УДК 004.451
V. Melnyk
Lutsk National Technical University

MODELING OF THE MESSAGES SEARCH MECHANISM IN THE MESSAGING PROCESS ON BASIS OF TCP PROTOCOLS

Мельник В.М. Моделювання механізму пошуку повідомлень в процесі їх обміну на базі протоколів TCP. В даній роботі змодельовано розширення традиційного сокетного інтерфейсу для здійснення TCP/IP-комунікації з залученням нового механізму пошуку даних замість традиційного їх отримування за порядком встановленої черги. Пошук повідомлення через TCP-сокет дає можливість програмі користувача отримувати очікуваний пакет даних обминаючи чергу попереднього проходження всіх необхідних пакетів для їх отримання в порядку з'єднання. З залученням процедури багаторазового пошуку під час обміну повідомленнями прикладна програма чи бібліотека може розглядати TCP-сокет як типовий список пакетів-повідомлень, які можуть бути отримані чи видалені разом із їхніми даними як з вершини буфера сокета, так і з будь-якої довільної його позиції в стеку повідомлень. Змодельований механізм пошуку повідомлень полегшить процедуру копіювання даних між бібліотекою повідомлень та кодом користувача, обходячи спочатку необхідність в операції небажаного копіювання даних в буфер бібліотеки перед їх отриманням і подальшим відображенням прийнятих повідомлень в буфері сокета.
Ключові слова: мережевий інтерфейс, сокети, пошук повідомлень, процедура копіювання, продуктивність.

Melnyk V.M. Modeling of the messages search mechanism in the messaging process on the basis of TCP protocols. In this paper a traditional socket interface expansion for the implementation of TCP/IP communication with the involvement of a new mechanism is modeled for data retrieval instead of their traditional receipt in the established queue order. A message finding approach through a TCP socket allows the user program to receive the expected data packet by skipping the queue for the previous passage of all required packages to receive them in connection order. With the use of the reciprocal search procedure for messaging, the application or library can consider the TCP socket as a typical list of message packets that can be received or deleted along with their data from both: the top of the socket buffer and any arbitrary position of the socket in the message stack. The simulated message search approach facilitates the data copying process between the message library and the user code, bypassing first the need for unwanted data copying operations in the library buffer before they are received and subsequent displaying of the received messages in the socket buffer.
Keywords: network interface, socket, message search, procedure of copying, performance.

Мельник В.М. Моделирование механизма поиска сообщений в процессе их обмена на базе протоколов TCP. В данной работе смоделировано расширение традиционного сокетного интерфейса для осуществления TCP/IP-коммуникации с привлечением нового механизма поиска данных вместо традиционного их получения в порядке установленной очереди. Поиск сообщений через TCP-сокет позволит программе пользователя получать ожидаемый пакет данных минуя очередь предыдущего прохождения всех необходимых пакетов для их получения в порядке соединения. С привлечением процедуры многократного поиска при обмене сообщениями приложение или библиотека может рассматривать TCP-сокет как обычный список пакетов-сообщений, которые могут быть получены или удалены вместе с их данными как с вершины буфера сокета, так и с любой произвольной его позиции в стеке сообщений. Смоделирован механизм поиска сообщений облегчит процедуру копирования данных между библиотекой сообщений и кодом пользователя, обходя сначала необходимость в операции нежелательного копирования данных в буфер библиотеки перед их получением и последующим отображением принятых сообщений в буфере сокета.
Ключевые слова: сетевой интерфейс, сокеты, поиск сообщений, процедура копирования, производительность.

1 Introduction. High-performance cluster computing are mainly used to perform long-term cumbersome calculations of a scientific or applied nature. Such calculations can be divided into separate computational parts and assign each of them to run on one or even several computers. Computers with the appropriate frequency are sending the data and messages to each other in order to update the information integrity related to the computing distribution, or to provide other computers with the data required as parameters or additions for the subsequent parts to make calculation. The leading structures based on cluster components [1-3] can provide a flexible and efficient environment for applications with intensive data processing on distributed platforms [4]. For such structures, the applications are specifically developed from a set of selected interacting software components, which, along with the computing resources, are important in terms of flexibility and optimization of the program performance.

The publication [4] also describes that in many applications with intensive data processing, the volume of data can be divided into user-defined sub-blocks that can be computed like pipelined one. If the organization of work processing and communication may overlap, then the productivity improvement also depends on the computing blocking and the size of messages, named data sub-blocks. There is noted that small blocks of data lead to improve the loading balance and piping, and in the communication practice many messages for this matter are generated with blocks of small sizes. However, the larger blocks reduce the number of messages and reach the higher bandwidth in the communication channel, but probably make a load imbalance and reduce the conveyance.

Along with the demands of performance and data processing intensity, the applications written using kernel and TCP/IP socket interfaces also have other requirements, such as performance guarantee, scalability of these guarantees and adaptability to heterogeneous networks. To allow such applications to get an advantage of high-performance protocols usage, the researchers have used a several approaches, including performance of high-performance socket levels through the user-level protocols. To these approaches are included: the virtual interface architecture implementation and InfiniBand technology architecture [5]. The applications used them should be written in taking into account the saving of the connection performance in TCP/IP.

The literature results show that because of the individual components reorganization in such applications there are achieving the significant improvements in application performance, which leads to an increase of the performance guaranty scalability and balancing with the small blocks loading. This approach in turn makes these applications more adapted to heterogeneous networks. The different socket characteristics worked on the virtual interface architecture allow for more effective data division at the output nodes, which increases the performance up to one order. In union with high performance the sockets with low overhead allow the applications to achieve quality results in many measurement areas that can be used in designing, developing and implementation of applications with intensive computing on modern clusters.

Despite the cluster structure allows many computers to be used as fully as possible to reduce the time spent on computing, yet clustered computer systems suffer overhead during the required high-performance computer communications. The overheads to perform such calculations also depend on the number of computers in the cluster, the libraries used to facilitate communication, and the choice of the interface between computers and other component units within the computing cluster. Despite the wide experimental results for cluster interactions, some studies have demonstrated the interdependence of efficient library design for messaging in clusters that use TCP/IP and Ethernet components to achieve the performance comparable to proprietary interconnects [6, 7].

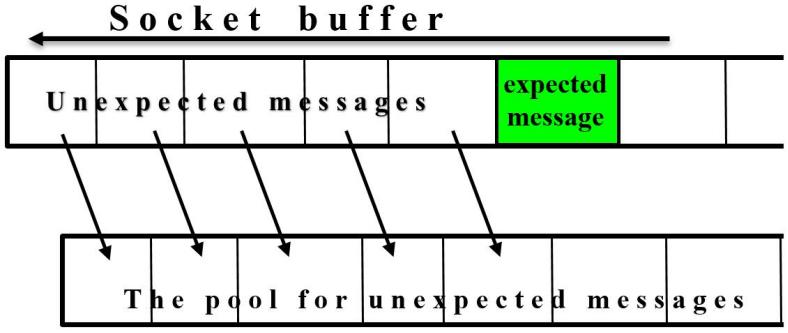

Fig. 1. The dependence of the expected message receiving effectiveness from the previous messages in the queues and unexpected messages

The overheads dependence that appeared when using messages based on TCP/IP protocols depends not only on the message sizes and their receipt frequency, but also on whether the message is expected or unexpected. A message may occur as an unexpected one if its data is received by the receiver before a system call to the library in order to receive such message in the memory buffer on the user program level. It is known [7, 8] that unexpected data is first copied to a temporary library buffer. For sending messages through TCP/IP protocols, the message can be considered as received when its data appears on the network, and the TCP stack places it in the connection socket buffer between the two communicated hosts. Fig. 1 shows us a situation where host expects the appointed message along with unexpected messages in the socket buffer that were arrived to the system at the same time or before. For example, the system expects a message with the interface of its transmission with a specified type of tag [9]. The message passing interface (MPI) is a standardized and portable messaging system. It was developed by a group of scientific and industrial researchers and appointed for implementation on a variety of parallel architectures to make a data computing. In cluster communication practice, there is present several quite effective interface implementations, many of which are free-license available and open to use. These raised an approach that pushed for parallel software development and large-scale and portable applications creation that are designed to perform parallel computing. In case of the network communication, the operating system socket interface for TCP protocols receives certain bytes from the formed connection in the established order, using one of the system calls: recv() or read().To count this it first need to release the socket from the previous unexpected messages in order to access the expected message in next step. In the general process of message receiving by applications, these data most likely

are to be needed later, then each of them should be copied to the reception pool of unexpected messages. Such operations of unexpected messages copying cause significant overheads. In the next step, before to receive the expected message on application level, it need to check first the pool of unexpected messages receiving, and only then to call for the message to receive it at the socket level.

The purpose of this work is to simulate the usual socket interface of the operating system for its extension through which it can be possible to access the random expected message, i.e. to its location in the socket buffer, bypassing the receiving queue of all previous incoming messages that were arrived before expected one. The new advanced interface should perform the search for the required information (message) in the socket buffer and allow the user to receive the expected message from any stack position bypassing the operation of copying the previous messages into the pool of unexpected messages that arrived before the specified expected message (fig. 1). From the side of the development requires implementation it need to involve the multiple search process. The messaging program or the usage library should consider the TCP socket as a list of messages that can be received by the user application, and during the receiving procedure it need immediately to delete the received data not only from the top but from the arbitrary point of the socket buffer, where expected message was set.

An interface for sockets that search for messages in the stack by the method described above is complementary to those works that aim to increase the performance of the messaging process by using TCP protocols. Among them can be selected those works, that are focused to use a re-constructed libraries with a more efficient design. These libraries can be managed by events from the side of its own complementary architecture [10] and/or can use a special support from the side of hardware reconfiguration, other than the generally accepted, or the network card interface. The second type of re-constructed libraries can be those that are based on TCP-level splitting [7, 11]. However, this work is based on the efficiency of using the operating system interface on the socket level. Therefore, the model described in this paper should collaborate with ideas that focus on supporting the network card interface or on usage a well-designed library to enhance the performance of other system components for TCP messaging.

The interface to perform a search procedure of the expected message on the socket can be developed and implemented on the basis of the Linux kernel and verified by using the method of a simple microbenchmark, the data for which is obtained outside the query, implemented for this socket. The results of some qualitative calculations reveal that reducing of the processing time with using sockets that directly search the message in the stack should reach more than on a third part. The performance growth of such system will depend on the receipt of expected messages with large volume or the number of messages that are to be bypassed in the receiving queue.

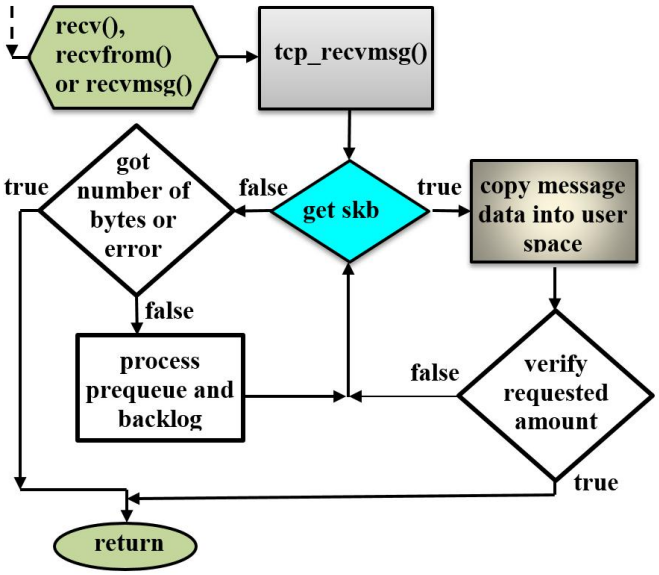

Fig. 2. The algorithm of the recvmsg() function for the Linux-based TCP messaging

2 Basics of the functions work on Linux TCP. The TCP stack under control of the Linux kernel works with socket buffers that are mentioned in literary sources as sk_buff's, sock_buf's or simply skb's. The data packets in the receiving process are received by the network device, they are placed in a ring buffer, and sk_buff is assigned to the data and associated to them. The buffer sk_buff saves the metadata

for each package, and the stack of the Linux operating system for TCP/IP, interacting with sk_buff to process the data packet. For each particular connection sk_buff is placed in the socket buffer queue.

To facilitate the procedure for receiving them in the correct order and managing the packages, for each of them is assigned a serial number in the queue, which specifies the number of bytes sent for each packet particularly during the connection. The implementation like this allows to recover packets on the reception side during retry at the network level and remove data from socket buffers. The user application does not know or should know the ordering of the data in the packets or the timeliness of their receipt in the network. Tasks that are solely associated with a user's application are considering in that, how data elements are sent from a source and their real fixed lengths.

When the user proggram makes a call any of recv(), recvfrom() or recvmsg() functions on the TCP socket, then tcp_recvmsg() function is used in the kernel (e.g., net/ipv4/tcp.c). Fig. 2 shows a working algorithm chart for these functions. The tcp_recvmsg() function begins the data copying from sock_buff's buffers in the socket queue which point to the actual data placed 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 requested, the tcp_recvmsg() function leaves a "reminder" in the socket buffer queue. If the user requests more data than the data amount placed on the first skb, it is selecting along with the corresponding data in the ring buffer, and the specified steps described above continue with the next going skb in the queue. By default, the function tcp_recvmsg() returns all query data from the socket buffer after the full read procedure in full volume, removing all the skb's buffers that contained their data, and updates the sequence number of the first byte to perform the next socket reading operation.

TCP usually uses sequence numbers to track the way that was read from the socket buffer, and determines the continuation order of the readout procedure. At each step, the copied_seq variable, which records the order of the duplication, receives the sequence number for which the data will be read. In other words, this variable will contain an ordered sequence of what has already been copied and that will be read further from the reception queue. In this case, if the sequence of numbers copied_seq was greater than the number of the first basic skb sequence, and the part has 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. Thus, using the sequence numbers will be used to determine the data that the reception request should read from the socket.

3 Implementation of additional socket search procedure. The main purpose of using sockets that are directly seeking for a message in a queue is to receive data placed on the socket receiving buffer in any order. When the data is copied from the socket, the corresponding skb with the copied data must be removed from the buffer, and the current list of skb's should also be corrected. As data that was read in the sequence numbers is no longer present and available, then the subsequent requests for their receipt on the socket should "know" and take into account that these data are missing in the buffer. This can be implemented through a connection list that contains the initial and final sequence numbers for each "hole" in the socket reception buffer. Creating a hole frees up memory space in the socket buffer and allows to normalize the behavior of the TCP stream independently from the location place in the buffer from which the data was deleted. When the reception request begins the process of data copying to the user, it avoids any met hole on the path and continues normally to receive the data from the ordinal number that follows after the hole. In the process of the data receiving the list of holes increases with their integrity, and at the same time takes place the dynamic reduction associated with their removal.

The model developed in this paper proposes to create a new streaming protocol SOCK_FIND_STREAM. It should use the same stack as the TCP and ordinary SOCK_STREAM sockets. However, the basic functions should be modified so that it could be possible to make search of the sockets with SOCK_FIND_STREAM type. If a socket request is performed to a socket that does not make the search procedure, or if the call does not find the package that is expected to receive, then the way through the TCP stack with its functions is almost identical to the code used by the normal Linux kernel. But when the search request is performed on the socket that makes the search procedure, then the way through the TCP stack remains the same, only may change the code way through separate functions. Basically, the changes are related to the tcp_recvmsg() function code that is introduced into this function. All the introduced modifications are required for managing the list of holes, their sequence numbers and skb 's, which have already been read yet.

Another major change should be made in the function tcp_recvmsg() that relates to the procedure for socket obtaining on which the search procedure has to be made. It includes a disabling of the TCP queue pre-loading mechanism. Although the TCP queue pre-loading mechanism allows some better manage with the stream resources in the exchange process, however it causes a slight decrease in

performance. Also in the general messaging procedure, it is not possible to easily change the downloading queue to allow further searches. In connection with the foregoing, the pre-loading mechanism was turned off at the receiving moment on the socket that makes the search.

After the SOCK_FIND_STREAM socket has been created, the known above functions recv() and recvmsg() can be used as ordinary functions. The new search function find_recv() should be realized like a system call, that needs to get the following arguments:

size_t find_recv(int s, void *buf, size_t len, int flags, size_t offset);

The arguments to call find_recv() are identical to the function recv(), with the change added to indicate the number of bytes to be transmitted to the stream. This offset always points to the first byte that must be obtained through the use of recv() system call.

Since the call find_recv() changes structure msghdr, and then calls the general function sock_recv(), then the standard library function recvmsg() can also be used to search for recipients. The variable msg_find has been added to the msghdr to indicate a search offset. By modifying the msghdr structure taken to the function recvmsg() it is easier to search for the package that is expected to be received without the new special function involvement. In order to be able to search previous messages of a large size, it need to increase the maximum message reception buffer size. This parameter adjusts the system variable net.core.rmem_max(sysctl). It allows to set the maximum buffer size for each received message. The maximum buffer size can be reset by using the known function setsockopt(). Consequently, the parameter sysctl should be set to a sufficiently large number of bytes, and the reception buffer should be increased, if necessary, over the entire interval of the user program work. In the case where the buffer is fully filled, then the usual TCP actions are executed, and the call to receive the result in the course of the search implementation should return an error. Then the program should clearly respond to the error of the socket buffer overflow and delete some data remaining in the buffer to release more space in the kernel.

5 Conclusion. In this paper the model for a new socket level extension is proposed, which allows to get an irregular access to expected message from a single TCP connection. The interface for sockets with searching procedure for expected message and obtaining data outside the query is proposed, which in the opinion of the authors can be implemented in Linux system. The model shows that for developing start of an application code are required only small changes to implement message exchange.

1. Beynon, M. D. Distributed processing of very large datasets with DataCutter. [Text] / Beynon M. D., Kurc T., Catalyurek U., Chang C., Sussman A., Saltz J.// Parallel Computing – October 2001 – 27(11) – P.1457-1478.
2. Oldfield, R. Armada: A parallel file system for computational. [Text] / R.Oldfield, D. Kotz //In Proceedings of CCGrid2001 – May 2001. – P.194--201.
3. Plale, B. dQUOB: Managing Large Data Flows by Dynamic Embedded Queries [Text] / Plale B., Schwan K.// IEEE High Performance Distributed Computing (HPDC) – August 2000 – P.1-8.
4. В. М. Мельник, Н. В. Багнюк, К. В. Мельник. Вплив високопродуктивних сокетів на інтенсивність обробки даних / Науковий журнал «ScienceRise» №6/2(11)2015. – с. 38-48.
5. Infiniband Trade Association. [Електронний ресурс] / Режим доступу: http://www.infinibandta.org.
6. M. Allman, V. Paxson, and W. Stevens. TCP Congestion Control. IETF RFC 2581, April 1999.
7. N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE MICRO*, 15(1):29–36, 1995.
8. P. Buonadonna and D. Culler. Queue Pair IP: A Hybrid Architecture for System Area Networks. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 247–256, May 2002.
9. D. Dunning, G. Regnier, G.McAlpine, D. Cameron, B. Shubert, F. Berry, A. Merritt, E. Gronke, and C. Dodd. The Virtual Interface Architecture. *IEEE MICRO*, 18(2):66–76, March 1998.
10. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol.-HTTP 1.1. IETF RFC 2616, June 1999.
11. S. Majumder and S. Rixner. Comparing Ethernet and Myrinet for MPI Communication. In *Proceedings of the Seventh Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR 2004)*, pages 83–89, October 2004.
12. S. Majumder, S. Rixner, and V. S. Pai. An Event-driven Architecture for MPI Libraries. In *Proceedings of the 2004 Los Alamos Computer Science Institute Symposium*, October 2004.
13. S. H. Rodrigues, T. E. Anderson, and D. E. Culler. High-Performance Local Area Communication With Fast Sockets. In *Proceedings of the 1997 USENIX Technical Conference*, pages 257–274, January 1997.
14. T. M. P. I. Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.
15. V. Melnyk, N. Bahnyuk, K. Melnyk, O. Zhyharevych, N. Panasyuk. Implementation of the simplified communication mechanism in the cloud of high performance computations. East-European journal of Enterprise Technologies. – Kharkiv (Scopus, DOI: 10.15587/1729-4061.2017.98896). – 2017. – № 2/2/86. – p. 24-32.
16. P. Gilfeather and A. B. Macabe. Making TCP Viable as a High Performance Computing Protocol. In *Proceedings of the Third LACSI Symposium*, October 2002.