Documentation

The program is coded with **Python 2.7**

Assignment requirements:

- The server and client can be started as separate processes by running their main files called server3.py and client3.py or serverxor.py and clientxor.py
- Messages are sent using UDP by using Python standard socket implementation
- Sent data is split to packets that contain a maximum of 100 bytes. These 100 bytes include the programs' own header
- Packet loss is implemented by the server not sending a packet based on a chance
- Python random library is used along with user defined or hardcoded loss percentage value to determine whether to lose a packet or not
- User can pass a float value between 0 and 1 to the server3.py and serverxor.py. This value is used as the loss percentage
- If server is started without any loss percentage, it will execute itself multiple times with different loss percentages between 0 and 1. The step of the percentages to use is hardcoded
- The data being sent is randomly generated with the help of Python's random library.
- To ensure the data is the same between executions and when checking the validity of the data, a new random generator object is created and it is given the same seed value each time so it generates the same sequence of values each time.

Assumptions:

Additional assumptions for the implemented system

- Server sends every time the same string as a set of packets
- The delay between sending each string is over 0.1 seconds
- There can be a maximum of 255 packets for a single string being sent
- The header is a single byte that represents the packet ID
- The packets in a single set have sequential packet ID numbers
- The packet ID numbers start from 0
- The character '_' (underscore) is assumed not to be included in the string. It has been used to indicate lost packet data visually during debugging and development
- The character ' ' (space) is assumed not to be included in the string. It has been used as dummy data for padding and unused data
- Packet length is 100 bytes

Implementation and usage:

FEC with sending three redundant packets:

- client3.py is the client and when started it waits for a packet forever
- server3.py is the server and when started it sends packets according to given loss percentage
- The server splits the sent string to packets of 99 bytes
- The last packet can have less than 99 bytes of content
- The packets are sent three times and before sending a one byte header representing the packet ID is attached in front of the packet
- After sending the packets the server will wait for 0.15 seconds before sending the next set of packets for the next string that is given
- After all strings are sent the server writes data gathered from sending packets to output_server.json and closes
- Upon receiving a packet the client determines if it was already received or not based on the packet ID. This is possible because client knows that a sequence of three packet IDs are the same and the packet IDs start from zero.
- If the client received packet has expected packet ID we save it. If the received packet has higher ID than any of the expected ones then packets filled with '_' are saved to indicate lost data and the new packet is received normally. If we receive packets that have lower ID than expected then we ignore them.
- After no packets have been received for 0.1 seconds the input has been assumed to be finished and the resulting packets can be joined to see the resulting message
- Any '_' in the message is lost data
- After assuming a finished input the client will again wait forever for a packet to arrive - starting to wait for a new input
- After all data has been received the client must be manually closed by the user by sending a termination signal by pressing CTRL+C. This will cause the client to save gathered data to output_client.json and close

FEC with sending XOR packet of every two packets:

- clientxor.py is the client and when started it waits for a packet forever
- serverxor.py is the server and when started it sends packets according to given loss percentage
- The server splits the sent string to packets of 99 bytes
- The last packet must not have less than 99 bytes of content. Space is used to fill the packet up to 99 bytes and space is reserved for this purpose and must not be contained in the string being sent
- Since the FEC needs two packets to create the XOR packet we need the length of the list of packets to be even. If it is odd, we will add a packet containing only spaces
- After every two packets are sent a third packet is sent that consists of an XOR operation on the bytes of the two packets. Before sending a packet a one byte header representing the packet ID is attached in front of the packet
- After sending the packets the server will wait for 0.15 seconds before sending the next set of packets for the next string that is given
- After all strings are sent the server writes data gathered from sending packets to output_server.json and closes
- Upon receiving a packet the client determines which packet of a set of three packets was received and adds it and placeholders for the missing packets to a list based on the packet ID. This is possible because client knows that a sequence of three packet IDs are the same and the packet IDs start from zero
- If the client received packet has expected packet ID we save it. If the received packet has higher ID than any of the expected ones then packets filled with '_' are saved to indicate lost data and the new packet is received normally. After no packets have been received for 0.1 seconds the input has been assumed to be finished
- After assuming a finished input the client will again wait forever for a packet to arrive - starting to wait for a new input
- After all data has been received the client must be manually closed by the user by sending a termination signal by pressing CTRL+C. This will cause the client to save gathered data to output_client.json and close.

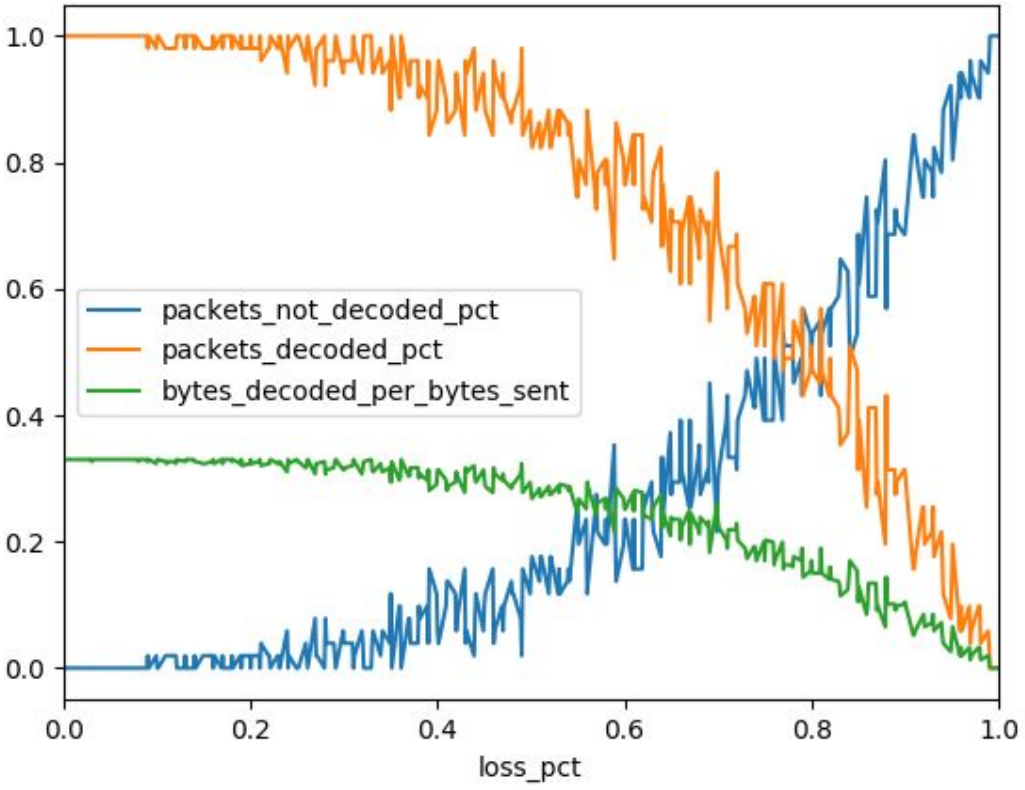
Assignment questions and answers

FEC with sending three redundant packets:

For each loss percentage we sent a string of 5000 bytes.
For each of the strings 153 packets were sent, 15153 bytes were sent, 51 packets with actual content were sent.

How does increasing loss rate affect the success rate of decoding?
The decoding success rate decreases in an exponential fashion. However even at 80% of packets being lost we still get about 50% of actual content through successfully, which is quite good if we can retry sending a single packet or we don't need all of the data to be received. If we do need all of the data to succeed in a single transmission then even a 90% success rate is too low for a single packet, because the chance of overall success would be close to zero.

What was the overhead, i.e., how many additional bytes you needed to send to get a certain number of bytes successfully decoded?
About 33% or less bytes were actually decoded from the bytes sent depending on the loss percentage. 15153 bytes being sent is quite a lot for a message of only 5000 bytes. The packet system overhead was 153 bytes, one for each packet we had to send. The redundancy overhead was 10000 bytes of content, which is three times the string being sent. Two thirds from the packet system overhead is due to the extra packets we had to send for redundancy.

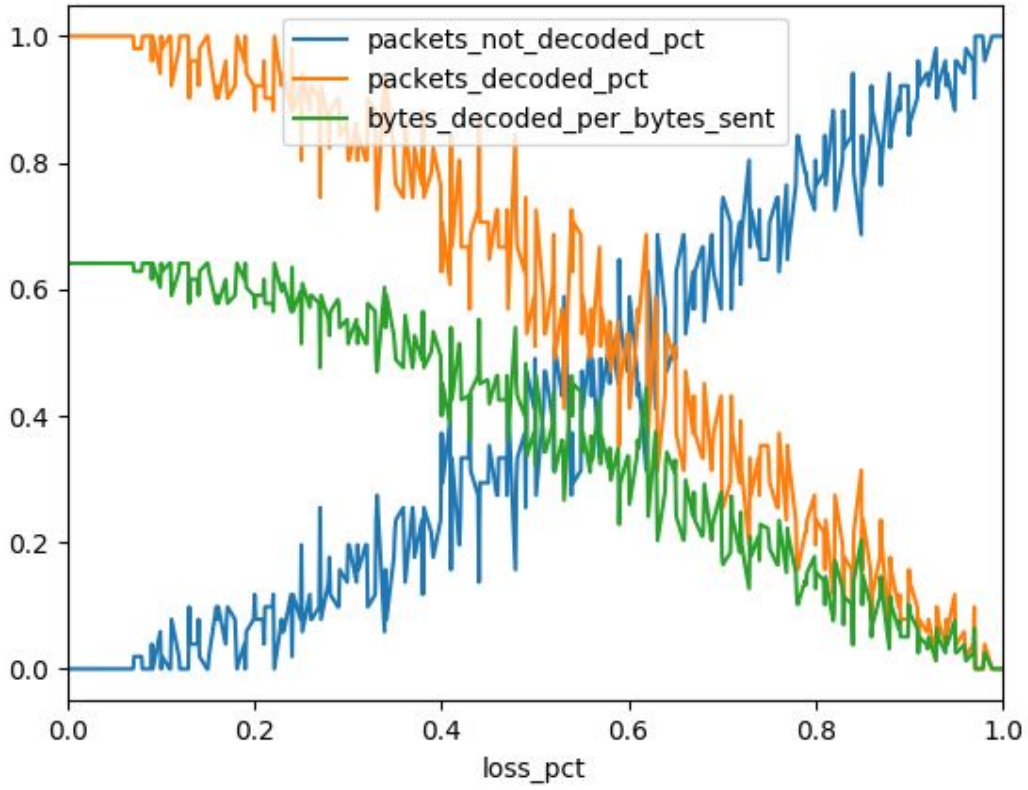


FEC with sending XOR packet of every two packets:

For each loss percentage we sent a string of 5000 bytes.
For each of the strings 78 packets were sent, 7800 bytes were sent, 51 packets with actual content were sent.

How does increasing loss rate affect the success rate of decoding?
The decoding success rate decreases in a linear fashion. However already at 50% of packets being lost we get 50% of actual content through successfully, which is OK if we can retry sending a single packet or we don't need all of the data to be received.

What was the overhead, i.e., how many additional bytes you needed to send to get a certain number of bytes successfully decoded?
About 66% or less bytes were actually decoded from the bytes sent depending on the loss percentage. 7800 bytes being sent in total is not that much overhead for a message of 5000 bytes. The packet system overhead was around 78 bytes, one for each packet we had to send. The redundancy overhead was 2673 bytes of content, which around one third overhead due to the extra packets we had to send. One third from the packet system overhead is due to the extra packets we had to send for redundancy.



Resultplotter

resultplotter.py is a small python program that draws a plot from the output_server.json and output_client.json. You can use it to visualize your data. It was used to create the plots in this document.