High-speed UDP Data Transmission with Multithreading and Automatic Resource Allocation

Dmytro Syzov, Dmitry Kachan, Eduard Siemens
Anhalt University of Applied Sciences - Faculty of Electrical, Mechanical and Industrial Engineering,
Bernburger Str. 57, 06366 Köthen, Germany
E-mail: {dmytro.syzov, d.kachan, e.siemens}@emw.hs-anhalt.de

***Abstract*— In this paper a utilization of the high data-rates channels by threading of sending and receiving is studied. As a communication technology evolves the higher speeds are used more and more in various applications. But generating traffic with Gbps data-rates also brings some complications. Especially if UDP protocol is used and it is necessary to avoid packet fragmentation, for example for high-speed reliable transport protocols based on UDP. For such situation the Ethernet network packet size has to correspond to standard 1500 bytes MTU[1], which is widely used in the Internet. System may not has enough capacity to send messages with necessary rate in a single-threaded mode. A possible solution is to use more threads. It can be efficient on widespread multi-core systems. Also the fact that in real network non-constant data flow can be expected brings another object of study –- an automatic adaptation to the traffic which is changing during runtime. Cases investigated in this paper include adjusting number of threads to a given speed and keeping speed on a given rate when CPU gets heavily loaded by other processes while sending data.**

***Keywords:* high-speed data transport, threading, automatic resource management.**

I. INTRODUCTION

High-speed content delivery is a service that is more and more demanded by society over the time. And for some purposes, like a transmission of huge amount of data, TCP may be not appropriate and another reliable transport protocol is needed. UDP serves as a base for such protocols.

Protocols that are built on top of UDP (for example: RBUDP [2], UDTv4 [3], RWTP [4], RMDT [5]) rely on the message consistency. Thus, it is crucial that MTU of the generated message will not be bigger than maximum size of the data field of the frame used in channel layer. This requirement especially important for high-speed transport protocols and solutions [6][7]. For Ethernet standard MTU size is 1500 bytes.

Unlike TCP sockets, UDP sockets preserve message boundaries [8]. This is why it is safe to queue multiple calls to the system as message consistency will not be violated. This fact allows creation of a multi-threaded UDP sender and receiver, which can be used in different kinds of UDP-based protocols for reliable transmission. Flexible thread handling with automatic resource allocation and control over threads parameters can simplify development.

For threads management there are certain problems to be solved:

1. Data rate control.
2. Automatic resource allocating.

For send rate regulation certain means need to be implemented for the ability to generate messages with a constant data-rate and change it in a runtime. To automatically allocate resources – an algorithm of making decisions about allocation that are based on information which can be collected without significant overhead should be added . Because of the fact, that traffic parameters can vary or amount of system resources, that are available for the application, may be changed while the application is running – mentioned functionality is important for such a system.

II. RELATED WORK

Research [9] shows the basic problems of traffic generating for a 40 Gbps channel. It is comparing different traffic generators for network testing: DITG, packETH, Ostinato. For experiments held in [9] payload varies from 64 to 8950 bytes. Both TCP and UDP traffics were measured. Results describe the exact problem that is studied further in this work: none of traffic generators is capable of achieving the full bandwidth utilization, unless packets with the high payload were used. In case of D-ITG even using 8950 bytes payload was not sufficient. Although, authors in [9] use 40 Gbps link, the problems remains for a 10 Gbps link, as on packet size of 1500 bytes, traffic generators were unable to achieve 10 Gbps.

Solution, suggested in [9], is to utilize system resources by using threading. It shows that, when D-ITG is using 16 threads, the higher data-rate can be achieved with less payload per packet, in comparison with D-ITG running in a single-threaded mode. Same approach for achieving higher data-rates is used in this work and researched deeper.

In paper [10] authors also suggest using multi-threading for sending and receiving. Among other subjects studied, research shows bottlenecks of achieving close to channel capacity performance. Using threading helps to get high bandwidth utilization for channels with high data-rates. The handling of such systems is studied in [10] by researching the effects of system parameters on a throughput.

In this work handling of multiple threads and management of them is studied further from algorithmic and practical point of view.

III. METHODS

Several algorithms for solving problems described in introduction are presented in this section. They are implemented using "High Precision Timer" library [11] to get accurate time as they heavily rely on time stamps and to implement precise thread sleeping mechanism.

Data rate can be controlled in different ways:

1. Basic principle is blocking(force the thread to sleep some time) each thread for a specific time to decrease data-rate. They may have the same delay time or different if necessary. After each sending, time of the next message sending is calculated. It can be done by adding to the previous expected time a given time delay.
2. Other way is to add delay to the current time after sending.

The first approach will have different effect than the second. If a sender process is constantly delayed for some time due to some external influence, for example the other process is using the same CPU resource, and after that resource has been freed – the thread will start sending packages frequently until the real sending time will meet ideal expected time. The second approach does not give rate more than requested and application will try to keep the data rate constant.

To get the ability to change speed during runtime threads periodically update information about size of its inter-packet time interval. Threads do not necessarily have the same send rate, it can be assigned individually.

Initial inter packet interval time for each thread is calculated by formula (1):

$$T_{interval} = \frac{MSS \times 8 \times N_{th}}{R}, \quad (1)$$

where $T_{interval}$ – an inter-packet interval for a thread; MSS – MSS of a packet to be sent by a thread in bytes; 8 – constant amount of bits in one byte; N_{th} – given amount of threads; R – a data-rate of the whole multi-threaded system in bytes per second.

Other method is used to automatically handle threading. The information, used to make a decision about running or stopping sender thread is:

- the time of a full send loop (that can include, apart from send() system call, message generation, additional calculations, etc.),
- time of sending,
- time lag – the difference between actual sending time and expectation time (time when message should be sent to achieve given speed).

More precise, the time lag can be sufficient for a decision to spawn send processes, but additional time-related data give more information about sender behavior.

Approach is based on the assumption that if a total time lag of all threads is higher than zero, then capacity of existing threads is not enough. If it is less than zero, it is assumed that existing threads have more capacity than needed to achieve requested rate.

Theory behind this assumption is that if a thread can not send within a given time interval it will accumulate within each iteration the difference between target time for sending and factual time – time lag. Another case, when thread can perform the send operation within shorter period than the given inter-packet interval, this will result in an accumulation of the time it has to wait by blocking itself manually. Resulting lag – positive or negative difference – is used to evaluate current performance. Flow chart that illustrates algorithm is presented in the Fig. 1.

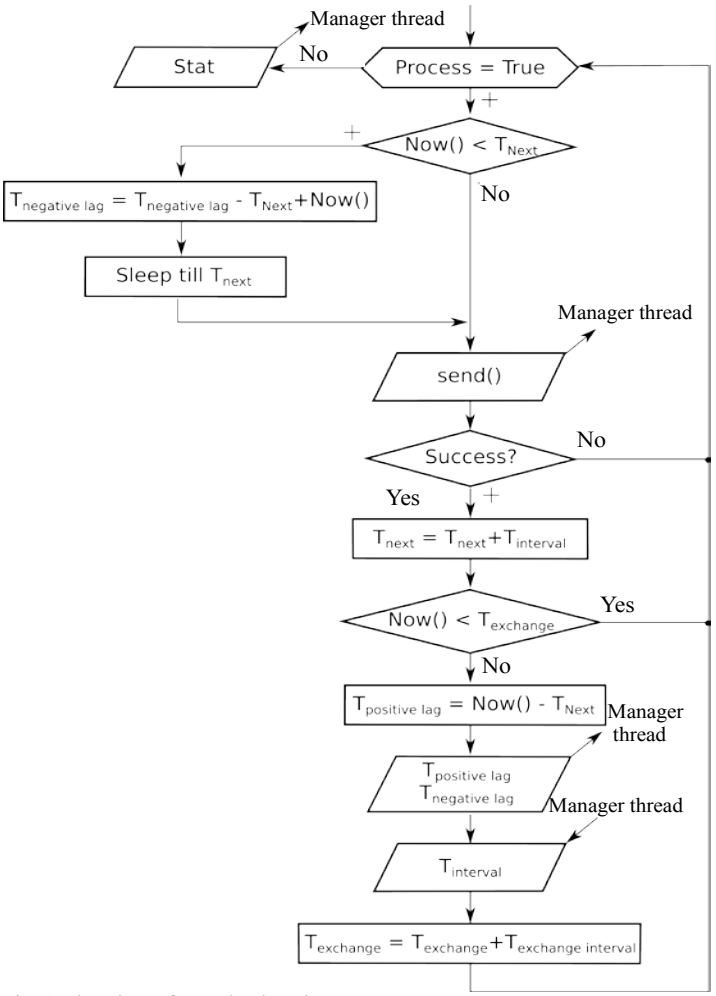

Fig. 1. Flowchart of a sender thread

in the Fig. 1:

$T_{interval}$ – the inter-packet interval for a thread; $Stat$ – a structure that contains the information about time parameters and collected at the end of the sending session; $T_{positivelag}$ – variable that accumulates the difference between expected and factual time of sending messages; $T_{negativelag}$ – variable that accumulates the time spent on waiting, caused by manual blocking; $Process$ – boolean variable that is used to stop thread; T_{next} – time when next send call should be made; $Now()$ – "High Precision Timer" function that provides current time; $Send()$ – UDP send call; $T_{exchange}$ – time when next data exchange between sender and main threads should be

made; $T_{exchangeinterval}$ –interval between such exchanges.

One specific thread manages all others to perform the functionality described previously: collecting information from other threads, adjusting necessary parameters and initiating spawn of a thread. This manager thread is blocked most of the time, and it unblocks periodically for performing its functionality.

An ideal case is when necessary rate is achieved without blocking and, thus, system resources are maximally utilized. If the requested data rate is not equal to a multiplication of a certain amount of threads' maximum generating rate – then, to achieve given rate, some thread blocking must be performed and at least one of the threads will have time lag less than zero.

IV. Testbed Topology Description

The core element of the tested topology is WAN emulator AppositeNetropy 10G [12] that can be used to create an emulation of WAN links with different impairments such as packet loss ratio up to 100%, delays of up to 100000ms and delay jitter with an accuracy of about 20ns. The Emulator allows a transmission of Ethernet traffic with an overall throughput of up to 21 Gbps on both, copper and fiber optic links. Apart from Netropy, setup contains two PC servers. They are connected via an Extreme Networks Summit x650 10 Gbps Ethernet switch and the WAN Emulator. Fiber optics with 10 Gbps bandwidth acts as a medium for transmission between compartments. There is no background traffic used for experiments, since in the focus of presented investigations is research of the pure traffic generation.

Each server is equipped as follows:

−CPU: Intel Xeon X5690 @3.47GHz;

−RAM: 42 GiBytes (speed 3466 MHz);

−OS: Linux CentOS 6.3;

−NIC: Chelsio Communications Inc T420-CR, 10Gbps

Also, for comparison of the performance on a different system, some tests were performed on servers with different CPU: *Intel Xeon E2630 @2.30GHz.*

V. Experimental Results

Firstly, the performance of the sender is compared for a different amount of threads, MSS and speed. MSS is taking following values: 1024, 1472, 8972 bytes. Data-rate that were tested is 10 Gbps.

First experiment is for MSS of 1472. Results of the experiment for 10 Gbps rate are presented in Fig 2. Since 1472 bytes MSS corresponds to 1500 bytes MTU in Ethernet [1] this test is especially important to the experiment as its results are of interest for transport protocols, used in TCP/IP networks, which often use Ethernet technology on the channel level. As was mentioned in the introduction, the consistency of a message is a hard condition for the protocols built on top of the UDP protocol.

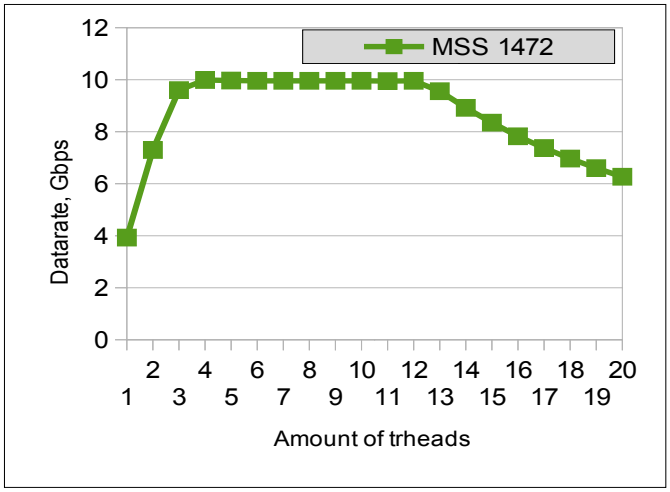

Fig. 2. Dependence of the datarate from the amount of threads, MSS 1472

As can be seen on the graphs, frames with size of 1500 bytes does not meet speed requirements when only one sender is present. When the amount of threads is higher than number of cores (or Hyper-threads) the opposite effect can be observed. The overhead of context-switches decreases performance rapidly. Thus, the data-rate of traffic generating is limited by the amount of CPU threads.

Additional test, run on the CPU with lower frequencies but same amount of CPU threads, gave same behavior, but higher number of senders is needed to achieve 10 Gbps.

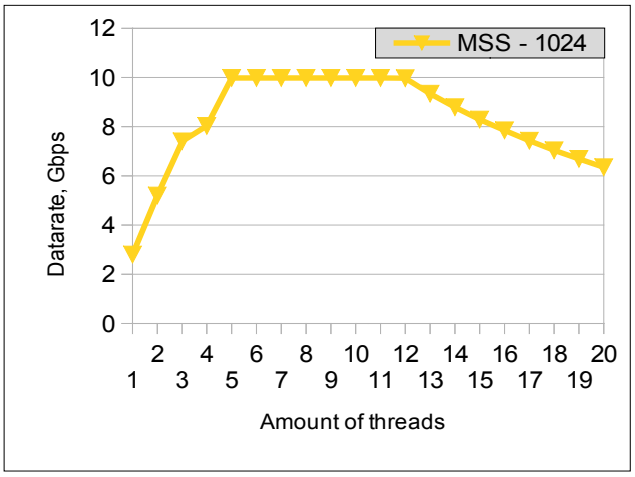

Fig. 3. Datarate dependence from the amount of threads

Next, two traffics with significantly different MSS are tested to check if the same pattern can be observed for other packet sizes, and discover possible dependencies.

Results for 10 Gbps are presented on the Fig. 3 and Fig. 4.

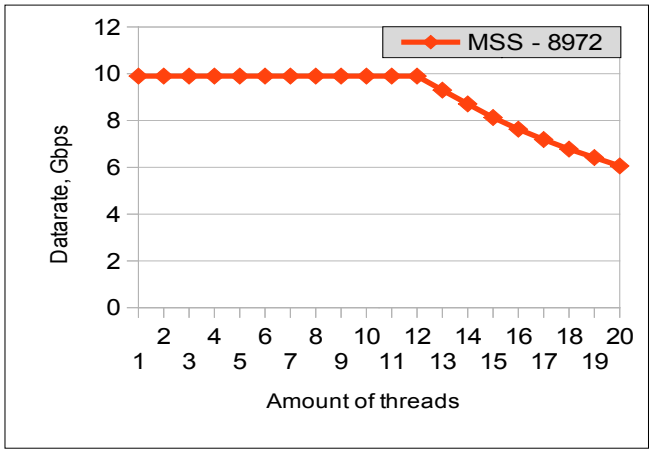

Fig. 4. Datarate dependence from the amount of threads.

It is clearly visible that the bigger MSS makes achieving higher speed easier. Also, whereas at lower number of threads there is a clear difference in speed, when CPU's threads limit is reached the speed drop is approximately the same in both cases. Thus, conclusion is made that using maximal MTU for the channel is beneficial and has no negative influence.

Receivers were also tested and, as they handle traffic much faster than senders, 3 reception threads are sufficient for any MSS or speed, used for experiments held in this work. Thus, this part of system is less critical than sender side as it requires less resources and for tested system does not have large difference in number of threads in the studied cases. It is clearly dependent from speed – higher rates need more threads. In this experiment no automatic receivers' allocation was implemented – testing is performed by manually stating the number of receivers.

Test was made with an application that does not control data rate by manual blocking to compare how system handles multiple senders by itself. This resulted in a different form of traffic as it has more fluctuations than traffic generated by application that implemented manual blocking threads. Other result is higher number of errors caused by trying to access a resource which is taken by another process.

For comparison, simple send threading was also performed on a PC that has processor with lower frequency (number of CPU's threads is still 12). Rate generated by each thread is lower on CPU with lower frequency. A sender on a 2.3 GHz processor has 3.2 Gbps rate (MTU – 1500). A sender on a 2.47 GHz CPU creates 3.9 Gbps traffic.

Auto-spawning sender threads were tested with different speeds, initial amount of time, and time interval between evaluations. The fluctuations of the time lag are around one sender loop time, which corresponds to the mean value of 3.5 µs with standard deviation of 1.4 µs on the tested setup. Thus, whereas the time lag that is considered to be sufficient for spawning new thread is dependent on a particular situation, it is unsafe to take the decision based on a sign of a time lag, as it may lead to changing amount of threads because of a random small deviation. Some limit must be given to prevent unjustified thread spawning. Although not all causes for the fluctuations are clear from this experiment it is clear that higher limit gives less probability to spawn redundant process, but higher probability of not getting requested speed. Higher thread-spawning limit of lag gives wider interval of allowed data rates. The requested speed is only one value from this range.

At last, behavior of the application under utility "stress", which is an utility for imposing load on a system for Linux [13], is checked. Tests showed that new thread is started on the first information analysis (which is performed by main thread) after "stress" uses cores that are already used by senders. Interval between information gathering and analysis used it tests is 0.5 second. At this stage of development for each 0.5 seconds only one thread can be initialized. Thus it takes 1 second to achieve 10 Gbps rate if only one sender was initialized at the start of the application.

VI. Conclusions

The behavior of a single-thread traffic generator is examined in a real network. With MTU of 1500, using single sender, 10 Gbps data rate is not achievable on a tested topology. Using multi-threaded send and receive methods proved to be a working solution as it allows to achieve 10 Gbps speed which is full bandwidth of a tested topology. Theoretically it allows to get any rate, supported by network equipment, though it is limited by the amount of CPU threads. Increasing MSS results in higher speed per thread.

When the amount of threads is higher than number of CPU threads the opposite effect can be observed. The overhead of context-switches decreases performance rapidly. This is observed for all tests with different packet sizes and data rates. Thus, the data-rate of traffic generating is limited and if necessary data rate is not achieved with CPU, fully loaded by application, the conclusion about impossibility to provide requested rate can be made.

[Dmitry Ka1]The automatic thread management basic algorithms proved to be working in a simple environment with constant traffic, although some deviations from theoretical behavior were experienced as, for example, time jitter or accidental rate decrease of a particular thread, while the others have the expected rate.

VII. Future Work

Possible continuation of this work is developing and testing more complex algorithm with advanced thread management and smart statistical data evaluation.

First logical improvement of the existing application is the implementation of the automatic temporary stopping or permanently terminating thread. In this work only cases when rate is not achieved, but not the case of decreasing speed in time which brings necessity to free resources occupied by redundant threads. Also, algorithm of automatic receiver threads management has to be developed.

More tests should be run for different setups. Of special interest are tests with constantly changing traffic. Adaptation to such kinds of traffic is one of the main goals for algorithms described in this paper. Based on the results of such testings, they have to be improved to be able to handle variety of situation correctly.

Next step could be combining functions of pure send and receive with other, often used, operations – for example I/O.

Finally, if all functionality will be proved to work correctly, it can be tested as a part of an UDP-based

transport protocol for high speed data transmission such as, for example, RMDT.

REFERENCES

[1] "RFC 894 - A Standard for the Transmission of IP Datagrams over Ethernet Networks." [Online]. Available: https://tools.ietf.org/html/rfc894. [Accessed: 04-Mar-2016].

[2] E. He, J. Leigh, O. Yu, and T. A. DeFanti, "Reliable blast UDP: Predictable high performance bulk data transfer," in *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, 2002, pp. 317–324.

[3] Y. Gu and R. Grossman, "Udtv4: Improvements in performance and usability," in *Networks for Grid Applications*, Springer, 2008, pp. 9–23.

[4] S. Höhlig, "Optimierter Dateitransfer über 100 Gigabit/s," in *100-Gigabit/s-Workshop in Mannheim, Mannheim*, Germany, Sep-2011.

[5] "Big Data Transmission | F I L A." [Online]. Available: https://fila-lab.de/index.php/our-work/big-data-transmission/. [Accessed: 09-Mar-2016].

[6] D. Kachan, E. Siemens, *Comparison of Contemporary Protocols for High-speed Data Transport via 10 Gbps WAN Connections.* Proceedings of 2nd International Conference on Applied Innovations in IT. Köthen, pp. 21-27, 2014 (DOI: 10.13142/kt10002.04);

[7] D. Kachan, E. Siemens, V. Shuvalov, *Available bandwidth measurement for 10 Gbps networks.* Proceedings in 2015 International Siberian Conference on Control and Communications (SIBCON), 2015, pp. 1–10.

[8] *Linux Programmer's Manual,* Linux, p. "socket".

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

[10] V. Vishwanath, T. Shimizu, M. Takizawa, K. Obana, and J. Leigh, "Towards terabit/s systems: Performance evaluation of multi-rail systems," in *High-Speed Networks Workshop, 2007*, 2007, pp. 51–55.

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

[12] "Apposite Technologies :: Linktropy and Netropy Comparison." [Online]. Available: http://www.apposite-tech.com/products/index.html. [Accessed: 04-Mar-2016].

[13] *Linux Programmer's Manual,* Linux, p. "stress".