PCD - H1 - Technical Report

Pasat Tudor Cosmin

[1. Introduction](#_1ekbzzrewn91)

[1.1. Homework objectives](#_vkc06xsb02bm)

[1.2. Implementation requirements](#_81nn46kbw91e)

[1.3. Output requirements](#_5ji2odoe5tgk)

[2. Experiment](#_bauhry2gyihm)

[2.1. TCP](#_jdnruavyswyz)

[2.1.1. Streaming](#_b2k13fhus0h2)

[2.1.2. Stop and wait](#_opr25s6vs0ss)

[2.2. UDP](#_8ju34kudlua1)

[2.2.1. Streaming](#_9kpoldqxcuvh)

[2.2.2. Stop and wait](#_1qr4qv1oswv1)

[3. Discussion](#_hvwu0ifb3u43)

[3.1. Local host](#_1b0tim2azq35)

[3.2. EC2 Instance](#_7v2dijep0h1b)

[4. Conclusion](#_shmipetw4xcl)

[5. References](#_yhgt5yxyxi9p)

# Introduction

This technical report explores the performance analysis of data transfer protocols, with a focus on UDP (User Datagram Protocol) and TCP (Transmission Control Protocol). The primary objective is to measure the time required for transferring varying amounts of data across different message sizes and transmission mechanisms.

## 1.1. Homework objectives

The task entails designing a program capable of measuring data transfer time for different message sizes (1 to 65535 bytes) and amounts of data using UDP and TCP protocols. Additionally, the program must support both streaming and stop-and-wait transmission mechanisms.

## 1.2. Implementation requirements

* **Supported Protocols:** UDP and TCP protocols must be supported as parameters for both client and server components.
* **Message Size:** The program should facilitate data transfer across a range of message sizes to assess performance implications.
* **Transmission Mechanisms:** Two transmission mechanisms, streaming and stop-and-wait, should be implemented for comprehensive analysis.

## 1.3. Output requirements

* **Server Output:** The server will print the protocol used, number of messages read, and bytes read after each session.
* **Client Output:** Upon completion, the client will display transmission time, number of sent messages, and total bytes sent.

# Experiment

The experiments were done on a local host, using a laptop with Intel Core I5 processor and MacOS Ventura 13.4.1. I ran all the pairs, protocol and mechanism, 100 times on local host and 10 times on an EC2 instance from AWS, and computed the minimum, maximum and the average values for each of the statistics presented in section **1.3**.

For the stop and wait mechanism, I added to the client a timeout of 10 seconds. If he doesn’t receive the acknowledgement until the timer runs out, the client will not move to the next package and will try to send the same one until the acknowledgement is received.

## 2.1. TCP

I used for both of the mechanisms a buffer of 65535 bytes.

### 2.1.1. Streaming

| Type | Size | Messages | | | Bytes | | | Transmission time | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Min | Avg | Max | Min | Avg | Max | Min | Avg | Max |
| L  O  C  A  L | 500MB | 8001 | 8001.45 | 8036 | 524288000 | 524288000 | 524288000 | 0.18s | 0.22s | 0.5s |
| 1GB | 16385 | 16385.13 | 16388 | 1073741824 | 1073741824 | 1073741824 | 0.36s | 0.44s | 0.97s |
| 2GB | 32769 | 32769.39 | 32776 | 2147483648 | 2147483648 | 2147483648 | 0.63s | 0.96s | 2.64s |
| E  C  2 | 500MB | 24011 | 31204.7 | 44571 | 524288000 | 524288000 | 524288000 | 11.32s | 13.60s | 26.27s |
| 1GB | 56770 | 86617.7 | 103460 | 1073741824 | 1073741824.0 | 1073741824 | 23.63s | 26.53s | 29.92s |
| 2GB | 104484 | 153430.4 | 190772 | 2147483648 | 2147483648 | 2147483648 | 48.94s | 54.13s | 71.34s |

**Table 1**: TCPS Client statistics

| Type | Size | Messages | | | Bytes | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Min | Avg | Max | Min | Avg | Max |
| L  O  C  A  L | 500MB | 8156 | 8289.31 | 9675 | 524288000 | 524288000 | 524288000 |
| 1GB | 16533 | 16727.15 | 17229 | 1073741824 | 1073741824 | 1073741824 |
| 2GB | 33121 | 33692.17 | 39712 | 2147483648 | 2147483648 | 2147483648 |
| E  C  2 | 500MB | 48620 | 67516.4 | 105008 | 524288000 | 524288000 | 524288000 |
| 1GB | 120228 | 193105.5 | 239137 | 1073741824 | 1073741824 | 1073741824 |
| 2GB | 222792 | 349565.4 | 435726 | 2147483648 | 2147483648 | 2147483648 |

**Table 2**: TCPS Server statistics

### 2.1.2. Stop and wait

| Type | Size | Messages | | | Bytes | | | Transmission time | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Min | Avg | Max | Min | Avg | Max | Min | Avg | Max |
| L  O  C  A  L | 500MB | 8001 | 8001 | 8001 | 524288000 | 524288000 | 524288000 | 0.18s | 0.20s | 0.42s |
| 1GB | 16385 | 16385 | 16385 | 1073741824 | 1073741824 | 1073741824 | 0.38s | 0.48s | 1.29s |
| 2GB | 32769 | 32769 | 32769 | 2147483648 | 2147483648 | 2147483648 | 0.73s | 1.08s | 3.21s |
| E  C  2 | 500MB | 8001 | 8001 | 8001 | 524288000 | 524288000 | 524288000 | 485.19s | 487.54s | 488.89s |
| 1GB | 16385 | 16385 | 16385 | 1073741824 | 1073741824 | 1073741824 | 1070.96s | 1093.12s | 1105.93s |

**Table 3**: TCPSW Client statistics

| Type | Size | Messages | | | Bytes | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Min | Avg | Max | Min | Avg | Max |
| L  O  C  A  L | 500MB | 8070 | 8119.47 | 8240 | 524288000 | 524288000 | 524288000 |
| 1GB | 16542 | 16716.57 | 17210 | 1073741824 | 1073741824 | 1073741824 |
| 2GB | 33013 | 33374.60 | 34246 | 2147483648 | 2147483648 | 2147483648 |
| E  C  2 | 500MB | 58811 | 81002.33 | 116296 | 524288000 | 524288000 | 524288000 |
| 1GB | 105339 | 177299.66 | 219937 | 1073741824 | 1073741824 | 1073741824 |

**Table 4**: TCPSW Server statistics

## 

## 2.2. UDP

I used for both of the mechanisms a buffer of 9216 bytes on the local host. For the tests on EC2, I tried using different values for the buffer, ranging from 1000 to 65535 and finally I found that 1000 is the value that is the best to use in order to minimise the number of data loss, on the streaming mechanism and 60000 on stop and wait.

### 2.2.1. Streaming

| Type | Size | Messages | | | Bytes | | | Transmission time | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Min | Avg | Max | Min | Avg | Max | Min | Avg | Max |
| L  O  C  A  L | 500MB | 56889 | 56889 | 56889 | 524288000 | 524288000 | 524288000 | 0.76 | 1.16 | 0.92 |
| 1GB | 116509 | 116509 | 116509 | 1073741824 | 1073741824 | 1073741824 | 1.66s | 1.91s | 3.45s |
| 2GB | 233017 | 233017 | 233017 | 2147483648 | 2147483648 | 2147483648 | 3.09s | 3.76s | 7.89s |
| E  C  2 | 500MB | 524288 | 524288 | 524288 | 524288000 | 524288000 | 524288000 | 14.85s | 15.07s | 15.60s |
| 1GB | 1073742 | 1073742 | 1073742 | 1073741824 | 1073741824 | 1073741824 | 30.49s | 30.64s | 30.86s |
| 2GB | 2147484 | 2147484 | 2147484 | 2147483648 | 2147483648 | 2147483648 | 61.14s | 63.09s | 64.61s |

**Table 5**: UDPS Client statistics

| Type | Size | Messages | | | Bytes | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Min | Avg | Max | Min | Avg | Max |
| L  O  C  A  L | 500MB | 56738 | 56883.16 | 56889 | 522896384 | 524234178.56 | 524288000 |
| 1GB | 116047 | 116501.64 | 116509 | 1069484032 | 1073673994.24 | 1073741824 |
| 2GB | 231733 | 232985.95 | 233017 | 2135650304 | 2147197491.2 | 2147483648 |
| E  C  2 | 500MB | 509366 | 512014.4 | 514833 | 509366000 | 512014400.0 | 514833000 |
| 1GB | 865633 | 926466.2 | 987946 | 865632824 | 926466024.0 | 987945824 |
| 2GB | 1891844 | 1935846 | 2109622 | 1891843648 | 1935845648 | 2109621648 |

**Table 6**: UDPS Server statistics

### 2.2.2. Stop and wait

| Type | Size | Messages | | | Bytes | | | Transmission time | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Min | Avg | Max | Min | Avg | Max | Min | Avg | Max |
| L  O  C  A  L | 500MB | 56889 | 56889 | 56889 | 524288000 | 524288000 | 524288000 | 2.02s | 2.18s | 3.11s |
| 1GB | 116509 | 116509 | 116509 | 1073741824 | 1073741824 | 1073741824 | 4.18s | 4.53s | 7.35s |
| 2GB | 233017 | 233017 | 233017 | 2147483648 | 2147483648 | 2147483648 | 8.35s | 9.37s | 13.34s |
| E  C  2 | 10  MB | 274 | 280.66 | 290 | 16397280 | 16806773.33 | 17371520 | 190.79s | 202.12s | 218s |

**Table 7**: UDPSW Client statistics

| Type | Size | Messages | | | Bytes | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Min | Avg | Max | Min | Avg | Max |
| L  O  C  A  L | 500MB | 56889 | 56889 | 56889 | 524288000 | 524288000 | 524288000 |
| 1GB | 116509 | 116509 | 116509 | 1073741824 | 1073741824 | 1073741824 |
| 2GB | 233017 | 233017 | 233017 | 2147483648 | 2147483648 | 2147483648 |
| E  C  2 | 10MB | 176 | 176 | 176 | 10531520 | 10531520 | 10531520 |

**Table 8**: UDPSW Server statistics

# Discussion

## 3.1. Local host

We can see from the tables above that the UDP protocol, using the streaming mechanism, is the worst when considering the number of bytes that actually get to the server. There are packages of bytes lost even for the smaller amounts of data.

As for the other 3, all the data gets to the server, but the transmission times are significantly different, between the 2 TCP programs and the UDP one. This is mostly because I used a much smaller buffer size for the UDP, in order to avoid getting more expensive getting lost.

The TCP with stop and wait mechanism has a higher latency than the streaming version, because the client needs to wait for acknowledgement before sending the next package.

## 3.2. EC2 Instance

* TCP with Streaming - unlike the other methods, just worked when I run it on cloud, without any modification. We can see from **Table 1**, that compared to the local host, the transmission time is significantly higher. This happens because the communication goes through the internet and there are several more factors that intervene, like the reliability of the network, the bandwidth, which is never unlimited, the latency, which is never 0, etc.
* TCP with Stop and Wait - for this mechanism I modified the code a little bit, because socket library implements the TCP communication and write to a buffer from where we, as users of this library, read the messages, we can get just a piece of the package in one message, the rest being read the next iteration. Because of this, I added to send the acknowledgement only when the entire package is sent, which, we can see from **Table 3**, affects the transmission time, going from 13 seconds on average for the streaming mechanism, to 487 seconds, for this one, on 500MB of data. This shows that the latency is a lot higher for this one, proving again that it is never 0.
* UDP with Streaming - because the bandwidth isn’t unlimited for the network communication with the cloud, I had to add a sleep for the UDP with the streaming mechanism version of 0.1 seconds, every 6000 packages sent, otherwise an exception is thrown, because the buffer is full. Also, as I mentioned before, I had to use a package size of only 1000 bytes, in order to assure that the data loss was minimal. Although this configuration increases the transmission time, because the number of packages is higher and the time spent sleeping. In the end, the transmission time is actually a little higher than the one of TCP streaming and we don’t have the reliability of TCP
* UDP with Stop and Wait - this mechanism brought the most problems. Since it’s a UDP connection, there is a probability to lose both the packages sent by the client and the acknowledgment sent back by the server and at some point the client and the server are both stuck waiting for messages from each other To solve this I added a timeout on the client side for it to receive the acknowledgement. If it expires it tries to send the same package again. Also on the server side I computed the hash of the current package received and compared it to the last one. If they are the same it assumes that the acknowledgement was lost and the same package was sent, and just sends the same message back, without saving the package. Since there is this retransmission method, I tried using packages of different sizes, finally using the 60000 value. Unfortunately, all these new steps that need to be done increase the transmission very much, taking 200 seconds to send only 10MB of data. Most of the time is spent waiting for responses that never actually come and end up with a timeout exception and retransmission of the package.

# Conclusion

When using a remote server, there are fallacies of distributed computing that need to be taken into consideration. As we saw from the experiments, we can’t assume the bandwidth is unlimited or that the latency is zero. Also on the local host I was the only administrator, meanwhile when using the cloud, the data goes through different subnets, until it gets to my implemented server. Also, I can only control the network topology in my home, outside there can always be changes in the structure of the network and as a result change the experiment's outcomes.

In the end, we can say that one protocol or mechanism is better than another. All have their advantages and disadvantages, and we need to analyse the use case to find which is best suited for it.

# References

* [https://profs.info.uaic.ro/~adria/teach/courses/pcd](https://profs.info.uaic.ro/~adria/teach/courses/pcd/homework/PCD_Homework1.pdf)
* <https://profs.info.uaic.ro/~adria/teach/courses/pcd/homework/PCD_Homework1.pdf>
* <https://docs.python.org/3.11/library/socket.html#module-socket>
* <https://pythontic.com/modules/socket/introduction>
* <https://pythontic.com/modules/socket/udp-client-server-example>
* [https://forums.developer.apple.com/forums/thread/74655#:~:text=By%20default%20macOS%20has%20limited,following%20command%20in%20the%20terminal.](https://forums.developer.apple.com/forums/thread/74655#:~:text=By%20default%20macOS%20has%20limited,following%20command%20in%20the%20terminal)