Computer Networks assignment 3

Hesam Asadollahzadeh,

Borna Tavasoli,

Melika Heidari Dastjerdi

Fall 2022

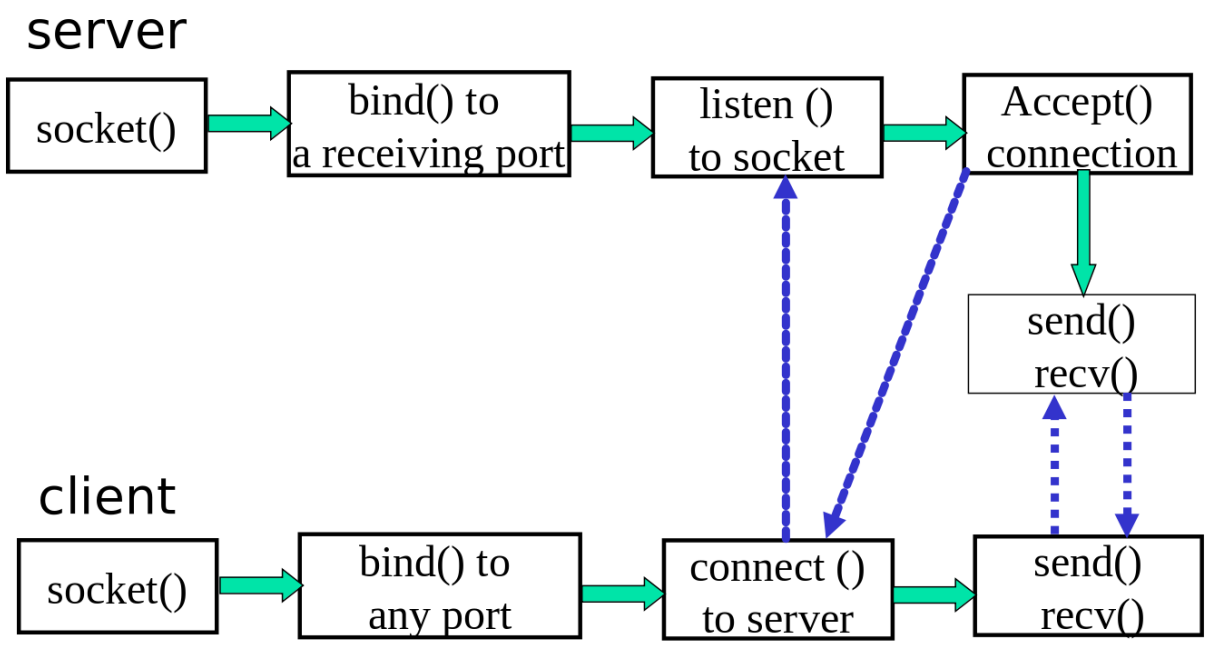
Selected protocol to handle packet drop: Go-Back-N

1. **TCP is a connection-oriented protocol, whereas UDP is a connectionless protocol.**

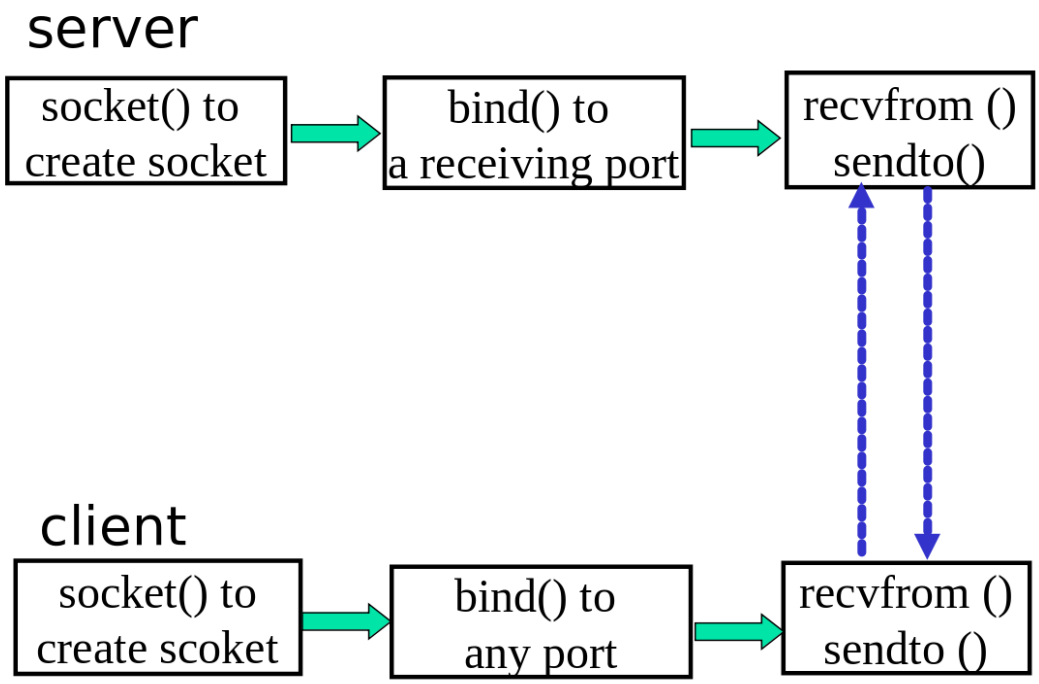
Basis	Transmission control protocol (TCP)	User datagram protocol (UDP)
Type of Service	TCP is a connection-oriented protocol. Connection-orientation means that the communicating devices should establish a connection before transmitting data and should close the connection after transmitting the data.	UDP is the Datagram-oriented protocol. This is because there is no overhead for opening a connection, maintaining a connection, and terminating a connection. UDP is efficient for broadcast and multicast types of network transmission.
Reliability	TCP is reliable as it guarantees the delivery of data to the destination	The delivery of data to the destination cannot be guaranteed in UDP.

	router.	
Error-checking mechanism	TCP provides extensive error-checking mechanisms. It is because it provides flow control and acknowledgment of data.	UDP has only the basic error checking mechanism using checksums.
Acknowledgment	An acknowledgment segment is present.	No acknowledgment segment.
Sequence	Sequencing of data is a feature of Transmission Control Protocol (TCP). this means that packets arrive in order at the receiver.	There is no sequencing of data in UDP. If the order is required, it has to be managed by the application layer.
Speed	TCP is comparatively slower than UDP.	UDP is faster, simpler, and more efficient than TCP.
Retransmission	Retransmission of lost packets is possible in TCP, but not in UDP.	There is no retransmission of lost packets in the User Datagram Protocol (UDP).
Header Length	TCP has a (20-60) bytes variable length header.	UDP has an 8 bytes fixed-length header.
Weight	TCP is heavy-weight.	UDP is lightweight.
Handshaking Techniques	Uses handshakes such as SYN, ACK, SYN-ACK	It's a connectionless protocol i.e. No handshake
Broadcasting	TCP doesn't support Broadcasting.	UDP supports Broadcasting.
Protocols	TCP is used by HTTP, HTTPs, FTP, SMTP and Telnet.	UDP is used by DNS, DHCP, TFTP, SNMP, RIP, and VoIP.
Stream Type	The TCP connection is a byte stream.	UDP connection is message stream.
Overhead	Low but higher than UDP.	Very low.

TCP:


server
socket()
bind() to
a receiving port
listen ()
to socket
Accept()
connection
send()
recv()
client
socket()
bind() to
any port
connect ()
to server
send()
recv()

UDP:


server
socket() to
create socket
bind() to
a receiving port
recvfrom ()
sendto()
client
socket() to
create scoket
bind() to
any port
recvfrom ()
sendto ()

2. در هر ردیف در جدول پایین، ستون خانه ای که سبز شده است مشخص می‌کند که کدام الگوریتم مزیت دارد. ردیف های زرد شده یعنی هر دو الگوریتم در معیار ردیف یکسان هستند. باقی معیار ها صرفا برای مقایسه دو روش هستند.

مورد مقایسه	Selective-repeat	Go-Back_N
Window size	For sender: N For reciever: N	For sender: N For reciever: 1
Complexity	The receiver window must sort the frames so it has higher complexity.	Easier implementation since we don't need to sort the received data. (window size = 1)
Efficiency	Efficiency = N/(1+2a) a = propagation delay/transmission delay N = number of packets sent	Efficiency = N/(1+2a) a = propagation delay/transmission delay N = number of packets sent
Acknowledgement	individual	cumulative
Out-of-Order packets	Out-of-order packets will be accepted and sorted on the sender side.	Out-of-order packets won't be accepted and the whole window for sender has to be sent again.
Minimum Sequence Number	2N	N+1
Bandwidth consumed for data packets	Doesnt use extra bandwidth since it only resends the lost packets.	Uses high rate of bandwidth since for every lost packet, the whole window has to be resent, so if the error-rate decreases, most of the

		bandwidth would be used for duplicate data.
Bandwidth consumed for ack packets	Since all acks are sent individually, one ack is sent for each packet and more bandwidth is used.	Since all acks are sent cummolatively, less number of acks are sent and less bandwidth is used for acks.

یکی دیگر از معایب Go-Back-N که به خاطر cumulative بودن ack ها به وجود می‌آید این است که اگر یک پکت ack از بین برود، برای چند پکت متوجه نمی‌شویم که ack شده اند. در حالیکه در selective-repeat فقط برای یک پکت این مشکل پیش می‌آید.

۲.

برای هندل کردن packet loss از Go-Back-N استفاده کردیم.

طرف فرستنده(A)

```
sendto(sockfd, num_char, BUFFER_SIZE+1,
       MSG_CONFIRM, (const struct sockaddr *) &servaddr,
           sizeof(servaddr));

pthread_create(&tid, nullptr, recieveAck, nullptr);

while(lfs < frames.size()) {
    if(lfs-lar < SWS) {
        string frameIndexString = to_string(lfs+1);
        int n = frameIndexString.length()+frames[lfs].length()+1;
        char char_array[n + 1];
        strcpy(char_array, (frameIndexString+" "+frames[lfs]).c_str());

        writeSendLog(lfs, char_array);
        sendto(sockfd, char_array, BUFFER_SIZE+1,
        MSG_CONFIRM, (const struct sockaddr *) &servaddr,
            sizeof(servaddr));
        lfs++;
    }
}
```

```
    pthread_join(tid, nullptr);
    close(sockfd);
```

در ابتدا تعداد فریم هایی که قرار است فرستاده شود به روتر و گیرنده اطلاع داده می‌شود.(برای اینکه در انتهای عملیات همه با هم بسته شوند) همانطور که در کد بالا مشاهده می‌شود، یک حلقه while می‌زنیم و تا وقتی که last frame sent برابر با تعداد فریم ها نشده است، پیش می‌رویم. در هر بار اجرای حلقه، در صورتی که lfs -lar با سایز window برابر بود یعنی باید منتظر یک ack بمانیم تا بتوانیم پکت بعدی را بفرستیم. برای همین هیچ پکتی را ارسال نمی‌کنیم. در غیر اینصورت یک پکت ارسال کرده و lfs را یک واحد افزایش می‌دهیم. سپس باید عملیات دریافت ack انجام شود. اما از آنجایی که پروتکل Go-Back-N اجازه فرستادن چند پکت بدون دریافت ack را می‌دهد، عملیات receive نباید جلوی فرستادن پکت های بعدی را بگیرد. پس برای اجرای این عملیات یک thread جدید تشکیل می‌دهیم و در آن thread منتظر دریافت ack می‌مانیم.

```
void alarm_handler(int a) {
    cout << "time out occured!\n\n";
    unregisteredAcks.clear();
    lfs = lar;
}

void* recieveAck(void* args) {
    while(lar < frames.size()) {
        struct itimerval it_val;
        it_val.it_value.tv_sec =     INTERVAL/1000;
        it_val.it_value.tv_usec =    (INTERVAL*1000) % 1000000;
        it_val.it_interval = it_val.it_value;
        memset(&buffer, 0, sizeof(buffer));
        signal(SIGALRM, alarm_handler);
        //alarm(TIME_OUT_DUR);
        setitimer(ITIMER_REAL, &it_val, NULL);
        recvfrom(sockfd, (char *)buffer, BUFFER_SIZE+1,
            MSG_WAITALL, ( struct sockaddr *) &servaddr,
            (socklen_t*)&servaddr);
        int SN = aquireFrameIndex(buffer);
        cout << "ack received for packet " << atoi(buffer) << endl << endl;
        cerr << lar << " " << SN << endl;
        if(lar+1 == SN) {
            cout << "sliding the window to the right...\n";
            sort(unregisteredAcks.begin(), unregisteredAcks.end(), greater <>());
            lar++;
            for(int i = unregisteredAcks.size()-1; i >= 0; i--)
                if(lar+1 == unregisteredAcks[i]) {
                    lar++;
                    unregisteredAcks.pop_back();
                }
```

```
        } else {
            unregisteredAcks.push_back(SN);
        }
    }
    char *transmitComplete =  (char*)(to_string(frames.size()+1).c_str());
    sendto(sockfd, transmitComplete, BUFFER_SIZE+1,
           MSG_CONFIRM, (const struct sockaddr *) &servaddr,
               sizeof(servaddr));
    return nullptr;
}
```

همچنین برای دریافت ack برای هر پکت یک timeout قرار می‌دهیم. این timeout با سیگنال پیاده سازی شده است. به این این صورت که اگر بعد از گذشتن زمان به اندازه TIME_OUT_DUR هنوز روی receive بلاک شده بودیم، تابع alarm_handler این timeout را اعلام کرده و پنجره را به عقب شیفت می‌دهد تا دوباره کل window از اول فرستاده شود.

در هر بار فرستادن یک پکت، sequence number آن پکت به آن concat می‌شود.

در فرستنده ack ها، پنجره تنها در صورتی جابه جا می‌شود که ack مربوط به پکتی که lar+1 به آن اشاره می‌کند برسد. در غیر اینصورت lar را تغییر نمی‌دهد اما ack ها رسیده را در یک آرایه به نام unregisteredAcks ذخیره می‌کند. از آنجایی که سایز پنجره فرستنده ۴ است یعنی باید اجازه رسیدن پکت ها خارج از ترتیب را بدهد. به همین دلیل پکت هایی که ack آنها قبل از ack پکت lar+1 رسیده اند ذخیره می‌شوند. در صورتی که ack پکت lar+1 قبل از timeout برسد، دیگر لازم نیست برای پکت های بعدی که ack آنها قبلا رسیده بود، مجدد ack دریافت شود و پنجره به اندازه همه آنها جا به جا می‌شود. البته اگر مثلا unregisteredAcks به صورت {۲ ۴} باشد، و سپس ack پکت ۱ برسد، lar تا ۲ پیش می‌رود و سپس منتظر ۳ ack پکت می‌ماند. در صورت رسیدن ack پکت ۳، lar به ۴ می‌رود.

در صورتی که مثلا ack پکت ۱ قبل از timeout نرسد، دیگر ack ها بعد از آن معنی ندارند و باید دوباره پنجره از اول فرستاده شود. پس unregisteredAcks خالی می‌شود.

طرف دریافت کننده(B)

```
recvfrom(sockfd, (char *)buffer, BUFFER_SIZE,
                MSG_WAITALL, ( struct sockaddr *) &cliaddr,
                (socklen_t*)&len);

   int numOfFrames = atoi(buffer);
```

```
cout << numOfFrames << endl;

while(lfr < numOfFrames) {
    memset(&buffer, 0, sizeof(buffer));
    cerr << "B: receiving \n";
    recvfrom(sockfd, (char *)buffer, BUFFER_SIZE,
            MSG_WAITALL, ( struct sockaddr *) &cliaddr,
            (socklen_t*)&len);

    cerr << buffer << endl;

    int frameIndex = aquireFrameIndex(buffer);

    cout << "received packet sequence number is " << frameIndex << endl;
    cout << "Packet content is: \n";
    cout << buffer << endl;

    if(frameIndex <= lfr+1) {
        char *ack =  (char*)(to_string(frameIndex).c_str());

        sendto(sockfd, ack, strlen(ack),
            MSG_CONFIRM, (const struct sockaddr *) &cliaddr,
                len);
        if(frameIndex == lfr+1) {
            lfr++;
            cout << "ack sent for packet " << frameIndex << endl;
        } else
            cout << "ack resent for packet " << frameIndex << endl;
    }
    cout << endl;
}
```

در ابتدا تعداد پکت هایی که قرار است ارسال شوند از روتر گرفته می شود.

سایز پنجره در طرف گیرنده ۱ است. به این ترتیب پکت ها خارج از ترتیب دریافت نمی‌شوند و احتیاجی به sort کردن پنجره دریافت کننده نیست. از آنجایی که سایز پنجره یک است، نیازی به نگه داشتن دو پوینتر برای دو سر پنجره نیست و فقط پوینتر lfr نگه داشته شده است. با دریافت پکت ابتدا sequence number آن را از اولش جدا کرده و می‌خوانیم. در صورتی که این شماره با lfr+1 برابر بود یعنی پکت در پنجره گیرنده قرار دارد و در نتیجه گیرنده ack آن را ارسال میکند. در غیر اینصورت پیام ack ارسال نمی‌شود و پکت هم دریافت نمی‌شود.

از آنجایی که ممکن است پکت ack هم توسط روتر drop شود، حتی برای پکت هایی که به دست B هم رسیده اند می‌توانند در طرف فرستنده timeout شوند و دوباره ارسال شوند. پس باید برای این پکت ها از سمت B مجدد پکت ack ارسال شود که چرخه

timeout ادامه پیدا نکند. برای هندل کردن این مساله اگر یک پکت رسید که sequence number آن از lfr+1 کوچکتر بود، یعنی پکتی است که دریافت شده و ack آن به دست A نرسیده. برای این پکت ها مجدد ack با پیام متفاوت ارسال می‌شود اما داده های آنها مجدد ذخیره نمی‌شود.

۰۳

اگر در ارسال و دریافت پیام توسط روتر هیچ خطایی نداشته باشیم و زمان انتقال پیام هم تقریبا یکی باشد، می‌تواند نتیجه گرفت که به ازای هر پکت ارسالی، ack آن بلافاصله دریافت می‌شود. پس همیشه فاصله lar و lfs یک است.

با اجرای برنامه، فایل یک مگابایتی به ۶۸۳ پکت یک و نیم کیلو بایتی تقسیم شده و ارسال می‌شود. بخشی از نتایج ثبت شده توسط فایل ها به صورت زیر است:

برای A:

```
lar and lfs are: 681 682
sending packet with sequence number 682
packet content is:
682 462508528270698881282293013821846637452402315984734969903099873124429031624887761612735499494351478416043024959196016615882742830122973352334874876869327916725048489398915713428049162676158971668102772531708192335206637574282302481044051005388546609274299123919446207336590235297840044149569787830675448852079217813254125078565940555357151112153929694801251132171203795549941163920747617536901298055993811831099011084017459914432588387806536110562815283778208909313580608280517256206761724523177106183887958947684699198712572849021955925163448332238519286917786966547590475837028485455090808992541529135984272160073022623552235846075052239277572270449478667844784629193508957236434028423841383547401202560790562268848472666933520025809536945273481956952236071064705073558122375415417200921965563405533347843275197814453910296613637292816403322858277400213988846375567124674868467361795016474572838515685283202962908042906475735120765699184181768011549531548203964447954894448843978831165318769281707481984071449736153863148471821045992638104240396092287658789941231792157150943672833273030209880065897929747213683338415136587003149454336344563685020221252259142980926069027074077129168059113331677771782227727933843623094566219277157186574916058687802285827552124321505079455861793177558832003907432025223801480259783659063482422724970348352957418600618397550938600084844369264735850122514236116739426797080281609958666072657715644232036778799686799299853700442673482403206170333790725867563734900546052302194099038667954693947456728385156852832029629080429064757351207656991841817680115495315482039644479548944488439788311653187692817074819840714497361538631484718210459926381042403960922876587899412317921571509436728332730302098800658979297472136833384151365870031494543363445636850202212522591429809260690270740771291680591133316777717822277279338436230945662192771571865749160586878022858275521243215050794558617931775588320039074320252238014802597836590634824227249703483529574186006183975509386000848443692647358501225142361167394267970802816099586660726577156442320367787996867992998537004426734824032061703337907258675637349005460523021940990386679546969474567283851568528320296290804290647573512076569918418176801154953154820396447954894448843978831165318769281707481984071449736153863148471821045992638104240396092287658789941231792157150943672833273030209880065897929747213683338415136587003149454336344563685020221252259142980926069027074077129168059113331677771782227727933843623094566219277157186574916058687802285827552124321505
```

برای B:

```
received packet sequence number is 682
Packet content is:
682 462508528270698881282293013821846637452402315984734969903099873124429031624887761612735499494351478416043024959196016615882742830122973352334874876869327916725048489398915713428049162676158971668102772531708192335206637574282302481044051005388546609274299123919446207336590235297840044149569787830675448852079217813254125078565940555357151112153929694801251132171203795549941163920747617536901298055993811831099011084017459914432588387806536110562815283778208909313580608280517256206761724523177106183887958947684699198712572849021955925163448332238519286917786966547590475837028485455090808992541529135984272160073022623552235846075052239277572270449478667844784629193508957236434028423841383547401202560790562268848472666933520025809536945273481956952236071064705073558122375415417200921965563405533347843275197814453910296613637292816403322858277400213988846375567124674868467361795016474572838515685283202962908042906475735120765699184181768011549531548203964447954894448843978831165318769281707481984071449736153863148471821045992638104240396092287658789941231792157150943672833273030209880065897929747213683338415136587003149454336344563685020221252259142980926069027074077129168059113331677771782227727933843623094566219277157186574916058687802285827552124321505
```

مشاهده می‌شود که دو پکت آخر به درستی ارسال شده اند و محتوای آنها یکسان است. همچنین مقادیر lar و lfs نیز مطابق انتظار است. برای مشاهده لاگ کامل کامپیوتر های A و B به ترتیب به فایل های زیر مراجعه کنید:

1-3-a.txt , 1-3-b.txt

زمان انتقال داده یک مگابایتی:

```
Duration of transmission for a 1MB file sample.txt: 72914 microseconds
```

4. در این بخش برای فرستادن داده ها روتر به اندازه ۵ میلی ثانیه منتظر می‌ماند و A هم بعد از 100 میلی ثانیه timeout اعلام می‌کند. زمان timeout با توجه به اندازه بافر و تاخیر ۵ میلی ثانیه ای روتر انتخاب شده است. به عنوان مثال اگر پکتی که می‌خواهیم ارسال کنیم آخرین خانه queue باشد، و پکت ack آن هم در آخرین خانه queue قرار بگیرد، با توجه به تاخیر بافر روتر، باید 5*10 میلی ثانیه برای رفت پکت داده و 5*10 میلی ثانیه برای برگشت ack زمان بگذرد. اما از آنجایی که همیشه دقیقا بعد از دریافت یک پکت ack آن توسط B فرستاده می‌شود و A هم حداکثر می‌تواند سه پکت بعد از پکت فعلی که هنوز ack آن نرسیده ارسال کند(با توجه به سایز پنجره اش)، یعنی پکت ack حداکثر پکت چهارم در مسیر برگشت خواهد بود. یعنی ۱۰۰ میلی ثانیه یک حد بالا برای زمان انتظار برای پکت ack محسوب شده و می‌توان بعد از گذشت آن زمان با اطمینان اعلام کرد که drop شده است.

فایل های A و B نسبت به بخش قبل بی تغییر باقی مانده اند. بخشی از تابع main روتر به صورت زیر است:

```
// making threads for sending and receiving
  pthread_create(&tid[0], nullptr, sendPackets, nullptr);
  pthread_create(&tid[1], nullptr, receiveA, nullptr);
```

```
    pthread_create(&tid[2], nullptr, receiveB, nullptr);

    // waiting for send tread to close
    pthread_join(tid[0], NULL);
    pthread_cancel(tid[1]);
    pthread_cancel(tid[2]);
```

فایل روتر پس از دریافت تعداد پکت ها سه thread برای فرستادن پکت، دریافت پکت از A و دریافت پکت از B ایجاد می‌کند. به این ترتیب ارسال و دریافت از هیچ طرفی، باعث بلاک شدن طرف دیگر و عملیات های دیگر نمی‌شود.

همچنین روتر یک ساختار پکت دارد که در آن محتوای پکت، sequnece number آن و مقصدش ذخیره می‌شود:

```
struct Packet {
    Packet(char _body[], bool _dest, int frameSN): dest(_dest), SN(frameSN) {
        for(int i = 0; i < MAXLINE; i++)
            body[i] = _body[i];
    };
    char body[MAXLINE];
    bool dest; //0 for A and 1 for B
    int SN; // sequence number
};
```

بافر روتر، یک آرایه از این ساختار است.

تابع sendPackets به صورت زیر است:

```
void* sendPackets(void* args) {
    while(true) {
        cerr << "size is " << packetsBuffer.size() << endl;
        while(packetsBuffer.size() == 0); // busy waiting
        sleep_for(std::chrono::milliseconds(5));

        Packet p = packetsBuffer.front();
        //pthread_mutex_lock(&lock);
        if(p.dest) {
            std::cout << "forwarding packet " << p.SN << " to B...\n";
            if(p.SN == numOfFrames+1)
                break;
            sendto(sockfd2, p.body, MAXLINE,
                MSG_CONFIRM, (const struct sockaddr *) &servaddr2,
                sizeof(cliaddr2));
        } else {
            std::cout << "forwarding ack for packet " << p.SN << " to A...\n";
            sendto(sockfd1, p.body, MAXLINE,
                MSG_CONFIRM, (const struct sockaddr *) &cliaddr1,
                sizeof(cliaddr1));
        }
        packetsBuffer.pop();
```

```
        cout << "end of forward\n";
        sent++;
        //pthread_mutex_unlock(&lock);
    }
    return nullptr;
}
```

این تابع همواره در حال اجراست. اگر queue خالی باشد، هیچ کاری نمی‌کند تا پر شود. سپس پکت های داخل آن را با فاصله زمانی 5 میلی ثانیه به مقصد ذخیره شده در پکت ارسال می‌کند. برای سیگنال دادن پایان فرآیند ارسال از سمت A پس از دریافت ack آخرین پکت، یک پکت با sequence number بیشتر از تعداد پکت های ارسالی که در ابتدا اعلام کرده بود فرستاده می‌شود. روتر هنگام ارسال این پکت متوجه می‌شود که فرآیند ارسال تمام شده و thread ارسال را به اتمام می‌رساند. همانطور که در بالا مشاهده شد با join شدن thread ارسال، thread های دریافت هم بسته شده و کار روتر خاتمه می‌یابد.

توابع receiveA و receiveB مشابه هستند. تابع receiveA به صورت زیر است:

```
void* receiveA(void* args) {
    socklen_t len1 = sizeof(cliaddr1);
    while(true) {
        memset(&recA, 0, sizeof(recA));
        recvfrom(sockfd1, (char *)recA, 1537,
            MSG_WAITALL, ( struct sockaddr *) &cliaddr1,
            (socklen_t*)&len1);

        addPacket("Adding packet ", " from A to queue...",
         "dropping data packet ", 0, recA);
    }
    return nullptr;
}
```

این تابع با دریافت یک پکت با استفاده از تابع addPacket آن را به queue اضافه می‌کند:

```
void addPacket(string mssg1, string mssg2, string mssg3, int sender, char recBuffer[]) {
    char temp[MAXLINE];
    copyBuffer(recBuffer, temp);
    int frameSN = aquireFrameIndex(temp);

    int loss = distrib(gen);
    if(loss == 1) {
        std::cout << mssg3 << frameSN << endl;
        return;
    }

    if(packetsBuffer.size() < PACKET_BUFFER_SIZE) {
        std::cout << mssg1 << frameSN << mssg2 << endl;
        packetsBuffer.push(Packet(temp, !sender, frameSN));
    } else
```

```
        std::cout << mssg3 << frameSN << endl;
}
```

تابع addPackat به احتمال ۱۰ درصد پکت را drop می‌کند. اگر پکت drop نشود، در صورتی که بافر روتر پر نشده باشد، پکت را به صف اضافه کرده و در غیر اینصورت آن را drop می‌کند.

با فرستادن یک فایل یک مگابایتی و ذخیره لاگ ها، درستی عملکرد این سیستم مشاهده می‌شود.

بخشی از لاگ A:

در

```
sliding the window to the right...
sending packet with sequence number 670
ack received for packet 667

sliding the window to the right...
sending packet with sequence number 671
time out occured!

sending packet with sequence number 668
sending packet with sequence number 669
sending packet with sequence number 670
sending packet with sequence number 671
ack received for packet 668

sliding the window to the right...
sending packet with sequence number 672
ack received for packet 669

sliding the window to the right...
sending packet with sequence number 673
ack received for packet 670

sliding the window to the right...
sending packet with sequence number 674
ack received for packet 671
```

این بخش ack مربوط به پکت ۶۶۸ دریافت نشده و پیغام timeout occured چاپ شده است. سپس مجدد پنجره شامل این پکت فرستاده شده است. با دریافت هر ack پنجره یک واحد به سمت راست انتقال پیدا کرده است و پکت بعدی ارسال شده است. برای چک کردن درستی در فایل روتر چک می‌کنیم آیا پکت ۶۶۸ یا ack آن دراپ شده اند یا خیر.

بخشی از لاگ روتر:

```
Adding packet 668 from A to queue...
forwarding ack for packet 665 to A...
end of forward
dropping data packet 669
forwarding ack for packet 666 to A...
end of forward
Adding packet 670 from A to queue...
forwarding ack for packet 667 to A...
end of forward
Adding packet 671 from A to queue...
forwarding packet 668 to B...
end of forward
dropping ack packet for packet 668
forwarding packet 670 to B...
end of forward
forwarding packet 671 to B...
end of forward
Adding packet 668 from A to queue...
Adding packet 669 from A to queue...
Adding packet 670 from A to queue...
Adding packet 671 from A to queue...
forwarding packet 668 to B...
end of forward
Adding ack for packet 668 from B to queue...
forwarding packet 669 to B...
end of forward
```

مشاهده می‌شود که پکت ۶۶۸ ابتدا دریافت و به صف اضافه شده است. سپس به B فرستاده شده است و ack آن دریافت شده است اما drop شده و به صف اضافه نشده است. این موضوع باعث رخداد timeout در A شده و منجر شده است که پکت های پنجره دوباره به سمت روتر فرستاده شوند. بعد از forward شدن پکت ۶۷۱، چهار پکتی که در پنجره A قرار داشتند دریافت شده اند.

بخشی از لاگ B:

```
received packet sequence number is 668
Packet content is:
668 05129860514260354867904888339069915
85775401216311924130422013370079298732
57381457340139018470279737659767876827
75622078363568063179216778235359028956
66851444086303960333743845218310233483
75766098287829013493402725291576441171
73879981318085875297318017815989110467
62485764648811895698426933295691805999
05345708882441297255192846144669945941
78527219583352481707618291333262467977
50883703605452004337350930906887916653
64351871293686144499375805382969335317
ack sent for packet 668

received packet sequence number is 670
Packet content is:
670 64072482727222015829548898738407967
49539413303962688150994099772376150275
44266301245531426872732053306442815117
26796224155037445857765605493260244394
65494520782951959528519721018928687524
30957140109460583671154051116755882416
98054652986217390620111521778965945807
56358530834654437523993858096135326280
13970854943424381600669502141617262163
```

مشاهده می‌شود که پکت ۶۶۸ دریافت شده و ack آن ارسال شده است. اما پکت بعدی که رسیده پکت ۶۷۰ است.(در لاگ روتر مشخص است که ۶۶۹ drop شده است.) از آنجایی که پنجره B اجازه رسیدن پکت ها خارج از ترتیب را نمی‌دهد، ack برای پکت ۶۷۰ ارسال نشده است. در تصویر بعد مشاهده می‌شود که با فرستاده شدن مجدد پنجره ۶۶۸ ۶۶۹ ۶۷۰ ۶۷۱، ack مربوط به پکت ۶۶۹ هم ارسال شده است. همچنین ack مربوط به ۶۶۸ مجدد ارسال شده است:

```
received packet sequence number is 668
Packet content is:
668 05129860514260354867904888339069915
85775401216311924130422013370079298732
57381457340139018470279737659767876827
75622078363568063179216778235359028956
66851444086303960333743845218310233483
75766098287829013493402725291576441171
73879981318085875297318017815989110467
```

در این شکل پکت ۶۶۸ چون ack اش نرسیده بود مجدد دریافت

شده است و لاگ ack آن با resent مشخص شده است.

همچنین پکت ۶۶۹ به صورت موفقیت آمیز دریافت شده و ack

آن ارسال شده است.

برای مشاهده کامل نتایج لاگ های A و B و Router به ترتیب

به فایل های

1-4-a.txt , 1-4-b.txt , 1-4-r.txt

مراجعه کنید.

قسمت دوم

۰۱

ساده ترین روش برای برخورد با congestion مکانیزم tail-drop است. در این مکانیزم روتر تا زمانی که بافر پر شود اقدامی انجام نمی‌دهد و بعد از پر شدن بافر پکت ها را drop می‌کند تا در بافر جا به وجود بیاید. دو ایراد عمده این مکانیزم این است که فضای بافر را به صورت unfair بین جریان های ترافیک تقسیم می‌کند و همچنین اینکه همه ی اتصالات TCP وقتی متوجه می‌شوند که چند تا از پکت آنها drop شده با هم hold back می‌کنند که باعث خالی شدن بافر می‌شود و باز با خالی شدن بافر، همه ی فرستنده ها پکت های خود را می‌فرستند و یک سیکل معیوب به وجود می‌آید. برای رفع این مسائل، از مکانیزم کنترل ازدحام RED استفاده می‌شود. این مکانیزم به جای اینکه فقط وقتی بافر پر شده، پکت ها را دراپ کند، ابتدا میانگین سایز صف را پیدا می‌کند.

اگر بافر به نسبت این سایز خالی تر باشد، پکت ها را قبول می‌کند و اگر بافر پرتر باشد با احتمال مشخصی پکت ها را drop می‌کند. هر چه بافر پر تر شود این احتمال بیشتر شده و اگر بافر پر باشد احتمال drop یک می‌شود.

۲.

بعد از یکبار اجرای این برنامه با حداکثر SWS = 4، تاخیر ارسال ۵ میلی ثانیه و timeout برابر با ۲۰۰ میلی ثانیه، بدون مکانیزم RED بررسی کردیم که میانگین سایز صف چیزی بین ۱۲-۱۷ فریم است. پس میانگین را ۱۲ قرار دادیم. از ۱۲ به بعد با اضافه شدن هر پکت احتمال drop به اندازه مشخصی افزایش پیدا می‌کند. هنگامی که سایز از ۱۶ بیشتر می‌شود، احتمال drop از پنجاه درصد بیشتر می‌شود و در پایان زمانی که سایز صف بیست و در واقع بافر پر است، احتمال drop به ۱ می‌رسد.

در هر بار دریافت یک پکت جدید با استفاده از تابع updateDropProb، احتمال drop بر اساس سایز فعلی صف محاسبه شده و اعمال می‌گردد:

```
void updateDropProb() {
   dropProbability = packetsBuffer.size() >= MAX ? 10 :
                     packetsBuffer.size() - MIN > 5 ? (packetsBuffer.size() - MIN)/2+5 :
                     (packetsBuffer.size() - MIN)/2;
}
```

برای اینکه کامپیوتر های A بتوانند به ازدحام شبکه واکنش نشان داده و نرخ فرستادن داده را کاهش دهند از تکنیک AIMD استفاده کردیم. در صورتی که در یک کامپیوتر A تعداد timeout های متوالی بیشتر از ۲ عدد شود، برداشت می‌کند که شبکه در حال حاضر شلوغ است و سایز پنجره خود را نصف می‌کند. همچنین در صورت دریافت یک ack می‌فهمد که شبکه خلوتتر شده است و سایز پنجره خود را یک واحد افزایش می‌دهد. به این ترتیب توانستیم از وقوع congestion در شبکه جلوگیری کنیم و میانگین سایز صف همان ۱۵-۱۷ پکت ارزیابی شد. همچنین با انجام این عملیات نسبت به حالت قبل(بدون اعمال RED)، عملیات فرستادن فایل ها با سرعت بیشتری به اتمام رسید.

برای اینکه کامپیوتر B بتواند با چند مبدا هم درست کار کند به جای یک lfr از ۲۰ lfr استفاده کردیم. هر کدام از این lfr ها مربوط به یک کامپیوتر می‌شوند. همچنین هر پکت علاوه بر sequence number، مبدا خود را هم ذخیره می‌کند که B با استفاده از این اطلاعات متوجه می‌شود lfr مربوط به کدام کامپیوتر را جا به جا کند.

برای اجرا این برنامه از برنامه موجود در فایل driver.cpp استفاده شده است. این برنامه علاوه بر اجرای روتر و کامپیوتر B، بیست بار به صورت موازی کامپیوتر A را اجرا می‌کند و به هر کدام از اجراها شماره خاص خود را نسبت می‌دهد. نتیجه لاگ های این کامپیوتر

ها در فایل ها a1.txt تا a20.txt و نتیجه لاگ های b و router به ترتیب در b.txt و r.txt ذخیره می‌شوند. از آنجایی که حجم این لاگ ها بسیار زیاد شده است در فایل ارسالی تنها لاگ های مربوط به یکی از A ها و فایل دریافتی B ذخیره شده است.

توضیح درستی:

بخشی از نتایج کامپیوتر a17:

```
sending packet with sequence number 683
time out occured!

sending packet with sequence number 683
17 16
buffer:635 16
17 16
buffer:636 16
17 17
buffer:683 17
ack received for packet 683

Duration of transmission for a 1MB file sample.txt: 2720166546 microseconds
```

مشخص است که ack پکت 683 به صورت موفقیت آمیز دریافت شده است و زمان اجرا هم چاپ شده.

بخشی از نتایج کامپیوتر a3:

```
sliding the window to the right...
sending packet with sequence number 683
3 3
buffer:682 3
ack received for packet 682

sliding the window to the right...
3 3
buffer:683 3
ack received for packet 683

Duration of transmission for a 1MB file sample.txt: 2859272190 microseconds
```

مشخص است که ack پکت 683 به صورت موفقیت آمیز دریافت شده است و زمان اجرا هم چاپ شده.

باقی کامپیوتر ها نیز به همین صورت ack پکت آخر را دریافت کرده اند و زمان را یادداشت کرده اند.

حاصل دریافتی های B از هر کامپیوتر در یک آرایه به نام files ذخیره شده است. در انتهای برنامه files نتایج دریافتی های خود از یک کامپیوتر را نمایش می‌دهد. این نتایج در فایل sample_remade.txt ذخیره شده است. برای هر پکت sequence number، مبدا و سایز آن نیز نمایش داده شده است.

اعداد پیشفرض در این پروژه توسط اجراهای متوالی و آزمون و خطا بدست آمده اند. در این اجراها سعی کردیم استفاده از فضای صف را افزایش و میزان timeout ها را کاهش دهیم. همچنین در پایان زمان انتقال ها را مقایسه کرده و شیوه ای را انتخاب کردیم که کمترین زمان انتقال را داشت.

با درنظر داشتن پیشفرض های ذکر شده، از آنجایی که همه کامپیوتر ها به صورت موازی اجرا می‌شوند زمان انتقال کل برابر با بیشترین زمان انتقال می‌شود که در کامپیوتر ۳ اتفاق افتاد:

```
Duration of transmission for a 1MB file sample.txt: 207963226 microseconds
```

(حجم فایل لاگ روتر خیلی زیاد بود. برای همین تنها دو تا از کامپیوتر های a و پکت های دریافتی از کامپیوتر بیستم در فایل های 2-2-a3.txt, 2-2-a17.txt, 2-2-out.txt

قرار داده شده اند.)