LATVIJAS UNIVERSITĀTE

MAĢISTRA DARBS

RĪGA 2011

LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

DŽITER-BUFERA VADĪBAS ALGORITMA OPTIMIZĀCIJA LAIKKRITISKAI DATPLŪSMAI

MAĢISTRA DARBS

Autors: **Kirils Solovjovs**
Stud. apl. ks05020
Darba vadītājs: Dr. sc. comp.
Mihails Broitmans

RĪGA 2011

ANOTĀCIJA

No visiem laikkritiskajiem lietojumiem visprasīgākā pret datortīkla kvalitāti ir IP balss pārraide. Diemžēl IP tīkls nenodrošina garantētu kanāla ietilpību, latentumu vai paketes nogādāšanu galapunktā. Lai mīkstinātu šo problēmu ietekmi, tiek lietots džiter-buferis.

Maģistra darba mērķis ir izstrādāt efektīvāku džiter-bufera vadības algoritmu un novērtēt dažādu algoritmu darbību, izmantojot tīkla simulācijas modeli. Autors piedāvā trīs jaunus džiter-bufera vadības algoritmus. To būtiskākā raksturiezīme ir spēja sekot līdzi tam, kurā brīdi katra pakete ir nepieciešama kodekam un, balstoties uz to, attiecīgi izvietot ienākošās paketes džiter-buferī. Salīdzinot ar esošajiem fiksētā džiter-bufera vadības algoritmiem, autoram ir izdevies panākt būtisku uzlabojumu saglabāto pakešu skaitā un balss pārraides kvalitātē.

Atslēgvārdi: džiter-buferis, VoIP, balss pārraide, algoritms, G.729, VAD, MOS.

ABSTRACT

Of all the time-critical applications VoIP is the most sensitive to network problems. Unfortunately IP network does not guarantee any channel capacity, any specific latency or even delivery. To mitigate these issues a jitter-buffer is used.

The goal of this master's thesis is to develop a more efficient jitter-buffer management algorithm and to compare such different algorithms in simulation. The author suggests three new jitter-buffer management algorithms whose distinctive feature is constantly monitoring which packet is required by the codec at any given moment in time, and making decisions about incoming packet placement based on that information. These new algorithms offer a substantial improvement in both the count of retained valid packets and VoIP quality when compared to the current fixed jitter-buffer management algorithms.

Keywords: jitter-buffer, packet delay variation, VoIP, algorithm, G.729, VAD, MOS.

AUTOREFERĀTS

Maģistra darbā autors ir **izstrādājis jaunus džiter-bufera vadības algoritmus**. Darba ietvaros ir paveikts sekojošais:

- Darbu uzsākot, autors ir **izanalizējis pieejamo literatūru** par IP balss pārraidi, džiter-bufera vadības algoritmiem, tīkla modelēšanu u.c.
- Lai spētu izstrādāt simulācijas modeli, autors **apguvis *OMNeT*++ simulācijas vides** un bibliotēku lietošanu. Autors arī **guvis ieskatu GPSS modelēšanas valodā**, lai varētu salīdzināt citu autoru modelēto algoritmu darbību.
- Pēc tam autors **izstrādājis trīs jaunus džiter-bufera vadības algoritmus**.
- Autors savstarpēji **novērtējis dažādos džiter-bufera algoritmus** un salīdzinājis tos ar iepriekšējiem, analizējot no simulācijas modeļa ievāktos datus.
- Darba noslēgumā autors ir **secinājis, ka izvirzītā alternatīvā hipotēze** – vismaz viens no autora izstrādātajiem algoritmiem pie kādas no tīkla noslodzēm pēc darbības kvalitātes būtiski pārspēs labāko no jau esošajiem darbā apskatītajiem algoritmiem – **ir pareiza**, kā arī noskaidrojis, kāda ir VAD (*Voice Activity Detection*) ietekme uz džiter-buferi.

SATURS

APZĪMĒJUMU SARAKSTS

ITU-T – komiteja Apvienoto Nāciju Organizācijas aģentūrā telekomunikāciju jomā, kas nodarbojas ar tehnisko jautājumu izpēti un rekomendāciju izstrādi

pakešu aizkaves variācija, džiteris (packet delay variation, PDV, jitter) – pakešu pienākšanas laika nobīdes svārstības

džiter-buferis – atmiņas apgabals, kas nodrošina vienmērīgu datu plūsmu uz kodeku

laikkritiska(1) datplūsma (time-critical traffic) – tāda digitālas informācijas plūsma, kuras lietojumu pastiprināti ietekmē tīkla problēmas

IP balss pārraide (VoIP) – balss pārraide, izmantojot IP protokolu

maršrutētājs (router) – tīkla iekārta, kuras galvenais uzdevums ir nodrošināt datu pārraides maršruta izvēli

ciparsignālu procesors(2) (DSP, Digital Signal Processor) – mikroshēma, kas saspiež balss signālu, ģenerē toņus un dekodē saspiesto signālu

kodeks (codec) – algoritms, kas pārveido analogo signālu digitālā signālā vai otrādi

kodētājs (coder) – algoritms, kas katram ieejas signālam piekārto noteiktu izejas signālu

VAD (Voice Activity Detection) – kodeka spēja aprēķināt, vai cilvēks konkrētā brīdī runā vai nerunā

DTX (Discountinous Transmission) – kodeka spēja noteiktos apstākļos apturēt pakešu izsūtīšanu ar mērķi ietaupīt patērēto joslas platumu

MOS(2) (Mean Opinion Score) – plaši izmantots kritērijs, lai noteiktu subjektīvo skaņas kvalitāti

kbps(3) – kilobiti sekundē – 1000 biti (125 baiti) sekundē

cirkulārs buferis – tāda datu struktūra ar izmēru N (adreses no 0..N-1), kurai, piekļūstot caur jebkādu adresi M ārpus definētajām, notiek piekļuve caur adresi M mod N

GPSS (General Purpose Simulation System) – diskrēta simulācijas valoda

Tcl/Tk – atvērtā koda bibliotēka grafiskās lietotāja saskarnes nodrošināšanai ar vairāku platformu atbalstu

RTMP (Real Time Messaging Protocol) – protokols audio, video un citas informācijas straumēšanai

RTMPE (Real Time Messaging Protocol Encryption) – protokola RTMP versija, kas nodrošina audio un video straumēšanu šifrētā veidā

Programmas pirmkoda fragments, teksta faila vai skripta saturs darbā tiks apzīmēts šādi. Fails *dati.txt:*

```
Hello, world!
```

IEVADS

No visiem laikkritiskajiem lietojumiem visprasīgākā pret datortīkla kvalitāti ir tieši IP balss pārraide (*VoIP*). Tā mūsdienās ir ikdienas sastāvdaļa. To lietojam gan apzināti, piemēram, lietojot kādu IP balss pārraides programmatūru vai IP telefonu, gan arī neapzināti, piemēram, piezvanot kādam uzņēmumam vai draugam, kura telefona pieslēgums ir veikts, izmantojot IP balss pārraidi – šajā gadījumā balss dati savienojuma otrajā daļā tiks pārsūtīti, izmantojot IP pārraidi. Ar vien vairāk uzņēmumi pāriet no publiskā komutējamā telefonu tīkla (*PSTN*) uz IP balss pārraidi zemāku izmaksu dēļ, taču IP tīkls nenodrošina garantētu kanāla ietilpību, latentumu un paketes nogādāšanu galapunktā. Šo un citu faktoru dēļ paketes ierodas galiekārtā ar nejaušu aizkavi, turklāt tās var ierasties sajauktā secībā un pat pavisam pazust ceļā.

Lai mīkstinātu šo problēmu ietekmi, iekārtās, t.sk. maršrutētājos, kas nodrošina IP balss pārraidi, tiek lietots džiter-buferis, kas parasti novietots pirms ciparsignālu procesora.(4) Šī bufera mērķis ir nodrošināt, ka saņemtās datu paketes, kas dažādu iemeslu dēļ var pienākt ar patvaļīgu kavēšanos vai nepienākt vispār, tiek nosūtītas balss apstrādes kodekam vienmērīgi. Džiter-bufera uzvedību (aizpildīšanu) nosaka džiter-bufera vadības algoritms. Pirms kāda laika *Cisco* plaši pielietoja ļoti vienkāršu džiter-bufera vadības algoritmu(5, 6), kura uzvedība pie noslogota tīkla nav optimāla. Šis pats algoritms joprojām tiek pielietots(7) arī *Asterisk* 1.8.4, kas darba rakstīšanas brīdī ir jaunākā versija.(8)

Šim algoritmam ir izstrādāta uzlabota versija, kas fiksē pēdējās no bufera uz kodeku izsūtītās paketes numuru.(9)

Maģistra darba mērķis ir izstrādāt efektīvāku džiter-bufera vadības algoritmu, kas ļaus labāk kompensēt tīkla radītus traucējumus, kā arī novērtēt dažādus algoritmus, izmantojot autora izstrādātu tīkla simulācijas modeli.

Šajā darbā autors izmanto sekojošas **metodes mērķa sasniegšanai**:

- tīkla simulācija modeļa izstrāde;
- esošo algoritmu izpēte;
- jaunu algoritmu izstrāde;
- algoritmu rezultatīvo parametru salīdzinājums, izmantojot tīkla simulācijas modeli.

Šī darba ietvaros tiek formulēta **nulles hipotēze** – visi autora izstrādātie algoritmi pie jebkādas tīkla noslodzes pēc to darbības kvalitātes būtiski neatšķiras no jau izstrādātiem algoritmiem(6, 7, 9).

Balstoties uz nulles hipotēzi, formulēta **alternatīvā hipotēze** – vismaz viens no autora izstrādātajiem algoritmiem pie kādas no tīkla noslodzēm pēc darbības kvalitātes būtiski pārspēs labāko no jau esošajiem darbā apskatītajiem algoritmiem.

Jāņem vērā, ka maģistra darbs tiek izstrādāts, lietojot Ubuntu Linux distributīva dažādas versijas, līdz ar to darbā apskatītās instalācijas procedūras un skripti ir pielāgoti šai operētājsistēmai.

Darbs ir sadalīts nodaļās. Darba sākumā tiek apskatīta teorētiskā bāze un veikta literatūras izpēte, tad seko matemātikā modeļa izstrādes process un modeļa apraksts. Pēc tam aprakstīti gan esošie, gan autora izveidotie džiter-bufera vadības algoritmi. Tam sekojošajā nodaļā tiek novērtēta un salīdzināta algoritmu darbība. Darbs tiek noslēgts ar secinājumiem.

1. TEORĒTISKĀ BĀZE UN LITERATŪRAS APSKATS

Šajā nodaļā ir aprakstīta darba teorētiskā bāze, kas balstīta uz publikācijām žurnālos, standartiem, tehnisko literatūru, kā arī citu autoru pētījumiem.

Laikkritiska datplūsma tīklā tiek pārsūtīta dažādos gadījumos. Tas var notikt gan tad, kad tiek tiešsaistē (taču ne reālajā laikā) tiek skatīts kāds video, piemēram *South Park*(10), kas izmanto RTMPE protokolu, gan arī tad, kad notiek sazināšanās ar IP balss pārraides palīdzību, piemēram, lietojot kādu no IP balss pārraides programmatūrām(11) vai IP telefonu, kas izmanto ITU-T ieteiktos standartus. Arī skatoties reālā laika video tādās vietnēs kā *ustream.tv*(12), kur tiek lietots RTMP protokols, tiks pārsūtīta laikkritiska datplūsma.

No šiem lietojumiem uz datplūsmas pārsūtīšanas traucējumiem visjūtīgāk reaģē IP balss pārraide, tāpēc turpmākajā darbā ir analizēta tieši IP balss pārraide.

1.1. Literatūras apskata veikšanas metodoloģija

Lai veiktu tīkla modelēšanu un atrastu tīkla modeļa parametrus, kā arī noskaidrotu jau šobrīd ieviestos algoritmus un noteiktu kvalitātes izvērtēšanas metodoloģiju, jāveic literatūras apskats.

Ņemot vērā maģistra darba mērķi – izstrādāt džiter-bufera vadības algoritmu un salīdzināt dažādus algoritmus, izmantojot tīkla simulācijas modeli – tiek noteikti vairāki nodalīti izpētes jautājumi, kas katrs nosaka atsevišķu literatūras meklēšanas kategoriju ar saviem atslēgas vārdiem (atslēgas vārdi norādīti tādā valodā, kā tie meklēti):

1. Kurš ir populārs IP balss pārraides kodeks un kā tas funkcionē?
   - VoIP, codec, application, popular;
2. Kāda ir tīkla pakešu aizkaves variācijas (džitera) ietekme un rašanās iemesli?
   - jitter, packet delay variation, source, reason, latency;
3. Kādi fiksēti džiter-bufera vadības algoritmi jau eksistē?
   - jitter-buffer, fixed, algorithm;
4. Kā novērtēt balss pārraides kvalitāti?
   - voice, quality, impact, model;
5. Kā pareizi modelēt tīklu un pakešu aizkaves variāciju?
   - packet delay variation, network, analytical, model, simulation.

Atslēgas vārdi izmantoti, lai atrastu rakstus zinātniskajās datubāzēs (*SpringerLink, ACM Digital Library, IEEE Xplore un ScienceDirect*) un žurnālos, rīkā *Google Scholar*, kā arī ražotāju tehniskajā literatūrā (t.sk. ITU-T rekomendācijās) un Internetā.

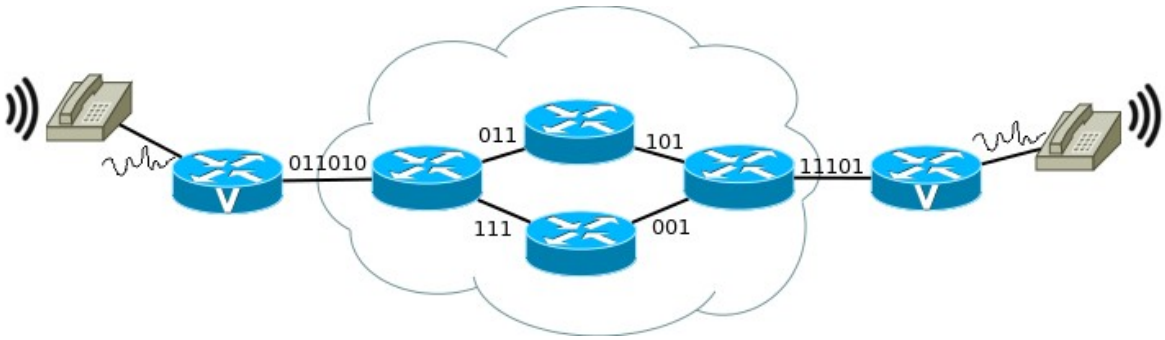
Pēc tam kategorijās, kur, spriežot pēc atrasto avotu daudzuma un satura, ir redzama nepieciešamība atrast papildus avotus, tie tiek meklēti, izmantojot atsauču metodi, t.i. sekojot šādiem soļiem:

1. tiek caurskatītas katra attiecīgajā kategorijā atrastā avota atsauces;
2. atsaucēm, kuru nosaukums liecina par atbilstošu saturu, tiek sameklēta un izlasīta anotācija;
3. ja anotācija norāda, ka raksts vai pētījums satur vajadzīgo informāciju, un šo rakstu ir iespējams iegūt lasīšanai, raksts tiek pievienots lasāmo rakstu sarakstam; ja rakstu nav iespējams iegūt lasīšanai, tad tā atslēgas vārdi tiek pievienoti meklējamiem atslēgas vārdiem cerībā atrast līdzīgus alternatīvus rakstus.

Šādā veidā tiek iegūts pilnīgs priekšstats par katru no izpētes jautājumiem. Izmantotie avoti ir norādīti darba noslēgumā, kā arī uz tiem ir atsauces viscaur darba tekstam.

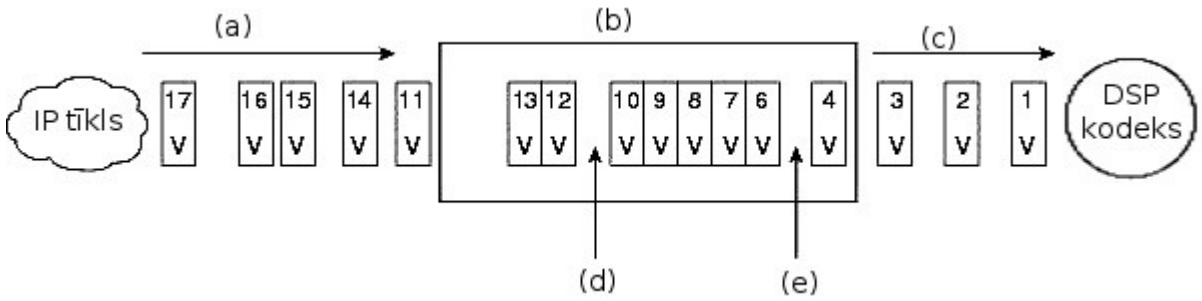
1.2. IP balss pārraide

IP balss pārraide ir balss signāla pārsūtīšana, izmantojot tīklu, kas darbojas IP protokolā, piemēram, globālais tīkls Internet. Tā atšķiras no tradicionālās balss pārraides, kas tiek veikta publiskajā komutējamā telefonu tīklā, ar to, ka IP balss pārraidē pārsūtāmais signāls tiek digitizēts un sadalīts paketēs.



1.1. att. **IP balss pārraides sistēmas shematisks attēlojums**

Lai uzskatāmāk demonstrētu digitizēšanu, 1.1. attēlā ir parādīts IP balss pārraides sistēmas kopskats, pieņemot, ka ir tiek lietoti standarta telefoni, bet adapteris (*Foreign Exchange Station*) atrodas pie balss maršrutētāja.



1.2. att. **Saņemošais balss maršrutētājs tuvplānā; pielāgots no (6)**

Saņemošā balss maršrutētāja darbības daļēja shēma redzama 1.2. attēlā. Tas sastāv no džiter-bufera, ciparsignālu procesora (*DSP*), balss kodeka u.c. Saņemošajā balss maršrutētājā paketes pienāk dažādos laika intervālos un var pat pienākt atšķirīgā secībā vai pavisam pazust ceļā(a). Džiter-buferī paketes tiek izvietotas(b) tā, lai ciparsignālu procesors un kodeks tās varētu pārveidot atpakaļ balss signālā un atskaņot. Kodekam ir nepieciešams saņemt paketes pareizā secībā un fiksētos laika intervālos(c). Attēlā ir redzams, ka buferī šobrīd trūkst divas paketes. 5. pakete(e) visdrīzāk ir pazudusi un nepienāks. Tā vietā uz kodeku nekas netiks sūtīts, bet tiks ieturēta attiecīga izmēra pauze pirms nākamās paketes. Savukārt 11. pakete(d) drīzumā ienāks buferī un tiks novietota sev paredzētajā vietā vēl tā būtu jāsūta uz kodeku.

1.2.1. Pakešu aizkaves variācija

Nosūtošā puse izsūta paketes plūsmā, katru no paketēm vienmērīgi atdalot vienu no otras. Tīkla noslodzes vai citu iemeslu dēļ atstarpe starp šīm paketēm var samazināties un/vai palielināties, tādā veidā radot džiteri.(13)

Džiteris jeb pakešu aizkaves variācija starp jebkurām divām paketēm vienā plūsmā tiek definēta kā modulis no starpības starp 1. paketes aizkavi un 2. paketes aizkavi. Aizkaves tiek mērītas starp diviem fiksētiem punktiem tīklā.(14) Citi avoti piedāvā papildus aprēķina metodoloģijas.(15)

Kā džitera un tam līdzīgu pakešu pārraides kropļojumu rašanās iemesli literatūrā tiek minēti šādi aspekti:(15, 16)

- apstrādes aizkave, kas rodas starpmezgliem (piemēram, maršrutētājiem) apstrādājot paketes;
- izsūtošās iekārtas pakešu sistēmas nespēja nodrošināt vienmērīgu pakešu izsūtīšanu, piemēram, programmatiskajos telefonos (*softphones*);
- aizkaves variācija, kas rodas tā dēļ, ka maršrutētāji var izvēlēties dažādus ceļus katrai paketei;

- lokālā tīkla pārslodze;
- Interneta pieslēguma pārslodze;
- pakešu zudumi, kas raksturīgi IP tīkla darbībai (*best-effort* piegāde);
- ugunsmūris, kas "savāc" IP plūsmu vienā pusē un no jauna izveido otrā;
- katru reizi, kad notiek maršrutētāju tabulu atjaunošana, tie uz īsu brīdi nedaudz aizkavē pārējās "parastās" plūsmas;
- nosūtošās un saņemošās iekārtas pulksteņu novirze, t.i. laika kristālu vai rezonatoru uzbūves īpatnība, kuras dēļ nav iespējas nodrošināt oscilācijas ar bezgalīgu precizitāti – tā var novest pie pulksteņu nesakritības un nobīdes ar ātrumu līdz pat 2ms 60 sekunžu laikā.

Tā kā eksistē augšminētie faktori, kuru dēļ paketes netiek piegādātas perfekti, lai nodrošinātu laikkritiskas datplūsmas patīkamu atskaņošanu lietotājam, IP balss pārraides gadījumā tiek lietots džiter-buferis. Vispārīgā gadījumā to dēvē par atskaņošanas aizkaves buferi (*playout delay buffer*).

1.3. Saņemošā maršrutētāja (galiekārtas) uzbūve

Kā redzams 1.2. attēlā, šī darba kontekstā būtiskākās galiekārtas komponentes ir ciparsignālu procesors, kas atbalsta vienu vai vairākus kodekus un džiter-buferis.

1.3.1. Kodeks

Kodeks nodrošina balss digitizēšanu un sapakošanu paketēs. Parasti kodeka uzvedību var pielāgot, mainot tā parametrus. Mūsdienās ir pieejami ļoti daudzi gan patentēti, gan brīvi pieejami kodeki. Tā kā darba mērķis ir izstrādāt džiter-bufera vadības algoritmu, **kas nav piesaistīts konkrētam kodekam** un ir pielietojams arī citos laikkritiskos lietojumos, tad kodeka izvēlei nav būtiskas nozīmes.

Neskatoties uz to, ir jāizvēlas kāds kodeks, uz ko bāzēt simulācijas modeli un tā parametrus, tāpēc kā sevi pierādījis standarts, kas tiek izmantots ar vien plašāk, tiek izvēlēts kodeks G.729 ar tā modifikācijām. Redzams, ka šī kodeku saime tiek atbalstīta ļoti daudzās iekārtās, tajā skaitā arī *Cisco* telefonos. Kodeka priekšrocības ir lieliska sarunas kvalitāte, vidēja aizkave un vidējas prasības pret apstrādes aparatūru, patērējot tikai 8kbps joslas platuma. Šo iemeslu dēļ tas tiekot izmantots gan IP balss pārraidē, gan dažādos sadarbības rīkos.(18)

1.3.1.1. G.729

Kodeks G.729 sastāv no balss kodētāja, kas izmanto fiksētā komata aritmētiku un kodē balsi ar ātrumu 8kbps. Šis kodētājs kodē audio signālu 10ms lielos kadros. Algoritms savā darbības gaitā skata balss signālu 5ms uz priekšu, līdz ar to kopējā algoritmiskā aizkave ir 15ms.(19) Tradicionāli vienā IP paketē iekļauj divus G.729 kadrus.(20) Ņemot vērā šo faktu, katras IP paketes algoritmiskā aizkave ir *5ms+10ms*2 = 25ms*.(21)

1.3.1.2. G.729a

ITU-T rekomendācijai G.729 ir arī vairāki pielikumi, kas norāda, kā ieviešami kodeka G.729 papildinājumi. Viens no pielikumiem – G.729a – apraksta kodeku, kuram ir zemākas prasības pret aparatūru, taču tas ir pilnībā saderīgs ar G.729 plūsmas līmenī, t.i. plūsmu, kas kodēta ar G.729, ir iespējams atkodēt ar G.729a un otrādi.(19)

Ir svarīgi ņemt vērā, ka, pielietojot G.729a kaut vienā galā, var kristies sakaru kvalitāte. Grāmatā minēts, ka G.729 kodeks nodrošina MOS 3.92, bet G.729a tikai 3.70.(22) Sīkāk par MOS skalu var izlasīt šī maģistra darba apakšnodaļā "1.5. Novērtēšanas metodoloģija".

G.729a kodeka algoritmiskā aizkave ir tāda pati kā G.729.(21)

1.3.1.3. G.729b

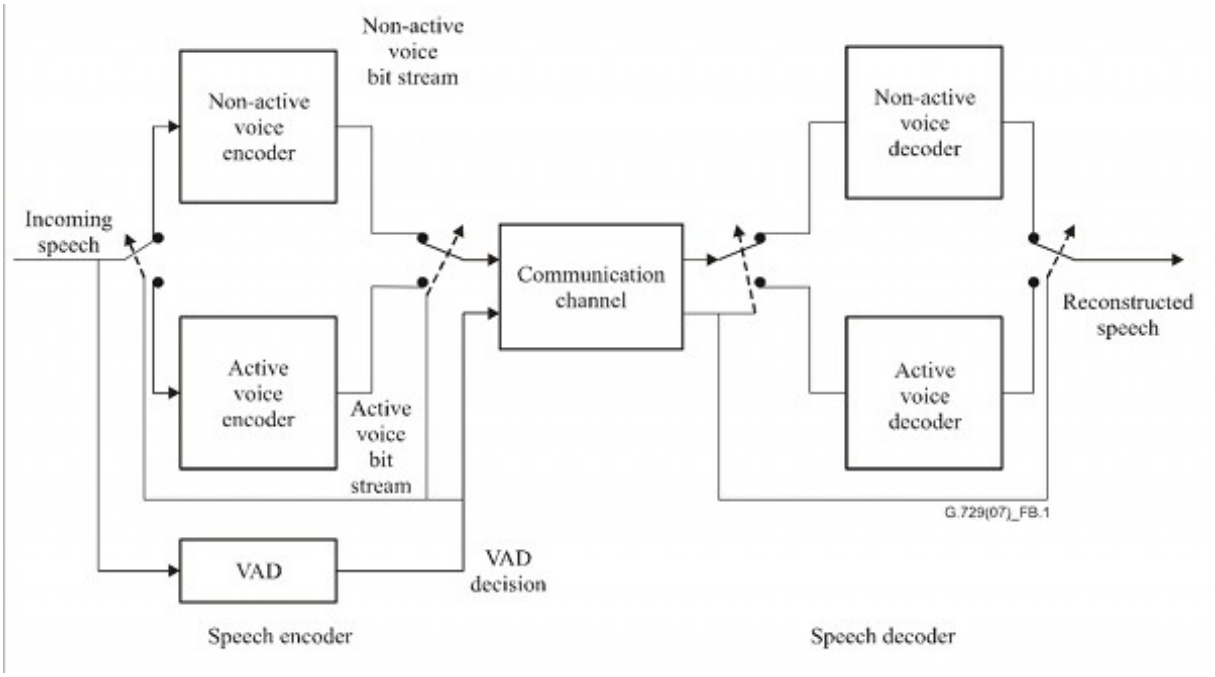
Pielikums G.729b apraksta VAD, DTX un CNG (*comfort noise generator*) algoritmus. Šo pielikumu iespējams pielietot gan G.729, gan G.729a kodekiem.(19) Ja G.729b tiek lietots kopā ar G.729a, tad šādu kodeku apzīmē G.729ab.

VAD ir kodeka spēja pamanīt, vai cilvēks konkrētā brīdī runā vai arī klusē un līdz kodētājam nonāk tikai fona troksnis. Tas nav triviāls uzdevums, jo ir situācijas, kad fona trokšņa līmenis ir pielīdzināms cilvēka balss līmenim, t.i. ir zema signāla-trokšņa attiecība (*signal-to-noise ratio*). G.729 kodekā tas tiek noteikts ik pēc 10ms.

DTX ir kodeka spēja pieņemt lēmumu nesūtīt paketes tad, ja VAD ir konstatējis, kas cilvēks šobrīd nerunā. Tas ļauj būtiski samazināt nosūtīto datu apjomu.

Savukārt CNG ir vēl viena interesanta konstrukcija, kas saistīta ar abām iepriekšējām. Cilvēks ir pieradis pie tā, ka sarunas gaitā ir noteikts fona trokšņu līmenis. Un arī brīdī, kad viņš pats runā, bet otra puse klausās, viņš sagaida, ka sarunā būs dzirdams fona troksnis, pat ja tas ir apziņai nemanāmā līmenī. Tāpēc algoritmos, tajā skaitā arī kodeka G.729b algoritmā, ir sastāvdaļa, kas tiek saukta par komforta trokšņa ģeneratoru. Tā ir atbildīga par to, lai brīdī,

kad saņēmējam nepienāk nekādas balsi saturošas paketes, tiktu ģenerēts ticams fona troksnis atbilstošā skaļumā.



1.3. att. **VAD darbības shēma(19)**

Attēlā 1.3. ir parādīta vispārīga VAD darbības shēma. Ja sistēmā ir arī DTX modulis, tad tas atrodas tieši komunikācijas kanāla sākumā. Šī shēma ir būtiska, jo tīkls tiks modelēts gan ar, gan bez VAD/DTX.

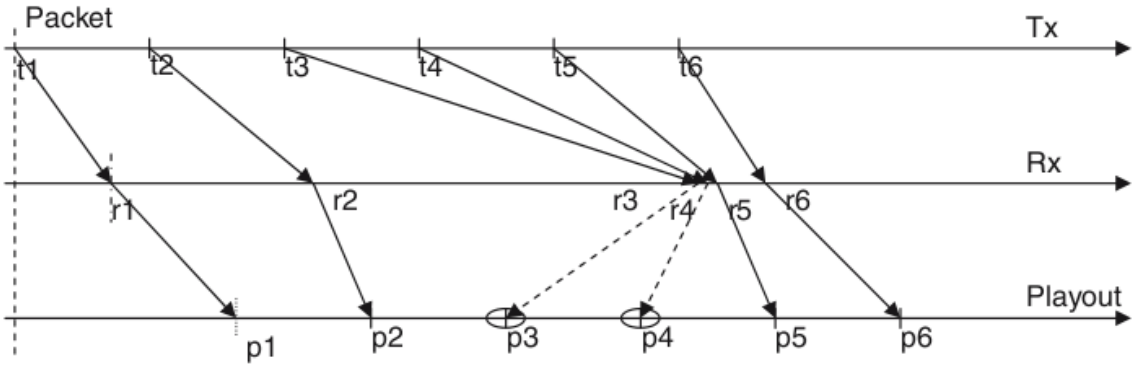
1.3.2. Džiter-buferis

Iepriekš tika apskatīti dažādi iemesli, kas var izraisīt to, ka paketes no tīkla nenonāk galiekārtā vienmērīgi. Tāpēc ir nepieciešams džiter-buferis – atmiņas apgabals tīkla pārraides iekārtās, kas nodrošina vienmērīgu datu plūsmu uz kodeku. Džiter-bufera uzvedību nosaka tā vadības algoritms.

Eksistē fiksēti un adaptīvi džiter-bufera vadības algoritmi. Fiksētajiem algoritmiem ir raksturīgs nemainīgs bufera izmērs (piemēram, 6 paketes) un salīdzinoši vienkārša uzbūve. Adaptīvajiem algoritmiem ir mainīgs bufera izmērs un/vai arī algoritms veic laika skalas modifikācijas. Tam ir nepieciešams, lai algoritmam būtu spēja sadarboties ar kodeku tādā mērā, ka tas var atļauties ignorēt prasību padot paketes kodekam ik pēc noteikta fiksēta laika intervāla. Tā vietā, lai sekotu šai prasībai, džiter-bufera vadības algoritms, izanalizē pakešu parametrus un, iespējams, arī saturu un izlemj padot paketi uz kodeku nedaudz vēlāk vai agrāk.

Šāda pieeja maģistra darba ietvaros nav pieļaujama, jo tad džiter-bufera vadības algoritms nebūtu universāls – to nevarētu vienlīdz labi pielietot gan balss pārraidei, gan citām laikkritiskām datplūsmām.

Jānorāda, ka visi adaptīvie džiter-bufera vadības algoritmi, ko autors atrada, pētot literatūru, paļaujas uz ciešu sadarbību ar kodeku un pieļauj nobīdi no fiksētā intervāla, kādā paketes jāizsūta uz kodeku.(23)



1.4. att. **Džiter-bufera laika diagramma(24)**

Laika diagrammā, kas redzama 1.4. attēlā, ir parādīta fiksēta džiter-bufera, kura izmērs ir 1 pakete, darbība. Pirmā ass ("*Tx*") norāda izsūtīšanas laiku. Redzams, ka paketes tiek izsūtītas vienmērīgā intervālā. Otrā ass ("*Rx*") norāda piegādes laiku gala iekārtā. Redzams, ka tīkla apstākļu rezultātā šis laiks svārstās, citiem vārdiem sakot, ir novērojama pakešu aizkaves variācija. Trešā ass ("*Playout*") norāda laiku, kurā kodeks saņem paketi atskaņošanai. Starp tiem ir jābūt fiksētiem intervāliem, turklāt tiem jābūt tik pat lieliem kā izsūtīšanas intervāliem.

Attēlā redzams, ka 3. un 4. pakete aizkavējas tik ļoti, ka to pienākšana buferī notiek vēlāk nekā tām būtu jānonāk kodekā, līdz ar to tās nav iespējams atskaņot. Kodeka uzdevums ir trūkstošo pakešu saturu ekstrapolēt vai veikt citas darbības saskaņā ar kodeka algoritmu, lai cilvēkam šī aizkavēšanās radītu pēc iespējas mazākas neērtības.

1.4. Esošie algoritmi

Veicot literatūras izpēti, noskaidrots, ka pirms dažiem gadiem *Cisco* plaši pielietoja triviālu džiter-bufera vadības algoritmu(5, 6), kura raksturpazīmes, samazinoties tīkla kvalitātei, strauji kritās. Šis pats algoritms (1. pielikums) joprojām tiek pielietots(7) arī *Asterisk* – populāras atvērtā pirmkoda telefonu centrāles programmatūras produkta – jaunākajā versijā, kas darba rakstīšanas brīdī ir 1.8.4.(8)

Cisco algoritmam ir izstrādāta uzlabota versija, kas fiksē pēdējās no bufera uz kodeku izsūtītās paketes numuru.(9)

Abi augstākminētie algoritmi tiks izmantoti par bāzlīniju, pret ko salīdzināt autora izstrādātos algoritmus. Šie abi algoritmi sīkāk apskatīti maģistra darba 3. nodaļā.

Literatūrā tika atrasti arī vairāki adaptīvie algoritmi. Viens no šādiem algoritmiem balstās uz faktu, ka saruna sastāv no runāšanas un klusēšanas fāzes, kā arī nobīda paketes laiku pirms sūtīšanas uz kodeku.(23)

Arī cits adaptīvs adaptīvs algoritms der tikai IP balss pārraidei, jo nobīda laiku, kad pakete tiek izsūtīta uz kodeku(24). Šī paša raksta atsaucēs ir minēti vēl 5 citi algoritmi, taču arī tie ir adaptīvi tādā mērā, ka nav pielietojami vispārīgam gadījumam.

Saskaņā ar iepriekšējās apakšnodaļas pēdējā punktā minēto, šī darba ietvaros netiks apskatīti tādi algoritmi, kas veic laika skalas modifikācijas, jo šādi džiter-bufera vadības algoritmi nav universāli pielietojami neatkarīgi no kodeka.

1.5. Novērtēšanas metodoloģija

Autora izstrādātie džiter-bufera vadības algoritmi un bāzlīnijas algoritmi ir jāsalīdzina, balstoties uz to spēju nodrošināt relatīvi kvalitatīvus sakarus, t.i. tādus, kas lietotājam ir subjektīvi patīkamāki.

Subjektīvās sakaru kvalitātes novērtēšanai ITU-T ir izstrādājis vairākas rekomendācijas. Piemēram, rekomendācija P.800 *"Subjective quality determination"* apraksta kā veikt subjektīvu novērtēšanu(25), bet rekomendācija G.107 *"The E-model, a computational model for use in transmission planning"* paredz precīzu metodoloģiju (1.1) kā noteikt pārraides kvalitātes faktoru *R*.(26)

$$R = Ro - Is - Id - Ie\text{-}eff + A \qquad (1.1)$$

Šajā formulā *Ro* apzīmē signāla-trokšņa attiecību, *Is* ir visu to traucējumu kombinācija, kas rodas reizē ar balss signālu. *Id* raksturo traucējumus, ko izraisa aizkave, bet *Ie-eff* parāda traucējumus, ko rada kodeks, kā arī iekļauj traucējumus, kas rodas pakešu zuduma rezultātā. *A* ir priekšrocības faktors, kas atspoguļo traucējumu psiholoģisku kompensēšanu, ja lietotājs uzskata, ka viņam ir pieejamas kādas papildus priekšrocības.(26)

Sīkāki skaidrojumi, kā aprēķināt formulas (1.1) saskaitāmos, lasāmi ITU-T rekomendācijā G.107. Pētījumā(27) ir dotas dažas vienkāršotas metodoloģijas, kā novērtēt pakešu zuduma ietekmi uz balss kvalitāti.

ITU-T rekomendācijas G.113 1. pielikumā ir noteikts, ka pie pakešu zuduma 0% protokolam G.729ab faktors *Ie*=11, bet faktors *Bpl*=19.0 gan protokolam G.729a, gan G.729ab.(20) Protokolam G.729a pie pakešu zuduma 0% apmērā *Ie*=10. (28)

Tā kā darba mērķis ir izstrādāt un novērtēt džiter-bufera vadības algoritmu neatkarīgi no citas negatīvas vai pozitīvas ietekmes, tad aprēķinot pārraides kvalitātes faktoru *R*, tiek pieņemts, ka *Is*=0, *A*=0.(29) Līdz ar to $R = Ro - Id - Ie\text{-}eff$. Perfektas kvalitātes kanālam *R*=93.2 (26), tātad *Ro*=*R*=93.2.

Tāpat šī pētījuma ietvaros, ņemot vērā darba mērķi, tiek pieņemts, ka *Id*=*Idd*, bet *BurstR*=1. (Proporcija *BurstR* ir nepieciešama, rēķinot *Ie-eff*.)

Balstoties uz augšminētajiem pieņēmumiem un literatūrā pieejamajām konstantēm, tiek izvesta formula (1.2), kas ļauj noskaidrot **R no pakešu zuduma *Ppl* (%) un kopējā aizkaves laika *Ta* (ms)**. Ja sarunā tiek lietots VAD/DTX, tad *Ie*=11. Ja netiek, tad *Ie*=10.

$$R = 93.2 - \frac{Ie + (95 - Ie)\,Ppl}{Ppl + 19} - Idd\,,$$

$$kur\ Idd = \left\{ \begin{array}{c} 0,\ ja\ Ta \le 100 \\ 25\left((1 + \log_2^6(Ta/100))^{\frac{1}{6}} - 3\left(1 + \left(\frac{\log_2(Ta/100)}{3} \right)^6 \right)^{\frac{1}{6}} + 2 \right),\ ja\ Ta > 100 \end{array} \right\} \qquad (1.2)$$

Lai novērtētu sarunas kvalitāti ITU-T iesaka lietot MOS skalu. Tā ir nepārtraukta piecu punktu skala ar sekojošām atzīmēm, kas attēlots 1.1. tabulā.(25)

1.1. tabula

MOS skala(25)

Runas kvalitāte	*MOS*
Lieliska	5
Laba	4
Ciešama	3
Vāja	2
Slikta	1

Vadoties pēc rekomendācijas G.107 pielikumā esošās formulas (1.3) pārraides kvalitātes faktoru R ir iespējams pārveidot MOS skalā.(26)

$$\left\{ \begin{array}{ll} Ja\ R < 0, & tad\ MOS = 1 \\ Ja\ 0 < R < 100, & tad\ MOS = 1 + 0.035R + R*(R-60)*(100-R)*7*10^{-6} \\ Ja\ R > 100, & tad\ MOS = 4.5 \end{array} \right. \qquad (1.3)$$

Līdz ar to, vadoties pēc MOS skalas, ir iespējams salīdzināt džiter-bufera vadības algoritmus un viennozīmīgi secināt, kurš sniedz labāku sarunas kvalitāti noteiktos tīkla apstākļos.

2. TĪKLA MODELĒŠANA

Šajā nodaļā ir aprakstīts autora paveiktais darbs un izvēlētie risinājumi, lai sastādītu tīkla modeli.

Lai būtu iespējams pārliecināties par avotā(9) minētajiem rezultātiem, kā arī nolasīt šajā konferencē lietots simulācijas parametrus, autors iesākumā uzinstalēja GPSS *World* simulācijas vidi ar dokumentāciju.

Vēlāk tika instalēta OMNeT++ simulācijas vide, lai varētu izstrādāt datortīkla modeli atbilstoši iepriekšējā nodaļā apskatītajai teorijai. Pēc tam tika izstrādāts pats modelis.

2.1. GPSS *World* simulācijas vide

GPSS *World Student Version*(30) ir simulācijas vide, kurā uzdevumus iespējams uzdot, izmantojot valodu GPSS *(General Purpose Simulation System)*. Tā ir salīdzinoši jaudīga, taču novecojusi simulācijas vide. Darba gaitā autors nolēma šajā vidē savu modeli neizstrādāt, izvēloties par labu modernākai. Šī vide ierobežotā studentu versijā (līdz 200 objektiem) ir pieejama bez maksas.

Neliels piemērs GPSS valodā izskatītos šādi. Fails *Telephon.gps*:

```
; GPSS World Sample File - TELEPHON.GPS, by Gerard F. Cummings
************************************************************
*                                                          *
*                                                          *
*        Telephone System Model                            *
*                                                          *
************************************************************
*    Simple Telephone Simulation                           *
*       Time Unit is one minute                            *
Sets      STORAGE    2
Transit   TABLE      M1,.5,1,20     ;Transit times
          GENERATE   1.667,1        ;Calls arrive
Again     GATE SNF   Sets,Occupied  ;Try for a line
          ENTER      Sets           ;Connect call
          ADVANCE    3,1            ;Speak for 3+/-1 min
          LEAVE      Sets           ;Free a line
          TABULATE   Transit        ;Tabulate transit time
          TERMINATE  1              ;Remove a transaction
Occupied  ADVANCE    5,1            ;Wait 5 minutes
          TRANSFER   ,Again         ;Try again
************************************************************
```

Uzinstalēt GPSS *World Student Version* bija salīdzinoši vienkārši. Balstoties uz Internetā atrastu pamācību(31), pēc lejupielādes atlika izpildīt komandu `wine msiexec /i GPSS.msi`, un simulācijas vide bija gatava darbam. Pēc tam, izmantojot piemērus un programmai komplektā saņemto pamācību, autors guva ieskatu GPSS valodā.

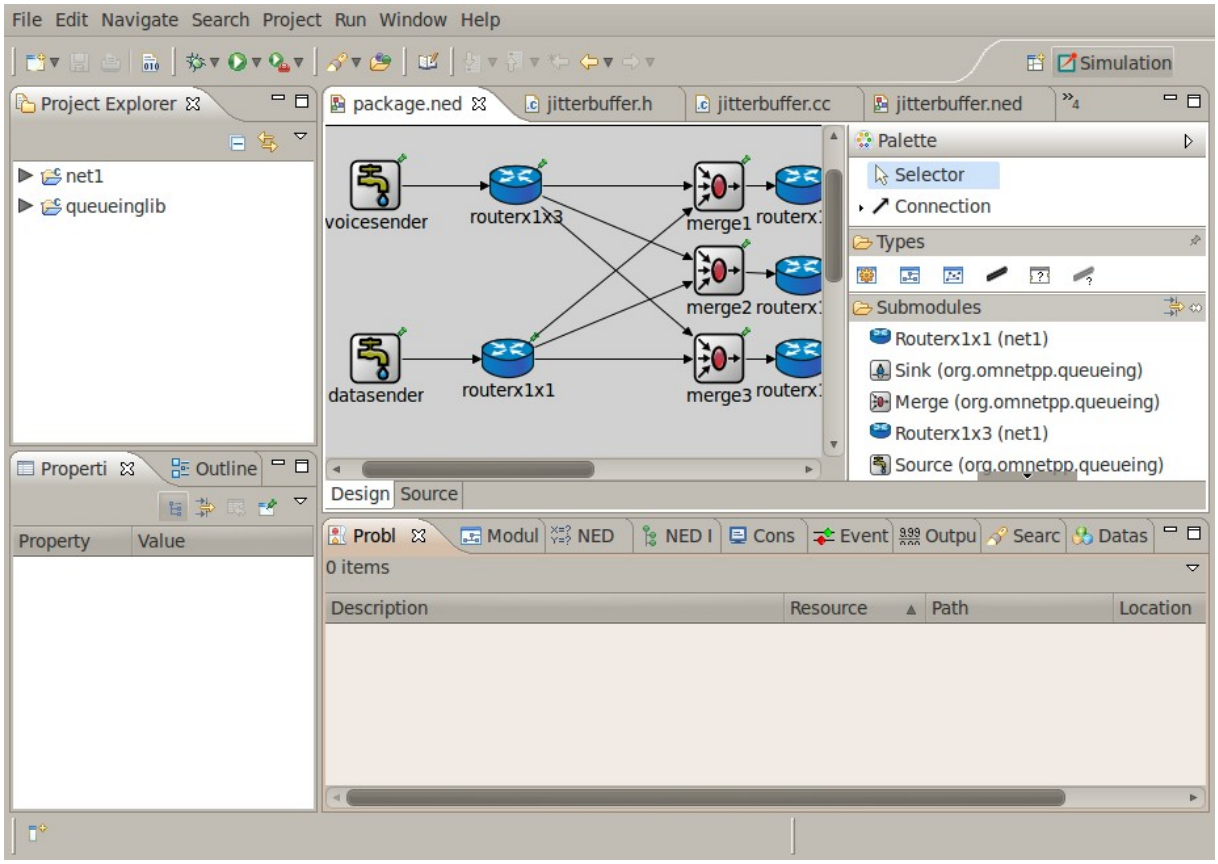
Palaižot augstākminēto piemēru *Telephon.gps* simulācijas vidē ar komandu `START 10`, kas šajā gadījumā norāda, ka simulācija notiks tik ilgi, kamēr būs pabeigtas vismaz 10 sarunas, iegūstam šādu atskaiti:

```
              GPSS World Simulation Report - Telephon.1.2


                   Saturday, May 8, 2010 11:40:27

           START TIME           END TIME  BLOCKS  FACILITIES  STORAGES
                0.000             25.572     9        0          1


              NAME                       VALUE
           AGAIN                          2.000
           OCCUPIED                       8.000
           SETS                       10000.000
           TRANSIT                    10001.000


 LABEL              LOC  BLOCK TYPE     ENTRY COUNT CURRENT COUNT RETRY
                    1    GENERATE            14             0       0
AGAIN               2    GATE                18             0       0
                    3    ENTER               11             0       0
                    4    ADVANCE             11             0       0
                    5    LEAVE               11             0       0
                    6    TABULATE            11             0       0
                    7    TERMINATE           11             0       0
OCCUPIED            8    ADVANCE              7             3       0
                    9    TRANSFER             4             0       0


STORAGE             CAP. REM. MIN. MAX.  ENTRIES AVL.  AVE.C. UTIL. RETRY DELAY
 SETS                 2    2   0     2       11   1    1.317  0.658    0    0


TABLE               MEAN    STD.DEV.       RANGE           RETRY FREQUENCY CUM.%
 TRANSIT            4.390    3.443                           0
                                    1.500  -        2.500            2    18.18
                                    2.500  -        3.500            4    54.55
                                    3.500  -        4.500            3    81.82
                                    4.500  -        5.500            0    81.82
                                    5.500  -        6.500            0    81.82
                                    6.500  -        7.500            1    90.91
                                    7.500  -        8.500            0    90.91
                                    8.500  -        9.500            0    90.91
                                    9.500  -       10.500            0    90.91
                                   10.500  -       11.500            0    90.91
                                   11.500  -       12.500            0    90.91
                                   12.500  -       13.500            0    90.91
                                   13.500  -       14.500            1   100.00


FEC XN    PRI         BDT      ASSEM  CURRENT  NEXT  PARAMETER    VALUE
    15     0          25.606     15      0       1
    12     0          26.189     12      8       9
     9     0          27.625      9      8       9
    14     0          29.238     14      8       9
```

Atskaitē citu datu starpā redzams, ka divas no 11 sarunām tika aizkavētas, jo nebija iespējams nodibināt savienojumu uzreiz. Turklāt viena no šīm sarunām tika aizkavēta trīsreiz, bet otra vienreiz.

2.2. *OMNeT++* simulācijas vide

OMNeT++ simulācijas vide(32) ir veidota modulāra un vienkārši paplašināma. Papildus modeļus iespējams rakstīt izmantojot programmēšanas valodu C++, turklāt šajā vidē darbojas visas standarta C++ bibliotēkas.



2.1. att. **OMNeT++ izstrādes vide, projektējuma skats**

Šajā vidē ir ar simulācijas failiem (“*.ned”*) iespējams strādāt divos skatos – pirmkoda skatā un projektējuma skatā. Projektējuma skats redzams 2.1. attēlā. *OMNeT++* nodrošina iespēju jaunu elementu programmēt pilnīgi “no tukšas lapas”, modificēt esošu elementu vai vienkārši pielabot objekta parametrus. Ir arī iespēja veidot moduli, savstarpēji kombinējot esošos elementus.

Vides instalācija nav tik triviāla kā GPSS *World* gadījumā, taču joprojām ir samērā vienkārša. Pēc faila lejupielādes ir jāizpilda šādas komandas; pielāgots no (33):

```
tar xzvf omnetpp-4.1-src.tgz
sudo apt-get install bison flex tcl-dev tk-dev libpcap-dev graphviz doxygen
export PATH=$PATH:~/omnetpp/bin
```

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/omnetpp/lib
export TCL_LIBRARY=/usr/share/tcltk/tcl8.4/
./configure
make
```

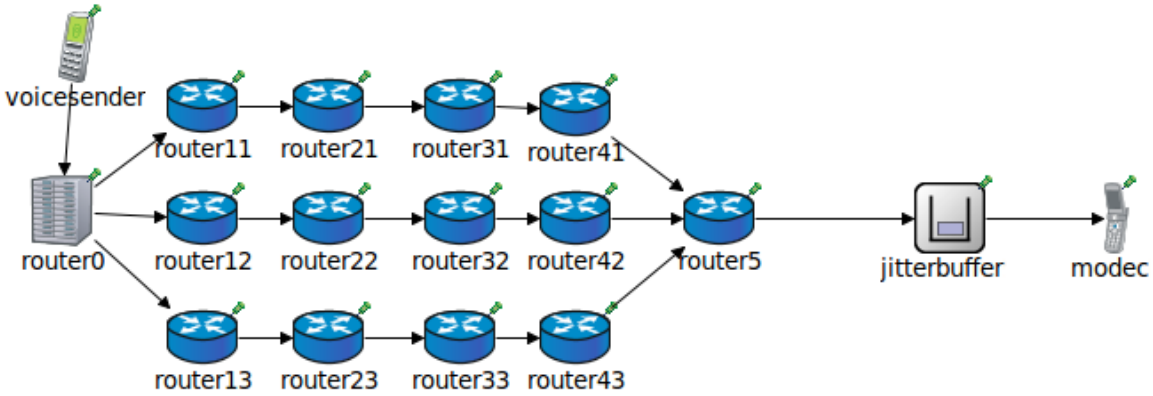
Lai nodrošinātu veiksmīgu *OMNeT++* darbību arī pēc datora pārlādes, faila *~/.bashrc* beigās jāpievieno sekojošas rindas:

```
export PATH=$PATH:~/omnetpp/bin
export TCL_LIBRARY=/usr/share/tcltk/tcl8.4/
```

Maģistra darba izstrādes gaitā pēc kārtējiem operētājsistēmas atjauninājumiem radās problēma ar *OMNeT++* fontiem – programmas ietvaros neviens uzraksts nebija salasāms. Pēc padziļinātas problēmas izpētes, autors to atrisināja ar rokām nokompilējot atšķirīgu *Tcl/Tk* versiju.

2.3. Tīkla modeļa izstrāde

Tīkla modelim par pamatu ir izvēlēts pētījumā(9) lietotais modelis, kas pieejams 2. pielikumā. Tas darīts, lai būtu iespējams precīzāk salīdzināt maģistra darba autora rezultātus ar pētījuma(9) rezultātiem. Pēc tam tīkla modelis gan uzlabots, ņemot vērā citus pētījumus par to, kā korekti jāmodelē tīkls. Modeļa būvēšanas procesā ievērotas darba teorētiskajā bāzē nospraustās prasības.



2.2. att. **Tīkla simulācijas modelis**

Tā funkcionālā uzbūve (attēls 2.2.) ir sekojoša:

1. avots *voicesender* ģenerē paketes ik pēc 20ms un izsūta tās, ja vien DTX nepieņem lēmumu tās neizsūtīt;
2. paketes nonāk maršrutētājā *router0* ar papildus 25ms nobīdi;
3. maršrutētājs *router0* izsūta paketes pa vienu no trijiem ceļiem;
4. paketes atkarībā no izvēlētā ceļa *X* nonāk maršrutētājā *router1X*, tad *router2X*, tad *router3X*, tad *router4X*; katram no tiem ir noteikts minimālais un vidējais

apstrādes laiks saskaņā ar 2.1. tabulu; viens maršrutētājs reizē var apstrādāt tikai vienu paketi; ja apstrādes laikā maršrutētājā ienāk nākamā pakete, tad tā tiek novietota neierobežota izmēra rindā;

5. pēc tam maršrutētājā *router5* visas trīs plūsmas tiek apvienotas vienā;
6. paketes tiek padotas uz saņēmēju *modec* caur džiter-buferi *jitterbuffer*;
7. saņēmēja elements rūpīgi seko līdzi pulkstenim un reģistrē, vai katra pakete viņam ir pienākusi laikā vai nē.

Uz katra no trijiem ceļiem atrodas 4 maršrutētāji, no kuriem divi simulē malējos (*edge*) maršrutētājus un divi – pamattīkla maršrutētājus. Šo maršrutētāju dažādā un varbūtiski sadalītā aizkave rada pakešu aizkaves variāciju. Literatūras avotos tiek minēti dažādi veidi, kā šo aizkavi labāk simulēt. Literatūras avotā(34) piedāvāts samērā izsmalcināts un interesants modelis, taču jaunākos darbos ir norādīts, ka aizkavei ir pilnībā jāatbilst eksponenciālai sadalījuma funkcijai(35, 36). Malējo maršrutētāju aizkave ir modelēta lielāka, jo tie parasti ir mazākas jaudas nekā pamattīkla maršrutētāji.

Lai empīriski noteiktu vidējos ceļa ātrumus, autors lietoja rīku *ping*, kas uzrāda aizkavi, kas rodas signālam ejot no punkta A uz punktu B un atpakaļ (*round trip time*). Līdz ar to, lai iegūtu savienojuma latentumu, šis skaitlis jādala ar divi. Ieskatoties globālajā Interneta savienojumu un to ātrumu kartē(37), radās iespēja noskaidrot, par kādu minimālo laiku šis latentums pieaugs, ja sūtīšana notiks starp kontinentiem.

Var secināt, ka modelī(9) piedāvātie aizkaves laiki (2. pielikums) aptuveni atbilst reālā tīkla sastopamiem laikiem, tāpēc šī darba tīkla modelī tiks izmantoti tie paši laiki, kas atrodami modelī(9). Aizkaves laiki, kas nepieciešami, lai katrs maršrutētājs apstrādātu vienu paketi, norādīti 2.1. tabulā. 1. un 4. ir malējie maršrutētāji.

2.1. tabula

Laiks, ko pakete pavada katrā no ceļiem

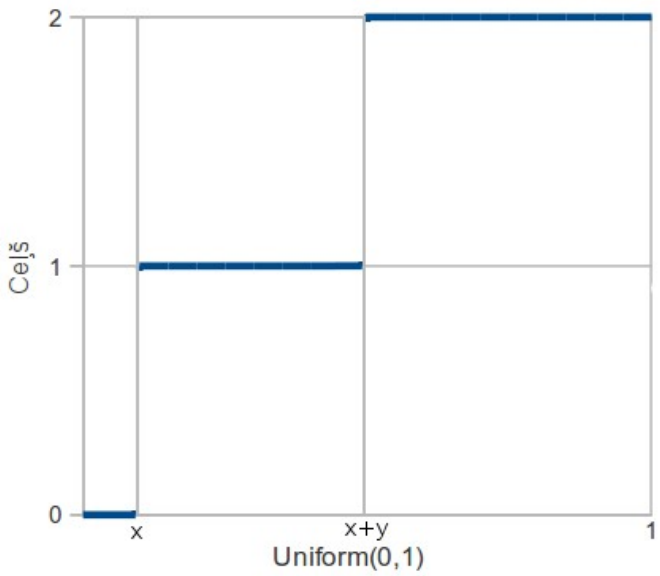
Ceļš	*1. maršrutētājs vai 4. maršrutētājs (ms)*	*2. maršrutētājs vai 3. maršrutētājs (ms)*	*Kopējais laiks ceļā(ms)*	*Vidējais laiks ceļā (ms)*
#1	*Exponential*(1)+2	*Exponential*(1)+1	4**Exponential*(1) + 6	10
#2	*Exponential*(12)+2	*Exponential*(6)+2	2**Exponential*(12) +2**Exponential*(6)+8	44
#3	*Exponential*(40)+8	*Exponential*(20)+4	2**Exponential*(40) +2**Exponential*(20)+24	144

Tiek pieņemts, ka ceļu varbūtiskais sadalījums ir brīvais parametrs. Tādā veidā ir iespējams pārbaudīt algoritmu darbību pie dažādā ceļa sadalījuma funkcijām. Ceļa sadalījums

tiek uzdots ar vienmērīgu varbūtisku sadalījumu, norādot, kāda daļa pakešu jāsūta caur 1. ceļu un kāda – caur otro. Atlikušā daļa pakešu (ja tā ir lielāka par 0) tiks nosūtīta caur 3. ceļu. Lai šādu funkcionalitāti realizētu, autors sastādīja formulu (2.1), kurai var padot svarus (pakešu procentuālo sadalījumu pa ceļiem), un kas ar vienmērīgu svērtu varbūtību atgriezīs attiecīgā ceļa numuru. Ir jāpievērš uzmanība, ka simulācijas vidē ceļu numerācija sākas no 0, līdz ar to iespējamie ceļi ir 0., 1. un 2.

$$Ceļš = min(2, \lfloor max(0, \frac{Uniform(0,1) - x}{y} + 1) \rfloor) \tag{2.1}$$

Attēlā 2.3. ir redzams formulas pārbaudes rezultāts, kas parāda, ka šī formula izpilda nepieciešamo prasību. Sadalījuma funkcija *Uniform(0,1)* atgriež skaitļus robežās [0;1) vienmērīgā sadalījumā. Ja sadalījuma funkcija atgriež skaitli [0;x), tad Ceļš = 0. Ja funkcija atgriež skaitli [x;x+y), tad Ceļš = 1. Bet, ja funkcija atgriež skaitli [x+y;1), tad Ceļš = 2.



2.3. att. **Ceļa atkarība no vienmērīga sadalījuma funkcijas *Uniform(0,1)***

Lai varētu kontrolēt, pa kuru ceļu dodas pirmā pakete, kas sasniegs buferi, pirmās 10 paketes tiek nosūtītas uz norādīto ceļu, ignorējot sadalījuma funkciju.

Apskatam varbūtiski vidējo sliktāko gadījumu. Pirmajai paketei, kas sasniegs džiter-buferi, ir jāiet pa 3. ceļu, kura vidējais ātrums saskaņā ar 2.1. tabulu ir 144ms. Iespējams, ka 11. pakete ies pa 1. ceļu un nonāks džiter-buferī vidēji 10ms laikā kopš nokļūšanas maršrutētāju tīklā. Paketes tiek izsūtītas no avota ik pēc 20ms. Tātad 11 paketes tiks izsūtītas ne ātrāk kā 200ms laikā. Tas ar lielu varbūtību ļaus 1. izsūtītajai paketei jau sasniegt džiter-buferi vidēji 144ms laikā, kamēr 11. pakete to var sasniegt vidēji 210ms laikā.

Tīkla modeļa izstrādē tika bagātīgi izmantota *OMNeT++* lietotāja pamācība(38) un komplektā iekļautā *queueinglib* bibliotēka. Tīkla modeļa pirmkods ir apskatāms 3. pielikumā.

2.3.1. Avota elementa izstrāde

Avota elementa pamatā ir paņemts elements *Source.* Tas ir papildināts ar radīto pakešu skaita statistikas uzskaiti, VAD/DTX algoritmu, kā arī, lai atvieglotu pakešu numuru noteikšanu, tam modificēts pakešu nosaukumu ģenerēšanas algoritms. Ja iepriekš pakešu nosaukumi bija "voice-1", "voice-2" utt., tad tagad tie ir vienkārši "1", "2" utt.

Literatūrā rakstīts, ka, lai gan kādreiz tika uzskatīts par pieņemamu modelēt VAD ciklus ar eksponenciālo sadalījumu, tagad ir atrasti precīzāki sadalījumi. Izrādās, ka balss aktivitātes laiks ir vistuvākais gamma sadalījumam, bet klusuma laiks ir vislabāk modelējams ar Veibula sadalījumu.(39)

Balss aktivitātes laika vidējai vērtībai ir jābūt 0.352ms, bet klusuma laika vidējai vērtībā 0.650ms.(40, 41)

Gamma sadalījuma vidējā vērtība ir $\theta*k$. Tātad balss aktivitātes laikā $\theta*k$=0.352ms. Tiek pieņemts, ka θ=1, tātad k=0.352ms.

Veibula sadalījuma vidējā vērtība ir $\lambda*\Gamma(1+1/k)$, tiek pieņemts, ka k=1, tātad $\lambda*\Gamma(2)$=0.650ms, bet no tā – λ=0.650ms.

Daļa no faila *Source.cc*, kas parāda VAD/DTX implementāciju*:*

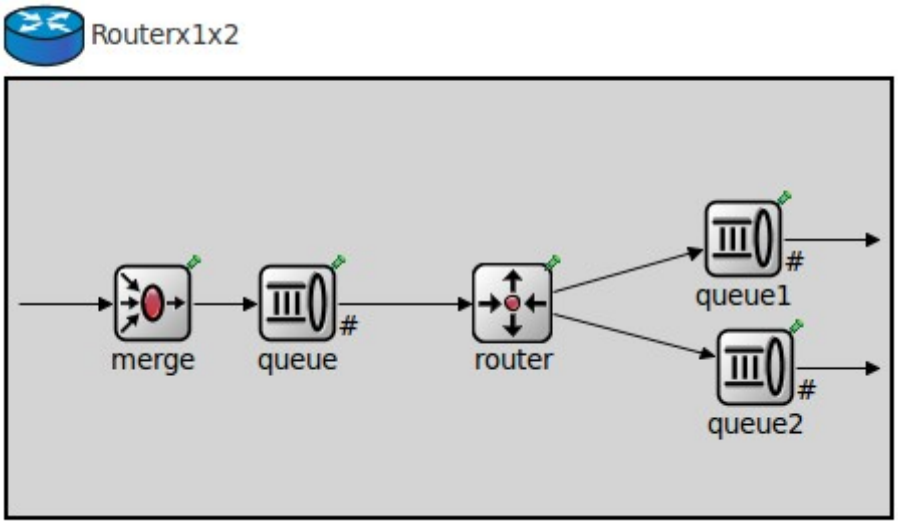
```
Define_Module(Source);
void Source::initialize()
{
    SourceBase::initialize();
    startTime = par("startTime");
    stopTime = par("stopTime");
    numJobs = par("numJobs");
    sending=true;
    nextchange=startTime.dbl()+gamma_d(0.352,1); //alpha*tetha = 0.352 sec,
tetha=1(assumed)

    // schedule the first message timer for start time
    scheduleAt(startTime, new cMessage("newJobTimer",1));
    if(par("VAD"))
     scheduleAt(nextchange, new cMessage("VAD",2));
}

void Source::handleMessage(cMessage *msg)
{
    ASSERT(msg->isSelfMessage());
    if(msg->getKind()==1){
             if ((numJobs < 0 || numJobs > jobCounter) && (stopTime < 0 || stopTime >
simTime()))
             {
                    // reschedule the timer for the next message
                    scheduleAt(simTime() + par("interArrivalTime").doubleValue(),
msg);
```

```
                    Job *job = createJob();
                    if(sending){
                     send(job, "out");
                    } else
                           delete job;
             }
             else
             {
                    // finished
                    delete msg;
             }
    }
    if(msg->getKind()==2){ // VAD toggle
              sending=!sending;
              scheduleAt(simTime().dbl() + (sending?gamma_d(0.352,1):weibull(0.650,1)),
msg);
       EV<< "VAD NOW "<<(sending?"":"NOT ")<<"SENDING"<<endl;
    }
}
```

2.3.2. *Maršrutētāja elementa izstrāde*

Modelī ir trīs tipu maršrutētāji:

- pirmais maršrutētājs – ar vienu ieeju un trim izejām;
- maršrutētāji ar vienu ieeju un vienu izeju;
- pēdējais maršrutētājs – ar trim ieejām un vienu izeju.



2.4. att. **Vispārīga maršrutētāja apakšmoduļi**

Attēlā 2.4. redzami maršrutētāja *N*x*M* apakšmoduļi. Attēlā parādīts maršrutētājs, kuram *N*=1 un *M*=2. Jebkurš maršrutētājs sastāv no:

- *Merge* elementa, kas apvieno visas ienākošās plūsmas vienā;
- *Queue* elementa, kas sevī ietver gan *PassiveQueue*, kur krājas visas ienākošās paketes, gan *Server*, kas katru no paketēm aizkavē uz noteiktu apstrādes laiku.

- **iebūvētās bibliotēkas** *queueinglib* elementa *Router*, kas nosaka, pa kuru izeju ārā no maršrutētāja ir jādodas katrai paketei;
- Izejas rindām *Queue* skaitā *M*.

Lai paātrinātu aprēķinus tīkla modeļa simulācijas izpildei, katrā no maršrutētājiem tiek atstāti tikai tie apakšmoduļi, kas nepieciešami.

Kā norādīts apakšnodaļas sākumā, eksistē nepieciešamība kontrolēt, pa kuru ceļu tiek nosūtītas pirmās paketes. Ceļu izvēli kontrolē *queueinglib* bibliotēkas elements *Router*. Tas tika modificēts, lai atbalstītu vajadzīgo funkcionalitāti.

Daļa no faila *Router.cc*:

```
void Router::handleMessage(cMessage *msg)
{
    int outGateIndex = -1;  // by default we drop the message

    switch (routingAlgorithm)
    {
        case ALG_RANDOM:
            outGateIndex = par("randomGateIndex");
            break;
        case ALG_ROUND_ROBIN:
            outGateIndex = rrCounter;
            rrCounter = (rrCounter + 1) % gateSize("out");
            break;
        case ALG_MIN_QUEUE_LENGTH:
            // TODO implementation missing
            outGateIndex = -1;
            break;
        case ALG_MIN_DELAY:
            // TODO implementation missing
            outGateIndex = -1;
            break;
        case ALG_MIN_SERVICE_TIME:
            // TODO implementation missing
            outGateIndex = -1;
            break;
        default:
            outGateIndex = -1;
            break;
    }
    if(remains>0){
       remains--;
       outGateIndex = par ("biasedGate");
    }
    // send out if the index is legal
    if (outGateIndex < 0 || outGateIndex >= gateSize("out"))
        error("Invalid output gate selected during routing");

    send(msg, "out", outGateIndex);
}
```

2.3.2.1. Maršrutētājs 1x3 (ar vienu ieeju un trim izejām)

Šī maršrutētāja funkcija ir sadalīt ienākošo plūsmu pa trim ceļiem proporcionāli parametriem *prob0*, *prob1* un *1-prob0-prob1*.

Maršrutētājs sastāv no modificēta apakšmoduļa *Router* un – tā kā tajā ir paredzēta iespēja norādīt apstrādes laiku – arī apakšmoduļa *Queue*.

Fails *router-1-3.ned*:

```
module Routerx1x3
{
    parameters:
        @display("i=abstract/router;bgb=439,217;bgl=2");
        volatile double prob0 = default(0.5);
        volatile double prob1 = default(0.4);
        volatile int weights = min(2,floor(max(0,(uniform(0,1)-prob0)/prob1+1))); //
(R-x)/y+1 (z=1-x-y), y≠0
        volatile int firstgate = default(0);
    gates:
        input in[];
        output out[];
    submodules:
        queue: Queue {
            @display("p=97,96");
            serviceTime = 0ms;
        }
        router: org.omnetpp.queueing.Router {
            @display("p=189,89");
            randomGateIndex = weights;
            biasedGate = firstgate;
            biasedCount = 10;
        }
    connections:
        queue.out --> IdealChannel --> router.in++;
        in++ --> queue.in++;
        router.out++ --> out++;
        router.out++ --> out++;
        router.out++ --> out++;
}
```

2.3.2.2. Maršrutētājs 1x1 (ar vienu ieeju un vienu izeju)

Šis maršrutētājs sastāv tikai no viena apakšmoduļa – *Queue,* jo viņa funkcija ir simulēt paketes apstrādi un padot to ārā pa vienīgo izeju.

Fails *router-1-1.ned*:

```
module Routerx1x1
{
    parameters:
        @display("i=abstract/router;bgb=439,217;bgl=2");
        volatile double entrydelay @unit(s) = default(0s);

    gates:
        input in[];
        output out[];
    submodules:
        queue: Queue {
            @display("p=97,96");
            serviceTime = entrydelay;
        }
    connections:
        in++ --> queue.in++;
        queue.out --> out++;
}
```

2.3.2.3. Maršrutētājs 3x1 (ar trim ieejām un vienu izeju)

Šajā maršrutētājā vienīgais apakšmodulis ir *Merge*, kas nodrošina, ka paketes no jebkuras ieejas tiks izsūtītas uz vienu un to pašu izeju.

Fails *router-3-1.ned*:

```
module Routerx3x1
{
    parameters:
        @display("i=abstract/router;bgb=439,217;bgl=2");
    gates:
        input in[];
        output out[];
    submodules:
        merge: Merge {
            @display("p=43,110");
        }
    connections:
        in++ --> IdealChannel --> merge.in++;
        in++ --> IdealChannel --> merge.in++;
        in++ --> IdealChannel --> merge.in++;
        merge.out --> out++;
}
```

2.3.3. Džiter-bufera elementa izstrāde

Džiter-buferis ir pilnīgi jauns elements, ko autors izstrādāja patstāvīgi. Tā implementācija ir atkarīga no konkrētā džiter-bufera vadības algoritma. Implementāciju būtība ir aprakstīta darba 3. nodaļā, bet pirmkods redzams 4. pielikumā.

Visu šo elementu kopīga iezīme ir statistikas uzskaite – katrs algoritms seko līdzi tam, cik paketes saņēmis, cik paketes nometis dažādu iemeslu dēļ, kā arī cik ilgi (vidēji) viena pakete uzturas buferī.

Faila *jitterbuffer.ned* daļa, kas apraksta viena bufera parametrus:

```
simple jitterbufferB
{
    @display("i=block/buffer");
    @statistic[dropped](title="the number of jobs dropped";
record=count;interpolationmode=none);
    @statistic[received](title="the number of jobs received";
record=count;interpolationmode=none);
    @statistic[servicetime](title="the number of seconds to process a job";
record=mean;interpolationmode=none);
    volatile double timesize @unit(s);
    volatile int jbsize = default(10);

    gates:
        input in;
        output voice;

}
```

2.3.4. Saņēmēja elementa izstrāde

Saņēmēja elements ir balstīts uz bibliotēkā *queueinglib* iekļauto *Sink* elementu. Tas ir papildināts ar statistikas skaitītāju, kas reģistrē, cik paketes nav pienākušas precīzi gaidītajā laikā. Tam pievienots arī papildus parametrs, kurā jānorāda pakešu pienākšanas intervāls, kam jāsakrīt ar avota elementa pakešu izsūtīšanas intervālu.

Fails *Sink.ned*:

```
simple Sink
{
    parameters:
        @group(Queueing);
        @display("i=block/sink");
        @signal[lifeTime](type="simtime_t");
        @signal[totalQueueingTime](type="simtime_t");
        @signal[totalDelayTime](type="simtime_t");
        @signal[totalServiceTime](type="simtime_t");
        @signal[queuesVisited](type="int");
        @signal[delaysVisited](type="int");
        @signal[generation](type="int");
        @statistic[lifeTime](title="lifetime of arrived jobs";
record=histogram;unit=s;interpolationmode=none);
        @statistic[totalQueueingTime](title="the total time spent in queues by arrived
jobs"; unit=s;interpolationmode=none);
        @statistic[totalDelayTime](title="the total time spent in delay nodes by
arrived jobs"; unit=s;interpolationmode=none);
        @statistic[totalServiceTime](title="the total time spent  by arrived jobs";
unit=s;interpolationmode=none);
        @statistic[queuesVisited](title="the total number of queues visited by arrived
jobs"; interpolationmode=none);
        @statistic[delaysVisited](title="the total number of delays visited by arrived
jobs"; interpolationmode=none);
        @statistic[generation](title="the generation of the arrived jobs";
interpolationmode=none);
        @statistic[dropped](title="dropped by codec";
record=count;interpolationmode=none);
        bool keepJobs = default(false); // whether to keep the received jobs till the
end of simulation
        volatile double timesize @unit("s");
    gates:
        input in[];
}
```

Lai gan tīkla modelis ir bāzēts uz G.729a un G.729ab kodekiem, darba ietvaros tiek pieņemts, ka kodeks māk atskaņot tikai to pakešu saturu, kas kodeku sasnieguši precīzi vajadzīgajā laikā. Šāds pieņēmums nodrošina, ka ir iespējams novērtēt džiter-bufera darbu neatkarīgi no blakus faktoriem.

Kodeka pārbaudes algoritms darbojas šādi:

1. tiek gaidīta pirmā kodekā ienākošā pakete;
2. kad tāda pakete ir saņemta, tā var būt vai nu pirmā izsūtītā pakete, vai arī tīkla traucējumu gadījumā – kāda cita pakete; tamdēļ, lietojot formulu (2.2), tiek **aprēķināts**, kad patiesībā bija jāpienāk pirmajai paketei;

3. visām turpmākajām paketēm tiek **pārbaudīta** tā pati sakarība (2.2); ja sakarība neizpildās, tad pakete ir pienākusi nelaikā un tiek atzīmēta statistikā kā neatskaņojama.

*Pirmās_Paketes_Laiks = Šībrīža_Laiks – (Saņemtās_Paketes_Numurs-1)*Intervāls* (2.2)

Tas, protams, nozīmē, ka pirmajai pakete nonākot buferī pa lēnu ceļu, bet vairumam pakešu – pa ātru, liela daļa pakešu tiks atzīmētas kā neatskaņotas. Taču šādas situācijas varbūtība ir aksiomātiski neliela. Pa ātru ceļu paketes atceļos stipri ātrāk, līdz ar to šādai situācijai ir nepieciešams, lai neviena no pirmajām paketēm nedodas pa ātru ceļu. Taču, ja tā patiešām notiek, tātad tīkls ir pārslogots (vai tml.) un visdrīzāk arī lielākā daļa pārējo pakešu turpinās nākt pa lēnu ceļu.

Šajā scenārijā var vilkt paralēles ar praksi, kad reizēm, veicot IP balss zvanu, jau no paša sākuma tā kvalitāte ir ļoti zema un neko nav iespējams saklausīt. Parasti šāds zvans vai nu pēc brīža pārtrūkst, vai arī sarunu partneri paši to pārtrauc, lai pārzvanītu vēlreiz. Gadās, ka, uzreiz veicot atkārtotu zvanu, kvalitāte ir ievērojami labāka.

Fails *Sink.cc*:

```
Define_Module(Sink);

void Sink::initialize()
{
    lifeTimeSignal = registerSignal("lifeTime");
    totalQueueingTimeSignal = registerSignal("totalQueueingTime");
    queuesVisitedSignal = registerSignal("queuesVisited");
    totalServiceTimeSignal = registerSignal("totalServiceTime");
    totalDelayTimeSignal = registerSignal("totalDelayTime");
    delaysVisitedSignal = registerSignal("delaysVisited");
    generationSignal = registerSignal("generation");
    dropSignal=registerSignal("dropped");
    keepJobs = par("keepJobs");
    starttime=0;
}

void Sink::handleMessage(cMessage *msg)
{
    Job *job = check_and_cast<Job *>(msg);

    // gather statistics
    emit(lifeTimeSignal, simTime()- job->getCreationTime());
    emit(totalQueueingTimeSignal, job->getTotalQueueingTime());
    emit(queuesVisitedSignal, job->getQueueCount());
    emit(totalServiceTimeSignal, job->getTotalServiceTime());
    emit(totalDelayTimeSignal, job->getTotalDelayTime());
    emit(delaysVisitedSignal, job->getDelayCount());
    emit(generationSignal, job->getGeneration());
    double z=par("timesize");
    if(starttime==0){ //first time
       starttime=simTime().dbl()-(atoi(msg->getName())-1)*z;
       EV << "real first packet should reach me at " << starttime <<"s"<<endl;
    } else{
      if(fabs(starttime+(atoi(msg->getName())-1)*z-simTime().dbl())>0.00001){
        EV << "got "<<msg->getName()<<" at "<<simTime().dbl()<<", but wanted this at "
```

```
<<(starttime+(atoi(msg->getName())-1)*z)<<endl;
        emit(dropSignal,1);
      }
    }
    if (!keepJobs)
        delete msg;
}

void Sink::finish()
{
    // TODO missing scalar statistics
}
```

3. DŽITER-BUFERA VADĪBAS ALGORITMI

Šajā nodaļā ir aprakstīti divi jau esoši džiter-bufera vadības algoritmi un trīs jauni algoritmi.

Esošie algoritmi, kas aprakstīti šajā nodaļā, ir:

1. Cisco(5, 6) un Asterisk(7) algoritms;
2. pirmā algoritma uzlabota versija, kas fiksē pēdējās no bufera uz kodeku izsūtītās paketes numuru.(9)

Būtiskās sastāvdaļas, kas nepieciešami labam algoritmam, ir:

- neliels bufera izmērs un neilga paketes aizkavēšana buferī;
- minimāls nomesto pakešu skaits;
- sekošana līdzi tam, lai nelaikā nenosūtītu paketes uz buferi.

Turpmākajās apakšnodaļās tiek aprakstīti 3.1. tabulā minētie algoritmi; tie turpmāk tiks saukti tā, kā norādīts tabulā. Lai gan tika radīti un izmēģināti vairāk nekā trīs jauni algoritmi, daļa no tiem tika atmesti, jo neizturēja sākotnējo testēšanu, vai arī tika radīts labāks algoritms, kas no atmestā atšķiras nebūtiski.

Lai novērtētu, vai algoritmu var ticami ieviest iekārtās ar ierobežotu skaitļošanas jaudu, katram algoritmam tiek arī veikts tā algoritmiskās sarežģītības novērtējums.

3.1. tabula

Algoritmu nosaukumu saīsinājumi un apraksti

Apakšnodaļas nosaukums un algoritma saīsinājums	*Algoritma apraksts*
Algoritms B	Cisco(5, 6) un Asterisk(7)
Algoritms CF	B algoritma uzlabota versija, kas fiksē pēdējās no bufera uz kodeku izsūtītās paketes numuru un nepieņem buferī par to vecākas paketes(9)
Algoritms E	algoritms seko līdzi kodekam reāli nepieciešamajiem pakešu laikiem un attiecīgi izvieto ienākošās paketes (arī tukšā buferī)
Algoritms AD2	adaptīvs algoritms, kas palielina džiter-buferi tiklīdz tas nepieciešams, bet samazina pēc noteikta laika intervāla
Algoritms AD3	adaptīvs algoritms, kas palielina džiter-buferi tiklīdz tas nepieciešams, bet samazina, kad džiter-buferis ir veiksmīgi pieņēmis vairāk kā 10 paketes un noraidījis mazāk nekā pieņēmis

3.1. Algoritms B

Šim algoritmam ir raksturīgi "pazaudēt atmiņu", ja džiter-buferis jebkāda iemesla dēļ tiek iztukšots. Tukša džiter-bufera gadījumā tas pirmo ienākošo paketi vienmēr novietos pēc iespējas tālāk no kodeka, kā arī nespēs izvērtēt, vai konkrēto paketi ir vērts ņemt vai nē.

3.1.1. Algoritma soļi

Solis 0: beigt darbu.

Ik pēc *N* ms: doties uz soli 1.

Kad saņemta ienākošā pakete: doties uz soli 10.

Solis 1: ja *buferis[0]* satur paketi, tad

Solis 1.1: izsūtīt to uz kodeku.

Solis 2: pārvietot visus *buferis[i+1]* uz *buferis[i]*, sākot no lielākā *i*; beigt darbu.

Solis 10: skaitot no 0 uz augšu, atrast tādu pirmo *j*, ka *buferis[j]* satur paketi.

Solis 11: ja tāds *j* neeksistē, tad

Solis 11.1: ievietot saņemto paketi *buferis[M-1]*, kur *M* ir džiter-bufera izmērs; beigt darbu.

Solis 12: noskaidrot paketes, kura atrodas *buferis[j]*, kārtas numuru *z*; ievietot saņemto paketi (ar paketes numuru *u*) vietā *buferis[j+u-z]*, ja tāda vieta eksistē; beigt darbu.

3.1.2. Darbības piemērs

Džiter-bufera vadības algoritma B darbības piemērs attēlots 3.2. tabulā. Tiek pieņemts, ka džiter-bufera izmērs ir 3 paketes un kodekam ir jāsaņem pakete ik pēc 20ms. Tabulas ailes ir sakārtotas **hronoloģiski.** Ja kādā ailē laiks nav norādīts, tad tas piemēra ietvaros tas var būt jebkurš brīdis starp iepriekšējo un nākamo aili.

Piemērā džiter-bufera vadības algoritms izmet paketes ar numuriem 6, 3 un 4, jo to pienākšanas brīdī buferī nav atbilstošas vietas to glabāšanai.

3.2. tabula

Algoritma B darbības piemērs

Laiks (ms)	*Tīkls →*	*→ Buferis →*	*→ Kodeks*
0	5 4 3 9 6 2 1	▒ ▒ ▒	
	5 4 3 9 6 2	1 ▒ ▒	

20	5 4 3 9 6 2	▒ 1 ▒	
40	5 4 3 9 6 2	▒ ▒ 1	
	5 4 3 9 6	▒ 2 1	
60	5 4 3 9 6	▒ ▒ 2	1
	5 4 3 9	▒ ▒ 2	1
80	5 4 3 9	▒ ▒ ▒	2 1
	5 4 3	9 ▒ ▒	2 1
100	5 4	▒ 9 ▒	▒ 2 1
	5 4	▒ 9 ▒	▒ 2 1
	5	▒ 9 ▒	▒ 2 1
120	5	▒ ▒ 9	▒ ▒ 2 1
140	5	▒ ▒ ▒	9 ▒ ▒ 2 1
		5 ▒ ▒	9 ▒ ▒ 2 1
160		▒ 5 ▒	▒ 9 ▒ ▒ 2 1
180		▒ ▒ 5	▒ ▒ 9 ▒ ▒ 2 1
200		▒ ▒ ▒	5 ▒ ▒ 9 ▒ ▒ 2 1

3.1.3. Sarežģītības novērtējums

Šī algoritma sarežģītība pie paketes izsūtīšanas ir *O(M)*, ko var samazināt līdz *O(1)*, lietojot cirkulāru buferi, ja tas nepieciešams. Sarežģītība pie paketes saņemšanas ir *O(M)*, kur *M* ir džiter-bufera izmērs.

3.2. Algoritms CF

Šis algoritms ir B algoritma uzlabojums. Tas atceras pēdējo paketes numuru, kas nosūtīta uz kodeku un nepieļauj, ka buferī ienāks pakete ar mazāku vai tādu pašu numuru. Taču arī šis algoritms tukša džiter-bufera gadījumā pirmo ienākošo paketi vienmēr novietos pēc iespējas tālāk no kodeka.

Algoritms ir pārrakstīts no 2. pielikuma.

3.2.1. Algoritma soļi

Solis 0: *lo* ← 0; beigt darbu.

Ik pēc *N* ms: doties uz soli 1.

Kad saņemta ienākošā pakete: doties uz soli 10.

Solis 1: ja *buferis[0]* satur paketi, tad

Solis 1.1: noskaidrot paketes, kura atrodas *buferis[0]*, kārtas numuru *z*.

Solis 1.2: ja *lo* < *z*, tad

Solis 1.2.1: *lo* ← *z*.

Solis 1.3: izsūtīt paketi no *buferis[0]* uz kodeku.

Solis 2: pārvietot visus *buferis[i+1]* uz *buferis[i]*, sākot no lielākā *i*; beigt darbu.

Solis 10: *u* ir saņemtās paketes kārtas numurs; ja *lo* > *u*,

Solis 10.1: tad beigt darbu.

Solis 11: skaitot no 0 uz augšu, atrast tādu *pirmo j*, ka *buferis[j]* satur paketi.

Solis 12: ja tāds *j* neeksistē, tad

Solis 12.1: ievietot saņemto paketi *buferis[M-1]*, kur *M* ir bufera izmērs; beigt darbu;

Solis 13: noskaidrot paketes, kura atrodas *buferis[j]*, kārtas numuru *z*; ievietot saņemto paketi (ar paketes numuru *u*) vietā *buferis[j+u-z]*, ja tāda vieta eksistē; beigt darbu.

3.2.2. Darbības piemērs

Džiter-bufera vadības algoritma CF darbības piemērs attēlots 3.3. tabulā. Tiek pieņemts, ka džiter-bufera izmērs ir 3 paketes un kodekam ir jāsaņem pakete ik pēc 20ms. Tabulas ailes ir sakārtotas **hronoloģiski.** Ja kādā ailē laiks nav norādīts, tad tas piemēra ietvaros tas var būt jebkurš brīdis starp iepriekšējo un nākamo aili.

Dotajā piemērā džiter-bufera vadības algoritms izmet paketes ar numuriem 6, 3 un 4, jo to pienākšanas brīdī buferī nav atbilstošas vietas to glabāšanai, bet pakete ar numuru 5 tiek izmesta tāpēc, ka algoritms **atceras,** ka 140ms **pēdējā uz buferi izsūtītā pakete** ir ar kārtas numuru 9.

3.3. tabula

Algoritma CF darbības piemērs

Laiks (ms)	*Tīkls →*	*→ Buferis →*	*→ Kodeks*
0	5 4 3 9 6 2 1	▒ ▒ ▒	
	5 4 3 9 6 2	1 ▒ ▒	
20	5 4 3 9 6 2	▒ 1 ▒	
40	5 4 3 9 6 2	▒ ▒ 1	
	5 4 3 9 6	▒ 2 1	
60	5 4 3 9 6	▒ ▒ 2	1

	5 4 3 9	▒ ▒ 2	1
80	5 4 3 9	▒ ▒ ▒	2 1
	5 4 3	9 ▒ ▒	2 1
100	5 4	▒ 9 ▒	▒ 2 1
	5 4	▒ 9 ▒	▒ 2 1
	5	▒ 9 ▒	▒ 2 1
120	5	▒ ▒ 9	▒ ▒ 2 1
140	5	▒ ▒ ▒	9 ▒ 2 1
		▒ ▒ ▒	9 ▒ 2 1
160		▒ ▒ ▒	▒ 9 ▒ 2 1

3.2.3. Sarežģītības novērtējums

Šī algoritma sarežģītība pie paketes izsūtīšanas ir *O(M)*, ko var samazināt līdz *O(1)*, lietojot cirkulāru buferi, ja tas nepieciešams. Sarežģītība pie paketes saņemšanas ir *O(M)*, kur *M* ir džiter-bufera izmērs.

3.3. Algoritms E

Algoritms E ir fiksēts džiter-bufera vadības algoritms, kas balstīts uz laika izsekošanu. Šis algoritms vienmēr atceras, kuru paketi kodeks šobrīd gaida, un uz to arī tie balstīta tā darbība. Ja kaut viena pakete jau ir tikusi izsūtīta uz kodeku, tukša džiter-bufera gadījumā tas precīzi zinās, kur novietot katru paketi, un vai vispār to pieņemt.

3.3.1. Algoritma soļi

Solis 0: *lt* ← 0; beigt darbu.

Ik pēc *N* ms: doties uz soli 1.

Kad saņemta ienākošā pakete: doties uz soli 10.

Solis 1: ja *buferis[0]* satur paketi, tad

Solis 1.1: noskaidrot paketes, kura atrodas *buferis[0]*, izsūtīšanas laiku *t*.

Solis 1.2: *lt* ← *max(lt+N , t)*.

Solis 1.3: izsūtīt paketi no *buferis[0]* uz kodeku.

Solis 2: ja *buferis[0]* nesatur paketi **un** *lt > 0*, tad

Solis 2.1: *lt* ← *lt+N;*

Solis 2: pārvietot visus *buferis[i+1]* uz *buferis[i]*, sākot no lielākā *i*; beigt darbu.

Solis 10: *v* ir saņemtās paketes izsūtīšanas laiks; ja $lt \geq v$, tad

Solis 10.1: beigt darbu.

Solis 11: ja $lt > 0$, tad

Solis 11.1: ievietot saņemto paketi vietā *buferis[(v-lt)/N-1]*, ja tāda vieta eksistē un ir brīva; beigt darbu.

Solis 12: skaitot no 0 uz augšu, atrast tādu pirmo j, ka *buferis[j]* satur paketi.

Solis 13: ja tāds *j* neeksistē, tad

Solis 13.1: ievietot saņemto paketi *buferis[M-1]*, kur *M* ir bufera izmērs; beigt darbu,

Solis 14: noskaidrot paketes, kura atrodas *buferis[j]*, izsūtīšanas laiku *w*; ievietot saņemto paketi (ar izsūtīšanas laiku *v*) vietā *buferis[j+(v-w)/N]*, ja tāda vieta eksistē un ir brīva; beigt darbu.

3.3.2. Darbības piemērs

Džiter-bufera vadības algoritma E darbības piemērs attēlots 3.4. tabulā. Tiek pieņemts, ka džiter-bufera izmērs ir 3 paketes un kodekam ir jāsaņem pakete ik pēc 20ms. Tabulas ailes ir sakārtotas **hronoloģiski.** Ja kādā ailē laiks nav norādīts, tad tas piemēra ietvaros tas var būt jebkurš brīdis starp iepriekšējo un nākamo aili.

Tabulā 3.4. redzamajā piemērā džiter-bufera vadības algoritms izmet 6. paketi, jo tās pienākšanas brīdī buferī nav atbilstošas vietas, kur to var saglabāt. Algoritms izmet 9. paketi, jo tā ir atnākusi stipri par ātru, bet 3. un 5., jo tās atnākušas par vēlu. Lai gan buferī attiecīgajā brīdī ir brīva vieta, džiter-bufera vadības algoritms E **zina, ka kodeks tās tikko jau vēlējās atskaņot** un līdz ar to ir par vēlu tās nosūtīt uz kodeku tikai tagad.

3.4. tabula

Algoritma E darbības piemērs

Laiks (ms)	*Tīkls →*	*→ Buferis →*	*→ Kodeks*
0	5 4 3 9 6 2 1	▒ ▒ ▒	
	5 4 3 9 6 2	1 ▒ ▒	
20	5 4 3 9 6 2	▒ 1 ▒	
40	5 4 3 9 6 2	▒ ▒ 1	
	5 4 3 9 6	▒ 2 1	
60	5 4 3 9 6	▒ ▒ 2	1

	5 4 3 9	▒ ▒ 2	1
80	5 4 3 9	▒ ▒ ▒	2 1
	5 4 3	▒ ▒ ▒	2 1
100	5 4 3	▒ ▒ ▒	▒ 2 1
	5 4	▒ ▒ ▒	▒ 2 1
	5	▒ ▒ 4	▒ 2 1
120	5	▒ ▒ ▒	4 ▒ 2 1
140	5	▒ ▒ ▒	▒ 4 ▒ 2 1
		▒ ▒ ▒	▒ 4 ▒ 2 1
160		▒ ▒ ▒	▒ ▒ 4 ▒ 2 1

3.3.3. Sarežģītības novērtējums

Šī algoritma sarežģītība pie paketes izsūtīšanas ir *O(M)*, kur *M* ir džiter-bufera izmērs. Ja tas nepieciešams, algoritma sarežģītību var samazināt līdz *O(1)*, lietojot cirkulāru buferi. Sarežģītība pie paketes saņemšanas ir *O(1)*, jo tiklīdz ir izsūtīta kaut viena pakete, tas darbojas lineārā laikā.

3.4. Algoritms AD2

Džiter-bufera izmēra ziņā šis ir adaptīvs algoritms, taču tas pieturas pie fiksēta pakešu izsūtīšanas laika uz kodeku. Algoritms ir bāzēts E algoritma versijā. Būtiskā atšķirība ir tāda, ka algoritmam AD2 nav fiksēts bufera izmērs – džiter-buferim tiek norādīts minimālais un maksimālais izmērs. Algoritms uzsāk darbu ar minimālo bufera izmēru, bet ir spējīgs to palielināt tiklīdz pienākošo paketi nav iespējams ietilpināt esošajā buferī. Izmērs **pakāpeniski** samazināsies, ja noteiktu laiku nebūs bijusi vajadzība to palielināt.

Simulējot tīkla darbību, apstākļos, kad fiksētā džiter-bufera izmērs ir *B*, adaptīvā džiter-bufera minimālais izmērs tiek uzstādīts uz $\lceil B*0.75 \rceil$, bet maksimālais – uz *min(10,B*2)*.

Attēlojot šī algoritma darbības rezultātus tiks ņemts vērā vidējais faktiskais bufera izmērs.

3.4.1. Algoritma soļi

Solis 0: $lt \leftarrow 0$; $st \leftarrow 0$; beigt darbu.

Ik pēc *N* ms: doties uz soli 1.

Kad saņemta ienākošā pakete: doties uz soli 10.

Solis 1: ja *buferis[0]* satur paketi, tad

Solis 1.1: noskaidrot paketes, kura atrodas *buferis[0]*, izsūtīšanas laiku *t*.

Solis 1.2: $lt \leftarrow max(lt+N\ ,\ t)$.

Solis 1.3: izsūtīt paketi no *buferis[0]* uz kodeku.

Solis 2: ja *buferis[0]* nesatur paketi **un** $lt > 0$, tad

Solis 2.1: $lt \leftarrow lt+N$.

Solis 2: pārvietot visus *buferis[i+1]* uz *buferis[i]*, sākot no lielākā *i*.

Solis 3: ja $st = 0$ **un** $M > BM$, kur *M* ir bufera izmērs un *BM* ir bufera minimālais izmērs, tad

Solis 3.1: $st \leftarrow M;\ M \leftarrow M-1$.

Solis 4: ja $st > 0$, tad

Solis 4.1: $st \leftarrow st-1$.

Solis 5: beigt darbu.

Solis 10: *v* ir saņemtās paketes izsūtīšanas laiks; ja $lt \geq v$, tad

Solis 10.1: beigt darbu.

Solis 11: ja $lt > 0$, tad

Solis 11.1: ievietot saņemto paketi vietā *buferis[(v-lt)/N-1]*, ja tāda vieta eksistē un ir brīva.

Solis 11.2: ja tāda vieta bija pieejama, tad

Solis 11.2.1: beigt darbu.

Solis 11.3: ja tāda vieta nebija pieejama, tad

Solis 11.3.1: ja $M < MM$, kur *M* ir bufera izmērs, bet *MM* ir bufera maksimālais izmērs, tad

Solis 11.3.1.1: $y \leftarrow min(MM,(v-lt)/N)$.

Solis 11.3.1.2: ja $y > M$, tad

Solis 11.3.1.2.1: $st \leftarrow y;\ M \leftarrow y$; doties uz soli 11.

Solis 11.3.2: beigt darbu.

Solis 12: skaitot no 0 uz augšu, atrast tādu pirmo *j*, ka *buferis[j]* satur paketi.

Solis 13: ja tāds *j* neeksistē, tad

Solis 13.1: ievietot saņemto paketi *buferis[M-1]*, kur *M* ir bufera izmērs; beigt darbu.

Solis 14: noskaidrot paketes, kura atrodas *buferis[j]*, izsūtīšanas laiku *w*; ievietot saņemto paketi (ar izsūtīšanas laiku *v*) vietā *buferis[j+(v-w)/N]*, ja tāda vieta eksistē un ir brīva; beigt darbu.

3.4.2. Darbības piemērs

Džiter-bufera vadības algoritma AD2 darbības piemērs attēlots 3.5. tabulā. Tiek pieņemts, ka džiter-bufera minimālais (un tātad arī sākuma) izmērs ir 3 paketes, bet **maksimālais izmērs ir 5 paketes**. Kodekam ir jāsaņem pakete ik pēc 20ms. Tabulas ailes ir sakārtotas **hronoloģiski**. Ja kādā ailē laiks nav norādīts, tad tas piemēra ietvaros tas var būt jebkurš brīdis starp iepriekšējo un nākamo aili.

Dotajā piemērā algoritms AD2, saņemot paketi ar numuru 6 un redzot, ka tai šobrīd nav vietas, **nekavējoties palielina džiter-bufera izmēru** līdz 5 paketēm un ievieto paketi ar numuru 6 tai atbilstošajā vietā. Vēlāk algoritms izmet 9. paketi, jo tā ir atnākusi stipri par ātru, bet 3. un 5., jo tās atnākušas par vēlu. Analoģiski algoritmam E, arī šis algoritms zina, ko kodeks katrā brīdi sagaida.

3.5. tabula

Algoritma AD2 darbības piemērs

Laiks (ms)	*Tīkls →*	*→ Buferis →*	*→ Kodeks*
0	5 4 3 9 6 2 1	▒ ▒ ▒	
	5 4 3 9 6 2	1 ▒ ▒	
20	5 4 3 9 6 2	▒ 1 ▒	
40	5 4 3 9 6 2	▒ ▒ 1	
	5 4 3 9 6	▒ 2 1	
60	5 4 3 9 6	▒ ▒ 2	1
	5 4 3 9	6 ▒ ▒ ▒ 2	1
80	5 4 3 9	▒ 6 ▒ ▒ ▒	2 1
	5 4 3	▒ 6 ▒ ▒ ▒	2 1
100	5 4 3	▒ ▒ 6 ▒ ▒	▒ 2 1
	5 4	▒ ▒ 6 ▒ ▒	▒ 2 1
	5	▒ ▒ 6 ▒ 4	▒ 2 1
120	5	▒ ▒ ▒ 6 ▒	4 ▒ 2 1
140	5	▒ ▒ ▒ ▒ 6	▒ 4 ▒ 2 1

		▒ ▒ ▒ ▒ 6	▒ 4 ▒ 2 1
160		▒ ▒ ▒ ▒ ▒	6 ▒ 4 ▒ 2 1
180		▒ ▒ ▒ ▒	▒ 6 ▒ 4 ▒ 2 1

3.4.3. Sarežģītības novērtējums

Šī algoritma sarežģītība pie paketes izsūtīšanas ir *O(M)*, kur *M* ir džiter-bufera vidējais izmērs. Sarežģītība pie paketes saņemšanas ir *O(2)=O(1)*, jo tiklīdz ir izsūtīta kaut viena pakete, algoritms darbojas lineārā laikā.

3.5. Algoritms AD3

Tāpat kā algoritms AD2, arī šis algoritms džiter-bufera izmēra ziņā ir adaptīvs, bet, vērtējot pēc pakešu izsūtīšanas laika uz kodeku, fiksēts. Algoritms ir identisks AD2, izņemot daļu, kas atbild par izmēra samazināšanu.

Šis algoritms samazinās džiter-bufera izmēru par 1, ja tam kopš pēdējās bufera izmēra izmaiņas būs **izdevies** buferī **ievietot vairāk par 10 paketēm** un buferis būs **pieņēmis vairāk pakešu nekā izmetis**.

Simulējot tīkla darbību, apstākļos, kad fiksētā džiter-bufera izmērs ir *B*, adaptīvā džiter-bufera minimālais izmērs tiek uzstādīts uz $\lceil B*0.75 \rceil$, bet maksimālais uz *min(10,B*2)*.

Attēlojot šī algoritma darbības rezultātus tiks ņemts vērā vidējais faktiskais bufera izmērs.

3.5.1. Algoritma soļi

Solis 0: $lt \leftarrow 0;\ ok \leftarrow 0;\ drop \leftarrow 0;$ beigt darbu.

Ik pēc *N* ms: doties uz soli 1.

Kad saņemta ienākošā pakete: doties uz soli 10.

Solis 1: ja *buferis[0]* satur paketi, tad

Solis 1.1: noskaidrot paketes, kura atrodas *buferis[0]*, izsūtīšanas laiku *t*.

Solis 1.2: $lt \leftarrow max(lt+N\ ,\ t)$.

Solis 1.3: izsūtīt paketi no *buferis[0]* uz kodeku.

Solis 2: ja *buferis[0]* nesatur paketi **un** $lt > 0$, tad

Solis 2.1: $lt \leftarrow lt+N$.

Solis 2: pārvietot visus *buferis[i+1]* uz *buferis[i]*, sākot no lielākā *i*.

Solis 3: ja *ok > 10* **un** *ok > drop* **un** *M > BM*, kur *M* ir bufera izmērs un *BM* ir bufera minimālais izmērs, tad

Solis 3.1: *ok ← 0; drop ← 0; M ← M-1.*

Solis 4: beigt darbu.

Solis 10: *v* ir saņemtās paketes izsūtīšanas laiks; ja *lt ≥ v*, tad

Solis 10.1: *drop ← drop+1*; beigt darbu.

Solis 11: ja *lt > 0*, tad

Solis 11.1: ievietot saņemto paketi vietā *buferis[(v-lt)/N-1]*, ja tāda vieta eksistē un ir brīva.

Solis 11.2: ja tāda vieta bija pieejama, tad

Solis 11.2.1: *ok ← ok+1*; beigt darbu.

Solis 11.3: ja tāda vieta nebija pieejama, tad

Solis 11.3.1: ja *M < MM*, kur *M* ir bufera izmērs, bet *MM* ir bufera maksimālais izmērs, tad

Solis 11.3.1.1: *y ← min(MM,(v-lt)/N).*

Solis 11.3.1.2: *ja y > M,* tad

Solis 11.3.1.2.1: *ok ← 0; drop ← 0; M ← y*; doties uz soli 11.

Solis 11.3.2: *drop ← drop+1;* beigt darbu.

Solis 12: skaitot no 0 uz augšu, atrast tādu pirmo *j*, ka *buferis[j]* satur paketi.

Solis 13: ja tāds *j* neeksistē, tad

Solis 13.1: ievietot saņemto paketi *buferis[M-1]*, kur *M* ir bufera izmērs; *ok ← ok+1;* beigt darbu.

Solis 14: noskaidrot paketes, kura atrodas *buferis[j],* izsūtīšanas laiku *w*; ievietot saņemto paketi (ar izsūtīšanas laiku *v*) vietā *buferis[j+(v-w)/N]*, ja tāda vieta eksistē un ir brīva.

Solis 15: ja izdevās ievietot, tad

Solis 15.1: *ok ← ok+1*.

Solis 16: ja izdevās ievietot, tad

Solis 16.1: *drop ← drop+1*.

Solis 17: beigt darbu.

3.5.2. Darbības piemērs

Džiter-bufera vadības algoritma AD3 darbības piemērs attēlots 3.6. tabulā. Tiek pieņemts, ka džiter-bufera minimālais (un tātad arī sākuma) izmērs ir 3 paketes, bet maksimālais izmērs ir 5 paketes. Kodekam ir jāsaņem pakete ik pēc 20ms. Tabulas ailes ir sakārtotas hronoloģiski. Ja kādā ailē laiks nav norādīts, tad tas piemēra ietvaros tas var būt jebkurš brīdis starp iepriekšējo un nākamo aili.

Šī algoritma atšķirība no AD2 izpaužas tajā, kādos apstākļos tiek samazināts džiter-bufera izmērs. Tabulā 3.6. redzams, ka algoritms džiter-buferi nesamazina 180ms. Ja salīdzina tabulas 3.5. un 3.6., tad ir uzskatāmi redzams, ka tā ir vienīgā atšķirība dotā piemērā ietvaros.

3.6. tabula

Algoritma AD3 darbības piemērs

Laiks (ms)	*Tīkls →*	*→ Buferis →*	*→ Kodeks*
0	5 4 3 9 6 2 1	▒ ▒ ▒	
	5 4 3 9 6 2	1 ▒ ▒	
20	5 4 3 9 6 2	▒ 1 ▒	
40	5 4 3 9 6 2	▒ ▒ 1	
	5 4 3 9 6	▒ 2 1	
60	5 4 3 9 6	▒ ▒ 2	1
	5 4 3 9	6 ▒ ▒ 2	1
80	5 4 3 9	▒ 6 ▒ ▒ ▒	2 1
	5 4 3	▒ 6 ▒ ▒ ▒	2 1
100	5 4 3	▒ ▒ 6 ▒ ▒	▒ 2 1
	5 4	▒ ▒ 6 ▒ ▒	▒ 2 1
	5	▒ ▒ 6 ▒ 4	▒ 2 1
120	5	▒ ▒ ▒ 6 ▒	4 ▒ 2 1
140	5	▒ ▒ ▒ ▒ 6	▒ 4 ▒ 2 1
		▒ ▒ ▒ ▒ 6	▒ 4 ▒ 2 1
160		▒ ▒ ▒ ▒ ▒	6 ▒ 4 ▒ 2 1
180		▒ ▒ ▒ ▒ ▒	▒ 6 ▒ 4 ▒ 2 1

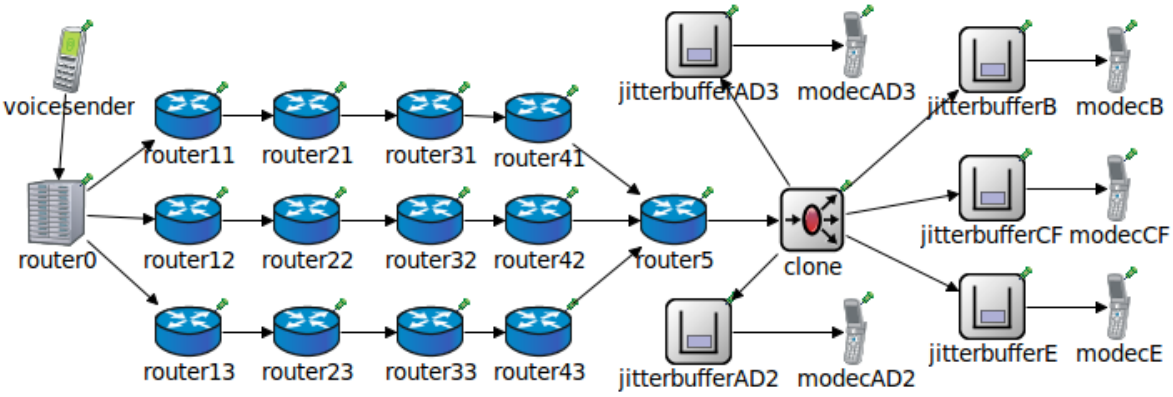
3.5.3. Sarežģītības novērtējums

Šī algoritma sarežģītība pie paketes izsūtīšanas ir *O(M)*, kur *M* ir džiter-bufera vidējais izmērs. Sarežģītība pie paketes saņemšanas ir *O(2)=O(1)*, jo tiklīdz ir izsūtīta kaut viena pakete, algoritms darbojas lineārā laikā.

4. ALGORITMU DARBĪBAS REZULTĀTI

Šajā nodaļā ir aprakstīti džiter-bufera vadības algoritmu darbības simulācijas rezultāti un veids, kā tie iegūti.

Lai varētu efektīvi un esot spēkā vienādiem nosacījumiem salīdzināt dažādu algoritmu darbību, tīkla simulācijas modelī uzreiz aiz elementa *router5* tika ievietots bibliotēkā *queueinglib* ietilpstošais elements *Clone,* aiz kura saslēgti visi pieci apskatāmiem džiter-bufera vadības algoritmi (4.1. attēls). Katram no viņiem arī pievienots atsevišķs kodeks, lai varētu ievākt statistiku paralēli.



4.1. att. **Tīkla simulācijas modelis, kas izmantots rezultātu ieguvei**

Simulācijas procedūra bija šāda:

1. elementa *voicesender* parametros tiek uzstādīts, vai VAD ir ieslēgts vai izslēgts;
2. elementa *router0* parametros tiek uzstādīta varbūtība paketēm izvēlēties labāko ceļu un vidējo ceļu; atlikušajos gadījumos maršrutētājs nosūta paketi pa sliktāko ceļu;
3. elementa *router0* parametros tiek uzstādīts, pa kuru ceļu jānosūta pirmās 10 paketes;
4. tīkls tiek palaists; palaišanas brīdī tiek ievadīts džiter-bufera izmērs;
5. tīkls darbība tiek simulēta tieši pus stundu jeb 1800 simulācijas sekundes; šajā laikā elementam *voicesender* būtu jārada ap 90 tūkstošiem pakešu;
6. tiek izsaukta funkcija *finish* un savākti dati;
7. tīkls tiek vēlreiz darbināts (5. solis), bet šoreiz jau ar citu bufera izmēru; tas tiek atkārtots, kamēr ir iegūti dati par visiem bufera izmēriem no 1 līdz 10 paketēm.

4.1. Datu apkopošana

Lai sistematizētu un atvieglotu datu apkopošanu un grafiku ģenerēšanu, pēc *finish* funkcijas izsaukšanas datu savākšanai un apkopošanai tiek izmantoti autora radīti skripti un izklājlapas.

Skripts *fetch-results:*

```
#!/bin/bash
cat General-0.sca |grep scalar|grep -v router|sed 's/Network\.//'| \
 awk '{print($2,$3, $4)}' > r.$1.stat
```

Šis skripts tiek izmantots, lai savāktu pēdējās simulācijas datus (ar konkrētā brīža bufera izmēru). Tas jāizsauc no komandrindas datu mapē un tam jānorāda šībrīža džiter-bufera izmērs, piemēram, `./fetch-results 4`.

Pēc tam, kad visi atsevišķie dati par dažādiem bufera izmēriem šajā tīkla konfigurācijā ir savākti, jāizsauc skripts *gen-tab:*

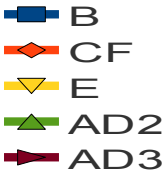
```
#!/bin/bash
for ((x=1;x<=10;x++)); do
 y=$(cat r.$x.stat | grep created | cut -d  " " -f 3)
 z=$(cat r.$x.stat | grep received | head -1 | cut -d  " " -f 3)
 echo -ne $x\\t$y\\t$z\\t
for yy in B CF E AD2 AD3; do
 pff1=$(cat r.$x.stat |grep $yy | grep jitterbuffer | grep " dropped" | \
  cut -d  " " -f 3)
 pff2=$(cat r.$x.stat |grep $yy | grep modec | grep " dropped" | cut -d  " " -f 3)
 st=$(cat r.$x.stat |grep $yy | grep jitterbuffer | grep servicetime | \
  cut -d  " " -f 3)
 echo -ne $pff1\\t$pff2\\t$st\\t
done
 s2=$(cat r.$x.stat | grep jitterbufferAD2 | grep jbsizea | cut -d  " " -f 3)
 s3=$(cat r.$x.stat | grep jitterbufferAD3 | grep jbsizea | cut -d  " " -f 3)
 echo -e $s2\\t$s3
done
```

Šis skripts uz ekrāna vai failā izvadīs ar tabulācijas simboliem atdalītus apkopotus datus, kurus var kopēt iekšā tam speciāli sagatavotā izklājlapā, kas izstrādāta, balstoties uz darba 1. nodaļā apskatīto teoriju, un spēj uzreiz attēlot datus vajadzīgo grafiku veidā, kā arī aprēķināt pārraides kvalitātes faktoru *R* un subjektīvo vērtējumu MOS skalā. Lai gan pētījumā(27) tika dotas vienkāršotas MOS aprēķina metodoloģijas, autors nolēma tās neizmantot, bet gan rēķināt MOS atbilstoši ITU-T rekomendācijām pēc formulas (1.2), jo maģistra darba ietvaros tika iegūti visi nepieciešamie mērījumi un parametri.

Jāņem vērā, ka aprēķinātie MOS vērtējumi ir spēkā tieši G.729a un G.729ab kodekiem un visdrīzāk neprecīzi parāda attiecīgo džiter-bufera vadības algoritmu ietekmi, ja tiek izmantoti citi kodeki.

Darba rezultātus var attēlot kā funkciju starp balss pārraides kvalitātes rādītājiem un džiter-bufera izmēru, kā arī ir iespējams novērot šīs funkcijas izmaiņas, mainot datortīkla

parametrus. **Grafikos katrs džiter-bufera vadības algoritms tiek attēlots atbilstoši leģendai, kas redzama 4.2. attēlā.** Ja kādā grafikā ir divas ordinātu asis, tad nepārtrauktās līknes norāda nomesto vai bojāto pakešu īpatsvaru, bet augošās – bufera vidējo latentumu.



4.2. att. **Džiter-bufera vadības algoritmu līkņu apzīmējumu saraksts**

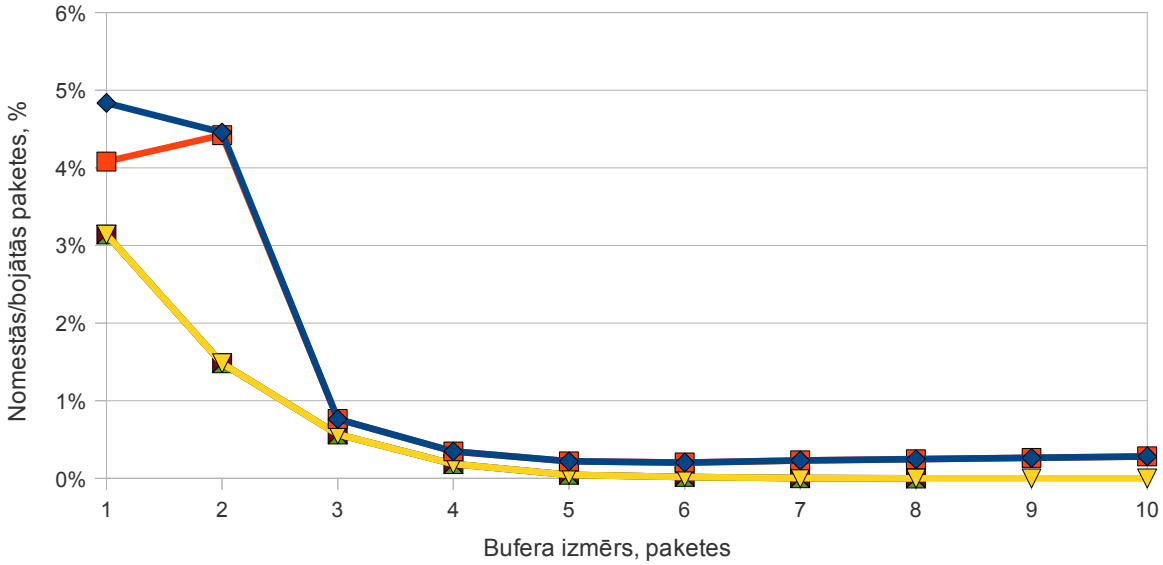
Turpmākajās apakšnodaļās ar vārdiem "tīkla sadalījums X/Y/Z-F" jāsaprot tāds tīkla modelis, kur X% pakešu iet pa ātrāko ceļu, Y% pa vidējo, bet Z% pa lēnāko, turklāt atkarībā no F pirmās paketes tiek izsūtītas par F=0 – ātrāko, F=1 – vidējo, F=2 – lēnāko ceļu. Ja F nav norādīts, tad jāpieņem, ka pirmās paketes tiek izsūtītas pa ātrāko ceļu.

Visi testi veikti ar ieslēgtu VAD/DTX, ja vien nav atsevišķi norādīts citādāk.

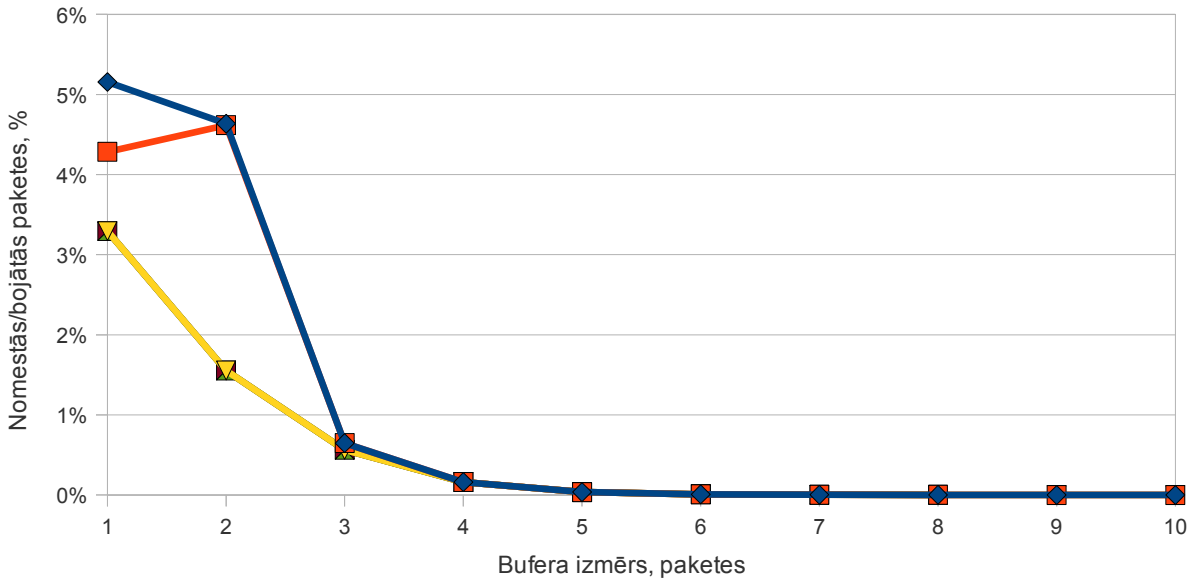
4.2. VAD/DTX ietekme

Lai noteiktu, vai VAD/DTX kaut kādā veidā ietekmē džiter-bufera vadības algoritma sniegumu, tika izpildītas divas simulācijas ar tīkla sadalījumu 97/3/0. Vienā no tām elements *voicesender* darbojās ar VAD/DTX, bet otrā bez.

Kā redzams, salīdzinot attēlus 4.3. un 4.4., algoritmi B un CF ir salīdzinoši jūtīgi pret VAD, uzrādot sliktākus rādītājus, kad VAD ir ieslēgts. Savukārt algoritmus E, AD2 un AD3 VAD gandrīz neietekmē.



4.3. att. **Pakešu zudumi pie tīkla sadalījuma 97/3/0 ar VAD/DTX**



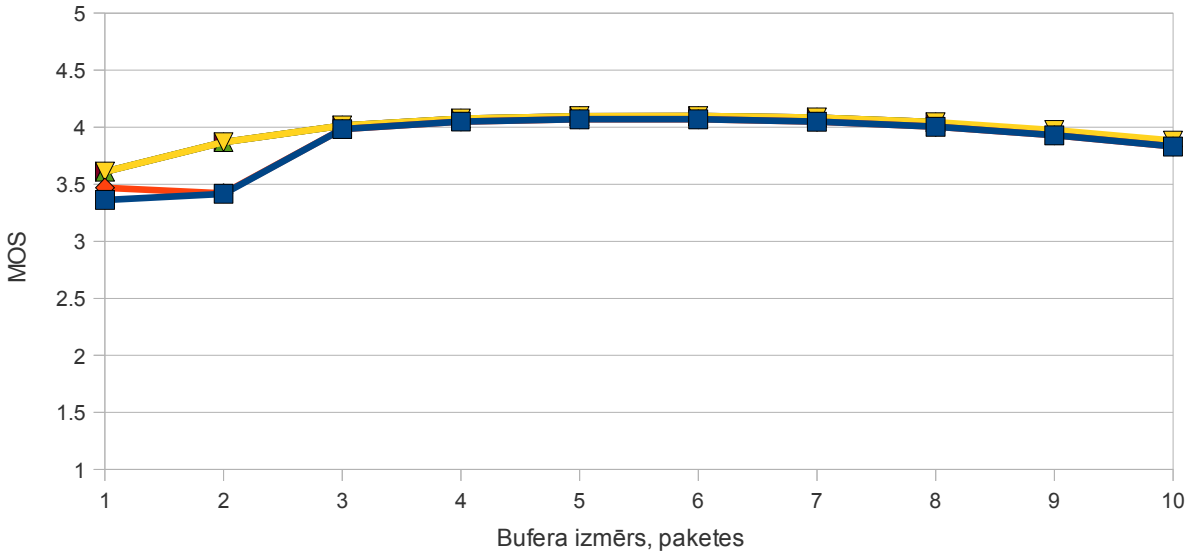
4.4. att. **Pakešu zudumi pie tīkla sadalījuma 97/3/0 bez VAD/DTX**

Ja apskata datus tuvāk – 4.1 tabulā, kur atlasīts pakešu zudums algoritmiem B, CF un E – tad var saskatīt, ka **visi trīs algoritmi darbojas labāk bez VAD**, taču īpaši izteikti tas ir B un CF algoritmiem, kuru darba kvalitāte, palielinoties džiter-bufera izmēram, krītas.

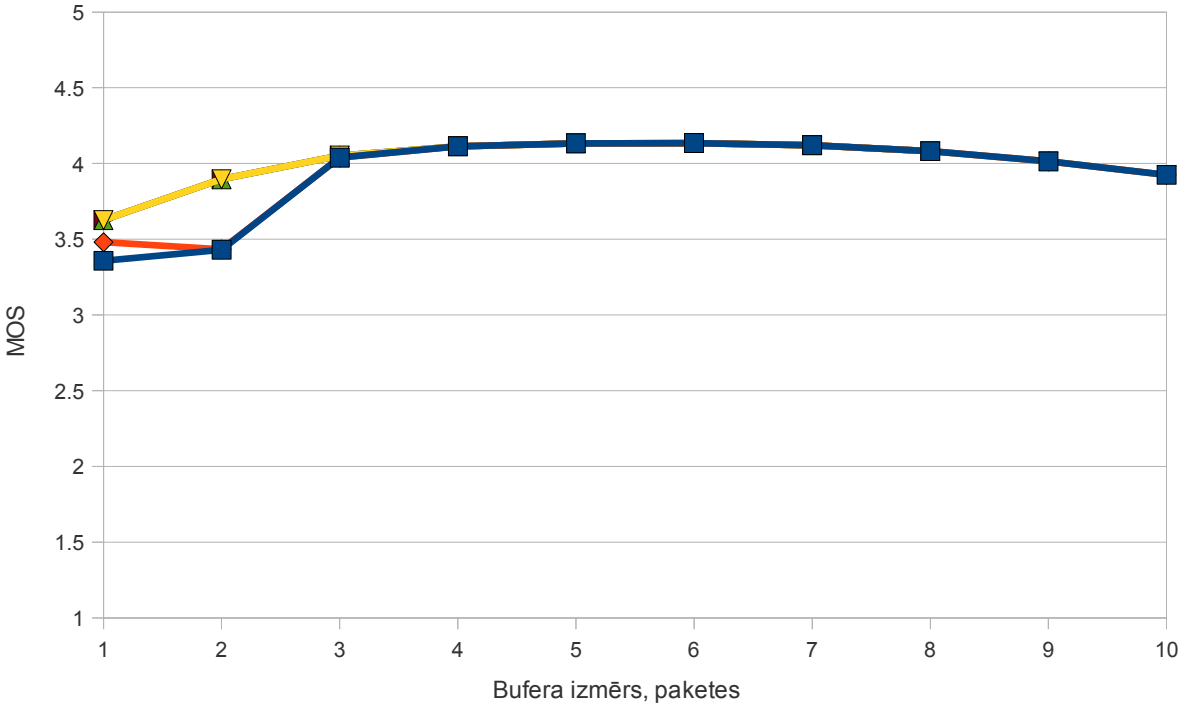
4.1 tabula

Pakešu zuduma salīdzinājums ar un bez VAD algoritmiem B, CF un E

Bufera izmērs	*ar VAD/DTX*			*bez VAD/DTX*		
	B	*CF*	*E*	*B*	*CF*	*E*
1	4.83%	4.08%	3.14%	5.16%	4.28%	3.29%
2	4.45%	4.42%	1.48%	4.64%	4.62%	1.55%
3	0.76%	0.76%	0.57%	0.65%	0.65%	0.56%
4	0.35%	0.35%	0.19%	0.16%	0.16%	0.16%
5	0.22%	0.22%	0.05%	0.04%	0.04%	0.04%
6	0.20%	0.20%	0.02%	0.01%	0.01%	0.01%
7	0.23%	0.23%	0.00%	0.00%	0.00%	0.00%
8	0.25%	0.25%	0.00%	0.00%	0.00%	0.00%
9	0.26%	0.26%	0.00%	0.00%	0.00%	0.00%
10	0.28%	0.28%	0.00%	0.00%	0.00%	0.00%



4.5. att. **MOS pie tīkla sadalījuma 97/3/0 ar VAD/DTX**



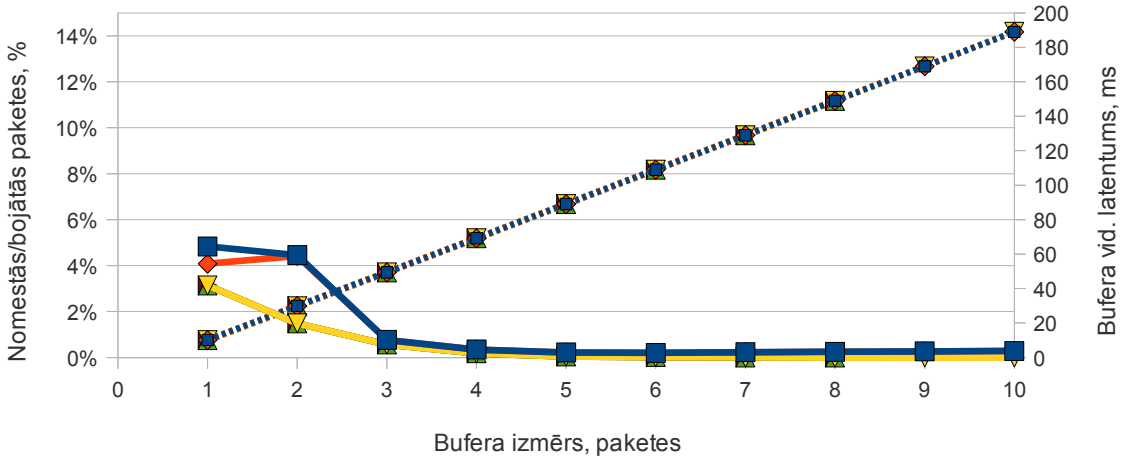
4.6. att. **MOS pie tīkla sadalījuma 97/3/0 bez VAD/DTX**

MOS rezultātos (attēli 4.5. un 4.6.) būtiska atšķirība nav manāma.

Rezumējot var acīmredzami secināt, ka **algoritmi B un CF nav optimizēti darbam ar VAD/DTX**. Iemesls tam varētu būt tas, ka VAD/DTX dēļ salīdzinoši bieži tiek iztukšots džiter-buferis, bet kā jau aprakstīts iepriekšējā nodaļā, šie algoritmi tukšā buferī paketi vienmēr novieto vistālāk no kodeka. Līdz ar to – jo lielāks buferis – jo lielāks negatīvais efekts.

4.3. Algoritmu salīdzinājums pie dažādas kvalitātes tīkla

Lai savstarpēji salīdzinātu visus 5 algoritmus – B, CF, E, AD2 un AD3 – šajā apakšnodaļā tika simulēti dažādas uzvedības tīkli – no ļoti labiem līdz pat nekvalitatīviem.

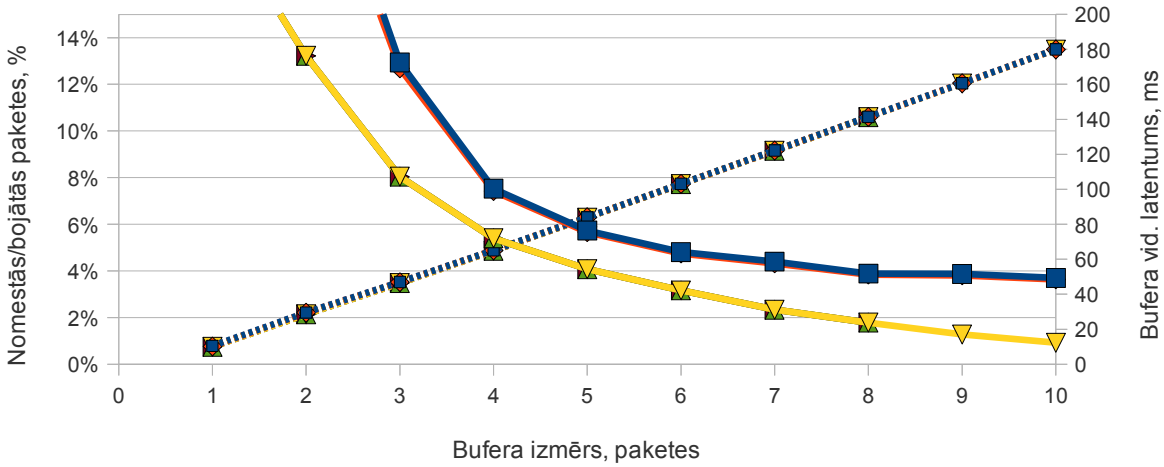


4.7. att. **Kvalitātes parametri pie tīkla sadalījuma 97/3/0**

Attēlā 4.7. bufera vidējā latentuma funkcija visiem pieciem algoritmiem ir identiska.

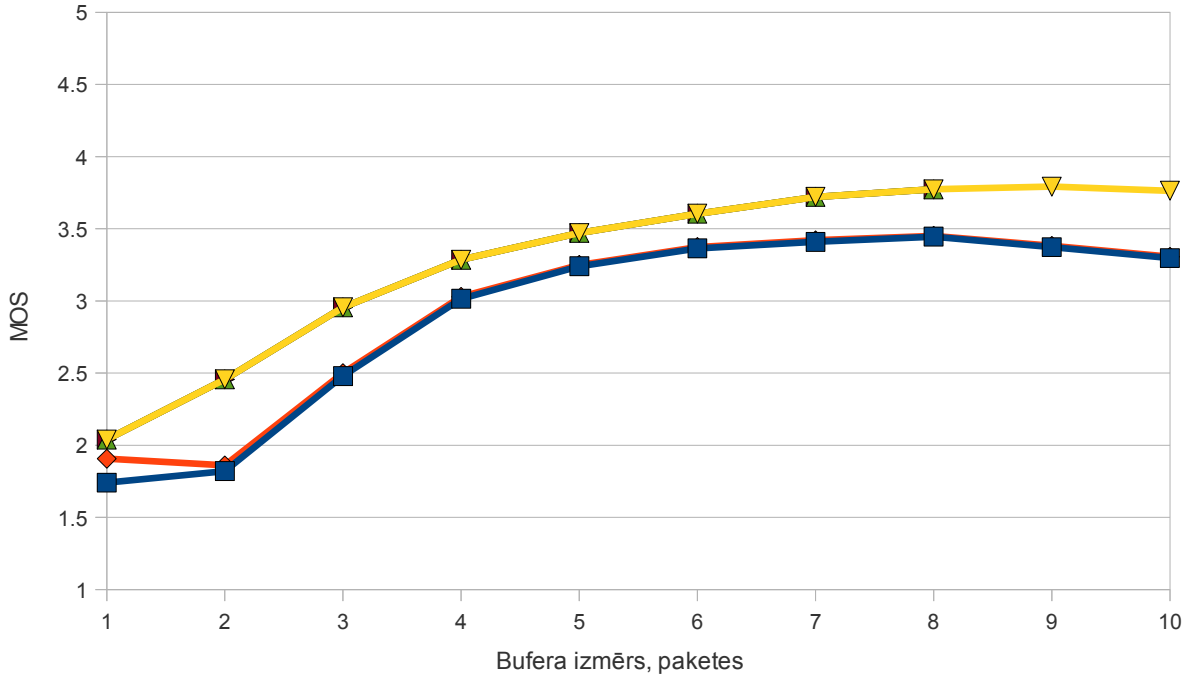
Pie tīkla sadalījuma 97/3/0 (4.5. un 4.7. attēli) visiem algoritmiem ir zemi pakešu zudumi (sākot no 3 pakešu liela bufera – mazāk par 1%). Ja bufera izmērs ir no 4 līdz 7 paketēm, tad **visi algoritmi MOS skalā nodrošina labu runas kvalitāti** (t.i. vismaz 4.0).

Taču neatkarīgi no bufera izmēra, ir iespējams novērot šādu algoritmu darba rezultātu sakarību: **AD2 = AD3 = E > CF > B.**



4.8. att. **Kvalitātes parametri pie tīkla sadalījuma 80/15/5**

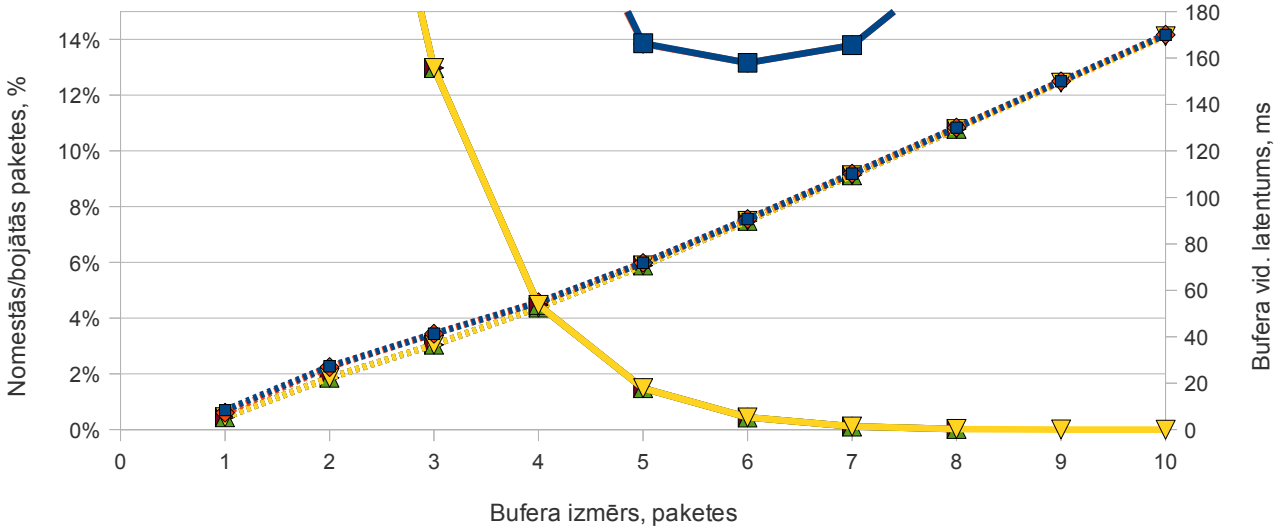
Attēlā 4.8. bufera vidējā latentuma funkcija visiem pieciem algoritmiem ir identiska.



4.9. att. **MOS pie tīkla sadalījuma 80/15/5**

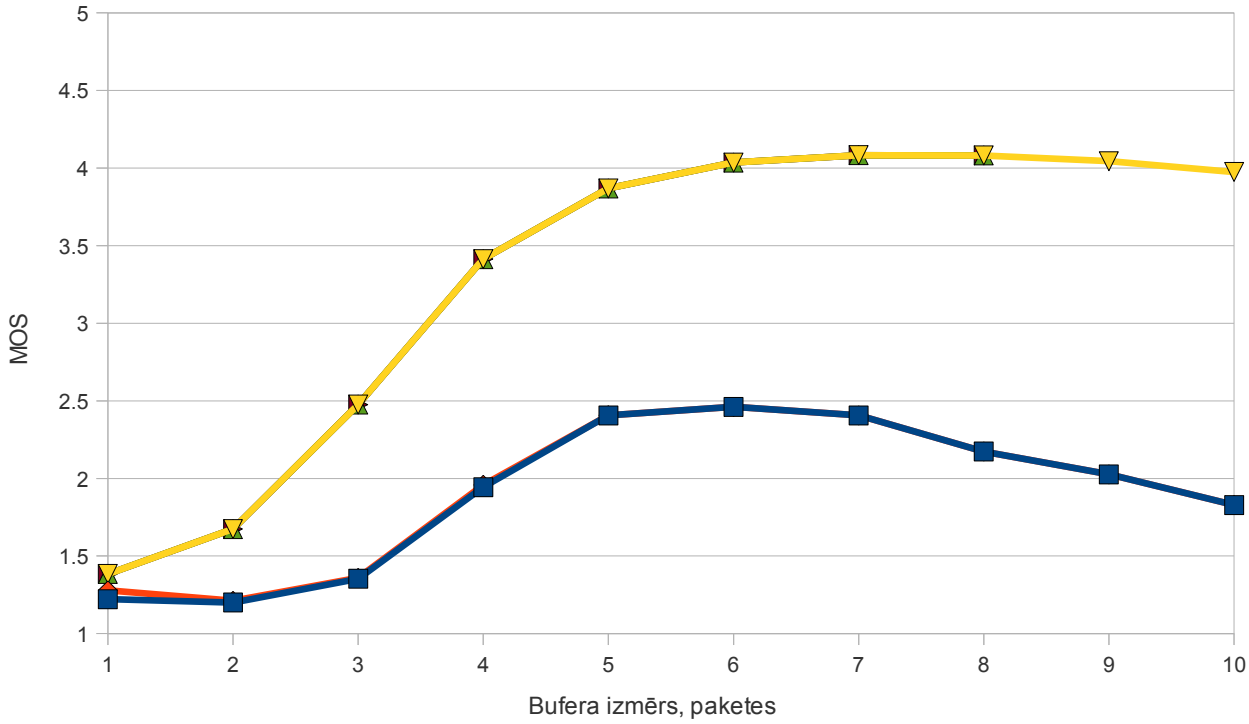
Ja tīkla sadalījums ir 80/15/5, tad visi algoritmi jau izjūt efektivitātes samazinājumu. Turklāt uzskatāmi kļūst redzams, ka CF ir nedaudz labāks par B, bet E ir būtiski labāks par CF. Piemēram, attēlā 4.8. var saskatīt, ka aptuveni 13% pakešu pazūd tad, kad CF un B algoritmu džiter-bufera izmērs ir 3 paketes, bet E šādu pašu rezultātu var sasniegt ar 2 pakešu lielu džiter-buferi.

Attēlā 4.9. redzams, ka algoritms E brīžam sasniedz MOS atzīmi 3.8, bet algoritmi CF un E nesasniedz pat 3.5. Jāpiebilst, ka pie džiter-bufera izmēra 7 vai 8 paketes, visi algoritmi varētu sniegt gandrīz labu kvalitāti.



4.10. att. **Kvalitātes parametri pie tīkla sadalījuma 60/40/0**

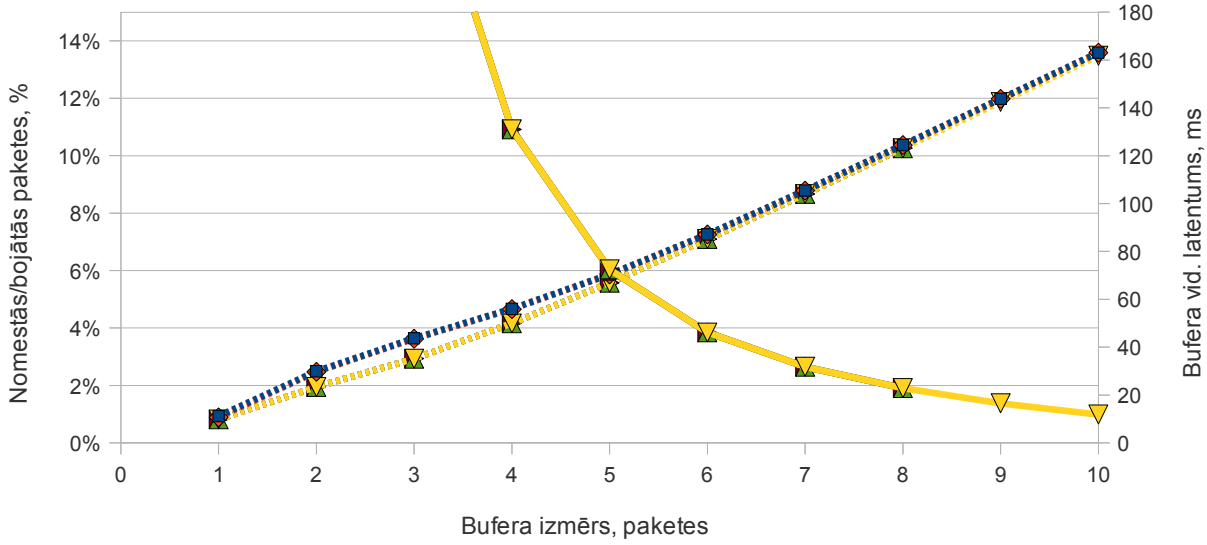
Attēlā 4.10. algoritmos B un CF ir **izteikti redzams apakšnodaļā 4.2. aprakstītais efekts.** Tā dēļ veidojas U-veida līkne.



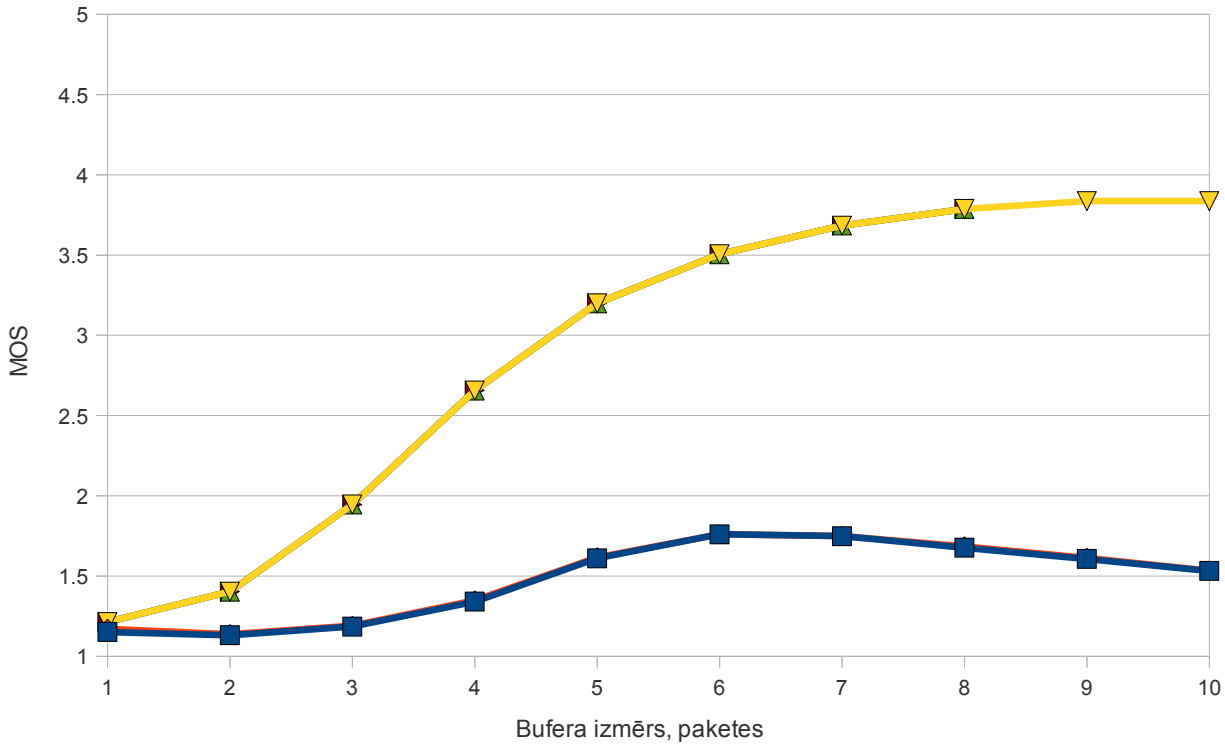
4.11. att. **MOS pie tīkla sadalījuma 60/40/0**

Pie sadalījuma 60/40/0, kur 60% tiek pārsūtīti pa labu ceļu, bet pārējie 40% pa vidēji ātru, algoritmi B un CF sniedz diezgan nožēlojamu rezultātu. Lai gan pie maza bufera izmēra B un CF spēj nodrošināt nedaudz mazāku latentumu, apskatot MOS skalu (attēls 4.11.) var skaidri nolasīt jau citos tīkla sadalījumos pamanīto sakarību: **E = AD2 = AD3 > CF > B**.

Šoreiz gan CF ir pārākums pār B ir nedaudz izteiktāks.

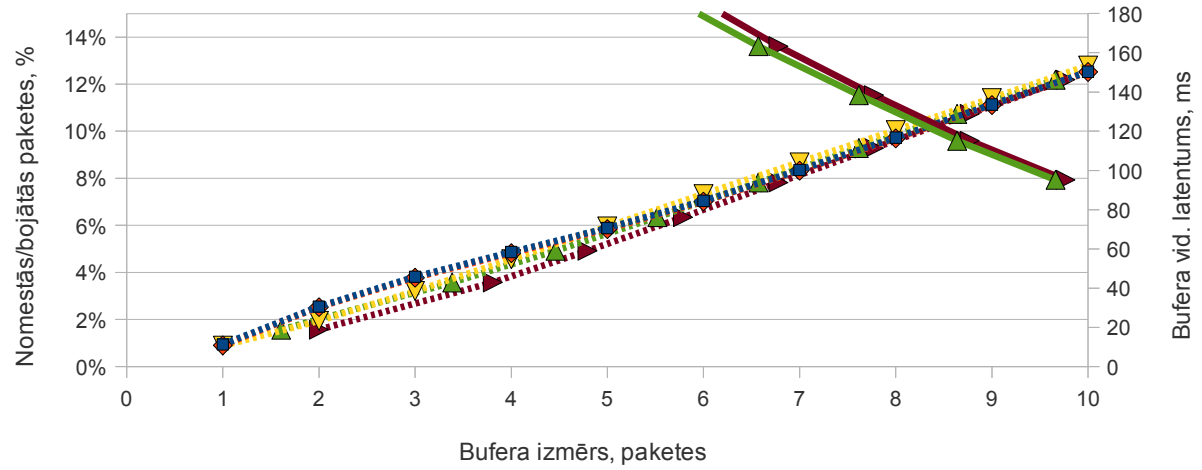


4.12. att. **Kvalitātes parametri pie tīkla sadalījuma 40/55/5**



4.13. att. **MOS pie tīkla sadalījuma 40/55/5**

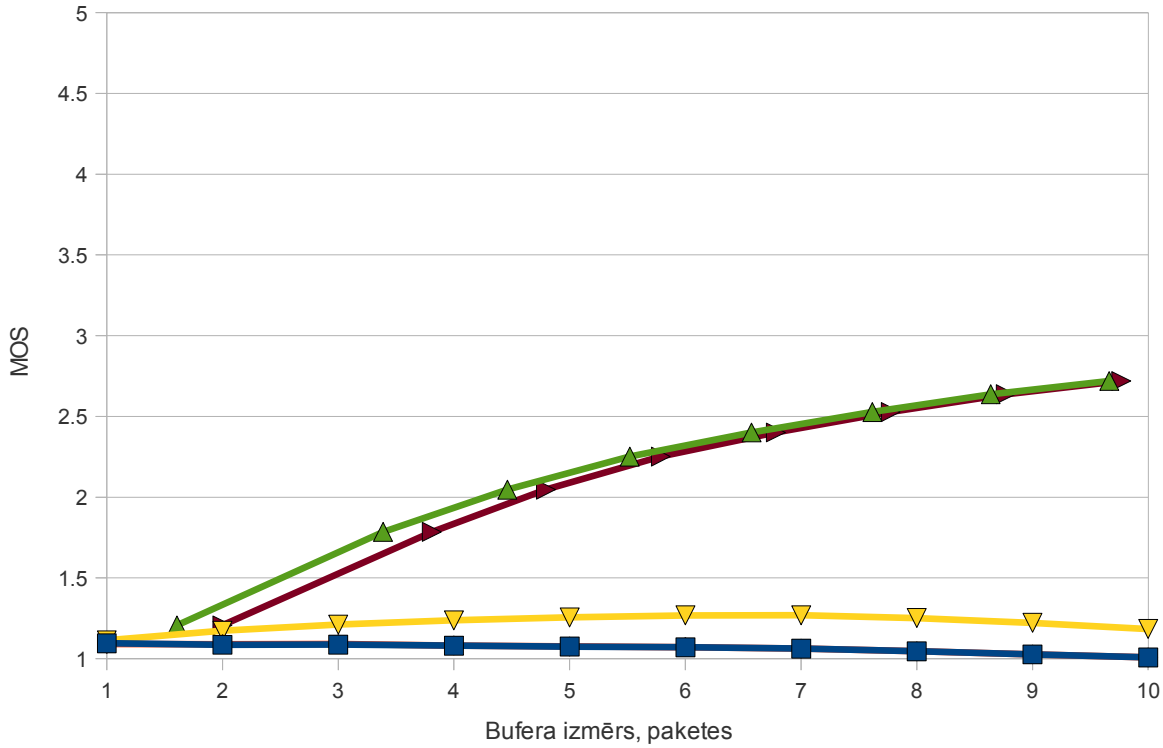
Sadalījums 40/55/5 (attēli 4.12. un 4.13.) pēc simulācijas rezultātiem ne pārāk atšķiras no sadalījuma 60/40/0 (attēli 4.10. un 4.11.) Izskatās, it kā līknēm ir pievienots koeficients, kas mazāks par 1, un tagad **labāko MOS sasniedz algoritms E pie džiter-bufera izmēra 9 paketes**. Iepriekš pie tīkla sadalījuma 60/40/0 tie bija algoritmi E, AD2, AD3 pie džiter-bufera izmēra 7 paketes.



4.14. att. **Kvalitātes parametri pie tīkla sadalījuma 20/60/20-1**

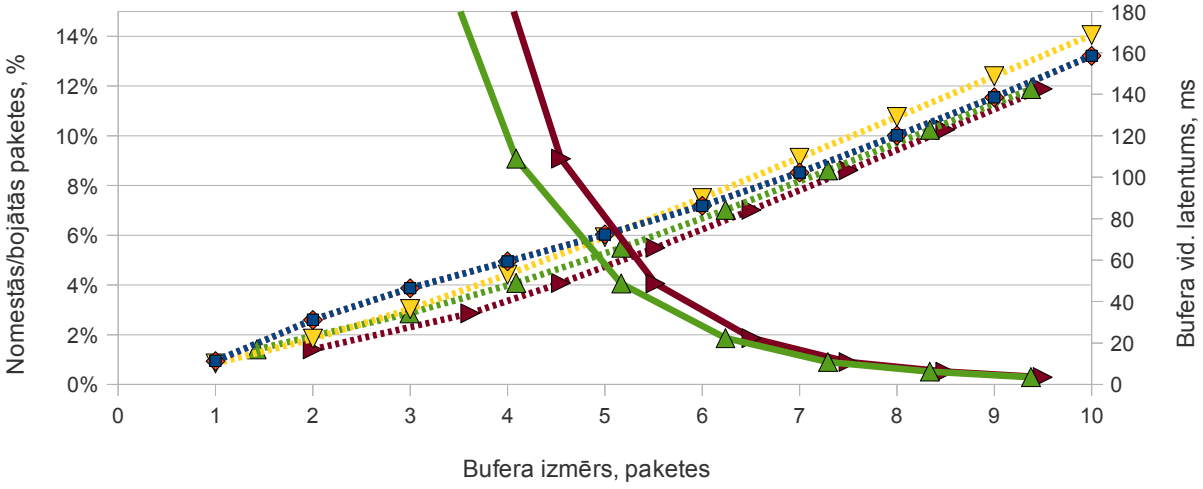
Tīkla sadalījums 20/60/20-1 nozīmē, ka 60% pakešu tiek sūtītas pa vidējā ātruma ceļu, bet pārēji 40% vienmērīgi sadalīti pa ātro un lēnu ceļu. Turklāt džiter-buferi (un tātad arī kodeku) pirmā sasniegs pakete, kas pārvietojās tieši pa vidējā ātruma ceļu.

Kā redzams 4.14 attēlā šoreiz ne tikai B un CF, bet arī E algoritms stipri atpaliek no AD2 un AD3. Turklāt tagad AD2 un AD3 neuzrāda vienādus rezultātus.



4.15. att. **MOS pie tīkla sadalījuma 20/60/20-1**

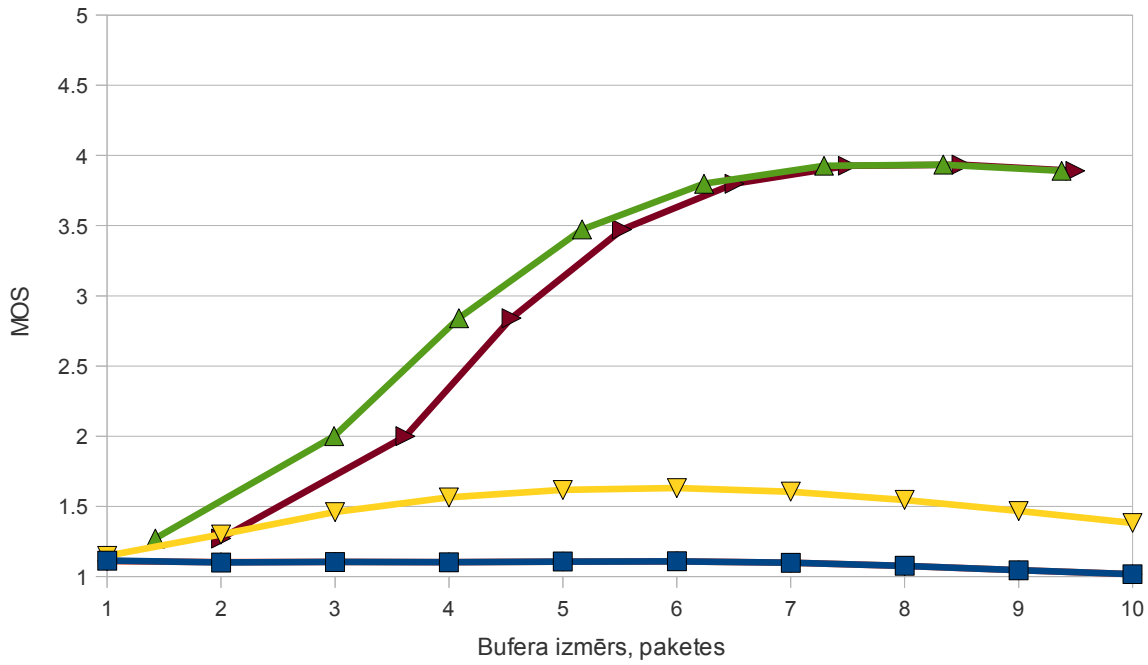
Vērtējot pēc MOS (4.15. attēls) var izsecināt šādu sakarību: **AD2 > AD3 > E > CF = B**. Taču šoreiz **runas kvalitāte pat ar 9 vai 10 pakešu garu buferi** ir tikai nedaudz **zem ciešamas** (apmēram 2.7).



4.16. att. **Kvalitātes parametri pie tīkla sadalījuma 10/89/1-1**

Tīkla sadalījumā 10/89/1-1 atšķirību starp AD2 un AD3 var redzēt vēl asāk. Attēlā 4.16. skaidri parādās vairāku procentu starpība pazudušajās paketēs, kad bufera izmērs ir 4 līdz 5 paketes.

Džiter-buferi (un tātad arī kodeku) pirmā sasniegs pakete, kas pārvietojās pa vidējā ātruma ceļu.

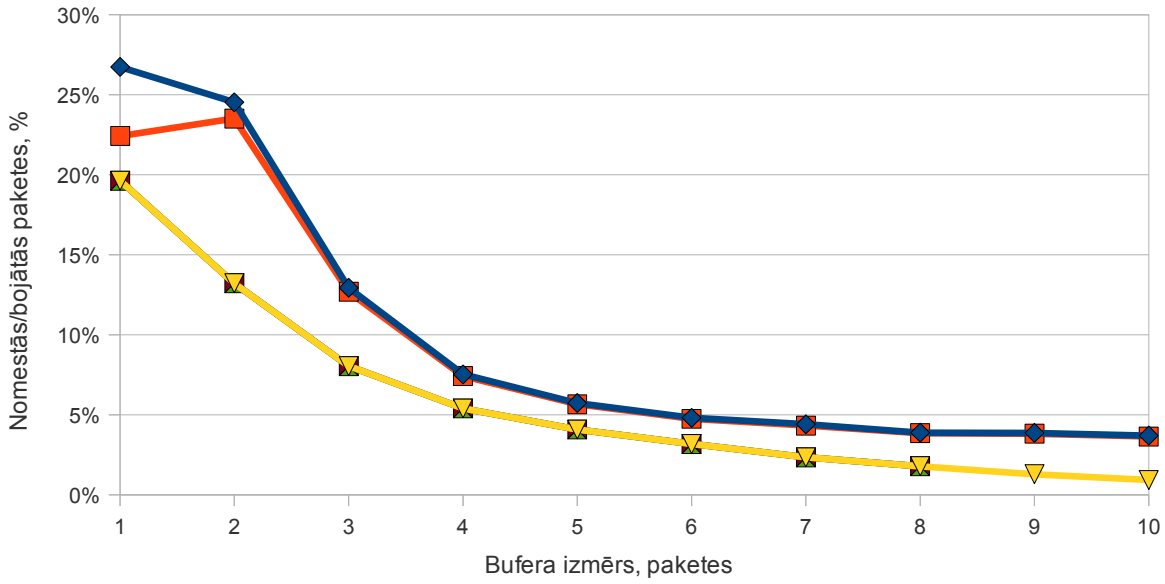


4.17. att. **MOS pie tīkla sadalījuma 10/89/1-1**

MOS rezultāti (4.17. attēls) sadalījumam 10/89/1-1 ir iepriecinoši. Lai gan džiter-bufera vadības algoritmi B, CF un E uzrāda sliktus un ļoti sliktus rezultātus, a**lgoritmi AD2 un AD3 sasniedz gandrīz labu runas kvalitāti** pie bufera izmēra 7 paketes. MOS līkne algoritmam CF sakrīt ar līkni algoritmam B.

4.4. Pirmās saņemtās paketes nozīmīgums

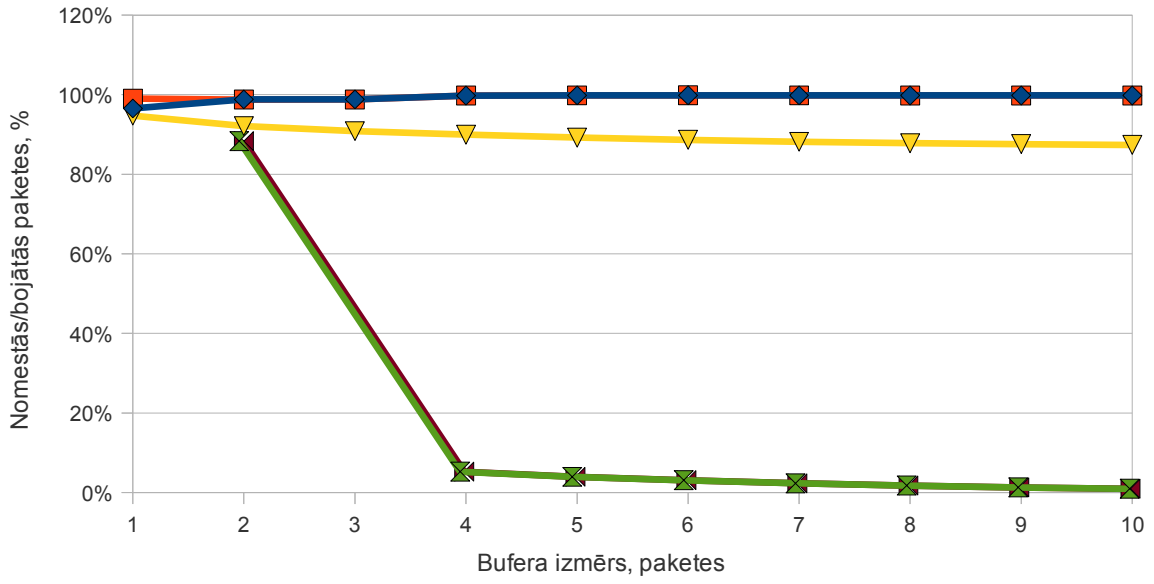
Tika veiktas trīs simulācijas ar vienu un to pašu sadalījumu – 80/15/0. Atšķirīgais faktors bija ceļš, pa kuru tiek izsūtītas pirmās paketes. Tīkla sadalījumam paliekot nemainīgam, vienā gadījumā pirmās paketes tika sūtītas pa ātrāko ceļu, citā pa vidējo, bet vēl citā – pa lēnāko ceļu. Ar šo simulāciju palīdzību būtu jāspēj noteikt, tieši kāda ir pirmās saņemtās paketes loma džiter-bufera vadības algoritmā un kodekā.



4.18. att. **Pakešu zudumi pie tīkla sadalījuma 80/15/5-0**

Attēlā 4.18. ir redzams pakešu zudums gadījumā, ja pirmā pakete sasniedz džiter-buferi pa ātrāko ceļu. Pakešu zudumu funkcijas katram algoritmam var uzskatīt par bāzlīniju šīs apakšnodaļas kontekstā.

Sākot no bufera izmēra 6 pakešu lielumā, visiem algoritmiem pakešu zudums nav lielāks par 5%.



4.19. att. **Pakešu zudumi pie tīkla sadalījuma 80/15/5-1**

Interesants fenomens ir novērojams (4.19. un 4.20. attēls), ja pirmā pakete sasniedz džiter-buferi pa vidējo ceļu. Lai gan algoritmi B un CF pazaudē gandrīz 100% pakešu un

algoritms E pazaudē vairāk kā 80%, **tomēr AD2 un AD3 pazaudē mazāk par 10%**, ja vien džiter-bufera faktiskais izmērs ir vismaz 4 paketes. Nevar apstrīdēt, ka **tas ir būtisks uzlabojums.**

Piemēram, fiksētais džiter-buferis (algoritms E) ar izmēru 4 paketes nepieņems 6. paketi (4.2. tabula), taču adaptīvais džiter-buferis (algoritmi AD2, AD3), kura maksimālais izmērs nepārsniedz 4 paketes, varēs pieņemt šādu paketi (4.3. tabula).

4.2. tabula

Piemērs: Algoritms E ar bufera izmēru 4 paketes

Laiks (ms)	*Tīkls →*	*→ Buferis →*	*→ Kodeks*
0	6 3 2 1	▒ ▒ ▒ ▒	
	6 3 2	1 ▒ ▒ ▒	
20	6 3 2	▒ 1 ▒ ▒	
	6 3	2 1 ▒ ▒	
40	6 3	▒ 2 1 ▒	
	6	3 2 1 ▒	
60	6	▒ 3 2 1	
		▒ 3 2 1	
80		▒ ▒ 3 2	1
100		▒ ▒ ▒ 3	2 1
120		▒ ▒ ▒ ▒	3 2 1

Kā redzams 4.2. tabulā, **laikā starp 60ms un 80ms algoritms E nevar pieņemt 6. paketi**, jo vienīgā brīvā vieta ir rezervēta 4. paketei.

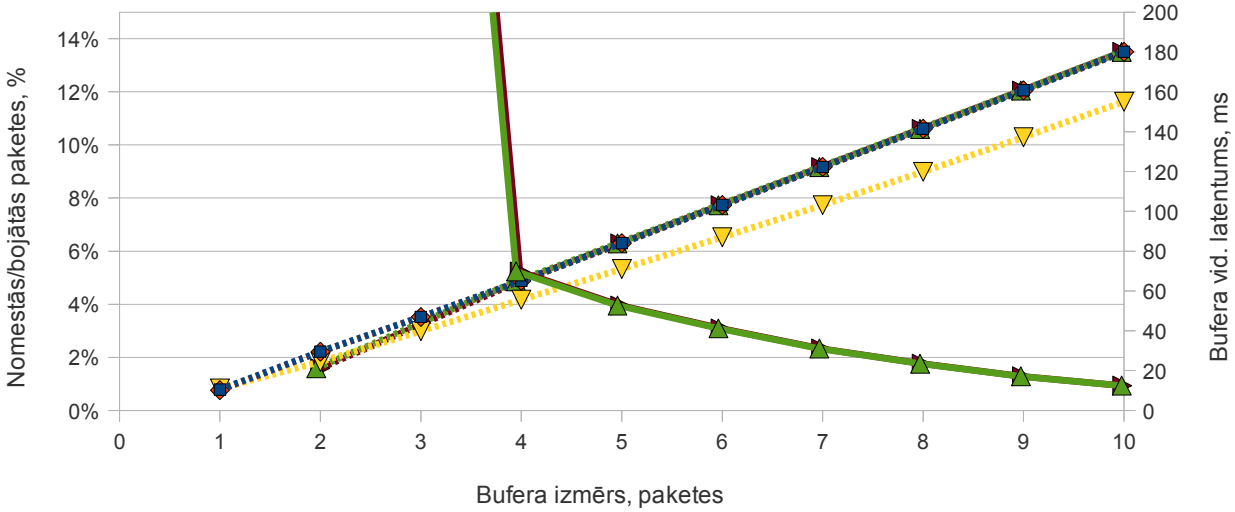
4.3. tabula

Piemērs: Algoritms AD2/AD3 ar maksimālo bufera izmēru 4 paketes

Laiks (ms)	*Tīkls →*	*→ Buferis →*	*→ Kodeks*
0	6 3 2 1	▒ ▒	
	6 3 2	1 ▒	
20	6 3 2	▒ 1	
	6 3	2 1	
40	6 3	▒ 2	1
	6	3 2	1
60	6	▒ 3	2 1
		6 ▒ ▒ 3	2 1
80		▒ 6 ▒ ▒	3 2 1

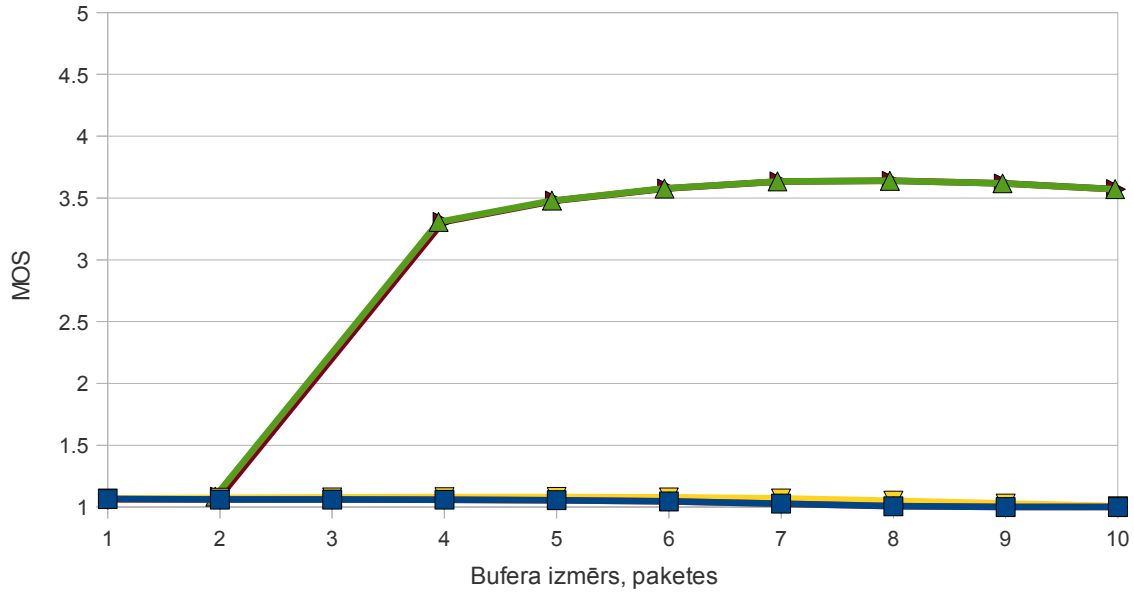
100		▒ ▒ 6 ▒	▒ 3 2 1
120		▒ ▒ ▒ 6	▒ ▒ 3 2 1
140		▒ ▒ ▒ ▒	6 ▒ ▒ 3 2 1

Adaptīvais džiter-buferis sākotnēji ir mazs un tad pēc vajadzības izplešas. Tabulā 4.3. var redzēt kā uz identisku situāciju tīklā reaģē algoritmi AD2 un AD3, kuru maksimālais bufera izmērs ir 4 paketes, bet sākotnējais izmērs ir 2 paketes. Tie **var pieņemt 6. paketi** laikā starp 60ms un 80ms, jo pirms tam buferī tika glabātas tikai paketes ar numuriem 2 un 3, turklāt 2. tikko ir izsūtīta. Līdz ar to buferis varēs nekavējoties "izaugt" līdz 4 pakešu lielam izmēram un pieņemt paketi ar numuru 6 bufera tālākajā vietā.



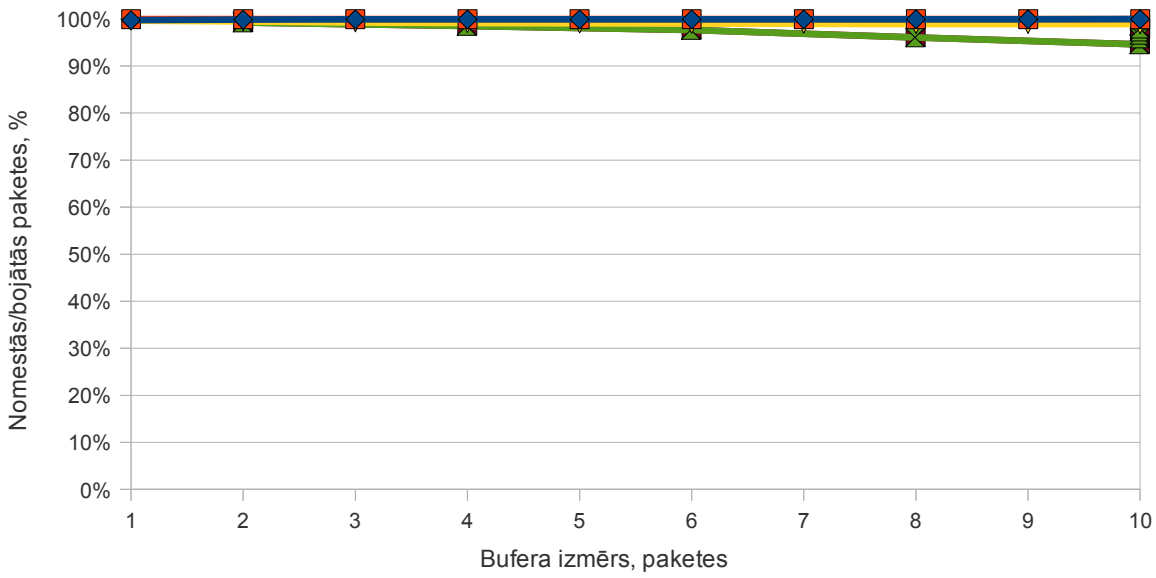
4.20. att. **Kvalitātes parametri pie tīkla sadalījuma 80/15/5-1**

Attēlā 4.20. algoritmu CF un B latentuma līknes sakrīt.



4.21. att. **MOS pie tīkla sadalījuma 80/15/5-1**

Pateicoties augstāk aprakstītajai īpatnībai, autora izstrādātie adaptīvie algoritmi AD2 un AD3 spēj noturēt runas kvalitāti (4.21. attēls) gandrīz labā līmenī (virs 3.5), kamēr visi pārējie algoritmi – B, CF un E – nodrošina sliktu kvalitāti (1.0).



4.22. att. **Pakešu zudumi pie tīkla sadalījuma 80/15/5-2**

Ja pirmā pakete sasniedz džiter-buferi pa lēnāko ceļu, tad gan džiter-buferis, gan kodeks būs noskaņojušies uz lielu aizkavi. Tāpēc tīkla sadalījumā 80/15/5-2 var labākā gadījumā cerēt uz apmēram 95% pazaudētu pakešu. Taču pie 5% varbūtības katrai paketei nonākt uz lēnākā ceļa, **varbūtība**, ka neviena cita pakete nepaspēs to apdzīt un pirmā nonākt līdz džiter-buferim, **ir neliela**.

5. SECINĀJUMI

No literatūras izpētes seko, ka lielākā daļa adaptīvo algoritmu veic laika skalas modifikācijas un līdz ar to nav pielietojami universāli un neatkarīgi no kodeka.

Tika izstrādāti 3 jauni džiter-bufera vadības algoritmi (skatīt 3. nodaļu):

- algoritms E, kas seko līdzi tam, kurā brīdi katra pakete ir nepieciešama kodekam un, balstoties uz to, attiecīgi izvieto ienākošās paketes;
- adaptīvs algoritms AD2, kas palielina džiter-buferi tiklīdz tas nepieciešams, bet samazina pēc noteikta laika intervāla;
- adaptīvs algoritms AD3, kas palielina džiter-buferi tiklīdz tas nepieciešams, bet samazina, kad džiter-buferis ir veiksmīgi pieņēmis vairāk kā 10 paketes un noraidījis mazāk pakešu nekā pieņēmis.

Savstarpēji salīdzinot algoritmu darbību, var secināt, ka jebkādos tīkla apstākļos vērtējot pēc algoritma nodrošinātās sakaru kvalitātes: **AD2 ≥ AD3 ≥ E > CF ≥ B**. Turklāt eksistē tādi apstākļi, kad **CF > B**, kā arī tādi apstākļi, kad **AD2 > AD3 > E**.

Tātad **nulles hipotēze ir apgāzta, bet alternatīvā hipotēze – pierādīta**: vismaz viens (visi) no autora izstrādātajiem algoritmiem (E, AD2, AD3) pie kādas no tīkla noslodzēm (jebkuras no darbā apskatītajām) pēc darbības kvalitātes būtiski pārspēja algoritmu CF.

Svarīgs sasniegums ir tādu džiter-bufera vadības algoritmu piedāvāšana, kas spēj nodrošināt augstas kvalitātes rezultātus situācijās, kad sarunas sākumā paketes pienāk lēnāk, taču vēlāk piegādes ātrums paaugstinās. Algoritmi AD2 un AD3 uzrāda būtiskus uzlabojumus pār algoritmu E tad, ja pakete, kas pirmā sasniedz džiter-buferi, ir pārvietojusies pa vidēji ātru ceļu. Tas ir saistīts ar to, ka šajā gadījumā džiter-buferis sinhronizējas laikā pret vidējā ātruma datu plūsmu un nākamajām paketēm pastāv pietiekami liela varbūtība ierasties džiter-buferī ātrāk par nepieciešamo laiku. Ja džiter-buferis sinhronizētos laikā pret ātrāko datu plūsmu, tad pastāvētu varbūtība saņemt paketi vēlāk nekā vajag, bet noteikti ne ātrāk.

Tā kā VAD/DTX dēļ bieži tiek iztukšots galiekārtas džiter-buferis, tad var secināt, ka VAD/DTX atstāj negatīvu iespaidu uz balss pārraides kvalitāti. Šāds secinājums gan izdarīts ar pieņēmumu, ka joslas platums nav ierobežots. Jāsecina, ka džiter-bufera vadības algoritmi B un CF nav optimizēti VAD/DTX lietojumiem, un, jo lielāks to džiter-bufera izmērs, jo lielāka ir VAD/DTX negatīvā ietekme. Tam par iemeslu kalpo šo algoritmu īpatnība – tukšā buferī ienākošu paketi tie vienmēr novietos vistālāk no kodeka.

PATEICĪBAS

Vēlos izteikt pateicību cilvēkiem bez kuriem šī darba tapšanas process būtu bijis ilgāks un arī darba rezultāti – mazāk spoži.

Paldies maniem kolēģiem par to, ka darbā spēja vairākas nedēļas iztikt (gandrīz) bez manis! Es saprotu, ka tas nebija pārlieku vienkārši. Īpašs paldies manam priekšniekam John Urness un viņa kolēģiem par akadēmiskā atvaļinājuma piešķiršanu un manu aizvietošanu prombūtnes laikā.

Saku paldies arī saviem draugiem, kas izpalīdzēja ar tehnisko nodrošinājumu un kādu padomu!

IZMANTOTĀ LITERATŪRA UN AVOTI

1. *Gaisa kuģu ekspluatācija, 6. pielikums, III daļa, 6. izdevums*. **ICAO.** (Valsts Valodas centra tulkojums.)
2. The Course Glossary for IP Telephony v1.0, **Cisco Systems**, 2005.
3. *Letter symbols to be used in electrical technology: Telecommunications and electronics.* **IEC 60027-2.**
4. *Cisco Networking Academy Program: IP Telephony v1.0*, **Cisco Systems,** 2005.
5. **Бройтман, М., Матанцев, С.** Определение оптимального размера джиттер-буфера приемного маршрутизатора. *Автоматика и вычислительная техника*, 2008, No. 3, стр. 33-39.
6. CISCO IOS Software Releases 12.1T [tiešsaiste] – [atsauce 10.05.2011.] Pieejams: http://www.cisco.com/en/US/docs/ios/12_1t/12_1t5/feature/guide/dt_pod.html
7. Arhivēts fails fixedjitterbuf.c [tiešsaiste] – [atsauce 15.05.2011.] Pieejams: http://downloads.asterisk.org/pub/telephony/asterisk/asterisk-1.8.4.tar.gz
8. Asterisk Downloads [tiešsaiste] – [atsauce 23.05.2011.] Pieejams: http://www.asterisk.org/downloads
9. **Broitmans, M.** VoIP trafika buferizācijas jaunie algoritmi. ***No:*** *LU 66. zinātniskā konference, Datortīklu sekcija*. Rīga, 2008.
10. Full Episode Player - South Park Studios [tiešsaiste] – [atsauce 04.05.2011.] Pieejams: http://www.southparkstudios.com/episodes/random.php
11. Comparison of VoIP software [tiešsaiste] – [atsauce 04.05.2011.] Pieejams: http://en.wikipedia.org/wiki/Comparison_of_VoIP_software
12. Free Live Video Streaming, Online Broadcasts [tiešsaiste] – [atsauce 04.05.2011.] Pieejams: http://www.ustream.tv/
13. Understanding Jitter in Packet Voice Networks (Cisco IOS Platforms) [tiešsaiste] – [atsauce 12.05.2011.] Pieejams: http://www.cisco.com/en/US/tech/tk652/tk698/technologies_tech_note09186a00800945df.shtml
14. *IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)*. **RFC 3393.**

15. Indepth Articles: Jitter [tiešsaiste] – [atsauce 10.05.2011.] Pieejams: http://www.voiptroubleshooter.com/indepth/jittersources.html
16. Digital Speech Algorithms - Adaptive Jitter Buffer [tiešsaiste] – [atsauce 12.05.2011.] Pieejams: http://dspalg.com/jitter_buffer.html
17. Understanding Delay in Packet Voice Networks [tiešsaiste] – [atsauce 11.04.2011.] Pieejams: http://www.cisco.com/en/US/tech/tk652/tk698/technologies_white_paper09186a00800a8993.shtml
18. G.729: multi-purpose ITU-T standard for increased bandwidth availability and short delay [tiešsaiste] – [atsauce 30.04.2011.] Pieejams: http://www.sipro.com/g729_about.php
19. *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP).* **ITU-T G.729.**
20. *Transmission impairments due to speech processing.* **ITU-T G.113 amend. 2.**
21. **Cole, R. G., Rosenbluth, J. H.** *Voice over IP performance monitoring.* Middletown: AT&T Laboratories, 2001. 16 p.
22. **Froehlich, A.** *CCNA Voice Study Guide.* Hoboken: Sybex, 2010. 648 p.
23. **Narbutt, M., Murphy L.** *VoIP playout buffer adjustment using adaptive estimation of network delays.* Dublin: University College Dublin, 2003. 10 p.
24. **Qiao, Z., Venkatasubramanian, R. K., Sun, L., Ifeachor, E. C.** *A New Buffer Algorithm for Speech Quality Improvement in VoIP Systems.* Plymouth : University of Plymouth, 2007. 19 p.
25. *Methods for subjective determination of transmission quality.* **ITU-T P.800.**
26. *The E-model, a computational model for use in transmission planning.* **ITU-T G.107.**
27. **Yamamoto, L. A. R., Beerends, J. G.** *Impact of network performance parameters on the end-to-end perceived speech quality.* Leidschendam : KPN Research, 1997. 11 p.
28. *Methodology for derivation of equipment impairment factors from subjective listening-only tests.* **ITU-T P.833.**
29. **Sun, L., Ifeachor, E. C.** Voice Quality Prediction Models and Their Application in VoIP Networks. *IEEE Transactions on Multimedia*, 2006, no. 4, vol. 8, p. 809-820

30. GPSS World Downloads [tiešsaiste] – [atsauce 08.04.2010.] Pieejams: http://minutemansoftware.com/downloads.asp
31. GPSS World en Ubuntu [tiešsaiste] – [atsauce 08.04.2010.] Pieejams: http://daianamurgan.blogspot.com/2008/04/gpss-world-en-ubuntu.html
32. OMNeT++ 4.1 (source + IDE, tgz) [tiešsaiste] – [atsauce 27.01.2011.] Pieejams: http://omnetpp.org/download/release/omnetpp-4.1-src.tgz
33. How to Install Omnet++ in Ubuntu 8.04 [tiešsaiste] – [atsauce 27.01.2011.] Pieejams: http://www.arejae.com/blogv2/how-to-install-omnet-in-ubuntu-804.html
34. **Daniel, E. J., White, C. M., Teague, K. A.** *An Inter-arrival Delay Jitter Model using Multi-Structure Network Delay Characteristics for Packet Networks.* Stillwater: Oklahoma State University, 2003. 5 p.
35. **Dahmouni, H., Girard, A., Sansò, B.** *An analytical model for jitter in IP networks.* Rabat: Institut National des Postes et Télécommunication, 2011. 10 p.
36. **Abate, J., Choudhury, G. L., Whitt, W.** *Exponential approximations for tail probabilities in queues, I: waiting times.* New Jersey: AT&T Bell Laboratories, 1993. 17 p.
37. Submarine cable map 2011 [tiešsaiste] – [atsauce 03.05.2011.] Pieejams: http://www.telegeography.com/assets/website/images/maps/submarine-cable-map-2011/submarine-cable-map-2011-x.jpg
38. OMNeT++ version 4.1 User Manual [tiešsaiste] – [atsauce 05.02.2011.] Pieejams: http://www.omnetpp.org/doc/omnetpp/manual/usman.html
39. **Seger, J.** *Modelling Approach for VoIP Traffic Aggregations for Transferring Tele-traffic Trunks in a QoS enabled IP-Backbone Environment* . Dortmund : University of Dortmund , 2003. 4 p.
40. **Brun, O., Bockstal, C., Garcia, J. M.** *Analytical approximation of the jitter incurred by CBR traffics in IP networks.* Toulouse: LAAS-CNRS, 2006. 29 p.
41. **Sriram, K., Whitt, W.** Characterizing Superposition Arrival Processes in Packet Multiplexers for Voice and Data. *IEEE Journal on Selected Areas in Communications*, 1986, no. 6, vol. SAC-4, p. 833-846.

PIELIKUMU SARAKSTS

Maģistra darbam ir sekojoši pielikumi:

- Pielikums nr. 1: "Asterisk fiksētā džiter-bufera realizācija"
  - ↗ Pielikums satur vienu no Asterisk džiter-bufera realizācijām, kas brīvi pieejama un tiek izplatīta ar viņu programmatūru.(8)
- Pielikums nr. 2: "VoIP buferizācijas algoritma modelis"
  - ↗ Pielikumā ir GPSS simulācijas valodā veidots tīkla modelis. Tas satur arī algoritma CF etalona realizāciju (*reference implementation*).
  - ↗ Tīkla modeļa autors ir Mihails Broitmans.(9)
- Pielikums nr. 3: "Tīkla simulācijas modelis"
  - ↗ Šis pielikums satur galveno simulācijas failu, kas integrē visus tīkla simulācijas moduļus.
- Pielikums nr. 4: "Algoritmu implementācija"
  - ↗ Pielikumā atrodas faktiskās džiter-bufera vadības algoritmu B, CF, E, AD2 un AD3 realizācijas, kas rakstītas simulācijas videi *OMNeT++*.
  - ↗ Šajā pielikumā ir divi faili – deklarācijas fails un implementācijas fails.

PIELIKUMS NR. 1: "ASTERISK FIKSĒTĀ DŽITER-BUFERA REALIZĀCIJA"

Fails *fixedjitterbuf.c*(7):

```
/*
 * Copyright (C) 2005, Attractel OOD
 *
 * Contributors:
 * Slav Klenov <slav@securax.org>
 *
 * See http://www.asterisk.org for more information about
 * the Asterisk project. Please do not directly contact
 * any of the maintainers of this project for assistance;
 * the project provides a web site, mailing lists and IRC
 * channels for your use.
 *
 * This program is free software, distributed under the terms of
 * the GNU General Public License Version 2. See the LICENSE file
 * at the top of the source tree.
 *
 * A license has been granted to Digium (via disclaimer) for the use of
 * this code.
 */

/*! \file
 *
 * \brief Jitterbuffering algorithm.
 *
 * \author Slav Klenov <slav@securax.org>
 */

#include "asterisk.h"

ASTERISK_FILE_VERSION(__FILE__, "$Revision: 273641 $")

#include <assert.h>

#include "asterisk/utils.h"
#include "fixedjitterbuf.h"

#undef FIXED_JB_DEBUG

#ifdef FIXED_JB_DEBUG
#define ASSERT(a)
#else
#define ASSERT(a) assert(a)
#endif

/*! \brief private fixed_jb structure */
struct fixed_jb
{
       struct fixed_jb_frame *frames;
       struct fixed_jb_frame *tail;
       struct fixed_jb_conf conf;
       long rxcore;
       long delay;
       long next_delivery;
       int force_resynch;
};


static struct fixed_jb_frame *alloc_jb_frame(struct fixed_jb *jb);
```

```
static void release_jb_frame(struct fixed_jb *jb, struct fixed_jb_frame *frame);
static void get_jb_head(struct fixed_jb *jb, struct fixed_jb_frame *frame);
static int resynch_jb(struct fixed_jb *jb, void *data, long ms, long ts, long now);

static inline struct fixed_jb_frame *alloc_jb_frame(struct fixed_jb *jb)
{
	return ast_calloc(1, sizeof(*jb));
}

static inline void release_jb_frame(struct fixed_jb *jb, struct fixed_jb_frame *frame)
{
	ast_free(frame);
}

static void get_jb_head(struct fixed_jb *jb, struct fixed_jb_frame *frame)
{
	struct fixed_jb_frame *fr;

	/* unlink the frame */
	fr = jb->frames;
	jb->frames = fr->next;
	if (jb->frames) {
		jb->frames->prev = NULL;
	} else {
		/* the jb is empty - update tail */
		jb->tail = NULL;
	}

	/* update next */
	jb->next_delivery = fr->delivery + fr->ms;

	/* copy the destination */
	memcpy(frame, fr, sizeof(struct fixed_jb_frame));

	/* and release the frame */
	release_jb_frame(jb, fr);
}


struct fixed_jb *fixed_jb_new(struct fixed_jb_conf *conf)
{
	struct fixed_jb *jb;

	if (!(jb = ast_calloc(1, sizeof(*jb))))
		return NULL;

	/* First copy our config */
	memcpy(&jb->conf, conf, sizeof(struct fixed_jb_conf));

	/* we don't need the passed config anymore - continue working with the saved one
*/
	conf = &jb->conf;

	/* validate the configuration */
	if (conf->jbsize < 1)
		conf->jbsize = FIXED_JB_SIZE_DEFAULT;

	if (conf->resync_threshold < 1)
		conf->resync_threshold = FIXED_JB_RESYNCH_THRESHOLD_DEFAULT;

	/* Set the constant delay to the jitterbuf */
	jb->delay = conf->jbsize;

	return jb;
}
```

```
void fixed_jb_destroy(struct fixed_jb *jb)
{
	/* jitterbuf MUST be empty before it can be destroyed */
	ASSERT(jb->frames == NULL);

	ast_free(jb);
}


static int resynch_jb(struct fixed_jb *jb, void *data, long ms, long ts, long now)
{
	long diff, offset;
	struct fixed_jb_frame *frame;

	/* If jb is empty, just reinitialize the jb */
	if (!jb->frames) {
		/* debug check: tail should also be NULL */
		ASSERT(jb->tail == NULL);

		return fixed_jb_put_first(jb, data, ms, ts, now);
	}

	/* Adjust all jb state just as the new frame is with delivery = the delivery of
the last
	   frame (e.g. this one with max delivery) + the length of the last frame. */

	/* Get the diff in timestamps */
	diff = ts - jb->tail->ts;

	/* Ideally this should be just the length of the last frame. The deviation is the
desired
	   offset */
	offset = diff - jb->tail->ms;

	/* Do we really need to resynch, or this is just a frame for dropping? */
	if (!jb->force_resynch && (offset < jb->conf.resync_threshold && offset > -jb-
>conf.resync_threshold))
		return FIXED_JB_DROP;

	/* Reset the force resynch flag */
	jb->force_resynch = 0;

	/* apply the offset to the jb state */
	jb->rxcore -= offset;
	frame = jb->frames;
	while (frame) {
		frame->ts += offset;
		frame = frame->next;
	}

	/* now jb_put() should add the frame at a last position */
	return fixed_jb_put(jb, data, ms, ts, now);
}


void fixed_jb_set_force_resynch(struct fixed_jb *jb)
{
	jb->force_resynch = 1;
}


int fixed_jb_put_first(struct fixed_jb *jb, void *data, long ms, long ts, long now)
{
	/* this is our first frame - set the base of the receivers time */
	jb->rxcore = now - ts;
```

```
	/* init next for a first time - it should be the time the first frame should be
played */
	jb->next_delivery = now + jb->delay;

	/* put the frame */
	return fixed_jb_put(jb, data, ms, ts, now);
}


int fixed_jb_put(struct fixed_jb *jb, void *data, long ms, long ts, long now)
{
	struct fixed_jb_frame *frame, *next, *newframe;
	long delivery;

	/* debug check the validity of the input params */
	ASSERT(data != NULL);
	/* do not allow frames shorter than 2 ms */
	ASSERT(ms >= 2);
	ASSERT(ts >= 0);
	ASSERT(now >= 0);

	delivery = jb->rxcore + jb->delay + ts;

	/* check if the new frame is not too late */
	if (delivery < jb->next_delivery) {
		/* should drop the frame, but let first resynch_jb() check if this is not
a jump in ts, or
		   the force resynch flag was not set. */
		return resynch_jb(jb, data, ms, ts, now);
	}

	/* what if the delivery time is bigger than next + delay? Seems like a frame for
the future.
	   However, allow more resync_threshold ms in advance */
	if (delivery > jb->next_delivery + jb->delay + jb->conf.resync_threshold) {
		/* should drop the frame, but let first resynch_jb() check if this is not
a jump in ts, or
		   the force resynch flag was not set. */
		return resynch_jb(jb, data, ms, ts, now);
	}

	/* find the right place in the frames list, sorted by delivery time */
	frame = jb->tail;
	while (frame && frame->delivery > delivery) {
		frame = frame->prev;
	}

	/* Check if the new delivery time is not covered already by the chosen frame */
	if (frame && (frame->delivery == delivery ||
			delivery < frame->delivery + frame->ms ||
			(frame->next && delivery + ms > frame->next->delivery)))
	{
		/* TODO: Should we check for resynch here? Be careful to do not allow
threshold smaller than
		   the size of the jb */

		/* should drop the frame, but let first resynch_jb() check if this is not
a jump in ts, or
		   the force resynch flag was not set. */
		return resynch_jb(jb, data, ms, ts, now);
	}

	/* Reset the force resynch flag */
	jb->force_resynch = 0;
```

```
	/* Get a new frame */
	newframe = alloc_jb_frame(jb);
	newframe->data = data;
	newframe->ts = ts;
	newframe->ms = ms;
	newframe->delivery = delivery;

	/* and insert it right on place */
	if (frame) {
		next = frame->next;
		frame->next = newframe;
		if (next) {
			newframe->next = next;
			next->prev = newframe;
		} else {
			/* insert after the last frame - should update tail */
			jb->tail = newframe;
			newframe->next = NULL;
		}
		newframe->prev = frame;

		return FIXED_JB_OK;
	} else if (!jb->frames) {
		/* the frame list is empty or thats just the first frame ever */
		/* tail should also be NULL is that case */
		ASSERT(jb->tail == NULL);
		jb->frames = jb->tail = newframe;
		newframe->next = NULL;
		newframe->prev = NULL;

		return FIXED_JB_OK;
	} else {
		/* insert on a first position - should update frames head */
		newframe->next = jb->frames;
		newframe->prev = NULL;
		jb->frames->prev = newframe;
		jb->frames = newframe;

		return FIXED_JB_OK;
	}
}


int fixed_jb_get(struct fixed_jb *jb, struct fixed_jb_frame *frame, long now, long
interpl)
{
	ASSERT(now >= 0);
	ASSERT(interpl >= 2);

	if (now < jb->next_delivery) {
		/* too early for the next frame */
		return FIXED_JB_NOFRAME;
	}

	/* Is the jb empty? */
	if (!jb->frames) {
		/* should interpolate a frame */
		/* update next */
		jb->next_delivery += interpl;

		return FIXED_JB_INTERP;
	}

	/* Isn't it too late for the first frame available in the jb? */
	if (now > jb->frames->delivery + jb->frames->ms) {
		/* yes - should drop this frame and update next to point the next frame
```

```
(get_jb_head() does it) */
                get_jb_head(jb, frame);

                return FIXED_JB_DROP;
        }

        /* isn't it too early to play the first frame available? */
        if (now < jb->frames->delivery) {
                /* yes - should interpolate one frame */
                /* update next */
                jb->next_delivery += interpl;

                return FIXED_JB_INTERP;
        }

        /* we have a frame for playing now (get_jb_head() updates next) */
        get_jb_head(jb, frame);

        return FIXED_JB_OK;
}


long fixed_jb_next(struct fixed_jb *jb)
{
        return jb->next_delivery;
}


int fixed_jb_remove(struct fixed_jb *jb, struct fixed_jb_frame *frameout)
{
        if (!jb->frames)
                return FIXED_JB_NOFRAME;

        get_jb_head(jb, frameout);

        return FIXED_JB_OK;
}
```

PIELIKUMS NR. 2: "VOIP BUFERIZĀCIJAS ALGORITMA MODELIS"

Fails *NetOver1.gps*(9):

```
**********Existing QoS**********

      ***Variables***

      INITIAL X$AdvTime,20        ;Buffer shift (to codec) interval
      INITIAL X$FirstBuffSt,0             ;Time when first arrived packet was
                           ; held in first buffer position
      INITIAL X$BuffIsActive,0            ;Defines if buffer is active (shifts)
      INITIAL X$BuffPosCount,10           ;Number of positions in the buffer
      INITIAL X$EmptEntPos,1              ;In case the buffer is emty the number
                           ; of position in the buffer, that
                           ; incoming packet should occupy
      BuffMatrix MATRIX,10,3              ;The first parameter (Matrix) -
                           ; abstraction of buffer pull [the
                           ; defininition of variable Buffmatrix
                           ; as array(MATRIX)]
                           ;the second parameter - number of rows
                           ; in array,
                           ;the third parameter - number of columns
                           ; in array:
                           ;1)the first column will contain the
                           ; number of voip packets in this buffer
                           ;2)the second column will contain the
                           ; number of buffer in buffer pull
                           ; in which the packet was putted for the
                           ; first time
                           ;3)the third column will contain time of
                           ; packet generation

      INITIAL X$CommPackNumb,0            ;Number of packets entered jitter buffer
      INITIAL X$LostPackageCount,0        ;Lost packets Count
      INITIAL X$EmptyBuffArrivals,0              ;Number of packets that meet buffer
empty
      INITIAL X$TryPutInOrder,0           ;Sequentialy arrived packets Count
      INITIAL X$PuttedInOrder,0           ;Sequentialy arrived and putted in buffer
                           ; packets Count
      INITIAL X$TryPutInside,0            ;Unsequentialy arrived packets Count
      INITIAL X$PuttedInside,0            ;Unsequentialy arrived and putted in
                           ; buffer packets Count
      INITIAL X$PuttedInExp,0             ;Putted in expected first position of
                           ; buffer package Count
      INITIAL X$PuttedInBuff,0            ;Overall putted in buffer package Count
      INITIAL X$BufferSteps,0             ;Steps of buffer count
      INITIAL X$Max1,0
      INITIAL X$DelInCodec,0              ;The number of packets that will be deleted
                           ; by codec, not by buffering algorithm,
                           ; because they are out of order.
                           ;If a packet came to buffer out of order and
                           ; catch the buffer empty, it will be entered
                           ; to the buffer because buffering algorithm
                           ; (without our modification)
                           ; doesn't fix the number of last packet
                           ; entered codec, and so couldn't defines
                           ; that this packet is out of order and have
                           ; to be deleted. Instead, packet will be put
                           ; into the buffer, will move through the
                           ; buffer till the codec and will be deleted
                           ; by codec.
```

```
       INITIAL X$LastInCodec,0             ;Number of the last packet entered Codec

       INITIAL X$XnVal,0            ;Variable for numbering
       INITIAL X$LastArrived,0             ;Stores Last Arrived Package Number
       INITIAL X$LastArrivedTime,0         ;Time of last arrival in buffer
       INITIAL X$GenReceiveRate,20         ;Packats Generation rate by source codec
       INITIAL X$ExpectedNumber,0


       *** The duration of simulation ***
       GENERATE  ,,,1

       ASSIGN InitPos,1

       ***Initialize Buffer with empty values***
InitCycle     TEST LE P$InitPos,X$BuffPosCount,BuffInited
       MSAVEVALUE BuffMatrix,P$InitPos,1,0
       MSAVEVALUE BuffMatrix,P$InitPos,2,0
       MSAVEVALUE BuffMatrix,P$InitPos,3,0
       ASSIGN InitPos+,1
       TRANSFER ,InitCycle

BuffInited SEIZE    TimeF
       *ADVANCE  30000              ;(ms) (30sec)
       *ADVANCE  60000              ;(ms) (1min)
       ADVANCE       600000         ;(ms) (10min)
       *ADVANCE  3600000            ;(ms) (1hour)
       *ADVANCE  36000000           ;(ms) (10hour)
       RELEASE       TimeF
;      TEST GE X$CommPackNumb,15000,BuffInited
       TERMINATE 1


       ***Voice Channel Activity Detection***
       LoopCount VARIABLE 2
       GENERATE      ,,,1
VADLoop       SEIZE  voicing
       ADVANCE(Exponential(1,0,10000))           ; 10 s. speaking
       RELEASEvoicing
       SEIZE  listening
       ADVANCE(Exponential(1,0,10000))           ; 10 s. listening
       RELEASElistening
;      SAVEVALUE     BuffIsActive,0
;      SAVEVALUE AdvTime,0.001
;      SAVEVALUE     FirstBuffSt,0
       ASSIGN LoopCount,2
       LOOP   LoopCount,VADLoop
       TERMINATE


       ***Voice Package Genetion***
VoipGen       GENERATE X$GenReceiveRate


       ***Transact Param Assigning***

       ASSIGN   TimeStamp,AC1
       ASSIGN   NetDivider,(UNIFORM(2,0,10000))
       SAVEVALUE     XnVal+,1
       ASSIGN   Xnv,X$XnVal

       TEST E    0,F$listening,Ending      ;Testing VAD if not listening phase -
                  ; process packet, else - ignore


       ***Selecting Network to transfer package through***
```

```
       TEST LE  P$NetDivider,9300,Net1_3
       TEST LE  P$NetDivider,6000,Net1_2

       ***First Network Routine***
Net1_1 QUEUE NetworkQ1_1
       SEIZE NETWORK1_1
       DEPART NetworkQ1_1
       ASSIGN TimeNet1_1,(Exponential(3,2,1))
       TEST G P$TimeNet1_1,X$Max1,NotMax
       SAVEVALUE Max1,P$TimeNet1_1
NotMax ADVANCEP$TimeNet1_1
;      ADVANCE  (Exponential(3,0,20))
       RELEASE  NETWORK1_1
;      TRANSFER ,Torout2
       TRANSFER ,Net2_1


Net1_2 QUEUE NetworkQ1_2
       SEIZE    NETWORK1_2
       DEPART NetworkQ1_2
       ASSIGN Net1_2,(Exponential(4,2,12))
       ADVANCEP$Net1_2
;      ADVANCE  (EXPONENTIAL(3,4,40))
       RELEASE  NETWORK1_2
;      TRANSFER ,Torout2
       TRANSFER ,Net2_2

Net1_3 QUEUE NetworkQ1_3
       SEIZE    NETWORK1_3
       DEPART NetworkQ1_3
       ASSIGN Net1_3,(Exponential(5,8,40))
       ADVANCEP$Net1_3
;      ADVANCE  (EXPONENTIAL(3,4,60))
       RELEASE  NETWORK1_3
       TRANSFER ,Net2_3


;ToRout2      ASSIGN   NetDivider,(UNIFORM(2,0,10000))
;      TEST LE  P$NetDivider,9950,Net2_3
;      TEST LE  P$NetDivider,9700,Net2_2

Net2_1 QUEUE NetworkQ2_1
       SEIZE    NETWORK2_1
       DEPART NetworkQ2_1
       ASSIGN Net2_1,(Exponential(3,1,1))
       ADVANCEP$Net2_1
;      ADVANCE  (Exponential(3,0,20))
       RELEASE  NETWORK2_1
;      TRANSFER ,Torout3
       TRANSFER ,Net3_1

Net2_2 QUEUE NetworkQ2_2
       SEIZE    NETWORK2_2
       DEPART NetworkQ2_2
       ASSIGN Net2_2,(Exponential(4,2,6))
       ADVANCEP$Net2_2
;      ADVANCE  (EXPONENTIAL(3,4,40))
       RELEASE  NETWORK2_2
;      TRANSFER ,Torout3
       TRANSFER ,Net3_2

Net2_3 QUEUE NetworkQ2_3
       SEIZE    NETWORK2_3
       DEPART NetworkQ2_3
       ASSIGN Net2_3,(Exponential(5,4,20))
```

```
       ADVANCEP$Net2_3
;      ADVANCE  (EXPONENTIAL(3,4,60))
       RELEASE  NETWORK2_3
       TRANSFER ,Net3_3


;ToRout3      ASSIGN   NetDivider,(UNIFORM(2,0,10000))
;      TEST LE  P$NetDivider,9950,Net3_3
;      TEST LE  P$NetDivider,9700,Net3_2

Net3_1 QUEUE NetworkQ3_1
       SEIZE    NETWORK3_1
       DEPART NetworkQ3_1
       ASSIGN Net3_1,(Exponential(3,1,1))
       ADVANCEP$Net3_1
;      ADVANCE  (Exponential(3,0,20))
       RELEASE  NETWORK3_1
;      TRANSFER ,Torout4
       TRANSFER ,Net4_1


Net3_2 QUEUE NetworkQ3_2
       SEIZE    NETWORK3_2
       DEPART NetworkQ3_2
       ASSIGN Net3_2,(Exponential(4,2,6))
       ADVANCEP$Net3_2
;      ADVANCE  (EXPONENTIAL(3,4,40))
       RELEASE  NETWORK3_2
;      TRANSFER ,Torout4
       TRANSFER ,Net4_2

Net3_3 QUEUE NetworkQ3_3
       SEIZE    NETWORK3_3
       DEPART NetworkQ3_3
       ASSIGN Net3_3,(Exponential(5,4,20))
       ADVANCEP$Net3_3
;      ADVANCE  (EXPONENTIAL(3,4,60))
       RELEASE  NETWORK3_3
       TRANSFER ,Net4_3


;ToRout4      ASSIGN   NetDivider,(UNIFORM(2,0,10000))
;      TEST LE  P$NetDivider,9950,Net4_3
;      TEST LE  P$NetDivider,9700,Net4_2

Net4_1 QUEUE NetworkQ4_1
       SEIZE    NETWORK4_1
       DEPART NetworkQ4_1
       ASSIGN Net4_1,(Exponential(3,2,1))
       ADVANCEP$Net4_1
;      ADVANCE  (Exponential(3,0,20))
       RELEASE  NETWORK4_1
       TRANSFER ,BuffContr

Net4_2 QUEUE NetworkQ4_2
       SEIZE    NETWORK4_2
       DEPART NetworkQ4_2
       ASSIGN Net4_2,(Exponential(4,2,12))
       ADVANCEP$Net4_2
;      ADVANCE  (EXPONENTIAL(3,4,40))
       RELEASE  NETWORK4_2
       TRANSFER ,BuffContr

Net4_3 QUEUE NetworkQ4_3
       SEIZE    NETWORK4_3
       DEPART NetworkQ4_3
```

```
       ASSIGN Net4_3,(Exponential(5,8,40))
       ADVANCE  P$Net4_3
;      ADVANCE  (EXPONENTIAL(3,4,60))
       RELEASE  NETWORK4_3


       ***Entry position Definer***
BuffContr     SAVEVALUE LastArrivedTime,AC1
       SAVEVALUE LastArrived,P$Xnv
       SAVEVALUE CommPackNumb+,1

       ASSIGN TestPos,1
       ASSIGN FirstSzdPos,0

       ***Finding First Seized Position***
SzdFndCcl     TEST LE MX$BuffMatrix(P$TestPos,1),0,SzdFnd
       ASSIGN TestPos+,1
       TEST LE P$TestPos,X$BuffPosCount,SzdNotFnd
       TRANSFER ,SzdFndCcl

       ***Buffer Is Not Empty routine***

SzdFnd ASSIGN FirstSzdPos,P$TestPos
       ASSIGN FirstUser,MX$BuffMatrix(P$FirstSzdPos,1)

       TEST G P$Xnv,P$FirstUser,PutInside

       ***Sequential package arrived***
PutInOrd      SAVEVALUE TryPutInOrder+,1
       ASSIGN PackNumDiff,(P$Xnv-P$FirstUser)
       TEST L P$PackNumDiff,P$FirstSzdPos,Lost
       ASSIGN NewPos,(P$FirstSzdPos-P$PackNumDiff)
       SAVEVALUE PuttedInOrder+,1
       TRANSFER ,SaveInMx

       ***UnSequential package arrived***
PutInside     SAVEVALUE TryPutInside+,1
       ASSIGN PackNumDiff,(P$FirstUser-P$Xnv)
       TEST LE P$PackNumDiff,(X$BuffPosCount-P$FirstSzdPos),Lost
       ASSIGN NewPos,(P$FirstSzdPos+P$PackNumDiff)
       SAVEVALUE PuttedInside+,1
       TRANSFER ,SaveInMx

       ***Buffer Is Empty routine***
SzdNotFnd ASSIGN NewPos,X$EmptEntPos
       SAVEVALUE EmptyBuffArrivals+,1
       TEST L P$Xnv,X$LastInCodec,SaveInMx ;The number of received packet
                           ; smaller than the number of the
                           ; packet last entered Codec?
       SAVEVALUE DelInCodec+,1             ;If yes, add 1 to count
                           ; "DelInCodec" and ignore packet
       TRANSFER ,Ending

       ***Putting Into Buffer Matrix***
SaveInMx  SAVEVALUE BuffIsActive,1
SaveMxV       MSAVEVALUE BuffMatrix,P$NewPos,1,P$Xnv
       MSAVEVALUE BuffMatrix,P$NewPos,2,P$NewPos
       MSAVEVALUE BuffMatrix,P$NewPos,3,P$TimeStamp
       TRANSFER ,Ending

Lost   SAVEVALUE LostPackageCount+,1

Ending TERMINATE

       *** Buffer Shifting Process (Independed from all other times). Shift all
       *** packets in the buffer toward codec***
```

```
AdvContr      GENERATE X$AdvTime

       *** The beginning of de-jitter buffer shift procedure  ***

StartAdv      ASSIGN DestPos,1
       ASSIGN CcNum,0
       ASSIGN CcPos,0
       ASSIGN CcTime,0

AdvCycle      TEST LE P$DestPos,X$BuffPosCount,BuffShftd

       SAVEVALUE Dest,P$DestPos

       ASSIGN ExcNum,MX$BuffMatrix(P$DestPos,1)   ;Save the content of "P$DestPos"
       ASSIGN ExcPos,MX$BuffMatrix(P$DestPos,2)   ; position of the buffer in
       ASSIGN ExcTime,MX$BuffMatrix(P$DestPos,3)  ; variables (starting from
                             ; the last position of the buffer

       MSAVEVALUE BuffMatrix,P$DestPos,1,P$CcNum  ;Store to "P$DestPos" position of
       MSAVEVALUE BuffMatrix,P$DestPos,2,P$CcPos  ; the buffer the values from
       MSAVEVALUE BuffMatrix,P$DestPos,3,P$CcTime ; previos position (P$DestPos-1)
                             ; or zero if "P$DestPos" is the last
                             ; position of the buffer(closest to
                             ;the network)

       ASSIGN CcNum,P$ExcNum        ;Save previous content of "P$DestPos"
       ASSIGN CcPos,P$ExcPos        ; position of the buffer in other
       ASSIGN CcTime,P$ExcTime             ; variable set for further store in
                             ; next buffer position(nearer to codec)

       ASSIGN DestPos+,1
       TRANSFER ,AdvCycle

BuffShftd     SAVEVALUE ExpectedNumber+,1
       TEST LE X$LastInCodec,P$CcNum,CountSt      ;Is the number of last packet entered
                             ; codec smaller than the number of the
                             ; paxket that should enter codec now?
       SAVEVALUE LastInCodec,P$CcNum              ;Yes. Save the numer of the new last
                             ; entered Codec packet.

CountSt       SAVEVALUE     BufferSteps+,1
EndBuffR      TERMINATE


;SAVEVALUE Results To Pay Attention to
 ;LOSTPACKAGECOUNT        - Lost Package Count
 ;TRYPUTINORDER           - Sequentialy arrived package Count
 ;PUTTEDINORDER           - Sequentialy arrived and putted in buffer package Count
 ;TRYPUTINSIDE            - Unsequentialy arrived package Count
 ;PUTTEDINSIDE            - Unsequentialy arrived and putted in buffer package Count
```

PIELIKUMS NR. 3: "TĪKLA SIMULĀCIJAS MODELIS"

Fails *package.ned*:

```
package net1;

import org.omnetpp.queueing.Clone;
import ned.DelayChannel;
import org.omnetpp.queueing.Sink;
import ned.IdealChannel;
import org.omnetpp.queueing.Source;


@license(LGPL);
//
// TODO documentation
//
// @author Kirils Solovjovs
//

network Network
{
    @display("bgb=713,257,white,white;bgl=7");
    volatile int buffersize;
    // centrāli norādāms bufera izmērs
    volatile double packettime @unit(s) = default(20ms);
    // pakete tiek ģenerēta ik pēc packettime sekundēm

    submodules:
        voicesender: Source {
            // šis avots ģenerē visas paketes
            @display("p=42,37;i=device/cellphone");
            interArrivalTime = packettime;
            jobName = "voice";
            jobType = 1;
            VAD = true; // vai lietot VAD/DTX
        }
        modecB: Sink {
            @display("p=672,37;i=device/cellphone2");
            keepJobs = false;
            timesize = packettime;
        }
        router0: Routerx1x3 {
            @display("i=device/router;p=35,129");
            prob0 = 0.01;
            prob1 = 0.05;
            firstgate = 2;
        }
        router11: Routerx1x1 {
            @display("p=114,70");
            entrydelay = exponential(1ms)+2ms;
        }
        router12: Routerx1x1 {
            @display("p=114,133");
            entrydelay = exponential(12ms)+2ms;
        }
        router13: Routerx1x1 {
            @display("p=114,200");
            entrydelay = exponential(40ms)+8ms;
        }
        router5: Routerx3x1 {
            @display("p=405,133");
        }
```

```
        jitterbufferB: jitterbufferB {
            @display("p=596,37");
            timesize = packettime;
            jbsize = buffersize;
        }
        clone: Clone {
            @display("p=488,133");
        }
        modecAD3: Sink {
            @display("p=514,28;i=device/cellphone2");
            timesize = packettime;
        }
        jitterbufferE: jitterbufferE {
            @display("p=596,185");
            timesize = packettime;
            jbsize = buffersize;
        }
        modecE: Sink {
            @display("p=672,185;i=device/cellphone2");
            timesize = packettime;
        }
        jitterbufferAD2: jitterbufferAD2 {
            @display("p=420,200");
            timesize = packettime;
            //adaptīviem algoritmiem simulācijas ietvaros
            //izmērs tiek rēķināts pēc formulām
            basejbsize = ceil(buffersize*0.75);
            maxjbsize = min(buffersize*2,10);
        }
        modecAD2: Sink {
            @display("p=514,200;i=device/cellphone2");
            timesize = packettime;
        }
        jitterbufferCF: jitterbufferCF {
            @display("p=596,114");
            timesize = packettime;
            jbsize = buffersize;
        }
        modecCF: Sink {
            @display("p=672,114;i=device/cellphone2");
            timesize = packettime;
        }
        router41: Routerx1x1 {
            @display("p=324,72");
            entrydelay = exponential(2ms)+1ms;
        }
        router42: Routerx1x1 {
            @display("p=324,133");
            entrydelay = exponential(12ms)+2ms;
        }
        router43: Routerx1x1 {
            @display("p=324,200");
            entrydelay = exponential(40ms)+8ms;
        }
        router21: Routerx1x1 {
            @display("p=185,70");
            entrydelay = exponential(1ms)+1ms;
        }
        router22: Routerx1x1 {
            @display("p=185,133");
            entrydelay = exponential(6ms)+2ms;
        }
        router23: Routerx1x1 {
            @display("p=185,200");
            entrydelay = exponential(20ms)+4ms;
        }
```

```
        router31: Routerx1x1 {
            @display("p=259,70");
            entrydelay = exponential(1ms)+1ms;
        }
        router32: Routerx1x1 {
            @display("p=259,133");
            entrydelay = exponential(6ms)+2ms;
        }
        router33: Routerx1x1 {
            @display("p=259,200");
            entrydelay = exponential(20ms)+4ms;
        }
        jitterbufferAD3: jitterbufferAD3 {
            @display("p=420,28");
            timesize = packettime;
            //adaptīviem algoritmiem simulācijas ietvaros
            //izmērs tiek rēķināts pēc formulām
            basejbsize = ceil(buffersize*0.75);
            maxjbsize = min(buffersize*2,10);
        }
    connections:
        jitterbufferB.voice --> modecB.in++;
        router0.out++ --> IdealChannel --> router11.in++;
        router0.out++ --> IdealChannel --> router12.in++;
        router0.out++ --> IdealChannel --> router13.in++;
        router5.out++ --> clone.in++;
        clone.out++ --> jitterbufferB.in;
        voicesender.out --> DelayChannel {  delay = 25ms; } --> router0.in++;
        clone.out++ --> jitterbufferE.in;
        jitterbufferE.voice --> modecE.in++;
        clone.out++ --> jitterbufferAD2.in;
        jitterbufferAD2.voice --> modecAD2.in++;
        jitterbufferCF.voice --> modecCF.in++;
        clone.out++ --> jitterbufferCF.in;
        router11.out++ --> router21.in++;
        router21.out++ --> router31.in++;
        router31.out++ --> router41.in++;
        router12.out++ --> router22.in++;
        router22.out++ --> router32.in++;
        router32.out++ --> router42.in++;
        router13.out++ --> router23.in++;
        router23.out++ --> router33.in++;
        router33.out++ --> router43.in++;
        router43.out++ --> router5.in++;
        router42.out++ --> router5.in++;
        router41.out++ --> router5.in++;
        clone.out++ --> jitterbufferAD3.in;
        jitterbufferAD3.voice --> modecAD3.in++;
}
```

PIELIKUMS NR. 4: "ALGORITMU IMPLEMENTĀCIJA"

Džiter-bufera vadības algoritmu klašu deklarācijas redzamas failā *jitterbuffer.h*:

```
// (C) Kirils Solovjovs, 2011

#ifndef JITTERBUFFER_H_
#define JITTERBUFFER_H_


class jitterbufferB : public cSimpleModule {
	private:
		cMessage* stack[100]; // ziņojumu glabāšanai
		bool stackv[100]; // vai konkrētā vieta ir aizņemta
		double stacktime[100]; // ziņojuma ienākšanas laiks
		int jbsize; // bufera izmērs

		// signāli palīdz ievākt statistiku
		simsignal_t dropSig;
		simsignal_t getSig;
		simsignal_t sendSig;
	protected:
		virtual void initialize();
		virtual void finish();
		virtual void handleMessage(cMessage *msg);
};


class jitterbufferCF : public cSimpleModule {
	private:
		cMessage* stack[100]; // ziņojumu glabāšanai
		bool stackv[100]; // vai konkrētā vieta ir aizņemta
		double stacktime[100]; // ziņojuma ienākšanas laiks
		int jbsize; // bufera izmērs
		int lastout; // pēdējās uz buferu izsūtītās paketes numurs
		// signāli palīdz ievākt statistiku
		simsignal_t dropSig;
		simsignal_t IdropSig;
		simsignal_t QeSig;
		simsignal_t getSig;
		simsignal_t sendSig;
	protected:
		virtual void initialize();
		virtual void finish();
		virtual void handleMessage(cMessage *msg);
};



class jitterbufferE : public cSimpleModule {
	private:
		cMessage* stack[100]; // ziņojumu glabāšanai
		bool stackv[100]; // vai konkrētā vieta ir aizņemta
		double stacktime[100]; // ziņojuma ienākšanas laiks
		int jbsize; // bufera izmērs
		int lastout; // paketes numurs, kuru kodeks šobrīd atskaņo
		// signāli palīdz ievākt statistiku
		simsignal_t dropSig;
		simsignal_t IdropSig;
		simsignal_t getSig;
		simsignal_t sendSig;
	protected:
		virtual void initialize();
```

```
		virtual void finish();
		virtual void handleMessage(cMessage *msg);
};



class jitterbufferAD2 : public cSimpleModule {
	private:
		cMessage* stack[100]; // ziņojumu glabāšanai
		bool stackv[100]; // vai konkrētā vieta ir aizņemta
		double stacktime[100]; // ziņojuma ienākšanas laiks
		int jbsize; // bufera izmērs

		int maxjbsize,basejbsize; //maksimālais un sākotnējais izmērs
		int lastout; // paketes numurs, kuru kodeks šobrīd atskaņo
		int shrinktimer;
		// skaitītājs, kas skaita, pēc cik ilga laika jāsamazina buferis

		// signāli palīdz ievākt statistiku
		simsignal_t dropSig;
		simsignal_t getSig;
		simsignal_t getSSig;
		simsignal_t sendSig;
	protected:
		virtual void initialize();
		virtual void finish();
		virtual void handleMessage(cMessage *msg);
};


class jitterbufferAD3 : public cSimpleModule {
	private:
		cMessage* stack[100]; // ziņojumu glabāšanai
		bool stackv[100]; // vai konkrētā vieta ir aizņemta
		double stacktime[100]; // ziņojuma ienākšanas laiks
		int jbsize; // bufera izmērs

		int maxjbsize,basejbsize; //maksimālais un sākotnējais izmērs
		int lastout; // paketes numurs, kuru kodeks šobrīd atskaņo
		int dropped,okeyed; // uzskaita nepieņemtās un pieņemtās paketes

		// signāli palīdz ievākt statistiku
		simsignal_t dropSig;
		simsignal_t getSig;
		simsignal_t getSSig;
		simsignal_t sendSig;
	protected:
		virtual void initialize();
		virtual void finish();
		virtual void handleMessage(cMessage *msg);
};


#endif /* JITTERBUFFER_H_ */
```

Džiter-bufera vadības algoritmu klases un to metodes *initialize*, *finish* un *handleMessage*, ir redzamas failā *jitterbuffer.cc*:

```
// (C) Kirils Solovjovs, 2011

#include <omnetpp.h>
#include "jitterbuffer.h"
#include <string>
using namespace std;
```

```
Define_Module(jitterbufferB);

void jitterbufferB::initialize(){
       jbsize=par("jbsize");
       dropSig=registerSignal("dropped");
       getSig=registerSignal("received");
       sendSig=registerSignal("servicetime");

       for(int i=0;i<jbsize;i++)stackv[i]=false;
       scheduleAt(0, new cMessage("tick")); // izsūtīšanas atzīme
}

void jitterbufferB::finish(){
       for(int i=0;i<jbsize;i++){
              if(stackv[i])
                     delete stack[i];
       //            send(stack[i],"voice");
     }
}

void jitterbufferB::handleMessage(cMessage *msg){
// algoritms B, CISCO

       if (msg->isSelfMessage()){ // saņemta izsūtīšanas atzīme
              scheduleAt(simTime() + par("timesize").doubleValue(), msg);
              //reģistrējam nākamo atzīmi
              if(stackv[0]){
                     // ja ir ko izsūtīt, tad izsūtam
                     send(stack[0],"voice");
                     EV<<"sending out "<< stack[0]->getName() <<endl;
                     emit(sendSig,simTime().dbl()-stacktime[0]);
              }

              // nobīdam visu buferi par 1
              for(int i=0;i<jbsize-1;i++){
                     stack[i]=stack[i+1];
                     stackv[i]=stackv[i+1];
                     stacktime[i]=stacktime[i+1];
              }
              stackv[jbsize-1]=false;

       } else {
        //saņemta ienākošā pakete
              emit(getSig,1);
              EV<<"got "<<msg->getName();

              int jbElement=-1;
              int placeme=-1;
              for(int i=0;i<jbsize;i++){ // vai buferī šobrīd ir kāda pakete?
                     if(stackv[i]){
                            jbElement=i;
                            break;
                     }
              }

              if(jbElement==-1) // nav pakešu
                     placeme=jbsize-1; // ievietosim bufera galā
              else {
                     placeme=jbElement+atoi(msg->getName())-
atoi(stack[jbElement]->getName());
                     if(placeme<0 || placeme>=jbsize)
                            placeme=-1;
                     else if(stackv[placeme])placeme=-1;
                     // ievietosim atbilstošajā vietā, ja vien tāda eksistē
                     // un nav aizņemta
```

```
		}

		if(placeme==-1){ //nometam
			EV<<" dropped, potential place was "<<jbElement+
atoi(msg->getName())-atoi(stack[jbElement]->getName())<<endl;
			delete msg;
			emit(dropSig,1);
		} else { // ievietojam
			stackv[placeme]=true;
			stack[placeme]=msg;
			stacktime[placeme]=simTime().dbl();
			EV<< " ACCEPTED"<<endl;
		}
	}
}


Define_Module(jitterbufferCF);
void jitterbufferCF::initialize(){
	jbsize=par("jbsize");
	dropSig=registerSignal("dropped");
	getSig=registerSignal("received");
	sendSig=registerSignal("servicetime");
	IdropSig=registerSignal("Idropped");
	QeSig=registerSignal("Qempty");
	lastout=0;
	for(int i=0;i<jbsize;i++)stackv[i]=false;
	scheduleAt(0, new cMessage("tick"));
}

void jitterbufferCF::finish(){
	for(int i=0;i<jbsize;i++){
		if(stackv[i])
			delete stack[i];
	//		send(stack[i],"voice");
	}

}

void jitterbufferCF::handleMessage(cMessage *msg){
// algoritms CF, Broitman
	if (msg->isSelfMessage()){ //saņemta izsūtīšanas atzīme
		scheduleAt(simTime() + par("timesize").doubleValue(), msg);
		if(stackv[0]){ // ir ko izsūtīt
			if(lastout<atoi(stack[0]->getName()))
				lastout=atoi(stack[0]->getName());
			// fiksējam pēdējo uz kodeku izsūtītās paketes numuru
			send(stack[0],"voice");
			EV<<"sending out "<< stack[0]->getName() <<endl;
			emit(sendSig,simTime().dbl()-stacktime[0]);
		}

		for(int i=0;i<jbsize-1;i++){ //nobīdam buferi
			stack[i]=stack[i+1];
			stackv[i]=stackv[i+1];
			stacktime[i]=stacktime[i+1];
		}
		stackv[jbsize-1]=false;
	} else { //saņemta ienākošā pakete
		emit(getSig,1);
		EV<<"got "<<msg->getName();

		int jbElement=-1;
		int placeme=-1;
		for(int i=0;i<jbsize;i++){
			if(stackv[i]){
```

```
                        jbElement=i;
                        break;
                }
        }

        if(jbElement==-1)
                placeme=jbsize-1; //buferis ir tukšs,
                                // pakete tiks ievietota bufera galā
        else {
                placeme=jbElement+atoi(msg->getName())-
atoi(stack[jbElement]->getName());
                if(placeme<0 || placeme>=jbsize)
                        placeme=-1;
                else if(stackv[placeme])placeme=-1;
                // pakete tiks ievietota atbilstošajā vietā,
                // ja vien tas ir iespējams tā nav pilna
        }

        if(lastout>atoi(msg->getName())){
                // pakete ir vecāka par to, kas jau izsūtīta uz kodeku
                EV << " *late*";
                placeme=-1;
                emit(IdropSig,1);
        } else if (jbElement==-1)
                emit(QeSig,1);
        if(placeme==-1){
                // nometam paketi
                if(jbElement>-1)
                        EV<<" dropped, potential place was "<<jbElement+
atoi(msg->getName())-atoi(stack[jbElement]->getName())<<endl;
                else
                        EV<<" dropped, empty q, too late"<<endl;
                delete msg;
                emit(dropSig,1);
        } else {
                // pieņemam paketi
                stackv[placeme]=true;
                stack[placeme]=msg;
                stacktime[placeme]=simTime().dbl();
                EV<< " ACCEPTED"<<endl;
        }
    }

}


/*
 * Piezīme par algoritmiem E, AD2 un AD3.
 *
 * Lai gan algoritmi pēc būtības darbojas ar paketes izsūtīšanas laiku,
 * šī modeļa ietvaros vienkāršības labad tiek salīdzināti to radīšanas numuri,
 * kas 1:1 atbilst ar to radīšanas laiku (skat. Source.cc).
 */

Define_Module(jitterbufferE);
void jitterbufferE::initialize(){
    jbsize=par("jbsize");
    dropSig=registerSignal("dropped");
    IdropSig=registerSignal("Idropped");
    getSig=registerSignal("received");
    sendSig=registerSignal("servicetime");

    lastout=0;
    for(int i=0;i<jbsize;i++)stackv[i]=false;
    scheduleAt(0, new cMessage("tick"));
}
```

```
void jitterbufferE::finish(){
       for(int i=0;i<jbsize;i++){
              if(stackv[i])
                     delete stack[i];
       //            send(stack[i],"voice");
     }
}

void jitterbufferE::handleMessage(cMessage *msg){
// algoritms E, Solovjovs, 03.05.2011. ar labojumiem
       if (msg->isSelfMessage()){ // saņemta izsūtīšanas atzīme
              scheduleAt(simTime() + par("timesize").doubleValue(), msg);
              if(stackv[0]){ // ir ko izsūtīt
                     lastout=max(lastout+1,atoi(stack[0]->getName()));
                     // fiksējam uz buferi izsūtītās paketes numuru, kas nav mazāks
                     // par iepriekšējo, plus 1
                     send(stack[0],"voice");
                     EV<<"sending out "<< stack[0]->getName() <<endl;
                     emit(sendSig,simTime().dbl()-stacktime[0]);
              } else
                     if(lastout>0)lastout++;
                     // arī, ja nav ko izsūtīt, palielinam numura skaitītāju
                  // (ja vien izsūtīšana vispār ir sākta)

              for(int i=0;i<jbsize-1;i++){ //nobīdam buferi par 1
                     stack[i]=stack[i+1];
                     stackv[i]=stackv[i+1];
                     stacktime[i]=stacktime[i+1];
              }
              stackv[jbsize-1]=false;
       } else { //saņemta ienākošā pakete
              emit(getSig,1);
              EV<<"got "<<msg->getName();

              int placeme=-1;
              if(lastout>0) // kaut kas jau ir ticis izsūtīts
                     placeme=atoi(msg->getName())-lastout-1;
                     // tātad paketei ir konkrēta vieta
              else { // ja nē, tad rēķinam vajadzīgo vietu ar standarta algoritmu
                     int jbElement=-1;
                     for(int i=0;i<jbsize;i++){
                            if(stackv[i]){
                                   jbElement=i;
                                   break;
                            }
                     }
                     if(jbElement==-1)
                            placeme=jbsize-1;
                     else
                            placeme=jbElement+atoi(msg->getName())-
atoi(stack[jbElement]->getName());

              }

              if(placeme<0 || placeme>=jbsize)
                     placeme=-1;
              else if(stackv[placeme])placeme=-1;

              if(lastout>=atoi(msg->getName())){
                     // paketes numurs ir mazāks vai vienāds ar to,
                     // kam ir jau jābūt kodekā
                     EV << " *late*";
                     placeme=-1;
                     emit(IdropSig,1);
                     // metīsim nost
```

```
		}

		if(placeme==-1){
			//nometam paketi
			EV<<" dropped, potential place was "<<+atoi(msg->getName())-
lastout-1<<endl;
			delete msg;
			emit(dropSig,1);
		} else {
			//pieņemam paketi
			stackv[placeme]=true;
			stack[placeme]=msg;
			stacktime[placeme]=simTime().dbl();
			EV<< " ACCEPTED"<<endl;
		}
	}
}

Define_Module(jitterbufferAD2);

void jitterbufferAD2::initialize(){
	jbsize=par("basejbsize");
	dropSig=registerSignal("dropped");
	getSig=registerSignal("received");
	getSSig=registerSignal("jbsizea");
	sendSig=registerSignal("servicetime");
	lastout=0;
	shrinktimer=0;
	maxjbsize=par("maxjbsize");
	basejbsize=par("basejbsize");
	for(int i=0;i<maxjbsize;i++)stackv[i]=false;
	scheduleAt(0, new cMessage("tick"));
}

void jitterbufferAD2::finish(){
	for(int i=0;i<jbsize;i++){
		if(stackv[i])
			delete stack[i];
	//		send(stack[i],"voice");
     }

}


void jitterbufferAD2::handleMessage(cMessage *msg){
// algoritms AD2, Solovjovs, 14.05.2011. ar labojumu
	if (msg->isSelfMessage()){ // saņemta izsūtīšanas atzīme
		scheduleAt(simTime() + par("timesize").doubleValue(), msg);
		if(stackv[0]){ // ir ko izsūtīt
			lastout=max(lastout+1,atoi(stack[0]->getName()));
			send(stack[0],"voice");
			EV<<"sending out "<< stack[0]->getName()<< " after "
<<(simTime().dbl()-stacktime[0])<<"s"<<endl;
			emit(sendSig,simTime().dbl()-stacktime[0]);
		} else
			if(lastout>0)lastout++; // saskaņā ar E algoritmu

		for(int i=0;i<jbsize-1;i++){
			stack[i]=stack[i+1];
			stackv[i]=stackv[i+1];
			stacktime[i]=stacktime[i+1];
		}
		stackv[jbsize-1]=false;
		//cout << "ST JBS BJBS "<<shrinktimer<< " "<<jbsize <<" "
<<basejbsize<<endl;
		if((shrinktimer==0)&&(jbsize>basejbsize)){
```

```
                        // ja ir laiks un buferis ir palielināts,
                        // tad samazinam bufera izmēru
                         shrinktimer=jbsize;
                         jbsize--;
                         EV << "decreasing AJB to "<< jbsize<<endl;
                }

                if(shrinktimer>0) // samazinam taimeri
                        shrinktimer--;
                /*
                printf("STACK DEBUG: ");
                for(int i=0;i<jbsize;i++){
                        printf("%d ",stackv[i]?atoi(stack[i]->getName()):-1);
                }
                printf("\n");*/
        } else {
                // ienākošā pakete
                // apstrādājam saskaņā ar E algoritmu
                emit(getSig,1);
                EV<<"got "<<msg->getName();

                int placeme=-1;

                if(lastout>0)
                        placeme=atoi(msg->getName())-lastout-1;
                else {
                        int jbElement=-1;
                        for(int i=0;i<jbsize;i++){
                                if(stackv[i]){
                                        jbElement=i;
                                        break;
                                }
                        }
                        if(jbElement==-1)
                                placeme=jbsize-1;
                        else
                                placeme=jbElement+atoi(msg->getName())-
atoi(stack[jbElement]->getName());
                }

                if(placeme<0 || placeme>=jbsize)
                        placeme=-1;
                else if(stackv[placeme])placeme=-1;

                if(lastout>=atoi(msg->getName())){
                        EV << " *late*";
                        placeme=-1;
                }

                if(placeme==-1){ // nav kur novietot
                        /*cout << jbsize<<" "<<maxjbsize<<endl;
                        cout << "lastout = "<<lastout<<endl;
                        cout << atoi(msg->getName())<<endl;*/
                        if((jbsize<maxjbsize)&&(lastout>0)&&
(atoi(msg->getName())>lastout)){
                                // mēģinam palielināt bufera izmēru
                                int newsize=atoi(msg->getName())-lastout;
                                if(newsize>maxjbsize)
                                        newsize=maxjbsize;
                                newsize-=jbsize;
                                if(newsize>0){ // izdodas palielināt
                                        shrinktimer=jbsize+newsize;
                                        // uzstādam taimeri uz jauno bufera izmēru
                                        EV << "increasing AJB by "<< newsize <<" to "
<<(jbsize+newsize)<<endl;
                                        for(int i=jbsize;i<jbsize+newsize;i++)
```

```
                                    // inicializējam palielināto bufera daļu
                                           stackv[i]=false;
                                    jbsize+=newsize;

                                    // mēģinam novietot paketi vēlreiz
                                    placeme=atoi(msg->getName())-lastout-1;
                                    if(placeme<0 || placeme>=jbsize)
                                           placeme=-1;
                                    else if(stackv[placeme])placeme=-1;

                             }
                      }
               }

               if(placeme>-1){
                      // novietojam paketi
                      stackv[placeme]=true;
                      stack[placeme]=msg;
                      stacktime[placeme]=simTime().dbl();
                      EV<< " ACCEPTED"<<endl;
               } else {
                      // nometam paketi
                      EV<<" dropped, potential place was "<<+atoi(msg->getName())-
lastout-1<<endl;
                      emit(dropSig,1);
                      delete msg;
               }

               emit(getSSig,jbsize);
       }
}

Define_Module(jitterbufferAD3);

void jitterbufferAD3::initialize(){
       jbsize=par("basejbsize");
       dropSig=registerSignal("dropped");
       getSig=registerSignal("received");
       getSSig=registerSignal("jbsizea");
       sendSig=registerSignal("servicetime");
       lastout=0;
       dropped=0;
       okeyed=0;
       maxjbsize=par("maxjbsize");
       basejbsize=par("basejbsize");
       for(int i=0;i<maxjbsize;i++)stackv[i]=false;
       scheduleAt(0, new cMessage("tick"));
}

void jitterbufferAD3::finish(){
       for(int i=0;i<jbsize;i++){
              if(stackv[i])
                      delete stack[i];
       //             send(stack[i],"voice");
      }
}

void jitterbufferAD3::handleMessage(cMessage *msg){
// algoritms AD2, Solovjovs, 15.05.2011. ar labojumu
           if (msg->isSelfMessage()){ // saņemta izsūtīšanas atzīme
               scheduleAt(simTime() + par("timesize").doubleValue(), msg);
               if(stackv[0]){ // ir ko izsūtīt
                      lastout=max(lastout+1,atoi(stack[0]->getName()));
                      send(stack[0],"voice");
                      EV<<"sending out "<< stack[0]->getName()<< " after "
<<(simTime().dbl()-stacktime[0])<<"s"<<endl;
```

```
                        emit(sendSig,simTime().dbl()-stacktime[0]);
            } else
                if(lastout>0)lastout++; // saskaņā ar E algoritmu

                  for(int i=0;i<jbsize-1;i++){
                        stack[i]=stack[i+1];
                        stackv[i]=stackv[i+1];
                        stacktime[i]=stacktime[i+1];
                  }
                  stackv[jbsize-1]=false;

                  if((jbsize>basejbsize)&&(okeyed>dropped)&&(okeyed>10)){
                        // ja var buferi samazināt un ir veiksmīgi saņemtas vairāk
                        // kā 10 paketes,
                        // kā arī veiksmīgi saņemto pakešu skaits ir lielāks par
                        // nomesto, tad samazinam buferi
                        okeyed=dropped=0; //reset
                        jbsize--;
                        EV << "decreasing AJB to "<< jbsize<<endl;
                  }
                  /*
                  printf("STACK DEBUG: ");
                  for(int i=0;i<jbsize;i++){
                        printf("%d ",stackv[i]?atoi(stack[i]->getName()):-1);
                  }
                  printf("\n");*/
          } else {
                  //ienākošā pakete, apstrāde tiek veikta saskaņā ar E algoritmu
             emit(getSig,1);
             EV<<"got "<<msg->getName();

             int placeme=-1;
             if(lastout>0)
                        placeme=atoi(msg->getName())-lastout-1;
                  else {
                        int jbElement=-1;
                        for(int i=0;i<jbsize;i++){
                              if(stackv[i]){
                                    jbElement=i;
                                    break;
                              }
                        }
                        if(jbElement==-1)
                              placeme=jbsize-1;
                        else
                              placeme=jbElement+atoi(msg->getName())-
atoi(stack[jbElement]->getName());
                  }
                  if(placeme<0 || placeme>=jbsize)
                        placeme=-1;
                  else if(stackv[placeme])placeme=-1;

             if(lastout>=atoi(msg->getName())){
                        EV << " *late*";
                        placeme=-1;
                  }

                  if(placeme==-1){ //nav kur novietot
                        /*cout << jbsize<<" "<<maxjbsize<<endl;
                        cout << "lastout = "<<lastout<<endl;
                        cout << atoi(msg->getName())<<endl;*/

                        if((jbsize<maxjbsize)&&(lastout>0)&&
(atoi(msg->getName())>lastout)){
                              // mēģinam palielināt bufera izmēru
                              int newsize=atoi(msg->getName())-lastout;
```

```
                                    if(newsize>maxjbsize)
                                           newsize=maxjbsize;
                                    newsize-=jbsize;
                                    //cout << "could try to increase by "
<<newsize<<endl;
                                    if(newsize>0){
                                           // izdodas palielināt
                                           EV << "increasing AJB by "<< newsize <<" to "
<<(jbsize+newsize)<<endl;
                                           for(int i=jbsize;i<jbsize+newsize;i++)
                                                  stackv[i]=false;

                                           jbsize+=newsize;
                                           okeyed=dropped=0; //reset

                                           // atkārtoti mēģinam ievietot paketi buferī
                                           placeme=atoi(msg->getName())-lastout-1;
                                           if(placeme<0 || placeme>=jbsize)
                                                  placeme=-1;
                                           else if(stackv[placeme])placeme=-1;

                                    }
                             }
                      }
                      if(placeme>-1){
                             stackv[placeme]=true;
                             stack[placeme]=msg;
                             stacktime[placeme]=simTime().dbl();
                             EV<< " ACCEPTED"<<endl;
                             // pieņemam paketi un palielinam skaitītāju
                             okeyed++;
                      } else {
                             EV<<" dropped, potential place was "<<+
atoi(msg->getName())-lastout-1<<endl;
                             emit(dropSig,1);
                             // nometam paketi un palielinam skaitītāju
                             dropped++;
                             delete msg;
                      }

                      emit(getSSig,jbsize);
           }
}
```

DOKUMENTĀRĀ LAPA

Maģistra darbs “Džiter-bufera vadības algoritma optimizācija laikkritiskai datplūsmai” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka maģistra darbs veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: Kirils Solovjovs

Rekomendēju darbu aizstāvēšanai.

Vadītājs: Dr. sc. comp. Mihails Broitmans

Recenzents: vadošais pētnieks Dr. sc. comp. Jānis Martinsons

Darbs iesniegts maģistrantūras sekretariātā

Metodiķe:

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. __, vērtējums

Komisijas sekretārs: