**Programming Project Phase 5:** **Implement Go-Back-N protocol over an unreliable UDP channel.**

Group Members: Isabella Leonard, Charles Undag, Isra Chokwatana

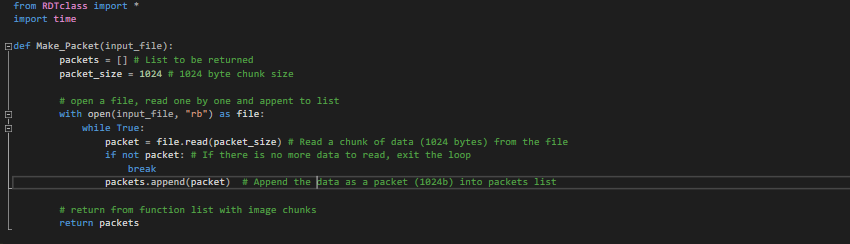
**Purpose**

The purpose of Phase 5 is to serve as an introduction to implementing a fixed window size, pipelined data transfer, and Go-Back-N protocols. This phase also builds upon the previous 4 phases by recycling some functions such as checksum and the general class structure. The goal is to send and receive packets from the client to the server through pipelined data transfer.

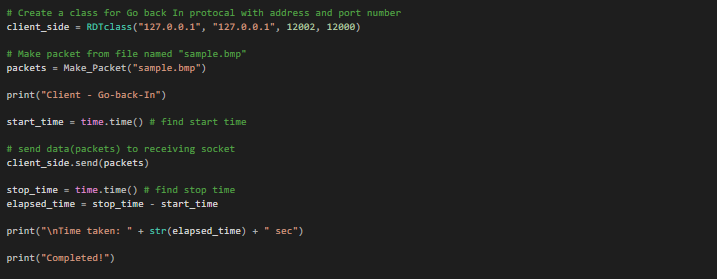
**Code Explanation**

There are 3 files that were used in this phase, being the client, server, and RDTclass file.

**Client.py:**

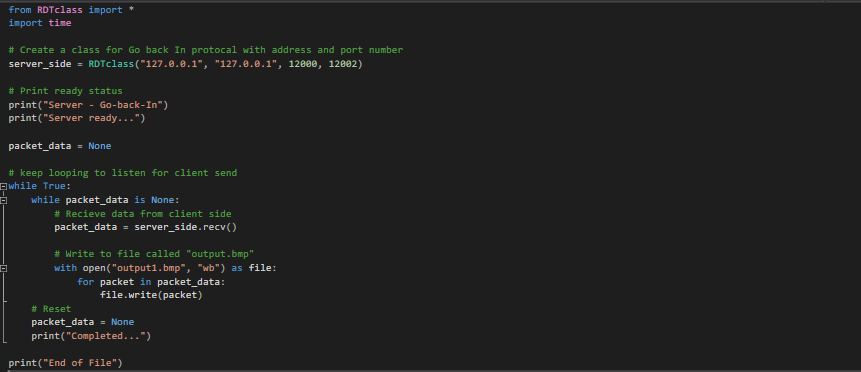


The first portion of the client code import necessary module then defines a function *Make\_packet*, this takes the bmp image (in this case, can also be JPG if needed) named “sample.bmp” and split the picture into 1024-byte chunks.



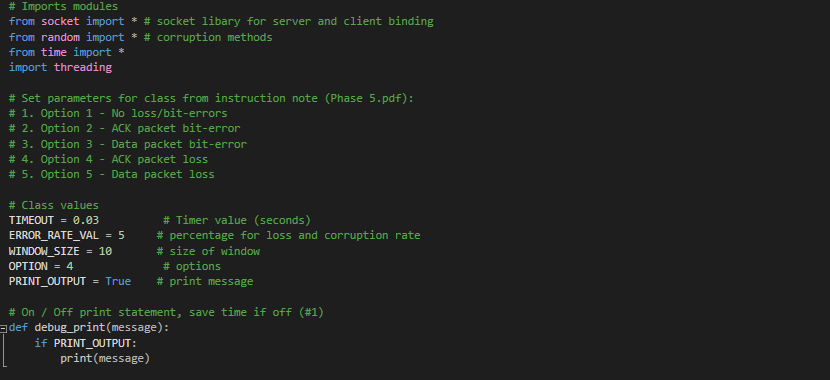
Using the *RDTclass*, technically Go-back-In class now, all argument is set, this creates the client bind. This is done by using the *Make\_Packet* function, then sending the chunk via the class method “*send(packets)*”, thus sending the packets to the server side. Time is used to keep track of the start and end time, giving us the total time elapsed for the client send.

**Server.py:**

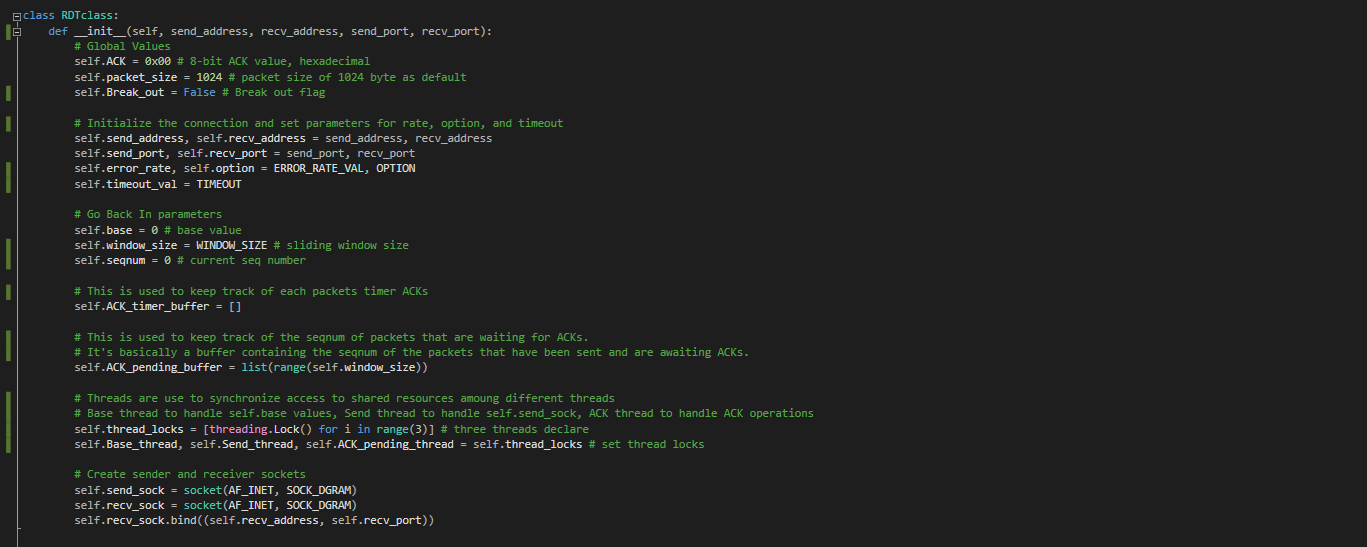


The first portion of the server code import necessary module then uses the class setting all arguments, this creates the server bind. The rest of the code creates a loop for the server side. The *RDTclass* (Go-back In) method “*recv()*” is used to receive any chunks coming from the client side. Data received is then written to a file named “*output1.bmp*” after which the inner for loop is then completed, ready for another receive.

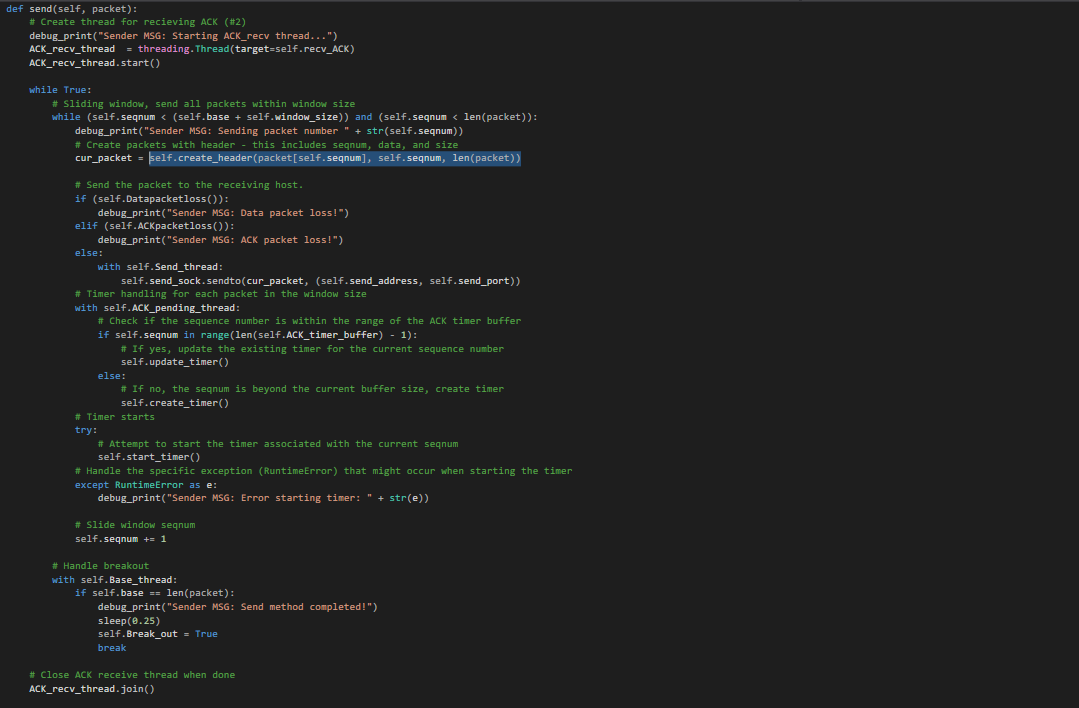
**RDTclass.py:**



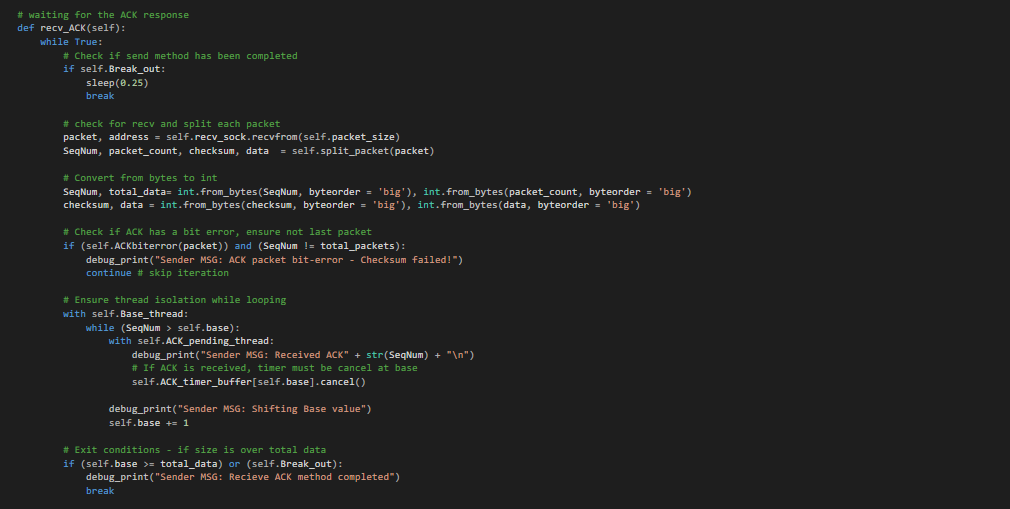
Class files start with the import needed being socket, random, time, and thread. Default values are set here, as such, changing options and all other parameters will be done here. A debug function is created to switch on/off the print statement within this code. This is required for a more accurate time elapsed reading.



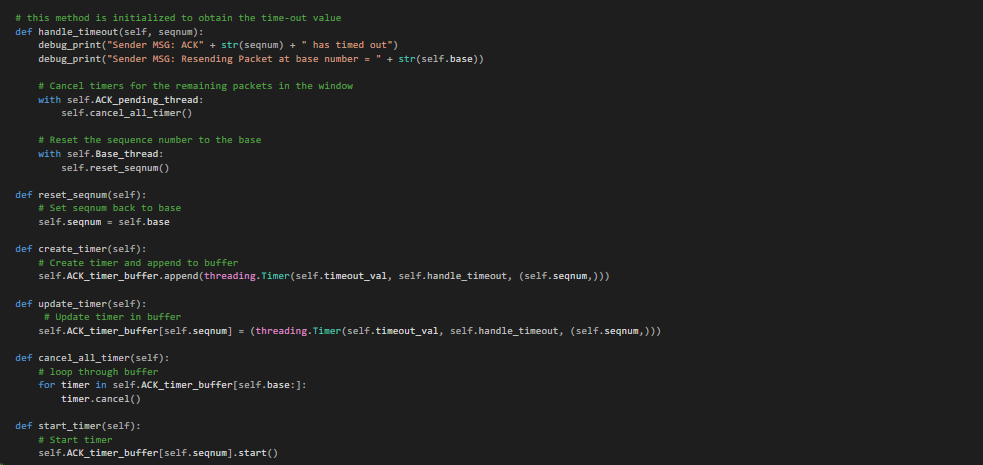
The standard class initialize is defined, being all the arguments that are necessary for the socket. The socket is then created and binded to the client/server, this ensures different port number for the two files.



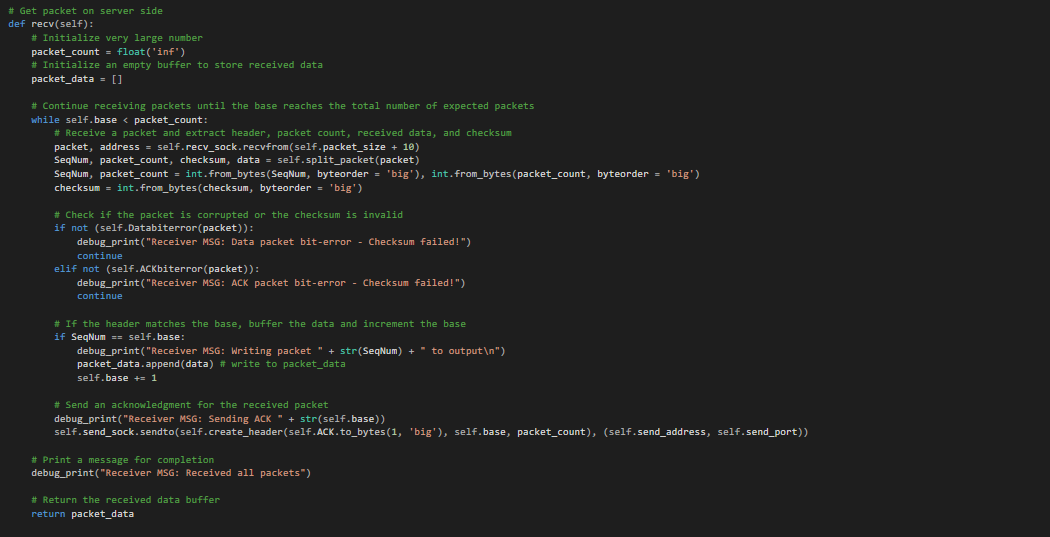
The send method is the method used by the client side, this is used to send packet (after creating a header) and receive *ACK*. A thread is used to start the ACK receive function, then a while loop starts to receive the *ACK.* Once *ACK* is received, the sliding window is shifted. Timer is also created and updated in this function, this is to ensure proper timeout in case of data/ACK loss.



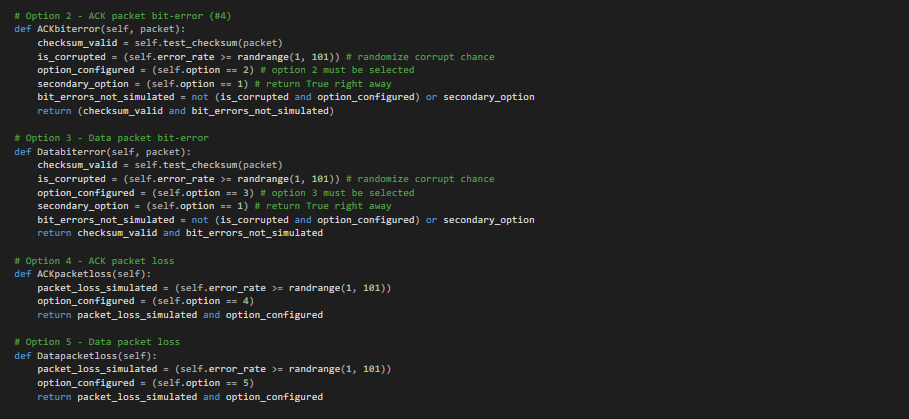
In the ACK\_*recv* method. As each ACK is received, each packet is unpack into four, after the error condition has been passed, the timer are then cancel to confirm ACK has been received. The global base values is then shifted to the next value.



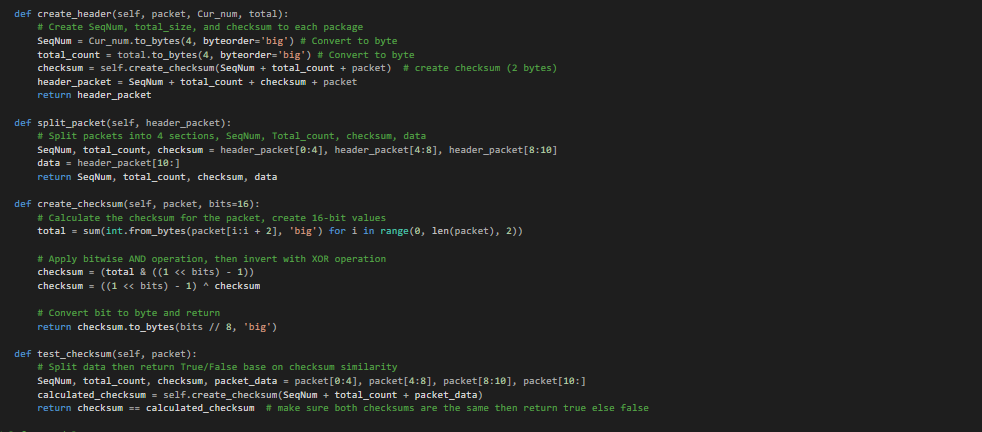
A timeout method is defined to handle all timeout cases. This method is automatically called in the send method when bit/loss error has occurred. Timer methods are used to handle the creation, updating, cancelling, and starting all timer threads for each packet in the window.



The receive method takes the packet data, split it, and write it to the output result. An ACK is use in conjunction with the receive method that would be sent to the client side, creating a header then sending the globally defined ACK value back to the sender.



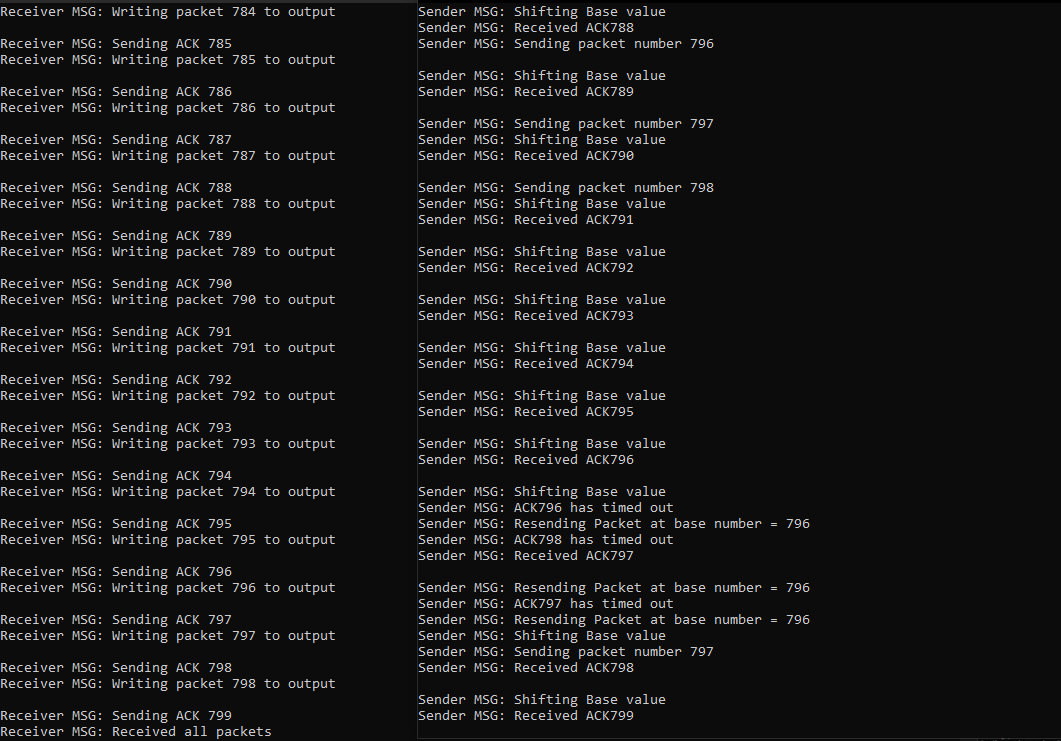
The next four methods are shorter methods to support the main three methods from before. These are conditional blocks use for error injection inside the code base on the option value and the error value.



More supporting methods, including creation/splitting of headers, and the checksum creation/test.

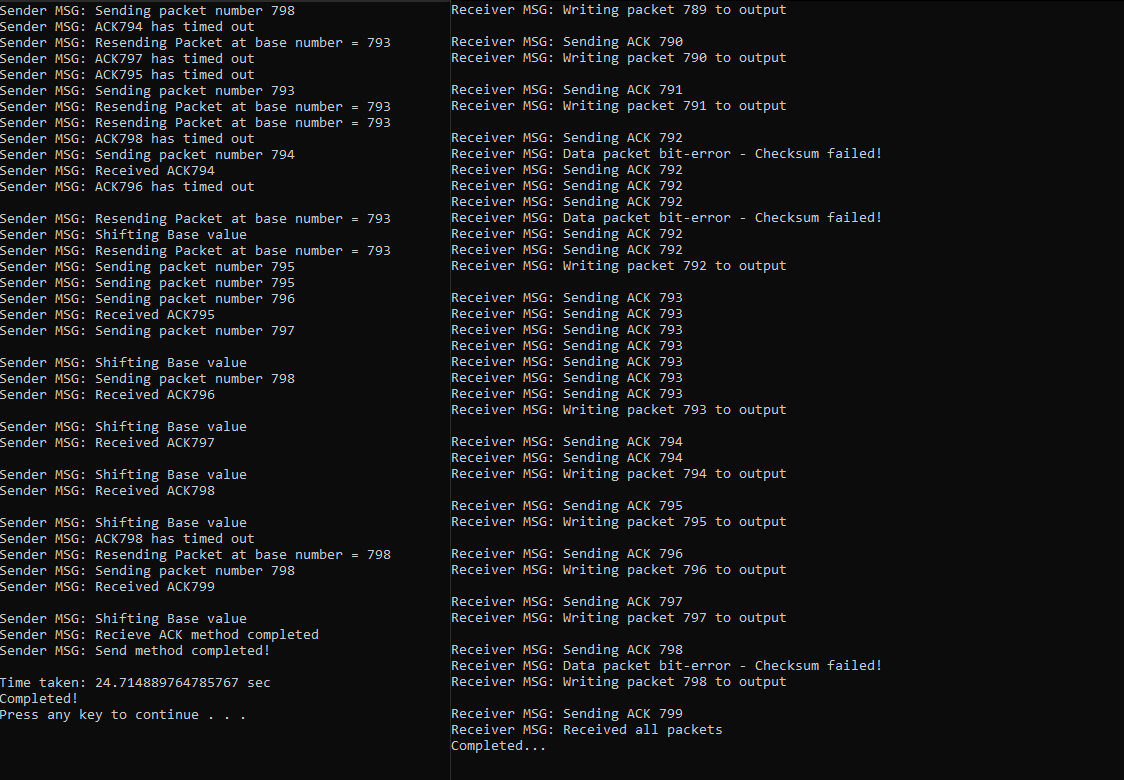
**Code Execution:**

Option 1 – no errors (server left, client right):



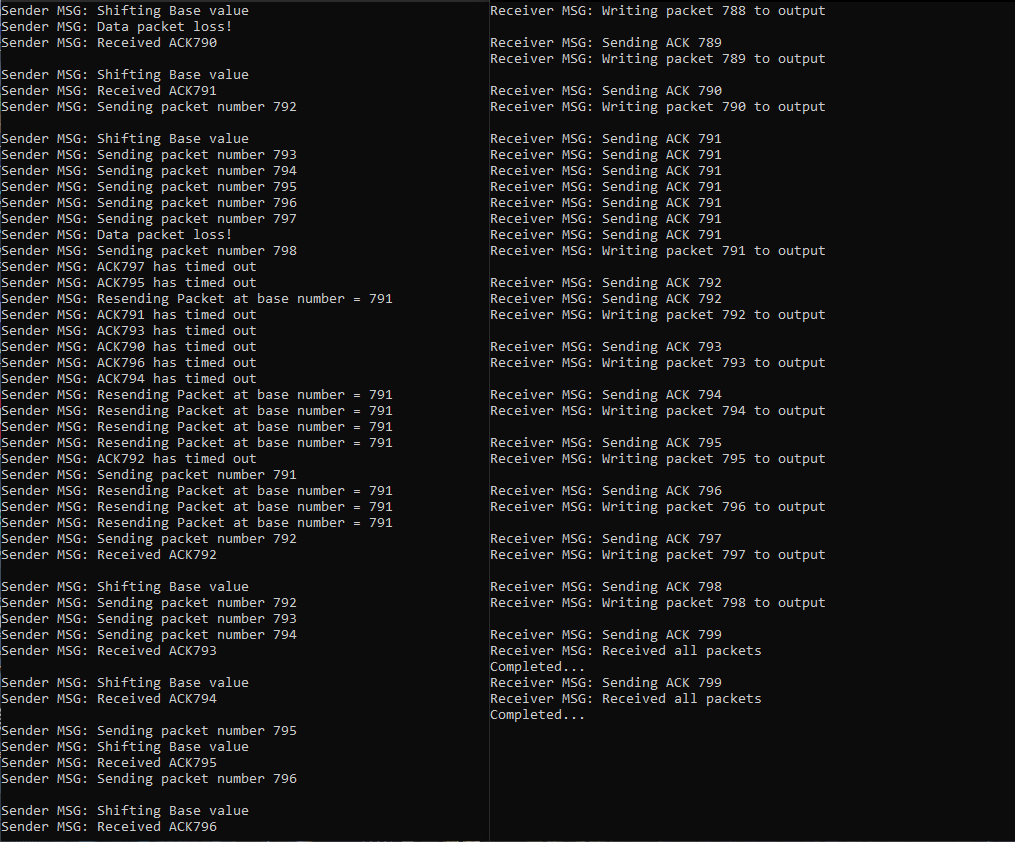
With no error, each packet is sent, base shifted, and written to output until the end of the data.

Option 3 – bit error (client left, server right):



Whenever a bit error occurs, resending happens back at the base value of the sliding window.

Option 5 – data packet loss (client left, server right):



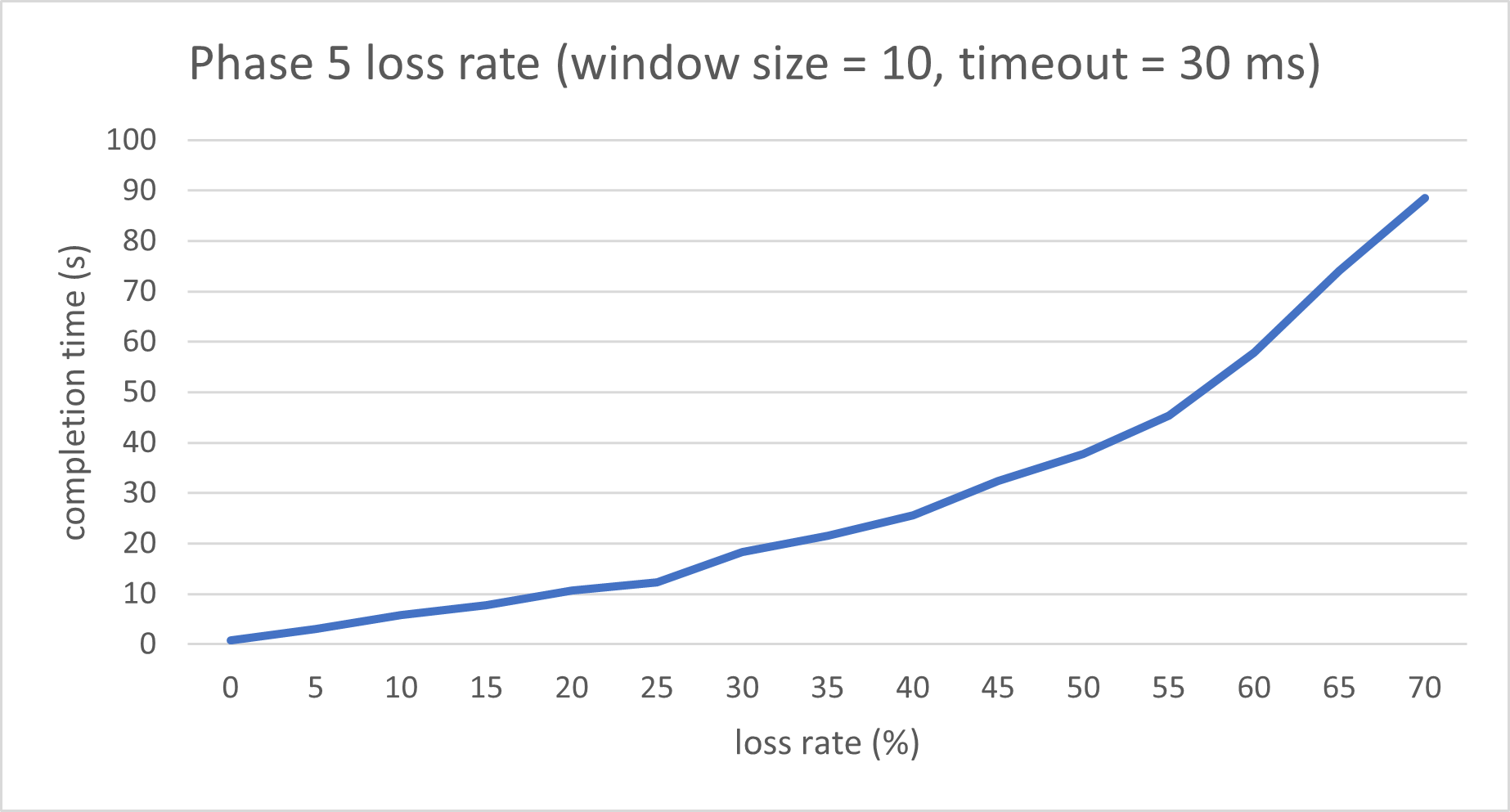
Whenever a loss occurs, resending happens back at the base value of the sliding window.

**Performance Graphs:**

For all graphs performance – the print statement is turned off.

Chart 1: Phase 5 performance

Option 3 - bit error



Option 4/5 – ACK/bit lost

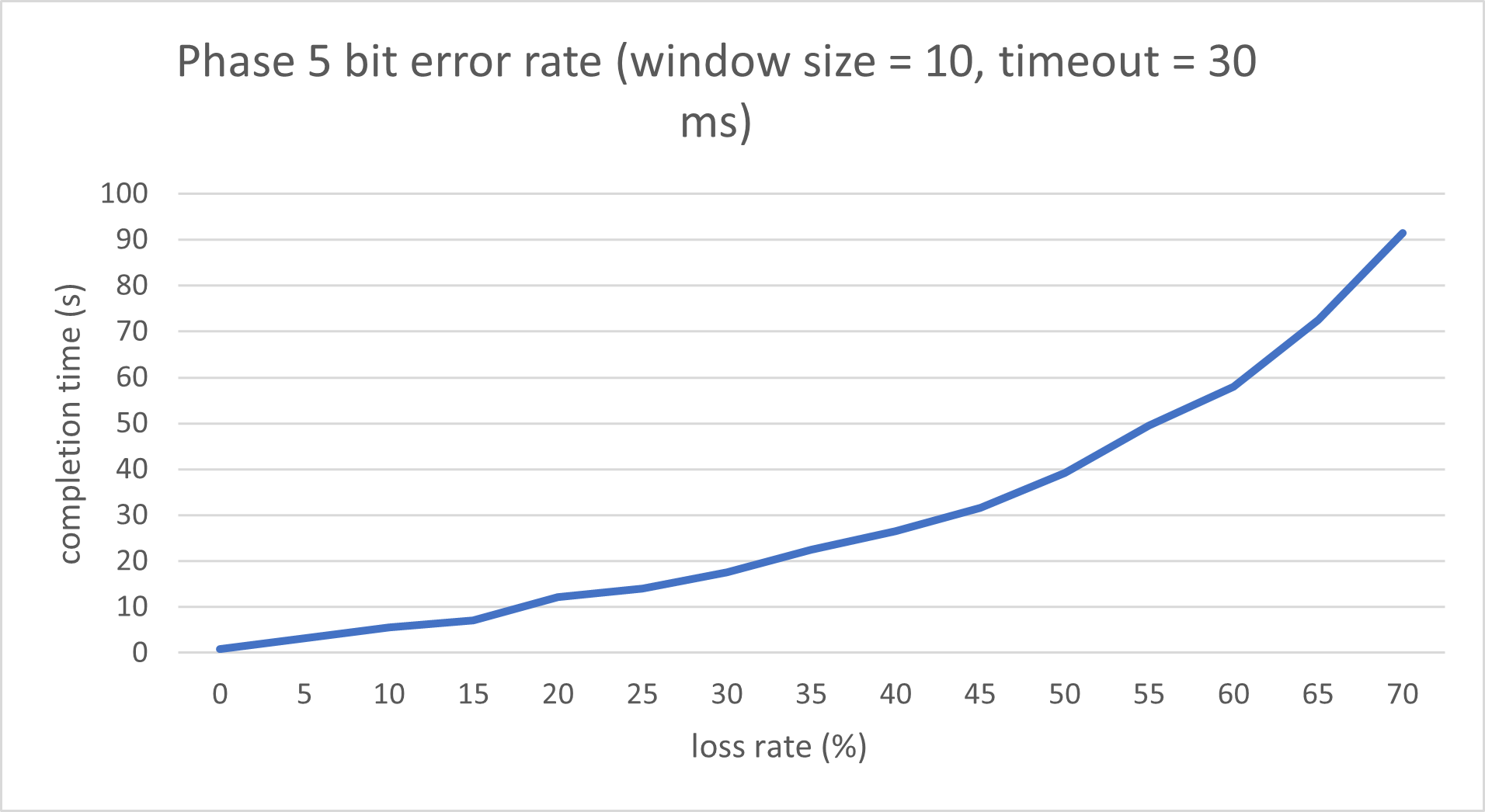
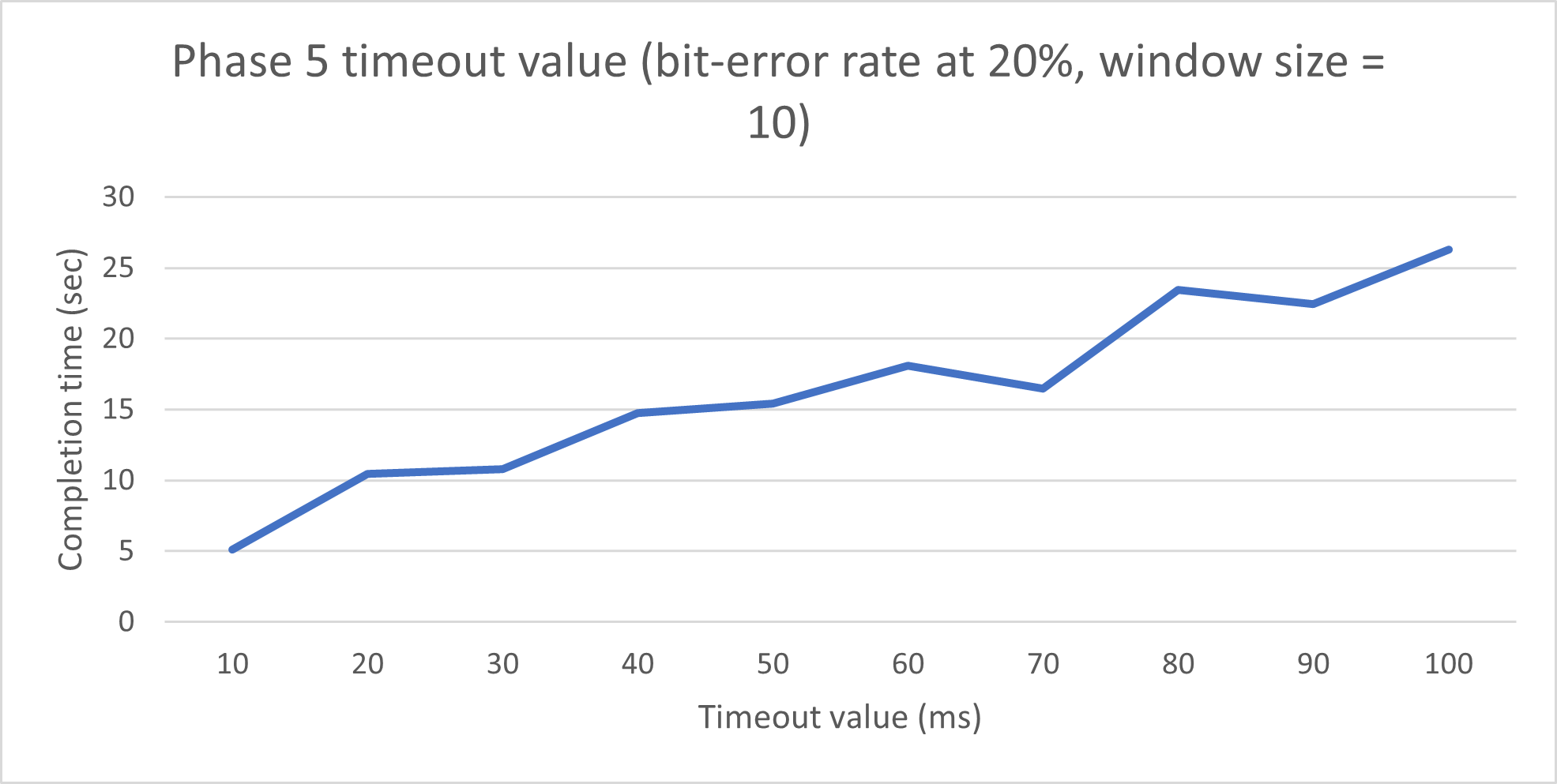
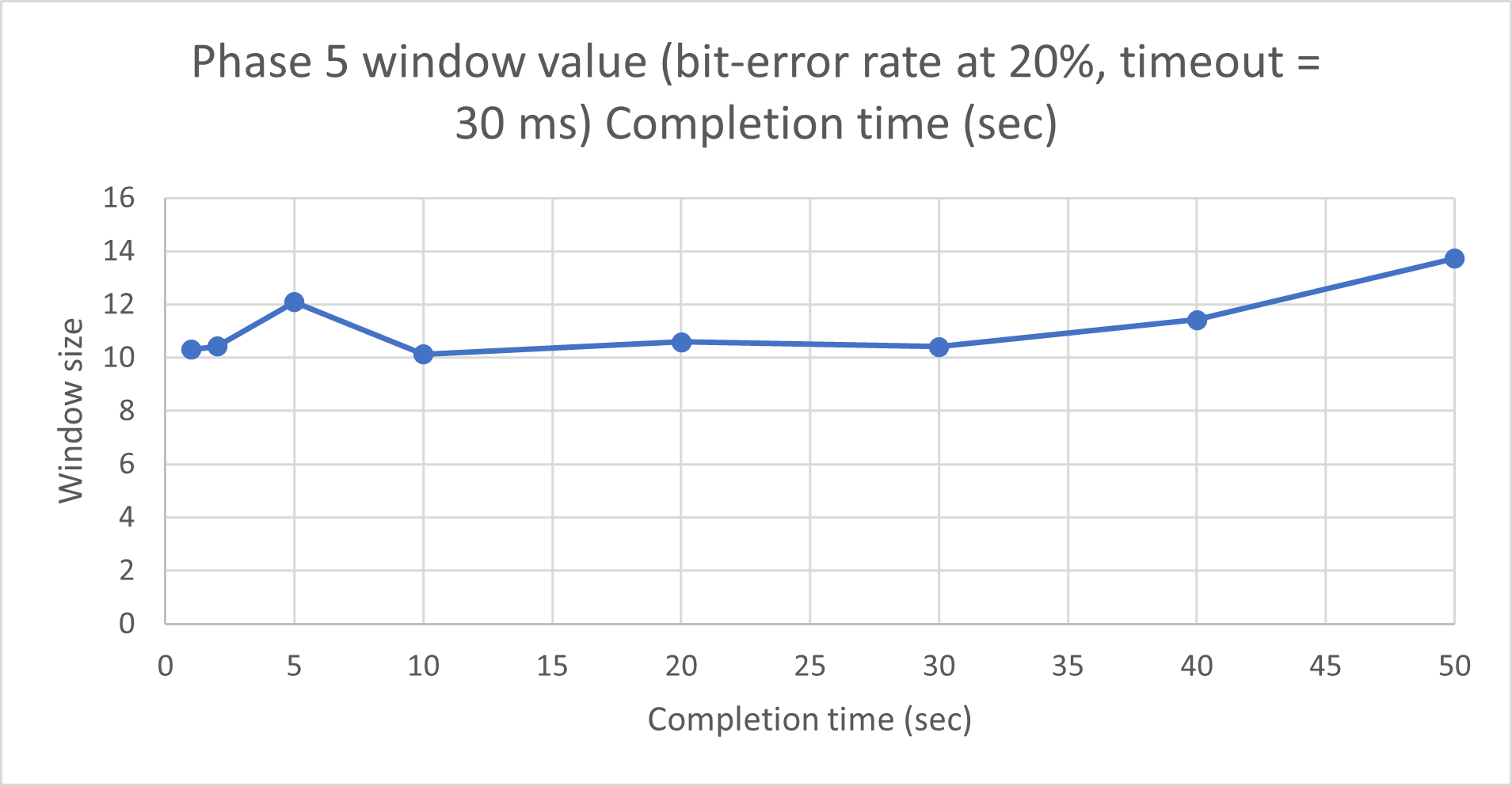


Chart 2: optimal timeout value



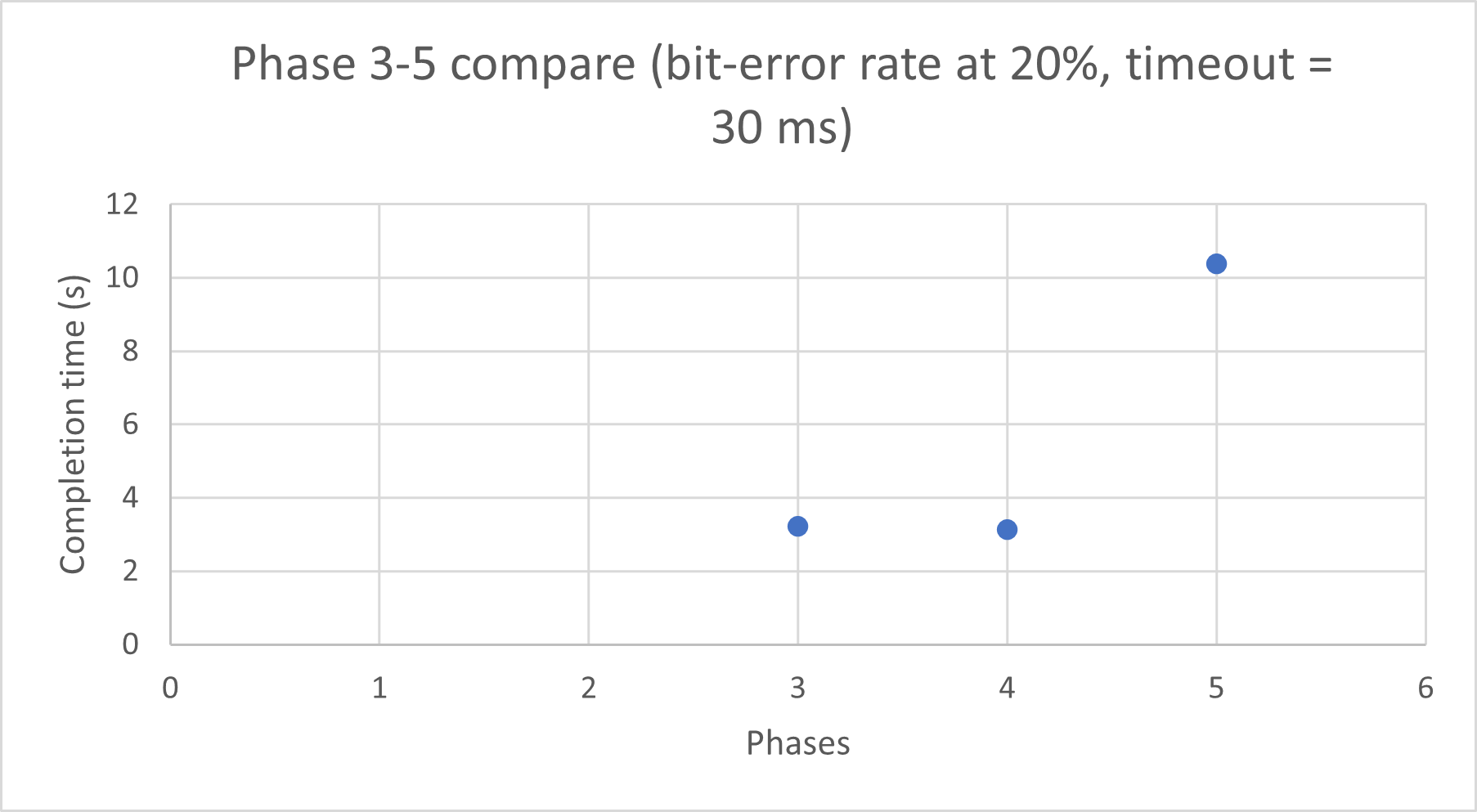
Timeout value seems to be better lower – however this can cause problems if timeout is too low as packet will not finish transferring sometimes.

Chart 3: optimal window size



There doesn't seem to be an optimal window size, however around 5-15 seems to be preferable. Window size of 1 can cause some problems.

Chart 4: Performance comparison of three phases at a fixed loss probability



Phase 3 and 4 completion time are much lower than that of Phase 5.