Advanced Network Programming

Go-Back-N

George Kozakjian

Contents

[1. Introduction 3](#_Toc506933847)

[2. Error Control 3](#_Toc506933848)

[3. Flow Control 4](#_Toc506933849)

[4. Sliding window protocol & Pipelining: 4](#_Toc506933850)

[4.1. Pipelining 4](#_Toc506933851)

[4.2. Sliding window 4](#_Toc506933852)

[5. ARQ Protocols 5](#_Toc506933853)

[5.1. Stop-and-Wait ARQ 5](#_Toc506933854)

[5.2. Go-Back-N ARQ 5](#_Toc506933855)

[6. Implementation and code 8](#_Toc506933856)

[6.1. Heavy use of Multi-threading 8](#_Toc506933857)

[6.2. Use of TCP Sockets 9](#_Toc506933858)

[6.3. Three way handshake SYN and FIN 9](#_Toc506933859)

[6.4. Window size negotiation 10](#_Toc506933860)

[6.5. File reading, Buffering and re-transmission of lost data 10](#_Toc506933861)

[6.6. Channel lose and error simulation 10](#_Toc506933862)

[7. Results 10](#_Toc506933863)

[8. Limitations 12](#_Toc506933864)

[8.1. Window size negotiation 12](#_Toc506933865)

[8.2. Channel propagation delay 12](#_Toc506933866)

[9. Improving the Go-Back-N ARQ protocol (bidirectional transmission: piggybacking) 12](#_Toc506933867)

# Introduction

The transport layer is responsible for end-to-end packet delivery of data between two nodes, over a communication medium. At first one might think that packet delivery is trivial and nothing wrong can go with it—machine A just puts the packet on the communication channel, and machine B just takes them off. Unfortunately, communication channels in real life are not ideal, and they make error occasionally. Also, they have a finite bandwidth, and propagation delay, which contribute negatively on designing efficient data transmission channels.

Hence, transmissions are prone to errors, and realistic systems need to be designed with the consideration that errors may occur, and may have drastic effects on the transmitted information. Hence, error detection and correction schemes need to be integrated in transmission systems from day one of their design (also known as error control schemes or methods).

Also, since the rate at which the receiver can process received data frames is much slower than the rate of transmission, due to the fact that it has to check and process each frame for errors, and send an acknowledgment or not. For this reason, each receiver has to have a buffer to store incoming data until they are processed. However, if the buffer begins to fill up, the sender must be notified to slow down or halt transmissions. The methods or techniques used to restrict the amount of data that a sender can send before waiting for acknowledgment is known as flow control.

Hence, at the data link or transport layer, flow control and error control need to be combined to achieve efficient delivery of data from one node to another. There are three protocols to achieve both techniques, which are subset of ARQ: (1) stop-and-wait ARQ, (2) Go-Back-N ARQ, (3) Selective Repeat ARQ.

# Error Control

Error control is about making sure the all packets are eventually delivered to the application layer at the destination mode, and in proper order. It’s composed of two different algorithms: (1) error detection algorithms, (2) error correction algorithms. They run independently, but are very closely connected to each other.

Error correction and detection can run at the link layer (Layer 2 of the OSI model), or the transport layer (Layer 4 of the OSI model). (1) A link is a point-to-point transport medium like a wire or wireless link which connect two devices that are of close proximity (e.g. in a LAN using Ethernet protocol IEEE 802.3). Link layer error detection and correction schemes are usually used for links that have high BERs like wireless links (e.g. IEEE 802.11 wireless link and physical layer). Since these links have high noises and collision rates, the transmissions have high probabilities of being prone to errors, and the packets being received as corrupt. (2) Transport layer error detection and correction schemes are used most of the time, in order to ensure end-to-end error-free transmissions.

There are fundamentally two categories of techniques for improving reliability in transmission channels. These are better known as error correction techniques.

First technique, known as ARQ (automatic repeat request). In an ARQ system, data are delivered from the higher layers like application layer, to lower layers, which buffers the data blocks, attaches control bits and passes them to lower layers. These layers encode the data blocks and add some parity bits from existing data bits, passes them to the modulator, which then transmits them over the channel. At the receiving end, the block is demodulated, and passed to decoder, which recomputes parity bits and compares with received parity bits. If both are equal, block is delivered to higher layer which notifies the sender, through suitable return channel, that the block is correctly received through an acknowledgment. If an error exist, the sender is notified to retransmit the block until it has been received without errors.

Second technique for dealing with transmission error is known as FEC (forward error control). In this technique, the decoder has some additional complex logic that attempt to find location of error from pattern difference between received and calculated parity bits, and tries to re-correct these error bits.

A third midway approach between the two techniques is hybrid ARQ scheme which consist of FEC contained within an ARQ system. Function of FEC is to reduce retransmission request by correcting as many blocks as possible.

# Flow Control

Error control algorithms ensure the delivery of a packet to destination, in channels that have high bit error rates (resulting from interference, path lose, propagation lose, etc.), however, it does not address the issue that occur when a sender wants to transmit faster than the receiver can accept.

Clearly, we need to do something to prevent the situation from occurring. There are two main approaches that are mainly used in this situation: (1) feedback-based flow control where the receiver sends back information to sender telling it that it can send more data, (2) rate-based flow control which limits rate at which sender can transmit data, without using feedback from receiver.

Feedback-based flow control schemes have well-known rules about when sender can transmit next packet. These rules prevent sender from sending until receiver gives it permission.

# Sliding window protocol & Pipelining:

## Pipelining

Pipelining is the basic idea of starting some task, before the previous task has ended. This idea is applied heavily in CPUs, in order to execute multiple instructions at a time. The idea of pipelining is not implemented in stop and wait ARQ, since the sender has to wait for an ACK frame, before starting next task (transmit next frame). Pipelining provides efficient use of bandwidth, since it allows for multiple frames to be sent at a time.

## Sliding window

Basic idea of the sliding window is to apply pipelining on ARQ, to improve the efficiency of transmissions. Sliding window is applied on Go-Back-N ARQ, and selective repeat ARQ. It allows to send up to *W* frames and keep a copy of them, until the ACKs arrive. Similarly, the receiver maintain a receiver window corresponding to frames it is allowed to receive.

In sliding window, sent frames are numbered sequentially, and stored in header of the frame. If the header allow bits for sequence number, then sequence number range from the sequence numbers are modulo, where is the size of sequence number field in bits.

The sliding window is used to hold the unacknowledged frames.

# ARQ Protocols

Flow control and error control are combined to achieve the efficient delivery of data from one node to another. There are three protocols for that combine the two techniques using ARQ, which are usually implemented in software using some programming language.

## Stop-and-Wait ARQ

Simplest flow and error control mechanism or technique used. The idea is that the sender node sends a frame, and then stops and waits for an acknowledgment from a receiver node. The sender node keeps a copy of transmitted frame, until it receives an acknowledgment (ACK). The sender starts a timer when it sends a frame, and if no ACK is received within allocated time period, the sender resends the frame, and this is repeated until an ACK is received. The sender discards the frame once an ACK is received. The frame and ACK are numbered alternatively 0 and 1 (two sequence numbers only). Adding a sequence numbers allows for identification of frame in case of duplicate transmissions. The receiver only sends positive ACKs. Sender has a control variable, that hold number of most recently sent frame (0, or 1). Receiver has control variable that hold number of next frame expected (0, or 1). The ACK number defines number of next expected frame (if frame 0 received, ten ACK 1 is sent).

Inefficient use of bandwidth, since the sender have to wait for ACK, after each sent frame. To improve efficiency, ACK should be sent after multiple frames.

## Go-Back-N ARQ

The receiver sends a positive ACK, if a frame has arrived and processed without errors and in the sequence order (frame is what it was expecting to be in its sliding window). If the frame is damaged or out of order, the receiver stays silent, and discards all subsequent frames. In this case, the sender will time out, and go back and resent all frames, beginning with one with expired timer. However, the receiver does not have to ACK each frame received. It can cumulate ACKs for several frames. In this case, the timer should be adjusted, so that It does not time out so quickly, giving the receiver the time to process the frames, and send back an cumulated ACK.

Go-Back-N is not efficient is high noisy channels or links, where frame have high probability of damage, which may result in resending multiple frames at a time.in this situation, Selective Repeat ARQ is used. It resends only damaged frames. It defines a negative ACK (NAK) that report sequence number of damaged frame before timer expires.

To find appropriate value for sender slider window size, we need to know how many frames can fit inside the channel as they propagate from sender to receiver. We can determine this quantity by multiplying the bandwidth in bits/sec with the one-way propagation delay, and divide that quantity by the number of bits per frame (let’s call this *quantity*). Then should be set to.

For example with bandwidth 50 kbps and one-way transmit time of 250 msec. the produce = 12.5 kbit or 12.5 frames of 1000 bits each. The product to = 26 frames. Hence sender window size = 26. Twice the bandwidth-delay is the number of frames that can be outstanding (waiting to be ACKed), if sender continuously sends and the RTT to receive an ACK is considered. Twice the bandwidth –delay is the number of packets that can be outstanding if sender continuously sends packet when the roud-trip time to receive an ACK is considered.

The acknowledgments from the receiver side could be individual, or accumulative. In the individual case, the receiver sends an ACK when it receives the next expected packet without any errors, otherwise it discards it causing the sender to time out, and re-transmit. Accumulative ACKs are sent for a group of packets that arrive at the receiver in order.

In this figure, we find an example of a packet flow that may occur in a go-back-n ARQ protocol.

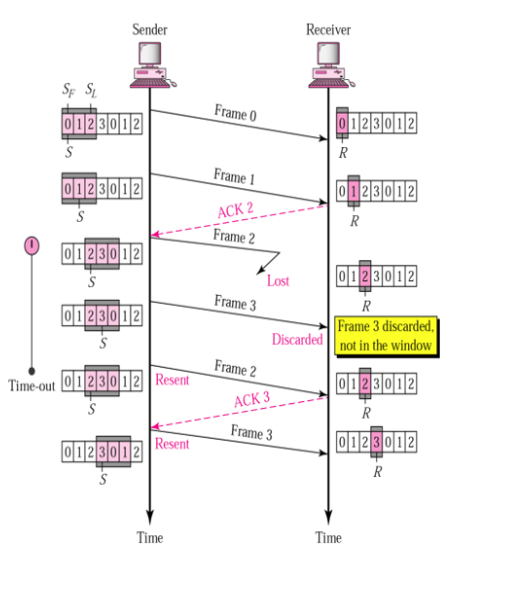


Figure 1: Packet flow in Go-Bcak-N

The content of sliding window for sender and receiver:

* Sender sliding window:

Is an abstract concept, which looks like a circular array of size (sequence number – 1). It can slide one or more slots when a valid acknowledgment arrives. *[Figure 1].*

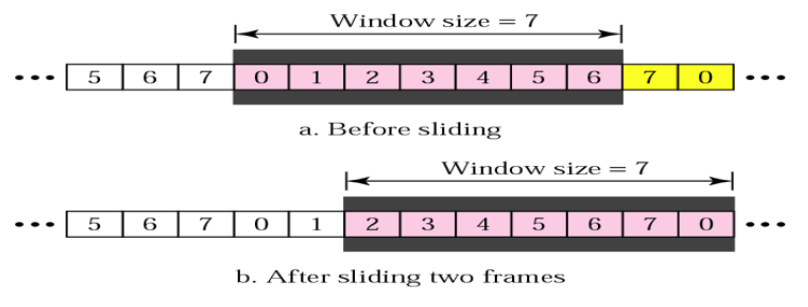


Figure 2: Sender Sliding Window

* Receiver sliding window:

Is an abstract concept, which looks like a circular array of size 1. The window slides when a correct frame has arrived, and sliding occur one slot at a time. *[Figure 2]*

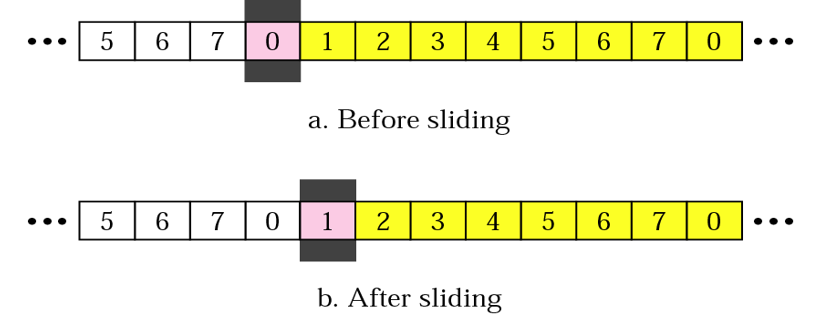


Figure 3: Receiver Sliding Window

# Implementation and code

To implement the go-back-n protocol, python will be used. The implementation tries to mimic the protocol as much as possible, however, some limitations exist due to lack of hardware, and lack of testing in real geographically separated networks.

The main components of the implementations include, but are not limited to:

## Heavy use of Multi-threading

The implementation utilizes threading heavily to make the protocol as parallel and as asynchronous as possible. Threading is the idea of running some set of code by several processors at different stage of execution, or running same code in parallel or simultaneously by using scheduling and pipeline in the processor, to service as much requests as possible.

In this simulation, threading is used first to initialize both the receiver and sender nodes from a single starting point. This facilitate running the simulation, adding multiple senders and receivers on multiple sockets, and controlling the order of execution of the sender and receiver nodes (typically the receiver starts before the sender, so it can listen on the socket and accept connections). In the figure below, we can see the code of starting the sender and receiver threads. In this example, the sender will create a socket on localhost (127.0.0.1/loopback address) and port 22222, and listen on that port for connections.

Threading is also used on the receiver and sender sides. On the sender side, an acknowledgment thread is created when a socket is connected by passing the thread the socket, so it can receive acknowledgment while the sender sends new frames simultaneously. The idea is that the sender need to send and receive at the same time, so threads need to be utilized here. The sender will create this thread and start it after finishing window size negotiation, so it can receive acknowledgments.

On the receiver side, threading is used to create a receive instance thread when a new connection is accepted passing it the established socket. The idea here is to have the receiver, receive data from multiple senders. To accomplish this, the receiver has to create a separate thread for each sender, so that communication can occur between multiple senders and the receiver over different sockets (different source port and maybe IP address on the sender side) each using its own thread that has the same copy of the receiver code implementation or service. The receiver creates this socket thread instance when it accepts a new connection.

## Use of TCP Sockets

The implementation uses python TCP sockets (stream sockets) to simulate the sender and receiver that are trying to communicate end-to-end over a packet-switched, shared network infrastructure. A socket is a high level API that is used to create transport layer end-to-end communication channel, which can be utilized to create application layer protocols and services. It is comprised of an IP address and a port number, which uniquely identifies it on the internet or intranet. This implementation uses sockets on the sender and receiver sides. The sender creates a socket specifying the destination service or receiver IP address and port number that it wants to communicate to. Once *socket.connect* is called, the TCP/IP stack on the sender (implemented in software and hardware) create a socket by specifying a random port number and an IP address assigned to one of the senders network interface cards. This socket is then used to carry out the communication between the two nodes. On the receiver side, a socket is created specifying the local IP address and port number that it wants to listen on or provide a service on or accept data on. The sender first executes *socket.listen,* and next socket.accept to listen and accept new connections.

## Three way handshake SYN and FIN

The implementation uses three way handshake to establish the connection between the two nodes. On top of TCP, the implementation includes its own handshake which is comprised of ACK, SYNACK, and SYN sequence sent and receives to complete the handshake phase. The handshake is started by the sender, sending an ACK packet. Followed by a SYNACK from the receiver, and an ACK from the sender to complete the handshake phase. The handshake data is passed using serialized JSON from one node to another over TCP sockets.

## Window size negotiation

The negotiation is implemented by sending two randomly generated, sender-supported window sizes from the sender node (assuming these window sizes were generated by sensing the underlying communication channel), receiving them at the receiver node, generating two random window sizes at the receiver node, and comparing the two to take the Max-Min window size that can be agreed on by both parties. The receiver after receiving the sender supported window sizes decides the final window size for each session, and communicates it back to the sender. The sender then sets the *receiver\_max\_window\_size.*

## File reading, Buffering and re-transmission of lost data

In this simulation, we use a file to read data and transmit to receiver. However, this creates the problem of buffering old data for packets that are outstanding in the sliding window. To solve this problem. We read the file into a buffer array, and transmit the index of the file in the JSON payload, so that in case of corruption, we can know which index to re-transmit to the other node.

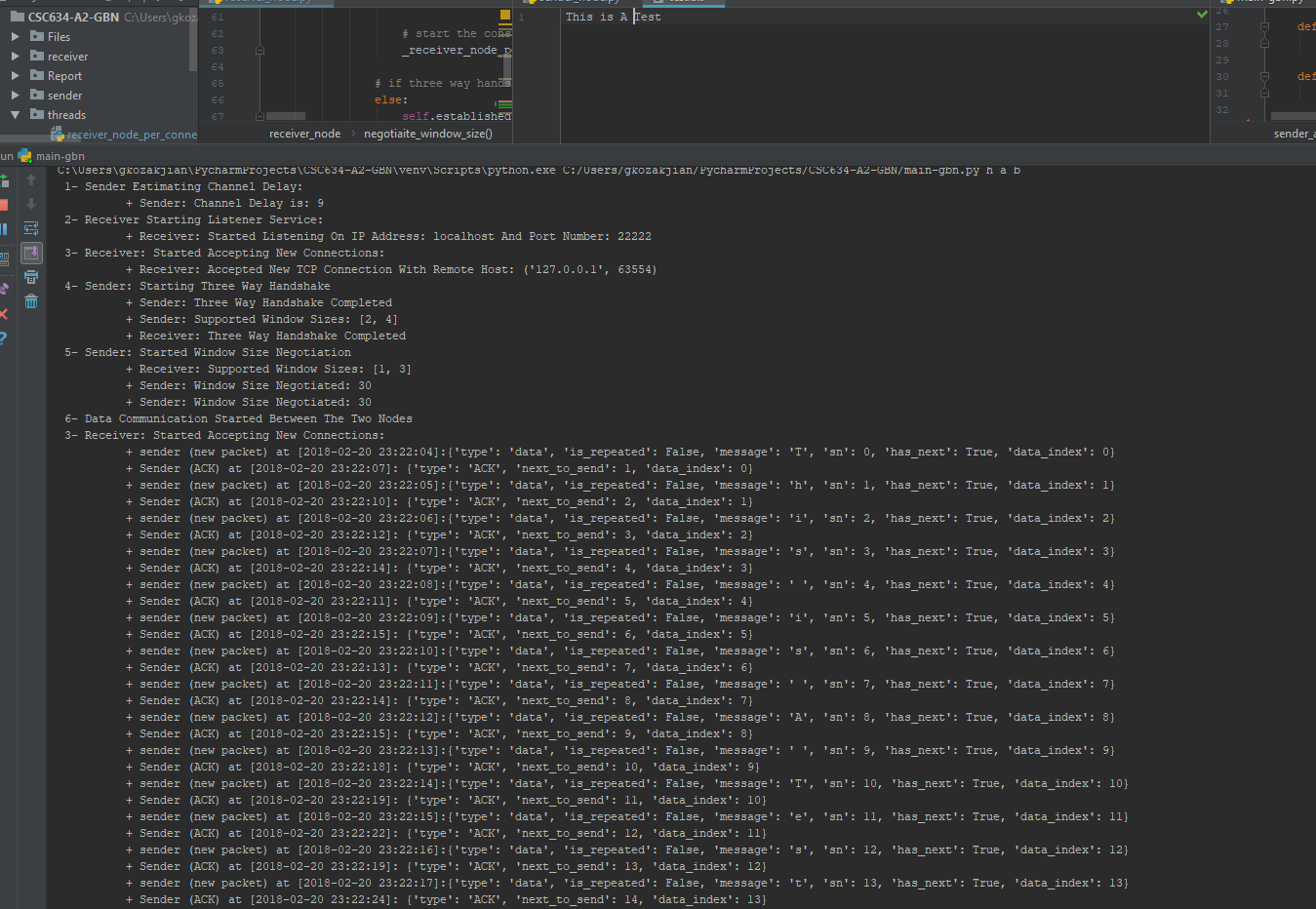
## Channel lose and error simulation

The implementation accounts for the congestion rates entered by the user to calculate the delay of the channel. Also, it has a channel lose simulator running in parallel, waiting for the input of the user to simulate channel lose. If run, the receiver will stop sending back ACK for two seconds, and then sender will recover and transmission continue

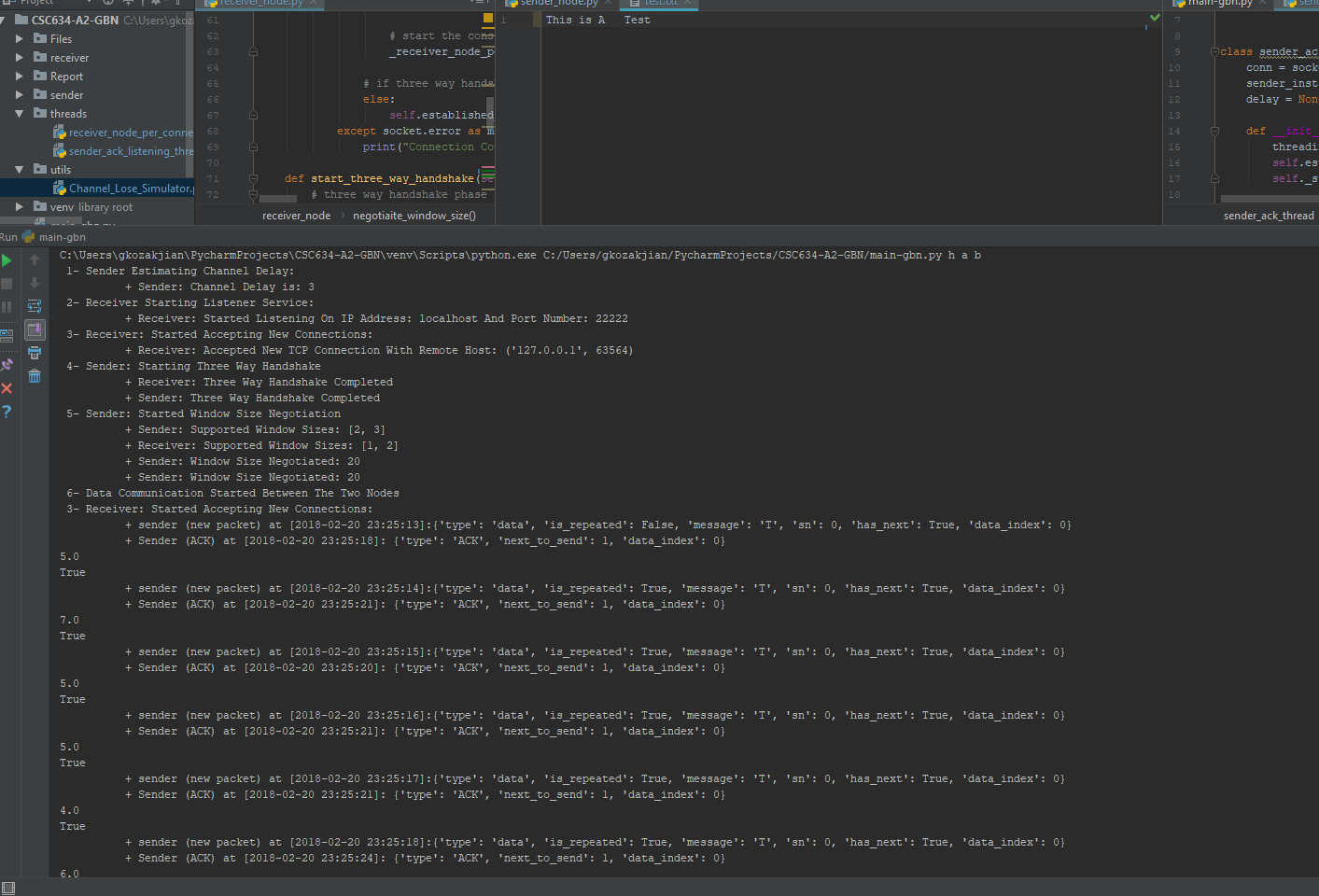
# Results

Although the python implementation is not perfect, it tries to mimic the real protocol well.

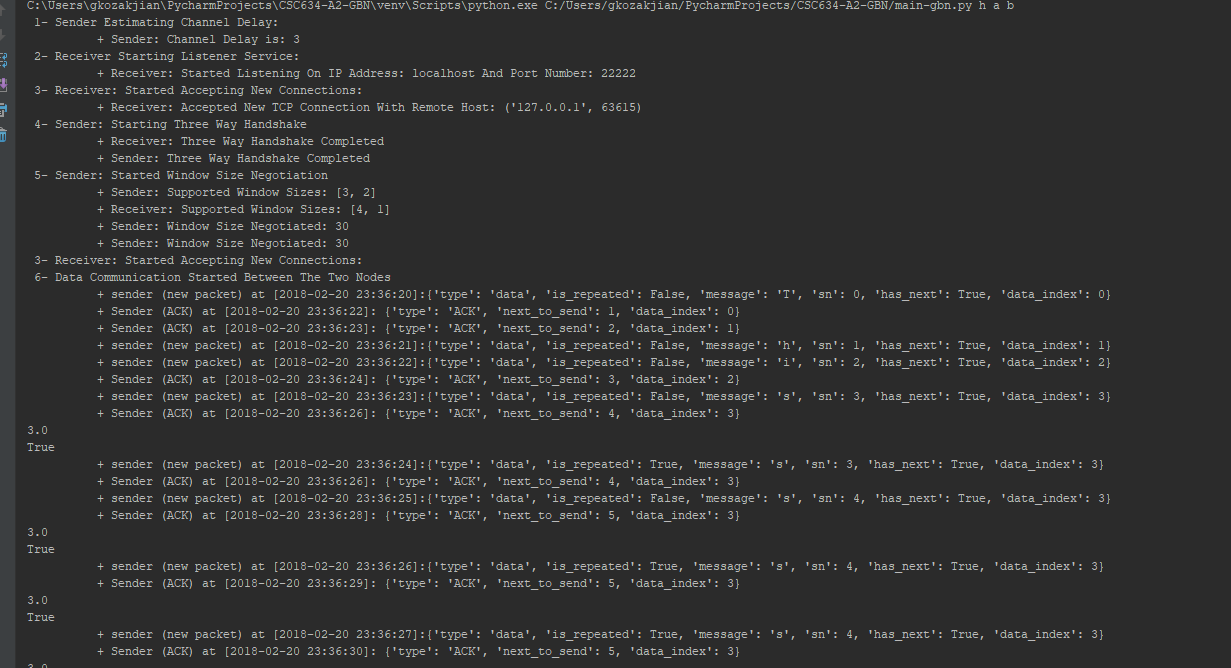
Here different results are demonstrated on different file sizes, delays and window sizes.



In this example, we have a high delay channel, and simulated a channel lose, so almost all the packets were lost.



In this example we see a mix of errors and normal transmissions. Our delay is 3, and congestion rate is low, so there is high probability of receiving packets well.



# Limitations

## Window size negotiation

In real-life go-back-n implementations, the window size negotiation at some node starts by senses the underlying network infrastructure and channel, trying to get the bandwidth and round-trip time, in order to calculate the window sizes that it can support in this implementation, adding channel sensing was inefficient since the loopback interface is used to send and receive the packets from the sender to the receiver.

## Channel propagation delay

The propagation delay plays a big role in the error correction world. High delays mean less packets can be transmitted, and more errors can occur in the transmission channel. One way Propagation delay also plays an important role in negotiating and deciding on a maximum supported window size for the communication session. In this implementation, a near-to-zero propagation delay (less than 1 ms) is assumed, since the communication is actually looped back from the local loopback virtual network interface. This limitation prevents this the simulation from having results that can be closely relatable to real-world implementations where the end-to-end transport channel may span continents, countries, and cities.

# Improving the Go-Back-N ARQ protocol (bidirectional transmission: piggybacking)

In stop-and-wait ARQ, data frames are transmitted in one direction only. To transmit in both directions, the protocol can run two instance, each using separate link for data traffic. Each link will contain a forward channel for data and a reverse channel for acknowledgments. This make capacity of reverse channel entirely wasted.

One better idea is use same link for data in both directions. In this model, data frames are intermixed with ACK frames. By looking at the *kind* field in the frame header, receiver can tell if its data or ACK. This also can be improved by attaching ACK frame in data frame.

This idea is known as piggybacking. Piggybacking is the idea of combining data frame with acknowledgment frames. It saves bandwidth, since both data and ACK frames are combined into just a single frame. This holds true as long as both node have data to transmit, otherwise Go-Back-N with cumulated ACK is used.

Another aspect is testing in real life environments. The code needs extensive testing and optimization to run on real environments.

A third area of improvement is cumulative acknowledgments. In this implementation we used individual ACKs which may be bandwidth consuming. To improve this, cumulative ACKs can be used to utilize bandwidth more efficiently.

A forth area of improvement is in measuring precisely the bandwidth and delay by using channel sensing technics. This will improve the accuracy of the protocol and window sizes, improving the overall performance.

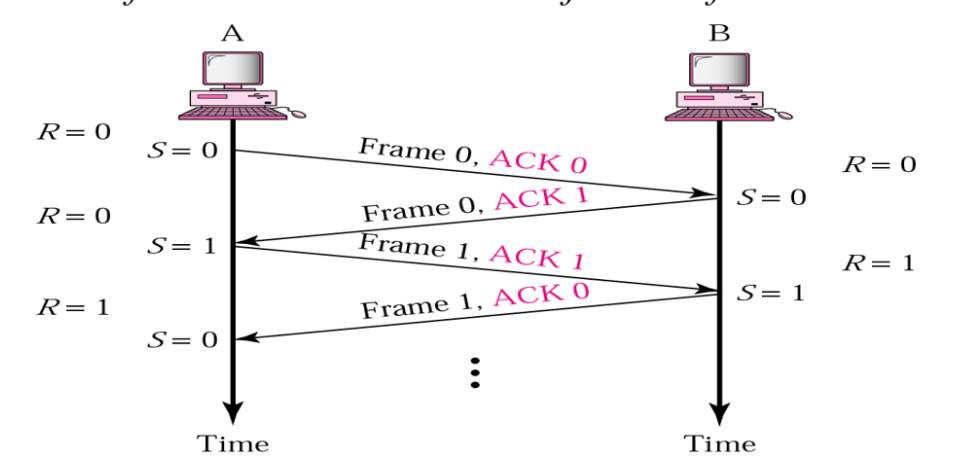


Figure 6: Piggybacking in Go-Back-N