDBUF and SO_RCVBUF that are used as a hint to set the underlying network I/O buffers. In other words, when used in set, these options are suggestions to the kernel from the application about the size of buffers to use for the data to be sent and received over the socket. The value of SO_RCVBUF is also used to set the TCP receive window that is advertized to the remote peer. In the case of this testing model, we used the default values of 8192 bytes for both options. All other Java SO parameters of TCP were assigned the default values as well. Each client as well as the server has SNDBUDF and RCVBUF TCP buffers.

There are two additional buffers besides the TCP buffers which are used for reading from the input stream and writing to the output stream of the socket. If these stream buffers (application buffers) are assigned sizes which are less than the default ones for the TCP socket buffers, then the performance of this connection definitely goes down. Therefore, in our experiments, we are using the same size and the sizes which are two, three, four, eight, and sixteen times greater than the default socket ones. The same issue concerns the equality of the stream buffers for the server and the client. If these buffers are not equal, then the performance goes down. Therefore, the stream buffers of the server and the client are always of equal sizes in our tests.

To make the tests straightforward and simplify our performance comparison from the results, we always send 100 MB ($100 * 2^{20}$ bytes) of data between a single thread of client and the server. So, if there are several client threads that are connected to the server, each thread will send 100 MB of data, putting load on the server. Because the clients all run on the same machine, increasing the number of threads might adversely affect performance due to threading overhead costs.

The approach described above allows us to adequately measure the performance of blocking and non-blocking sockets in I/O intensive applications, and allows for their evaluation.

IV. SYSTEM MODEL

As one of the goals of this paper is to compare the performance of blocking and non-blocking sockets in single-core and multi-core systems, we use computers with single-core, two-core and four-core Intel processors employing the testing model described above. The experience shows that there are no fair results for the tests between the computers with different amount of cores, so we run our tests only between peer computers, i.e. the computers with the same number of cores. The operating system installed in these computers is Windows XP with Java 1.6 platform. It is important to notice that Windows Vista has a new networking stack named “Next Generation TCP/IP Stack” which makes large improvements in all areas of network-related functionality. Consequently the tests may run differently on Windows XP and Vista producing different results.

The LAN used for the testing is built using 1 Gbit/s Fast Ethernet switch. This LAN is the same for all tests and is not loaded with any other traffic. In other words, at the moment of testing there are only two active computers where the tests are executed.

V. PERFORMANCE ANALYSIS

The parameters for the performance analysis of our testing model are the size of the stream buffers and the number of threads. Our tables reflect these parameters: columns show the number of threads and rows display the sizes of the stream buffers in bytes. The result of each test is the time in milliseconds needed to complete the test using the input parameters. The ms time is presented in the intersection of columns and rows. There are two kinds of times: the time to complete one client thread within a test and the time needed for the whole test to be completed (for all clients). The first kind is presented in tables on top and the second one is presented below in parentheses. Both values are a result of running multiple equivalent tests, and taking their average. Of course, both times are the same if there is only one client thread in the test.

In all, there are totally six tables with results: Table I - VI. Each table is defined for the particular type of connection (blocking server and client, and non-blocking server and client) and the number of cores in the systems executing the tests.

TABLE I
BLOCKING-BLOCKING CONNECTION FOR THE 1-CORE ARCHITECTURE

Buffer \ Threads	1	4	8	16	32	64
8192	5346 (5346)	2999 (11996)	2117 (16940)	1832 (29317)	1830 (58578)	1819 (116458)
16384	3687 (3687)	2148 (8592)	1556 (12453)	1533 (24541)	1498 (47950)	1507 (96462)
24576	3174 (3174)	1692 (6768)	1363 (10911)	1318 (21094)	1364 (43672)	1395 (89281)
32768	3024 (3024)	1414 (5659)	1257 (10062)	1329 (21265)	1303 (41698)	1343 (86010)
49152	2675 (2675)	1297 (5190)	1223 (9787)	1276 (20422)	1236 (39568)	1253 (80224)
65536	2703 (2703)	1403 (5615)	1237 (9902)	1280 (20489)	1281 (40999)	1313 (84043)
131072	2784 (2784)	1441 (5765)	1385 (11084)	1380 (22088)	1399 (44786)	1487 (95177)
262144	DL	DL	DL	DL	DL	DL

TABLE II
NON-BLOCKING-NON-BLOCKING CONNECTION FOR THE 1-CORE ARCHITECTURE

Buffer \ Threads	1	4	8	16	32	64
8192	7356 (7356)	3346 (13384)	2664 (21319)	2589 (41437)	2483 (79468)	2648 (169510)
8704	6928 (6928)	3551 (14206)	2576 (20614)	2474 (39593)	2395 (76656)	2519 (161216)
8768	UP	UP	UP	UP	UP	UP

TABLE III
BLOCKING-BLOCKING CONNECTION FOR THE 2-CORE ARCHITECTURE

Buffer \ Threads	1	4	8	16	32	64
8192	5692 (5692)	1932 (7729)	1515 (12124)	1240 (19844)	1091 (34916)	1321 (84578)
16384	3972 (3972)	1583 (6334)	1112 (8901)	970 (15523)	950 (30414)	1135 (72648)
24576	3302 (3302)	1371 (5484)	976 (7812)	912 (14594)	911 (29172)	1035 (66271)
32768	2890 (2890)	1266 (5067)	928 (7427)	913 (14614)	910 (29151)	1005 (64380)
49152	2640 (2640)	1121 (4484)	917 (7343)	905 (14495)	905 (28974)	903 (57802)
65536	2682 (2682)	1265 (5062)	1061 (8494)	913 (14620)	902 (28875)	900 (57648)
131072	1958 (1958)	1339 (5359)	1083 (8666)	1042 (16687)	1082 (34625)	1155 (73922)
262144	DL	DL	DL	DL	DL	DL

TABLE IV
NON-BLOCKING-NON-BLOCKING CONNECTION FOR THE 2-CORE ARCHITECTURE

Buffer \ Threads	1	4	8	16	32	64
8192	7524 (7524)	2735 (10942)	1985 (15880)	1712 (27401)	1637 (52400)	1713 (109648)
8704	6156 (6156)	2957 (11828)	1867 (14937)	1647 (26359)	1556 (49817)	1623 (103916)
8768	UP	UP	UP	UP	UP	UP

TABLE V
BLOCKING-BLOCKING CONNECTION FOR THE 4-CORE ARCHITECTURE

Buffer \ Threads	1	4	8	16	32	64
8192	5228 (5228)	2589 (10357)	1622 (12979)	1244 (19912)	1120 (35854)	1098 (70291)
16384	4251 (4251)	1492 (5971)	1231 (9854)	1077 (17237)	1065 (34086)	1068 (68364)
24576	4103 (4103)	1393 (5573)	1104 (8839)	1030 (16492)	1032 (33052)	1031 (66042)
32768	3345 (3345)	1276 (5106)	1045 (8362)	1038 (16618)	1034 (33119)	1029 (65885)
49152	2784 (2784)	1150 (4603)	1036 (8291)	1025 (16406)	1022 (32734)	1015 (64973)
65536	2665 (2665)	1442 (5768)	1405 (11243)	1187 (18997)	1205 (38578)	1156 (74015)
131072	2337 (2337)	1444 (5778)	1274 (10197)	1102 (17646)	1034 (33093)	1039 (66515)
262144	2234 (2234)	DL	DL	DL	DL	DL

TABLE VI
NON-BLOCKING-NON-BLOCKING CONNECTION FOR THE 4-CORE ARCHITECTURE

Buffer \ Threads	1	4	8	16	32	64
8192	5385 (5385)	2890 (11562)	1833 (14667)	1227 (19637)	1083 (34672)	1077 (68989)
8704	5089 (5089)	2146 (8587)	1538 (12306)	1213 (19421)	1082 (34636)	1077 (68963)
8768	UP	UP	UP	UP	UP	UP

Increasing the number of buffers may potentially create a deadlock what is denoted as DL in the tables. The nature of these deadlocks is described in the next section.

There were really many tests done for the NB-NB connections using even different pairs of computers with the same architecture but different speed to make the same kinds of tests. The tests showed the results that are very different from each other because some of them showed increase of speed, some showed decrease of speed, and some caused deadlocks. For this reason the results of these tests are not provided but denoted as UP what means unpredicted. In other words, the buffer size that is marked as UP and all greater buffers do not provide stable results and should not be considered.

VI. Deadlocks

In chapter 6 of [2], Calvert clearly describes the reasons for these deadlocks. They occur because the TCP sending and receiving buffers are finite, so when they fill up, the *TCP flow control mechanism* may cause the deadlock. The next paragraph contains a simple explanation taken from [2].

Let's assume the sizes of TCP *sending and receiving buffers* are *SQS* and *RQS*, respectively. A write() call with a *stream buffer* of size n such that $n > SQS$ will not return until at least $n - SQS$ bytes have been transferred to the receiving buffer at the receiving host. If n exceeds $(SQS + RQS)$, write() cannot return until after the receiving program has read at least $n - (SQS + RQS)$ bytes from the input stream. If the receiving program does not call read(), a large send() may not complete successfully. In particular, if both ends of the connection invoke their respective output streams' write() method simultaneously with buffers greater than $SQS + RQS$, deadlock will result: neither write will ever complete, and both programs will remain blocked forever.

As it was explained in section III, SO_SNDBUF and SO_RCVBUF are only the hints for the operating system to be used as the TCP buffer sizes. In practice, these sizes may be chosen and changed by the operating system dynamically. Of course, the policy of their changing heavily depends on the operating system. In our tests, we can assume that Windows was increasing the TCP buffer sizes suitable for the stream buffers of up to 65536 but stopped at some size later, so we already had deadlocks at 131072. Another important issue is that perhaps the buffers' sizes depend on the number of the opened TCP sockets within the operating system. For example, it is proved by the fact that we successfully completed single-threaded B-B test for 131072 but failed for a greater number of threads.

The deadlocks occur in B-B and NB-NB tests as well, however with greater stream buffers in the case of NB-NB. It's very easy to understand why they happen in B-B tests but rather obscured in NB-NB tests. The real issue that happens is that the deadlocked selection key never appears in the selector but really was not blocked on the last operation. We suspect that this fact is explained by the same issue as the case of B-B but on the underlying level of NIO.

VII. Conclusion

The performance analysis shows that blocking sockets are commonly faster for a relatively small number of threads. However, the benefit of non-blocking sockets becomes clear as the number of clients increases substantially. When that occurs, the sheer number of service threads and their associated management makes the server process spend proportionally more time managing the thread CPU and memory-switching than actually serving clients. From Yu [4] we conclude that I/O-intensive applications, such as ours, see performance benefits at lower client counts than service-oriented applications such as a web server. For example, we can refer to [4] where the throughputs of two web servers were compared: Tomcat (with blocking sockets) and Glassfish (with non-blocking sockets). Considering the Figure 3 that is taken from [4], the non-blocking socket starts to show a better throughput than the blocking one for the number of clients greater than 4,000 using a 4-CPU server. This figure illustrates when non-blocking sockets might perform better. These results may be different for various types of client-server connections. For instance, our testing model shows close results for as few as 64 simultaneous clients. Specifically, the 4-core architecture shows almost the same results.

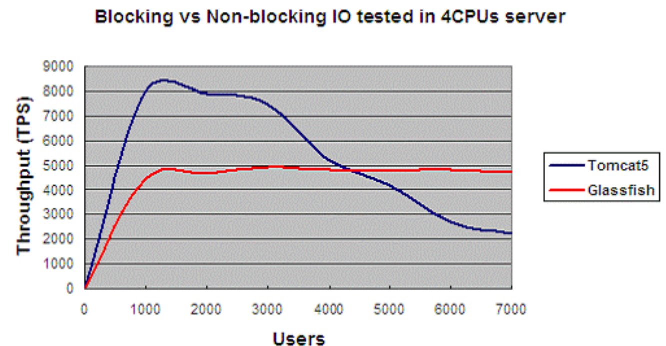

Figure 3. Throughput in a 4-CPU Server.

The reason the non-blocking sockets start to show better results for much smaller number of simultaneous clients than in [4], is because our testing model is not a web-server but rather a socket-I/O intensive streaming application. There is no file access or server-side processing as typical in web server applications.

Another important conclusion that can be drawn from the analysis is that non-blocking sockets always work correctly with the same stream buffer size as the default send TCP buffer size, which is 8192 (8K) bytes in Windows. If the stream buffer is greater than the TCP buffer size, then the speed increase is not guaranteed and behavior of the socket becomes unpredictable.

This paper is not intended to show the quantitative results of the performance for the particular architecture of central processors. While each pair of 1, 2, and 4-core machines was alike, CPU speeds were different across the core counts. Therefore all results should be taken proportionally to the presented results when applied to other processors with the same amount of cores. Speeds will increase with faster CPUs, but the time ratios should remain consistent.

The charts at Fig. 4 and Fig. 5 show speedup *per thread* versus a *normalized* 1.0 time for a single thread time. All lines begin at 1.0 as it is the base time for a single thread. As the number of threads goes up, the time per thread gets better, as the system increases its throughput.

Figure 4 shows the number of threads that increases from 1 to 64. The NB sockets time per thread consistently shows improvement against a single thread time. Specifically the 4-core system improves slowest, but ultimately shows best improvement as the client thread-count increases.

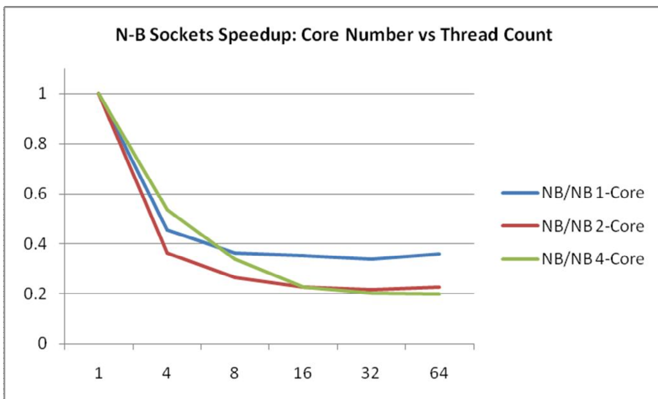

Figure 4. Non-blocking Sockets Speedup.

Figure 5 shows that blocking sockets have most initial gain in a 2-Core system, but the 4-Core per thread time is best as the number of threads grows to 64. It appears that higher thread counts will continue to improve until the server-side service threads' overhead begins to dominate the servers efficiency.

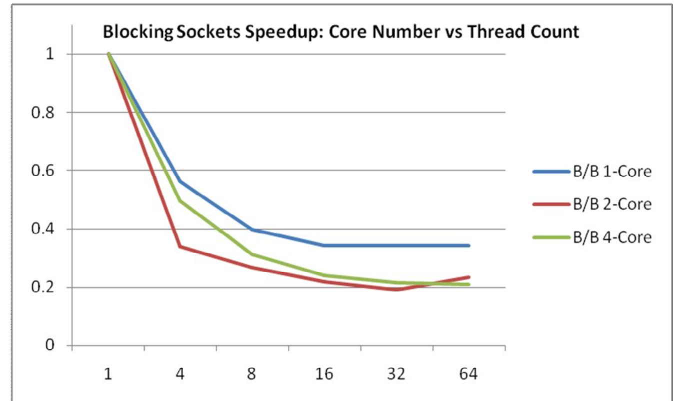

Figure 5. Blocking Sockets Speedup.

VIII. Future Research

In the future, it is reasonable to further study the benefits of non-blocking sockets in applications with a very large number of clients, by distributing the clients across many machines. The best model for these tests would be a web server-type environment even in I/O-intensive applications.

In addition, Oracle is planning to present the extension to the New I/O API called More New I/O API (or NIO.2) in JDK7 under the specification of JSR-203. As it is told in [8], this API will be a major feature of JDK7 what makes the future tests more attractive and presage a higher performance for non-blocking sockets in Java.

IX. References

[1] Brian "Beej" Hall, "Beej's Guide to Network Programming", 1995 – 2001.
[2] Kenneth L. Calvert and Michael J. Donahoo, "TCP/IP Sockets in Java: Practical Guide for Programmers", Second Edition, 2008.
[3] Sun Microsystems, "Java 2 Platform Std. Ed. v1.4.2", The Java Socket API Documentation (Socket.html), 2003.
[4] Wang Yu, "Scaling Your Java EE Applications", Sun Microsystems, 2008.
[5] Aruna Kalagnanam, Balu G., "Merlin brings non-blocking I/O to the Java platform", IBM, 2002.
[6] Lydia Parziale, David T. Britt, Chuck Davis, Jason Forrester, Wei Liu, Carolyn Matthews and Nicolas Rosselot, "TCP/IP Tutorial and Technical Overview", IBM Redbooks, December 2006.
[7] Catalyst Development Corporation, "An Introduction to TCP/IP Programming", 1997.
[8] Janice J. Heiss and Sharon Zakhour. "The Java NIO.2 API in JDK7", May 2009.