PCD Homework 1

Smoc George

March 2024

1 Introduction

The requirement is: Write a program that measure the time to transfer various amount of data (500MB, 1GB (1.5 milion of WhatsApp messages, 4000 photos, 10.000 emails or reuse a large buffer)) under various conditions (Make sure you are sending bytes. Do not use higher-level functions that assume you are writing strings.)

2 Possible options

The project's configuration is done via arguments passed to client and server scripts. Each of main_server.py and main_client.py implements –help command using argparse module, with a brief description of every possible argument. Example of output:

3 Structure

Project is separated into decoupled packages that only have as common interface the datatypes defined in utils package.

Package Name	**Description**
client	Abstract client and its implementations for both tcp and udp
server	Abstract server and its implementations for both tcp and udp
utils	Exception handling wrappers, data and tests generator, structures for ack and tcp/udp response

Table 1: Packages Structure

Generic implementations of both client and server are implemented to facilitate understanding of flows.

```
usage: main_server.py [-h] [--host HOST] [--port PORT]
                      [--package-size PACKAGE_SIZE] [--tcp] [--udp]
                      [--stop-and-wait] [--store-answers]

Server IP and Port Argument Parser

options:
  -h, --help            show this help message and exit
  --host HOST           Server IP address
  --port PORT           Server port number
  --package-size PACKAGE_SIZE
                        Size of a packet sent via network
  --tcp                 Use tcp for data transfer. Cannot be used together
                        with --udp
  --udp                 Use udp for data transfer. Cannot be used together
                        with --tcp
  --stop-and-wait       Ack for every message before next message
  --store-answers       Store answers on disk
```

Figure 1: Package size 256

4 Client Flow

Abstract client serves as both a pipeline for the existing samples and a template for the sub-operations that will be implemented accordingly in sub-classes based on the used protocol. An initial connection is done (for connection oriented protocols) or any other kind of protocol-related initialization, then data is loaded one by one with a generator, obtaining a descriptor and metadata to be later used in order to create any needed headers. For each sample, __send_file is called, which should also have a subclass implementation.

4.1 TCP Client Details

Headers with filename, file size and packages number are first sent before transferring a file, then each chunk is sent, without repeating the aforementioned headers. Each packet will be resent in case of –stop-and-wait options

4.2 UDP Client Details

Initially sent headers are DatagramType - needed, because order is not guaranteed, filename, file index and file size. Also, each message has a crc in order

to validate the integrity of the message. The messages with the actual data contain as headers DatagramType - for the same reason mentioned above, file index (in case initial headers were not received, a default name will be provided based on index, file size, packages no and packages index).

5 Server Flow

Abstract client serves as a template for the sub-operations that will be implemented accordingly in sub-classes based on the used protocol. Because of differences between connection oriented and connectionless oriented protocols, no default implementation is done in receive function

5.1 TCP Server Details

Bind and listen, followed by receiving data in the order provided bt the tcp client. As data order is guaranteed, flow could be replicated on the server.

5.2 UDP Server Details

Bind and listen, followed by receiving data and handling chunk according to its type (initial header - will map file index to filename, chunk - will put it in a map from file index to its chunks, end message - will stop the server).

6 Results

The following results were obtained on 10 runs for each configuration. Tests were done with 1000 random generated messages. 800 MB were sent for each test.

Protocol	Stop and Wait	Store	Client Time	Server Time	Client Packets	Server Packets
TCP	NO	NO	63.28	63.79	3228501	3228501
TCP	YES	NO	145.08	145.08	3228501	3228501
TCP	NO	YES	63.29	69.39	3228501	3228501
TCP	YES	YES	150.62	151.95	3228501	3228501
UDP	NO	NO	37.636	37.635	3228139	3228139
UDP	YES	NO	167.023	167.022	3228501	3228501
UDP	NO	YES	37.323	343.821	3228501	3215468
UDP	YES	YES	149.491	452.521	3228501	3228501

Figure 2: Package size - 256 - Bytes Sent and Times

Protocol	Client Bytes	Server Bytes	Packet Loss	Store Time
TCP	826240005	826240005	0	0
TCP	826240005	826240005	0	0
TCP	826240005	826240005	0	0.305
TCP	826240005	826240005	0	0.274
UDP	912595966	912493524	0.012	0
UDP	912595966	912595966	0	0
UDP	908905744	912595966	0.404	306.469
UDP	912595966	912595966	0	302.99

Figure 3: Package size - 256 - Packet Loss

Protocol	Stop and Wait	Store	Client Time	Server Time	Client Packets	Server Packets
TCP	NO	NO	12.588	12.588	808256	808256
TCP	YES	NO	33.025	33.025	808256	808256
TCP	NO	YES	13.734	13.801	808256	808256
TCP	YES	YES	13.791	13.827	808256	808256
UDP	NO	NO	10.886	10.887	808256	800437
UDP	YES	NO	41.071	41.070	808256	808256
UDP	NO	YES	10.850	311.190	808256	790883
UDP	YES	YES	43.665	346.522	808256	808256

Figure 4: Package size - 1024 - Bytes Sent and Times

Protocol	Client Bytes	Server Bytes	Packet Loss	Store Time
TCP	826240005	826240005	0	0
TCP	826240005	826240005	0	0
TCP	826240005	826240005	0	0.276
TCP	826240005	826240005	0	0.289
UDP	846481502	838276043	0.969	0
UDP	846481502	846481502	0	0
UDP	846481502	828255414	2.153	300.33
UDP	846481502	846481502	0	302.851

Figure 5: Package size - 1024 - Packet Loss

7 Conclusions

1. Udp:
+ Obtained better results without storage and stop and wait options - from 30 to 90 percent faster based on package size
- Headers size increases significantly, as reordering of packages/chech of crc32 is needed
- Reordering packages means more RAM usage and needs expansion of current acritecture with a Thread that handles I/O component
- Stop and wait version works slightly worse than tcp stop and wait
- Timeout is needed and exception handling for this case

2. Tcp:
+ File integrity is kept
+ Writes can be done after each response, resulting in simpler architecture (no extra Thread is needed) and faster writes
+ Less RAM usage in case of storage
+ Files can always be mapped to their names (headers with this information always arrive) + Specific order of messages means knowing the exact size of next message
- Slower as the package size decreases, but manages to be only 30 percent slower when package size if 1024
- Stop and wait needs an extra message that also require an ACK (equivalent to sending an ACK for an artificial ACK created for this case)