# Algorithm of Handling Out-of-Order Delivery for Multithreaded UDP-based Data Transport

Dmytro Syzov, Dmitry Kachan and Eduard Siemens
*Department of Electrical, Mechanical and Industrial Engineering,*
*Anhalt University of Applied Sciences, Bernburger Str. 55, 06366 Köthen, Germany*,
*{dmytro.syzov, dmitry.kachan, eduard.siemens}@hs-anhalt.de*

Keywords: High-Speed Data Transport, Mutli-threading, Out-Of-Order Delivery, Transport Protocols.

Abstract: As industry of information technologies evolves, demand for high speed data transmission steadily increases. The need in it can be found in variety of different industries – from entertainment withtrends for increasing of video to scientific research. One of the consequences is a demand for new improved transport protocols that would use the capacity of Long Fat Pipes by maximum, where common TCP performs much slower than it is expected. Such protocols are mostly based on UDP and work at the user space. To improve their network throughput, there is an option to implement sending data in a multi-threading way, but that can bring complications with it. One of the main obstacles is a possibility of out-of-order delivery due to race conditions. This problem is researched in current paper. Causes of reorder are studied regarding UDP-based transport protocols. Based on the results of the testing, a simple algorithm for compensating out-of-order delivery is proposed. It's effect then is analysed on the example of RMDT.

## 1 INTRODUCTION

The common limitation of operating systems – involving of significant resources on each system *send* and *receive* calls – leads to the performance limitation on sender side of such an application. Of particular interest is a problem of a high data rate traffic generation on a sender side. Especially in cases of point-to-multipoint communications, when the same data has to be transmitted to multiple destinations, as sender has to produce more traffic than each of the receivers has to process. This can be resolved by introducing of a multi-threaded send process into a transport protocol. The idea behind the use of multi-threading for performance improvement lies in fact that only part of system call actually concerns working with NIC. So, theoretically it could be possible to invoke *sendmsg()* or *recvmsg()* system calls, which can be used as "send" and "receive" operations on Linux from different cores and all processing, that is not concerned NIC, will be performed in parallel. Such approach can be applied as *sendmsg()* and *recvmsg()* are thread safe and re-entrant (Linux Programmer's Manual, 2017). Thus, these calls can be performed in parallel and so resulting data rate can be increased. Another important fact is that UDP preserves message boundaries (IEEE Standards Interpretations, 2017). Theoretically, there is no reason to assume that within this method there are some fundamental limitations of maximum data rate achievable.

Besides the speed boost, multi-threading in sending and receiving data can bring a number of problems on its own. One of them is a problem of efficient scalability regarding the system limitations. Another one is possible interleaving of packets due to asynchronous send operations, which is subject of investigations in current paper. For transport protocols this may present certain pitfalls as packets that are out of order could be considered lost by its ARQ algorithm. This work aims at provision of some insights into the packet reordering problem and proposes a simple algorithm to overcome it.

In general, there are mechanisms, such as signals, mutexes, conditional variables, that allow to avoid

such packet reordering problems. However, the downside of using these mechanisms is radically reduced performance as they usually include waiting for synchronization, and when the data rates are on the level of gigabits per second, even a block for a small amount of time can decrease output from NIC significantly. Thus, it is important to keep sender lightweight. Considering arguments, presented earlier, only lockless data exchange mechanisms are used in this work – precisely lockless queues.

For a simple application that consists of a lockless queue as IPC mechanism and *sendmsg()* system call which performs the interaction with network hardware, is considered as a test subject. In such an application – a few points of possible reorder are present:

- out-of-order timings of *dequeue()* operations
- out-of-order return of the object from a queue
- out-of-order send call

First and second points can be generally considered as one since they produce the same result – reordered read from the IPC queue.

There is a possibility to handle out-of-order packets without mechanisms that create additional load on sender. This work analyses the behaviour of a multithreaded data transmission application and analyses the proposed algorithm that handles the problem of reordering without locking and works on the receiver side, which is important, as its implementation does not create an additional load on the sender threads, thus does not decrease sender performance.

# 2 RELATED WORK

The lack of networking performance caused by CPU limitation is a problem that is of relevance in almost every multi-gigabit data transmission environment. This problem is clearly shown in research (Srivastava, 2014), which explores the problem of traffic generation for a 40 Gbps channel by comparison of several generators: D-ITG, packETH, Ostinato. As a result, S. Srivastava et al. state that no traffic generator was able to achieve the 40 Gbps rate. Authors suggest to use multithreaded generation of traffic. D-ITG - a generator from proposed research, which utilizes the channel using 16 threads. However, no additional research on impact of multithreading on packet-reordering was presented. To obtain more data on implementation of multi-threading the advantages of a multi-threaded approach for a network UDP-based application were investigated in a separate work (Syzov, 2016). Conclusion is, that multi-threading is beneficial for the fast traffic generation. It compares performance of cases with various amount of threads (from 2 to 20) on a 10 Gbps link. This work shows clear increase in performance with increasing number of threads as with 3 threads almost 10 Gbps rate has been achieved. With more than 12 threads, data rate starts decreasing. This number corresponds to exceed of the amount of CPUs and can be explained by overhead on threads management.

Another work (Nguyen D., 2007) shows the methodology for development of a multi-threaded network application, which correlates with this work. Research, among other subjects, considers two of the main pitfalls in a multithreaded network application - race conditions on data transport and inter-process communication. As explained by Nguyen D. et al., in an unsynchronized application, there is a possibility of data races and stresses the potential harm that it may cause due to reordering and data corruption. However, that work does not go into detail and does not propose a solution. In current research, the problem of possible reorders, caused by race conditions, is investigated further with tests made and a proposed algorithm for reordering avoidance.

# 3 TESTING ENVIRONMENT

All tests were performed in 10 GE Laboratory of Future Internet Lab Anhalt (FILA, 2017). The core element here is the WAN emulator Netropy 10G that can be used to create an emulation of WAN links.
During each test, 10 GB of data are transmitted. MSS is equal to 1472 bytes as it corresponds to common 1500 Ethernet v2 MTU (IETF, 2017). For sending and receiving, two Linux servers are used. Their specifications are presented in table 1.

Table 1: Servers' specifications.

| Name | Server 1 | Server 2 |
|---|---|---|
| Kernel | 4.4.0-38generic x86_64 | 4.4.0-45lowlatency x86_64 |
| CPU | Intel Xeon X5690 (6-core) 3.5 GHz | AMD Opteron(tm) 4238 (6-core) 3.3 GHz |
| Memory | 40 GB DDR3 | 32 GB DDR3 |
| NIC | Chelsio Communications Inc. T420CR | Intel Corporation 82599ES |

Since system call execution times can show significant spikes, all the figures with measurement results present filtered data – significant deviations are treated as outliers and are removed from data set. It is done in order to have a closer look on the behaviour of the tested configuration as original data often contains spikes that are rare and have different causes, which are not studied in this paper. The outlier filtering is performed by Tukey method (Frigge, 1989), it rejects outcomes, which are out of inter-quartile range (approximately 2.698σ).

For tests, apart from *C Library* and *C++ Standard Library*, following open source non-standard libraries were used:

- moodycamel::ConcurrentQueue (concurrent queue, 2017) for inter-process communication;
- HPTimer (Fedotova, 2013) for precise time measurements.

## 3.1 IPC Means

Since an intensive use of threads is present in this work, an appropriate IPC mechanism is required. Due to specific use case, there are some key requirements for a queue:

- Ability to work in a Single Producer, Multiple Consumers mode
- Low time of enqueue and dequeue operations.

Also a few additional requirements are given, that simplify usage of the queue and give more options to a developer:

- Ability to acquire approximate number of elements in the queue or avoiding overflow and gaining information on senders' performance without direct communicating with sender threads;
- Support of a dynamic allocation of additional memory for the option to increase queue size if senders significantly slows down for some period of time.

Following these requirements, *moodycamel::ConcurrentQueue* was chosen as it provides fast enough operations and also slow degradation of performance. It provides approximate amount of objects currently placed in the queue, which can be used to determine if threads work correctly without additional queue for the backward channel. Apart from this, the possibility to enqueue only if there is free allocated memory is present, which is useful if dynamic behaviour is not desired.

## 3.2 Time Acquisition

In order to retrieve data on timings of various operations a precise time acquiring mechanism is required. For this purpose the HPTimer library has been used, since it provides faster time acquisition than standard std::chrono library (Fedotova, 2013). It is worth to note that each measurement contains overhead of the timer itself which however is non-negligible.

# 4 TEST AND ANALYSIS OF REORDERS

For analysis and evaluation of reorder causes, some research should be made in order to analyze the behavior of a multi-threaded application in general. The stability of send call timings is of interest as inconsistency may lead to race conditions. In a real case, however, each send iteration includes additional operations that are not directly connected to a send call itself, the program as a whole is not executed constantly and, apart from all else, the system call may not take the same time on each iteration. To assess, how system handles *sendmsg()* call, some experiments have to be performed.

To acquire information on timings of main operations on sender threads' side, a test has to be performed with measurements of *sendmsg()* and *dequeue()* operations in sequence. The algorithm is minimalistic for precise measurements. It does not contain any operations apart from measured ones, time measurements and *std::vector::push_back()* operation to a reserved storage per loop. Results are presented on figures 1-3.

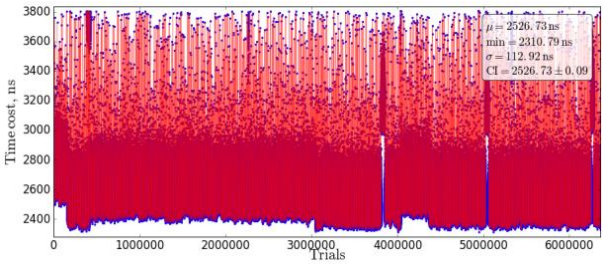


Figure 1: *sendmsg()* operation time measurements on Server 1 in a thread.

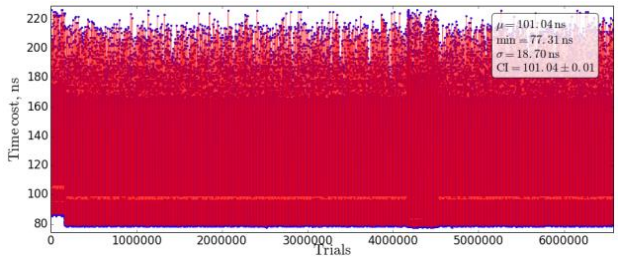


Figure 2: *enqueue()* operation time measurements on Server 1 in a thread.

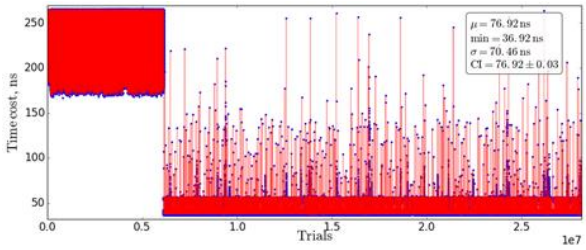


Figure 3: *dequeue()* operation time measurements on Server 1 in a thread.

On all figures, there is some inconsistency observed. The most prominent one is a significant drop on figure 3, that occurs when enqueue process on producer side (figure 2) is finished. However, it should be noted, that in tested case *dequeue()* takes much less time than *sendmsg()*. Also, in comparison to pure *sendmsg()* in a single thread, there are more inconsistencies in this case (deviation of 112ns vs. 63ns).

Next test aims to determine the volume of packet reorders in an application. As there are two main possible points of reorder causing operations, each of them is tested separately and then in combination. For this purpose, a set of test applications has been developed.

For tests, all data collection is placed at the receiver. In the test with no queue, the differentiation between sender threads is performed by setting predefined calculation of sequence numbers. The one, used in this test is defined by formula 1:

$$SN_i = ID * N_{threads} \quad (1)$$

where $SN_i$ it the sequence number of message *i*; *ID* – thread identification number; $N_{threads}$ – amount of threads. In that way, each sending thread has its own sequence of numbers, that differs from others. With this approach, it would be incorrect to count out-of-order numbering inside one loop of each thread. More appropriate would be to count reorder cases, when order of numbers differs on each loop or if one of the threads sends messages faster than others. For the final test with queue, no additional functionality on the sender side is required. Receiver simply gets the message, then separates and stores a sequence number. The amount of threads, that are of interest, are 2, 3 and 11. Amounts of 2 and 3 are important as in this cases the maximum bandwidth of a 10 Gbps link is reached. The case with 11 threads represents the maximum quantity of sender threads for having one thread per CPU as one thread is a main application. However, for a better overview of the behavior, two additional numbers of threads between 3 and 11 are also considered. Such test can provide some information about significance of reorders as necessity of handling them depends on it.

For each case 40 trials were conducted. Collected data is analysed and the mean percentage of reorders is calculated. Each deviation from the expected next number is treated as reorder in case if factual number is bigger than expected. Results are presented in table 2.

Table 2: Percentage of reordered packets on Server 1.

| Tested case, threads | 2 | 3 | 5 | 8 | 11 |
|---|---|---|---|---|---|
| sendmsg() | 50% | 33% | 21% | 16% | 10% |
| Sendmsg() + dequeue() | 0.02% | 4.2% | 6.2% | 14% | 31.3% |

As can be seen, *sendmsg()* is not handled well by the kernel in regard to proper ordering. Another conclusion is that internal blocking of the send call in kernel space can decrease the reordering percentage, since the increase in the amount of threads decreases reorder percentage. As for combined *sendmsg()* and *dequeue(),* there is an expected increase in percentage of out-of-order delivery. However, it is not linear. And in case of 2 threads, the percentage is small enough to be neglected.

To check if this behaviour is the same for different hardware, an additional test for a *sendmsg()+dequeue()* was conducted on a different server. Results are presented in table 3

Table 3: Percentage of reordered packets on Server 2.

| Tested case, threads | 2 | 3 | 5 | 8 | 11 |
|---|---|---|---|---|---|
| Sendmsg() + dequeue() | 2% | 3% | 6% | 18% | 30% |

As can be seen, while the percentage is different for some cases, the difference is generally not significant and the behavior remains the same.

Apart from percentage of reorders, the depth (in packets) between expected receive of a reordered packet and factual is of interest. It can show how long the application should wait before it can send NACK to get optimal performance. Results of processing collected data are presented in table 4 for cases with 2, 8 and 11 threads.

Table 4: Depth of reorders (in %).

| Depth / Scenario | 1 | 2 | 3 | 4 | 5 | 6 and more |
|---|---|---|---|---|---|---|
| Server 1, 2 threads | 1 | 69 | 1 | 7 | 7 | 15 |
| Server 1, 8 threads | 4 | 33 | 21 | 13 | 3 | 25 |
| Server 1, 11 threads | 4 | 25 | 16 | 9 | 4 | 42 |
| Server 2, 2 threads | 0 | 97 | 0.3 | 0.3 | 0.3 | 2 |
| Server 2, 8 threads | 2 | 11 | 7 | 3 | 1 | 76 |
| Server 2, 11 threads | 2 | 6 | 3 | 1 | 2 | 86 |

From data, presented in table 3 it can be concluded that generally reorders tend to have depth of 2 or 3. Also, there is a significant difference between results on server 1 and 2. While on server 1 most of reorders have depth of 2 or 3 even if the amount of threads is increased, on server 2 with additional threads added percentage significantly shifts to more deep.

In a more close to a real use scenario with a serialized sequence of *dequeue()* and *sendmsg()*, the presence of a single data producer via the queue mostly compensates the timing reordering of packets by the kernel. Also, the percentage of reorders in the case of two threads is negligible. This is important as in some cases two threads can already reach 10 Gbps data rate, which might be enough for most applications. However, with addition of more threads there is a rapid increase in out-of-order delivery percentage. This fact means that there is a necessity in a mechanism that would handle such behavior to avoid decrease in utilization due to packet reorders.

# 5 PROPOSED REORDER HANDLING IN THE PROTOCOL

To compensate out-of-order delivery an algorithm is suggested for implementation on the receiver side which handles the packet reordering in a feasible way. Basic principle of the algorithm is that every thread sends packets with thread-specific sequence numbering in addition to the connection-specific numbering. In the described multi-threaded sending scenario, is safe to assume that all packets that have numbers lower than the least number from received last from each thread, are either lost or received. For purposes of this algorithm, some bytes at the header have to be reserved for a number of a thread, that sends the data packet. This has two main consequences:

- Maximum amount of sender-threads is restricted by the maximum thread number in the respective header field;
- Additional operations for processing data are to be placed on the receiver side.

A flow chart of the described approach is shown on figure 4 and visual representation of packet reordering on figure 5.

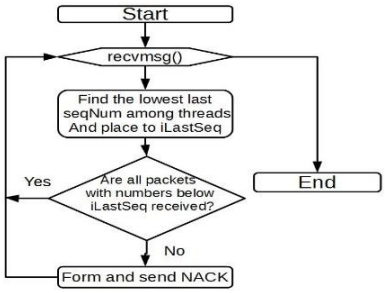


Figure 4: Flow chart of the reorder handling algorithm.

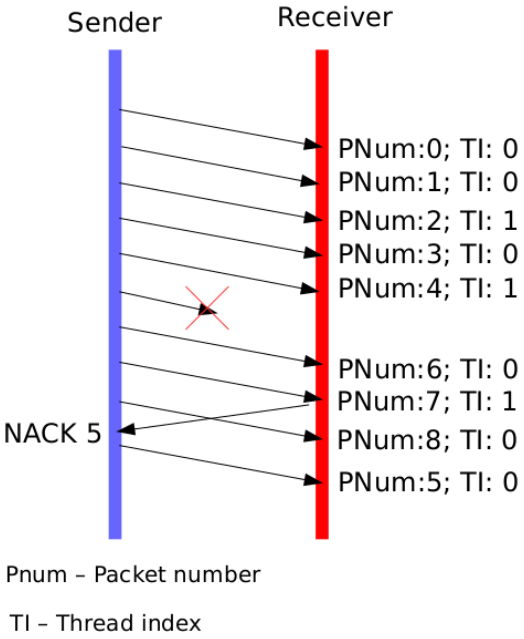


Figure 5: Visualization of the reorder handling algorithm.

Here TI is a unique thread ID and $P_{num}$ is a connection-global sequence number of a packet. As can be seen, packet 5 was lost during transmission, but receiver does not send NACK immediately, but

rather waits until it can be sure that the packet is actually lost. NACK is sent after packet 6 from thread 0 and packet 7 from thread 1 are received.

When implemented on the receiver side, it can handle reorders caused by multithreading by such approach. However, it does not cover other causes for out-of-order delivery. Also, in real-world scenarios, apart from principles, described in the algorithm, some modifications have to be applied. The reason for that is the possibility of packet losses. As receiver has to notify sender about missing packets at some point, some functionality regarding this has to be implemented. Two generally used solutions are:

- Setting the timeout. If a missing packet was not received in a predefined period of time it is considered to be lost;
- Defining a number of packets, that can be received after a missing one. If missing data was not received after that number, a packet is considered to be lost.

TCP, for example, implements both approaches as it has a defined window, but also TCP has a timeout for each packet to be received. If timeout is exceeded or if last message of a window is received, missing packets are considered to be lost. In regard to algorithm explained in this chapter, the number of packets that are received after a missing one depends on the depth of reorders.

With example of RMDT, the use of 8 theads with server 1 as a sender, unhandled reorders will result in 14% loss on it's own. And if transmission is performed throug channel with losses, the total percentage of packets retransmitted can be even higher, thus, decreasing overall performance of a protocol. However, by implementation of reorder handling algorithm with waiting window of 4, most of reorders will be handled and difference in performance between these two cases is more than 10%. On the other hand, in a scenario of 2 threads, the percentage of reorders is low enough to be ignored.

The main difference between this algorithm and simple wait for a defined number of packets or a timeout is that it allows to differentiate between loss and reorder on the run. Thus, it does not significantly decrease the performance of the ARQ protoclol.

# 5 CONCLUSIONS

There is a demand in transport protocols, that can efficiently and reliably transmit data. To develop such a protocol, a number of problems have to be considered. One of them is a preservation of the ordering of data packets as for some types of ARQ, an out-of-order packet might be equal to a lost packet. In this work, basic reasons for out-of-order delivery caused by multithreading were considered. With measurements on timings of operations involved and reorders themselves, some insight was provided into behaviour of a multithreaded network application. In a case of 2 threads, depending on hardware, the percentage of reorders ranged from 0.02% to 02% with depth mostly equal to 2 (from 69% to 97% of reorders).

For the problem of reordering, to optimize data integrity preservation, an algorithm was suggested and it's benefits evaluated on the example of RMDT.

# 5 FUTURE WORK

Possible continuation of this work is developing and testing more complex algorithm that would include handling out-of-order delivery in general, not only that caused by multi-threading. More work can be done on evaluating the influence of reorders in a real transport protocol. In particular, the subject of reordering in wide area networks should be researched. Such research may provide information necessary for developing an appropriate out-of-order handling mechanism in protocols that operate on wide area network.

Additional tests should also be performed for different setups. Of special interest are tests with different types of hardware and its´ configuration. Also, tests with dynamically changing load on CPU and memory usage are of interest. Based on the results of such tests, the proposed algorithm can be improved to be able to handle variety of situations correctly.

For testing approach as a part of a real protocol, if all functionality will be proved to work correctly, this approach can be tested as a part of an UDP-based multi-threaded transport protocol for high speed data transmission.

# ACKNOWLEDGMENTS

This work has been funded by Volkswagen Foundation for trilateral partnership between scholars and scientists from Ukraine, Russia and Germany within the project CloudBDT: Algorithms and Methods for Big Data Transport in Cloud Environments.

# REFERENCES


Linux. socket. In *Linux Programmer's Manual*

IEEE. *IEEE Standards Interpretations for IEEE Std 1003.1c. Amendment 2: Threads Extension.* [Online]. Available from: http://standards.ieee.org/findstds/interps/1003-1c-95_int/pasc-1003.1c39.html. 2017.02.12

Srivastava S., Anmulwar, S., Sapkal, A. M., Batra, T., Gupta, A., and Kumar, V., 2014. Evaluation of traffic generators over a 40Gbps link, in *Computer Aided System Engineering (APCASE)*, Asia-Pacific Conference, pp. 43–47.

Syzov, D., Kachan, D., Siemens E., 2016. High-speed UDP Data Transmission with Multithreading and Automatic Resource Allocation in *Proceedings of the 4th International Conference on Applied Innovations in IT*, Koethen : Hochschule Anhalt, pp. 51-56

Duc Chinh, N., Kandasamy, E., Yoke Khei, L., 2007. Efficient Development Methodology for Multithreaded Network Application in *The 5th Student Conference on Research and Development-SCOReD* 2007 11-12 2007, Malaysia

FILA. *Future Internet Lab Anhalt* [Online]. Available from: https://fila-lab.de. 2017.02.12

Apposite. *Apposite Technologies :: Linktropy and Netropy Comparison.* [Online]. Available from: http://www.apposite-tech.com/products/index.html. 2017.02.12

Internet Engineering Task Force. *RFC 894 - A Standard for the Transmission of IP Datagrams over Ethernet Networks.* [Online]. Available from: https://tools.ietf.org/html/rfc894. 2017.02.12

Frigge M.,, Hoaglin D. C., Iglewicz, B., 1989. *Some Implementations of theBoxplot,* The American Statistician, vol. 43, no. 1, pp. 50–54.

Concurrent queue*. A fast multi-producer, multi-consumer lock-free concurrent queue for C++11*. [Online]. Available from: https://github.com/cameron314/concurrentqueue/ 2017.02.12

Fedotova, I., Siemens, E., Hu, H., 2013. *A high-precision time handling library,* J. Commun. Comput., vol. 10, pp. 1076–1086.