COSC264 Networking Assignment

Ollie Chick (67683982)
Samuel Pell (57046549)

Percentage contribution: 50% each

9 September 2017

1 Deadlocks

In networking, a deadlock is where transmission of new data ceases due to the system state. A deadlock occurs in this protocol when the last acknowledgement packet from the receiver is lost. The receiver closes after it sends the last acknowledgement packet, so if this packet is lost, the sender thinks that the last data packet was lost and keeps resending the last packet. The sender is thus in a deadlock state: it tries to retransmit the last packet continuously, preventing it from transmitting any more data it may need to.

2 The `magicno` field

The `magicno` field ensures that the message came from this protocol and that some other program has not connected to one of the programs by mistake.

3 Bit errors

Bit errors were solved by adding a checksum to the header. This was calculated when a packet was created by taking the last three digits of the sum of the other header fields. When the sender and receiver received a packet, they calculated what the checksum should be and compared it to the actual checksum. If they were identical, they processed the packet, otherwise they discarded it.

4 The `select()` function

The `select` function, or in Python the `select.select` method, waits until data can be read from the socket (or a timeout, if specified, is exceeded). In channel, this method is used so that it can wait until it can read data from its sockets, and only then read from sockets that have new data. This is useful as it reduces code complexity and prevents the channel program from using CPU resources unnecessarily.

In sender, the `select` method is used to enact the timeout mechanism for re-sending data if an acknowledgement packet isn't received.

5 Verifying correct transferral

To check that the two files were the same, `diff <input_file> <output_file>` was run. This compares the contents of the two files and outputs nothing if they are identical.

6 Packet loss measurement

The number of packets required to send a 512,000-byte file with different packet loss rates was tested experimentally. To do this, the programs were run 10 times for each packet loss rate. The results of this are shown in Table 1, and are summarised in Figure 1.

The number of packets required increases quadratically with the packet loss, because another packet will have to be sent if either the data packet or the acknowledgement packet is dropped.

Table 1 - Number of packets required to send 512,000-byte file for different packet loss probabilities.

Packet Loss Probability	0	0.01	0.05	0.1	0.2	0.3
Packets Sent	134	126	142	162	185	227
	123	136	139	154	191	202
	128	125	138	151	178	272
	122	119	153	131	188	293
	119	130	137	159	167	269
	124	115	141	156	208	269
	124	128	139	147	219	292
	121	123	136	145	173	215
	121	126	142	146	183	274
	118	136	140	150	179	253
Average	123.4	126.4	140.7	150.1	187.1	256.6

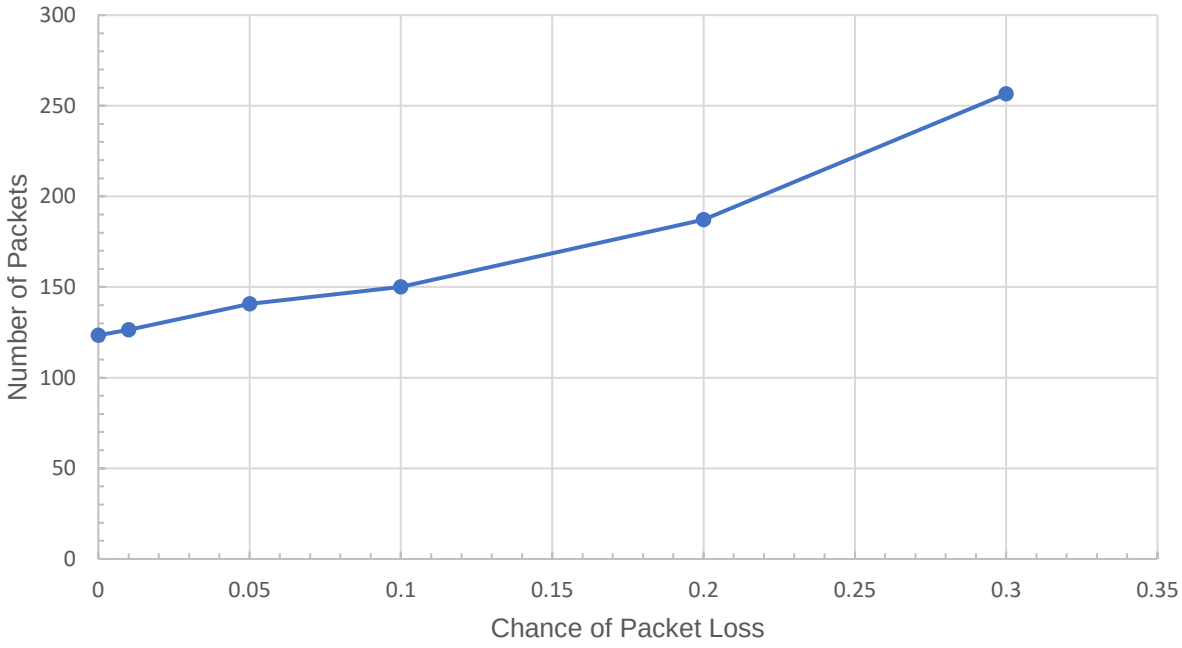

Figure 1 – The average number of packets required to transmit a file 512,000 bytes long.

7 Derivation of average packets required

Let *P* be the probability that a packet is dropped or has its data length field changed, and *N* be the number of packets that is needed to transmit some arbitrary file. Assuming that each transmission is statistically independent of all others, each transmission can be defined as a Bernoulli trial. Let the event the a given packet is dropped be a success and the event that a packet is successfully transmitted be a failure.

If a series of these Bernoulli experiments are performed then the results will have a negative binomial distribution. That is, have a probability mass function of

$$\binom{k+r-1}{k}(1-\gamma)^r\gamma^k, \qquad \text{where } \binom{k+r-1}{k} = \frac{(k+r-1)!}{k!\,(r-1)!} \tag{1}$$

where *k* is the number of successes, *r* is the number of failures, and *γ* is the probability of success. This distribution has the expected value of

$$\mu = \frac{r\gamma}{1-\gamma} \tag{2}$$

where *μ* is the expected value (Pennsylvania State University, 2017), (Weisstein, nd).

The negative binomial distribution measures the number of success before *r* failures occurs. Because of this definition, the average number of packets that must be sent to transmit successfully transmit the file is

$$\mu = N + \frac{N\gamma}{1-\gamma} \tag{3}$$

There should be constant of one as overhead from the protocol - an empty data packet is sent to signal the end of transmission - this packet can be lost and need retransmission. However, this is ignored as the last packet must pass through unmolested for successful transmission to occur.

However, the parameter *γ* is not the same as *P* — it is in fact

$$\gamma = 2P - P^2 \tag{4}$$

This is because if the acknowledgement packet from receiver is lost then a retransmission also occurs. Thus, the average number of packets needed to transmit a file that can be split into *N* packets is

$$\mu = N + \frac{N(2P - P^2)}{1 + P^2 - 2P} \tag{5}$$

which can be simplified to

$$\mu = N\left(1 + \frac{P(2-P)}{(P-1)^2}\right) \tag{6}$$

Table 2 shows that the predicted average number of transmissions closely matches the experimental average. The difference will be due to the small sample size of 10 and the fact that the random number generator in Python is only pseudo-random and not truly random.

Table 2 – The predicted average number of packets required to send a 100-packet file compared to an experimental average.

Packet Drop Rate	Predicted Average	Experimental Average
0	123.5	123.4
0.01	126.0	126.4
0.05	136.8	140.7
0.1	152.4	150.1
0.2	192.9	187.1
0.3	252.0	256.6

References

Pennsylvania State University (2017). *Key Properties of a Negative Binomial Random Variable*. Retrieved on 30th August 2017 from https://onlinecourses.science.psu.edu/stat414/node/79

Weisstein, Eric W. (nd). *Negative Binomial Distribution.* Retrieved 30th August 2017 from http://mathworld.wolfram.com/NegativeBinomialDistribution.html

Appendices

Appendix 1 – packet.py

```
"""
   Packet
   A class to represent a packet of information.

   Authors: Samuel Pell and Ollie Chick
   Date Modified: 30 August 2017

   Contains:
        __init__()
        __repr__()
        __str__()
        __len__()
        encode()
        decode()
        is_valid_ack()
        is_valid_data()
"""

PTYPE_DATA = 0
PTYPE_ACK = 1
MAGIC_NO = 0x497E


class Packet:

    def __init__(self, magic_no=0, packet_type=0, seq_no=0, data_len=0,
data=""):
        self.magic_no = magic_no       #determines if packet is dropped
        self.packet_type = packet_type #either dataPacket or
acknowledgementPacket
        self.seq_no = seq_no           #sequence number
        self.data_len = data_len       #number of bytes in the data
        self.checksum = (magic_no + packet_type + seq_no + data_len) % 1000
        self.data = data               #data carried by packet


    def __repr__(self):
        return self.__str__()


    def __str__(self):
        pt = 'unknown'
        if self.packet_type == 0:
            pt = 'data'
        elif self.packet_type == 1:
            pt = 'ack'

        s =  'Magic number: 0x{:X}\n'.format(self.magic_no)
        s += 'Packet type: {} ({})\n'.format(self.packet_type, pt)
        s += 'Seq no: {}\n'.format(self.seq_no)
        s += 'Data len: {}\n'.format(self.data_len)
        s += 'Data: "{}"'.format(self.data)
        return s
```

```
    def __len__(self):
        return len(self.encode())


    def encode(self):
        """ Returns the byte representation of the packet. """
        conv = str(self.magic_no)
        conv += str(self.packet_type)
        conv += str(self.seq_no)
        conv += "0" * (3 - len(str(self.data_len))) + str(self.data_len)
        conv += "0" * (3 - len(str(self.checksum))) + str(self.checksum)
        conv += str(self.data)
        return bytes(conv, "utf-8")


    def decode(self, data):
        """ Sets the fields of this packet to that of data. """
        try:
            data = data.decode()
            self.magic_no = int(data[:5])
            self.packet_type = int(data[5])
            self.seq_no = int(data[6])
            self.data_len = int(data[7:10])
            self.checksum = int(data[10:13])
            self.data = data[13:]
        except:
            print("Error decoding data ({}). Packet is
unchanged.".format(data))


    def is_valid_ack(self, next_no):
        """
           Checks if the packet is a valid acknowledgement packet with the
           correct sequence number, next_no.
        """
        valid_magic = self.magic_no == MAGIC_NO
        valid_type = self.packet_type == PTYPE_ACK
        valid_length = self.data_len == 0
        valid_seq_no = self.seq_no == next_no
        valid_checksum = self.checksum == (self.magic_no + self.packet_type
+ self.seq_no + self.data_len) % 1000

        return valid_magic and valid_type and valid_length and valid_seq_no
and valid_checksum


    def is_valid_data(self):
        """ Checks if the packet is a valid data packet. """
        valid_magic = self.magic_no == MAGIC_NO
        valid_type = self.packet_type == PTYPE_DATA
        valid_checksum = self.checksum == (self.magic_no + self.packet_type
+self.seq_no + self.data_len) % 1000

        return valid_magic and valid_type and valid_checksum
```

Appendix 2 – socket_generator.py

```
"""
   Socket generator
   Program to generate sending and listening sockets.
   Authors: Samuel Pell and Ollie Chick
   Date modified: 29 August 2017
"""
import socket

IP = '127.0.0.1'

def create_sending_socket(local_port, remote_port):
    """
       Creates a socket on the local_port and connects it to the
       remote_port socket, then returns that socket.
       If it fails, returns None.
    """
    try:
        new_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        new_socket.bind((IP, local_port))
        new_socket.connect((IP, remote_port))
    except IOError:
        new_socket = None

    return new_socket


def create_listening_socket(port):
    """
       Creates a socket to listen on the port given, then returns that
       socket. If it fails, returns None.
    """
    try:
        new_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        new_socket.bind((IP, port))
        new_socket.listen(1)
    except IOError:
        new_socket = None

    return new_socket
```

Appendix 3 – channel.py

```
"""
   channel
   A program for the COSC264-17S2 Assignment

   Authors: Samuel Pell and Ollie Chick
   Date Modified: 30 August 2017
"""

import socket, select, sys, packet, random
from packet import Packet, MAGIC_NO, PTYPE_DATA, PTYPE_ACK
from socket_generator import create_sending_socket, create_listening_socket

BIT_ERR_RATE = 0.1


def process_packet(data, drop_rate):
    """
       Process an input packet (as bytes) and randomly drop it or change
       its header. Returns the input packet as bytes.
       Returns the null byte if the input data is the null byte.
    """
    if data == b'':
        return data

    p = Packet()
    p.decode(data)

    if p.magic_no != MAGIC_NO:
        # magic numbers is wrong: drop it
        return None
    elif random.uniform(0, 1) < drop_rate:
        # drop packet by random chance
        return None
    elif random.uniform(0,1) < BIT_ERR_RATE:
        # create a bit error by random chance (increase data len field
randomly)
        p.data_len += random.randint(1, 10)

    return p.encode() # return the packet's byte conversion


def main_loop(sender_in, sender_out, recv_in, recv_out, drop_rate):
    """
       Wait to recieve packets on sender_in and recv_in. When it does,
       process the packet and send it on to either recv_out or sender_out
       respectively. Takes the four socket objects as arguments.

       When one of the sockets indicates it has closed it will stop
       watching it and when both sockets have closed it will return None
    """
    sockets_to_watch = [sender_in, recv_in]

    while True:
        readable, _, _ = select.select(sockets_to_watch, [], [])

        for s in readable:
            data = s.recv(1024)

            if data == b'':
```

```
                # a socket sent out the null byte (indicating it has
closed)
                if s == recv_in:
                    # receiver has closed; stop watching it
                    sockets_to_watch.remove(recv_in)
                else:
                    # sender has closed; stop watching it
                    sockets_to_watch.remove(sender_in)

                if len(sockets_to_watch) == 0:
                    # sender and receiver have closed; exit loop
                    print('\nChannel shut down.')
                    return

            elif len(sockets_to_watch) == 2:
                # sender and receiver are both open
                data_to_forward = process_packet(data, drop_rate)

                if data_to_forward != None:
                    # the packet hasn't been dropped

                    if s == sender_in:
                        # came from sender, send to receiver
                        print("sender -> channel -> receiver", end = '\r')
                        recv_out.send(data_to_forward)
                    else:
                        # came from receiver, send to sender
                        print("sender <- channel <- receiver", end = '\r')
                        sender_out.send(data_to_forward)


def main(args):
    """
       Pull the relevant numbers out of the command line arguments, check
       They are valid input, then create the appropriate sockets before
       entering into the main loop
    """
    # Check arguments are valid
    try:
        # Port numbers for this program
        sender_in_port = int(args[1])
        sender_out_port = int(args[2])
        recv_in_port = int(args[3])
        recv_out_port = int(args[4])

        # Port numbers of the sender and reciver
        sender = int(args[5])
        recv = int(args[6])

        # Probability of dropping a packet
        drop_rate = float(args[7])
    except:
        # User inputted wrong number of arguments, or non-ints/floats, etc.
        print("Usage: {} <sender_in_port> <sender_out_port> <recv_in_port>
<recv_out_port> <sender> <recv> <drop_rate>".format(args[0]))
        return

    # Check that ports are in the valid range
    for port in [sender_in_port, sender_out_port, recv_in_port, \
                 recv_out_port, sender, recv]:
        if port < 1024 or port > 64000:
```

```
            print("All port numbers should be integers in the range [1024,
64000].")
            return

    # Check that the drop rate is between 0 (inclusive) and 1 (exclusive)
    if (drop_rate >= 1) or (drop_rate < 0):
        print("drop_rate should be in the range [0, 1).")
        return

    # Create in sockets
    sender_in = create_listening_socket(sender_in_port)
    recv_in = create_listening_socket(recv_in_port)
    if None in [sender_in, recv_in]:
        sys.exit("One of the in sockets failed to be created.")

    input("Please start sender and receiver then press enter.")

    # Create out sockets and connect them
    sender_out = create_sending_socket(sender_out_port, sender)
    recv_out = create_sending_socket(recv_out_port, recv)
    if None in [sender_out, recv_out]:
        sys.exit("One of the out sockets failed to be created.")

    # Accept incomming connections to sender_in and recv_in
    try:
        sender_in, addr = sender_in.accept()
        recv_in, addr = recv_in.accept()
    except IOError:
        sys.exit("Error connecting sender_in or recv_in")

    # Enter the main loop
    main_loop(sender_in, sender_out, recv_in, recv_out, drop_rate)

    # Shut down then close all the sockets (then the program)
    sender_in.shutdown(socket.SHUT_RDWR)
    sender_in.close()
    sender_out.shutdown(socket.SHUT_RDWR)
    sender_out.close()
    recv_in.shutdown(socket.SHUT_RDWR)
    recv_in.close()
    recv_out.shutdown(socket.SHUT_RDWR)
    recv_out.close()


if __name__ == "__main__":
    # Get arguments from the command line.
    # These should be:
    # * four port numbers to use for the sockets c_s_in, c_s_out, c_r_in,
and c_r_out
    # * the port number where the socket s_in should be found
    # * the port number where the socket r_in should be found
    # * a packet loss rate P such that 0 <= P < 1

    args = sys.argv
    main(args)
```

Appendix 4 – receiver.py

```
"""
   A program to receive packets from a channel.
   For a COSC264 assignment.

   Author: Ollie Chick
   Date modified: 29 August 2017
"""

import sys, socket, os, select
from packet import Packet, MAGIC_NO, PTYPE_DATA, PTYPE_ACK
from socket_generator import create_sending_socket, create_listening_socket

def main(args):
    # Check arguments are valid
    try:
        in_port = int(args[1])
        out_port = int(args[2])
        channel_in_port = int(args[3])
        filename = args[4]
    except:
        print("Usage: {} <in_port> <out_port> <channel_in_port>
<filename>".format(args[0]))

    # Check that ports are in the valid range
    for port in [in_port, out_port, channel_in_port]:
        if port < 1024 or port > 64000:
            print("All port numbers should be integers in the range [1024,
64000].")
            return

    # Create sockets (and connect socket_out)
    socket_in = create_listening_socket(in_port)
    socket_out = create_sending_socket(out_port, channel_in_port)
    if None in [socket_in, socket_out]:
        sys.exit("One of the sockets failed to be created.")

    # Check if file exists
    if os.path.isfile(filename):
        sys.exit("Error: {} already exists.".format(filename))

    # Initialisation
    expected = 0
    file = open(filename, 'w')
    input("Please acknowledge on the channel that you have started the
receiver, then press enter.")

    # Accept connection from channel
    socket_in, addr = socket_in.accept()
    print("Receiving data...")

    # Main loop
    i = 0
    while True:
        readable, _, _ = select.select([socket_in], [], [])
        # got a response
        print("Got packet {}".format(i), end = '\r')
        i += 1
        s = readable[0]
        data = s.recv(1024)
        rcvd = Packet()
```

```
        rcvd.decode(data)

        if rcvd.is_valid_data():
            # got a valid data packet

            # Prepare an acknowledgement packet and send it
            magic_no = MAGIC_NO
            packet_type = PTYPE_ACK
            seq_no = rcvd.seq_no
            data_len = 0
            data = ""
            pack = Packet(magic_no, packet_type, seq_no, data_len, data)
            socket_out.send(pack.encode())

            if rcvd.seq_no == expected:
                expected = 1 - expected
                if rcvd.data_len > 0:
                    # has some data
                    file.write(rcvd.data)
                else:
                    # no data - indicates end of file
                    file.close()
                    socket_in.shutdown(socket.SHUT_RDWR)
                    socket_in.close()
                    socket_out.shutdown(socket.SHUT_RDWR)
                    socket_out.close()
                    print("\nData received.")
                    return


if __name__ == "__main__":
    # Get arguments from the command line.
    # These should be:
    # * two port numbers to use for the two receiver sockets r_in and r_out
    # * the port number where the socket c_r_in should be found
    # * a file name, indicating where the received data should be stored

    args = sys.argv
    main(args)
```

Appendix 5 – sender.py

```
"""
   A program to send packets to a channel.
   For a COSC264 assignment.

   Author: Ollie Chick and Samuel Pell
   Date modified: 29 August 2017
"""

import sys, socket, os, select
from packet import Packet, MAGIC_NO, PTYPE_DATA, PTYPE_ACK
from socket_generator import create_sending_socket, create_listening_socket

TIMEOUT = 1 #seconds
FILE_ENCODING = 'utf8'

def inner_loop(socket_out, socket_in, bytes_to_send, next_no):
    """
       Function to continuously send a packet until a valid acknowledgement
       packet is received. Returns the number of packets sent from sender
       to achieve successful transmission.
    """
    packets_sent = 0

    while True:
        # Send packet
        socket_out.send(bytes_to_send)
        packets_sent += 1

        # Await a response
        readable, _, _ = select.select([socket_in], [], [], TIMEOUT)

        if readable:
            # got a response
            s = readable[0]
            data = s.recv(1024)

            rcvd = Packet()
            rcvd.decode(data)

            if rcvd.is_valid_ack(next_no):
                # got a valid acknowledgement packet
                next_no = 1 - next_no
                return packets_sent, next_no


def main(args):
    # Check arguments are valid
    try:
        in_port = int(args[1])
        out_port = int(args[2])
        channel_in_port = int(args[3])
        filename = args[4]
    except:
        print("Usage: {} <in_port> <out_port> <channel_in_port>
<filename>".format(args[0]))
        return

    # Check that ports are in the valid range
    for port in [in_port, out_port, channel_in_port]:
        if port < 1024 or port > 64000:
```

```
            print("All port numbers should be integers in the range [1024,
64000].")
            return


    # Create sockets (and connect socket_out)
    socket_in = create_listening_socket(in_port)
    socket_out = create_sending_socket(out_port, channel_in_port)
    if None in [socket_in, socket_out]:
        sys.exit("One of the sockets failed to be created.")

    # Check if file exists
    if not os.path.isfile(filename):
        # file does not exist
        sys.exit("Error: {} does not exist.".format(filename))

    # Initialisation
    next_no = 0
    packets_sent = 0
    exit_flag = False
    file = open(filename, "rb")
    input("Please acknowledge on the channel that you have started the
sender, then press enter.")

    # Accept connection from channel
    socket_in, addr = socket_in.accept()
    print("Sending data...")

    # Outer loop
    i = 0
    while not exit_flag:
        # Read 512 bytes from file
        data = file.read(512)
        data = data.decode(FILE_ENCODING)

        # Prepare packet
        packet_type = PTYPE_DATA
        seq_no = next_no
        data_len = len(data)
        if data_len == 0:
            exit_flag = True
        pack = Packet(MAGIC_NO, packet_type, seq_no, data_len, data)

        # Inner loop
        bytes_to_send = pack.encode()
        print("Sending datum {}".format(i), end = "\r")
        packets_used, next_no = inner_loop(socket_out, socket_in,
bytes_to_send,
                                           next_no)
        packets_sent += packets_used
        i+=1

    # Clean up and close
    file.close()
    socket_in.shutdown(socket.SHUT_RDWR)
    socket_in.close()
    socket_out.shutdown(socket.SHUT_RDWR)
    socket_out.close()
    print("\nData sent.\nPackets sent: {}".format(packets_sent))
```

```
if __name__ == "__main__":
    # Get arguments from the command line.
    # These should be:
    # * two port numbers to use for the two sender sockets s_in and s_out
    # * the port number where the socket c_s_in should be found
    # * a file name, indicating the file whose contents should be sent

    args = sys.argv
    main(args)
```