Giuliano Scollo (Hrsg.): SKILL 2023,
0 Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2023

Performance Comparison Analysis of Data-Exchange Efficiency in Middleware Systems with UDP and Shared Memory

Giuliano Scollo[1]

Abstract: This work aims at creating a systematic comparison of the data transmission throughput using UDP or Shared Memory. Since there is no empirical comparison of these two methods this research aims to fill that gap. For testing, a middleware as well as a communication test tool were used that transferred data packets at a given frequency. By controlling the frequency and the size of the data packet a maximum possible data throughput was deducted.

Keywords: Performance; Analysis; UDP; Shared Memory; Middleware; Data transmissions; Comparison

[1] Technische Universität Braunschweig, Informatik, Universitätspl. 2, 38106 Braunschweig, Deutschland; German Aerospace Center, Institute of Transportation Systems, Lilienthalplatz 7, 38108 Braunschweig, Germany giuliano.scollo@tu-bs.de

1 Introduction

There is a lot of research regarding data handling and transmissions. This does not only include user-generated data, but also machine-readable data sets and telegrams that are exchanged on Machine2Machine (M2M) level to ensure certain vehicle functionalities.

Data-exchange middleware systems are commonly used to marshal the data transmission between the respective data sources and destinations. Middlewares commonly use the User Datagram Protocol (UDP) or Shared Memory (SHM) to transfer the data. This paper compares these data-transmission techniques.

UDP was developed as part of network communication to send data with a minimum of protocol overhead [7], as opposed to for example the Transmission Control Protocol which has to recover from data that is damaged, lost, duplicated, or delivered out of order by the internet communication system [6]. UDP is a protocol that is transaction-oriented, delivery and duplication protection are not guaranteed. UDP assumes that the Internet Protocol (IP) is used as the underlying protocol [7]. Using the localhost interface UDP can also be used to exchange data between processes on the same host. Some middlewares apply UDP to exchange data between clients on the same host.

SHM is a technology to achieve interprocess communication (IPC) between processes on the same host by giving multiple processes access to the same memory space [3]. The processes have to synchronize their SHM access by using a so-called semaphore. Prior to accessing the SHM a process has to acquire the semaphore and to release it afterwards. Other processes can wait at the semaphore until it is released to gain access to the SHM. SHM has the reputation of being a fast way of IPC on a host [1], so middlewares tend to apply it as an alternative to UDP although it is more difficult to implement.

However, no literature could be found about a systematic comparison. This paper aims at filling that gap. To achieve that, this research investigates the data throughput of each communication method and its reliability for local data transmissions.

Two parameters determine the data throughput: the frequency at which data packets are transferred and the size of the packets. A test tool was developed to measure the maximum packet sizes at different frequencies sent via UDP and SHM. Since the design of UDP and SHM is completely different a variety of possible influential factors come at play. To estimate their effects, the test environment measures the influential factors step by step.

2 Methods

2.1 Software

To conduct the research two independent softwares were applied:

- Dominion – A middleware developed at the German Aerospace Center, Institute for Transportation Systems.
- A communication data transmission tool specifically developed for this research.

Dominion is a middleware to exchange data in soft realtime between so-called Dominion applications (apps for short). To test the data throughput the set up consisted of a writing app (writer) and a reading app (reader). Dominion tranfsers data between apps on a local host using SHM and uses UDP to synchronize the SHMs of different hosts. Dominion calls the central working method of each app (called `run` function) regularly at fixed time intervals of 10 ms by default. The `run` function reads the data the app consumes from SHM, processes it, and writes the data to share with other apps back into data structures located in SHM. In case a consuming app is on another host the producing app sends its data also via UDP. The special transmission tool mimics this behavior by creating a writing and a reading subprocess. Since all our tests are performed locally the decision which method to use is not deducted by the host location but by the user. The transmissions happen at a cycle time similar to the execution of dominions `run` function.

To reach the limits of UDP and SHM communication both tools were deployed in Docker containers, which allows to easily limit the allocated CPU performance by a Docker parameter. In case of Dominion, the Docker containers could also be set up to enable or disable a common SHM to enforce UDP communication.

2.2 Data Transmission — Model View

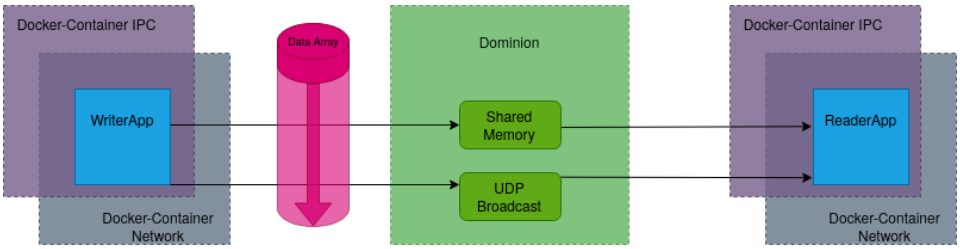


Fig. 1: Test environment with Dominion

Fig. 1 shows the test environment setup, used for testing data throughput with Dominion. For data exchange a WriterApp and a ReaderApp are created using Dominion. In every test the apps are configured according to the used method.

For UDP the apps are both in a dedicated Docker network that allows UDP broadcast between them. Testing the apps using a SHM space to allow interprocess communication (IPC). In this case the IPC happens between the Docker containers that run the applications. Encapsulation of the applications in containers enables controlled restrictions of the communication of the processes. The WriterApp sends data arrays to the ReaderApp, which in turn reads the content of the array. In this scenario Dominion as a middleware connects the applications. Dominion's architecture is such that the data is sent directly from the WriterApp to the ReaderApp without a data broker or routing instance in between[2].

2.3 Hardware

As primary test hardware a Dell Latitude 5501 notebook was used (NB). In addition, tests were conducted on a Raspberry Pi 4 (RPI) as alternative more limited hardware with a different chip architecture. Tab. 1 gives an overview over the main properties of the hardware platforms.

Name	CPU	OS	Kernel	RAM
NB	Intel(R) Core(TM) i7-9850H CPU @ 2.60GHz	Ubuntu 18.04	5.4.0-65-generic x86_64	32 GB
RPI	Broadcom BCM2711 quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz	Raspberry Pi OS (Raspbian)	5.4.83-v7+ armv7l	4 GB

Tab. 1: Test hardware

The most important differences are the available RAM and the chip architecture with x86_64 on the NB opposed to the armv7l on the RPI.

2.4 Pseudo code

Algorithm 1 describes how the general data exchange of the Communication Tool works. This algorithm is resembling the behaviour of the real code and does not include every subroutine.

[2] On initialization Dominion apps register at the so-called DominionServer, which tells the apps who are their data consumers.

Algorithm 1 Send/Receive data (Communication Tool)

```
input  int multiplicity                          (how big is the data array)
input  int cycle_time                            (in wich frequency shall the data be transmitted)
input  int max_cycles                            (how many transmissions shall happen idealy)
 1: for 0 → maxcycles do
 2:     prepare_packets(multiplicity)
 3:     int start_time = now
 4:     for all packets do
 5:         transmit_packet(i)
 6:     int send_duration = now - start_time
 7:     callback()
 8:     sleep(cycle_time - send_duration)
 1: for 0 → maxcycles do
 2:     for all packets do
 3:         receive_packet(i)
 4:     callback()
```

For sending and receiving data a similar sequence of actions were used. Before each test the `multiplicity` and `cycle_time` are set and given. They describe how big the data array is that will be sent and how often the transmission occurs. Also given is a limit how often this transmission is executed until a test run finishes. The main difference of both is that the sending process is scheduled by the `cycle_time`. Every execution starts by preparing all packets that will be sent. Here the data array is split according to the maximum transmission unit of either SHM or UDP. Then the first timestamp is saved for later reference. Next, all packets are transmitted. Now the second timestamp is set to get the time spent transmitting all packets. Since sending data should always happen in the same time interval no matter how big the data array gets, the raw sending time gets substracted from the `cycle_time` to get the remaining interval that this thread needs to wait before another execution. Before the thread sleeps the `callback` function is called wich will be explained in Algorithm 3. For our reading process this timing is not neccessary since all data that was sent will be processed.

Algorithms 2 and 3 show the subroutines detecting failed transmissions. Dominion's aproach of using a single semaphore was extended by the tool to react based on timer calls, because this job is handled by an event handler thread in Dominion.

Algorithm 2 Timer thread (timer increases counting semaphore after each cycle)

```
1: if semaphore_count > 100 then
2:     print(timer_error_message)
3:     reset_semaphore(10)
```

The main job of Dominions timer thread is to keep track of a counting semaphore that indicates the state of the current interaction. If it has reached a value of 100, the thread signals a timer error and resets the semaphore value to 10. This error message is our main

indicator for a failed data transmission, since it represents asynchronous behaviour that had to be corrected by Dominion.

Algorithm 3 Callback check

```
input  int cycle_time                          (in wich frequency shall the data be transmitted)
input  int last_callback_call                  (timestamp of last callback call)
 1: wait cb_semaphore
 2: if last_callback_call > now - cycle_time then
 3:     print(timer_error_message)
 4: last_callback_call = now
 5: release cb_semaphore
```

This process is executed when the `callback` function is called. To know what the set cycle time was and to keep track of execution a timestamp of the last time this function was executed is saved. Guarded by a semaphore this method checks how long ago the last callback call was. If the time that passed is longer than the set cycle time an error message is printed. Either way the last callback-call timestamp is set to the current time for the next check. This mechanism is a criterion for the success of a test. The callback check mimics the thread timer error Dominion uses but registers every failed test and does not only reacts if 100 violations happened.

2.5 Testing procedure

The success of a test using Dominion is determined by the timer thread error. If this message is contained in the output the given multiplicity is considered too high to reliably be transmitted at this cycle time. Given this information the resulting maximum was approximated by first testing with a multiplicity equal to half of the limit. This limit was set based on the results produced by Dominion's UDP transmissions and corrected up to 1 000 000 since SHMs results were way higher than Dominion's average UDP results.

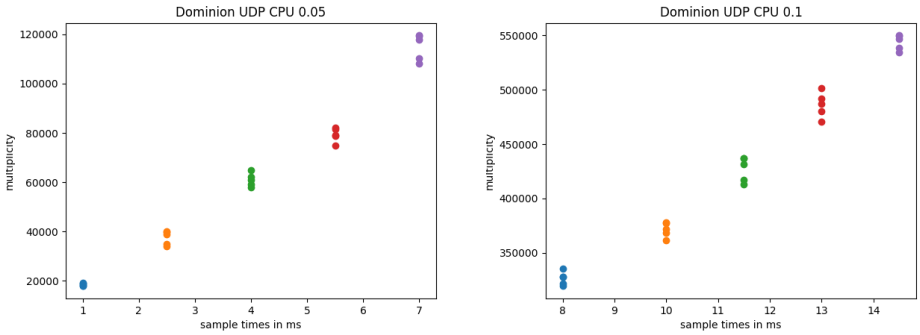
If the test was successful the multiplicity was increased by half of the last multiplicity. Else the multiplicity is decrased by half of the last multiplicity. If the last used multiplicity and the new one are at a difference of 2 the test is finished and the highest multiplicity resulting in a test is passed as the maximum. The success of a test using the communication tool is determined by the callback timer error as well as how many cycles were completed. A cycle stands for one transmission of the full multiplicity. If the writer sent the multiplicity 1000 times and the reader catched 99.95 % of those transmissions the test was considered as successfull. As representative enough without producing too much data 1000 transmissions per test were sent. Reducing the success rate from 100 % to 99.95 % did not significantly change the test results for cycle times above 5 ms but improved the success rate for lower cycle times especially for UDP. Testing without any fault tolerance is reserved for future research. The callback error message is used to determine how often a cycle was completed too late. This is an approximation on how reliable the respective methods are.

3 Tests and output description

Creating an environment that allows to compare the two methods is quite difficult, because Linux kernel developers put a lot of effort into perfomance optimization such that there are many parameters to adapt it to special requirements[4]. In the assumption that most middleware users are not aware of such settings the Linux kernel configuration were left at the default.

All plots show how much data can be transmitted at most for each cycle time. All tests conducted compare communication using UDP and SHM. For all diagrams the x-axis contains the cycle time in Milliseconds. The y-axis contains the maximum size of UInt32 data arrays that could be transmitted as data packets on a test run.

3.1 Test with Dominion



(a) (b)

Fig. 2: Results using dominion at 0.05 CPU and 0.1 CPU

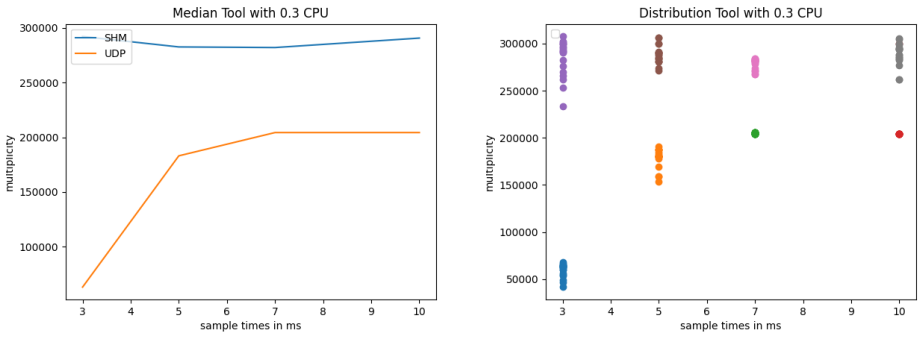
In figure 2(a) the data cycle time is in the range of 1 ms to 7 ms. To estimate the range of the y-axis scale low cycle times at a low CPU usage of 0.989 % were tested. This corresponds to 0.05 CPU for any hardware using 6 cores, since 0.0989 % of 6 is 0.05. All resulting values seem to have a rather low deviance and end up to inherit a steady growth.

In figure 2(b) the resulting values for testing at higher cycle times and CPU usages are plotted. The cycle times range from 8 ms up to 14 ms on the x-axis, while the multiplicity on the y-axis is significantly higher than in the previous test. Using about 1.978 % of the overall available CPU usage this was expected behaviour.

3.2 Test with transmission tool

To create a new environment that allows to test the data throughput without further optimization a communication tool was implemented in C++. With this tool, similar experiments were conducted as with the Dominion middleware, having the tests' sucess-condition deducted from Dominion's internal mechanics. Further detail into this is given by Algorithm 3.

To get an overview Fig. 3 shows the median values of all data points in this scenario. The test results in figure 3(a) show that UDP's maximum possible data throughput is increasing as the cycle time along the x-axis increases. SHM's values do not deviate as much but are all significantly higher than the ones using the UDP results. In figure 3(b) the distribution of each resulting data point is shown. All values above 200 000 multiplicity are results produced using SHM, while thoose below belong to UDP's tests.



(a) (b)

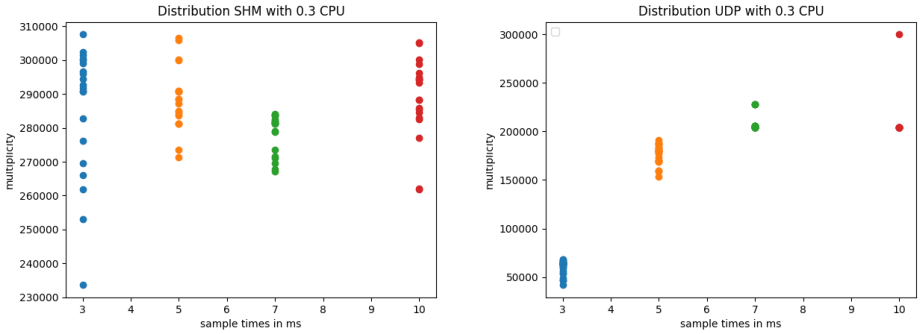
Fig. 3: Results tested with transmission tool at 0.3 CPU

The distribution of SHM's values is shown in figure 4(a) and differs in a range of about 40 000. UDP's value distribution is shown in figure 4(b) and behaves in a similar matter except for higher cycle times. At 10 ms the deviance is consistently 100 000. For all values below, the deviance is as small or smaller than the respective deviance of SHM's values.

To ensure that this is not caused by the CPU restrictions, further tests were conducted with 1.5 CPU, which is the maximum capability of a Docker container.

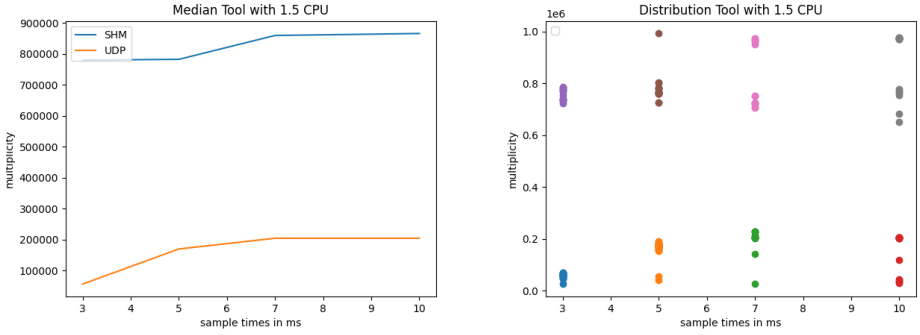
In Fig. 5, the CPU usage restriction mainly affected the data throughput while using SHM. Here, the highest possible multiplicity is more than double the size as before, while UDP's data throughput has not changed significantly. Another aspect to notice is that at cycle times above 6 ms SHM is nearly always fast enough to reach the limit set for this experiment.

With a higher CPU usage SHM's results follow a similar distribution to UDP. The deviances are both in a range of about 200 000 although SHM has overall higher data throughput.



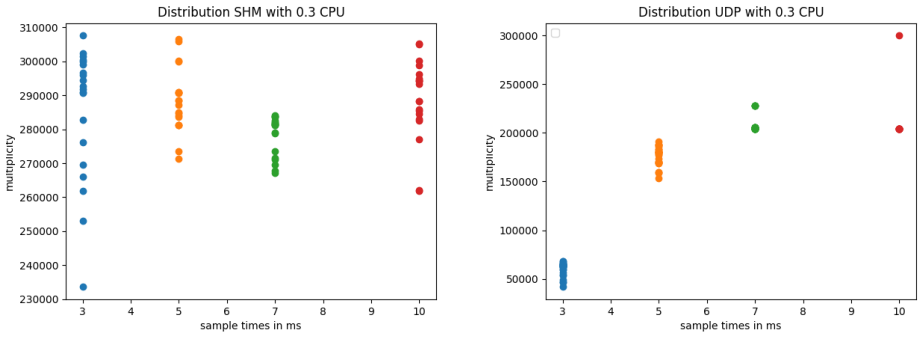
(a) (b)

Fig. 4: All data points gathered for each method at 0.3 CPU



(a) (b)

Fig. 5: Results tested with transmission tool at 0.3 CPU



(a) (b)

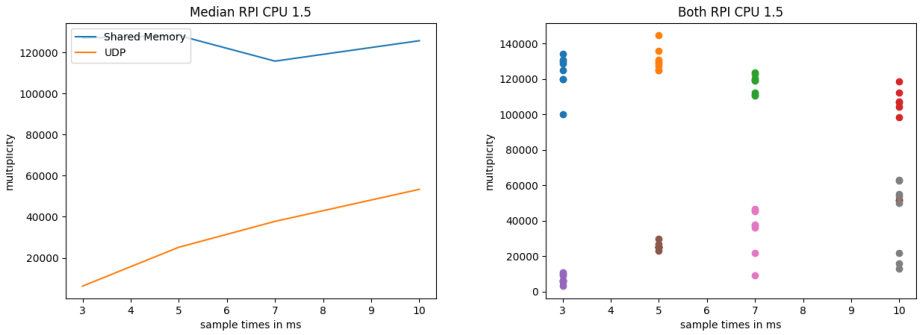
Fig. 6: All data points gathered for each method at 0.3 CPU

To further ensure the results were not mainly caused by kernel optimization or the chip architecture, the same tests were conducted on different hardware.

3.3 Test with Raspberry Pi

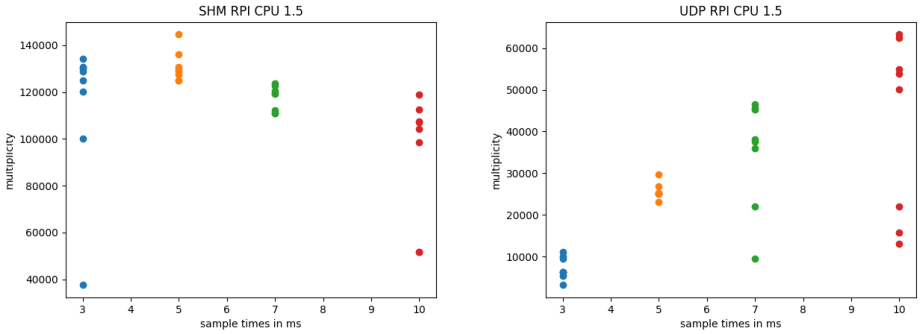
Because the RPI only has 4 cores and in general is less performant than the NB, maximum CPU usage was used for all tests done on RPI.

The data throughput of the RPI was almost similar for both methods. Although SHM has generally a higher throughput than UDP. This shows that the results gathered are replicable even on another chip architecture.



(a) (b)

Fig. 7: Results tested on RPI at 1.5 CPU



(a) (b)

Fig. 8: All data points gathered for each method at 1.5 CPU

4 Discussion

4.1 Discussion of the test results

The hypothesis of this paper is that SHM communication is way more efficient and reliable than using UDP for data transmission in a middleware. Testing with Dominion showed that UDP's data throughput capacities for low cycle times are significantly lower than for higher cycle times. Given this behavior the first tests with the communication tool were conducted with a lower CPU usage. The results using UDP were similar to those seen in the tests using Dominion. The results with SHM approved the initial hypothesis and exceeded the maximum possible data throughput of UDP consistently. An interesting aspect to notice is that at cycle times around 10 ms the deviance in performance between the two was rather small, which indicates that there are cases in which the data throughput of SHM is nearly identical to the maximum capabilities using UDP. Looking at the general deviance of the test results for UDP and SHM at low CPU usage showed that the average deviance for SHM was higher than for UDP except for tests at 10 ms cycle time. This implies that the general reliability of transmissions using UDP is higher than using SHM for most cases, which falsifies our initial hypothesis. Ensuring this is not just an edge case a new set of tests were conducted without the CPU limitation to see the influence on reliability and quality of transmissions.

Comparing both communication methods with the highest available CPU usage supported the initial hypothesis by showing a clear difference in the maximum possible data throughput. In terms of reliability the distribution pattern of both methods look similar. The deviance in values of SHM's results are higher than the deviance of UDP's. The cases in which the result deviated for SHM always ended up at a higher maximum data throughput than the average result. For UDP the results show that especially for higher cycle times deviances more often are smaller than the other data points at this cycle time, which leads to a saturation in the median graph.

The final set of tests conducted on the RPI verified that SHM is capable of achieving consistently higher data throughput. On different hardware the deviance of SHM's test results significantly decreased, while UDP's deviance behavior is similar to the tests conducted on the NB at low CPU usage. Although the deviance in results is higher with increasing cycle time while using UDP, the resulting values were higher than the average data throughput.

This shows that for low performance environments the advantage of using SHM instead of UDP gets smaller as the cycle time gets higher. For environments with higher CPU capabilities SHM outperforms UDP.

4.2 Conclusions

The data gathered suggests that there might be a break even point where the data throughput of SHM and UDP are consistently the same or—in low performance environments—UDP's throughput is even higher than SHM's, which requires further research.

Given that the general data throughput of SHM is consistently higher than UDP's the implementation of SHM communication can be beneficial. Having both options for transmissions ensures to get the most reliable and most efficient data throughput possible, particularly for low transmission cycle times.

Other research [2] supports this new hypothesis by showing the same behaviour in data loss and data throughput capabilities over IP. This indicates that the way UDP is build defines it's use case for the most part. Furthermore different implementations of SHM Algorithms can change the data throughput and performance of data transmissions. [5] Another aspect to notice is that parallelization of the network stack showed influence on TCP's performance [8], which could also be an influence on UDP's performance.

All those factors leave space for more research on which aspect have a measurable impact and if so how big is the impact on performance.

5 References

[1] A Venkataraman and KK Jagadeesha. *Evaluation of inter-process communication mechanisms*. 2015. URL: http://pages.cs.wisc.edu/~adityav/Evaluation_of_Inter_Process_Communication_Mechanisms.pdf (cit. on p. 1).

[2] D. I. Axiotis and D. Xenikos. "UDP Performance Measurements Over TETRA IP". In: *2007 IEEE 65th Vehicular Technology Conference - VTC2007-Spring*. Apr. 2007, pp. 1331–1335. DOI: 10.1109/VETECS.2007.279 (cit. on p. 12).

[3] Leslie Lamport. "On interprocess communication". In: *Distributed Computing* 1.2 (June 1986), pp. 77–85. ISSN: 1432-0452. DOI: 10.1007/BF01786227. URL: https://doi.org/10.1007/BF01786227 (cit. on p. 1).

[4] Michael Smith and Steve Bishop. *Flow control in the Linux network stack*. 2005 (cit. on p. 6).

[5] M. Stumm and S. Zhou. "Algorithms implementing distributed shared memory". In: *Computer* 23.5 (1990), pp. 54–64. DOI: 10.1109/2.53355 (cit. on p. 12).

[6] *Transmission Control Protocol*. RFC 793. Sept. 1981. DOI: 10.17487/RFC0793. URL: https://www.rfc-editor.org/info/rfc793 (cit. on p. 1).

[7] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://rfc-editor.org/rfc/rfc768.txt (cit. on p. 1).

[8] Paul Willmann, Scott Rixner, and Alan L Cox. "An Evaluation of Network Stack Parallelization Strategies in Modern Operating Systems." In: *USENIX Annual Technical Conference, General Track*. 2006, pp. 91–96 (cit. on p. 12).