Improving Networking Performance of a Linux Cluster

A. Bogdanov[1,a], V. Gaiduchok[1,2], N. Ahmed[2], P. Ivanov[2], I. Gankevich[1]

[1] Saint Petersburg State University, 7/9, Universitetskaya emb., Saint Petersburg, 199034, Russia

[2] Saint Petersburg Electrotechnical University, 5, Professora Popova st., Saint Petersburg, 197376, Russia

E-mail: [a] bogdanov@csa.ru

Networking is known to be a "bottleneck" in scientific computations on HPC clusters. It could become a problem that limits the scalability of systems with a cluster architecture. And that problem is a worldwide one since clusters are used almost everywhere. Expensive clusters usually have some custom networks. Such systems imply expensive and powerful hardware, custom protocols, proprietary operating systems. But the vast majority of up-to-date systems use conventional hardware, protocols and operating systems. For example, Ethernet network with OS Linux on cluster nodes. This article is devoted to the problems of small and medium clusters that are often used in universities. We will focus on Ethernet clusters with OS Linux. This topic will be discussed by an example of implementing a custom protocol. TCP/IP stack is used very often, it is used on clusters too. While it was originally developed for the global network and could impose unnecessary overheads when it is used on a small cluster with reliable network. We will discuss different aspects of Linux networking stack (e.g. NAPI) and modern hardware (e.g. GSO and GRO); compare performance of TCP, UDP, custom protocol implemented with raw sockets and as a kernel module; discuss possible optimizations. As a result several recommendations on improving networking performance of Linux clusters will be given. Our main goal is to point possible optimization of the software since one could change the software with ease, and that could lead to performance improvements.

Keywords: computational clusters, Linux, networking, networking protocols, kernel, sockets, NAPI, GSO, GRO.

This work was supported by Russian Foundation for Basic Research (projects N 16-07-01111, 16-07-00886, 16-07-01113) and St. Petersburg State University (project N 0.37.155.2014).

Possible optimizations

First of all, since the subject of this article is a network of a small computational clusters, let us highlight basic features of such network in comparison with generic one (probably global). As opposed to the generic network, computational cluster network has the following features: it is very reliable; all nodes of a cluster are usually the same (they use the same protocols, works at the same speed, etc.); congestion control could be simplified; number of nodes is known and usually does not change; no need in sophisticated routing subsystem (even more, cluster nodes usually belongs to the same VLAN); quality of service questions could be simpler (clusters are usually dedicated to several well-known applications with predictable requirements; there are usually only a small number of users in case of small cluster); fewer security issues since administrator of a cluster manages all the nodes of the network (even more, cluster nodes are usually not accessible from the outside world); overall overheads could be small. So, computational cluster network (even in case of small cluster and not very powerful network) is a special case. This case is simpler than a global network. That is why we can omit many assumptions required for a generic network (since the network is well-known and determined). That is why the algorithms and the implementation of protocols in this case could be much more simpler and efficient. All in all, when designing a protocol stack for the described case one could assume the following possible optimizations: simple addressing, simple routing, simple flow control, simple checksumming, explicit congestion notifications, simple headers (processing time is more important than header size in case of computational cluster with powerful reliable network (in other cases one e.g. could use ROHC)). All these assumptions lead to a simpler and fast implementation. Actually, estimating the possible improvements (in terms of performance) is the main purpose of this article.

Overview of Linux networking stack

In our case when some user application wants to start some communication via conventional network it creates a socket. When that application wants to send some data via network it finally does ‘send’ or similar system call. L4 protocol implementation ‘sendmsg’ function (pointed to by ‘sendmsg’ field of ‘ops’ field of ‘struct socket’) will be invoked. This function does necessary checks, finds appropriate device to send from and then creates a new buffer (‘struct sk_buff’) which represents the buffer for packet to send. It will copy the data passed by the user application to the buffer. Then it fills the protocol header as necessary (or headers in case e.g. when we use that implementation to work with transport and network layers) and tries to send the data. It usually will call ‘dev_queue_xmit’, which, in turn, will call other functions – these function will e.g. pass the packet to ‘raw sockets’ (if any presents), queue the buffer (qdisc [Almesberger, Salim, …, 1999], if used). Finally our buffer will be passed to the NIC driver which will send the passed buffer as is (at this point we should have all the necessary headers at 2-7 layers). The ‘sk_buff’ structure allows one (in general case) to copy the data only once, from user buffer to kernel buffer and then (inside the kernel) work conventionally with headers at different layers without additional data copying.

When the user application wants to receive some data it finally does ‘recv’ or similar system call. L4 protocol implementation ‘recvmsg’ function (pointed to by ‘recvmsg’ field of ‘ops’ field of ‘struct socket’) is called. This function checks the receive queue of the socket (to which the user application refers). If the queue is empty, it sleeps (if it is blocking). If there is some data for this socket, this function handles the data – it will eventually copy the data from kernel buffer to user buffer.

When packet arrives on the wire NIC will raise an interrupt. So, appropriate NIC driver function will be called. That function will form the ‘sk_buff’ buffer and call ‘netif_rx’ or similar (‘netif_receive_skb’ in case of modern distribution with NAPI) function which will check for special handling (and do it if necessary), pass packet to ‘raw sockets’ (if any) and finally call L3 protocol implementation function registered via ‘dev_add_pack’. That function searches for matching socket (taking into account appropriate headers of the received packet) and adds the packet to the socket receive queue (if any found). It could also awake a thread waiting for the data for this socket in 'recvmsg' function (using internal fields related to this socket). The described approach of receiving the data was substantially modified. If we look at the networking I/O evolution we will see polled I/O, I/O with interrupts, I/O with DMA. Then developers tries to use some interrupt mitigation techniques. And finally, NICs start to support offloading (offload some networking related work from CPU to NIC).

So, one should take into account NAPI. NAPI ('New API') is an interface that uses interrupt coalescing. In case of modern driver that uses NAPI the steps described above will be modified in the following way: on packet reception NIC driver disables the interrupts from the NIC and schedules NAPI polling; NAPI polling is invoked after some time (we hope that during that time NIC could receive new packets) and we handle the packet(s); if we cannot handle all packets (taking into account NAPI budget), we schedule another polling, otherwise we enable interrupts from NIC and so returns to the initial state. Such scheme could improve the performance since in the described approach we decrease the number of interrupts and handle several packets at once – and so decrease the overheads (e.g. time for context switching). More details about NAPI could be found in [Leitao, 2009].

Another possible option is GSO/GRO - techniques that could improve outbound/inbound throughput by offloading some amount of work to the NIC. In case of GSO/GRO NIC is responsible to do some amount of work which is usually done by OS on CPU: in case of GSO CPU prepares initial headers (at 2-4 layers) and passes them with pointer to big chunk of data (which should be sent in several packets) while NIC splits the data into several packets and fill the headers (2-4 layers) taking into account the initial template headers. In most cases such NIC supports only TCP/IP and UDP/IP. Of course, OS should support such feature of the NIC. And Linux kernel supports it. GSO/GRO could improve the networking performance since dedicated hardware device - NIC - could take big amount of work that was previously done by CPU. Actually, NAPI, GSO/GRO are not a new techniques. But one should always remember about that options since proper configuration is still required.

One could usually turn on or off GSO/GRO (e.g. via 'ethtool -K <interface> <option> <state>). Another important parameters are 'tx' and 'rx' ring buffer size (for send and for receive). These parameters could impact the performance since e.g. while receive buffer is full (we free some elements during polling) NIC will drop all subsequent packets. These limitations are usually could be changed via command line tool (e.g. ethtool - 'ethtool -G <interface> rx <number_1> tx <number_2>). Another generic option that directly impact the performance is MTU. It could be changed via command line (e.g. via ip - 'ip set <interface> mtu <value>'). It is usually good idea to set big MTU in case of computational cluster since it will reduce the overall overheads. Finally, socket send and receive queues limitations are options that one should always remember. Administrators should adjust them as necessary since small values will lead to packet drop. These options could be changed via command line (generic limitations are in '/proc/sys/net/core/{r,w}mem_{default,max}'). NAPI budget for a NIC could be changed via '/proc/sys/net/dev_weight'. Options related to queuing discipline (qdisc) are usually handled via 'tc' command line tool. Then protocol-related parameters should be configured. In case of TCP/IP they are available in /proc/sys/net/ipv4/ (e.g. tcp_mem, tcp_wmem, tcp_rmem, tcp_autocorking, tcp_congestion_control, etc.). Of course, there are some other simple ways which could be used by administrators to improve the performance – e.g. keep all node names locally in '/etc/hosts' (do not ask DNS servers) and keep ARP entries all the time a node is up since names and addresses (IP and MAC addresses) do not change frequently in case of computational cluster.

Implementing a protocol

It is well-known that people get used to systems, environments, instruments they use. And so to protocols. TCP/IP is widely used and if a Linux application requires networking in most cases it will use TCP/IP. That stack is really good and is suitable in most cases. But using conventional approaches, means and configurations is not always good idea. We want to estimate the overheads imposed by the traditional approach: usage of fully-featured TCP/IP stack implementation for applications on a computational cluster. We want to inspect the current state using modern kernel versions.

One can implement L3 protocol in Linux as a part of the kernel (kernel module or built into the kernel) or using 'raw sockets'. The mentioned 'raw sockets' provide a programmer with a way to get the full access to the networking: they allow user application to get all the packets (of all L3 protocols) that are received on the NIC and are sent from the NIC. They are implemented in net/packet directory of the Linux source code (the source could be obtained from [The Linux Kernel Archives]). The corresponding address family is 'AF_PACKET'. Despite the fact they allow programmer to implement own L3 protocol, this approach is not viable: a thread should have 'CAP_NET_RAW' capability; the application will get all the packets the system gets (and have to handle them some way – even just discard them). But 'raw sockets' are often used for debugging and monitoring. A better way (in order to get usable protocol implementation) is implementation in kernel. In order to estimate the software performance we implemented simple approach: send data using only L2 layer (Ethernet in our case). All the test nodes belonged to the same VLAN, so we used just Ethernet header to specify the sender and the receiver, and it is quite enough for this test case. For test purposes (estimate the performance in both cases) we implemented this approach using 'raw sockets' and as kernel module.

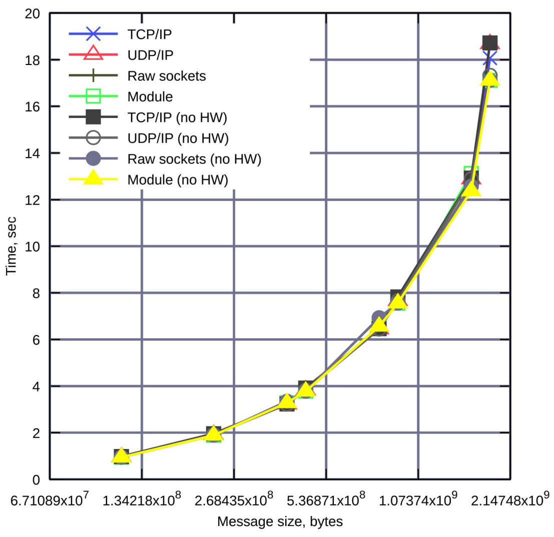
Our module initialization function calls 'proto_register', 'sock_register', 'dev_add_pack', 'register_pernet_subsys' and 'register_netdevice_notifier'. Since our test case is simple we pointed all implemented functions in 'struct proto_ops' and these functions are: 'release', 'bind', 'connect', 'getname', 'ioctl', 'sendmsg', 'recvmsg'. We use MAC addresses for addressing the nodes of the cluster. We implemented the 'bind' and 'connect' functions just for convenience (there are no connections, of course): user application could use them in order to bind to some local address (NIC with particular MAC address) and to remote address (MAC address of the NIC on the remote side). We implemented some optional optimizations in 'sendmsg', 'recvmsg' and receive function on packet reception on the NIC (registered via 'dev_add_pack'). These optimizations concern internal aspects (e.g. queuing the buffer to send). The busy-waiting in 'recvmsg' function is also possible. The implementation is SMP-aware, of course. Our target kernel version was 4.4, but we also included support for earlier kernel branches (see below). More details about Linux networking is in [Linux Foundation Wiki].

Tests

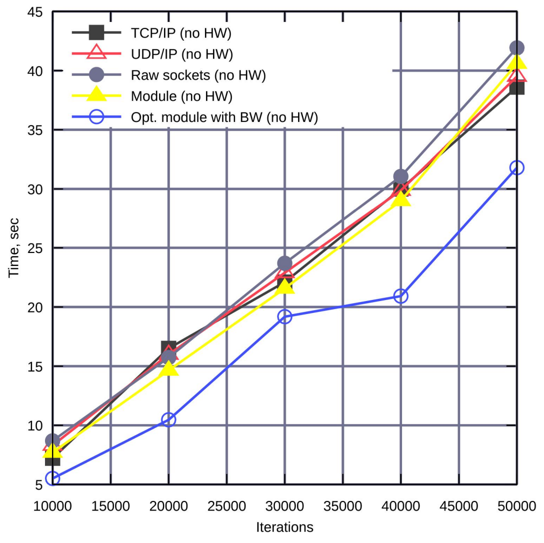
We performed tests on desktop systems (CPU (4 cores), 1Gbit Ethernet NIC (no GSO/GRO), OS Ubuntu 16.04 with kernel 4.4) and server nodes (cluster with the following nodes: CPU (8 cores), 1Gbit Ethernet NIC (with GSO/GRO for TCP/IP), OS CentOS 7 with kernel 3.19). We tested TCP/IP, UDP/IP and described approach implemented using 'raw sockets' and as kernel module. We used jumbo frames since big MTU makes sense for computational clusters. We also changed the 'tx' and 'rx' ring buffer sizes of NICs to available maximum and increased send and receive socket queues sizes (generic and for TCP/IP). Other parameters (e.g. TCP/IP parameters) were default for the described systems. We did tests on the server side with GSO/GRO capabilities turned on and off.

We performed two basic tests: file transfer and 'send-recv' (when a node transfers small amount of data to another node, waits for the answer (similar amount of data), then sends the same amount of data again and so forth). The first test is typical, while the second could show all possible problems of the software implementation: in first test we can do many parts of the work in parallel on different CPU cores, while the second test is serial since each side should wait for the full data transfer and processing before sending the answer. So, the performance of such test depends heavily on the performance of the software. The most representative results are on picture 1 and 2.

Test 1 is the file transfer test between two server nodes ('no HW' means GSO/GRO features were turned off). Test 2 is the 'send-recv' test between two desktop nodes. As one could see, there is almost no difference in case of file transfer (the desktop systems showed similar results). The 'send-recv' test showed small difference between TCP/IP and simple protocol kernel module implementation in case of server nodes. While the TCP/IP with GSO/GRO was 15% faster than TCP/IP without in that case.



Pic. 1. Test 1 results



Pic. 2. Test 2 results

While in case of desktop nodes one could see the difference: the kernel module implementation with all possible optimizations (and busy waiting - busy waiting could make sense in case when nodes dedicated to some particular application) showed the best performance. It could be described by the fact that in case of desktop nodes we have no hardware offloading and so the results depend solely on the software stack and software configuration. The 'raw sockets' implementation shows results similar to kernel module one only when there are no other networking applications running (just our test), otherwise (e.g. in case of big file transfer in parallel with our test) it shows the worst performance (because it handles all the packets). Of course, the described approach is not viable: only single application on a node could communicate, no congestion avoidance (such approach could lead to congestions in computational cluster networks). But it was used to illustrate the possibilities to improve the networking.

Conclusions

Using conventional approaches and protocols are common. Many Linux Ethernet clusters use TCP/IP for communication. TCP/IP stack is very good and suitable for many cases. But computational cluster case is special: TCP/IP was initially designed for big unpredictable networks while in case of computational cluster we have small reliable network. So, one could use simpler protocols and lightweight implementations. Of course, hardware acceleration (GSO/GRO) could improve the performance as it was showed by our tests for TCP/IP NIC offloading. But such offloading requires hardware support (which is not always available even now – e.g. on Beowulf clusters). Using simpler protocols requires no hardware changes (or lead to much more simpler hardware implementation) while it could lead to performance improvement. The simple case described here showed that there is a possibility to such improvements (when designing and implementing real L3 and L4 protocols).

References

Leitao B. H. Tuning 10Gb network cards on Linux // Proceedings of the 2009 Linux Symposium 2009.

Almesberger W., Salim J. H., Kuznetsov A. Tuning Differentiated services on linux // Globecom99 – 1999. – №. LCA-CONF-1999-019. – P. 831-836.

The Linux Kernel Archives [Electronic resource]: https://www.kernel.org

Linux Foundation Wiki [Electronic resource]: https://wiki.linuxfoundation.org/networking