TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Del Remi Liik

Java andmevoogude implementatsioonide analüüs ja võrdlus DataStream'i ning ObjectStream’i näitel

Bakalaureusetöö (9 EAP)

Juhendaja: Simmo Saan MSc

Tartu 2021

Java andmevoogude implementatsioonide analüüs ja võrdlus DataStream'i ning ObjectStream’i näitel

Lühikokkuvõte:

Mitmetest osadest koosnevates infotehnoloogilistes süsteemides omandab tihti kriitilise tähtsuse andmevahetus nende süsteemide erinevate osade vahel. Java programmeerimiskeeles kirjutatud süsteemides on üheks serveritevahelise andmevahetuse rakendamise võimaluseks kasutada andmevoogusid. Käesolevas bakalaureusetöös tutvustati Java andmevoogude loogilist tausta ja implementeeriti kolm erinevat andmevoogudega serveritevahelise andmevahetuse lahendust, millest kaks kasutasid *DataStream*’i ja üks *ObjectStream*’i. Implementeeritud lahendusi analüüsiti ning võrreldi implementeerimise protsessi ja jõudluse põhjal. Järeldati, et erinevatel implementatsioonidel on erinevad tugevad ja nõrgad küljed ning et nende seast üheselt parimat valida ei ole võimalik. Võrreldud lahendustele pakuti sobilikke kasutusvaldkondi.

Võtmesõnad: Java, andmevoog, DataStream, ObjectStream
CERCS: P170 - Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Analysis and comparison of data stream implementations in Java using the examples of DataStream and ObjectStream

Abstract:

In IT systems consisting of multiple parts, the communication between those parts is often of critical importance. In systems written in Java, one way of implementing communication between different parts of a system is using data streams. In this thesis an introduction of the logic behind data streams is made and three different ways of communication between servers with data streams are implemented. Of those implementations two use DataStream and one uses ObjectStream. Implementations are analysed and compared based on the process of implementation and performance. It is concluded that different implementations have different stronger and weaker sides and that choosing one that is best in all situations is not possible. Suitable usages for the compared implementations are offered.

Keywords: Java, data stream, DataStream, ObjectStream
CERCS: P170 - Computer science, numerical analysis, systems, control

Sisukord

1. Sissejuhatus

Mida aeg edasi, seda suuremaks ja keerulisemaks muutuvad igapäevaselt kasutatavad infotehnoloogilised süsteemid. Tihti koosnevad need süsteemid eraldiseisvatest serveritest, millest kõik võivad mängida erinevat rolli ja täita erinevaid ülesandeid. Sellest tulenevalt on ka andmed, mida serverid hoiavad, erinevad. Süsteemi terviklikkuseks on vaja, et selle eraldiseisvate osade poolt toodetud andmed oleksid kättesaadavad ka teiste süsteemi osade poolt. Mida suuremaks kasvavad masinate vahel saadetud andmete mahud, seda kriitilisema tähtsuse omandab efektiivse andmevahetuse tarkvaraline rakendamine. Et omavahel suhtlevad serverid üksteisele saadetud andmetest samamoodi aru saaksid, tuleb andmevahetusele defineerida kindel protokoll, mis teeb üheselt mõistetavaks, kuidas andmeid saatma ja vastu võtma peab [1].

On tavaline, et rääkides andmete saatmisest ja vastuvõtmisest, kujutatakse andmeid mingisuguse kogumi kujul. Näiteks on vaja ühest seadmest teise saata mingisugust üksikut faili, sõnumit, päringut või muud sellist. Kuid tihti on andmeid vaja saata erinevate masinate vahel pidevalt ja eraldamatult. See võib olla tingitud sellest, et saadetavad andmed on mahu poolest lihtsalt liiga suured, et neid ühe tervikuna saata, või saadetavaid andmeid tekitatakse pidevalt juurde ja need genereeritud andmed moodustavad ühe loogilise terviku. Java programmeerimiskeelt kasutavates rakendustes on selliseks serveritevaheliseks suhtluseks üheks võimaluseks kasutada andmevoogusid.

Käesoleva töö eesmärgiks on tutvustada ja analüüsida Java programmeerimiskeeles andmevoogudega serveritevahelise andmevahetuse loogilist tausta, võrrelda implementeerimise võimalusi *DataStream*'i ja *ObjectStream*'i näitel ning anda hinnang erinevate implementatsioonide tugevustele, nõrkustele ja sobilikele kasutuskohtadele. Töö esimene osa uurib ja selgitab Java andmevoogude taga olevat loogikat ja levinud andmevoogudega suhtluse implementeerimise viise asjakohase kirjanduse põhjal. Töö teine osa uurib praktiliselt kolme erinevat serveritevahelise andmevoogudega andmevahetuse implementatsiooni, millest kaks tükki kasutavad *DataStream*'i ja kolmas *ObjectStream*'i loogikat. Töö kolmandas osas luuakse keskkond, mis võimaldab valida erinevate andmevoogudega andmevahetuse implementatsioonide vahel ja simuleerida pidevat andmevahetust erinevate serverite vahel, ning võrreldakse töö teises osas loodud andmevoogudega andmevahetuse implementatsioone. Java programmeerimiskeele andmevoogusid analüüsivaid uurimistöid Tartu Ülikooli arvutiteaduse instituudis varem tehtud ei ole, mistõttu on käesoleva töö

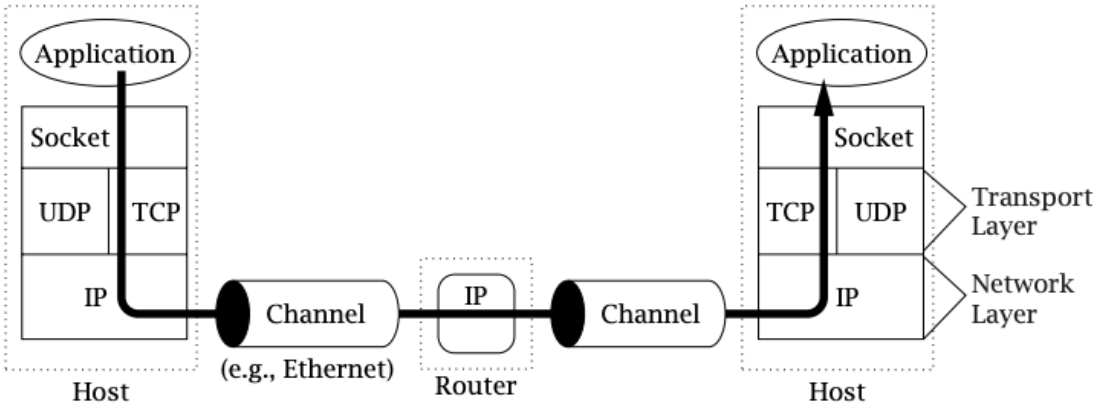
lisaeesmärgiks andmevoogude loogika tutvustusega olla aluseks järgnevatele selle-teemalistele uurimustele.

2. Andmevood

Andmevoogusid kasutava suhtluse implementeerimiseks peab suhtlust läbiviiva rakenduse arendaja välja mõtlema lahenduse, et andmevoogu turvaliselt ja efektiivselt saata ja vastu võtta [2]. Sellise rakendusepoolse lahenduse jaoks mitu erinevat võimalust, millest üheks on kasutada Java klasse *InputStream* ja *OutputStream* [3].

2.1 Soklid ja võrguprotokollid

Kui kahe serveri vahel alustatakse andmevahetust, siis mõlemas serveris avatakse selle andmevahetusega tegelev sokkel. Üks server võib avada mitu erinevat soklit, tagades seeläbi võimekuse suhelda samaaegselt mitme erineva serveriga ja nende serverite sees omakorda erinevate rakendustega [1].



Joonis 1. Suhtlus kahe eraldiseisva masina peal jooksva rakenduse vahel kasutades TCP/IP protokolli [4]

Järgnev lõik tugineb Calverti ja Donahoo [4] selgitusele edukat ülevõrgulist andmevahendust tagavatest protokollidest: IP ehk internetiprotokoll ja TCP ehk edastusohje protokoll või UDP ehk kasutajadatagrammi protokoll. Kui IP-d on vaja selleks, et andmed jõuaksid andmete saatja võrgumasinast õigesse andmete saaja võrguseadmesse, siis TCP-d või UDP-d on vaja selleks, et andmevahetust täpsustada rakenduse tasemeni. Need protokollid annavad masinatele teada millisest kindlast rakendusest andmed tulevad ja millisesse rakendusse nad jõudma peavad. TCP ja UDP peamine vahe seisneb selles, et TCP kontrollib, et andmete saaja poolt kättesaadud andmed oleksid samal kujul, mis nad olid saatmise hetkel, ehk et andmete vahendamisel saatjamasinast saajamasinasse ei ole andmetes tekkinud vigu. Vea korral üritab TCP vastuvõetud andmeid korrigeerida või pärib vähemalt osaliselt andmete uuestisaatmist. UDP sellist kontrolli ja korrigeerimist ei soorita.

Sokkel vahendab andmeid rakenduskihil jooksva rakenduse ja edastusohje protokolli kasutava transpordikihi vahel. Joonis 1 visualiseerib ühest võrgumasina rakendusest teise võrgumasina rakendusse saadetud andmevoogu kasutades sokleid, TCP-d, IP-d ja internetti.

2.2 *OutputStream* ja *InputStream*

Javas vastab ühele soklile üks *Socket* klassi isend, mille meetodid *getOutputStream* ja *getInputStream* võimaldavad kätte saada soklile vastava väljamineva ja sissetuleva andmevoo *OutputStream* ja *InputStream* klassi isendite näol [1]. *OutputStream* on vajalik rakendusest väljaminevasse andmevoogu andmete kirjutamiseks ja *InputStream* on vajalik rakendusse sissetulevast andmevoost andmete lugemiseks. *OutputStream* ja *InputStream* on Java sisend/väljund-programmeerimise alusklassid, läbi mille kõikide andmevoogude edastamine käib [5].

Järgnev lõik tugineb Java juhendile [6] klassist *OutputStream*. Klassil *OutputStream* on andmete kirjutamiseks põhiliselt kaks meetodit, millest *write(int b)* kirjutab andmevoogu ühe baidi, ja *write(byte[] data)* mõeldud baidijärjendite andmevoogu kirjutamiseks. Klassil *OutputStream* on lisaks meetodid *close()* ja *flush()*, millest esimene on mõeldud andmevoo sulgemiseks ja teine kirjutab kõik puhverdatud baidid andmevoogu.

Järgnev lõik tugineb Java juhendile [7] klassist *InputStream*. Peale andmete andmevoogu kirjutamist on *InputStream* esimene klass andmeid vastuvõtva serveri pool, mis peab edastatud andmed andmevoost kätte saama. Sarnaselt *OutputStream* klassile on *InputStream*'il meetodid nii ühe baidi kui ka mitmest baidist koosneva järjendi sisselugemiseks - nendeks on meetodid *read()* ja *read(byte[] b)*.

InputStream klass on olemuselt keerulisem kui *OutputStream*. See on tingitud sellest, et andmete andmevoogu kirjutamisel teab Java täpselt kui palju andmeid kokku on vaja kirjutada ja voogu kirjutamisel ei saa tekkida andmete kadu, kuid andmete sisselugemisel peab Java pidevalt kontrollima kas ja kui palju baite on andmevoos *InputStream*'i jaoks lugemiseks kättesaadaval.

Klassid *InputStream* ja *OutputStream* on abstraktsed, mis tähendab, et nende klasside funktsionaalsuse kasutamiseks tuleb luua neid klasse realiseerivad alamklassid [8]. Klassi *OutputStream* alamklass peab üle kirjutama klassi meetodi *write()* ja klassi *InputStream* alamklass peab üle kirjutama klassi meetodi *read()* [6, 7].

Klassid *InputStream* ja *OutputStream* on põhilised klassid, millele kahe serveri vaheline andmevoogudega andmevahetus põhineb, kuid Javas on nendele klassidele mitmeid alamklasse ja omakorda alamklasside alamklasse, mis alusklasside tööd ühel või teisel moel täiendavad. Need alamklassid on vajalikud, sest need võimaldavad kohandada andmevoogudega andmevahetust mingi konkreetse süsteemi vajadustele vastavaks, aitavad saavutada igale olukorrale kõige efektiivsema andmevahetuse ja tagavad võimalikult lihtsa andmevahetust toetava tarkvara arendamise protsessi.

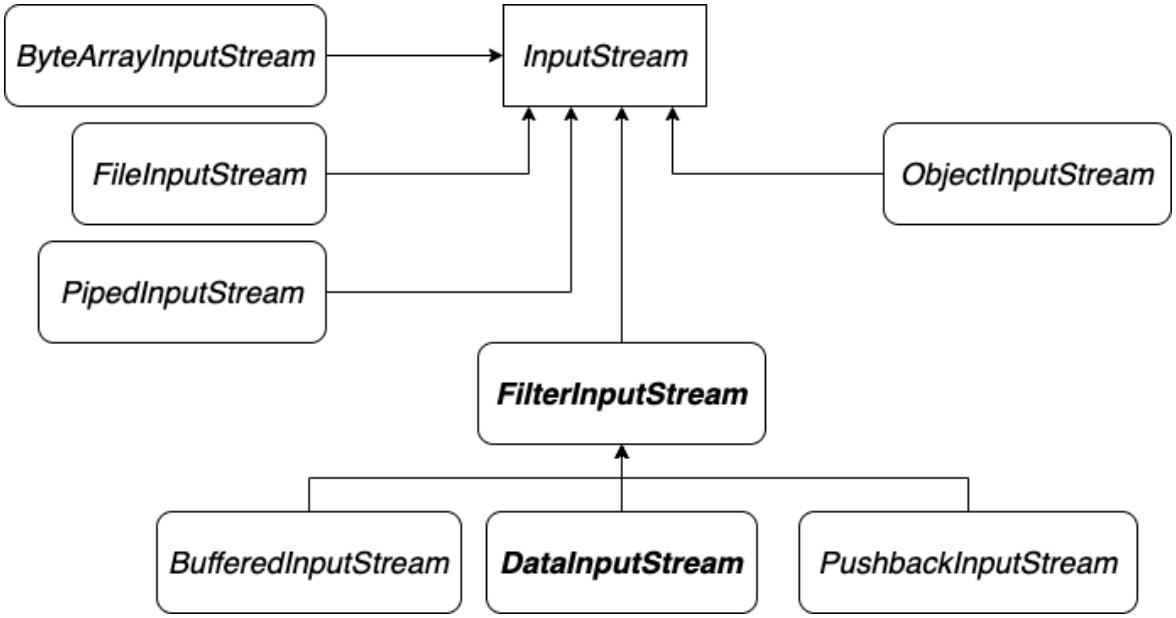
2.3 *DataOutputStream* ja *DataInputStream*

Tüüpiliseks probleemiks tarkvaraarenduses on, et kahe serveri vahel on vaja saata ja vastu võtta mingisugust andmestruktuuri. Andmestruktuur võib koosneda erinevatest Java objektidest, mille sees olevad andmed taanduvad kõik lõpuks Java primitiivsetele andmetüüpidele. Üheks võimaluseks sellist andmestruktuuri ühest serverist teise saatmiseks on kasutades klassipaari *DataInputStream* ja *DataOutputStream*.

DataInputStream ja *DataOutputStream* võimaldavad andmevoogu kirjutada nii Java primitiivseid andmetüüpe kui ka sõnesid. Tänu sellele saab andmevoogu kirjutada järjest ja vaheldumisi erinevatest andmetüüpidest koosnevat andmete tervikut [5]. See omakorda võimaldab Java programmeerimiskeeles kirjutaval tarkvaraarendajal kirjutada andmeid andmevoogu ilma neid ise käsitsi baitideks teisendamata. See on oluline, sest modernsetes infotehnoloogilistes süsteemides kasutatakse andmete hoidmiseks erinevaid andmetüüpe, mille teisendamine baitideks nii-öelda käsitsi võib olla äärmiselt ajakulukas protsess.

Harold [5] selgitab, et *DataInputStream* ja *DataOutputStream* klasside otsesed ülemklassid on *FilterInputStream* ja *FilterOutputStream*, mis omakorda on klasside *InputStream* ja *OutputStream* alamklassid. *FilterInputStream* ja *FilterOutputStream* on klassid, mis hoiavad endas vastavalt sisend- ja väljundvoogu *InputStream* ja *OutputStream* klasside isendite näol. Samuti kirjutavad need klassid üle vastavalt *InputStream*'i ja *OutputStream*'i kõik meetodid. Ülekirjutatud meetodid ei lisa juurde funktsionaalsust, vaid lihtsalt suunavad kõik meetodi väljakutsed omakorda klassi sees isendina hoitud andmevoole. *FilterInputStream* ja *FilterOutputStream* klasse iseseisvalt ei kasutata, kuid need klassid on ülemklassideks kõikidele teistele *InputStream*'i ja *OutputStream*'i alamklassidele, mis ühel või teisel moel endas hoitud andmevoogu muudavad. Üheks selliseks *FilterInputStream*'i ja *FilterOutputStream*'i alamklasside paariks on *DataInputStream* ja *DataOutputStream*. Joonis 2 illustreerib klasside *FilterInputStream* ja *DataInputStream* asukohta sagedasti

kasutatud sisendandmevoogude klasside hierarhias, kusjuures klasside *FilterOutputStream* ja *DataOutputStream* asukoht sagedasti kasutatud väljundandmevoogude klasside hierarhias on analoogne [9].



Joonis 2. Klasside *FilterInputStream* ja *DataInputStream* asukohad teiste sagedasti kasutatud sisendandmevoo klasside hierarhias [9]

Allikate [10, 11] järgi on klassil *DataOutputStream* andmevoogu andmete kirjutamiseks mitu erinevad meetodit. Kõik selle klassi erinevad kirjutamise meetodid kasutavad sisemiselt sama klassi meetodit *write(int b)*, mis kirjutab andmevoogu ühe baidi. Teised kirjutamise meetodid võtavad argumendina erinevat tüüpi sisendeid, mille nad tõlgendavad endas defineeritud loogika kohaselt baitideks ja annavad andmevoogu kirjutamiseks edasi meetodile *write(int b)*. Klassil *DataOutputStream* on andmete kirjutamiseks andmevoogu meetodid *writeBoolean(boolean v)*, *writeByte(int v)*, *writeChar(int v)*, *writeDouble(double v)*, *writeFloat(float v)*, *writeInt(int v)*, *writeLong(long v)*, *writeShort(int v)*, mis kõik on mõeldud meetodi nimes defineeritud primitiivse andmetüübi kirjutamiseks andmevoogu. Lisaks on klassil *DataOutputStream* meetodid *writeBytes(String s)*, *writeChars(String s)*, *writeUTF(String str)*, mis kõik võtavad argumendina sisse sõne, kuid erinevad andmevoogu kirjutamise loogika poolest. Meetodid *writeBytes(String s)* ja *writeChars(String s)* kirjutavad sõnesid andmevoogu sarnaselt - mõlemad võtavad sõnest järjest tähemärke ja tõlgendavad need andmevoogu kirjutamiseks baitideks - erinevus tuleb aga sellest, et *writeBytes(String s)* tõlgendab ühe tähemärgi üheks baidiks, kuid *writeChars(String s)*

tõlgendab ühe tähemärgi kaheks baidiks. Need kaks meetodit ei kirjuta andmevoogu muud informatsiooni (nagu näiteks sisendi pikkus) andmevoogu ja nendele meetoditele ei ole klassis *DataInputStream* analoogseid lugemise meetodeid *readBytes()* ja *readChars()* [12, 13]. Meetod *writeUTF(String s)* tõlgendab tähemärke baitideks UTF-8 kodeeringu järgi. Ühele tähemärgile võib vastata üks, kaks või kolm baiti. Lisaks tähemärkidele kirjutab meetod *writeUTF(String s)* andmevoogu kogu kirjutatava sõne pikkuse baitides. Seda tehakse enne sõne baitide kirjutamist ja see võimaldab klassis *DataInputStream* meetodi *readUTF()* olemasolu.

Sarnaselt klassile *DataOutputStream* on allikate [12, 13] järgi klassis *DataInputStream* andmevoost andmete lugemiseks mitu erinevat meetodit, mis vastanduvad klassi *DataOutputStream* andmete kirjutamise meetoditele. Kõik selle klassi andmevoost andmete lugemise meetodid kasutavad sisemiselt sama klassi meetodit *read(byte[] b)*, mis loeb andmevoost nii palju baite kui meetodile argumendina antud baidijärjendisse b mahub. Teised lugemise meetodid tõlgendavad loetud baidijärjendi endas kirjeldatud loogika kohaselt mingi teise primitiivi või sõne kujule.

Allikas [1] selgitab, et igat andmestruktuuri on teoreetiliselt võimalik üle võrgu saata nii, et andmeid edastav server kirjutab kõik andmestruktuuris olevad identifikaatorid ja väärtused ühte suurde sõnesse ja kirjutab selle sõne baitide järjendina *DataOutputStream*'i kaudu *OutputStream*'i andmevoogu. Andmeid vastuvõttev server saab seejärel *DataInputStream*'i kaudu *InputStream*'ilt sisseloetud sõnes olevatest andmetest konstrueerida uuesti andmestruktuuri algses formaadis. Kuid see meetod ei ole optimaalne, sest lisaks andmestruktuuri kirjeldavatele väärtustele on andmevoogu vaja kirjutada andmete eraldamiseks vajalikke lisaandmeid, ja keerulisemate andmestruktuuride puhul, kus erinevad objektid hoiavad enda sees omakorda teisi objekte, võib andmete mahutamine eraldatult ühte sõnesse kujuneda väga keeruliseks ülesandeks.

2.4 *ObjectOutputStream* ja *ObjectInputStream*

Teiseks võimaluseks andmestruktuure läbi andmevoo vahendada on kasutades klasse *ObjectOutputStream* ja *ObjectInputStream*. Nende klasside meetodid *writeObject(Object obj)* ja *readObject()* võimaldavad andmevoogu kirjutada terviklikke Java objekte [5]. See tähendab, et mingisuguse andmestruktuuri saatmiseks läbi andmevoo ei pea selle arendaja seda objekti ise primitiivseteks andmetüüpideks või sõnedeks dekomponeerima, vaid saab andmevoogu kirjutada mingi terve objekti ühe klassi *ObjectOutputStream* meetodi

writeObject(Object obj) väljakutsega ja selle sama objekti saab andmevoost lugeda ühe klassi *ObjectInputStream* meetodi *readObject()* väljakutsega.

Allikas [9] selgitab, et tervete objektide kirjutamiseks andmevoogu kasutab Java jadastamist ehk serialiseerimist. Mingi Java objekti jadastamine tähendab, et selle objekti olek esitatakse baidijadana. Selle baidijada saab siis kirjutada andmevoogu. Andmevoost baidijada lugemisel tuleb rakendada objektimise ehk deserialiseerimise tehnikat, mis on vastand jadastamisele, ja võimaldab jadastamisega saadud objekti baidijadast konstrueerida uuesti algse objekti isendit. Konkreetse Java klassi andmete serialiseerimine on võimalik ainult siis, kui see klass seadmestab Java *Serializable* liidest. Liidese *Serializable* seadmestamine ei nõua arendajalt ühegi liidese *Serializable* meetodi implementeerimist.

Allikatest [14, 15] näeb, et kasutades klassipaari *ObjectOutputStream* ja *ObjectInputStream* ei pea arendaja ise implementeerima andmevoogu kirjutatava objekti jadastamise protsessi. Kutsudes klassi *ObjectOutputStream* meetodit *writeObject(Object obj)* jadastatakse objekt *obj* vaikimisi Java poolt määratud jadastamise meetodiga baidijadaks ja kirjutatakse siis andmevoogu *OutputStream*. Objekti lugemisel andmevoost käitub Java analoogselt - klassi *ObjectInputStream* meetod *readObject()* loeb andmevoost *InputStream* jadastatud objekti baidijada ja deserialiseerib baidijada algse objekti isendiks.

2.5 Puhverdamine

Järgnev lõik tugineb Tartu Ülikooli õppematerjalile [16], millest näeb, et andmete lugemisel andmevoost on mõistlik kasutada puhverdamise tehnikat. Olenevalt lähenemisest võib andmevoost andmete kirjutamise ja lugemise meetodite väljakutseid olla palju. Puhverdamise kasutamine muudab kirjutamise ja lugemise efektiivsemaks, sest siis tehakse otseseid kirjutamise ja lugemise pöördumisi andmevoo poole alles siis kui etteantud suurusega puhver saab täis. Andmevoogu andmete kirjutamise ja sealt lugemise protsessid on suhteliselt ajakulukad tegevused ja nende vähendamine aitab parandada andmevoogudega suhtluse jõudlust.

```
new DataOutputStream(new BufferedOutputStream(socket.getOutputStream()))
```

Joonis 3. *DataStream*'i kasutava puhverdatud väljundandmevoo loomine Javas

Java programmeerimiskeeles on andmevoogude puhverdamiseks sobilikud klassid *BufferedInputStream* ja *BufferedOutputStream* [16]. Puhverdamise lisamiseks andmevoole saab nii andmete sisend- kui väljundvoogu ümbritseda vastava puhverdamise klassi

isendiga. Samuti saab puhverdatud andmevoogu omakorda ümbritseda *DataStream*'i või *ObjectStream*'i klassi isenditega, mis annab puhverdatud andmevoole vastava klassi funktsionaalsuse. Näide puhverdatud *DataStream*'i kasutavast väljundandmevoo loomisest on toodud joonisel 3. Näide on analoogne sisend- ja väljundvoogudel nii *DataStream*'i kui *ObjectStream*'i puhul.

3. Andmevoogudega andmevahetuse erinevad implementatsioonid

Serveritevahelise andmevoogudega andmevahetuse implementeerimiseks on erinevaid võimalusi. Käesoleva töö raames uuritakse *DataStream*'i ja *ObjectStream*'i kasutavaid lahendusi. See, kuidas andmevoogudega andmevahetust mingis rakenduses implementeeritakse, oleneb suuresti sellest, millist andmestruktuuri parasjagu on vaja saata ja vastu võtta - seetõttu on oluline, et erinevate kasutatud lahenduste võrdlemiseks oleks need lahendused implementeeritud sama struktuuriga andmete vahendamiseks.

Käesolevas peatükis defineeritakse implementatsioonide võrdluseks kasutatav andmestruktuur ja implementeeritakse selle andmestruktuuri saatmiseks ja vastuvõtmiseks andmevoo teel erinevad *DataStream*'i ja *ObjectStream*'i kasutavad lahendused. Nii *DataStream*'i kui *ObjectStream*'i lahendus kasutab puhverdamist läbi *BufferedInputStream* ja *BufferedOutputStream* klassi isendite, et saavutada mõlema implementatsiooni jaoks võimalikult hea jõudlus.

3.1 Implementatsioonide võrdluseks kasutatav andmestruktuur

Implementatsioonide võrdluseks kasutatav andmestruktuur on alati sama üldise struktuuriga, kuid selle andmestruktuuri erinevate isendite andmetes esineb väikseid numbrilisi või sõnelisi erinevusi. See iseloomustab olukorda, kus andmeid genereerivad serverid sooritavad sarnaseid operatsioone ja seetõttu genereerivad sama struktuuriga andmeid, kuid operatsioonide käigus tekkivad andmed on erinevate väärtustega, mis võib olla mingisuguse operatsiooni tulemusi kirjeldava statistika arvutamise aluseks.

```
List<DataUnit> data
```

Joonis 4. Andmeühikute järjend *data*

```
class DataUnit {

    long id;
    String idString;
    long time;
    long transaction;

    ...
}
```

Joonis 5. Andmeühikut esindava klassi *DataUnit* isendiväljade deklaratsioonid

Olgu uuritavaks andmehulgaks järjend nimega *data* (vt joonis 4). Antud järjend koosneb andmeühikutest, mida esindavad klassi *DataUnit* isendid (edaspidi andmeühikud). Andmeühikul on neli isendivälja: *id, idString, time* ja *transaction* (vt joonis 5). Väli *id* esindagu mingit numbrilist identifikaatorit, väli *idString* esindagu mingit sõnelist identifikaatorit, väli *time* esindagu andmeühiku genereerimise UNIX-i ajatemplit numbriliselt ja väli *transaction* esindagu mingisugust numbrilist väärtust. Sellised väljad on valitud, et simuleerida olukorda, kus andmed koosnevad nii sõnelistest kui numbrilistest väärtustest. Väljade loogiline sisu ei mängi andmete edastamise meetodite võrdlemise jaoks rolli, kuid väljad ja nende sisu on siiski valitud selliselt, et esindada mingit teoreetiliselt võimalikku struktuuri - andmeühiku keskseks informatsiooniosaks on mingi numbriline väärtus *transaction* ja selle andmeühiku tekkimise aeg *time*, andmeühikut eristavad teistest andmeühikutest identifikaatorid *id* ja *idString*.

```
class DataUnit {

    ...

    public void initializeWithRandomData() {
        id = generateId();
        idString = generateIdString();
        time = generateTimestamp();
        transaction = generateTransaction();
    }

    ...
}
```

Joonis 6. Klassi *DataUnit* isendimeetod *initializeWithRandomData()*, mis initsialiseerib klassi isendi isendiväljad suvaliselt genereeritud andmetega.

Klassis *DataUnit* on meetod *initializeWithRandomData()* kõikide enda isendiväljade initsialiseerimiseks suvaliste andmetega (vt joonis 6). Antud meetodit klassi isendil välja kutsudes saab väli *id* väärtuseks suvalise *long* tüüpi väärtuse, väli *idString* saab väärtuseks kümnest suvalisest tähemärgist koosneva sõne, väli *time* saab väärtuseks väljale väärtuse saamise aja kasutades *java.lang* paki meetodit *System.currentTimeMillis()* ja väli *transaction* saab väärtuseks suvalise *long* tüüpi väärtuse vahemikus -1,000,000 kuni 1,000,000. Suvaliste numbrite genereerimiseks on kasutatud *java.util.concurrent* paki meetodeid *ThreadLocalRandom.current().nextLong()* ja *ThreadLocalRandom.current()-.nextInt()*.

3.2 *DataStream*'i implementatsioon

Klasse *DataOutputStream* ja *DataInputStream* teoreetiliselt uurivas peatükis on näha, et antud klassipaari kasutades on võimalik andmevoogu andmeid kirjutada nii primitiividena kui ka sõnedena. See tähendab, et *DataUnit* klassi isenditest koosneva järjendi kirjutamiseks ja lugemiseks andmevoost kasutades klassipaari *DataOutputStream* ja *DataInputStream* on kaks erinevat võimalust. Esimene võimalus on järjend kirjutada andmevoogu ühe tervikliku sõnena ja lugeda see andmevoost samuti ühe tervikliku sõnena. Teine võimalus on andmevoogu kirjutada järjest eraldi kõik järjendi alamväärtused primitiivide ja sõnede tasemel -

nendeks alamväärtusteks on *DataUnit* klassi isenditest koosneva järjendi puhul *DataUnit* klassi isendi muutujate *id*, *idString*, *time* ja *transaction* väärtused.

Lisaks kahe meetodi eraldi kasutamisele on võimalik rakendada ka nende kahe meetodi hübriidi, kus mingi osa andmetest komplekteeritakse ühte sõnesse ja mingi osa andmetest kirjutatakse eraldiseisvate primitiividena. Käesolevas töös ei käsitleta sellist hübriid-lahendust kui eraldiseisvat lahendust, sest loogika poolest läheneb ta suuresti esimesele meetodile ja on vaadeldav kui selle meetodi optimisatsioon.

3.3 *DataStream*'i ühe sõne implementatsioon

Andmestruktuuri saatmiseks ühest serverist teise üle *DataStream*'i ühe sõnena on vaja saadetav andmestruktuur mahutada ühte sõnesse. Kuna tekitatav sõne on vaja andmeid vastuvõtva serveri pool tõlgendada uuesti esialgse andmestruktuuri kujule, siis mida keerulisem on saadetava andmestruktuuri ülesehitus, seda keerulisemaks kujuneb andmete mahutamine ühte sõnesse. Et andmevahetus oleks võimalikult efektiivne on vaja sõne panna andmetest kokku kasutades võimalikult vähe lisatähemärke. Samas tuleb kõik eraldiseisvad andmeühikud üksteisest eraldada selgelt ja üheselt, et andmeid vastuvõttev server oskaks vastuvõetud sõnest lugeda välja eraldiseisvaid andmeühikuid.

```
private void writeDataWithDataStreamAsOneString(List<DataUnit> data) throws IOException {
    //Andmete eeltöötlus
    StringJoiner dataStringJoiner = new StringJoiner(";");
    for (DataUnit dataPiece : data) {
        dataStringJoiner.add(dataPiece.toString());
    }
    String dataString = dataStringJoiner.toString();
    //Kirjutame andmed andmevoogu
    DataOutputStream os = (DataOutputStream) outputStream;
    os.writeUTF(dataString);
}
```

Joonis 7. Meetod andmete kirjutamiseks andmevoogu ühe sõnena kasutades *DataStream*'i

Kuna uuritavaks andmestruktuuriks on *DataUnit* klassi isenditest koosnev järjend, siis on vaja leida sõneline esituse vorm esiteks *DataUnit* klassi isendi informatsioonile ja lisaks järjendi elementide eraldaja sõnes. *DataUnit* isendi sõneliseks esitamiseks on mugav kasutada võimalikult efektiivset *toString()* meetodi implementatsiooni. Kõige vähem lisatähemärke kasutatakse siis, kui isendiväljade väärtused kirjutakse järjest sõnesse eraldades neid mingi tähemärgiga, mida ühegi isendivälja väärtuses ei esine. *DataUnit* klassil on vaid üks sõneline isendiväli *idString*. Kuna *idString* väärtus koosneb ainult tähtedest ja numbritest, siis sobib eraldajaks näiteks tähemärk "," (koma). Järgnevalt on vaja saadetavas sõnes eraldada üksteisest erinevad *DataUnit* klassi isendid. See tähendab, et on

vaja leida eraldav tähemärk, mida ei esine *DataUnit* isendi sõnelises esituses. Selleks sobib näiteks tähemärk ";" (semikoolon). Andmestruktuuri saab andmevoogu seega kirjutada meetodiga, mis võtab argumendina kirjutatava andmestruktuuri, tõlgendab antud andmestruktuuri üheks sõneks ja kirjutab andmed vastuvõtva serveri andmevoogu (vt joonis 7).

```
private List<DataUnit> readDataWithDataStreamAsString(InputStream inputStream) throws IOException {
    //Loeme andmevoost andmed ühe sõnena
    DataInputStream is = (DataInputStream) inputStream;
    String dataString = is.readUTF();
    //Tõlgendame sõne esialgse andmestruktuuri kujule
    List<DataUnit> data = new ArrayList<>();
    for (String dataUnitString : dataString.split(";")) {
        String[] dataUnitChunks = dataUnitString.split(",");
        long id = Long.parseLong(dataUnitChunks[0]);
        String idString = dataUnitChunks[1];
        long time = Long.parseLong(dataUnitChunks[2]);
        long transaction = Long.parseLong(dataUnitChunks[3]);
        data.add(new DataUnit(id, idString, time, transaction));
    }
    return data;
}
```

Joonis 8. Meetod andmete lugemiseks andmevoost ühe sõnena kasutades *DataStream*'i

Saadetavat sõne vastu võttes tuleb lugeda andmevoost sinna kirjutatud sõne, jagada sõne *DataUnit* isendeid kirjeldavateks sõneosadeks ja lugeda igast sõneosast välja ühte *DataUnit* klassi isendit kirjeldavad isendiväljade väärtused (vt joonis 8). Andmeid lugev meetod võtab argumendina andmeid saatva serveri andmevoo viite.

3.4 *DataStream*'i eraldiseisvate primitiivide ja sõnede implementatsioon

```
private void writeDataWithDataStreamAsChunks(List<DataUnit> data) throws IOException {
    //Kirjutame andmed andmevoogu
    DataOutputStream os = (DataOutputStream) outputStream;
    os.writeInt(data.size());
    for (DataUnit dataPiece : data) {
        os.writeLong(dataPiece.id);
        os.writeUTF(dataPiece.idString);
        os.writeLong(dataPiece.time);
        os.writeLong(dataPiece.transaction);
    }
}
```

Joonis 9. Meetod andmete kirjutamiseks andmevoogu eraldiseisvate primitiivide ja sõnedena kasutades *DataStream*'i

Andmestruktuuri saatmisel eraldiseisvate primitiivide ja sõnedena ei ole andmeid vaja ümber töödelda teisele kujule. *DataUnit* klassi isenditest koosneva järjendi kirjutamisel saab andmevoogu kirjutada lihtsalt järjest kõiki isendeid kirjeldavate isendiväljade väärtused. Et andmeid vastu võttev server teaks, kui palju andmeid mingi loetava struktuuri jaoks

andmevoost lugema peab, kirjutab andmeid saatev server enne andmestruktuuri andmevoogu saadetava andmestruktuuri suuruse. *DataUnit* klassi isenditest koosneva järjendi puhul on selleks suuruseks järjendi pikkus. (vt joonis 9)

```
private List<DataUnit> readDataWithDataStreamAsChunks(InputStream inputStream) throws IOException {
    //Loeme andmevoost andmed ja tõlgendame esialgse andmestruktuuri kujule
    DataInputStream is = ((DataInputStream) inputStream);
    List<DataUnit> data = new ArrayList<>();
    int dataLength = is.readInt();
    for (int i = 0; i < dataLength; i++) {
        long id = is.readLong();
        String idString = is.readUTF();
        long time = is.readLong();
        long transaction = is.readLong();
        data.add(new DataUnit(id, idString, time, transaction));
    }
    return data;
}
```

Joonis 10. Meetod andmete lugemiseks andmevoost eraldiseisvate primitiivide ja sõnedena kasutades *DataStream*'i

Andmeid lugedes tuleb esimesena lugeda saadetud andmete hulk. Seejärel saab andmevoost lugeda tsükliga kõik saadetud andmed eraldiseisvate primitiivide ja sõnedena. Kui ühte *DataUnit* klassi isendit kirjeldavad isendiväljade väärtused on loetud, siis saab luua uue *DataUnit* isendi, mis on samal kujul kui enne saatmist. (vt joonis 10)

3.5 *ObjectStream*'i implementatsioon

```
private void writeDataWithObjectStream(List<DataUnit> data) throws IOException {
    //Kirjutame andmed andmevoogu
    ObjectOutputStream os = (ObjectOutputStream) outputStream;
    os.writeObject(data);
}
```

Joonis 11. Meetod andmete kirjutamiseks andmevoogu kasutades *ObjectStream*'i

Klasse *ObjectOutputStream* ja *ObjectInputStream* kirjeldavates peatükkides on näha, et *ObjectStream*'i kasutav andmevoogudega suhtlemise lahendus võimaldab andmevoogu kirjutada jadastatud kujul otse terviklikke objekte. Selle jaoks on vajalik, et saadetav objekt seadmestaks *Serializable* liidest. Kui saadetav objekt hõlmab endas teisi objekte, siis peavad antud liidest seadmestama ka nende objektide klassid. Näite puhul, kus saadetav andmestruktuur on *DataUnit* isenditest koosnev *ArrayList* klassi isend, tuleb seadmestamine lisada vaid *DataUnit* klassile endale, sest *ArrayList* klassi implementatsioon juba seadmestab seda [17]. Peale *Serializable* liidese seadmestamise ei pea *ObjectStream*'iga andmete saatmisel tegema andmetel muid eeltegevusi ja peale andmevoo avamist saab saadetava objekti ühe meetodi väljakutsega andmevoogu kirjutada (vt joonis 11).

```
private List<DataUnit> readDataWithObjectStream(InputStream inputStream) throws IOException, ClassNotFoundException {
    //Loeme andmevoost andmed
    ObjectInputStream is = (ObjectInputStream) inputStream;
    Object o = is.readObject();
    //Muudame andmed algsele kujule
    List<DataUnit> data = (List<DataUnit>) o;
    return data;
}
```

Joonis 12. Meetod andmete lugemiseks andmevoost kasutades *ObjectStream*'i

Andmevoost saab saadetud objekti lugeda sarnaselt kirjutamisele ühe meetodi väljakutsega. Lugemine tagastab objekti klassi *Object* isendina ja see tuleb konverteerida tagasi algsele *List<DataUnit>* kujule. (vt joonis 12)

4. Andmevoogudega andmevahetuse erinevate implementatsioonide võrdlus

Andmevoogudega andmevahetuse erinevate implementatsioonide võrdluseks luuakse keskkond, kus saab simuleerida reaalset andmevahetust nende implementatsioonidega. Olgu olukord, kus mingi infotehnoloogiline süsteem koosneb mitmest serverist. Neist serveritest kõik peale ühe (edaspidi tööserverid) täidavad mingit tööülesannet, mille produktiks on pidevalt tekkivad andmestruktuurid. Need tööserverid saadavad enda poolt genereeritud andmed ühele tsentraalsele serverile (edaspidi andmekogumisserver), mille ülesandeks on koguda andmeid tööserveritelt ja teostada nendel andmetel mingisuguseid edaspidisi operatsioone. Sellise põhimõttega infotehnoloogiline süsteem võib esineda mitme erineva eesmärgi täitmiseks - näiteks kui on vaja koguda statistikat mingisuguse tööserveri poolt tehtud operatsiooni kohta.

Eelkirjeldatud süsteemis on vaja tööserveri poolt genereeritud andmed saata tööserverist andmekogumisserverisse nii, et andmed on enne tööserveri poolt saatmise alustamist ja peale andmekogumisserveri pool andmete vastu võtmist samal kujul. Saates andmeid andmevoona on selle tulemuse saavutamiseks võimalik kasutada nii *DataStream*'i kui ka *ObjectStream*'i kasutavat implementatsiooni. Et teada saada, millised on nende lahenduste praktilised erinevused, simuleeritakse olukord, kus tööserver genereerib järjest andmeid, töötleb need andmevoogu kirjutamiseks vajalikule kujule ja kirjutab andmed andmevoogu kasutades kas *DataStream*'i või *ObjectStream*'i. Andmekogumisserver loeb kirjutatud andmed vastavalt läbi *DataStream*'i või *ObjectStream*'i sisse ja töötleb andmed tagasi algsele kujule. Seejärel võrreldakse erinevate implementatsioonide vahel andmevoogu kirjutamisele eelnevat andmete töötlemist, andmevoogu kirjutamist, andmevoost lugemist ja andmevoost loetud andmete töötlemist esialgsele kujule.

Loodud keskkond on tööga kaasas lisana (vt lisa 1).

4.1 Andmekogumisserver

Andmekogumisserver on server, mis peale käivitamist jääb ootama tööserverite poolt ajendatud võrguühendusi ja peale ühenduse loomist loeb tööserverite poolt saadetud andmeid. Andmekogumisserveri ülesandeks on lugeda andmevoost andmeid ja töödelda neid nii, et nad jõuaksid tagasi samale kujule mis nad olid enne tööserveri poolt andmevoogu kirjutamist.

Andmekogumisserver koosneb ühest klassist *MainServer*. Klassi meetod *start(...)* loob uue sokli, mis ootab üle võrgu ühenduvaid tööservereid. Kui tööserver ühendub, siis luuakse iga tööserveriga suhtlemiseks uus lõim. Lõimel jookseb meetod, mis loeb andmevoona tööserverilt saadetud andmeid ja muudab need tagasi algsele kujule. Antud meetodi implementatsioon määrab ära millise lahendusega andmevoost andmeid loetakse ja peab olema vaste tööserveri poolt andmevoogu andmete kirjutamise meetodile. Võimalikke implementatsioone kirjeldati varasemates, spetsiifiliselt *DataStream*'i või *ObjectStream*'i implementatsioone käsitlevates peatükkides.

4.2 Tööserver

Tööserver on server, mida esindab klass *WorkerServer*, ja mille tööd alustab sama klassi meetod *start(...)*. Peale käivitamist loob tööserver esimese asjana võrguühenduse andmekogumisserveriga. Kui ühendus on loodud, siis hakkab tööserver täitma töötsükleid. Iga töötsükli käigus genereeritakse varasemas peatükis kirjeldatud andmehulk *data* ja kirjutatakse see andmekogumisserveri andmevoogu. Andmevoogu kirjutamise meetodi erinevatest implementatsioonidest räägiti varasemates, spetsiifiliselt *DataStream*'i või *ObjectStream*'i implementatsioone käsitlevates peatükkides.

```
private static List<DataUnit> generateData() {
    List<DataUnit> data = new ArrayList<>();
    int maxAmount = 10;
    int minAmount = 5;
    int dataAmount = ThreadLocalRandom.current().nextInt(minAmount, maxAmount + 1);
    for (int i = 0; i < dataAmount; i++) {
        DataUnit dataUnitPiece = new DataUnit();
        dataUnitPiece.initializeWithRandomData();
        data.add(dataUnitPiece);
    }
    return data;
}
```

Joonis 13. Tööserveri andmete genereerimise meetod *generateData()*

Andmete genereerimiseks kasutab tööserver klassi *WorkerServer* meetodit *generateData()* (vt joonis 13). Meetod tagastab järjendi *DataUnit* klassi isenditest, millest kõik on algväärtustatud suvaliste andmetega kasutades *DataUnit* klassi meetodit *initializeWithRandomData()*.

4.3 Implementatsioonide võrdlus

Erinevate käsitletud lahenduste peamised erinevused implementatsioonis seisnevad järgmistes aspektides:

1. andmevoogu kirjutamisele eelnev andmete eeltöötlus;
2. andmevoogu andmete kirjutamise loogika;
3. andmevoost andmete lugemise loogika;
4. andmevoost loetud andmete töötlemine algsele kujule.

Põhjalikku andmete eeltööd uuritud andmestruktuuril nõudis vaadeldud lahendustest vaid *DataStream*'i kasutav ühe sõnena kirjutamise lahendus, kus tuli kogu andmestruktuur mahutada ühte sõnesse. Uuritud andmestruktuuri puhul ei esinenud sellisel eeltöötlusel suuri raskusi, kuid suuremate ja keerulisemate andmestruktuuride puhul muutub keerulisemaks ka uuritud andmestruktuuri puhul kirjeldatud sõnesiseste eraldajate leidmise ülesanne. Põhiline probleem seisneb andmestruktuuris olevate erinevate tasandite eraldamises nii, et algne struktuur oleks sõne hilisemal lugemisel ja töötlemisel taastatav. *DataStream*'i kasutav tükkidena saatmise lahendus ja *ObjectStream*'i kasutav lahendus analoogset andmete eeltöötlust ei vajanud ja on andmete eeltöötluse implementatsiooni aspekti poolest oluliselt lihtsamini implementeeritavad lahendused.

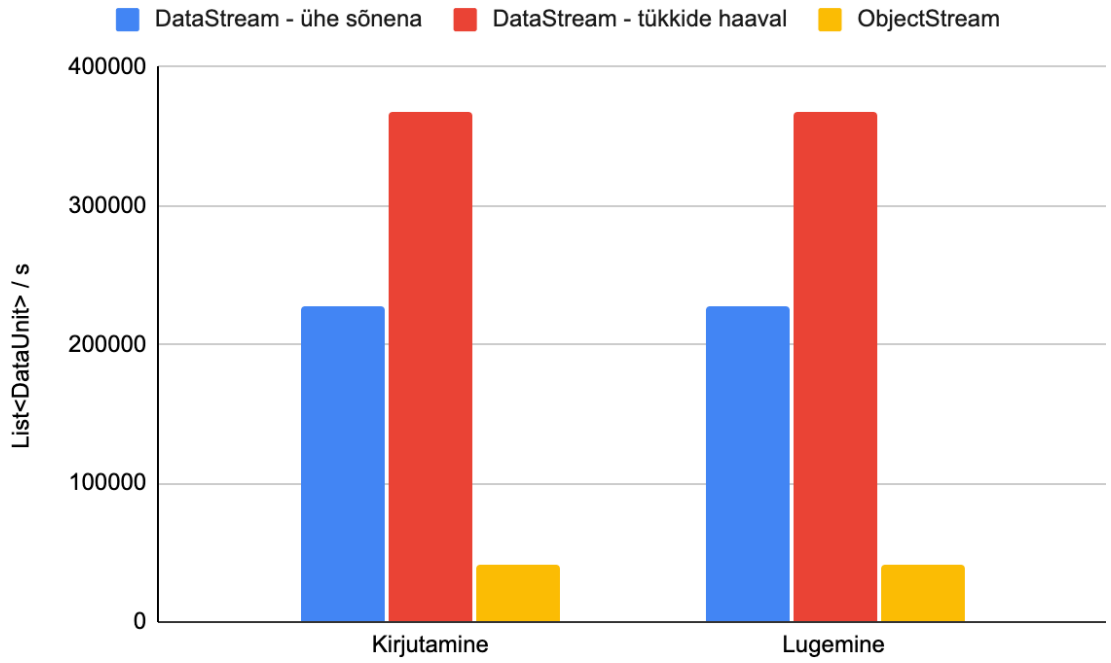
Andmevoogu andmete kirjutamise loogika poolest sarnanesid suuresti *DataStream'*i ühe sõnena kirjutamise lahendus ja *ObjectStream*'i lahendus. Mõlemal juhul koosnes andmevoogu kirjutamine ühest meetodi väljakutsest, vastavalt *DataOutputStream.writeUTF(String str)* ja *ObjectOutputStream.writeObject(Object obj).* Esimesel juhul kirjutati andmevoogu struktuuri *List<DataUnit>* esindav sõne ja teisel juhul kirjutati andmevoogu *List<DataUnit>* objekt jadastatud kujul. Kolmas, *DataStream*'i kasutav eraldisesivate primitiivide ja sõnedena saatmise lahendus nõudis keerulisemat andmevoogu andmete kirjutamise implementatsiooni. Esiteks, kuna genereeritud ja saadetud järjendi pikkus varieerus, siis oli tarvilik kirjutada andmevoogu esimese asjana saadetava järjendi pikkus, et andmeid vastuvõttev server teaks mitu tsükli kordust *DataUnit* isendit kirjeldavaid andmeid lugema peab. Teiseks tuli iga järjendi elemendi jaoks välja kutsuda andmevoogu kirjutamise meetodit neli korda - iga *DataUnit* isendivälja jaoks kord. Keerulisema ja suurema andmestruktuuri korral võib see protsess olla veelgi pikem: mida rohkem väärtusi saadetavas andmestruktuuris on, seda rohkem tuleb väljakutseid andmevoogu kirjutavatele meetoditele, ja kui saadetavas struktuuris oleks rekursiivselt rohkem varieeruva suurusega alamstruktuure (näiteks järjendeid), siis tuleks enne nende struktuuride kirjutamist kirjutada andmevoogu ka järgneva struktuuri pikkus (sarnaselt näites *List<DataUnit>* pikkuse kirjutamisele). Eriti keeruliseks ülesandeks kujuneks sõnastiku (näiteks *HashMap)* kirju-

tamine. Sellisel juhul tuleks andmevoogu kirjutada esiteks sõnastiku võtmete arv ja kõik võti-väärtus paarid, kirjutades esimesena andmevoogu võtme ja seejärel väärtuse.

Andmevoost lugemise loogika vastandub andmevoogu kirjutamise loogikale. *DataStream*'i ühe sõne lahenduse ja *ObjectStream*'i lahenduse puhul toimus andmete lugemine sarnaselt kirjutamisele ühe andmevoost lugemise meetodi väljakutsega ja *DataStream*'i tükkidena saatmise lahenduse puhul pidi andmed lugema ükshaaval samas järjekorras millega nad andmevoogu kirjutati. *DataStream*'i tükkidena saatmise lahenduse andmete lugemisel esinevad analoogsed raskused ja probleemid kui sama lahenduse kirjutamise protsessil.

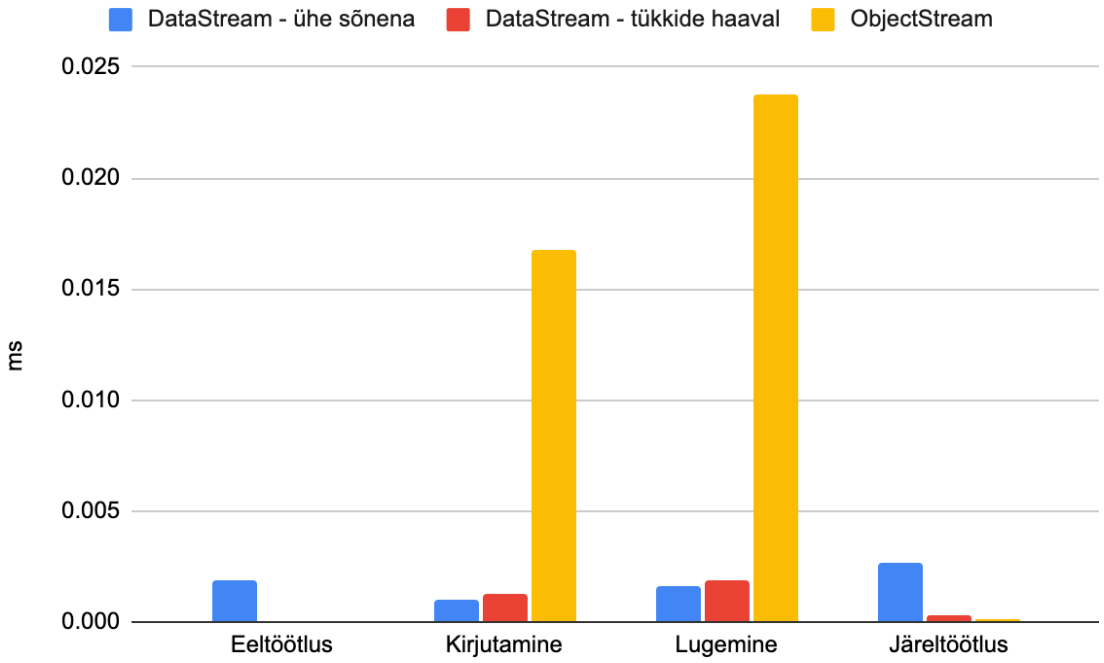
Kõige keerulisem lugemisjärgne andmete töötlemise protsess oli *DataStream*'i ühe sõne lahendusel. Selle lahenduse puhul tuli andmevoost loetud sõne kirjutamisel defineeritud eraldajate järgi tükeldada ja sõneosadest parsiti numbrilisi väärtusi. Tükeldatud väärtuste põhjal väärtustati uued *DataUnit* klassi isendid. *DataStream*'i tükkidena saatmise lahenduse puhul sai *DataUnit* klassi isendid väärtustada kohe peale andmete lugemist andmevoost ja ainsaks andmete järeltöötluseks oli *DataUnit* klassi isendite uuesti loomine ja andmete järjendisse lisamine. *ObjectStream*'i lahenduse puhul oli andmete järeltöötlus kõige lihtsam, sest ainukeseks vajalikuks tegevuseks oli andmevoost loetud *Object* tüüpi muutuja konverteerimine esialgsele *List<DataUnit>* kujule.

Lahenduste võrdluseks jõudluse poolest simuleeriti olukord, kus käivitati andmekogumisserver ja tööserver. Tööserver lõi ühenduse andmekogumisserveriga ja hakkas genereerima ning saatma andmeid. Andmekogumisserver luges andmevoost tööserveri poolt kirjutatud andmed ja tõlgendas need esialgsele kujule. Tööserver arvutas andmete kirjutamise käigus keskmise andmevoogu kirjutatud andmestruktuuride hulga sekundi kohta, keskmise eeltöötluse aja andmestruktuuri kohta millisekundites ja keskmise andmete kirjutamise aja andmestruktuuri kohta millisekundites. Andmekogumisserver arvutas andmete lugemise käigus keskmise andmevoost loetud andmestruktuuride hulga sekundi kohta, keskmise andmete lugemise aja andmestruktuuri kohta millisekundites ja keskmise andmete järeltöötlemise aja andmestruktuuri kohta millisekundites. Tööserveri ja andmekogumisserveri statistilised näitajad väljastati serverite töö ajal konsooli iga sekundi järel. Simulatsiooni jooksutati arvutil, millel on 2.6 GHz sagedusega kuuetuumaline Intel Core i7 protsessor ja 16 GB 2400 MHz sagedusega muutmälu.



Joonis 14. Erinevate andmevoogudega andmevahetuse implementatsioonide jõudluste võrdlus

Joonisel 14 on toodud simulatsiooni jõudluse tulemused. Ühikuteks on andmestruktuuride *List<DataUnit>* hulk sekundi kohta - vasakus klastris on kujutatud andmete kirjutamise kiirus ja paremas klastris andmete lugemise kiirus. Simulatsioonist on näha, et selgelt kõige parema jõudluse saavutas *DataStream*'i tükkide haaval saatmise lahendus, mis saavutas umbes 62% parema jõudluse kui *DataStream*'i ühe sõnena saatmise lahendus ja lausa 793% parema jõudluse kui *ObjectStream*'i kasutav lahendus.



Joonis 15. Erinevate andmevoogudega andmevahetuse implementatsioonide kirjutamise ja lugemise protsessi osadele kuluv keskmine aeg andmestruktuuri kohta

Joonisel 15 on toodud simulatsiooni käigus arvutatud erinevate kirjutamise ja lugemise protsessidele kuluvad keskmised ajad ühe saadetud andmestruktuuri kohta. Erinevateks andmevahetuse protsessi osadeks on loetud andmete eeltöötlus, andmete kirjutamine andmevoogu, andmete lugemine andmevoost ja andmete järeltöötlus. Eeltöötluse all mõeldakse andmete muutmist andmevoogu kirjutamiseks sobivale kujule, kirjutamise all mõeldakse otseseid andmevoogu kirjutamise meetodite väljakutseid, lugemise all mõeldakse otseseid andmevoost lugemise meetodite väljakutseid ja järeltöötluse all mõeldakse andmevoost loetud andmete muutmist algsele *List<DataUnit>* kujule. Joonisel 15 toodud tulemused toetavad joonisel 14 nähtud jõudluse tulemusi. On näha, et kahe parema jõudlusega meetodil oli andmevoogu kirjutamisele ja sealt lugemisele kuluv aeg suhteliselt võrdne - *DataStream*'i tükkidena saatmise lahendusel on vastavad ajad ainult veidi kõrgemad, kui *DataStream*'i ühe sõnena saatmise lahendusel. Sellest võib järeldada, et kirjutamise ja lugemise meetodite väljakutsete arv tõstab kirjutamisele ja lugemisele kulunud aega, kuid puhverdamist kasutades teeb seda minimaalselt. Jooniselt on tuletatav, et eelmainitud kahe meetodi suhteliselt suur jõudluse vahe tuleneb hoopis andmete eel- ja järeltöötlusele kulunud ajast, mis *DataStream*'i ühe sõne lahendusel oli palju suurem kui *DataStream*'i tükkidena saatmise lahendusel, mille puhul andmete eeltöötlust üldse vaja ei olnudki. Samuti on jooniselt selgelt näha, et *ObjectStream*'i kasutav lahendus vajab oluliselt

rohkem arvutusressurssi kui teised võrreldud lahendused, millest tuleneb joonisel 14 toodud jõudluse vahe teiste võrreldavatega.

Eelnevast lahenduste implementatsioonide analüüsist saab näha, et erinevatel lahendustel on erinevad tugevad ja nõrgad küljed. See tähendab, et erineva olukorra jaoks võivad olla optimaalsed erinevad andmevoogudega andmevahetuse implementatsioonid. Andmete eeltöötluse, kirjutamise, lugemise ja järeltöötluse poolest osutus kõige lihtsamini implementeeritavaks lahenduseks selgelt *ObjectStream*'i kasutav lahendus. Samas osutus see lahendus jõudluse poolest võrreldutest halvimaks valikuks. Eelnev tähendab, et *ObjectStream*'i kasutav lahendus on hea valik siis, kui jõudlus ei mängi erilist rolli või kui saadetavaid andmeid on oluliselt vähem kui näidissimulatsioonis. Kõige parema jõudluse saavutas *DataStream*'i tükkidena saatmise lahendus, ehk see on hea lahendus siis, kui jõudlus on kriitiline. *DataStream*'i ühe sõne lahendus ei olnud kõige parema jõudlusega ja oli suhteliselt keeruliselt implementeeritav. Sellegipoolest võib ka see lahendus olla mõnes olukorras hea valik. Näiteks süsteemides, kus olulisel kohal on üle võrgu tulevate ja minevate sõnumite täpne logimine, oleks see lahendus sobiv, sest nii andmete kirjutamine andmevoogu kui nende sealt lugemine toimub ühe meetodi väljakutsega ja andmed on lihtsasti logitava sõne kujul.

Kuigi võrreldud lahendused on üles ehitatud samal andmevoogude loogikal, mis baseerub Java klassidel *InputStream* ja *OutputStream*, ja kõik täidavad sama ülesannet, milleks on andmete vahendamine serverite vahel, siis implementatsioonide poolest on võrreldud lahendused väga erinevad. Implementatsiooni spetsiifikast tulenevad lahenduste tugevad ja nõrgad küljed, kuid eelnevast analüüsist on näha, et kõikide võrreldud lahenduste jaoks on kasutusvaldkonnad, kus see implementatsioon sobib paremini kui mõni teine. Sellest tuleneb, et andmevoogudega andmevahetuse implementatsiooni valimisel on oluline esimese asjana analüüsida olukorda, kus andmevoogudega andmevahetust vaja on, ja valida implementatsioon vastavalt selle olukorra vajadustele.

5. Kokkuvõte

Käesolevas töös analüüsiti Java programmeerimiskeeles andmevoogudega serveritevahelise andmevahetuse implementeerimise võimalusi *DataStream*'i ja *ObjectStream*'i näitel. Töö esimeses osas tutvustati Java programmeerimiskeeles andmevoogude loogikat ja erinevaid andmevoogude implementatsioone. Töö teises osas implementeeriti kolm erinevat andmevoogudega andmevahetust võimaldavat lahendust – nendeks olid *DataStream*'i kasutav ühe sõne meetod, *DataStream*'i kasutav eraldiseisvate primitiivide ja sõnede meetod ning *ObjectStream*'i meetod. Defineeriti andmestruktuur, mida kõik erinevad lahendused edastavad, et implementatsioone saaks võrrelda võrdsetel tingimustel. Töö kolmandas osas loodi keskkond, milles on võimalik simuleerida pidavat serveritevahelist andmevahetust loodud implementatsioonidega ja mõõta implementatsioonide jõudlust etteantud varieeruva suurusega andmestruktuuride vahendamisel. Töö teises osas implementeeritud lahendusi kasutades simuleeriti loodud keskkonnas andmevahetust ja koguti statistilisi näitajaid iga implementatsiooni jõudluse kohta. Implementeeritud lahendusi võrreldi implementeerimise protsessi ja simulatsiooni tulemuste põhjal.

Võrdluses nähti, et andmestruktuuri saatmiseks ja vastuvõtmiseks andmevoona oli võrreldud lahendustest kõige lihtsama implementatsiooni protsessiga *ObjectStream*'i kasutav lahendus, kuid kõige parema jõudluse saavutas *DataStream*'i eraldiseisvate primitiivide ja sõnede lahendus. Võrreldud meetodite tugevustele ja nõrkustele anti hinnang ning soovitati võimalikke kasutusvaldkondi. Võrdlusest järeldati, et ei ole andmevoogude implementatsiooni, mis on teistest igas olukorras parem, vaid kõige sobilikuma implementatsiooni valimisel on oluline analüüsida olukorda, kus andmevoogudega andmevahetust vaja on, ja valida implementatsioon, mis vastab selle konkreetse olukorra vajadustele.

Töö edasiarendamiseks on mitmeid võimalusi. Üheks võimaluseks on käesolevas töös võrreldud andmevoogude implementatsioonide erinevate jõudluse aspektide detailsem mõõtmine ning jõudluse optimeerimise võimaluste edasine uurimine - näiteks on võimalik uurida andmevoogude puhvri suuruse optimaalset valikut ning erinevaid serialiseerimise võimalusi peale Java standardse serialiseerimise. Samuti on võimalik tööd edasi arendada uurides teisi andmevoogude implementatsioonide peale *DataStream*'i ja *ObjectStream*'i.

6. Viidatud kirjandus

[1] Graba, J. 2013, "An Introduction to Network Programming with Java", Springer London, pp. 1-8, 12-13, 105-108, doi: 10.1007/978-1-4471-5254-5.

[2] Estipona, J. 1998, "Getting A Jump-Start In Java Network Programming", IEEE Software, Software, IEEE, IEEE Softw, 15(5), pp. 116–117, doi: 10.1109/MS.1998.714881.

[3] Pendergast, M. 2011, "Performance, overhead, and packetization characteristics of Java application level protocols", ACM SIGITE Newsletter (ACM Digital Library), 8(1), pp. 4–15, doi: 10.1145/1941528.1941529.

[4] Calvert, K. L. & Donahoo, M. J. 2008, "TCP/IP Sockets in Java: Practical Guide for Programmers (2nd ed.)", Oxford, England, Morgan Kaufmann.

[5] Harold, E. R. 1999, "Java I/O", Sebastopol, California, O'Reilly Media.

[6] Oracle. Class OutputStream. https://docs.oracle.com/javase/7/docs/api/java/io/OutputStream.html (04.02.2021).

[7] Oracle. Class InputStream. https://docs.oracle.com/javase/7/docs/api/java/io/InputStream.html (04.02.2021).

[8] Oracle. Abstract Methods and Classes. https://docs.oracle.com/javase/tutorial/java/IandI/abstract.html (04.02.2021).

[9] Sharan, K. 2014, "Beginning Java 8 Language Features", Apress, pp. 309, 310, 314, 322-327, doi: 10.1007/978-1-4302-6659-4.

[10] Oracle. Class DataOutputStream. https://docs.oracle.com/javase/7/docs/api/java/io/DataOutputStream.html (04.02.2021).

[11] Oracle. Interface DataOutput. https://docs.oracle.com/javase/7/docs/api/java/io/DataOutput.html (04.02.2021).

[12] Oracle. Class DataInputStream. https://docs.oracle.com/javase/7/docs/api/java/io/DataInputStream.html (04.02.2021).

[13] Oracle. Interface DataInput. https://docs.oracle.com/javase/7/docs/api/java/io/DataInput.html (04.02.2021).

[14] Oracle. Class ObjectInputStream. https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html (04.02.2021).

[15] Oracle. Class ObjectOutputStream. https://docs.oracle.com/javase/7/docs/api/java/io/ObjectOutputStream.html (04.02.2021).

[16] Tartu Ülikooli arvutiteaduste instituut. Objektorienteeritud programmeerimine, praktikum 9. https://courses.cs.ut.ee/2021/OOP/spring/Main/Practice9 (15.04.2021)

[17] Oracle. Class ArrayList. https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html (04.02.2021)

Lisad

I. Implementatsioonide võrdlemiseks loodud keskkond

Lähtekood keskkonnas Github: https://github.com/Delremi/Andmevoogude-simulatsioon

II. Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, **Del Remi Liik**

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teose
   Java andmevoogude implementatsioonide analüüs ja võrdlus DataStream'i ning ObjectStream'i näitel,
   mille juhendaja on **Simmo Saan**,
   reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.
2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.
3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.
4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Del Remi Liik

07.05.2021