Fast, Reliable File Transfer

EE542 Lab Report #4

ANTISLEEP

In networks with high latency and packet loss, traditional TCP can’t provide satisfying throughput performance. In this lab, we tried to create a file transfer utility which provides you more competitive performance than the existing solutions.

75 Mbits/s

Maximum Throughput

75 Mbits/s

Maximum Throughput

Under designed circumstance of 20% packet loss and 200ms RTT, the throughput of the network is impacted largely.

We re-designed the whole file transfer utility of Linux Socket for better speed performance. With dedicated optimization for the given network settings, the file transfer utility provides you high reliability and boosted throughput at the same time.

```
Retransmit packet id: 1483
Retransmit packet id: 5
Retransmit packet id: 379716
Retransmit packet id: 1483
Retransmit packet id: 5
Transfer file size = 8589.93 Mbits
Total file transmission time = 115.223sec
Throughput = 74.5505 Mbits/s
File transmission end
ubuntu@ip-10-0-1-172:~/lab4$
```

Throughput Performance

Our solution 75 Mbits/s

Common solution 60 Mbits/s

Traditional TCP solution 0.1 Mbits/s

Buffer Accelerate

Buffer Accelerate

With transmission data loaded into the buffer at the beginning, all IO process afterward can be extremely accelerated.

```
void readFile(const char* path, char* &buff_p) {
    // open file
    FILE *read_file;
    if((read_file = fopen(path, "rb")) == NULL){
        printf("Error opening specified file.\n");
        exit(1);
    }
    fseek(read_file, 0, SEEK_END);
    thisFileInfo.fileSize = (int)ftell(read_file);
    fseek(read_file, 0, SEEK_SET);
    thisFileInfo.packetNum = thisFileInfo.fileSize / PACKET_SIZE;
    if((thisFileInfo.fileSize % PACKET_SIZE) != 0) thisFileInfo.packetNum++;



    //map file into buffer
    // buff = (char *)malloc(thisFileInfo.fileSize + 1);
    // if (buff == nullptr) std::cerr << "bad_alloc!" << std::endl;
    // if (g_debug) std::cerr << "---DBG--- successfull allocate for " << thisFileInfo.fileSize + 1 << " memory" <<
std::endl;
    // memset(buff, 0, thisFileInfo.fileSize + 1);
    // fread(buff, 1, thisFileInfo.fileSize, read_file);
    fclose(read_file);

    g_fd = open(path, O_RDONLY);
    int flen = lseek(g_fd, 0, SEEK_END);
    buff_p = (char *)mmap(NULL, flen, PROT_READ, MAP_PRIVATE, g_fd, 0);


    cout<<"Total file size = "<< thisFileInfo.fileSize << endl;
    cout<<"Total packet number = "<< thisFileInfo.packetNum <<endl;
    if (g_debug) std::cerr << "---DBG--- successfull load file to memory" << std::endl;
}
```

Dual Thread

Dual Thread

In a network with high packet loss, the retransmission process would be verbose. With Dual Thread, we can detect the corrupted packets then retransmit them immediately and efficiently in order to improve the network performance significantly.

Sender
Receiver
readFile
createUDP
sendInfo
sendRequest
recvRequest
sendFile
send end signal
Recv lost packet#
wait for end signal
retransmit lost packet
pthread
Initial Transmission
Retransmission
createUDP
recvInfo
recvRequest
SendRequest
recvFile
recv end signal
find lost packet#
send as a packet
send end signal
receive lost packet
pthread
writeFile

Selective Retransmit

Selective Retransmit

Due to high loss rate, it is certain that we need to retransmit the packet many times. Optimize this process is vital for the performance of the utility. With selective retransmit, the time of retransmit process can be minimized and increase the throughput.

```
void *sendRetransmit(void *data) {
    endRetransmit = 0;
    int retransmitID;

    while(endRetransmit == 0) {
        for (int i = 0; i < PACKET_SIZE; i++) {
            memcpy(&retransmitID, retransmitInfo + i, 4);
            if ((retransmitID > 0) && (retransmitID <= thisFileInfo.packetNum)) {
                memset(&retransmitPacket, 0, sizeof(retransmitPacket));
                memcpy(retransmitPacket.packetData, buff + (retransmitID - 1)*PACKET_SIZE, PACKET_SIZE);
                retransmitPacket.packetID = retransmitID;
                for (int j = 0; j < RETRANSMIT_LOST_PACKET; j++) {
                    if(sendto(sockUDP, &retransmitPacket, sizeof(retransmitPacket), 0, (struct sockaddr *) &serverAddr,
sizeof(serverAddr)) == -1)
                    {
                        error("Failed to send lost packet");
                        exit(1);
                    }

                    cout<<"Retransmit packet id: "<< retransmitID <<endl;
                    usleep(PACKET_SPACE);
                }
            }

        }
        //endRetransmit = recvAll;
        memcpy(&endRetransmit, endFlag, 4);
    }
    pthread_exit(NULL);
}

void retransmit() {
    cout << "Start retransmission lost packet" << endl;

    retransmitInfo = (char *)mmap(NULL, PACKET_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, 1, 0);
    memset(retransmitInfo, 0, PACKET_SIZE);
    endFlag = (int *)mmap(NULL, 4, PROT_READ | PROT_WRITE, MAP_SHARED | MAP_ANONYMOUS, -1, 0);
    memset(endFlag, 0, 4);
    recvAll = 0;

    pthread_t thread_id;

    pthread_create(&thread_id, NULL, sendRetransmit, NULL);

    recvLostInfo();

    usleep(500);
}
```

Lost and Found

Lost and Found

Our Lost and Found mechanism is created for implementing selective retransmit. The receiver is designed to actively report the lost packet number to the sender. The lost packets can be recorded after the first round of transmission and the information will be sent back to the sender side before retransmission.

```
void sendLostInfo() {
    char isPacketRecv;
    int retransmitPacketIndex;

    while(1) {
        lostPacketCount = 0;
        retransmitPacketIndex = 0;
        memset(&lostPacketID, 0, sizeof(lostPacketID));

        // find and count the lost packet
        for (int i = 1; i <= thisFileInfo.packetNum; i++) {
            // memcpy(&isPacketRecv, receiveCount + i, 1);
            if (receiveCount[i] == false) {
                if (g_debug) std::cerr << "---DBG--- requesting for packet" << i << std::endl;
                memcpy(lostPacketID.packetData + 4*retransmitPacketIndex, &i, 4);
                retransmitPacketIndex++;
                lostPacketCount++;

                // send lost packet ID when fulfill a packet
                if (lostPacketCount % (PACKET_SIZE / 4) == 0) {
                    lostPacketID.packetID = 0;
                    for(int j = 0; j < LOST_PACKET_INFO; j++){
                        if(sendto(sockUDP, &lostPacketID, sizeof(lostPacketID), 0, (struct sockaddr *) &clientAddr,
sizeof(clientAddr)) == -1)
                        {
                            error("Failed to send lost packet info");
                            exit(1);
                        }
                        usleep(PACKET_SPACE);
                    }
                    memset(&lostPacketID, 0, sizeof(lostPacketID));
                    retransmitPacketIndex = 0;
                }
            }
        }

        cout << "Number of lost packets: " << lostPacketCount << endl;

        if(lostPacketCount == 0){
            //recvAll = 1;
            memset(endFlag, 1, 4);
            break;
        }

        // send the remain lost packet ID which not fulfill a whole packet
        if((PACKET_SIZE - (lostPacketCount * 4)) > 0){
            memset(lostPacketID.packetData + (lostPacketCount*4), 0, (PACKET_SIZE - (lostPacketCount * 4)));
        }

        for(int j = 0; j < LOST_PACKET_INFO; j++){
            if(::sendto(sockUDP, &lostPacketID, sizeof(lostPacketID), 0, (struct sockaddr *) &clientAddr,
sizeof(clientAddr)) == -1)
            {
                error("Failed to send last lost packet ID");
                exit(1);
            }
            usleep(PACKET_SPACE);
        }
    }
}
```

Customized Data Structure

Customized Data Structure

To achieve a higher throughput, we designed a customized packet structure from the ground up to meet the needs. With a special considered control flow, our new packet structure can be used as either data delivery or control signal at the same time. It works as the key part of the retransmission. We also created a file information structure dedicated for transmission control.

packet struct

fileInfo struct

MD5 Check

MD5 Check

High packet loss can cause the file corrupted after transmission. After complicated transmission process, MD5 check provides us an easy way to ensure that file is intact and complete.

```
untu@ip-10-0-1-172:~/test$ md5sum data.bin
11778c02d62890fe89c946f17db8eb  data.bin
```

```
ubuntu@ip-10-0-2-10:~/test$ md5sum data.bin
ed11778c02d62890fe89c946f17db8eb  data.bin
```

Critical Thinking

Critical Thinking

In Case 2 and Case 3, throughputs of TCP differ up to 20x. That is because running TCP in a high loss environment will trigger the window control mechanism too frequently and result in the window stays to a extremely small limit. The overflow in buffer may caused TCP also didn’t play well enough in Case 3. Throughputs of UDP are similar because though we don’t loss packets in Case 3 but the bottleneck decided the maximum throughput.

Fast, Reliable File Transfer

EE542 Lab Report #4

ANTISLEEP