

به نام خدا

## گزارش پروژه چهارم درس شبکه های کامپیوتری

سارا رضائی منش ۸۱۰۱۹۸۵۷۶

یرنا توسلی ۸۱۰۱۹۸۳۷۴

## عنوان پروژه

آشنایی با کارکرد TCP و پیاده سازی مکانیسم ارسال درست اطلاعات همراه با کنترل ازدحام در شبکه

## قسمت اول

۰۱

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

مورد مقایسه	Go-Back_N	Selective-repeat
Window size	برای فرستنده: N برای گیرنده: 1	برای فرستنده: N برای گیرنده: N
Complexity	چون نیازی به sort کردن داده های دریافتی نداریم (چون سایز window دریافت ۱ است)، پیاده سازی آن ساده تر است.	در دریافت کننده Receiver window باید فریم ها را sort کند. پس پیاده سازی آن سخت تر است.
Efficiency	$\text{Efficiency} = N/(1+2a)$ a = propagation delay/transmission delay N = number of packets sent	$N/(1+2a)$ a = propagation delay/transmission delay N = number of packets sent
Acknowledgement	cumulative	individual
Out-of-Order packets	پکت هایی که خارج از ترتیب فرستاده می شوند accept نمی شوند و کل window طرف sender باید مجدد ارسال شود.	پکت هایی که خارج از ترتیب فرستاده می شوند accept میشوند و سپس در طرف گیرنده sort می شوند.
Minimum Sequence Number	N+1	2N
Bandwidth	Bandwidth زیادی مصرف می کند. چراکه	از آنجایی که فقط پکت هایی که lost

می شوند را مجدد ارسال می کند، bandwidth اضافه مصرف نمی کند.	برای هر پکتی که از دست می رود باید کل window مجدد ارسال شود. در صورتی که error-rate بالا باشد، بخش زیادی از bandwidth صرف فرستادن داده های تکراری می شود.	consumed for data packets
از آنجایی که ack ها را به صورت individual ارسال می کند، به ازای هر پکت یک ack ارسال می شود و bandwidth بیشتری مصرف می شود.	از آنجایی که ack ها را به صورت cumulative ارسال می کند، تعداد ack های کمتری ارسال می کند و در نتیجه برای ack ها bandwidth کمتری مصرف می شود.	Bandwidth consumed for ack packets

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

## ۰۲

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 می شود. (برای اینکه در انتهای عملیات همه با هم بسته شوند) همانطور که در کد بالا مشاهده می شود، یک حلقه while می زنیم و تا وقتی که last frame sent برابر با تعداد فریم ها نشده است، پیش می رویم. 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 نمی کنیم. در غیر اینصورت یک پکت ارسال کرده و lfs را یک واحد افزایش می دهیم. سپس باید عملیات دریافت ack انجام شود. اما از آنجایی که پروتکل Go-Back-N اجازه فرستادن چند پکت بدون دریافت ack را می دهد، عملیات receive نباید جلوی فرستادن پکت های بعدی را بگیرد. پس برای اجرای این عملیات یک thread جدید تشکیل می دهیم و در آن thread منتظر دریافت ack می مانیم.

```

void alarm_handler(int a) {
    cout << "time out occurred!\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 سایر

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 = acquireFrameIndex(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 نمی شوند 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 می خوانیم. 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 نمی شود و پکت هم دریافت نمی شود.

از آنجایی که ممکن است پکت ack هم توسط روتر drop شود، حتی برای پکت هایی که به دست B هم رسیده اند می توانند 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 می شود اما داده های آنها مجدد ذخیره نمی شود.

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 فایل‌ها به صورت زیر است:

برای A:

```
lar and lfs are: 681 682
sending packet with sequence number 682
packet content is:
682 4625085282706988812822930138218466374524023159847349699030998731244290316248877616127354994943514784160430249591960166158827428
3012297335233487487686932791672504848939891571342804916267615897166810277253170819233520663757428230248104405100538854660927429
9123919446207336590235297840044149569787830675448852079217813254125078565940555357151112153929694801251132171203795549941163920
7476175369012980559938118310990110840174599144325883878065361105628152837782089093135806082805172562067617245231771061838879589
4768469919871257284902195592516344833223851928691778696654759047583702848545509080899254152913598427216007302226235522358460750
5223927757227044947866784478462919350895723643402842384138354740120256079056226884847266693352002580953694527348195695223607106
470507355812237541541720092196556340553347843275197814453910296613637292816403322858277400213988846375567124674868467361795016
4745728385156852832029629080429064757351207656991841817680115495315482039644479548944488439788311653187692817074819840714497361
5386314847182104599263810424039609228765878994123179215715094367283327303020988006589792974721368333841513658700314945433634456
3685020221252259142980926069027074077129168059113331677771782227727933843623094566219277157186574916058687802285827552124321505
0794558617931775588320039074320252238014802597836590634824227249703483529574186006183975509386000848443692647358501225142361167
3942679708028160995866607265771564423203677879968679929985370044267348240320617033379072586756373490054605230219409903866795469
ack received for packet 682
```

```
lar and lfs are: 682 683
sending packet with sequence number 683
packet content is:
683 7536599276764983130315949383875187945686502994166320326189211280377933687730984141460802381512939077916932152569652666724046409
0292377825630966235158683241512971091944539929600065707699283877623772501033680549048220288927590692487589517712276010516389098
414177762762311777741074118289732264693577443233759348970290971583479384990205235889491696359494182882711821042742475522982593
0263099400776234259640984881940007947968250580544744304077736076557953616095573458814306253535718606350692981623255928091096507
0588327094996262431505059042095661828596107333578122924612340170790916991622052806671333685104366780235136494027277232872825498
1523246308282008538061408160202517775966824053244715248653671495274650673682535042749033828795099010214859776685551489176975506
5811209762314534687747281829159103097186141033482266881873367741901377849867980657673782571505747169485443159028925476795003998
9634367858634565656906133482371306765239555613722480562207025106068336976469898723556668691361827599343403798798405762012284113
ack received for packet 683
```

برای B:

```
received packet sequence number is 682
Packet content is:
682 4625085282706988812822930138218466374524023159847349699030998731244290316248877616127354994943514784160430249591960166158827428
3012297335233487487686932791672504848939891571342804916267615897166810277253170819233520663757428230248104405100538854660927429
9123919446207336590235297840044149569787830675448852079217813254125078565940555357151112153929694801251132171203795549941163920
7476175369012980559938118310990110840174599144325883878065361105628152837782089093135806082805172562067617245231771061838879589
4768469919871257284902195592516344833223851928691778696654759047583702848545509080899254152913598427216007302226235522358460750
5223927757227044947866784478462919350895723643402842384138354740120256079056226884847266693352002580953694527348195695223607106
470507355812237541541720092196556340553347843275197814453910296613637292816403322858277400213988846375567124674868467361795016
4745728385156852832029629080429064757351207656991841817680115495315482039644479548944488439788311653187692817074819840714497361
5386314847182104599263810424039609228765878994123179215715094367283327303020988006589792974721368333841513658700314945433634456
3685020221252259142980926069027074077129168059113331677771782227727933843623094566219277157186574916058687802285827552124321505
0794558617931775588320039074320252238014802597836590634824227249703483529574186006183975509386000848443692647358501225142361167
39426797080281609958666072657715644232036778799686799299853700442673482403206170333790725867563734900546052302194099038667954
ack sent for packet 682
```

```
received packet sequence number is 683
Packet content is:
683 7536599276764983130315949383875187945686502994166320326189211280377933687730984141460802381512939077916932152569652666724046409
0292377825630966235158683241512971091944539929600065707699283877623772501033680549048220288927590692487589517712276010516389098
414177762762311777741074118289732264693577443233759348970290971583479384990205235889491696359494182882711821042742475522982593
0263099400776234259640984881940007947968250580544744304077736076557953616095573458814306253535718606350692981623255928091096507
0588327094996262431505059042095661828596107333578122924612340170790916991622052806671333685104366780235136494027277232872825498
1523246308282008538061408160202517775966824053244715248653671495274650673682535042749033828795099010214859776685551489176975506
5811209762314534687747281829159103097186141033482266881873367741901377849867980657673782571505747169485443159028925476795003998
9634367858634565656906133482371306765239555613722480562207025106068336976469898723556668691361827599343403798798405762012284113
ack sent for packet 683
```

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 10\*5 میلی ثانیه برای رفت پکت داده و 10\*5 میلی ثانیه برای برگشت ack زمان بگذرد. اما از آنجایی که همیشه دقیقاً 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 سائز 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، sequence 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 پکت‌های 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 پکت‌های 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، thread‌های دریافت هم بسته شده و کار روتر خاتمه می‌یابد.

تابع 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 = acquireFrameIndex(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 addPacket 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 این بخش ack مربوط به پکت ۶۶۸ دریافت نشده و پیغام  
 timeout occurred 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.

```
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 occurred!

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
```

بخشی از لاگ روتر:

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.

```
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
```

بخشی از لاگ B:

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 پکت‌ها 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
857754012163119241304220133700792987322
573814573401390184702797376597678768277
756220783635680631792167782353590289564
668514440863039603337438452183102334839
757660982878290134934027252915764411718
738799813180858752973180178159891104678
624857646488118956984269332956918059997
053457088824412972551928461446699459410
785272195833524817076182913332624679770
508837036054520043373509309068879166539
643518712936861444993758053829693353179
ack resent for packet 668

received packet sequence number is 669
Packet content is:
669 58642611750418531657892097086207148
193683232098619057021641337774778340961
664301499401602876081979682970007095620
927714977211026676551654219900938997997
512258298536318513410347249946753415497
570203917047282396991260164329689930777
067101035894549299110572928351765201531
850495215197325676554311377519094645231
831878514735930924407388339196836838503
544796579408402288133622472115043047462
749857039207003117655626372483810440740
423525177016061300083013098306310098448
ack sent for packet 669
```

```
received packet sequence number is 668
Packet content is:
668 05129860514260354867904888339069915
857754012163119241304220133700792987322
573814573401390184702797376597678768277
756220783635680631792167782353590289564
668514440863039603337438452183102334839
757660982878290134934027252915764411718
738799813180858752973180178159891104678
624857646488118956984269332956918059997
053457088824412972551928461446699459410
785272195833524817076182913332624679770
508837036054520043373509309068879166539
643518712936861444993758053829693353179
ack sent for packet 668

received packet sequence number is 670
Packet content is:
670 640724827272220158295488987384079672
495394133039626881509940997723761502752
442663012455314268727320533064428151172
267962241550374458577656054932602443945
654945207829519595285197210189286875246
309571401094605836711540511167558824166
980546529862173906201115217789659458071
563585308346544375239938580961353262800
139708549434243816006695021416172621631
871893895783157994939826108374494866947
920498903191687128322751794283074400383
4510621681082500438705444746385938892179
```

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 لاگ‌های A و B و Router به ترتیب به فایل‌های

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

مراجعه کنید.

## قسمت دوم

۰.۱

ساده ترین روش برای برخورد با congestion مکانیزم tail-drop است. در این مکانیزم روتر تا زمانی که بافر پر شود اقدامی انجام نمی دهد و بعد از پر شدن بافر پکت ها را drop می کند 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 می کند و همچنین اینکه همه ی اتصالات TCP وقتی متوجه می شوند که چند تا از پکت آنها drop شده با هم hold back می کنند که باعث خالی شدن بافر می شود و باز با خالی شدن بافر، همه ی فرستنده ها پکت های خود را می فرستند و یک سیکل معیوب به وجود می آید. برای رفع این مسائل، از مکانیزم کنترل ازدحام RED استفاده می شود. این مکانیزم به جای اینکه فقط وقتی بافر پر شده، پکت ها را دراپ کند، ابتدا میانگین سائز صف را پیدا می کند. اگر بافر به نسبت این سائز خالی تر باشد، پکت ها را قبول می کند و اگر بافر پرتر باشد با احتمال مشخصی پکت ها را drop می کند. هر چه بافر پر تر شود این احتمال بیشتر شده و اگر بافر پر باشد احتمال drop یک می شود.

۰.۲

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 سائز 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 می کند. هنگامی که سائز از ۱۶ بیشتر می شود، احتمال drop از پنجاه درصد بیشتر می شود و در پایان زمانی که سائز صف بیست و در واقع بافر پر است، احتمال drop به ۱ می رسد. در هر بار دریافت یک پکت جدید با استفاده از تابع updateDropProb، احتمال drop بر اساس سائز فعلی صف محاسبه شده و اعمال می گردد:

```
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 می کند که شبکه در حال حاضر شلوغ است و سائز پنجره خود را نصف می کند. همچنین در صورت دریافت یک ack می فهمد که شبکه خلوتر شده است و سائز پنجره خود را یک واحد افزایش می دهد. به این ترتیب توانستیم از وقوع congestion در شبکه جلوگیری کنیم

و میانگین سائز 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 Ifr از ۲۰ Ifr استفاده کردیم. هر کدام از این Ifr ها مربوط به یک کامپیوتر می شوند. همچنین هر پکت علاوه بر sequence number، مبدا خود را هم ذخیره می کند که B با استفاده از این اطلاعات متوجه می شود Ifr 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 می کند و به هر کدام از اجراها شماره خاص خود را نسبت می دهد. 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 می شوند. 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 occurred!

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 می دهد. این نتایج در فایل sample\_remade.txt ذخیره شده است. برای هر پکت sequence

number، مبدأ و سائز 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 می شوند زمان انتقال کل برابر با

بیشترین زمان انتقال می شود 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.)