Networking Assignment 2024

Final: updated link and date

Assignment description and communication scenario

It is frequently necessary to customize a protocol to fit the existing platform or to develop a new one when initiating a new project.

In this instance, we will request that you construct a client and a server. They will communicate using a custom protocol inspired by the slow start mechanism in TCP.

We require you to implement a slow start congestion control mechanism over a UDP connection. By adhering to our guidelines, you will notice how this differs from the traditional version and lacks several additional features.

The client will ask for a resource (like a simple FTP), and the server will wait for the request and provide it to the client split into multiple data chunks.

Both the client and server must always handle issues accordingly and terminate in a controlled way. There must be no unhandled exceptions.

The protocol

The protocol works as follows

1. The client sends a *Hello*-message to the server, including the threshold for the slow start in the content section.
2. The server replies with a *Welcome*-message
3. The client sends a *RequestData*-message
4. The server starts to send *Data*-messages starting with 1 message (initial size) and waits for the *Ack*-message (within a short timeout)
    - The Data ID is included in the message content. Check the format of an ID in the given example.
5. The client receives each *Data*-message and confirms it with an *Ack*-message.
    - The received *Data ID* from the message is included in the content section of the *Ack* message.
6. The server now has two options:
    - [Option 1: happy flow] The ~~client~~ **server** receives an *ACK* message for all messages sent *(within the short timeout)* and **doubles** the number of messages to be sent next.
        - Check further details (threshold usage) in the description.

    - [Option 2: sad flow] The ~~client~~ **server** receives part of the acknowledgements or no *Acks (within the short timeout)* and restarts slow start, resetting the number of packets sent per time, starting from the last received confirmation Ack.
7. When the server has completed, sent the last data, and received all the *Acks*, will send a *End*-message.
8. The client will terminate the activity once the *End*-message is received, or a long timeout is reached with no activity received.
9. The server will not terminate, once the End is sent, will stay on to get the next client.

The supported messages.

```
public class Message
    {
        public MessageType Type { get; set; }
        public string? Content { get; set; }
    }
    public enum MessageType
    {
        Hello,
        Welcome,
        RequestData,
        Data,
        Ack,
        End,
        Error,
    }
```

Structure of the messages:

Every message will have a type and a content field. Types are taken from the enumeration.

Hello will be the first message that the client sends to the server. It will contain the threshold decided by the client in the *content*. The value is a number (converted into a string), and the default value will be “20”.

Welcome is sent only by the server in reply to the *Hello* message, this will be the first message received by the client. The *Welcome*-message will have no content.

RequestData is the message sent by the client to the server to request a resource. The resource will be the name of a file present in the root directory of the project, "*hamlet.txt*" is the file name used for this assignment. The *content* will be the string of the name of the resource.

Data is the actual message containing the information (divided into "chunks") fetched from the server. Think of and test an appropriate size (a number of characters) to be sent in one message, this will always be the same value beside the last message, which might contain less than that. The number of messages will be defined by the "slow start algorithm", doubling every time the number of messages. The server will double the number of messages until the threshold is reached, if the calculated new value is more than the default value, the server will stop doubling the value and continue to send the last known amount. The *Data content* has the following structure:

> the first 4 characters are a number referring to the index of the data (ID number of message sent from the server), and the rest (up to the decided size) will be the content of the information read from the resource. Ex.: "0001Act I\n ...", red is the ID and blue is the text taken from the text file.

Ack is the confirmation sent by the client to the server for each received *Data*, the *content* is the numerical ID shown in the previous example (see the red text).

The *Error* message is there to communicate to either of the other party that there was an error, please be specific about what is the error in the content of the message. Upon reception of an error, the server will reset the communication (and be ready again). The client will terminate printing the error.

The *End* has no content and marks the final message after the last *Data* sent.

Welcome, *Data*, and *End* can be only sent from the server.

Hello, *RequestData* and *Ack* can be sent only by the client.

The client

The client will implement minimal logic and must collect all the file parts in order in a file without replication (a.k.a., recreate the same file requested in the project home of the client).
Suggestion: use a proper data structure to store all the received *Data* chunks temporarily.

Suggestion: The client is expected to maintain the number of messages received (*Data* messages) to discover missing numbers and check for doubles.

Missing a *Data* message is not an error to communicate to the server. It will not communicate errors unless the format is wrong or other critical errors are found.

The client replies with the *Ack* for **each** received *Data* message.

The server

The server will communicate with only one client at a time; it cannot communicate with multiple clients simultaneously; they must interact in sequence.

The server will retrieve the content of the text file requested from the main project directory and dispatch it upon request. It will transmit the dictated number of *Data* at once according to the slow start algorithm as described before (1, 2, 4, 8 ...), without pausing for the *Acks*. For each *Data* sent, the server must keep track of what has been sent and of how many messages have been sent on the last round (see the algorithm from before), this will be useful for the next step. Following the transmission, it will collect all the pending incoming messages (*Ack*s) and verify if any are missing by confronting them with the *ID* of the sent messages (*Data*). If not all the *Acks* are retrieved after a **short** timeout, the protocol will be reset, and it will restart from the first missed *Ack*.
Unless an error is received (wrong format, etc...), the server will continue to send and check for *Acks.*

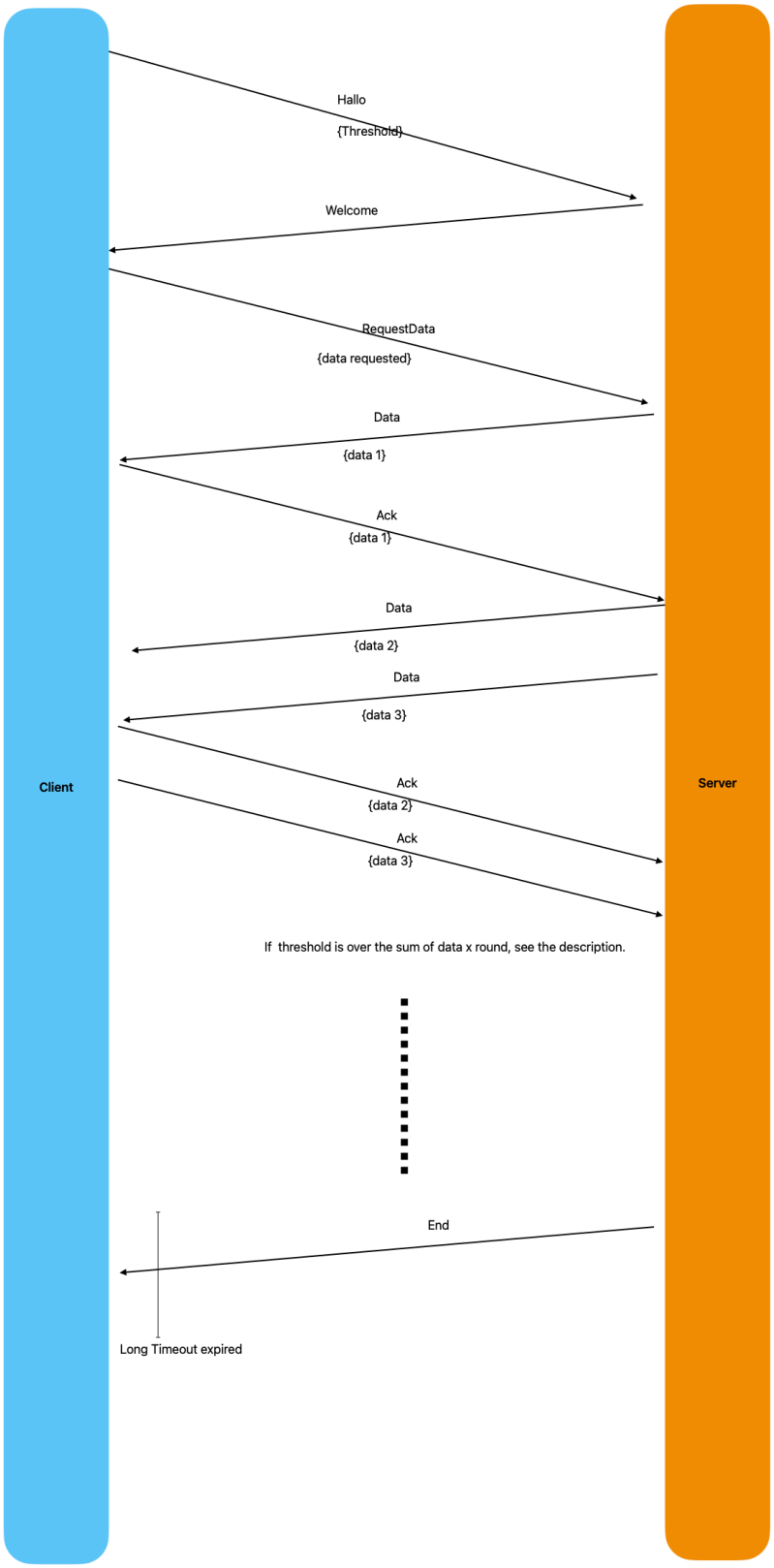
Missing *Acks* are not errors to communicate.

The server will always stay online after terminating the operation waiting for a new Hello.

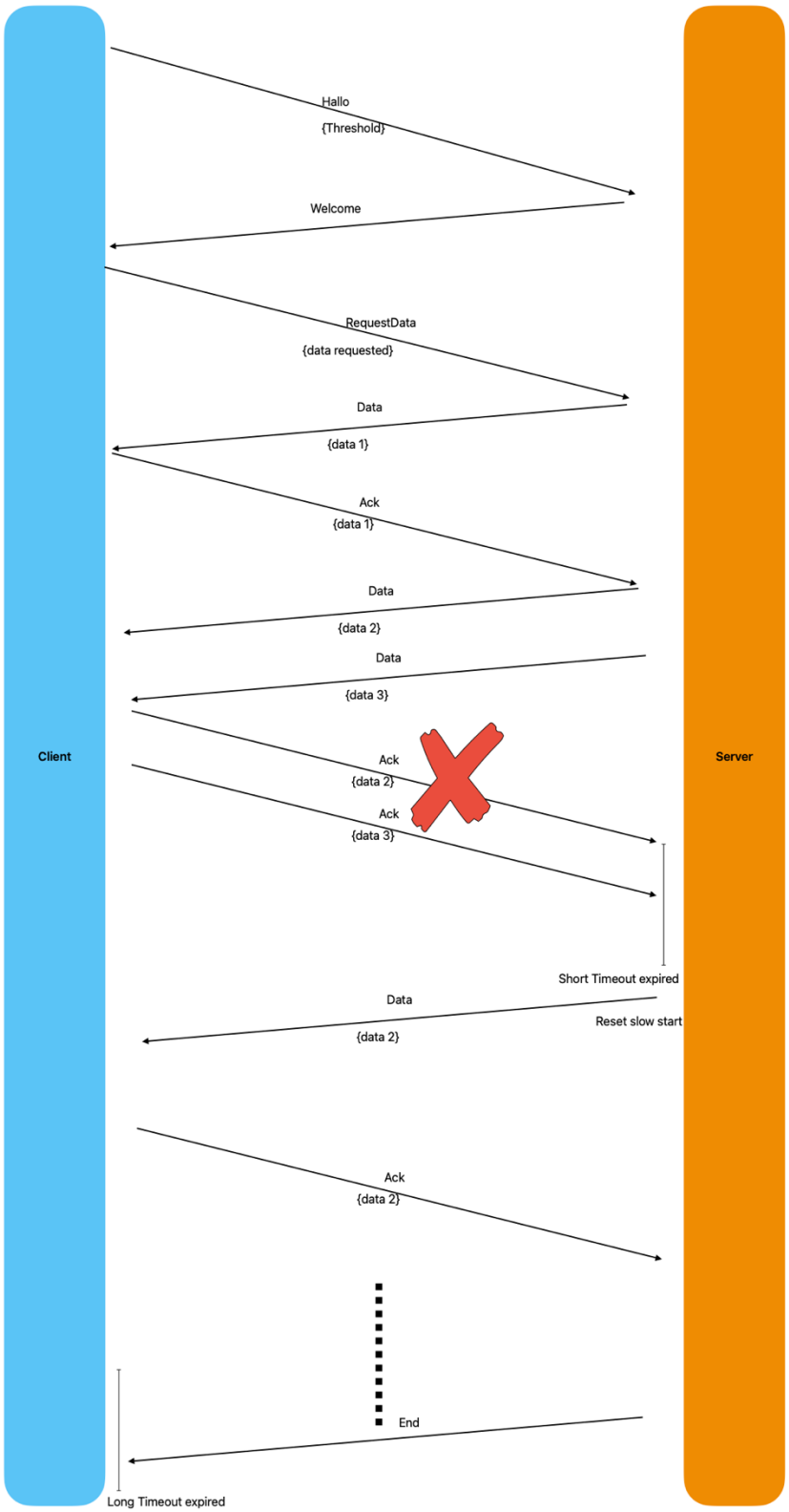
Ports: the server will wait on port 32000.

More details...

Short timeout: should be used for acknowledgements check and be at least 1 second (change this value if your machine is slow but leave it at 1 second in the delivery).
Long timeout: should be used to terminate activities in case something else has gone wrong and no activity is detected, should be at least 5 seconds.
IP addresses: should be automatically retrieved by your code when we run your application. Check the C# documentation on how to automatically retrieve IP-addresses.



In the above picture: the happy flow.



In the above picture: an example of sad flow where one of the acks was not received in time.

Grading requirements

Any missing, imprecise, partial, or incomplete requirement will be graded as a failure. Other requirements details can be extracted from the description.

To have a sufficient grade:

- The implementation must be compatible with the message data structure we provide.
- The implementation must handle all types of errors
    - issues with the socket creation
    - wrong or incomplete message. Ex.: when missing an expected value, in a received message
    - The client or server is sending the wrong message orders (for example: expected *RequestData* but received *Welcome)*
    - Others ...
- The implementation must use a low-level UDP socket.
- The implementation resembles the intended behaviour of the custom slow-start protocol.
- The implementation resembles the intended fail-save behaviour when an acknowledgement is missing.
- **The implementation of both client and server must print out (nicely formatted) messages about the data it sends and receives and the actions** it conducts during processing. For example, by printing the Ack number every time one is received and printing the ID of Data(s) messages when they are sent.
- The program should be written on .NET 6.*x*.
- No external libraries are allowed.
- Built-in libraries such as System and System.IO are allowed if it does not conflict with the learning goals. Other libraries are those that are NOT involved in the networking and handling of messages (ex: no “UdpClient”, nor “UdpListener”, or any other that avoid port binding and automated message handling) and are part of .NET can be allowed.
- No installation of new libraries or EXTRA FILES will be necessary to run the application.
- There is NO NEED FOR THREADS, the application will be synchronous (NOT CONCURRENT).
- When running, the application must never ask for user input.
- No need for external configs unless already in the template.
- You need to use our project template to implement your application. Do not change any existing filename in the provided template AND FILL IT APPROPRIATELY.
- When running the application there must be no errors/warnings. Apply proper exception handling where needed (failure to do so will result in a FAIL).

- All the communication between the programs must be carried out with the proper socket kind with appropriate settings, wrong settings will be regarded as a syntax error in the code (ex: wrong kind of socket, ex: TCP, wrong port communication settings, etc.).
- Do not write all the logic in one method, for example, "*start()*".
- Any path/location should be handled independently from the running operating system (Mac/Windows/Linux).
- The solution should be the original work of the submitting group (an automatic check for plagiarism will be carried out).
- We are explicitly requiring that no autogenerated code be used in this assignment (no ChatGPT or other AIs).
- The submitting group can be composed of up to 2 students with the same teacher. Retakers can pick any class/companion.
- The solution MUST include the names and student numbers of the participants as a comment in the template files and/or where indicated or fill in the right variables if they are indicated as placeholders for names.

Testing:

Your program will be tested using different sequences of errors in the protocols (anything that is a valid format but contains different values). It will be run from the shell with "dotnet run".

If files are needed, they will be in the project directory.

Write your own code to test edge cases, "forgotten" Acks and wrong timeouts.

The delivery of the assignment

The delivery is expected before **04/**05/2024 (11:30 pm).

Submission:

submit the same template folder with your solution in it as a zip file.
The zip file should have the following name format:
WILL BE PROVIDED AS SOON AS POSSIBLE

Link for the submission: **https://forms.office.com/e/Gs2v935Lxj**

Remarks:

In the code, there are comments to help your development.

If you find errors or unclear parts let us know. This includes description, code and/or delivery.

Ask per time. Asking later will not be useful. Feel free to give feedback if you think there is any imperfection or doubt (please do so in the appropriate manner and channel).