Project Report PhaseB

MicrotTCP - A lightweight implementation of tcp

by Isidoros Chatzichrysos 4338 - Ilias Kapsis 4652 - Valerios Grammatikakis 4616

Introduction

This project is a lightweight implementation of Tcp, made using the C programming language during the winter Semester of 2023. MicroTcp was designed for the subject : HY-335/ Computer Networks, and in this paper we will analyze our implementation. It involves implementing core TCP functionalities using the UDP protocol, including error checking, packet sequencing, and retransmissions. The report will cover the goals and phases of the project, the development of the microTCP library, and its key components. It will also include screenshots of code and performance analysis

Implementation

The project was developed in two separate phases. In phase A, we implemented the following functions:

- microtcp_socket
- microtcp_bind
- microtcp_connect
- microtcp_accept
- microtcp_shutdown , as well as 2 test files.

In phase B we implemented

- microtcp_recv
- microtcp_send,

and we completed the banwith_test.c file . We followed the steps given to us by the project announcement, and we tried to implement every TCP functionality as we could .

Error Checking

Error checking is implemented within the microTCP library using UDP by incorporating checksums for each packet. The sender calculates a checksum based on the packet contents and appends it to the packet. Upon receiving the packet, the receiver computes its own checksum based on the received packet and compares it with the checksum sent by the sender to detect any transmission errors.

```c
memcpy(buff, &msgServer, sizeof(microtcp_header_t));
csum = crc32(buff, sizeof(microtcp_header_t));
//  Check if the calculated checksum matches the received checksum
if (csum != receivedChecksum)
{
  printf("Error transmitting first server packet, tcp handshake not established %u,%u\n", csum, receivedChecksum);
  socket->state = INVALID;
  return -1;
}
```

In microtcp_send we do calculate the checksum for the packet we are sending, but we haven't added a check inside microtcp_recv. We spent a lot of time trying to implement it but in the end we always encountered a bug with it and every single checksum was false even though 90% of the data was sent correctly.

The way we could've resolved it was a redesign of how we calculated checksum in send and then calculating it symmetrically in receive . We couldn't do it in time, for that reason, txt files for example open but files like .zip end up corrupted because of the checksum error.

Important notice: All types of files are transmitted normally, the transmitted file keeps its size which means we transmit all the data, just some packets are corrupted.

Packet Sequencing

The implementation of packet sequencing involves assigning sequence numbers to packets. The sender numbers each packet sequentially before transmission, and the receiver uses these sequence numbers to order incoming packets, reconstructing the original data stream. The sequence number is chosen randomly during the handshake between client and server. If a packet which is out of order arrives in microtcp_recv, we send a duplicate Ack with the actual packet number we are expecting to receive.

```c
if (header.seq_number == next_seq) //correct packet recieved
{
  // Print the values of pointers and lengths before memcpy
  if (header.data_len > indexed_bytes - sizeof(microtcp_header_t))
  {
    fprintf(stderr, "Header data length (%u) is larger than the received data size (%d).\n", header.data_len, indexed_bytes - sizeof(microtcp_header_t));
    // Handle error, maybe break or continue to the next iteration
  }

  if (total_bytes_read + header.data_len > length)
  {
    fprintf(stderr, "Not enough space in the user buffer. Can't copy %u bytes.\n", header.data_len);
    // Handle error, maybe break or continue to the next iteration
  }

  memcpy(buffer + total_bytes_read, buff + sizeof(microtcp_header_t), header.data_len);
  total_bytes_read += header.data_len;
  totalIndexedBytes += header.data_len;
  ++next_seq;
  socket->ack_number = next_seq; //update next seq number
  header.ack_number = htonl(next_seq);
  header.control = htons(ACK);
  header.window = htonl(MICROTCP_RECVBUF_LEN);
  header.checksum = htonl(0);
  header.data_len = htonl(0);
  if (sendto(socket->sd, (void *)&header, sizeof(microtcp_header_t), 0, (struct sockaddr *)&client_address, client_address_len) < 0)
  {
    perror("couldnt transmit packet\n");
    return EXIT_FAILURE;
  }
```

Retransmissions

In the event of packet loss, the microTCP library facilitates retransmissions by utilizing a timeout mechanism. When a sender does not receive an acknowledgement for a packet within a certain timeframe, it retransmits that packet.

```
timeout.tv_sec = 0;
timeout.tv_usec = MICROTCP_ACK_TIMEOUT_US;
```

```
if (setsockopt(socket->sd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(struct timeval)) < 0)
{
  perror("Set Timeout\n");
  socket->state = INVALID;
  return 0;
}
if (recvfrom(socket->sd, &header, sizeof(microtcp_header_t), 0, NULL, NULL) < 0)
{
  perror("TIMEOUT HAPPENED\n"); //timeout happened , we leave from loop , and go straight to timeout condition
  check = TIMEOUT; // timeout condition
  i--; // go back to the last correct ack
  break;
}
```

Retransmit

```
if (check == TIMEOUT)  // timeout condition
{
  socket->ssthresh = socket->cwnd / 2;
  socket->cwnd = minimum(MICROTCP_MSS, socket->ssthresh, MICROTCP_RECVBUF_LEN);
  seq = socket->seq_number - chunks + i + 1;
  data_sent = data_sent - curr_data_sent + verified_data;
  remaining -= verified_data;
  if(socket->cwnd == 0){//to never stop
    socket->cwnd = 1;
  }
}
```

Also when the receive function encounters the same ACK number 3 times in a row , we assume that a segment has been lost , and we activate the fast retransmit mechanism to retransmit the lost data.

```
else if (check == DUPLICATE3) //fast retransmit condition
{
  socket->ssthresh = socket->cwnd / 2;
  socket->cwnd = socket->cwnd / 2 + 1;
  remaining -= verified_data;
  seq = socket->seq_number - chunks + i + 1;
  data_sent = data_sent - curr_data_sent + verified_data;
  remaining -= verified_data;

}
```

After retransmitting the lost data, we enter a phase called fast recovery where the congestion window size is set to 3*MSS.

After fast recovery we enter slow start phase.

Flow Control and Congestion Control

1) Flow control is achieved through the use of techniques such as sliding window protocol to manage the flow of data between sender and receiver.

First it is established during the handshake and the window size is set to MICROTCP_WIN_SIZE.

The size of the receive buffer is defined by MICROTCP_RECVBUF_LEN and is chosen to be equal or larger that the window size , allowing for efficient data reception.

Throughout the communication session, the sliding window protocol effectively manages the flow of data packets. The sender initially transmits a packet of size X bytes, adhering to the Maximum Segment Size (MSS) constraints, with X not exceeding the size determined by the constant MICROTCP_MSS. As the receiver acknowledges the received packets, it informs the sender of its readiness to receive more data by indicating the remaining window size in the ACK packets.

```
while (data_sent < length)
{
  curr_data_sent = 0;
  verified_data = 0;
  check = CONG_SLOW; // starting condition
  size_t bytes_to_send = minimum(socket->cwnd, socket->curr_win_size, remaining); // decides how much packets to send
  chunks = bytes_to_send / MICROTCP_MSS; //how we calculate number of chunks
  for (i = 0UL; i < chunks; ++i)
  {
    memptr = (uint64_t)(buffer) + (i * MICROTCP_MSS); // buffer goes to current chunk
    //preparing packet
    header.seq_number = htonl(seq);
    header.control = htons(SYN);
    header.window = htons(MICROTCP_RECVBUF_LEN - socket->buf_fill_level);
    header.data_len = htonl(MICROTCP_MSS);
    header.checksum = htonl(crc32(memptr, MICROTCP_MSS));
    memcpy(buff, &header, sizeof(microtcp_header_t));
    memcpy(buff + sizeof(microtcp_header_t), (void *)(memptr), MICROTCP_MSS);
    if (sendto(socket->sd, buff, MICROTCP_MSS + sizeof(microtcp_header_t), 0, (struct sockaddr *)socket->servAdd, *socket->serverAddrLen) < 1) //sending it to server
    {
      perror("error transmitting packet\n");
      return EXIT_FAILURE;
    }
    socket->seq_number = seq; // saving seq number
    ++seq; // next seq we will send
  }
```

This ensures that the sender only transmits an amount of data that the receiver is capable of handling, preventing buffer overflows and potential data loss.

```
else
{
  previous_ack = header.ack_number;
  if (previous_ack == initial_seq + i) //correct ack recieved
  {
    verified_data += verified_data + header.data_len;
    if (socket->cwnd <= socket->ssthresh) // slow start
      socket->cwnd = socket->cwnd + MICROTCP_MSS;
    else // congestion avoidance
      socket->cwnd = socket->cwnd + 1;
    duplicates = 0;
    check = CONG_SLOW;
  }
}
```

The receiver, upon processing, increases its available window size, thereby allowing for the reception of additional data.

However we have not completely implemented the mechanism where the window size becomes 0 and we need to wait for a random duration between 0 and MICROTCP_ACK_TIMEOUT_US we just make the window 1 if it ever becomes 0.

2) Congestion control strategies, such as slow start and congestion avoidance, are also incorporated to regulate the transmission rate and avoid network congestion. During congestion avoidance with every RTT the congestion window increases MSS bytes. When we receive 3 duplicate ACKS we do the following actions :

When a timeout occurs we do :

ssthresh = cwnd/2;

cwnd = min(MICROTCP_MSS , ssthresh);

and the slow start algorithm is enabled.

This screenshot includes the fast retransmit as well as, congestion control red highlighted

```
else if (check == DUPLICATE3) //fast retransmit condition
{
    socket->ssthresh = socket->cwnd / 2;
    socket->cwnd = socket->cwnd / 2 + 1;
    remaining -= verified_data;
    seq = socket->seq_number - chunks + i + 1;
    data_sent = data_sent - curr_data_sent + verified_data;
    remaining -= verified_data;
}
```

Performance

Server Run

```
csd4652@DESKTOP-4PH9E16:/mnt/c/Users/ilias/csd/hy335/Final/build/test$ ./bandwidth_test   -f  a.txt   -a 127.0.0.1 -p 8000.
Starting sending data...
Data sent. Terminating...
csd4652@DESKTOP-4PH9E16:/mnt/c/Users/ilias/csd/hy335/Final/build/test$ ./bandwidth_test   -f  a.txt -m  -a 127.0.0.1 -p 8000
CONNECT DONE
server ip is 127.0.0.1
Starting sending data...
Data sent. Terminating...
sending final packet
CLIENT SUCCESFULLY TERMINATED
csd4652@DESKTOP-4PH9E16:/mnt/c/Users/ilias/csd/hy335/Final/build/test$
```

Client run

```
csd4652@DESKTOP-4PH9E16:/mnt/c/Users/ilias/csd/hy335/Final/build/test$ ./bandwidth_test  -p 8000 -s -f b.txt
Data received: 1022.509008 MB
Transfer time: 56.569419 seconds
Throughput achieved: 18.075296 MB/s
csd4652@DESKTOP-4PH9E16:/mnt/c/Users/ilias/csd/hy335/Final/build/test$ ./bandwidth_test -m -p 8000 -s -f b.txt
[ACCEPT DONE] 3-way handshake completed
entered inside
i am gonna close the connection
Connection closed successfully
Data received: 1022.509008 MB
Transfer time: 101.979396 seconds
Throughput achieved: 10.026623 MB/s
csd4652@DESKTOP-4PH9E16:/mnt/c/Users/ilias/csd/hy335/Final/build/test$
```

$$perf\ ormance = \frac{Throughput\ Microtcp}{Troughput\ Tcp} = \frac{10.02}{18.07} = 0.5545$$

Which means tcp is 82% faster than microTCP.

Evaluation

The testing and evaluation of the microTCP library were conducted with the following aspects:

- Error checking: The library's ability to detect and handle errors in data transmission. Which isn't fully implemented
- Packet sequencing: The sequencing and reassembly of packets to ensure accurate data delivery. Which is correctly implemented
- Retransmissions: The effectiveness of the library in retransmitting lost or corrupted packets. We believe we implemented it correctly, but the disappearance of some lines of the txt files show otherwise..
- Correct implementation of flow and congestion control.
- Correct packet format. Some file types are corrupted (ex. zip).
- Correct use of the info that a header contains. Sequencing is handled correctly , and the control field is used extensively in our project.

Challenges

During the implementation of the microTCP library, several challenges were encountered. We struggled with testing on the university computers. An error that only appeared in the university machines, where we couldn't find a way to fix it.

Moreover ,the task of running bandwidth test between 2 machines wasn’t accomplished. No guidelines of how to establish connection between the machines could be found on the course’s website.

Also checksum validation ended up being a challenge in the receive function. Given more time we believe we could have fixed the challenges we faced, but we sincerely believe that the final version we are delivering, satisfies the demands of the course with the way we tried to recreate all the TCP mechanisms.