Low-level UDP network programming performance comparison on Linux

Viktor Devecseri
Alba Regia Technical Faculty
Óbuda University
Budapest, Hungary
devi86@gmail.com

***Abstract*—This article presents some low-level UDP based network programming methods and compares them by measuring the number of transmitted messages per second on x64 and POWER9 servers. The examined methods are: a naive implementation with busy waiting variants; single- and multi-threaded synchronous and single-threaded asynchronous implementations using Boost.Asio; and an implementation using the `sendmmsg()` and `recvmmsg()` system calls for sending multiple messages at once.**

***Index Terms*—udp, user datagram protocol, socket, performance, comparison, linux, busy waiting, boost.asio, asio, sendmmsg, recvmmsg**

I. Introduction

The User Datagram Protocol (UDP) is a connectionless communication model with no guarantee of delivery, ordering of the messages, or that the messages arrive only once [1]. It is widely used, e.g. the Domain Name System (DNS) builds upon it, multi-player games and other real-time applications like streaming use it where low latency is important. Since UDP doesn't include mechanisms for handling packet loss as Transmission Control Protocol (TCP) does, this scenario must be handled by the application, if needed.

In this article different low-level network programming methods are presented for UDP sockets and their impact on the total transmitted messages per second. UDP sockets were used to avoid the additional logic and messaging built into the TCP protocol. The methods presented here were mainly inspired by [2] and [3]. The *echo protocol* was implemented in C++, where the server immediately sends back the received data to the client without any processing. After the client receives the reply from the server, it sends the next message and waits for the response. For comparison the total number of sent and received packets were measured.

Section III introduces the methods used for messaging. Section IV provides an in-depth comparison of the methods with different message sizes. There are two scenarios: (1) one client with one server, (2) four clients with one server. In the tests the client and server processes were running on different physical servers.

II. Testing environment

A. Hardware and Operating System

The measurements were done in a lab environment with multiple x64 and POWER9 based servers with CentOS 7 operating system. The same measurements were carried out on both architectures without mixing them to have a clear comparison between the architectures. The following type of servers were used:

- IBM Power System LC922 (9006-22P) – *ppc64le*: dual IBM POWER9 02CY228 (2.7 GHz, turbo: 3.8 GHz) CPU; 512 GiB DDR4 RAM; Mellanox ConnectX-5 100 Gbit/s NIC.
- Supermicro SSG-6048R-E1CR60L – *x86_64*: dual Intel Xeon E5-2683 v4 (2.1 GHz, turbo: 3.0 GHz) CPU; 512 GiB DDR4 RAM; Mellanox Connect-X 4 100 Gbit/s NIC.

Note that on the POWER9 servers all-in-all 160 threads could run in parallel (2 CPUs · 20 cores/CPU · 4 threads/core), whereas on the x64 servers 64 threads (2 CPUs · 16 cores/CPU · 2 threads/core).

B. Software

The test programs were compiled on both architectures with *gcc 7.3.1* with the *-O2* flag and *boost 1.74.0* [4] is used in some tests.

The client applications count the sent and received number of messages and report them every second for one minute duration.

III. Methods

This section introduces the different networking methods and compares them on both the x64 and POWER9 servers. In these measurements there was one client and one server, and the message size was 10 bytes.

A. Baseline

To have a comparison baseline for further measurements the textbook example was created using the BSD Socket API (`recvfrom()`, `sendto()`) [1]. The server synchronously receives from a single socket and sends back the received data to the client. The client works similarly but first it sends the data to the server and then waits for the reply. Even though this article is about UDP performance comparison, this test was carried out for TCP too.

On Fig.1 the results are presented. The POWER9 servers achieved a bit better result with 57 160.4 MPS (round trip time: 17 μs) than the x64 servers with 52 941.3 MPS (round

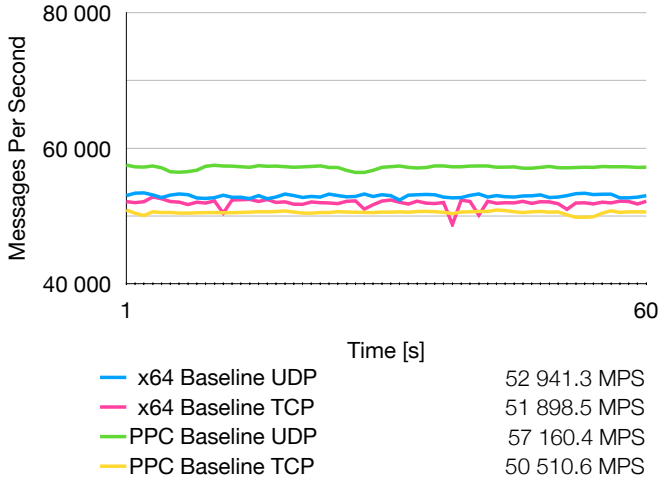

Fig. 1. Comparison of the TCP and UDP baseline measurements on the x64 and POWER9 servers over one minute with 10 bytes long messages.

trip time: 19 μs). For TCP the order is reversed, i.e. the x64 servers perform better than the POWER9 servers. Note, that in this scenario TCP is just slightly slower than UDP.

B. Busy waiting

By default `recvfrom()` blocks when there are no messages available for the application and the process is put to a sleeping state by the kernel and only waken up when there is a new message [1]. As [3] suggests the context switches could be avoided by busy waiting, where the socket is continuously polled by passing the `MSG_DONTWAIT` flag to the `recvfrom()` call. This modification was applied both to the server and to the client which increased the messaging rate by about 35-40% (Fig.1) at the cost of increased CPU usage.

The kernel's scheduler assigns the threads to a logical CPU to run it, but the threads could be rescheduled during runtime. To avoid this change, both the server and client were pinned to the 0^{th} logical CPU. In case of the POWER9 servers this didn't change the results significantly, but the variance decreased. However, the performance dropped on the x64 servers by about 14% (Fig.2).

C. Boost.Asio

Boost.Asio is a cross-platform C++ library for network and low-level I/O programming with both synchronous and asynchronous programming models. Boost.Asio also provides the interfaces and functionality specified by the "C++ Extensions for Networking" Technical Specification [4].

1) Synchronous API: The Boost.Asio library includes a low-level socket interface based on the BSD Socket API [4]. Since it is a thin layer over the API used in the baseline measurement (Section III-A), similar results were measured (Fig.3) with about 3.5% decrease (which could be due to different network utilization by other users).

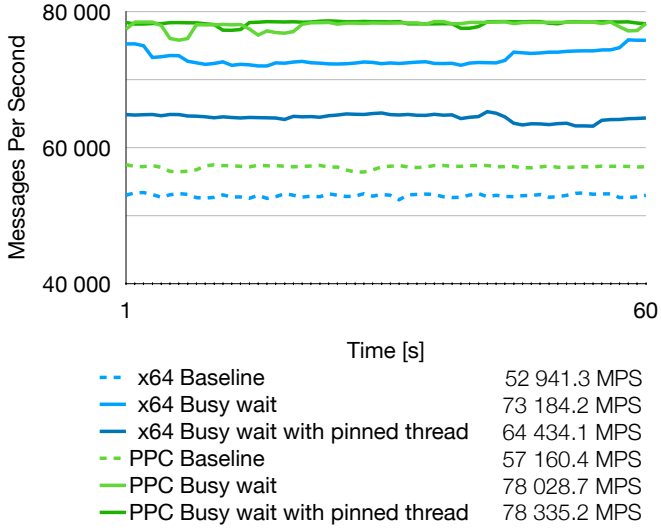

Fig. 2. Comparison of the busy waiting measurements on the x64 and POWER9 servers over one minute with 10 bytes long messages.

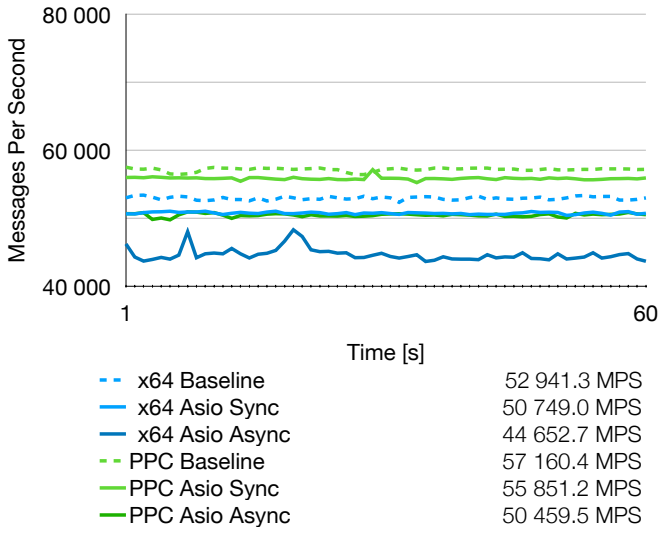

Fig. 3. Comparison of the Boost.Asio synchronous and asynchronous APIs based measurements on the x64 and POWER9 servers over one minute with 10 bytes long messages.

2) Asynchronous API: The Boost.Asio library also provides asynchronous support based on the Proactor design pattern [4]. The same *echo protocol* was implemented with this API. The server starts an asynchronous receive operation, and when it completes successfully an asynchronous send operation is started. When the send operation is finished a new asynchronous receive operation is started. It performs about 15% worse compared to the synchronous APIs (Fig.3).

Since the asynchronous model requires more housekeeping (both in the application and in the kernel) it is not worth it for a single socket. However, it provides a clean and event driven programming model which allows to handle many concurrent sockets on a single thread. It could easily be scaled to multiple threads, which could be beneficial especially when

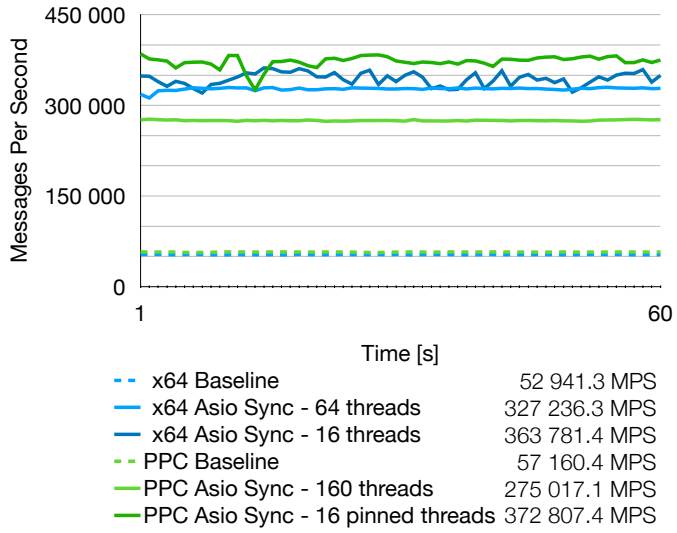

Fig. 4. Comparison of the multi-threaded measurements on the x64 and POWER9 servers over one minute with 10 bytes long messages.

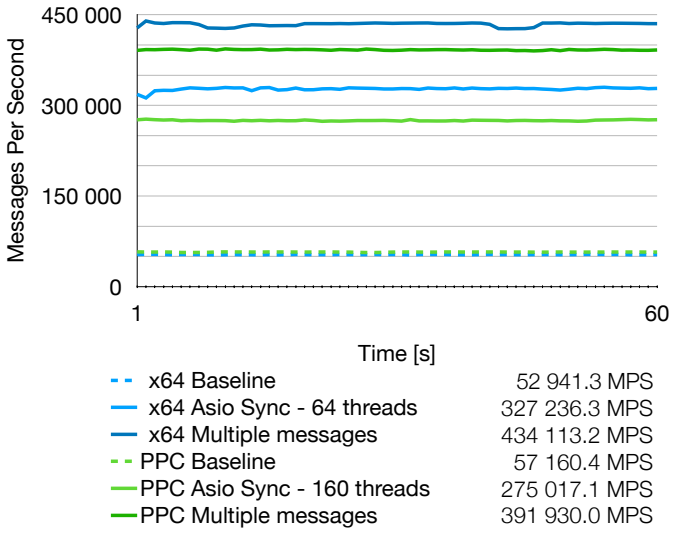

Fig. 5. Comparison of the multi-threaded and the multiple message APIs based measurements on the x64 and POWER9 servers over one minute with 10 bytes long messages.

the completion handlers of the asynchronous operations do more processing.

D. Multi-threading

A single socket was used from multiple threads via the Boost.Asio synchronous API (as in section III-C1). The implementation was done in a simple way, so it is possible that in the client a different thread will receive the response of another thread's message. However, all threads send the same message and this way the underlying network stack's raw performance was measured.

First as many threads are started as the number of logical CPUs, i.e. 160 on the POWER9, and 64 on the x64 servers. The x64 servers outperformed the POWER9 servers (Fig.4) by about ∼50 000 MPS which is surprising because in the baseline test the POWER9 servers performed better.

Since many threads access the same socket, the kernel spends more time waiting for the internal locks belonging to the socket. When the number of threads were limited to 16 the performance increased (328 419.1 MPS for POWER9 and 363 781.4 MPS for x64). When these threads are pinned to separate logical CPUs, the performance increased for the POWER9 servers to 372 807.4 MPS, but decreased for the x64 servers to 343 920.6 MPS (as seen in section III-B). On Fig.4 the better results are shown. Note that the variance increased significantly with the 16 threads compared to the maximum number of concurrent threads supported by the CPU.

E. Multiple messages

Since Linux 3.0 with the `sendmmsg()` and `recvmmsg()` system calls it is possible to send multiple messages on a socket to different endpoints with a single system call [5] [6].

Sending and receiving 64 messages with a single system call outperforms the multi-threaded approach (section III-D) with less CPU utilization. The x64 servers go beyond 400 000 MPS and the POWER9 servers approaches it (Fig.5).

IV. Comparison

Fig.6 provides an overall comparison of the previously described methods with different payload sizes (10, 100, 1 000, 2 000 bytes) with one client and one server measured on the POWER9 servers.

Since the Ethernet MTU (Maximum Transmission Unit) was set to 1 500 bytes on the servers used in the measurements, the larger messages (i.e. 2 000 bytes) were fragmented into multiple IP packets. The effect of this is clearly noticeable in the measurements too (Fig.6).

Fig.7 shows the results when four clients message one server at once measured on the POWER9 servers. For the *baseline*, *busy waiting*, *Asio Sync* and *Asio Async* methods the server's transmission rate scales almost linearly with some drop per client compared to the single client case.

However, for the *multi-threaded* and *multiple messages* tests the messaging rate of the clients are not distributed evenly. Furthermore, in some cases of the *multiple messages* test one or two clients' messages were dropped and didn't reach the test applications.

Based on the tests on the POWER9 servers the upper limit of a socket's receive rate seems to be around 400 000 MPS by default.

V. Conclusion

In this article some networking methods were presented and their effect on the performance measured by the total transmitted Messages Per Second (MPS) on a single UDP socket. It was observed that the method used for communication has a large impact on the overall performance. Starting with ∼50 000 MPS as the *baseline*, up to ∼400 000 MPS could be achieved with the *multiple messages* method. However, there is a place for future improvements too.

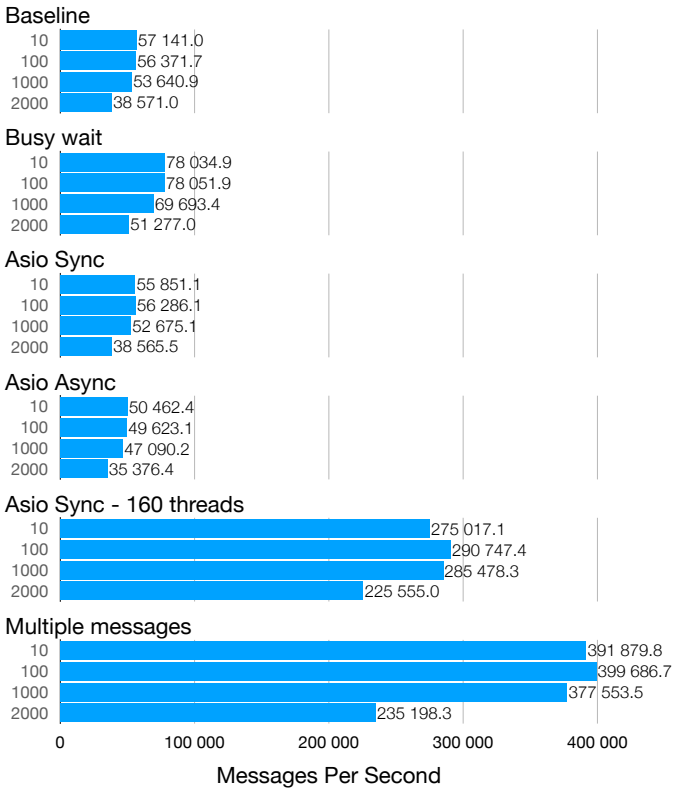

Fig. 6. Comparison of different message sizes with different methods on the POWER9 servers with one client and one server.

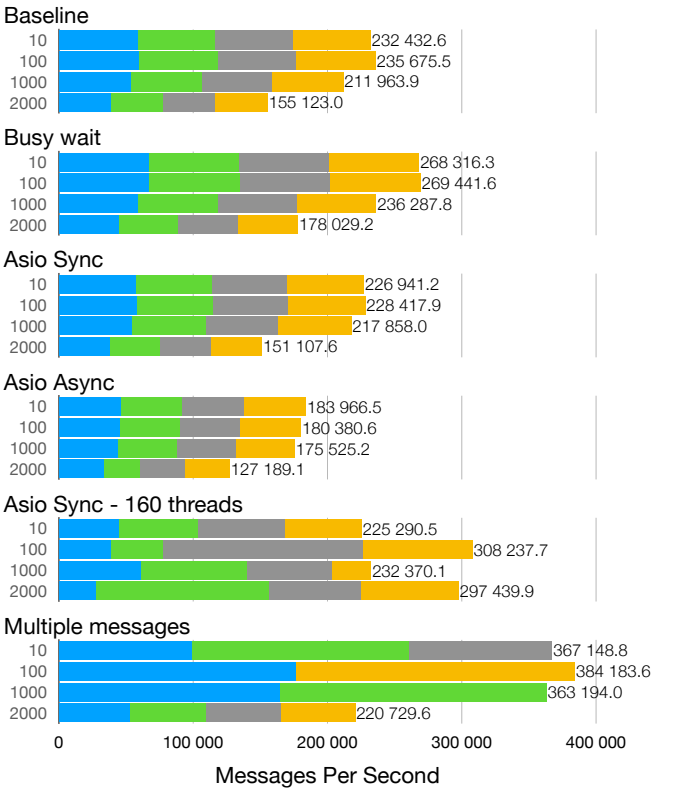

Fig. 7. Comparison of different message sizes with different methods on the POWER9 servers with four clients and one server. Each color represents the messaging rate of a POWER9 client. Note, that for the last two methods they are not evenly distributed.

A. *Future work*

The tests except the *multi-threaded* and *multiple messages* examined synchronous communication, i.e. the client waited for a reply before sending a new message. It should worth investigating having multiple pending messages, i.e. the client sends N messages at once, and only sends a new one when it receives a reply. This way there would always be N pending messages on the network. Comparing a manual implementation with the *multi-threaded* and *multiple messages* case would be interesting.

In the measurements carried out there was only one socket in the client, and one socket in the server. By having multiple sockets, the overall performance could be improved significantly [2]. By using the `SO_REUSEPORT` socket option multiple sockets could be bound to the same address and port, i.e. the clients don't need to address different ports. By having a clear understanding how the networking stack handles the packets [2] [3] [7] and fine tuning some kernel and NIC specific parameters, on a 10 Gbit/s network 7 010 000 Packets per Second (9 310 Gbit/s) could be achieved [8].

Currently the measures were only carried out with C++ implementations, however it would be interesting to see the impact of other languages (e.g. Java, C#, Python) as well.

Up to this point the networking was done via the kernel, however by having a user space network stack it could be omitted. This could give higher performance for some specific applications (e.g. in high performance trading) [9] however, the network interface would be exclusively used by only one application [10].

References

[1] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming, Vol. 1*, 3rd ed. Pearson Education, 2003.

[2] M. Majkowski. (2015, Jun. 16) How to receive a million packets per second. Cloudflare. [Online]. Available: https://blog.cloudflare.com/how-to-receive-a-million-packets/

[3] M. Majkowski. (2015, Jun. 30) How to achieve low latency with 10Gbps Ethernet. Cloudflare. [Online]. Available: https://blog.cloudflare.com/how-to-achieve-low-latency/

[4] (2020, Oct.) Boost C++ Libraries. [Online]. Available: https://www.boost.org

[5] *sendmmsg(2) - Linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man2/sendmmsg.2.html

[6] *recvmmsg(2) - Linux manual page*. [Online]. Available: https://man7.org/linux/man-pages/man2/recvmmsg.2.html

[7] T. Herbert and W. de Bruijn. Scaling in the Linux Networking Stack. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/scaling.txt

[8] T. Makita, "Boost UDP Transaction Performance," in *LinuxCon + ContainerCon Japan 2016*, Jul. 13–15, 2016. [Online]. Available: https://events.static.linuxfound.org/sites/events/files/slides/LinuxConJapan2016_makita_160712.pdf

[9] J. Evans. (2016, Jun. 30) Why do we use the Linux kernel's TCP stack? [Online]. Available: https://jvns.ca/blog/2016/06/30/why-do-we-use-the-linux-kernels-tcp-stack/

[10] M. Majkowski. (2016, Jul. 7) Why we use the Linux kernel's TCP stack. Cloudflare. [Online]. Available: https://blog.cloudflare.com/why-we-use-the-linux-kernels-tcp-stack/