

TARTU ÜLIKOOL
Füüsika-keemiateaduskond
..... instituut

LAAS TOOM

**KLIENT-SERVER SÜSTEEM PERIFEERIASEADMETE
JUHTIMISEKS ÜLE INTERNETI**

Diplomitöö

Juhendaja: amet, teaduskraad ALVO AABLOO

Tartu 2003

SISUKORD

SISUKORD.....	
Sissejuhatus.....	
1. PROBLEEMI PÜSTITUS JA TEOREETILINE LAHENDUS.....	
1.1 Probleemi püstitus.....	
1.2 Teoreetiline lahendus.....	
2. RAAMPROGRAMMIDE EHITUS.....	
2.1 Ülesehituse planeerimine.....	
2.2 Realisatsioon.....	
2.3 Tulemus.....	
3. REAALNE KASUTAMINE.....	
3.1 Sissejuhatus.....	
3.2 Serveri kirjeldus.....	
3.3 Kliendi kirjeldus.....	
VIITED.....	
LISA 1: Ingliskeelne programmi juhend.....	
LISA 2: Programmi koodi listing.....	
1. RDC BAASPROGRAMMIDE KOOD.....	
1.1. RDC Server.....	
1.1.1. RDCServ.java.....	
1.1.1. DeviceInterface.java.....	
1.1.2. BasicDeviceInterface.java.....	
1.1.3. ServerThread.java.....	
1.2. RDC Client.....	
1.2.1. RDCClient.java.....	
1.2.2. ClientThread.java.....	
1.2.3. BasicClientThread.java.....	
1.2.4. ClientGUI.java.....	
1.2.5. BasicClientGUI.java.....	
2. PSCC juhtimisprogrammide koodid (nimetus arvutiklastrite järgi).....	
2.1. Cluster Client.....	
2.1.1. ClusterClient.java.....	
2.2. Cluster Server.....	
2.2.1. ClusterServer.java.....	
2.2.2. ProcessListener.java.....	

Sissejuhatus

Tänapäeval on arvutite ja interneti levik juba nii laialdane, et vähesed kujutavad ette elu ilma nendeta. Kontorites pole enam ainult üks arvuti, mida varemalt kasutati tähtsamate dokumentide koostamisel ning internetist vajalike andmete välja printimiseks. Nüüd on igal töötajal oma arvuti ning peetakse loomulikuks, et kogu töö saab nii tehtud. Suuremates firmades on kasutusel ka mitmesuguseid laivõrgu süsteemide variante, et erinevad osakonnad omavahel efektiivselt üle interneti koostööd teha saaks.

Järjest rohkem tekib ka igasuguseid teenuseid, mida interneti vahendusel saab tellida, näha või kasutada. Pole mingiks probleemiks interneti abil raamatuid osta, telefoniarvet tasuda, või kasvõi kinno kohti broneerida. Tihti on internetiteenused ka odavamad tavateenustest, rääkimata mugavusest ning aja kokkuhoiust, mis saavutatakse tänu sellele, et inimene ei pea oma kontori või koduse arvuti tagant näiteks pangaosakonna järjekorda seisma minema.

Saadaval on ka mobiiltelefonid, mis võimaldavad interneti lehekülgi külastada või pisemaid Java programme käivitada. Seega saab juba puhkusel olles või nädalavahetusel maakodust kiiremad asjatoimetused ära teha.

Kõik see näib viitavat asjaolule, et õige pea on oodata poodidesse ka internetiga ühendatud kodutehnikat: näiteks saunakeris, mida on võimalik internetist sisse-välja lülitada. Kellele ei meeldiks võimalus oma soojavee boilerit paar tundi enne reisilt koju jõudmist sisse lülitada, et saabudes oleks hea kohe dušši alla minna.

Sellised seadmed on aga reeglina kallid ning kui praegu olemasolev seade töötab korralikult, siis ei taha keegi suurt hulka raha kulutada, et omale uuem versioon soetada. Siiski võivad mõned huvilised üritada ise ehitada oma saunakerisele interneti tuge. Kuna ilmselt sellisel juhul on vaja laialdasi teadmisi nii elektroonikast, kui programmeerimisest,

siis võtab selle süsteemi valmistamine omajagu aega ning tahtmist.

Selle diplomitöö eesmärgiks ongi taoliste süsteemide arendamise lihtsustamine. Selleks valmistasin raamistiku programmidest, mida on kindlasti vaja kasutusele võtta, et saaks seadmeid interneti abil juhtida. See tähendab, et see diplomitöö kujutab endast baasprogramme, millele saab minimaalse programmeerimisega juurde ehitada juba mingit konkreetset seadet juhtiva *plug-in*'i¹. Taoliste *plug-in*'ite koostamine nõuab põhimõtteliselt baasteadmisi Java keelest ning siis sügavamaid teadmisi sellest keelest, milles seadme juhtimise programm koostatakse (süsteem võimaldab reaalsel juhtimist üle anda välistele programmidele). Edaspidi tuleb lähemalt juttu juba konkreetsest implementatsioonist.

¹ *ingl. k.* pistikplokk, pistikkomplekt; IT-s plugin. Siin tähendab seda viimast ehk siis pisemat programmi, mis täiendab põhiprogrammi mingis spetsiifilises osas.

1. PROBLEEMI PÜSTITUS JA TEOREETILINE LAHENDUS

1.1 Probleemi püstitus

Diplomitöö idee sai alguse teisel kursusel kuulatud aineksest Arvutijuhitavad mõõtmised (FKKF.03.036), mille raames valmistasin kaamera juhtimise süsteemi kahest sammumootorist ja arvuti *joystick*'ist. Seda süsteemi valmistades tekkis küsimus, et kui lihtne oleks seda *joystick*'i asendada üle interneti toimiva juhtimissüsteemiga.

Sellise vahetuse tegemiseks peab:

- 1 asendada mootorite juhtimissüsteemi sellisega, mis võtab *joystick*'i asemel sisendiks hoopis internetist tulevat infot.
- 2 asendada *joystick*'i veebis kuvatava graafilise kasutajaliidesega.

See tähendab, et lisaks riistvara juhtimise programmeerimisele tuleb programmeerida nüüd ka kasutajaliides ning klient-server süsteem, mis suudab infot vahetada üle interneti. Kuna järjest rohkem hakkab aktuaalseks muutuma info autentsuse ja turvatuse probleem, siis tuleks kogu sellele süsteemile lisada veel nõue, et vahetatav info oleks krüpteeritud, sest olenevalt rakenduse kohast, võib süsteemi omanikule osutada väga kahjustavaks, kui ühendust on võimalik pealt kuulata.

Kes sellise süsteemi juba kord ehitanud on, sellel tekib järgmise arendamisel kohe küsimus, et kas ei oleks võimalik kasutada eelmise töö tulemusi, et seekord lihtsam oleks. Vastus on, et suure tõenäosusega on võimalik vähemalt mingit osa kasutada. Kuipalju täpselt, see sõltub juba konkreetsest keelest ning programmide ülesehitusest.

Sellest tulenevalt sõnastan probleemi:

Kuidas luua sellist programmide kogumikku, mis võimaldab:

- luua turvatud ühendust üle interneti
- kerge vaevaga „õpetada“ seda süsteemi mingit konkreetset seadet juhtima

- pakub lihtsat, kuid muudetavat kasutajaliidest.

1.2 Teoreetiline lahendus

Esmalt läheme tagasi selle lõigu juurde, mis kirjeldas minutehtud kaamerajuhtimise süsteemi üleviimist interneti. Kui neid kahte punkti pisut lähemalt analüüsida, siis võib neid jagada omakorda nii, et tulemuseks on:

- 1 asendada kaamerajuhtimise süsteem sellisega, mis võtab infot teatud sisendist (näiteks standardne sisendvoog, käsurida või teatud funktsiooni käivitamine)
- 2 luua selline programm, mis vahendab infot kaamera juhtimise programmi (punkt 1) ning interneti vahel
- 3 luua selline programm, mis vahendab infot interneti ning kasutajaliidese vahel
- 4 asendada *joystick* sellise kasutajaliideseaga, mis kuvatakse veebis ning vahetab infot punktis 3 nimetatud programmiga.

Sellise süsteemi korral on võimalik punktides 2, 3 ja 4 toodud programme kasutada erinevate kaamerate juhtimiseks, kus iga konkreetse kaamera juhtimiseks koostatakse selle kaamera liidese (COM, LPT, USB, PCI kaart jne) jaoks oma juhtimisprogramm. Kui aga kooskõlastada punktide 3 ja 4 vaheline suhtlus teatud protokollide abil, siis on võimalik asendada ka punktis 4 toodud programm iga rakenduse jaoks spetsiaalselt loodud kasutajaliideseaga. See aga tähendab, et selle sama süsteemi abil on nüüd võimalik juhtida juba mitte ainult kaameraid vaid juba palju laiemat hulka seadmetest.

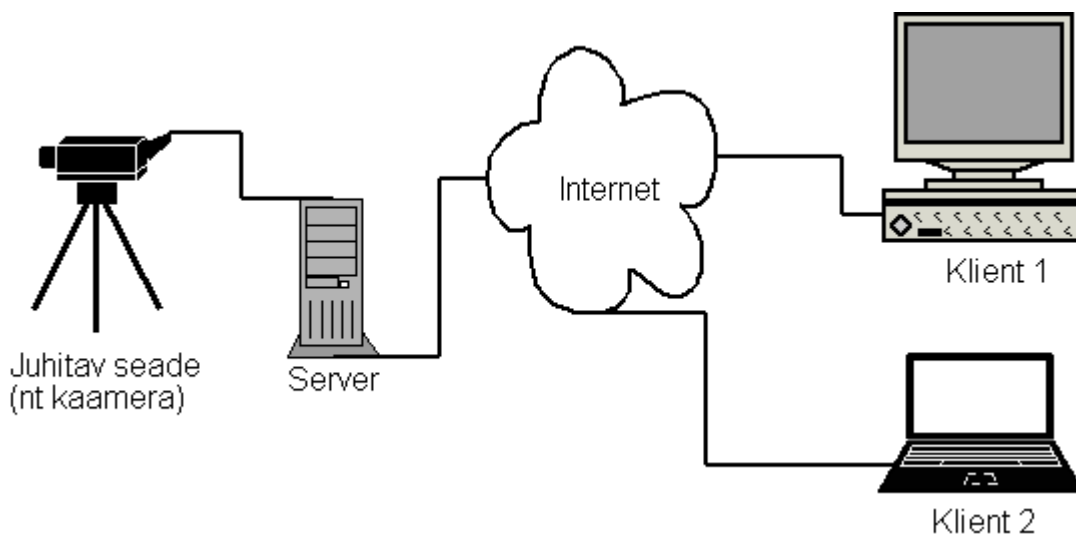
Nüüd jõuamegi probleemi lahenduseni. Tuleb koostada sellised kaks programmi, mis suhtlevad omavahel üle interneti ning kumbki võimaldab endale lisada teatud protokollide abil *plug-in*'id, mis siis serveri programmile lisab oskuse suhelda vajaliku seadmega ning kliendi programmile lisab kasutajaliidese, mis on kohandatud just nimelt selle seadme juhtimiseks. Kuna sellise süsteemi täpsem kirjeldus langeb kokku programmide ehituse kirjeldusega, siis sellest tuleb juttu järgmises peatükis.

2. RAAMPROGRAMMIDE EHITUS

2.1 Ülesehituse planeerimine

Vajalike programmide koostamise esimeseks sammuks on ülesehituse planeerimine.

Mingi juhtimissüsteemi internetti viimisel tekib enamasti järgmine struktuur:



Joonis 1. Klient-server süsteemi üldine ehitus. Joonisel on näha juhitud seade, seda kontrolliv arvuti (server) ning kaks klienti, kelle arvutid erinevad oma arvutusliku võimsuse poolest (sülearvutid on reeglina väiksema võimsusega, kui lauaarvutid).

Joonisel on kirjeldatud olukord, kus server teenindab korraga kahte klienti. Olenevalt rakendusest, võib olla lubatud ainult ühe kliendi, mingi kindlaksmääratud arvu või piiramatut arvu klientide samaaegne lubamine. Kogu süsteem peab olema planeeritud juba algusest peale selliselt, et oleks võimalik kasutajate arvu muuta (mingi parameetri seadmisega).

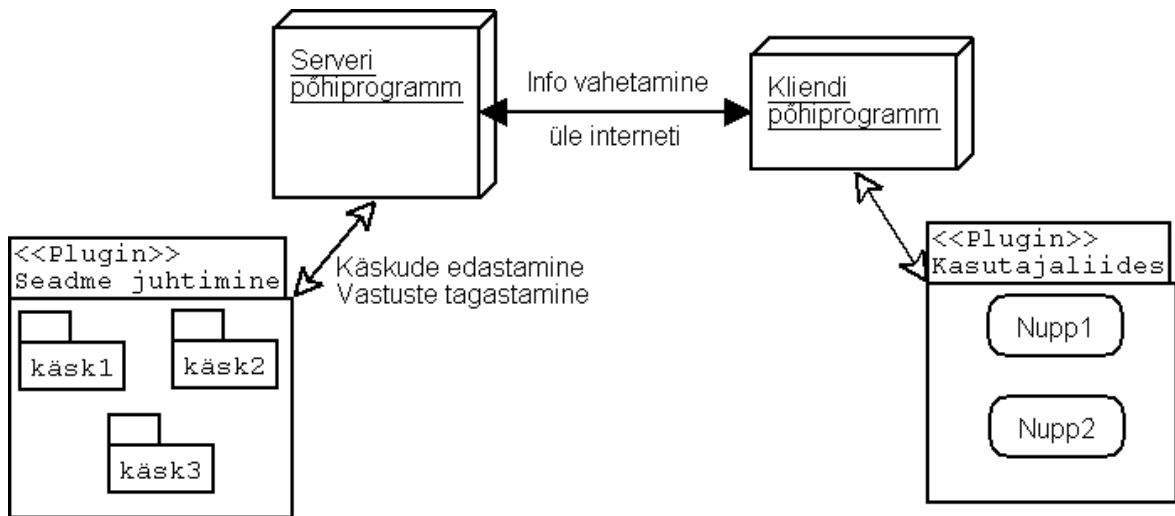
Teine oluline punkt on, et vähemalt kliendi poolne tarkvara peab olema disainitud minimaalse võimaliku ressursivajadusega ehk siis ebavajalikke pilte ja muid ilustusi ei ole mõtet põhissüsteemi planeerida, kuna see suurendab allalaadimise aega ning võtab käivitamisel rohkem ressursi, mis aga muutub oluliseks, kui tegemist on väikese

arvutusvõimsusega seadmega (näiteks pihuarvuti või mobiiltelefon).

Süsteemi väljatöötamist alustasin kõikvõimalike riistvaraseadmete läbimõtleemisest, mida võiks selle süsteemiga juhtida saada. Algne idee oli koostada just üks ning laia haardega programm, mis suudaks võimalikult palju. See tähendab, et jättes siiski lahti võimaluse edasiarenduste lisamiseks, oleks programm defineerinud teatava hulga liideseid, mille kaudu ta riistvaraga suhelda suudab. Selle programmi kohandamine toimus keeruka konfiguratsioonifaili abil, mis oli jagatud sektsioonideks, millest igaüks koosnes paljudest käskudest ning üks sektsioon kujutas endast ühte konkreetset käsku, mida klient nupuvajutusega täide saata võib. Võimalik oleks olnud ka mitut tegevust üksteise järel täita ning sedasi moodustada käskude kette. Kui selline konfiguratsioonifail on veel mõledav ning võib-olla polegi nii segane aru saada (ehituselt sarnane näiteks Apache http serveri konfiguratsioonifailile), siis sellest failist info välja lugemine ning mõistliku kasutajaliidese kavandamine osutusid keerukaks protsessiks. Nimelt oleks vähegi korralikuma kasutajaliidese defineerimiseks vaja läinud kümnekonna parameetri seadmist ning kui kõik käsud kokku arvestada ning siis selle selgeks saamisele kuluvat aega vaadelda, siis oli tulemus sama hea, kui uue programeerimise keele selgeks õppimine.

Seetõttu jõudsingi õige pea arusaamisele, et ei tuleks üritada kasutaja eest ära defineerida, kuidas mingite seadmetega töötama peaks, pakkudes talle ainult teatud nimekirja võimalikest tegevustest, seeläbi piirates programmi kasutusvaldkonda. Vaid hoopis peaks põhilise tähelepanu koondama võimalikult laialdaselt kasutatava tuuma koostamiseks ning jätma konkreetse seadme juhtimise defineerimine süsteemi rakendaja hooleks.

Seda silmaspidades kavandasingi programmi ehituse selliselt, et süsteem koosneb mitmest osast, millest mõned laetakse sisse alles töö käigus dünaamiliselt ning kompileerimise ajal pole vaja teada, mida need osad tegema hakkavad.



Joonis 2. Programmide suhtluse skeem. Nii kliendi kui ka serveri programmid sisaldavad vähemalt ühte *plug-in*'it, mis laetakse sisse dünaamiliselt, alles käivitamise hetkel.

Nagu joonisel 2 näha, on süsteemi nii-öelda jäikadeks osadeks kaks programmi (joonisel vastavalt serveri ja kliendi põhiprogrammid). Need on eelkompileeritud ning peaks üldiselt jääma muutumatuks, isegi süsteemi uute seadmete jaoks kohandamisel. Käivitumisel aga laetakse sisse moodulid (*plug-in*), mis on kohandatud just konkreetse seadme jaoks.

Sellise skeemi alusel koostatud süsteemi eeliseks on, et uute seadmete jaoks toe programmeerimisel tuleb kirjutada ainult minimaalne osa koodi – see, mis viib läbi reaalselt juhtimist (näiteks kirjutab baite kuhugi porti) ning mis tuleks nagnii programmeerida, kui mingit seadet juhtida tahetakse. Ainuke erinevus tavalise ühe arvuti süsteemi koostamisega võrreldes on selles, et programmeerimisel tuleb jälgida teatud protokollit (et *plug-in* ühilduks põhiprogrammiga).

Järgmise sammuna peale planeerimist tuli valida keel, milles programmid koostada. See valik kaldus juba algusest peale java keele kasuks, kuna see võimaldab platvormisõltumatust. Samuti on võimalik javas kirjutada rakendeid (*applet*), mis pakuvad kasutajaliidese tegemisel suuremat võimsust, võrreldes tavalises HTML-is esitatud veebilehega. See tähendab, et on võimalik kliendipoolel teha palju arvutusi ning alles nende tulemused saata üle võrgu serverisse ning võib kindel olla, et kogu süsteem töötab täielikult

ning korralikult. HTML-is pakub sama võimalust javascript, kuid seda kasutades ei saa kunagi täiesti kindel olla, kas kõik brauserid on kasutatavaid käskke implementeerinud ning kas need kajastavad tulemusi just täpselt nii, nagu loodetud (javascript ei ole standardiseeritud ning seda ei arenda ühtne organisatsioon, nagu java korral, vaid iga brauseri tootja on sellest keelest oma versiooni arendanud).

2.2 Realisatsioon

Kuna süsteemi projekteerides mõlkus meeles peamiselt riistvara juhtimine ning programmile nime valides jäi tähelepanuta tema märgatav võime ka tarkvara juhtida (on võimalik käivitada ka juba olemasolevaid programme), siis sai süsteemile nimeks pandud *Remote Device Controller* ehk lühendatult *RDC* (eesti keeles võiks nimi kõlada *Seadmete Kaugkontroller* ehk *SKK*). Nimi ning ka kogu programm on inglise keelne, sest internet on kättesaadav üle terve maailma ning nii-öelda interneti keeleks on kujunenud inglise keel. Seega kui programm võimaldab interneti abil midagi juhtida, siis peaks ta olema arusaadav võimalikult suurele hulgale inimestest. Teiseks põhjuseks oli demonstreerimiseks koostatud reaalne rakendus – suure tõenäosusega hakkavad seda kasutama mitte eesti keelt kõnelejad.

Süsteemi programmeerides pidasid kogu aeg silmas, et seda oleks võimalik rakendada võimalikult erinevate seadmete juhtimiseks. Seetõttu alustasingi programmeerimisel justnimelt tuuma kirjutamisest. See on programmi nendest osadest, mis teostavad internetis suhtlust ning vahendavad saadud infot programmi teistele osadele.

Java keeles on võimalik teostada interneti suhtlust kahel põhilisel viisil:

- kasutades URL-e: HTTP, FTP jt protokolle kasutades
- kasutades *SOCKET*-eid: luuakse otseühendus teise arvutiga

On ka võimalus kasutada UDP datagramme, kuid kuna UDP protokollis ei ole garanteeritud, et andmepaketid jõuavad kohale õiges järjekorras ning ei kasutata

veakorrektsiooni, siis on mõistlik siinkohal eelistada TCP protokoll [1] ning seega siis sokerite andmevooge (*datastream*).

Kogu info, mis üle interneti liigub, on reaalselt andmejada kujul (edastamisel loomulikult pakettideks jaotatud). Sellise jada lihtsaim näide on sõne (*string*), mida ka enamasti kasutatakse andmete vahetamiseks üle interneti. Mina aga otsustasin, et efektiivuse mõttes, on otsarbekam mitte kasutada sõnesid andmete edastamiseks, vaid hoopis andmevektoreid. Java keeles võib andmevektor sisaldada kõiki objekte, välja arvatud primitiivsed andmetüübid. Kuid ka nende lisamiseks on võimalus – iga andmetüübi jaoks on olemas mähisklass (*wrapper class*), mis defineerib selle tüübi ümber objekti ning seetõttu on seda nüüd võimalik vektorisse lisada. Vektorite kasutamise eesmärk on teha võimalikult lihtsaks programmides andmete vahetamine. Selge on see, et teatud käsked peab täitma ka tuumaprotsessid (näiteks kliendi lahkumisteade). Samas enamus infot on serverile tundmatu ning võib-olla polegi hea, kui server üritab sellest midagi välja lugeda. Seetõttu on minu süsteem koostatud niimoodi, et vektori esimest elementi käsitletakse kui esimese astme käsunime. See peab olema sõne (*string*) kujul, muidu tekib suuremaid probleeme, sest nii serveri kui ka kliendi põhiprotsessid üritavad sealt sõnet välja lugeda. Seda käsku kontrollitakse ka põhiprotsessi poolt ning kui see on talle tuttav, siis täidetakse see. Edasi saadetakse kogu vektor aga juba järgmise astme programmile – pluginile. Kuna vektori elementide hulk pole piiratud ning teda internetist vastu võttes ei pea teadma kui pikk vektor on, siis võib see vektor sisaldada peale esimese astme käsunimedele veel ka teise, kolmanda jne astme käsunimesid. Samuti võib ta sisaldada mis tahes kujul andmeid ning kasvõi omakorda mingeid andmeobjekte (näiteks vektoreid).

Vektorite edastamiseks kasutan javasse sisse ehitatud omadusi – objektide jadaks muutmist (*serialize*). Need jadad edastatakse üle interneti ning teises otsas pannakse jälle objekt kokku.

Kuna need objektid võivad sisaldada kõiksugu informatsiooni, sealjuures parooli või muid isiklikke andmeid, siis on kasutusele võetud viimasesse java versiooni juba integreeritud (vanematele versioonidele tuleb ise juurde installeerida) JSSE pakett ning SSL v 3.0 standard [2]. Selle tulemusena luuakse andmeühendus kahe arvuti vahel üle turvalise

kanali, mille krüptosüsteemiks valitakse mõlemalt poolt toetatud kõige tugevam algoritm. Kuna J2SE 1.4.1_02 versiooni kaasatud JSSE's ei ole realiseerinud veel AES standard, siis kõige tugevamaks osutub ning kasutusse läheb 128-bitise võtmega RC4 krüptosüsteem. Mõninagte testide andmeil [3] (seal on RC4 nimetatud ARC4) on RC4 süsteem ka kiirem, kui AES (viidatud lehel oma originaalnime Rijndael all).

Kuna server ja klient ei tööta sünkroonses režiimis, siis peab mõlemal olema üks lõim (*thread*), mis jälgib on ootab pidevalt sissetulevaid andmeid ning nende saabumisel alustab nende töötlemisega. Need lõimed on tegelikult väga lihtsa ehitusega ning ei peaks tekkima vajadust nende ehitust muuta mingi seadme juhtimiseks, kuid sellegi poolest võib seda teha, sest ka need klassid loetakse dünaamiliselt sisse ning seetõttu on võimalik nende hilisem muutmine.

Viimase, kuid siiski arvatavasti kõige tähtsama komponendina kirjeldan nii kliendi kui ka serveri *plugin*'eid.

Server

Serveri töö alustamisel loetakse konfiguratsiooni failist, milline klass on määratud *plugin*'iks ning laetakse see sisse. See tähendab, et luuakse isendiväli, mille kaudu siis pääsetakse ligi seadme juhtimiseks vajalikele funktsioonidele. Täpsemalt öeldes küll edastatakse ainult teatud sõnumeid (nagu: klient saabus, klient lahkus, käsk saabus) ning nendega tegelemine jäetakse täielikult selle klassi enda hooleks.

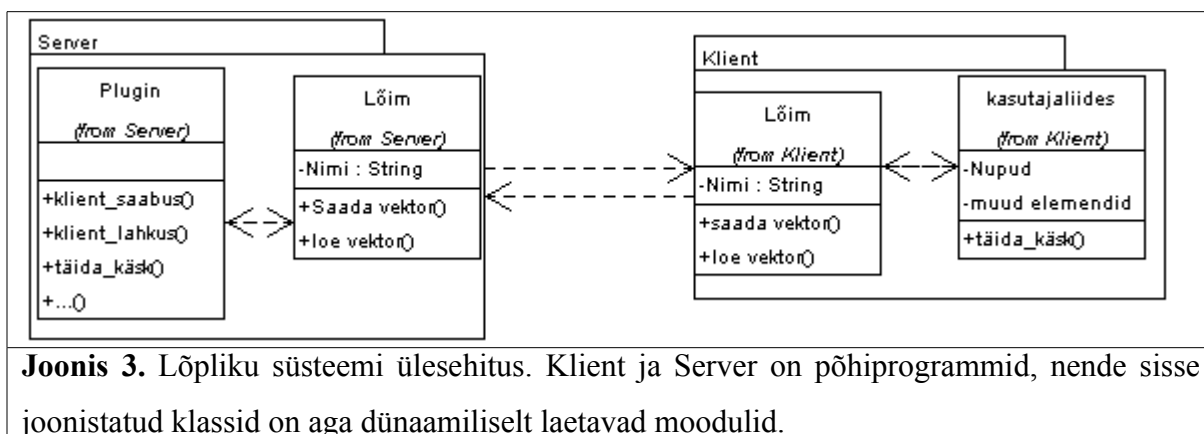
Klient

Kliendi põhiliseks klassiks on kasutajaliides. See peab olema nii ehk nii spetsiifiline, et mingeid seadmeid juhtida, seega võib nende juhtimiseks vajaliku koodi koondada samuti sellesse klassi. Nagu serveri puhul, nii ka kliendi puhul loetakse konfiguratsioonifailist, millised klassid tuleb laadida töökäigus.

Nagu öeldud, on nii serveril, kui ka kliendil moodulitena realiseeritud ka lõimed, mis

jälgivad võrguühendust. Kõikide nende klasside laiendamiseks (ehk siis mingi seadme jaoks spetsiifilisemaks muutmiseks) ent samas süsteemiga ühilduvuse säilitamiseks on defineeritud abstraktsed klassid, mis loetlevad kõik funktsioonid, mida süsteem nõuab nendelt klassidelt, et nendega suhelda. Kuna lõimed ei peaks muutuma süsteemi muutumisel, siis ma ei defineerinud eraldi abstraktseid klasse lõimede jaoks, vaid uute lõimede loomiseks peab laiendama juba olemasolevaid lõimi.

2.3 Tulemus



Programmide valmides oli tulemuseks selline süsteem, kus on võimalik lihtsa vaevaga asendada mingi arv klasse ning seeläbi panna süsteemi tegema hoopis midagi muud. Testimiseks on süsteemiga kaasa pandud nii kliendi kui ka serveri *plugin*'id, mis demonstreerivad nii kasutamise kui ka programmeerimise seiuskohast, kuidas seda süsteemi rakedada tuleb. Ühenduse testimiseks on kliendi programmis kasutajale üks nupp, millele vajutades saadetakse serverile hetke kellaeg. Kui server selle kätte saab, siis lisab sinna omapoolse lause ning saadab kliendile tagasi. Kui selline kliendi kasutajaliides ei oma suurt mõtet, siis serveri poolne *plugin* on hea ka hilisemates rakendustes kasutada. Nimelt on seda võimalik kasutada kliendiprogrammi silumisel – ta peegeldab tagasi kogu info, mis klient talle saadab.

Sellest juba olemasolevast süsteemist on aga triviaalne teha interneti jututuba. Selleks on vaja kliendile lisada ainult teiste jutu kuvamiseks suurem tekstiaken ning teksti sisestamiseks üks pisem aken ning selle nupu funktsioon ümber muuta kellaaja saatmisest sisestatud teksti saatmiseks ning ongi olemas jututoa klient. Sellele kliendile võib lisada veel igasuguseid omadusi. Näiteks võib ta kuvada enda saadetud teksti teiste saadetud tekstist erinevas värvis või siis üldse võimaldada värve ning fonte ise valida. Kogu see info on võimalik lihtsalt paigutada vektori erinevatesse elementidesse ning vastuvõtval kliendil ei ole muud teha vaja, kui need õigetesse kohtadesse kirjutada.

Serveri muutmine on isegi veel lihtsam. Seal tuleb võtta ära see rida, mis lisab serveri enda lause ning samale kliendile tagasi saatmise asemel tuleb kasutada serveri funktsiooni *Broadcast* (ingl. k. levitama), mis saadab selle sõnumi kõikidele klientidele.

Sellise modifikatsiooni tegemiseks ei peaks kuluma eriti palju aega. Kuid samas on see ka üpris lihtne rakendus. Järgnevalt aga vaatleme pisut keerulisemat olukorda.

3. REAALNE KASUTAMINE

3.1 Sissejuhatus

Kogu see eelnev jutt on olnud suuremalt jaolt teoreetiline ning ei tõesta kuigi selgelt, kas seda süsteemi on üldse võimalik rakendada reaalses elus või mitte. Selle ilmestamiseks on mulle lisäülesandeks tehtud interneti teel juhtima hakata seadet nimega *Power and Serial Connection Controller* ehk *PSCC*.

Lühidalt PSCC seadmest.

See seade on Alvo Aabloo juhendatava semestritöö raames valmistatud. Seadme eesmärk on pakkuda mingi arvutivõrgu administraatorile võimalust teha arvutitele restarti, olukorras, kus arvutid on lõpetanud igasuguse TCP ja muu suhtluse ning administraatoril pole võimalik kohale minna. Sellisel juhul tuleb administraatoril ühendada ennast (telneti või SSH vahendusel) ühte selle seadme kontrollarvutitest ning vastava programmi abil sellele seadmele öelda, millisele arvutile tuleb restart teha. Restardi tegemise all mõeldakse siin lihtsalt voolu katkestamist teatud ajaks (liiga kiire tagasilülitamine on arvutile ohtlik).

Lisaks restardi tegemisele võimaldab see seade kokku omavahel ühendada kontrollarvuti ning suvalise alamarvuti jadaväratid ning seeläbi saada juhtarvutisse teise arvuti konsool. Tegeliku konsooli saamiseks muidugi tuleb juba spetsiaalseid programme kasutada (nagu Minicom või Microcom).

Kogu see süsteem töötab väga hästi, kuni selle hetkeni, mil administraator ei viibi enam oma isikliku tööarvuti juures, vaid on kuskil, kus SSH ei ole installeeritud ning ei ole ka õigusi seda teha ja telnet on liiga ebaturvaline (kogu tekst, sh paroolid liiguvad avatekstina üle interneti). Sellisel juhul olekski vaja sellist süsteemi, mis oskaks seda seadet juhtida ning pakuks kasutajaliidest, mis avaneb veebi ning ei nõua mingit täiendavat installeerimist.

Seega on sellise süsteemi juhtimiseks vaja mingil moel PSCC seadmega suhelda ning samas pakkuda ka kasutajaliidest, mis oleks funktsionaalne. Järgnevalt tulebki lähemalt juttu mõlemast aspektist eraldi.

3.2 Serveri kirjeldus

Kuna PSCC juhtimiseks koostati C-keelne programm, mis selle juhtimisega edukalt toime tuli, siis pole enam mõtet samu algoritme uuesti jahas üle kirjutada. Selle asemel võib käima panna selle sama programmi ning selle abil kontrollida siis juba seadet.

Selleks tuleb koostada järgmised programmid:

- serveri *plugin*, mis suhtleb serveriga ning vahendab käsked kontrolleri
- lõimed, mis käivitavad kontrolleri ning jäävad kuulama tema väljundeid (standardne väljundvoog ja standardne veavoog). Mõlemad vood suunatakse kliendile edasi.

Kuna PSCC võimaldab ka jadaväratite ühendamist, siis tuleb ka see omadus üle interneti kättesaadavaks teha. Selle jaoks on vaja kasutusele võtta veel teinegi väline programm (mina kasutan programmi nimega Microcom [4], tema väiksuse tõttu – kõigest 16 kB, kuid võib kasutada ka ükskõik millist programmi, mis ei nõua terminali olemasolu, vaid kuvab informatsiooni lihtsalt ridade kaupa ekraanile). Ka selle programmi jaoks on vaja ülalnimetatud kahte programmi, kuid neid ei pea uuesti programmeerima vaid saab nende samade klassidega hakkama – tuleb lihtsalt uued isendid luua.

Kuna PSCC seadet tohib korraga kontrollida ainult üks inimene (vastupidine variant oleks mõeldamatu – läbisegi käskude andmine), siis on serveri poolt lubatud ainult üks klient ning *plugin* on ehitatud teadlikult seda arvestades, et kliente on korraga ainult üks (teatud muutujate salvestamiseks on kasutusel lihtmuutujad, mitte massiivid, nagu mitjekasutaja

režiimis oleks olnud).

Kliendi ühendudes käivitatakse PSCC kontrollprogramm – NetPSCC ning kui klient saadab käske, siis need edastatakse selle programmi standardsisendisse. Kui aga programm genereerib mingisugust väljundit, siis see edastatakse kliendile täiesti sõltumatult.

Serveri üles seadmiseks on vaja korrigeerida tema konfiguratsioonifaili, seades korrektseks käskude otsiteed (*path*) ning määrates ära, mitu arvutit on PSCC külge ühendatud.

Kuna server sõltub ainult netpscc väljundist, siis jadaväratite kokkuühendamise nõudmise järel hakatakse väljundist otsima teatud sõnet, mis annaks märku, et väratite ühendamine õnnestus. Kuna programmi kahes erinevas versioonis oli see sõne erinevalt toodud, siis otsustasin ka selle konfiguratsioonifaili välja tuua, nii et seadistusel saab ka selle määrata, millise sõne esinemise peale netpscc väljundis käivitatakse konsooli programm.

3.3 Kliendi kirjeldus

PSCC juhtimiseks on terve rida käske:

- *login, logout ja force login* – need käsud on selleks, et seade võtaks kuulda seda arvutit
- *connect N, connect main* – need käsud on selleks, et ühendada server-arvuti jadavärat kokku kas N-nda alamarvuti või siis teise kontrollarvuti väratiga.
- *reboot N, reboot main, reboot self, reboot device* – need käsud on selleks, et katkestada vool vastavalt N-ndal alamarvutil, teisel juhtarvutil, sellel samal arvutil või seadmel endal. Kusjuures *reboot self* on hetkeline ning ei ole aega mingeid ühendusi sulgeda, seega olenevalt katkemise hetkest võib kliendipool hakata väga palju veateateid andma selle kohta, et ühendus katkes.
- *status* – selle käsu peale saadab seade 4 baiti informatsiooni, mille siis netpscc programm lahti kodeerib ning kliendile arusaadavas sõnastuses esitab.

Iga käsu kohta on kliendil üks nupp, ning käskudes, kus N on parameetrik (reboot ja connect), on iga arvuti kohta üks selline nupp – arvutite arv küsitakse serverilt

ühendumisel.

INGLISE KEELNE RESÜMEE

RDC system is ment to offer a basic framework for taking some tasks to WWW. This uses a secure tunnel between server and client and can easely be customized for various tasks. The main idea is to make easier deploying web-enabled hardware controllers by cutting off the need to program the web interface. Using this system it is only needed to develop both ends of the program.

The diploma work consists of two parts:

1. Developing the basic framework
2. Deploying it to control PSCC (Power and Serial Connection Controller)

Developing the framework

The framework is built up using modular schema to make it more abstract and allow customers to modify it for their needs. Both the client and the server load their moduls at runtime and do not need to be recompiled, when a new module is added (or changed). The moduls use specified API to make themselves compatible with the baseprograms. But to accomplish various tasks, these moduls add their own commands to that API so that it can be used to control some devices or execute system commands.

Controlling PSCC

PSCC is a device that is able to power off other computers when it recieves such a command from a master computer. It also is able to connect the master computers serial port to one of the slave comuters port or to the other master computers port.

Adding this to the RDC makes it possible to control PSCC over network without having to log in via SSH. This is especially useful when SSH is not an option and only web browser can be used. RDC is also able to display the other computers console via the web so that basic administration may be carried out on that computer.

VIITED

- 1] RFC (Request for Comments) dokumentide kogumik
<http://www.rfc.net/rfc768.html>
- 2] SUN Microsystems, Inc. JSSE dokumentatsioon
<http://java.sun.com/j2se/1.4.1/docs/guide/security/jsse/JSSERefGuide.html>
- 3] Wei Dai krüptosüsteemide kiirusetest
<http://www.eskimo.com/~weidai/benchmarks.html>
- 4] Microcom Serial Terminal Emulator
<http://microcom.port5.com/>

LISA 1: Ingliskeelne programmi juhend

* RDC - Remote Device Control *

CONTENTS:

- 1. Basics
 - 1.1 About
 - 1.2 Requirements
 - 1.3 Setting up the system
 - 1.4 Running the Server
- 2. Structure of Programs
 - 2.1 The server
 - 2.2 The client
 - 2.3 The communication

1. BASICS

1.1 ABOUT

RDC system is ment to offer a basic framework for taking some tasks to WWW. This uses a secure tunnel between server and client and can easly be customized for various tasks. The main idea is to make easier deploying web-enabled hardware controllers by cutting off the need to program the web interface. Using this system it is only needed to develop both ends of the program. It may be possible to part a single-machine program into GUI and motor,

and connect these to RDC: motor to RDCServ and GUI to RDCClient. This is accomplished by adding some methods to both ends so that using the RDC interface these ends may communicate (basically: they send each other messages like "call method1(args)").

1.2 REQUIREMENTS

- * Java runtime environment: J2SE 1.4.1 is preferred, but 1.3 or 1.2 with JSSE installed should work too (not tested). This must be installed on both: server and client hosts.

- * webserver - if used over internet, then webserver is needed to serve the applet html page.

- * TCP port - at least one port is needed (depends on deployment) to communicate over internet with client.

1.3 SETTING UP THE SYSTEM

RDC system consists of two parts: the server and the client. The client is an applet that is loaded in clients browser and from there connects to the host it came from. If the server is running on that host, the connection is simply established. But if the server is running on some other machine (e.g. in a LAN behind a firewall and only the webserver is able to connect to that host), then somekind of port forwarding or relaying may be used to trick the client so it thinks that the server is running on the webserver. (This webpage may be of

help:

http://www.ssh.com/support/documentation/online/ssh/adminguide/32/Port_Forwarding.html).

RDC server side can be placed wherever in the file hierarchy, as long as all of it is in the same place (or linked together via CLASSPATH variable so the Java VM can find the classes).

The system was built using J2SE 1.4.1_02 (which has integrated JSSE), but as RDC uses the the very basic of java, then it should be able to run with J2SE 1.3 (or 1.2) with JSSE add-on.

The servers configfiles are located at the root of the package. There are some variables that can be modified and also the names of the classes to be loaded at runtime. All parameters in the files are REQUIRED and errors occur when the are missing.

After configuring the system, all of the files can be packaged in a jar file, to be more compact.

RDC Client should be placed somewhere the webbrowsers have access to. The RDCClient.html is the html-page that loads the applet. This can be fully customized to fit your needs.

These files too can be packaged into a jar file.

1.4 RUNNING THE SYSTEM

Starting the server is the same way as all java programs are started. The main class is RDCServ in package ee.ut.physic.rdc.server, so concluding in the full name: ee.ut.physic.rdc.server.RDCServ. This should be passed to java command as the filename to be executed.

for example (at the root of the package):

```
java ee.ut.physic.rdc.server.RDCServ
```

if this should fail (no classes found), try this:

```
java -classpath . ee.ut.physic.rdc.server.RDCServ
```

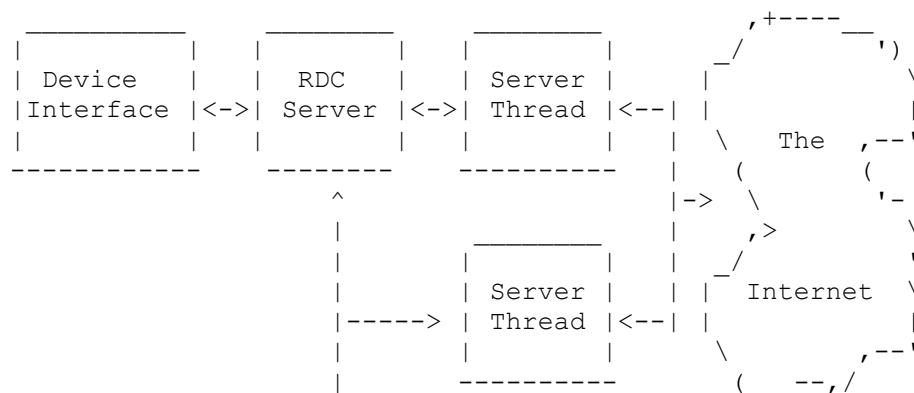
When started, the server tries to bind the port specified in configfile. This port should be opened in firewall (if one exists).

The RDCClient is viewed with browser by navigating to a specific URL (the location of RDCClient.html).

2. THE STRUCTURE OF PROGRAMS

2.1 THE SERVER

The server is built in such way that it should be very simple to extend. Simple layout of the server:



RDC is built up using a plugin-schema. This means that the main server loads its plugins

(that are specified in config file) dynamically at runtime and needs not to know the names of the classes at compile time. This allows RDC to be extended without any modification to its framework.

The main server (middle part) is the thread that coordinates the whole communication. It is started on the server host and then it binds predefined TCP port and waits for clients.

The main server also starts the DeviceInterface. Actually it loads a class that extends the DeviceInterface, and thus providing the server the basic means for communicating with the plugin. To customize RDC for some specific task, this plugin needs to be customized. All data that client sends is sent to this plugin. Therefore the RDCServ needs not to be aware of the task the system is doing and only DeviceInterface is responsible for being competent on the client-sent commands.

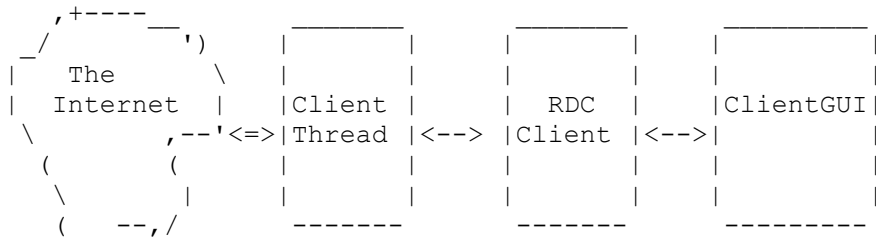
When a client connects, a new instance of ServerThread is created. This class is for constantly listening for client data and most of the time this plugin needs not to be customized. Nevertheless server loads this class also dynamically, offering the possibility for change. This thread has a unique name, by which it is recognized and communicated to in multiuser mode.

When a client sends data the ServerThread reads it and passes it on (to the left on the figure). This command is analyzed first by server (basically only for outputting some text to servers stdout if needed) and then passed on to the DeviceInterface that is supposed to handle it and probably send some feedback to client.

See API for details.

2.2 THE CLIENT

The client is similar to server - it consists of three parts - the thread to monitor server output, the main process and the plugin that does the work.



When the client is started, it loads (also dynamically) the two classes: ClientThread and ClientGUI.

ClientThread is basically the same as ServerThread, offering methods for listening on socket and forwarding incoming data upwards to the handlers. All data is sent to RDCClient and from there to ClientGUI.

ClientGUI is a Panel. This may contain buttons, text areas, what ever.

NOTE: the ClientGUI is NOT added to the client applet by the RDCClient process, but it should be done in the client somewhere. This is to offer the possibility to make some changes to the layout at runtime and before the panel is showed.

2.3 THE COMMUNICATION

RDC uses TCP sockets to exchange data between client and server. This is the reason why SSL is used (because HTTPS protocol does not encrypt socket data).

At startup server starts listening to a port. Client connects to that port and both server and client open datastreams over the connection. On opening of connection, SSL handshaking takes place, where the best (strongest) encryption method available on both ends is selected. If none is found, the connection is dropped (this should not happen, because both ends use the same provider - SUN JSSE and therefore have access to the same methods).

The data that is moved across network consists of messages. Each of which are independent commands. These messages are not of type String, but are objects - instances of Vector.

This method was chosen to simplify the command parsing (no need to tokenize strings, all separate data can be accessed in random mode). The first element of vector is considered to be a command.

If this command is one of:

- * serverOutputCommand - This is used to simply print the data to servers stdout. this can be changed at runtime or in config file.

- * clientOutputCommand - this means that the second element in vector is the command that is to be used when outputting on client. This can be different for every client.

- * connectionClosed - this indicates that client has ended the session. Appropriate methods are called.

The command is executed on RDCServ and then sent to DeviceInterface. Other commands are sent directly to DeviceInterface.

If the vector is longer than 1, the rest of the data shall be examined only at the destination, not at every step in the middle.

3. CUSTOMIZING RDC FOR SPECIFIC TASK

The customizing is simple process. In most cases only the DeviceInterface needs to be subclassed and then the config file changed to reflect the change. Then, at runtime, the RDCServ loads the new class and is ready for the new task.

The same needs to be done at client side: ClientGUI needs to be subclassed and configuration modified.

These two classes are the ones that do the processing and therefore need to be coherent to each other. If the wrong DeviceInterface is used in pair with a ClientGUI then arbitrary actions may occur. Therefore ClientGUI should make sure it is connected to the correct server.

See API for details.

LISA 2: Programmi koodi listing

1. RDC BAASPROGRAMMIDE KOOD

1.1.RDC Server

1.1.1.RDCServ.java

```
package ee.ut.physic.rdc.server;
/*
 * RDCServ.java
 *
 */

/**
 *
 * @author laas
 */

import java.io.*;
import java.net.*;
import javax.net.*;
import javax.net.ssl.*;
import java.util.*;
import java.lang.reflect.Constructor;
import ee.ut.physic.rdc.server.*;

public class RDCServ {

    public boolean
DEBUG=(Boolean.valueOf(config.getString("DEBUG"))).booleanValue();

    /** The factory to create SSL Server Sockets */
    ServerSocketFactory serverSocketFactory=null;
    /** The main socket field */
    SSLServerSocket SSLserverSocket=null;
    /** The port to bind */
    public int bindPort=Integer.parseInt(config.getString("bindPort"));
    /** The number of clients to accept simultaneously. Value of -1 means
no limits */
    public int
MAX_CLIENTS=Integer.parseInt(config.getString("MAX_CLIENTS"));
    /** The number of clients currently running */
    int cur_clients=0;
    /** This communicates with devices */
    DeviceInterface deviceInterface=null;
    /** This communicates with clients */
    ServerThread serverThread=null;
```

```

/** This holds all threads to clients */
Hashtable clients=new Hashtable();
/** This makes it possible to redirect standard output. */
public PrintStream out=System.out;
/** This makes it possible to redirect error output */
public PrintStream err=System.err;

public Hashtable threadLock = new Hashtable();

/**
 * Use this string as command to get server output the data.
 * Sent to client on connect.
 */
public String serverOutputCommand =
config.getString("serverOutputCommand");

/** the configuration bundle */
private static java.util.ResourceBundle config =
java.util.ResourceBundle.getBundle("rdcserv");

/** Creates a new instance of RDCServ */
public RDCServ() {
//System.setProperty("javax.net.debug","all");
java.security.Security.addProvider(new
com.sun.net.ssl.internal.ssl.Provider());
try {
// get the default factory for SSL server sockets
if (DEBUG) out.println("Getting SSLServerSocketFactory");
serverSocketFactory = SSLServerSocketFactory.getDefault();
// create the server socket
if (DEBUG) out.println("About to create SSLServerSocket");
SSLserverSocket=getSSLServerSocket(bindPort);
SSLserverSocket.setNeedClientAuth(false);
out.println("Server started on port: "+bindPort);
if (DEBUG) out.println("Supported cipher suites:
"+SSLserverSocket.getSupportedCipherSuites());
} catch(Exception e){
System.err.println("Error binding port "+bindPort + ": "+e);
// copying data to redirected stream
if (!this.err.equals(System.err)){
err.println("Error binding port "+bindPort + ": "+e);
}
System.exit(1);
}
if (DEBUG) out.println("About to create DeviceInterface");
deviceInterface=(DeviceInterface)
getObject(config.getString("DeviceInterface"));

if (DEBUG) out.println("About to run run()");
run();
}

```

```

    /**
     * Returns ServerSocket, that is created using default
    SSLServerSocketFactory
     * @param port The port to bind
     * @returns ServerSocket bound to specified port
     */
    public SSLServerSocket getSSLServerSocket(int port) throws Exception{
        if (DEBUG) out.println("Creating SSLServerSocket");
        SSLServerSocket SSLsocket =
    (SSLServerSocket) serverSocketFactory.createServerSocket();
        SSLsocket.bind(new InetSocketAddress(port));
        SSLsocket.setEnabledCipherSuites(SSLsocket.getSupportedCipherSuites());
        if (DEBUG) out.println("SSLServerSocket created");
        return SSLsocket;
    }

    /**
     *This method handles the connecting clients. When client connects,
    this method
     *creates new threads to handle each client.
     */
    void run(){
        while (true){
            try{
                if (DEBUG) out.println("Waiting for clients...");
                SSLSocket SSLsocket =(SSLSocket) SSLserverSocket.accept();
                /** if MAX_CLIENTS == -1 then it is supposed to be
    infinity */
                if ((clients.size() < MAX_CLIENTS) || MAX_CLIENTS == -1){
                    try{
                        out.println("Client connected:
    "+SSLsocket.getInetAddress());
                        //ServerThread bst =
    (ServerThread) serverThread.newInstance();
                        if (DEBUG) out.println("About to create
    ServerThread");
                        ServerThread bst=(ServerThread)
    getObject(config.getString("ServerThread"));
                        if (DEBUG) out.println("ServerThread created..
    running setSocket()");
                        bst.setSocket(SSLsocket);
                        if (DEBUG) out.println("Generate name for thread
    and register it");
                        this.registerClient(bst.generateName(),bst);
                        bst.start();
                    } catch (Exception e){
                        err.println("Error while initializing connection
    with client: "+e);
                    }
                }
            } else { // no connection allowed
                out.println("Client rejected");
            }
        }
    }

```

```

        SSLsocket.close();
    }
    } catch (Exception e){
        err.println("Error accepting client: "+e);
    }
}

/**
 * Register the new thread so others can communicate to it.
 * @param name    Unique name of the client (generated by thread)
 * @param bst     The instance of ServerThread to register
 */
protected void registerClient(String name, ServerThread bst){
    if (DEBUG) out.println("Registrering client thread: "+name);
    this.clients.put(name, bst);
    this.threadLock.put(name, new Object());
    deviceInterface.newClient(name);
}

public void removeClient(String name){
    ServerThread st = (ServerThread) clients.get(name);
    if (st != null){
        if (!st.isClosing){
            out.println("Removing client thread: "+name);
            deviceInterface.clientLeft(name);
            st.isClosing=true;
            st.interrupt();
            st.closing();
            this.clients.remove(name);
            this.threadLock.remove(name);
            st=null;
        }
    }
}

/**
 * Send all client the same data.
 * @param data    The data to be sent.
 */
void broadcast(Vector data){
    Enumeration names = clients.keys();
    if (DEBUG) out.println("Sending to all clients: "+data);
    while (names.hasMoreElements()){
        this.sendToClient(data, (String)names.nextElement());
    }
}

/**
 * Send data to client.
 * @param data    the data to be sent
 * @param name    the name of the client it is meant to.
 */
public void sendToClient(Vector data, String name){
    if (DEBUG) out.println("Sending to client "+name+": "+data);
}

```

```

        ServerThread thread = (ServerThread) clients.get(name);
        thread.sendToClient(data);
    }

    /**
     * This redirects standard output to given stream.
     * @param pw New location to print to
     */
    public void redirectOutput(PrintStream pw){
        this.out=pw;
    }
    /**
     * This redirect error output to given stream.
     * <H2>NOTE:</H2> fatal errors still get printed to System.err
     * @param pw New location to print to.
     */
    public void redirectErrorOutput(PrintStream pw){
        this.err=pw;
    }
    /**
     *This method returns an instance of Object, that is loaded from a
     *specified class. To instantiate, it uses constructor taking one
     *argument - RDCServ parent.
     * @param classname the name of the class to be loaded
     * @returns the instance of Object loaded,
     */
    private Object getObject(String classname){
        Object o=null;
        try{
            if (DEBUG) out.println("Trying to get class: "+classname);
            Class c = Class.forName(classname);
            if (DEBUG) out.println("Trying to get constructor");
            Constructor cc=c.getConstructor(new Class[]
{this.getClass()});
            o= (Object) cc.newInstance(new Object[] {this});
        }catch (Exception e){
            System.err.println("Could not load class: "+ classname);
            System.err.println(e);
            e.printStackTrace();
            // copying data to redirected stream
            if (!this.err.equals(System.err)){
                err.println("Could not load class: "+ classname);
                err.println(e);
            }
            System.exit(1);
        }
        if (DEBUG) out.println("Returning instance");
        return o;
    }
    /**
     * The main method to start RDC server
     * @param args the command line arguments
     */

```

```

public static void main(String[] args) {
    System.out.println("Starting server...");
    new RDCServ();
}

/**
 * This method is used to preuse info coming from client.
 * @param cmd the datavector coming from client
 * @param sender the name of the thread handling this client.
 * This name is used to communicate to this client afterwards.
 */
public void parseCommand(Vector cmd, String sender) {
    String command = cmd.elementAt(0).toString();
    if (command.equals(serverOutputCommand)) {
        output(cmd);
    }
    else if (command.equals("connectionClosed")) {
        this.removeClient(sender);
    }
    else if (command.equals("clientOutputCommand")) {
        setClientOutputCommand(cmd.elementAt(1).toString(), sender);
        sendServerOutputCommand(sender);
    }
    deviceInterface.parseCommand(cmd, sender);
}

/**
 * Used to send serverOutputCommand to client, after recieving
 clientOutputCommand
 */
public void sendServerOutputCommand(String name) {
    if (DEBUG) output("Sending serverOutputCommand to client:
"+name);
    Vector v=new Vector();
    v.add("serverOutputCommand");
    v.add(serverOutputCommand);
    sendToClient(v, name);
}

/**
 * Sets the name of the command to be used for clientOutput.
 * @param cmd the command
 * @param name name of the client the command applies
 */
public void setClientOutputCommand(String cmd, String name) {
    (this.getClient(name)).clientOutputCommand=cmd;
    out.println("Client output command set to: "+cmd);
}

/**
 * This returns client with specified name.
 * @param name name of the client to be returned
 */

```

```

public ServerThread getClient(String name){
    return (ServerThread) this.clients.get(name);
}

/**
 * This method is used to output datavectors on server.
 * It expects the first element of vector to be the
serverOutputCommand
 * and thus skips the first element.
 */
public void output(Vector data){
    Enumeration elements = data.elements();
    String command = (String) elements.nextElement();
    String outputString="";
    while (elements.hasMoreElements()){
        outputString=(elements.nextElement()).toString();
    }
    output(outputString);
}

/**
 *This method is used to output Strings on server
 */
public void output(String outputString){
    out.println(outputString);
}
}

```

1.1.1 DeviceInterface.java

```

/*
 * DeviceInterface.java
 *
 *
 */

package ee.ut.physic.rdc.server;

/**
 *
 * @author laas
 */

import ee.ut.physic.rdc.server.*;
import java.io.*;
import java.util.*;

public abstract class DeviceInterface {

    /** The pointer to the server instance */
    RDCServ parent;
    /** The redirectable outputstream */

```

```

PrintStream out;

/** The default constructor */
public DeviceInterface(){}

/** Creates a new instance of DeviceInterface */
public DeviceInterface(RDCServ parent) {
    this.parent=parent;
    // this can be changed to redirect output
    this.out=System.out;
}

/**
 * Redirect output to somewhere else.
 * @param pw The printstream new output gets redirected to.
 */
public void redirectOutput(PrintStream ps){
    this.out=ps;
}

/** The method to parse commands coming from client. The RDCServ
sends
 * commands it does not recognize to this method, adding
 * the name of the thread sending the command.
 * <br>
 * In BasicDeviceInterface parseCommand does nothing but prints out
 * (or wherever the output is directed to) all data coming from
client.
 * It also informs the client of receiving the data.
 * @param sender the name of sender
 * @param cmd command sent from client
 */
public abstract void parseCommand(Vector cmd, String sender);

/**
 * This triggers events that should be done when new client connects
 * @param name the name of the client.
 */
public abstract void newClient(String name);

/**
 *This is used to trigger some cleanup processes when a client left
 * @param name the name of the client that left
 */
public abstract void clientLeft(String name);
}

```

1.1.2 BasicDeviceInterface.java

```

/*
 * BasicDeviceInterface.java

```

```

*
*/

package ee.ut.physic.rdc.server;

/**
 *
 * @author laas
 */

import ee.ut.physic.rdc.server.*;
import java.io.*;
import java.util.*;

public class BasicDeviceInterface extends DeviceInterface{

    /** The pointer to the server instance */
    RDCServ parent;
    /** The redirectable outputstream */
    PrintStream out;
    /** Creates a new instance of BasicDeviceInterface */
    public BasicDeviceInterface(RDCServ parent) {
        this.parent=parent;
        // this can be changed to redirect output
        this.out=System.out;
    }

    /**
     * Redirect output to somewhere else.
     * @param pw The printstream new output gets redirected to.
     */
    public void redirectOutput(PrintStream ps){
        this.out=ps;
    }

    /** The method to parse commands coming from client. The RDCServ
sends
     * commands it does not recognize to this method, adding
     * the name of the thread sending the command.
     * <br>
     * In BasicDeviceInterface parseCommand does nothing but prints out
     * (or wherever the output is directed to) all data coming from
client.
     * It also informs the client of receiving the data.
     * @param sender the name of sender
     * @param cmd command sent from client
     */
    public void parseCommand(Vector cmd, String sender) {
        this.out.println("Data from: "+sender+": "+cmd);
        Vector response=new Vector();
        response.addElement(parent.getClient(sender).clientOutputCommand)
;
        // just some junk message, and mirroring of sent data.

```

```

        response.addElement("Data recieved: "+cmd);
        parent.sendToClient(response, sender);
    }

    /**
     * This triggers events that should be done when new client connects.
     * Here this does nothing.
     * @param name the name of the client.
     */
    public void newClient(String name) {
    }

    /**
     * This is used to trigger some cleanup processes when a client left
     * @param name the name of the client that left
     */
    public void clientLeft(String name){
    }
}

```

1.1.3 ServerThread.java

```

/*
 * ServerThread.java
 *
 *
 */

package ee.ut.physic.rdc.server;

/**
 *
 * @author laas
 */

import ee.ut.physic.rdc.server.*;
import java.io.*;
import java.net.*;
import java.util.*;
import javax.net.ssl.*;

public class ServerThread extends Thread {
    /** The pointer to server instance */
    RDCServ parent;
    /** The client's socket */
    SSLSocket socket;
    /** Socket's inputstream */
    InputStream sin;
    /** Socket's outputstream */
    OutputStream sout;
    /** Stream to output objects to client */

```

```

ObjectOutputStream out;
/** Stream to input objects from client */
ObjectInputStream in;

/** This is set true, when RDCServ is closing this thread down */
public boolean isClosing=false;

/**
 * Use this string as command to get client to output data to its
output window.
 * Asked from client on connect.
 */
public String clientOutputCommand ="clientOutput";
/** When set false, the thread stops running */
boolean listening=true;

/** The default constructor. */
public ServerThread() {}

/** Creates a new instance of ServerThread */
public ServerThread(RDCServ parent) {
    this.parent=parent;
}

/** Generate unique name to be identified by. This may be based on
 * socket information, date, time, etc
 */
public String generateName() {
    String oldname=getName();
    String s=socket.toString()+String.valueOf((new
Date()).getTime());
    setName(s);
    parent.output(oldname +" is now known as: "+getName());
    return s;
}

/** This method is used by RDCServ to send data to client.
 */
public void sendToClient(Vector data) {
    if (parent.DEBUG) parent.out.println(getName()+" : Sending to
client: "+data);
    try{
        out.writeObject(data);
        out.flush();
    } catch (IOException ioe) {parent.err.println("Error sending
data: "+ioe);}
}

/** Sets the socket to be listened to */
public void setSocket(SSLSocket s) {

```

```

        if (parent.DEBUG) parent.out.println(getName()+" Setting sockets
for client");
        this.socket=s;
        try {
            if (parent.DEBUG) parent.out.println(getName()+" Setting up
datastreams");
            this.sin=socket.getInputStream();
            this.sout=socket.getOutputStream();
            if (parent.DEBUG) parent.out.println(getName()+" Setting up
objectInputStream");
            this.oin=new ObjectInputStream(socket.getInputStream());
            if (parent.DEBUG) parent.out.println(getName()+" Setting up
objectOutputStream");
            this.oot = new ObjectOutputStream(socket.getOutputStream());

        } catch (Exception e) {
            parent.err.println(getName()+" Error getting socket streams:
"+e);
            // kill thread
            this.listening=false;
        }

        if (parent.DEBUG) parent.out.println(getName()+" setSocket()
done");
    }

    /** This method runs the thread
    */
    public void run() {
        if (parent.DEBUG) parent.out.println(getName()+" Waiting for
input data...");
        BufferedReader reader = new BufferedReader(new
InputStreamReader(sin));
        listening: while (isListening()){
            try{
                Object o=oin.readObject();

                if (o instanceof Vector){
                    parent.parseCommand((Vector)o, getName());
                }

            }
            catch (Exception e){
                parent.err.println(getName()+" Error reading object:
"+e);
                break listening;
            }

        }
        parent.output("Connection closed: "+getName());
        parent.removeClient(getName());
    }

    public boolean isListening(){

```

```

        if (isClosing) return false;
        synchronized (parent.threadLock.get(getName())){
            return listening;
        }
    }

    /** This method closes all open connections
     */
    public void closing() {
        synchronized (parent.threadLock.get(getName())){
            try{

                listening=false;
                sin.close();
                sout.close();
                oin.close();
                oout.close();
                socket.close();
            } catch (Exception e) {parent.err.println(getName()+" : Error
closing streams: "+e);}
        }
    }
}

```

1.2 RDC Client

1.2.1 RDCClient.java

```
/*
 * RDCClient.java
 *
 *
 */

package ee.ut.physic.rdc.client;

/**
 *
 * @author laas
 */

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.net.*;
import javax.net.ssl.*;
import java.util.*;
import java.lang.reflect.Constructor;

public class RDCClient extends java.applet.Applet {

    /** The SocketFactory to create socket to server */
    SSLSocketFactory socketFactory = null;
    /** The socket connected to server */
    SSLSocket socket = null;
    /** The port used for connection */
    int port=Integer.parseInt(config.getString("port"));
    /** The string, that servre can use to output data on client. Sent to
server on connect */
    String clientOutputCommand=config.getString("clientOutputCommand");
    /** The string that is sent from server and used for server basic
output */
    String serverOutputCommand="serverOutput";
    /** Socket's outputstream */
    OutputStream sos;
    /** Socket's inputstream */
    InputStream sis;
    /** ClientThread used to monitor for input data */
    ClientThread clientThread;
    /** clientGUI used for user input */
    ClientGUI clientGUI;

    /** The object used to synchronize threads */
    public Object threadLock = new Object();
}
```

```

    /** whether or not print debug data */
    public boolean DEBUG =
    (Boolean.valueOf(config.getString("DEBUG"))).booleanValue();

    /** Initializes the applet RDCCClient */
    public void init() {
        // Add new security provider
        //java.security.Security.addProvider(new
com.sun.net.ssl.internal.ssl.Provider());
        initComponents();
        output("Client is starting up...");
        if (DEBUG) output("Creating new instance of clientGUI");
        socketFactory = (SSLSocketFactory)SSLSocketFactory.getDefault();
        output("Connecting to server ...");
        try{
            socket = getSSLSocket(port);
            socket.setEnabledCipherSuites(socket.getSupportedCipherSuites
());
            try{
                if (DEBUG) output("Getting outputstream");
                sos=socket.getOutputStream();
                if (DEBUG) output("Getting inputstream");
                sis=socket.getInputStream();
            } catch (IOException ioe){output("Error getting streams:
"+ioe);}

            if (DEBUG) output("Creating new instance of ClientThread");
            clientThread = (ClientThread)
getObject(config.getString("ClientThread"));
            if (DEBUG) output("Seting clientThread's streams");
            clientThread.setStreams(sis, sos);
            if (DEBUG) output("Starting the thread...");
            clientThread.start();
            clientGUI = (ClientGUI)
getObject(config.getString("ClientGUI"));
            add(basePanel, java.awt.BorderLayout.CENTER);
            clientGUI.startCommunication();
        } catch (Exception e){}
    }

    /**
     * This returns the panel on witch all client GUI is built on. This
panel can be
     * modified by clientGUI itself, adding or removing components, etc.
     * @returns the basePanel that contains all of user configurable GUI
elements.
     */
    public Panel getPanel(){
        return basePanel;
    }

    /**

```

```

    * This is used to send data to server.
    */
public void sendToServer(Vector data){
    clientThread.sendToServer(data);
    if (DEBUG) output("Sent to server: "+data);
}

/** This method is used to output data on server */
public void outputOnServer(String text){
    if (DEBUG) output("Outputting on server: "+text);
    Vector v=new Vector();
    v.add(serverOutputCommand);
    v.add(text);
    sendToServer(v);
}

/**
 * Parses the command. If the first element of input data is
clientOutputCommand
 * then the data is outputted using output(), else the command is
forwarded to clientGUI instance.
 */
public void parseCommand(Vector data){
    String cmd = (String) data.elementAt(0);
    if (cmd.equals(this.clientOutputCommand)) output(data);
    else if (cmd.equals("serverOutputCommand"))
setServerOutputCommand((String)data.elementAt(1));
    else if (cmd.equals("connectionClosed")){
        this.stop();
    }
    clientGUI.parseCommand(data);
}

public void setServerOutputCommand(String command){
    this.serverOutputCommand=command;
    if (DEBUG) output("Server output command set to: "+command);
}

/**
 * Output to the clientOutputArea. This outputs a data vector.
 */
public void output(Vector data){
    Enumeration elements = data.elements();
    String command = (String) elements.nextElement();
    while (elements.hasMoreElements()){
        output((String)elements.nextElement());
    }
}

/**
 * Output to the clientOutputArea. This outputs a given string.
 */
public void output(java.lang.String text) {
    if (text != null) clientOutputArea.append("\n"+text);
}

```

```

        // put caret at the end of the textArea
        try{
            clientOutputArea.setCaretPosition((clientOutputArea.getText()
).length());
        } catch (Exception e){} // an exception is thrown when closing
connections
        if (DEBUG) System.out.println(text);
    }

/**
 * Get's the SSL socket to webserver.
 */
public SSLSocket getSSLSocket(int p) {
    if (DEBUG) output("Trying to get socket to port: "+p);
    if (DEBUG) System.out.println("The server:
"+this.getCodeBase().getHost());
    SSLSocket sock=null;
    try {
        sock=(SSLSocket) socketFactory.createSocket(this.getCodeBase()
.getHost(),p);

        } catch (IOException ioe){output("Error creating socket: "+ioe);}
    return sock;
}

/**
 * This method is used to get an object from a string representation
of classname.
 * It uses a prespecified constructor and passes it a single argument
- this (the
 * pointer to the instance of RDCClient).
 * @param    classname    string representation of classname to be
instantiated
 * @returns  the instance of that class represented as an instance of
Object
 */
private Object getObject(String classname){
    Object o=null;
    try{
        Class c = Class.forName(classname);
        Constructor cc=c.getConstructor(new Class[]
{this.getClass()});
        o= (Object) cc.newInstance(new Object[] {this});
    }catch (Exception e){
        System.err.println("Could not load class: "+ classname);
        System.err.println(e);
        System.exit(1);
    }
    return o;
}

/**
 * Handles the visibility option of ClientOutputArea.

```

```

    */
protected void doClientOutputAreaVisibility() {
    boolean visible = clientOutputArea.isVisible();
    setVisibilityOfClientOutputArea(!visible);
    // the outputArea got set visible
    if (!visible) clientOutputButton.setLabel("Hide Output Area");
    // the outputArea got set invisible
    else clientOutputButton.setLabel("Show Output Area");
}

/**
 * Sets clientOutputArea visibility. Can be called from external
classes
 * to handle the output area's visibility at startup.
 */
public void setVisibilityOfClientOutputArea(boolean visible) {
    clientOutputArea.setVisible(visible);
    validate();
}

/**
 * This method is called by appletcontext to indicate the closure of
applet.
 * This method should close all open connections and threads
 */
public void stop() {
    if (!this.clientThread.isClosing) {
        Vector v = new Vector();
        v.add("connectionClosed");
        sendToServer(v);

        try {
            this.clientThread.isClosing=true;
            this.clientThread.closing();
            this.socket.close();
        } catch (IOException ioe) {output("Had errors while closing
connection...");}
    }
}

/** This method is called from within the init() method to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() { //GEN-BEGIN:initComponents
    basePanel = new java.awt.Panel();
    clientOutputPanel = new java.awt.Panel();
    toolbar = new java.awt.Panel();
    clientOutputButton = new java.awt.Button();
    clientOutputArea = new java.awt.TextArea();

    setLayout(new java.awt.BorderLayout());

```

```

        basePanel.setLayout(new java.awt.BorderLayout());

        add(basePanel, java.awt.BorderLayout.CENTER);

        clientOutputPanel.setLayout(new java.awt.BorderLayout());

        toolbar.setLayout(new java.awt.BorderLayout());

        clientOutputButton.setLabel("Hide Output Area");
        clientOutputButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                clientOutputButtonActionPerformed(evt);
            }
        });

        toolbar.add(clientOutputButton, java.awt.BorderLayout.EAST);

        clientOutputPanel.add(toolbar, java.awt.BorderLayout.NORTH);

        clientOutputArea.setEditable(false);
        clientOutputPanel.add(clientOutputArea,
java.awt.BorderLayout.CENTER);

        add(clientOutputPanel, java.awt.BorderLayout.NORTH);

    }//GEN-END:initComponents

    private void
clientOutputButtonActionPerformed(java.awt.event.ActionEvent evt) { //GEN-
FIRST:event_clientOutputButtonActionPerformed
        doClientOutputAreaVisibility();
    }//GEN-LAST:event_clientOutputButtonActionPerformed

    // Variables declaration - do not modify//GEN-BEGIN:variables
    private java.awt.Panel clientOutputPanel;
    private java.awt.Panel basePanel;
    private java.awt.Panel toolbar;
    private java.awt.TextArea clientOutputArea;
    private java.awt.Button clientOutputButton;
    // End of variables declaration//GEN-END:variables

    private static final java.util.ResourceBundle config =
java.util.ResourceBundle.getBundle("rdcclient");
}

```

1.2.2 ClientThread.java

```
/*
 * ClientThread.java
 *
 */

package ee.ut.physic.rdc.client;

/**
 *
 * @author laas
 */

import java.util.*;
import java.io.*;

public abstract class ClientThread extends Thread {

    /** InputStream used to read from server */
    InputStream sis;
    /** OutputStream used to write to server */
    OutputStream sos;
    /** ObjectInputStream used to receive datavectors from server */
    ObjectInputStream ois;
    /** ObjectOutputStream used to send datavectors to server */
    ObjectOutputStream oos;
    /** The parent of this thread */
    RDCCClient parent;
    /** until true, listen for incoming data */
    boolean listening=true;

    /** changed by parent, when this thread is going to be closed */
    public boolean isClosing = false;

    /** default constructor. Does not do anything, exists only for
    extending */
    public ClientThread(){}

    /** Creates a new instance of ClientThread */
    public ClientThread(RDCCClient parent){
        this.parent=parent;
    }

    public abstract void run();

    /** This method is used by RDCServ to send data to client.
    */
    public abstract void sendToServer(Vector data);

    /** Sets the streams used in communication */
    public abstract void setStreams(InputStream in, OutputStream out);
}
```

```

    /** This method closes all open connections
    */
    public abstract void closing();

    /**
    *This method is used to synchronously get boolean listening. In
RDCCClient
    * there is an Object threadLock that can be used for this purpose.
    *@returns      the value of listening.
    */
    public abstract boolean isListening();
}

```

1.2.3 BasicClientThread.java

```

/*
 * BasicClientThread.java
 *
 *
 */

package ee.ut.physic.rdc.client;

/**
 *
 * @author laas
 */

import java.util.*;
import java.io.*;
import ee.ut.physic.rdc.client.*;

public class BasicClientThread extends ClientThread {

    /** InputStream used to read from server */
    InputStream sis;
    /** OutputStream used to write to server */
    OutputStream sos;
    /** ObjectInputStream used to receive datavectors from server */
    ObjectInputStream ois;
    /** ObjectOutputStream used to send datavectors to server */
    ObjectOutputStream oos;
    /** The parent of this thread */
    RDCCClient parent;
    /** until true, listen for incoming data */
    boolean listening=true;

    /** Creates a new instance of BasicClientThread */
    public BasicClientThread(RDCCClient parent) {

```

```

        this.parent=parent;
    }

    public void run() {
        Vector v = new Vector();
        v.add("clientOutputCommand");
        v.add(parent.clientOutputCommand);
        parent.sendToServer(v);
        Vector indata;
        if (parent.DEBUG) parent.output("Waiting for incoming data...");
        while (isListening()){
            try{
                indata =(Vector) ois.readObject();
                parent.parseCommand(indata);
            }
            catch (NullPointerException npe) {} // Sometimes
deserialization throws a NullPointerException. Similar problem seems to
be reported on GCJ from GCC 3.1, Linux
            catch(Exception e){
                parent.output("Error reading data: "+e);
            }

        }

        }
        parent.output("Connection closed.");
        parent.stop();
    }

    /**
     * This method is used to synchronously get the boolean Listening
     */
    public boolean isListening(){
        synchronized (parent.threadLock){
            return listening;
        }
    }

    /** This method is used by RDCServ to send data to client.
     */
    public void sendToServer(Vector data) {
        try {
            oos.writeObject(data);
            oos.flush();
            if (parent.DEBUG) parent.output("Data is on the way");
        } catch (IOException ioe) {parent.output("Error sending data:
"+ioe);}
    }

    /** Sets the streams used in communication */
    public void setStreams(InputStream in, OutputStream out) {
        if (parent.DEBUG) parent.output("Setting inputstream");
    }

```

```

        this.sis=in;
        if (parent.DEBUG) parent.output("Setting outputstream");
        this.sos=out;
        try{
            if (parent.DEBUG) parent.output("Setting
ObjectOutputStream");
            oos=new ObjectOutputStream(sos);
            if (parent.DEBUG) parent.output("Setting ObjectInputStream");
            ois=new ObjectInputStream(sis);

        } catch (IOException ioe){parent.output("Error setting streams:
"+ioe);}
        if (parent.DEBUG) parent.output("All streams set up");
    }

    /** This method closes all open connections */
    public void closing() {
        synchronized (parent.threadLock){
            try{
                listening=false;
                sis.close();
                sos.close();
                ois.close();
                oos.close();
            } catch (Exception e) {parent.output("Error closing streams:
"+e);}
        }
    }
}

```

1.2.4 ClientGUI.java

```

/*
 * ClientGUI.java
 *
 *
 */

package ee.ut.physic.rdc.client;

/**
 *
 * @author laas
 */

import ee.ut.physic.rdc.client.*;
import java.util.*;
import java.awt.event.*;
import java.awt.*;

```

```

public abstract class ClientGUI extends java.awt.Panel {
    /** The parent of this object */
    RDCCClient parent;

    /** The default constructor. Never used, exists only for extending.
    */
    public ClientGUI(){

    }

    /** Creates new form ClientGUI. To add this instance to parent
    applet, this
    * class should call parent.getPanel() and add it to the panel.
    * The returned panel has a default layout that is instance of
    BorderLayout.*/
    public ClientGUI(RDCCClient parent) {
        this.parent = parent;
    }

    public abstract void sendToServer(Vector data);

    /** Method used to parse commands sent from server and not recognized
    by parent */
    public abstract void parseCommand(Vector data);

    /** This method is called by parent to close all open connections and
    such */
    public abstract void closing();

    /** This method is used to trigger some init-time actions to take
    place.
    * This is called from parent as a last thing on startup.
    */
    public abstract void startCommunication();
}

```

1.2.5 BasicClientGUI.java

```

/*
 * BasicClientGUI.java
 *
 * Created on pühapäev, 25. Mai 2003. a, 14:08
 */

package ee.ut.physic.rdc.client;

/**
 *
 * @author laas
 */

```

```

import ee.ut.physic.rdc.client.*;
import java.util.*;
import java.awt.event.*;
import java.awt.*;

public class BasicClientGUI extends ClientGUI {
    /** The parent of this object */
    RDCCClient parent;
    /** Creates new form BasicClientGUI */
    public BasicClientGUI(RDCCClient parent) {
        this.parent = parent;
        initComponents();
        parent.getPanel().add(this, BorderLayout.CENTER);
    }

    public void sendToServer(Vector data){
        parent.sendToServer(data);
    }

    /** Method used to parse commands sent from server and not recognized
    by parent */
    public void parseCommand(Vector data){

    }

    /** This method is called by parent to close all open connections and
    such */
    public void closing(){

    }

    /** This method is called from within the constructor to
    * initialize the form.
    * WARNING: Do NOT modify this code. The content of this method is
    * always regenerated by the Form Editor.
    */
    private void initComponents() { //GEN-BEGIN:initComponents
        sendButton = new java.awt.Button();

        setLayout(new java.awt.BorderLayout());

        sendButton.setLabel("Send current time");
        sendButton.addActionListener(new java.awt.event.ActionListener()
{
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                sendButtonActionPerformed(evt);
            }
        });

        add(sendButton, java.awt.BorderLayout.NORTH);

    } //GEN-END:initComponents

```

```

    private void sendButtonActionPerformed(java.awt.event.ActionEvent
evt) { //GEN-FIRST:event_sendButtonActionPerformed
        parent.output("Sending the current time to server...");
        Vector v=new Vector();
        v.add("currentTime");
        Calendar cal = Calendar.getInstance();
        v.add("The current time is: "+cal.get(Calendar.HOUR_OF_DAY)
+":"+cal.get(Calendar.MINUTE)+":"+cal.get(Calendar.SECOND));
        this.sendToServer(v);
    } //GEN-LAST:event_sendButtonActionPerformed

    /** This method is used to trigger some init-time actions to take
place.
    * This is called from parent as a last thing on startup.
    */
    public void startCommunication() {
    }

    // Variables declaration - do not modify //GEN-BEGIN:variables
    private java.awt.Button sendButton;
    // End of variables declaration //GEN-END:variables
}

```

2 PSCC juhtimisprogrammide koodid (nimetus arvutiklastrite järgi)

2.1 Cluster Client

2.1.1 ClusterClient.java

```
/*
 * ClusterClient.java
 *
 * Created on esmaspäev, 26. Mai 2003. a, 12:24
 */

package ee.ut.physic.rdc.clusterclient;

/**
 *
 * @author laas
 */

import java.util.*;
import ee.ut.physic.rdc.client.*;
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.applet.*;

public class ClusterClient extends ClientGUI {
    /** The instance of RDCCClient */
    public RDCCClient parent;

    private String askConsolePort = "consolePort";
    private String askNumberOfComputers = "numberOfComputers";

    /** the password to be used for connecting to console port */
    private String setConsolePasswd = "consolePasswd";

    /** The port on what console service is listening */
    private int consolePort;
    /** The passwd to be used when connecting to console.
     * It is not for security, but for making sure, the right
     * client is connecting to console port
     */
    private String consolePasswd;

    /** the number of computers to be controlled */
    private int computers=-1;

    /** This holds pointers to all buttons */
}
```

```

Hashtable buttons=new Hashtable();

/** This is set to true, if console is showing */
public boolean consoleShowing=false;

/** The TextArea that displays console output data */
public TextArea consoleOutputArea=new TextArea();
/** The inputbox for console */
public TextField consoleInput;
/** The frame for console */
public Frame consoleFrame=new Frame("Console window");

/** Creates new form ClusterClient */
public ClusterClient(RDCCClient parent) {
    this.parent=parent;
    initComponents();
    parent.getPanel().add(this, BorderLayout.CENTER);
}

/** This method is called from within the constructor to
 * initialize the form.
 * WARNING: Do NOT modify this code. The content of this method is
 * always regenerated by the Form Editor.
 */
private void initComponents() { //GEN-BEGIN:initComponents
    loginPanel = new java.awt.Panel();
    loginButton = new java.awt.Button();
    forceLoginButton = new java.awt.Button();
    logoutButton = new java.awt.Button();
    statusButton = new java.awt.Button();
    connectMainButton = new java.awt.Button();
    rebootMainButton = new java.awt.Button();
    rebootSelfButton = new java.awt.Button();
    rebootDeviceButton = new java.awt.Button();
    buttonScroll = new java.awt.ScrollPane();
    buttonPanel = new java.awt.Panel();

    setLayout(new java.awt.BorderLayout());

    loginPanel.setLayout(new java.awt.GridLayout(2, 4, 5, 5));

    loginButton.setActionCommand("login");
    loginButton.setLabel("Login");
    loginButton.addActionListener(new java.awt.event.ActionListener()
{
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        buttonActionPerformed(evt);
    }
});

    loginPanel.add(loginButton);

    forceLoginButton.setActionCommand("force");
    forceLoginButton.setLabel("Force Login");

```

```

        forceLoginButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonActionPerformed(evt);
            }
        });

        loginPanel.add(forceLoginButton);

        logoutButton.setActionCommand("logout");
        logoutButton.setLabel("Logout");
        logoutButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonActionPerformed(evt);
            }
        });

        loginPanel.add(logoutButton);

        statusButton.setActionCommand("status");
        statusButton.setLabel("Status");
        statusButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonActionPerformed(evt);
            }
        });

        loginPanel.add(statusButton);

        connectMainButton.setActionCommand("connect main");
        connectMainButton.setLabel("Connect Main");
        connectMainButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonActionPerformed(evt);
            }
        });

        loginPanel.add(connectMainButton);

        rebootMainButton.setActionCommand("reboot main");
        rebootMainButton.setLabel("Reboot Main");
        rebootMainButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonActionPerformed(evt);
            }
        });

        loginPanel.add(rebootMainButton);

        rebootSelfButton.setActionCommand("reboot self");

```

```

        rebootSelfButton.setLabel("Reboot Self");
        rebootSelfButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonActionPerformed(evt);
            }
        });

        loginPanel.add(rebootSelfButton);

        rebootDeviceButton.setActionCommand("reboot device");
        rebootDeviceButton.setLabel("Reboot Device");
        rebootDeviceButton.addActionListener(new
java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                buttonActionPerformed(evt);
            }
        });

        loginPanel.add(rebootDeviceButton);

        add(loginPanel, java.awt.BorderLayout.NORTH);

        buttonPanel.setLayout(new java.awt.GridLayout(1, 1));

        buttonScroll.add(buttonPanel);

        add(buttonScroll, java.awt.BorderLayout.CENTER);

    } //GEN-END: initComponents

    private void buttonActionPerformed(java.awt.event.ActionEvent evt)
{ //GEN-FIRST: event_buttonActionPerformed
        buttonAction(evt);
    } //GEN-LAST: event_buttonActionPerformed

    public void buttonAction(java.awt.event.ActionEvent evt) {
        String actionCommand = evt.getActionCommand();
        Vector v = new Vector();
        v.add("controller");
        v.add(actionCommand);
        parent.sendToServer(v);
    }

    /** This method is called by parent to close all open connections and
such */
    public void closing() {
    }

    /** Method used to parse commands sent from server and not recognized
by parent */
    public void parseCommand(Vector data) {
        String command = (String)data.elementAt(0);
        if (parent.DEBUG) parent.output("Command recieved: "+command);
    }

```

```

        if (command.equals(askNumberOfComputers))
setNumberOfComputers(data);
        if (command.equals("consoleReady")) startConsole(data);
        if (command.equals("serialText")) sendToConsole(data);

    }

    public void sendToServer(Vector data) {
    }

    /** This method is used to trigger some init-time actions to take
place.
    * This is called from parent as a last thing on startup.
    */
    public synchronized void startCommunication() {
        askNumberOfComputers();

        while (computers == -1){
            try{
                wait(10000);
            } catch (InterruptedException ie){}
        }
        if (computers == -1) computers = 0;
        createButtonPanel();
    }

    private void createButtonPanel(){
        buttonPanel.removeAll();
        buttonPanel.setLayout(new GridLayout(computers,1,5,5));
        String restart ="Restart ";
        String check = "Check ";
        String getConsole = "Get console ";
        for (int i =1; i<=computers;i++){
            String number =
leftPad(String.valueOf(i),String.valueOf(computers).length()," ");
            Panel p = new Panel();
            p.setLayout(new FlowLayout(FlowLayout.CENTER,25,5));
            // Restart button
            String name=restart+number;
            Button b=new Button(name);
            b.setActionCommand("reboot "+i);
            buttons.put(name,b);
            b.addActionListener(new ActionListener(){
                public void actionPerformed(ActionEvent ae){
                    buttonAction(ae);
                }
            });
            p.add(b);

            // Get console button
            name=getConsole+number;

```

```

        b=new Button(name);
        buttons.put(name,b);
        b.setActionCommand("connect "+i);
        b.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent ae){
                buttonAction(ae);
            }
        });
        p.add(b);

        buttonPanel.add(p);
    }

    buttonScroll.add(buttonPanel);

    this.validateTree();
}

/** Pads a string from left with padding to ensure specified length.
 * @param s the string
 * @param length the wished length
 * @param pad the padding
 * @returns the padded string
 */
public String leftPad(String s,int length,String pad){
    String ns = new String(s);
    while (ns.length()< length ){
        ns=pad+ns;
    }
    return ns;
}

private void askNumberOfComputers(){
    Vector v=new Vector();
    v.add(askNumberOfComputers);
    parent.sendToServer(v);
}

private synchronized void setNumberOfComputers(Vector data){
    computers=Integer.parseInt((String)data.elementAt(1));
    notifyAll();
    parent.output("Computers: "+computers);
}

public void startConsole(Vector data){
    parent.output("Starting console");
    consoleFrame.removeAll();
    Panel panel = new Panel();
    consoleOutputArea = new TextArea();
    consoleOutputArea.setEditable(false);
    panel.setLayout(new BorderLayout());
    panel.add(consoleOutputArea,BorderLayout.CENTER);
}

```

```

        consoleInput=new TextField();
        panel.add(consoleInput, BorderLayout.SOUTH);
        consoleInput.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ae) {
                Vector v=new Vector();
                v.add("consoleInput");
                v.add(((TextField)ae.getSource()).getText());
                parent.sendToServer(v);
                ((TextField)ae.getSource()).setText("");
            }
        });
        consoleFrame.setSize(500,500);
        consoleFrame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent we) {
                consoleShowing=false;
                Vector v=new Vector();
                v.add("consoleClosed");
                parent.sendToServer(v);
                we.getWindow().setVisible(false);
            }
        });
        consoleFrame.add(panel);
        consoleFrame.show();
        consoleShowing=true;
    }

    public void sendToConsole(Vector data) {
        if (consoleShowing) {
            String text = (String) data.elementAt(1);
            consoleOutputArea.append("\n"+text);
            consoleOutputArea.setCaretPosition(consoleOutputArea.getText (
).length());
        }
    }
}

// Variables declaration - do not modify//GEN-BEGIN:variables
private java.awt.Button rebootSelfButton;
private java.awt.Button rebootMainButton;
private java.awt.Button connectMainButton;
private java.awt.ScrollPane buttonScroll;
private java.awt.Button loginButton;
private java.awt.Button rebootDeviceButton;
private java.awt.Button forceLoginButton;
private java.awt.Button logoutButton;
private java.awt.Panel loginPanel;
private java.awt.Button statusButton;
private java.awt.Panel buttonPanel;
// End of variables declaration//GEN-END:variables
}

```

2.2 Cluster Server

2.2.1 ClusterServer.java

```
/*
 * ClusterServer.java
 *
 * Created on esmaspäev, 26. Mai 2003. a, 10:52
 */

package ee.ut.physic.rdc.clusterserv;

/**
 *
 * @author laas
 */
import ee.ut.physic.rdc.server.*;
import java.util.*;
import java.io.*;

public class ClusterServer extends DeviceInterface {

    protected static final java.util.ResourceBundle config =
java.util.ResourceBundle.getBundle("clusterserv");

    /** The pointer to the server instance */
    RDCServ parent;
    /** The redirectable outputstream */
    PrintStream out;
    /** The number of computers */
    int computers;

    /** the name of program to be used for getting serial console */
    String consoleProgram=config.getString("consoleProgram");
    /** the name of the program used for controlling of device */
    String controller=config.getString("controller");
    /** This string is searched for in controller feedback, when program
waits for console */
    String successfulConnectString =
config.getString("successfulConnectString");

    /** The name of the client, that is currently using the console, ""
if none. */
    String consoleOwner ="";

    /** The number of the console currently connected */
    String currentConsole = null;

    /** The timeout in minutes, when client is disconnected */
```

```

    long timeout =
(Long.parseLong(config.getString("clientTimeout")))*60000;

    /** name to timestamp table for timer */
    Hashtable timeHashTable = new Hashtable();
    /** timestamp to name table for timer. This is also sorted by time,
     * so it is easy to discover timeouts */
    TreeMap timeTreeMap = new TreeMap();

    /** The timer that disconnects clients that are overdue */
    ClusterTimer clusterTimer;
    /** The processListener for the controller process */
    ProcessListener controllerListener;
    /** The processListener for the console process */
    ProcessListener consoleListener;

    /** Indicates that the command executed is connect command
     * and program should search for successful messages
     */
    boolean waitForConnect=false;

    /**
     * This creates new instance of ClusterServer
     * @param parent the instance of RDCServ that calls this
    constructor
     */
    public ClusterServer(RDCServ parent) {
        this.parent=parent;
        checkForPrograms();

        // this can be changed to redirect output
        this.out=System.out;
        computers=Integer.parseInt(config.getString("computers"));
        clusterTimer=new ClusterTimer(this);
        clusterTimer.start();

    }

    /**
     * This method is used to check, that the required programs exist as
    files.
     * If they do not, the whole server is shut down.
     */
    private void checkForPrograms() {
        try{
            StringTokenizer st = new StringTokenizer(controller);
            if (!(new File(st.nextToken())).exists()) {
                System.err.println("FATAL ERROR: the controller program
does not exist: "+controller);
                System.exit(1);
            }
        }
    }

```

```

        st = new StringTokenizer(consoleProgram);
        if (!(new File(st.nextToken()).exists())) {
            System.err.println("FATAL ERROR: the console program does
not exist: "+consoleProgram);
            System.exit(1);
        }
    } catch (Exception e) {System.err.println("Error checking for
programs: "+e);
        System.exit(1);
    }
}

/** The method to parse commands coming from client. The RDCServ
sends
* commands it does not recognize to this method, adding
* the name of the thread sending the command.
* <br>
* In BasicDeviceInterface parseCommand does nothing but prints out
* (or wherever the output is directed to) all data coming from
client.
* It also informs the client of receiving the data.
* @param sender the name of sender
* @param cmd command sent from client
*/
public void parseCommand(Vector cmd, String sender) {
    resetTimer(sender);
    String command=(String)cmd.elementAt(0);
    if (command.equals("numberOfComputers"))
sendNumberOfComputers(sender);
    else if (command.equals("controller")) controller(cmd, sender);
    else if (command.equals("consoleInput")) consoleInput(cmd);
    else if (command.equals("consoleClosed")) closeConsole();
    else out.println(command);
}

/**
*This method sends commands to the controller process
*@param data the datavector containing the commands
*@param sender the name of the client who sent the data
*/
public void controller(Vector data, String sender){
    consoleOwner=sender; // this is allowed, because
ClusterServer allows only one client at a time.
    String command = (String) data.elementAt(1);
    try {
        if (command.indexOf("connect") >=0){
            if ((currentConsole != null)) closeConsole();
            currentConsole = command;
            waitForConnect = true;
        }
        controllerListener.write(command);
    } catch (IOException ioe) {out.println("Error sending command to

```

```

controller: "+ioe);}
    }

    /**
     *Sends the number of computers connected to this server.
     *@param sender the name of client
     */
    private void sendNumberOfComputers(String sender){
        Vector v=new Vector();
        v.add("numberOfComputers");
        v.add(String.valueOf(computers));
        parent.sendToClient(v, sender);
    }

    /**
     * This is used to execute the logout command (useful, when client
    timed out)
     *@returns the output of logout command
     */
    private void doLogout(){
        try {
            controllerListener.write("logout");
            consoleListener.stop();
        } catch (Exception e) {out.println("Error logging out: "+e);}
    }

    /**
     *This method sends text to client. The text is outputted on client
    bascreen
     *@param text the text to be sent
     */
    public void sendText(String text){
        Vector v=new Vector();
        v.add(parent.getClient(consoleOwner).clientOutputCommand);
        v.add(text);
        parent.sendToClient(v, consoleOwner);
        if (waitForConnect){
            waitForConnect = false;
            if (text.indexOf(successfulConnectString) >=0)
doGetConsole();
        }
    }

    /**
     *This method prepares the console for communication and informs the
    client.
     */
    void doGetConsole(){
        Vector v = new Vector();
        v.add("consoleReady");
        parent.sendToClient(v, consoleOwner);
        consoleListener = new ProcessListener(this, consoleProgram, 2);
    }

```

```

}

/**
 *This method closes the console.
 */
public void closeConsole(){
    try {
        controllerListener.write("disconnect");
        consoleListener.stop();
    } catch (Exception e) {out.println("Error closing console: "+e);}
}

/**
 *This method sends console output to client
 *@param    text    the console output
 */
public void consoleOutput(String text){
    Vector v = new Vector();
    v.add("serialText");
    v.add(text);
    parent.sendToClient(v,consoleOwner);
}

/** This method sends client input to console.
 *@param    data    the datavector containing the text to be sent to
console
 */
public void consoleInput(Vector data){
    try{
        consoleListener.write((String)data.elementAt(1));
    } catch (IOException ioe){out.println("Error sending command to
console: "+ioe);}
}

/**
 * Resets the timer for the client.
 *@param    sender    the name of sender
 */
public void resetTimer(String sender) {
    Long oldtime = (Long)timeHashTable.get(sender);
    Long newtime = new Long(new Date().getTime()+timeout);
    timeTreeMap.remove(oldtime);
    timeHashTable.remove(sender);
    timeHashTable.put(sender, newtime);
    timeTreeMap.put(newtime, sender);
}

/**
 * This triggers events that should be done when new client connects.
 * Here this starts new Timer for this client to disconnect it when
it timeouts.
 *@param    name    the name of the client.
 */

```

```

public void newClient(String name) {
    // the time when this client is timed out
    consoleOwner = name;
    Long timeoutTime = new Long(new Date().getTime()+timeout);
    timeHashTable.put(name,timeoutTime);
    timeTreeMap.put(timeoutTime,name);
    controllerListener = new ProcessListener(this, controller,1);
}

/** This is used by Timer to disconnect the timeout client */
public void disconnectClient (Long timestamp) {
    String name = (String) timeTreeMap.get(timestamp);
    //if (consoleOwner.equals(name)) resetTimer(name);
    //else{
        Vector v = new Vector();
        v.add(((ServerThread)parent.getClient(name)).clientOutputComm
and);
        v.add("Connection timed out.. closed");
        parent.sendToClient(v, name);
        v = new Vector();
        v.add("connectionClosed");
        parent.sendToClient(v, name);
        parent.removeClient(name);
    //}
}

/**
 *This is used to trigger some cleanup processes when a client left
 *@param name the name of the client that left
 */
public void clientLeft(String name){
    doLogout();
    Long timestamp = (Long)timeHashTable.get(name);
    try{
        controllerListener.stop();
        timeHashTable.remove(name);
        timeTreeMap.remove(timestamp);
    } catch (Exception e) {}
}

}

/**
 * ClusterTimer is used to disconnect timed out clients.
 */

class ClusterTimer extends Thread {

    ClusterServer parent;
    public boolean running=true;

```

```

    /** This creates new instance of ClusterTimer */
    public ClusterTimer(ClusterServer parent){
        this.parent=parent;
    }

    /** This checks clients for being idle for too long. The interval is
    5 sec */
    public void run(){
        search:
        while (running){
            if (parent.timeTreeMap.size() <1) {
                do_sleep(2000);          // Sleep for 2 secs if no clients
                continue search;
            }
            long now = new Date().getTime();
            Long firstLong = (Long)
parent.timeTreeMap.firstKey();          // the Long value of timestamp
            long first =
firstLong.longValue();                  // the long value of
timestamp

            if (first < now ) parent.disconnectClient(firstLong);

            do_sleep(5000);              // sleep for 5 sec
        }
    }

    /** sleeps for given milliseconds.
    *@param millis the milliseconds to sleep.
    */
    private void do_sleep(long millis){
        try {
            sleep(millis);
        } catch (InterruptedException ie){}
    }
}

```

2.2.2 ProcessListener.java

```

/*
 * ProcessListener.java
 *
 */

package ee.ut.physic.rdc.clusterserv;

/**
 *

```

```

    * @author laas
    */

import java.util.*;
import java.io.*;

public class ProcessListener {

    /** The parent of this listener */
    ClusterServer parent;
    /** The name of the command that gets executed */
    String command;
    /** stdout listener */
    InputStreamListener stdout;
    /** stderr listener */
    InputStreamListener stderr;
    /** The stdin of the process */
    PrintWriter stdin;

    /** The type shows whether this is controller process or console
process */
    int type;

    /** The process */
    Process p;

    /** Creates a new instance of ProcessListener */
    public ProcessListener(ClusterServer parent, String command,int type)
{
        this.parent = parent;
        this.command=command;
        this.type=type;
        try{
            p = Runtime.getRuntime().exec(command);
            stdout = new InputStreamListener(this,p.getInputStream());
            stderr = new InputStreamListener(this,p.getErrorStream());
            stdin=new PrintWriter(p.getOutputStream());
            stdout.start();
            stderr.start();
        }catch(IOException ioe){parent.out.println("Error creating
process: "+command);}
    }

    /** This sends the process generated text upwards */
    public void sendText(String text){
        if (type ==1) parent.sendText(text);
        else if (type == 2) parent.consoleOutput(text);
    }

    /** Stop the process */
    public void stop() throws IOException{

```

```

        stdout.interrupt();
        stdout.stopListener();
        stderr.interrupt();
        stderr.stopListener();
        stdin.close();
        p.destroy();
    }

    /** Write data to process's stdin*/
    public void write(String text) throws IOException{
        stdin.println(text);
        stdin.flush();
    }

    /**
     *The class that monitors the inputStream and outputs data via
    parent's method sendText
    */
    class InputStreamListener extends Thread{

        /** The parent of this listener */
        public ProcessListener parent;
        /** The InputStream to be listened on */
        InputStream is;
        /** The reader used to read stream*/
        BufferedReader reader;
        /** The boolean, indicating that reading is to be done */
        public boolean listening = true;
        /** The object providing thread synchronization */
        Object threadLock = new Object();

        /** The constructor */
        public InputStreamListener(ProcessListener parent, InputStream
is){
            this.parent=parent;
            this.is=is;
            this.reader=new BufferedReader(new InputStreamReader(is));
        }

        /** This method does the works */
        public void run(){
            String line="";
            while (isListening()){
                try{
                    while ((line = reader.readLine()) != null) {
                        parent.sendText(line);
                    }
                } catch (NullPointerException npe) {break;}
            }
        }
    }

```

```
        catch (Exception e) {e.printStackTrace();}
    }
}

public boolean isListening(){
    synchronized (threadLock){
        return listening;
    }
}

public void stopListener(){
    synchronized (threadLock){
        listening=false;
    }
}
}
```