Реалізовано механізм прямого пошуку з розширенням традиційного сокетного TCP-інтерфейсу для отримання повідомлень, обходячи традиційний порядок встановленої черги. Даний механізм може застосовуватись для високопродуктивних та кластерних комп'ютерних систем з метою інтенсифікації обміну даними та неперервності підтримки максимального навантаження на обчислювальні машини. Інтерфейс для прямого пошуку повідомлень реалізований на базі ядра операційної системи Linux. Отримання експериментальних результатів тестування здійснено за допомогою простого набору microbenchmark-міток. В ході тестування відправник надсилає необхідне число повідомлень сталого розміру на встановлене з'єднання, а приймач пропускає неочікувані повідомлення і зчитує очікувані повідомлення в простір користувача. Спосіб відшукання очікуваних повідомлень реалізований завдяки багаторазовому пошуку для випадку, коли сокетний додаток розглядає TCP-сокет як список повідомлень з можливістю приймати і видаляти дані не тільки з вершини, але з будь-якого місця в сокетному буфері. Всі очікувані повідомлення розпізнаються і обробляються розробленим викликом seek_recv(). Кожен тест містить ~80 повторень, які включають операції відкриття сокета, пересилання 800–1000 повідомлень відповідно до політики прийому і закриття сокета. Система використовує тільки один активний сокет в один і той же час.

Отримані результати доводять помітне зниження процесорного часу обробки повідомлень на 36–40 % та загальне зростання продуктивності. Однак, при наближенні до об'єму повідомлень 1000 байт, близького до характерного розміру корисного завантаження TCP-пакета, спостерігається падіння продуктивності процесу обміну

Ключові слова: TCP-сокети, пошук повідомлення, розширення інтерфейсу, продуктивність

UDC 004.451
DOI: 10.15587/1729-4061.2019.167995

INFLUENCE OF THE DIRECT MESSAGE SEARCH MECHANISM BASED ON THE TCP PROTOCOLS ON THE EXCHANGE PROCESS

V. Melnyk
PhD, Associate Professor*
E-mail: melnyk_v_m@yahoo.com
K. Melnyk
PhD, Associate Professor*
E-mail: ekaterinamelnik@gmail.com
S. Lavrenchuk
PhD, Associate Professor*
E-mail: swit_lanka@ukr.net
I. Burchak
PhD, Professor**
E-mail: i.burchak@lntu.edu.ua
O. Kaganiuk
PhD, Associate Professor*
E-mail: alexrakaganiuk@gmail.com
*Department of Computer Engineering and Cybersecurity***
Department on Engineer and Computer Graphics*
***Lutsk National Technical University
L'vivska str., 75, Lutsk, Ukraine, 43018

1. Introduction

High-performance cluster computing is widely used to perform long-term weighty calculations. It is well known that such calculations can be divided into separate computational parts, each of which can be performed on separate computers. To perform such computing tasks, computers have to exchange data with the appropriate frequency of their receiving. This approach is also used to obtain the integrated information related to the computing distribution on different computers, and for order of posting the appropriate data portion to implement the following computational steps. It should be added that in our time cluster components can provide an efficient environment for applications with intensive data processing on distributed platforms [1–4]. For this, the data should be organized in special structures, and all applications in such an environment should be developed to easily and skillfully operate with such data. To operate with such structures, the application is developed from a set of software components, which together with the computing resources play a leading role for flexibility and productivity optimization.

In many intensive computing applications, the volume of data can be diffracted into sub-blocks of a specified byte volume that is beneficial for conveyor processing of them. It needs to note that in practice for the organization of such computing the processing and communication may overlap in order to improve the performance of such a system. In [5], it is shown that during the transmission process small data blocks lead to the improvement of the conveying and load balancing, which takes place in the communication practice. It is also noted that larger data blocks reduce the number of messages for transmission but stimulate an increase in system bandwidth and bring inconsistencies in the load, which reduces the conveyance for data processing.

Applications written using a TCP/IP-based socket interface, along with their performance and data processing intensity, also put some other requirements such as compatibility with heterogeneous networks, as well as a guarantee of processing efficiency and scalability. To get an advantage in using high-performance protocols in cluster applications, some specially developed approaches were applied, including those involving high-performance socket levels at the user level via active protocols. These include the virtual interface architecture and IBA [5], the applications for which should be written with the account of the network productivity saving. During the individual software components reorga-

© V. Melnyk, K. Melnyk, S. Lavrenchuk, I. Burchak, O. Kaganiuk, 2019

nization, improvements in the performance of the data transfer process and computing results, as well as the increase of application scalability with adaptation to heterogeneous networks, are observed. According to the virtual interface architecture, to increase the performance significantly, some socket characteristics support efficient data splitting on the output nodes. With high performance and low overhead, sockets give an ability to achieve high-quality results for applications in many directions.

Despite the cluster structure sets the requirement for working computers to be fully loaded in order to reduce the calculation time, yet cluster systems suffer overheads during the course of the communication process. Such overheads also depend on the number of computers in the cluster, the use of libraries for the communication arrangement, the choice of an interface for inter-computer communications and communication between other units within the computing cluster. Despite a variety of experimental results, the latest researches demonstrated the interdependence of efficient library design for messaging in clusters that use Ethernet and TCP/IP components to achieve the network performance [6, 7]. Therefore, the aim is to improve the communication process, focused on the messages transferring based on TCP protocols, due to some restructuring of specific socket levels.

2. Literature review and problem statement

To solve the problem of improving the communication process efficiency, various authors focused on a dual approach. One of them took into account the rebuilding of the library design based on TCP protocols responsible for the messages exchange, and the other one pointed to hardware restructuring. Thus, the work [8] proposed the implementation of the hardware support approach using the network card interface to disconnect some critical parts during the protocols processing.

However, a large number of authors are focused on the messages transmission support by using specific socket connections via rebuilt interfaces applied at the user level or at the individual protocols levels [9–13]. Such approaches also increase the performance and productivity, reduce latency and improve the overall communication process. But they use the costly operations of data copying and do not take into account the communication directed to a particular user network application, following the order of the set socket queue.

The work [14] proposes a stream control transfer protocol (SCTP) that is a reliable transport layer protocol and similar to the TCP protocol in the mechanism of the network general stream control between two operating workstations. However, unlike TCP, such a connection can be called associative that can comprise several independent message streams in a single communication process. The SCTP ensures the messages ordering within the stream, but not across the entire stream association. Such a mechanism could be used to provide advantages for direct search sockets, depending on the specific circumstances in which the data can be processed outside the established order. For example, the MPI tag calls to receive the data can be implemented as outside-separated united streams, i.e., independent web responses on a pipelined connection. But the direct search for a scheduled message in a queue and its receipt by the user application outside the established order in the queue has not been implemented yet in these works.

However, direct search sockets are more flexible, because they allow deviations from the established order of receiving a response to a call outside the message receiving queue. In addition, the API interface for search sockets is the only extension on the receiving side compared with the traditional socket interface. The SCTP requires far more advanced extensions and modifications to manage the sending, receiving and connecting functions [15].

This communication tool is orthogonal to direct search sockets. Here it is allowed to use a few IP addresses for each computer as a part of the established stream association potentially using multiple network paths to increase the transmission process reliability.

As it was mentioned above, the overheads incurred on TCP-based message exchange depend not only on message size and transmission frequency, but also on whether the message is expected or unexpected (sudden) for the network application. The paper [16] submits that the message may arrive in a queue as unexpected if its data is received by the process-receiver before making the library call in order to receive such message to the memory buffer at the user program level. It is well known [7, 17] that unexpected data is first copied to a temporary library buffer. Such recopying operations are costly for the overall exchange process and reduce its performance.

During the message transferring based on TCP protocols, the message~~s~~ can be considered received if its data have appeared in the network and the TCP stack places it in the socket connection buffer between two communication hosts. In [16], the situation is presented when the user program expects a specific message along with unexpected messages in the queue that arrived in the system at the same time or before. For example, the system can expect a message with its *transmission interface* and a given *tag type* [18]. The message transmission interface (MTI) is like a consistent portable transmission system to implement it on a variety of parallel computing architectures.

These works also include those based on the use of a rebuilt library with a more efficient design that is based on the use of a rebuilt library with a more efficient design, driven by events from its own complementary architecture [19]. This should also include the kind of works that use special support of general hardware restructuring and network card interface, as well as those based on TCP splitting [7, 8].

The expected increase in performance comes from the specific reconstruction of the socket-level interface of the operating system. Based on the described model [16] and the practical implementation, it is completely unrelated to works that focus on the connectivity support with the use of a network card interface or used library. This work should cooperate with ideas that lead to the restructuring and improvement of other system components to enhance high-performance TCP communication.

In cluster communication practice, there are several quite effective implementations of the message interface, many of which are freely available or open to use. This gave impetus to the parallelization of software development and the development of large-scale and portable applications that are designed to perform parallel and distributed computing. During the communication process, the socket interface of the operating system for TCP protocols receives certain bytes from the connection made by using recv() or read() system calls. According to the queue order and messages alternate reception, it is first necessary to release the socket from the previous unexpected messages in order to access the expected message in the next step. Every unexpected message should be copied to the unexpected message pool by using additional copy operations that cause significant overheads of the delivery system. In this case, it is needed to additionally, before the message receiving at the application level, check the receiving pool of unexpected messages and only then call the receiving function of the expected message at the level of connection socket.

From the literature review, it is clear that the approach of direct message search in the socket stack is not sufficiently described and investigated. However, it is related to works approached to increasing the productivity of messaging exchange based on TCP protocols. The extended socket TCP interface to perform the direct access to randomly located messages within a single connection is not sufficiently explored yet. The influence of the proposed mechanism on the message exchange process and its characteristics taking into account the message parameters is also not justified.

3. The aim and objectives of the study

The aim of this work is to implement the mechanism of direct message search and its impact on the messaging process performance. This mechanism coupled with the extended socket interface for direct search can be used to ensure the conveyance of data delivery for processing in high-speed computer systems and to ensure the load stability on the computing machines.

To achieve this goal, the following objectives were set:

– to implement the simulated [16] mechanism for direct message search based on the established TCP communication between the machines and the integrated test method of simple microbenchmarks;

– to apply the extended socket TCP interface and search function to implement the approach of receiving the expected message from any stack position bypassing costly copying operations;

– to search for randomly located expected messages in a queue using multiple searches and release memory buffers after messages are received;

– using the microbenchmark testing of the system with the implemented mechanism, to reveal a reduction in CPU processing time of expected messages and the exchange performance increase depending on changes in their volume and quantity.

4. Features of Linux interaction to implement the direct message search mechanism

The socket interface to perform the direct search of expected messages on a socket can be developed and implemented on the basis of the Linux operating system kernel. The interface operation checking is carried out by using a method of microbenchmark testing, the data for which is obtained outside the request, implemented for this socket [20]. The results of qualitative calculations and assumptions [16] indicate a CPU processing time reduction for message processing using the direct search approach of the expected message in the stack. The performance growth for such a system will depend on the receiving of expected messages with larger volume or the number of messages that have to be bypassed in the receiving queue.

According to the submitted data [16] and Linux theory, the TCP stack under the Linux kernel control works with separate socket buffers, denoted in the literature as sock_buff's or skb's. Since the transmitted packets in their receiving process are accepted by the network device, the data about them are placed in the *ring buffer*, and sock_buff is assigned to the data itself, i. e. the useful part of the message. The sock_buff buffer saves the metadata for each package, and the stack of the Linux operating system for TCP/IP processes the package data by interacting with the sock_buff memory buffer, respectively. For each particular connection, the sock_buff buffer is placed in the common queue of the socket buffer. To simplify the packets acceptance in the correct order and manage them, each packet is assigned with a sequence number in the queue that specifies the number of bytes sent for each packet in particular during the connection performance. This solving method allows you to renew packets on the receiving side again by using repeatable requests at the network level and remove the received data from the socket buffers. The user's application does not know, but it should know the data order in packages or the alternation of their receipt. The tasks that are solely related to the user's application are how the data elements are sent from the source and their fixed lengths.

In [16], a block diagram of the algorithm of the functions involved in the receiving of the arrived message is presented. Specifically, the tcp_recvmsg() function is intended to copy the sock_buff's buffer messages in the socket queue, which are associated with the actual data located in the ring buffer. The function checks the first skb in the socket buffer and then copies the data to the user's space buffer. If skb has more data than is specified in the query, the tcp_recvmsg() function leaves a "reminder" in the socket buffer queue. If a user requests a larger amount of data than the one placed on the first skb, it is allocated along with the corresponding data on the ring buffer, and the described above specified steps continue with the next skb standing in the queue. In the end variant, the function tcp_recvmsg() returns all requested data from the socket buffer after the complete read-out procedure. It also removes all skb's buffers that contain the received data in full volume and updates the sequence number of the first byte to perform the next socket reading operation.

Also, TCP typically uses the sequence numbers to track the reader from the socket buffer and determines the continuation order of the readout procedure. At each step, the copied_seq variable that fixes the copy order receives the sequence number for which the data will be read. Thus, this variable will contain the complete order of what has already been copied and that will be read off from the receiving queue. If the sequence of numbers copied_seq was greater than the number of the first base skb sequence and some of them have already been read from it, then the full length of the data request will be copied starting from the sequence number specified in the copied_seq variable. So, by using sequence numbers, data determinations on the socket that the receiving request should read will be made.

5. Description and implementation of an additional search procedure for sockets

The main purpose to use sockets that are searching for a message in a queue is to receive the data about them that are placed on the socket reception buffer in random order. When the data is copied from the socket, the corresponding skbs with the copied data should be removed from the overall socket buffer, and the current skbs list should also be reduced by their number. Because the data that was read and removed under the corresponding sequence numbers is no longer available and is not accessed for any operations, then the subsequent requests on the socket should "know" and take into account that these data no longer exist in the buffer.

This approach is implemented through a connections list that contains the initial and final sequence numbers for each already selected data «hole» in the socket reception buffer. Creating a hole frees up memory space in the socket buffer and allows to normalize the TCP flow behavior regardless of the location in the buffer from which the data was read and deleted. When the request to receive the message begins the copying process of data into the user's space, it passes on its way any hole that is met and continues to normally receive data from the sequence number that goes after the hole. In the data receiving process with the user program, the list of holes usually increases. Then they are merging, and at the same time the dynamic buffer reduction associated with the holes removal takes place.

The implementation developed in this work requires the creation of a new stream protocol SOCK_SEEK_STREAM that uses the same stack as TCP and ordinary SOCK_STREAM sockets. However, their basic functions are to be modified, so that they can directly search the expected messages by using SOCK_SEEK_STREAM sockets.

If a socket request performed on the socket does not provide the direct search procedure of the expected message, or if the call doesn't find the required packet to be received, then its way through the TCP stack and used functions is almost identical to the code used by the traditional Linux kernel. When the search request is performed on a socket and requires the direct search of any expected message, then its way through the TCP stack remains the same, but the code track through separate socket functions may change. Basically, the changes will concern the code of the function tcp_recvmsg() that was entered into it. All additionally involved modifications are also required for managing the list of holes, sequence numbers and skb's that have already been read.

Another significant change should be made in the function tcp_recvmsg() that refers to the socket receiving procedure, which directly searches for the expected message. Such a change is to disable the TCP preload mechanism on the queue [16]. Despite the pre-loading mechanism of the TCP queue allows to manage better the stream resources during the message exchange process, it also causes a slight decrease in performance. On the other hand, it is not possible to change easily the download queue in the general messaging procedure in order to activate the process of further search. In connection with all submitted above, the pre-loading mechanism should be turned off at the moment of socket receiving when the socket performs a direct search of the expected message.

After the SOCK_SEEK_STREAM socket has been created, the well-known functions recv() and recvmsg() can be used as ordinary functions. The newly developed seek_recv() function can be implemented as a system call described below, which should provide the following arguments:

```
ssize_t seek_recv(int s, void *buf, size t len,
int flags, size_t offset);
```

As can be seen from the usage of the functions for ordinary sockets, the arguments for calling the seek_recv() function will be the same as for the function recv(), but with some change added to indicate the number of bytes to be transmitted into the stream. This offset always should point to the first byte that should be obtained through the use of the system call recv().

Since the call seek_recv()changes the msghdr structure and then calls the general function sock_recvmsg(), then the recvmsg()function from the standard library can be used to make the search for recipients. The msg_seek variable has been added to the msghdr structure in order to specify the search offset. With modifying the structure msghdr passed to the function recvmsg() as a parameter, it is easier to make a packet search that is expected to be received without involving a new special function.

To be able to make a search for previous messages of a large size, it is needed to increase the maximum size of the receiving buffer that is controlled through the system variable net.core.rmem_max and the sysctl parameter. Because of the setsockopt() function introduction, the variable net.core.rmem_max allows you to set the maximum buffer size for each accepted message that can be reset. Therefore, the sysctl parameter should be set accordingly for a sufficient amount of bytes and the receiving buffer should be increased, if necessary, throughout the entire user program interval.

In the case when the receiving buffer is completely filled in and the usual TCP operations are executed, then the call to receive the message search during its execution returns an error. And in case when new packages are received and the socket buffer is already full, they will not be accepted but resent again by the sender in accordance with the normal TCP flow control [21]. In this case, to release more space in the kernel, the application should clearly react to the socket buffer overflow error and delete some unnecessary data that could remain in the buffer.

6. Research results of the direct search mechanism for the messages on the basis of TCP protocols

According to model assumptions [16], the mechanism of message direct search was implemented on hosts with the established connection that allowed sending and receiving the messages. The main task in this mechanism was to implement a socket interface designed from the traditional one by rebuilding the usual socket functions and calls. This interface would allow you also to recognize the expected messages in the queue within the same connection, to deliver them to the user space and to perform the released memory buffers removing.

The socket interface for the direct message search was implemented on the basis of the Linux kernel 2.6.13. The experimental results were obtained according to model assumptions in [16] by testing to get a set of simple microbenchmarks. To get such a set at first look, two hosts can be used intended to send and receive messages via the established connection with the possibility to obtain a set of simple microbenchmarks outside the request.

For testing, the sender sends a message with a fixed configuration to the established connection several times. The receiver passes several times N queued messages, reads the expected message and then reads N missed messages that were set before the expected message. The message receiver is clearly configured to perform the recognition approach of the "unexpected" and "expected" message. In case of a regular message search, other N messages that are in front of it should first be copied from the socket buffer into an additional memory called the "pool of unexpected messages". In case of search for the expected message implemented in this paper, all missed unexpected messages are recognized and processed by using the developed seek_recv() system call.

Overall system performance can be estimated using the sum of both time intervals: system time and user time that can be explored in the message receiver system. The percentage of the processor active time that should be reduced in case of expected message direct search will be that metric indicator used to estimate the effectiveness of the implemented approach. Every test should consist of ~80 repetitions, each of which should include a set of operations such as socket opening, sending 800–1,000 messages to it in accordance with the acceptance politics described above and closing of the opened socket. The system should only use one active socket at the same time interval.

Fig. 1 shows the active CPU time reduction that is reached in the case of using the direct search interface. Several curves correspond to N values of unexpected messages set before the expected one in the range from 2 to 16, respectively. In the graphic with the logarithmic scale, the X-axis shows the message size in bytes. On the Y-axis, the reduction percentage of the CPU time is given for the case of using the socket with direct message search. It should be added that each data point has been checked 10 times minimum. The submitted curves show averaged values with standard error bands, in order to visually demonstrate the deviation of the experimental values present in the test studies. As can be seen from the dependency graph, the values obtained below the 0 % level point to those situations where the new interface still reduces the system performance. This especially concerns small-sized messages for which the scattering bands of test results are significantly larger. From the test experiment results shown in Fig. 1, a significant reduction of the error bands for messages with larger sizes is observed. This can be explained by the fact that the tests take more time and become less dependent on the random behavior of the operating system and caching. For the case of the message direct search mechanism and the developed interface involvement [16], the results indicate a noticeable decrease in CPU execution time by 36–40 %.

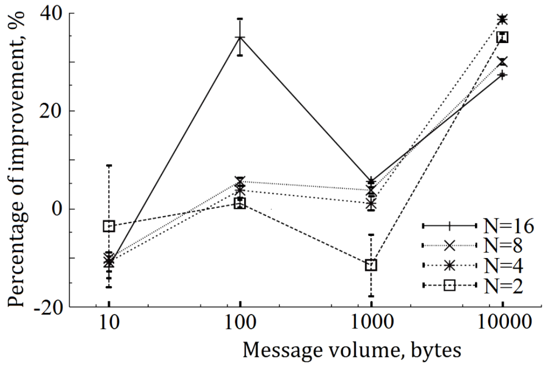

Fig. 1. Influence of various message sizes and their disordered amount N on the socket performance with direct message search

In general, the curves obtained in Fig. 1 show an improvement in performance indicators that are dependent on the message sizes increase or larger N values for unexpected messages. These results do not reflect the surprise if to take into account the need for more copy operations and additional their costs in the same tests. In the best of cases, the results indicate a noticeable decrease in the CPU execution time by 36–40 % for the case of involving newly developed interface [16] in operation.

7. Discussion of the research results for the message direct search mechanism

In overall, coming from the received research results for the direct search mechanism based on TCP protocols during the message exchange on a socket within one connection, a reduction in CPU processing time for the received expected messages is observed. This fact says about the overall increase in the exchange process performance for the case when the expected messages should be delivered to the user's space. The performance growth especially is observed in the process of receiving the expected messages that are set outside the established order of their receiving. The performance also increases significantly in case of sending the larger volume of bytes per message or in case of sending smaller volume with a larger number of expected messages.

However, along with all the positive results, it should be noted (Fig. 1) that the implementation of the search procedure sometimes leads to a general decrease in performance due to overheads increase because of the additional costs of the socket buffers processing. Such a decrease is observed especially for small values of N for previous unexpected messages, as well as for small messages in their small sending amount. This takes place, for example, in such cases as a list of "holes" managing for already received messages in order to expand the socket buffer.

There are also some performance decreases for such system that arise for 10-byte messages that are, for example, somewhat unrealistic amount of data representation in well-configured applications developed for high-performance computing systems. This approximation may be useful for application testing where it is difficult to coordinate the communication process, or by using the method of microbenchmarks to detect and compare the basic delay time in the communication channel.

During the testing process, some other interesting result of a performance reduction is revealed for the size of the messages of up to 1,000 bytes. This specified size for the message is closest to the size of the useful TCP packet download that is equal to 1,460 bytes. Consequently, the relative costs of managing the list of holes after reading messages, in this case, are somewhat higher than in the other mentioned cases.

The research results show that the socket interface for direct expected message search can be useful for communication and data transmission based on TCP. But in agreement with the obtained results, the high-speed power of the library to send messages around this interface is still not fully realized. The work experience with the code set for microbenchmarks says that it is needed to make only a few minor changes to individual functions to use direct message search sockets. Integrating this interface into the MPI library, it is possible to significantly improve additional ideas about the convenience of using it in the future. For example, applications will first need to check the message headers by using the regular network services, perhaps to take a look or pick them up before getting information about their number required for the search procedure.

On the other hand, the actual performance is likely to change depending on the number of sent messages, according to which messages are queued randomly. As it is known, the result of the set of microbenchmarks detects only the result of the amount of CPU processing time taken into consideration, but not its combination with the delay time during communication. According to the methodology [22], these results should be arranged to describe the overall performance of the proposed system. Expected results should also be largely independent of the physical intermediate devices use or the number of sockets. Since the changes in the operating system are limited to the socket level, they strictly take into account the basis of each socket.

The sockets with the direct message search can also result in performance improvement beyond the domain of the data computing cluster. For example, for a more effective TCP process, modern HTTP implementations transmit multiple simultaneous requests and replies to a single-directed conveyor connection [23]. Based on the common considerations, the sockets that perform direct message search can also be used for parallel data processing through an established one-way communication. This applies to heterogeneous stream readings, the information parting, i. e. its fragmentation, as well as the information content displaying of different connection parts.

As can be seen from the research analysis of the direct message search mechanism, it is necessary to consider the number of expected messages that are in the sample queue within a single connection. From the experimental results obtained by testing the implemented delivery system by the method of microbenchmarks, it is clear that this system distinguishes between expected and unexpected messages with variable byte volume due to the extended socket interface. However, the performance of such a system for large byte messages will depend more on the mechanism of their fragmentation into the corresponding parts before sending.

In the future, we plan to conduct a research on the use of the direct search mechanism with the extended socket interface integrated into a full-featured active library responsible for the messaging or communication. Important in this study will be performance (speed) dependence of the exchange system with the extended interface on various message fragmentation.

8. Conclusions

1. The direct search mechanism of randomly expected messages located in the stack during the established TCP communication between working machines is practically implemented in this paper. To obtain the research results, the system with the message direct search was tested with the method of simple microbenchmarks. Every test consisted of ~80 repetitions, where each of which included the sending of 800–1000 packages and other necessary operations.

2. The mechanism implementation used the extended socket TCP interface developed on a basis of traditional one, which searched and performed the receiving of messages from any stack position within a single connection, bypassing costly copying.

3. During the process to find the randomly located messages in the queue, the procedure of multiple their search was used. Along with message receiving, the message buffers are released to expand the memory of the overall socket buffer.

4. Based on the testing results, there is the CPU processing time reduction by 36–40 % for the received messages. The overall performance increase of the message exchange process has a significant manifestation in case of sending more messages or a large number of small messages. However, when the volume of messages is approached to the size of the useful TCP packet download, there is a decrease in the performance of the message exchange process.

References

1. Distributed processing of very large datasets with DataCutter / Beynon M. D., Kurc T., Catalyurek U., Chang C., Sussman A., Saltz J. // Parallel Computing. 2001. Vol. 27, Issue 11. P. 1457–1578. doi: https://doi.org/10.1016/s0167-8191(01)00099-0
2. Small-file access in parallel file systems / Carns P., Lang S., Ross R., Vilayannur M., Kunkel J., Ludwig T. // 2009 IEEE International Symposium on Parallel & Distributed Processing. 2009. doi: https://doi.org/10.1109/ipdps.2009.5161029
3. Managing Big Data with Information Flow Control / Pasquier T. F. J.-M., Singh J., Bacon J., Hermant O. 2015. URL: http://tfjmp.org/files/publications/cloud-2015.pdf
4. Melnyk V. M., Bahniuk N. V., Melnyk K. V. Influence of high performance sockets on data processing intensity // ScienceRise. 2015. Vol. 6, Issue 2 (11). P. 38–48. doi: https://doi.org/10.15587/2313-8416.2015.44380
5. Infiniband Trade Association. URL: http://www.infinibandta.org
6. Netgauge: A Network Performance Measurement Framework / Hoefler T., Mehlan T., Lumsdaine A., Rehm W. // High Performance Computing and Communications. 2007. P. 659–671. doi: https://doi.org/10.1007/978-3-540-75444-2_62
7. Majumder S., Rixner S., Pai V. S. An Event-driven Architecture for MPI Libraries // In Proceedings of the 2010 Los Alamos Computer Science Institute Symposium. 2010. URL: https://vjpai.github.io/Publications/majumder-lacsi04.pdf
8. Gilfeather P., Maccab A. B. An Extensible Message-Oriented Offload Model for High-Performance Applications // Los Alamos Computer Science Institute SC R71700H29200001. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.217.1085&rep=rep1&type=pdf
9. Veeraraghavan M., Jukan A. A hybrid networking architecture // University of Virginia. 2010. URL: https://pdfs.semanticscholar.org/62bb/a7fcb0e97adbf623a569c160d20843022b08.pdf
10. Aydin S., Bay O. F. Building a high performance computing clusters to use in computing course applications // Procedia – Social and Behavioral Sciences. 2009. Vol. 1, Issue 1. P. 2396–2401. doi: https://doi.org/10.1016/j.sbspro.2009.01.420

11. Pratt I., Fraser K. Arsenic: a user-accessible gigabit Ethernet interface // Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213). 2001. doi: https://doi.org/10.1109/infcom.2001.916688
12. Fenech K. Low-latency inter-thread communication over Gigabit Ethernet. University of Malta, 2005. 180 p. URL: http://citeseerx.ist.psu.edu/viewdoc/download?DOI=10.1.1.192.2018&rep=rep1&type=pdf
13. Ethernet Networking Interface // Techopedia. 2019. URL: https://www.techopedia.com/definition/24959/ethernet-networking-interface
14. Stream Control Transmission Protocol / Stewart R., Xie Q., Morneault K., Sharp C., Schwarzbauer H., Taylor T. et. al. // 2000. doi: https://doi.org/10.17487/rfc2960
15. Sockets API Extensions for Stream Control Transmission Protocol (SCTP) / Stewart R., Xie Q., Yarroll L., Wood J., Poon K., Tuexen M. // IETF Internet Draft. 2005. URL: https://tools.ietf.org/pdf/draft-ietf-tsvwg-sctpsocket-06.pdf
16. Melnyk V. Modeling of the messages search mechanism in the messaging process on basis of TCP protocols // Naukovyi zhurnal «Kompiuterno-intehrovani tekhnolohiyi: osvita, nauka, vyrobnytstvo». 2017. Issue 28-29. P. 20–24.
17. Liyanage M., Ylianttila M., Gurtov A. Fast Transmission Mechanism for Secure VPLS Architectures // 2017 IEEE International Conference on Computer and Information Technology (CIT). 2017. doi: https://doi.org/10.1109/cit.2017.46
18. Performance analysis of asynchronous Jacobi's method implemented in MPI, SHMEM and OpenMP / Bethune I., Bull J. M., Dingle N. J., Higham N. J. // The International Journal of High Performance Computing Applications. 2014. Vol. 28, Issue 1. P. 97–111. doi: https://doi.org/10.1177/1094342013493123
19. Implementation of the simplified communication mechanism in the cloud of high performance computations / Melnyk V., Bahnyuk N., Melnyk K., Zhyharevych O., Panasyuk N. // Eastern-European Journal of Enterprise Technologies. 2017. Vol. 2, Issue 2 (86). P. 24–32. doi: https://doi.org/10.15587/1729-4061.2017.98896
20. Obzor nekotoryh paketov izmereniya proizvoditel'nosti klasternyh sistem. URL: https://www.ixbt.com/cpu/cluster-benchtheory.shtml
21. Computer Network & TCP Congestion Control. URL: https://www.geeksforgeeks.org/computer-network-tcp-congestion-control/
22. 7 Steps Needed for Successful Benchmarking, using the COMPARE Method. URL: https://www.compare2compete.com/en/blog/7-steps-needed-for-successful-benchmarking-using-the-compare-method/
23. HTTP Definition. URL: https://techterms.com/definition/http