TARTU ÜLIKOOL
Füüsika-keemiateaduskond
Teoreetilise füüsika instituut

SANDER SÕNAJALG

TÖÖVOOGUDE KASUTAMINE ANDMETE VEEBITEENUSTEPÕHISEL HAJUSAL TÖÖTLEMISEL

Bakalaureusetöö

Juhendaja: dr. EERO VAINIKKO
Kaasjuhendaja: B.Sc. ILJA LIVENSON

Tartu 2007

SISUKORD

SISSEJUHATUS

Kinnitamist mitte-vajav on asjaolu, et 21. sajandi teaduse lahutamatuks osaks on muutunud interneti kiire arenguga tekkinud võimalused andmete hajutatud töötlemiseks. Hajustehnoloogiaid on ühelt poolt paljude aktuaalsete teadusuuringute seisukohast hädavajalikuks platvormiks ning töövahendiks, teisalt aga on nad ise väga kiiresti ja dünaamiliselt edasi arenemas. Viimase kümnendiga välja töötatud hajustehnoloogiatest enim kõneainet on ilmselt pakkunud erinevad Grid-tüüpi lahendused, kuid nendega paralleelselt või kohati ka tihedas seoses on välja arenenud ka teaduslike töövoogude ning töövoomootorite kontseptsioon. Ka töövoomootoreid endid on loodud erinevate eesmärkide ning spetsiifikaga: ühtede töövoomootorite ülesandeks on Grid-teenuste omavaheline mugavam integreerimine; teised üritavad lahendada sama probleemi veebiteenustena realiseeritud komponentide jaoks; kolmandal juhul realiseeritakse hajussuhtlus hoopis kolmandate tehnoloogiate abiga, kasutades näiteks Java platvormi RMI tehnoloogiat. Käesolevas töös keskendutakse eel-loetletutest kõige universaalsemale ja lihtsamale hajussuhtluse realiseerimise viisile ehk veebiteenustele, ning antakse ülevaade võimalustest, kuidas neid töövoomootorite abiga integreerida automaatseteks arvutusprotsessideks. Veebiteenuste levinud ja avatud standarditest tulenev universaalsus annab lootust, et juba hetkeks bioinformaatikute seas suurt populaarsust kogunud tehnoloogial on tulevikku ka füüsikas ning teistel teadusharudel.

Töö eesmärgiks on kaardistada veebiteenuste baasil koostatud töövoogude asukoht, roll ning osatähtsus hajustehnoloogiate üldisemal maastikul. Selleks selgitatakse esmalt hajustehnoloogiate olemust üldisemalt. Uuritakse, millised on tüüpilisemad motivatsioonid, mis panevad teadlasi oma arvutuste realiseerimisel otsima abi erinevatelt hajustehnoloogiatelt. Seejärel selgitatakse detailselt lahti veebiteenustel põhinevate töövoogude ning töövoomootorite olemus, tuues muuhulgas välja sellele lähenemisviisile spetsiifilised eelised ning vead. Üritatakse jõuda järeldustele, milliste rõhuasetustega probleemide lahendamisel on otstarbekas kasutada veebiteenuste ja töövoomootorite abi ning millal oleks vastupidi efektiivsem pöörduda alternatiivsete hajustehnoloogiaid poole (näiteks erid). Praktilises osas tuuakse illustreeriv näide sellest, kuidas taas-kasutatavate veebiteenuste uudsel viisil töövoogu ühendamise abil luuakse automaatne protsess, mis moodustab uue loogilise terviku (sisuliselt hajusa "programmi").

1. HAJUSTEHNOLOOGIAD

1.1 Hajustehnoloogiate üldine eesmärk

Interneti laialdane levik ning samuti viimasel aastakümnel saavutatud enneolematus suurusjärgus andmeedastuskiirused on andnud tehnilised võimalused mitmete probleemide uudsel viisil lahendamiseks paljudes erinevates eluvaldkondades. Dünaamika toimub pealtnäha kahes täiesti vastuolulises suunas. Esiteks ühelt poolt koonduvad üha enamad teenused kokku kesksetesse serveritesse ja portaalidesse (nt. e-kirjade lugemine teenusepakkuja internetileheküljelt, interneti-põhine kalender-märkmik või *messenger*i-klient). Kasutaja ees laual asetseva füüsilise arvuti roll liigub üha enam sekundaarse tähtsusega terminaliks olemise suunas, mille esmane eesmärk on võimaldada ligipääs nimetatud võrguteenustele. Vastupidi, teisest küljest on teadusemaailmas kuumaks sõnaks justnimelt **hajusarvutus**, tarkvara-arenduses analoogselt **teenuse-põhine arhitektuur** (*SOA* e. *service-oriented architecture*), mille mõlema kandvaks ideeks on ühe loogiliselt tervikliku protsessi füüsiliselt paljudesse erinevatesse kohtadesse laiali hajutamine, kasutades selleks interneti abi.

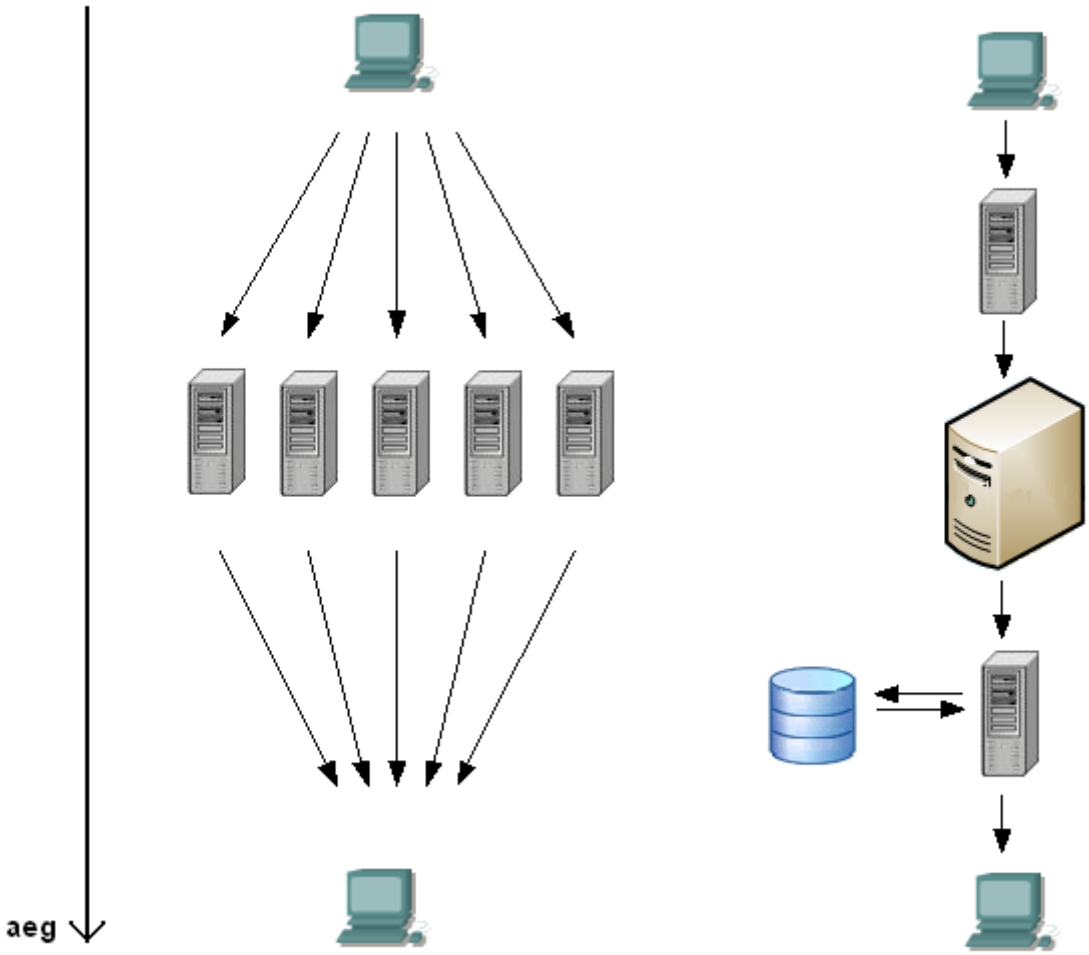
Mingit vastuolu siin tegelikult ei ole − osad tegevused lihtsalt on oma iseloomult sellised, et mida tsentraliseeritumalt suudetakse neid pakkuda, seda parem (koondamine kokku suurde serverisse); teised, vastupidi, nõuavad paljude eraldi-seisvate komponentide hästi-koordineeritud koos töötamist (hajutamine laiali erinevate hajustehnoloogiate abil). Vajadused, mis teatud tarkvaralahendust ühes või teises suunas kallutavad, on erinevad.

Tsentraliseeritusele motiveerivad näiteks vajadus ühtsele ja sünkroniseeritud informatsioonile paljude erinevate kasutajate jaoks. Samuti on selline süsteem ilmselt kiirem, töökindlam ning lihtsamini realiseeritav, kuna ei sõltu alamkomponentide suhtlemisest üle interneti. Tüüpine lahendus sellisel juhul on keskses serveris töötav infosüsteem, mis võib korraga tegeleda paljude kasutajatega, kes suhtlevad rakendusega reeglina oma interneti-*browser*'i abil (näiteks eBay, internetipangad jne.). Kõige puhtakujulisemateks tsentraliseeritud rakenduste näideteks on kõikvõimalikud jututoad või interneti-foorumid. Need on triviaalsete süsteemide näited, kus keskne infosüsteem ei pea informatsiooni vahetama ega muul moel integreeruma teiste, eraldiseisvate süsteemidega. Kõik muudatused selle süsteemi poolt kasutatavates andmetes toimuvad läbi selle süsteemi enda (nt. foorumisse teate lisamiseks ainuke võimalus on läbi selle foorumi veebiliidese).

Hajusate tehnoloogiate kasutamise järele võib tekkida vajadus erinevatel põhjustel:

1) **Koormuse jaotamine / ressursside jagamine** − Tihti esineb olukordi, mil üks riistvaraline masin ei suuda enam talle pandud ülesannetega üksi mõistliku ajaga toime tulla. Olgu tegu suure koormusega keskse veebiserveri, andmebaasiserveri või mahukat arvutust teha sooviva teadlase lauaarvutiga, kuskil asub alati ülemine piir, millest üle ükski kuitahes võimas masin ei suuda ressursside ja koormuse poolest enam rahuldada kogu tarbimisvajadust (suured andmetöötlus-, andmeedastusmahud või salvestamisvajadused). Kohati leiavad sellest probleemist ülesaamisel edukat rakendamist hajustehnoloogiad, mille abil koormust laiali jaotada ja vabu ressursse jagada.
2) **Iseseisvatest komponentide integreerimine** − Sisuliselt iseseisvad, erinevate juriidiliste isikute või institutsioonide halduses olevad infosüsteemid peavad tihti olema võimelised omavahel tihedalt suhtlema ja integreeruma. Samas ei ole võimalik koondada asutuste-vahelisi süsteeme kokku üheks süsteemiks, kuna see võib olla vastuolus osapoolte ärihuvide ja/või konfidentsiaalsuse-kaalutlustega või olla lihtsalt raskesti teostatav. Komponentide omavahel suhtlema-panemiseks on loodud hulganisti erinevaid hajus-tehnoloogiaid, millest töövoogude kontekstis on kõige olulisemaks **veebiteenused** (*web services*). Läbi rangelt-defineeritud liideste muudetakse oma partnerile üle interneti kasutatavaks täpselt see alamosa oma süsteemist, mis selleks ette nähtud on.

Kui esimene põhjus peaks olema küllaltki intuitiivselt mõistetav, siis teise põhjuse mitmetahulisust oleks ehk hea illustreerida näite varal. Eestis on arendatud Riigi Infosüsteemide Arenduskeskuse poolt raamistik X-tee, mis võimaldab veebiteenuste abil suhelda erinevatel riiklikel ning vastavaid volitusi omavatel era-infosüsteemidel üksteisega ning erinevate X-teega liitunud andmekogudega. Antud juhul muudab hajustehnoloogiate kasutamise möödapääsmatuks keerukas skeem, mille alusel loogiliselt iseseisvad süsteemid peavad suutma risti kasutada ja värskendada teineteiste andmeid, kusjuures peab olema rangelt tagatud andmete nähtavus ja muutmisvõimalused ainult selleks ettenähtud isikutele rangelt defineeritud ulatuses. Näiteks Rahvastikuregistrisse peavad saama muudatusi teha mitmete eri ministeeriumide paljude ametkondade infosüsteemid, kusjuures veel laiemal hulgal osapooltel on õigus teha lihtsaid andmepäringuid ilma andmeid muutmata, ning igal ajahetkel peab iga osapool saama kätte kindlasti kõige värskemad andmed. Selliseid nõudeid on võimalik täita ainult hajustehnoloogiate abil, ning konkreetselt sobivad selleks kõige paremini justnimelt veebiteenused.



Joonis 1: Horisontaalne ja vertikaalne hajutamine. Vasemal on kujutatud töö horisontaalset hajutamist koormuse jagamise eesmärgil, kus mitmed riistvaralised ressursid rakendatakse paralleelselt täitma sama ülesande erinevaid osi. Sellele vastandub vertikaalne hajutamine (kujutatud joonise paremas osas), mille eesmärgiks on kasutada ära serverite poolt pakutavaid unikaalseid teenuseid, pöördudes nende poole tööprotsessi erinevates etappides. Mõlemad protsessid algavad arvutuse algataja arvutist ning lõpevad tulemuste jõudmisega tagasi samasse arvutisse.

1.2 Hajustehnoloogiate rakendamine teadusarvutustes

Järgnevalt vaadeldakse, kuidas täpsemalt võiksid realiseeruda eelnevalt üldises plaanis kirjeldatud vajadused hajustehnoloogiate järele spetsiifiliselt teadusarvutuste kontekstis ning milliseid lahendusi on nende probleemide adresseerimiseks välja töötatud.

Esimeseks hajustehnoloogiate kasutamise üldiseks motivaatoriks oli ressursside piiratusest tulenev vajadus jaotada mingi terviklik protsess alamosadeks ning see füüsiliselt laiali hajutada, jagades suur koormus paljude masinate vahel ära. Kuna tänapäeva teaduses lahendatakse

arvutite abil palju ülikeerukaid modelleerimis- ja muid ülesandeid ning töödeldakse hiiglaslikke andmemahte, on koormuste jagamise vajadus ka kõige peamisemaks hajustehnoloogiate kasutamise põhjuseks teadusarvutustes. Otseselt sellel probleemi lahendamiseks on välja töötatud Grid, mis võimaldab kanda mahuka arvutuse teostamine teadlase tava-arvutilt üle spetsiaalsetele arvutusklastritele, mille võimsused on paari suurusjärgu võrra suuremad. Lihtsalt võimsamas keskkonnas käivitamise kõrval võimaldatakse ka arvutuse jagamine paralleelseteks harudeks ning nende töö omavaheline koordineerimine, kuid täiendava hajutamise põhjuseks on siin enamasti soov lahendada arvutusprobleem veelgi kiiremini, jagades koormus laiali paljude riistvaraliste ressursside vahel. Paralleelsetes harudes teostatavad toimingud on üldiselt ekvivalentsed.

Mõnevõrra väiksema tähelepanu osaliseks kui otsene arvutusressursi jagamine ja Grid, on saanud teadusarvutustes need vahendid, mis tegelevad hajustehnoloogiate teise tähtsa ülesande ehk hajusalt paiknevate unikaalset funktsionaalsust pakkuvate komponentide integreerimisega terviklikeks arvutusprotsessideks. Osapoolteks on siin reeglina teadusrühmade serveritesse kasutamiseks üles seatud taaskasutatavad veebiteenused, ning teadlane, kes soovib neid teenuseid ära kasutades midagi arvutada (koostades selleks reeglina töövoo ja käivitades selle töövoo mootoril). Kui kommertsiaalsetes infosüsteemides kasutati veebiteenustel põhinevat suhtlust pigem selleks, et kahte iseseisvat infosüsteemi omavahel bipolaarselt suhtlema ja infot vahetama panna, on teadusarvutustes kontroll reeglina hoopis selgemalt tsentraliseeritud. Keskseks osapooleks on klient, kes oma ühe konkreetse tööprotsessi alamosadena pöördub erinevate veebiteenuste poole, paludes neil teda "aidata" mingi osa tööst ära teha. Seda tööprotsessi, kui see on kuidagi automatiseeritud, võibki sisuliselt nimetada **töövooks**. Suurem tsentraliseeritus väljendub selles, et päringu esitab alati klient (masin, kus käivitati töövoog); veebiteenus aitab tal mingi etapi tööst ära teha ja saadab tagasi tulemused, kuid ei ole kunagi ise suhtlust algatavaks pooleks (ei pöördu mingite oma küsimustega kliendi poole). Siit ka nimetused "klient" ja "server", ehk vastavalt töövoogu käivitav osapool (kes on hajusate teenuste tarbijaks) ning võrguserver, kes seda teenust avalikult jagab (ing. k. *serve*). Nagu eelnevalt mainitud, leiabki antud töös käsitlemist nimelt viimane, veebiteenustel baseeruv hajusarvutuse suund.

2. TÖÖVOOGUDE OLEMUS

2.1 Töövood võrdluses traditsioonilise programmeerimisega

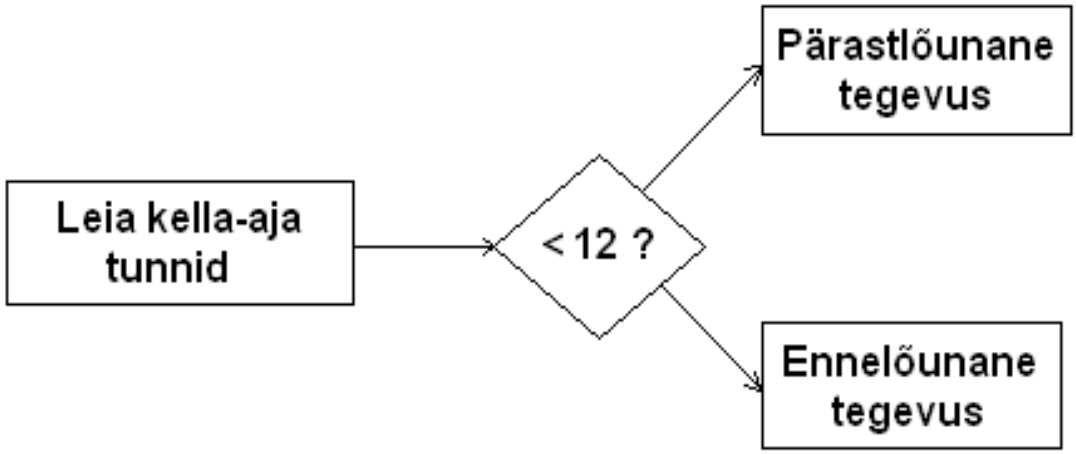
Kõige lihtsam on seletada töövoogude olemust läbi võrdluse traditsioonilise programmeerimisega. Sisulise poole pealt on kattuvus väga suur − kummalgi juhul on eesmärgiks ühese algoritmi või toimimiseeskirjade loomine, mille täitmisel jõuaks arvuti mingitest etteantud sisendandmetest lähtudes mingite soovitud tulemusteni. Kuigi selles küsimuses pole ühest konsensust, võib kõigi tunnuste ja omaduste põhjal ka töövoogu koos oma komponentidega siiski nimetada sõnaga "programm" (*application*).

Võrdluse kujukuse huvides võiksime tänapäeva traditsiooniliste programmeerimiskeelte (konkreetset keelt täpsustamata) programmikoodi tinglikult jagada kahte liiki lauseteks:

1. **keelelised kontroll-konstruktsioonid** − see osa süntaksist, mille abil luuakse algoritmi "skelett", millest lähtuvalt mingite lisatingimuste alusel otsustatakse, milliseid programmi alamosi tuleb järgnevalt täita ning millises järjekorras seda täpselt tehakse.
2. **funktsionaalsed kutsungid** − programmikoodi need laused, mis teevad ära reaalse töö: muutujatele väärtuste omistamised, kutsungid teistele komponentidele, muutujatele mingite funktsioonide rakendamised jne..

```
a = Time.now.hour
if (a < 12) do
  print "On ennelõuna."          # then-haru
else
  print "On pärastlõuna"         # else-haru
end
```

Joonis 2: Näide traditsioonilisest programmeerimisest. Antud näites on kontroll-konstruktsioonid esitatud kursiivis ning funktsionaalsed laused rasvases kirjas tekstiga. Muutujale `a` omistatakse väärtuseks hetke kella-aja tundide-osa. Järgneb kontroll-struktuur, mis vastavalt `a` väärtusele suunab programmi edasise täitmise kas harusse (i) või (ii). Kummagi haru funktsionaalne sisu on väljastada mingi teade. (Näide on toodud keeles *Ruby*).



Joonis 3: Töövoograaf. Eelnenud koodilõiguga ekvivalentne töövoog võiks visualiseerituna välja näha näiteks selline (kasutatud notatsioon on meelevaldne).

Antud skeemiga analoogsel viisil visualiseeritaksegi töövooge (erinevate töövoogude loomiseks mõeldud keskkondade puhul on töövoo visuaalsed esitused küll nüanssides erinevad, kuid antud käsitluse seisukohalt on selle põhimõtted siiski piisavalt sarnased). Edasises selgituses kasutatavale mõistele **töövoo komponent** vastavad joonisel ristkülikud. Noolte kasutamine väljendab komponentide käivitamise järjekorda; rombile vastab töövoo hargnemine kahte harusse sõltuvalt mingi tingimuse täidetusest (programmeerimisest tuntud vaste sellele oleks *if-then-else*-lause).

Töövoogude olemuse võtab kõige paremini kokku ülaltoodud skeem ise. Nagu näha, pannakse põhirõhk programmi stuktuurilise külje kirjeldamisele (edaspidi nimetatakse seda struktuurilist külge või "skeletti" lihtsalt **töövooks**). Kui eelnevalt jagasime programmikoodi tinglikult kontroll-struktuurideks ja funktsionaalseteks lausetekse, siis töövoogu võib sellise jaotuse korral suure täpsusega võrrelda esimese osa ehk kontroll-konstruktsioonidega. Teisele osale, ehk funktsionaalsetele kutsungitele, seaks selline käsitlus vastavusse **töövoo komponendid**. Sisuliselt tähendab see seda, et töövoo defineerimisega luuakse täidetava programmi üldine struktuur, näidates ära, millal ja millises järjekorras käivitub üks või teine komponent, kuid nende komponentide sisu kohta ei ütle töövoo visuaalne esitus mitte midagi − neid käsitletakse lihtsalt kui abstraktseid tööühikuid. Ometi on just töövoo komponendid see koht, kus töövoo täitmise juures tehakse ära kogu reaalne töö; töövoog ise kirjeldab vaid reegleid komponentide käivitamiseks ning nende omavahelise suhtlemise koordineerimiseks. Näiteks antud skeemil visandatud töövoog sisaldab endas kolme komponenti:

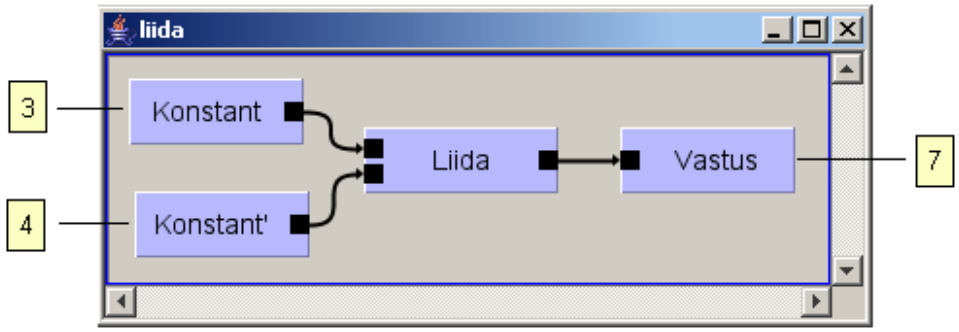
- komponent, mis tagastab hetke kellaaja
- komponent, mis teatab, et on ennelõuna
- komponent, mis teatab, et on pärastlõuna

Loomulikult ei saa ühegi reaalselt töötava programmi loomisel piirduda ainult struktuuri defineerimisega. Kui töövoo visuaalses esituses esinevad komponendid vaid kui abstraktsed tööühikud, siis peab töövoo mootor töövoo edukaks käivitamiseks siiski täpselt teadma, milline tegevus mingile komponendile vastab, kuidas seda käivitada, ning kuidas komponendilt tema töö lõppedes tulemused tagasi vastu saada. Üldpildist aru saamise huvides on töövoo komponentidest kõige hõlpsam mõelda kui eraldiseisvatest alamprogrammidest. Üldjuhul jäävad need väljapoole töövoogu ja seda käivitavat töövoomootorit ennast. Erinevatest konkreetsetest võimalustest komponentide loomiseks ja nendega suhtlemiseks (kuna tihti on tegu täiesti iseseisvate programmidega) tuleb pikemalt juttu peatükis 4.

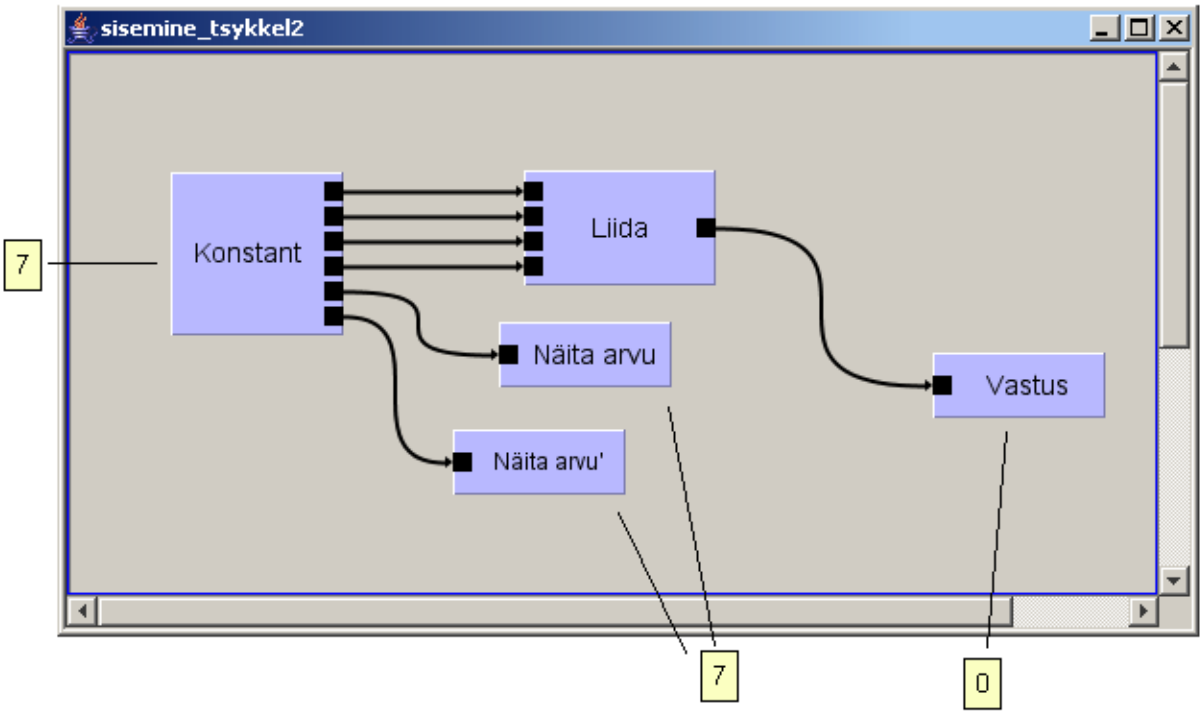
2.2 Töövoogude definitsioon kui tsükliteta suunatud graaf

Antud kontekstis võikski töövoogusid defineerida järgmiselt: **töövoog on suunatud ning tsükliteta graaf, mille tippudeks on töövoo komponendid ehk kasutatavad alamprogrammid, ning mille kaared väljendavad komponentide-vahelisi seoseid, defineerides komponentide käivitumise järjekorra ning komponentide-vahelised andmevood.** On oluline märkida, et tsükliteks ei loeta siinkohal olukorda, kus näiteks komponent A väljastab oma töö tulemusena 100 komplekti väljundandmeid, ning iga komplekti kohta käivitatakse andmeid sisendina kasutades komponent B (tavaprogrammeerimise seisukohalt oleks siinkohal tegu tsükliga). Sellise töövoo võiks aga ümber joonistada graafiks, kus komponendist A väljub 100 kaart, mis suubuvad 100-sse erinevasse koopiasse komponendist B, ning selline graaf oleks kindlasti tsükliteta (sõltuvalt töövoo-keskkonnast ei pea graafi tavaliselt siiski niimoodi välja joonistama; näiteks Taverna (pikemalt Tavernast peatükis 4; [3]) töövoogude keskkonna puhul hoolitseb taolise varjatud tsükli korrektse käivitamise eest töövoomootor ise). Küll aga on keelatud olukord, kus komponendi A väljundil käivitatakse komponent B, ning omakorda B väljundil uuesti komponent A (siin on tegu ilmse otsese tsükliga graafi läbimise mõttes).

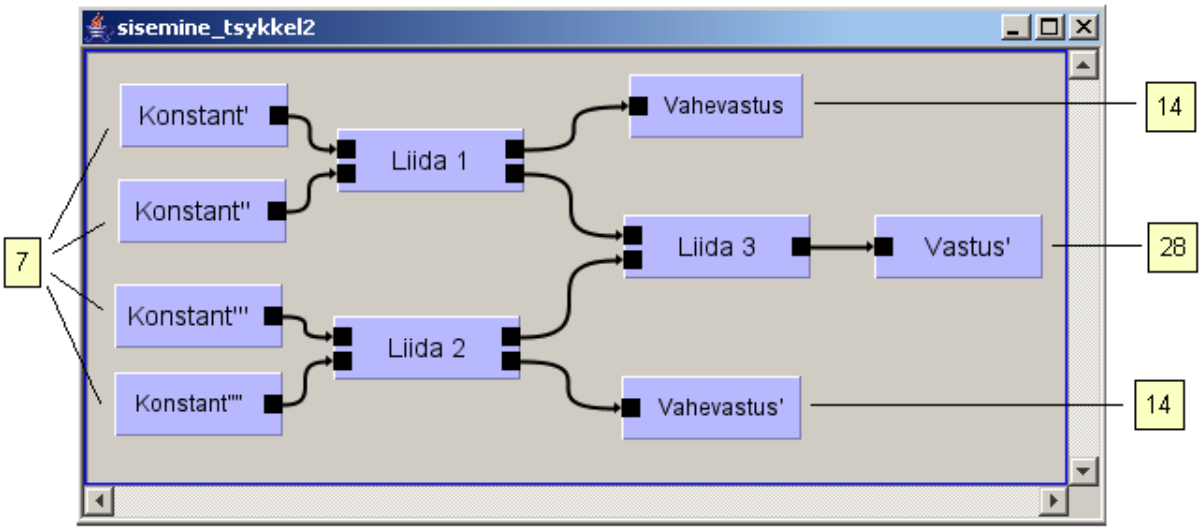
Järgnevad mõningad Triana töövoogude-keskkonnas (pikemalt käsitletud peatükis 4; [1]) koostatud sisult triviaalsed töövood, mis illustreerivad hästi töövoogude definitsiooni. Kõigis näidetes on kasutatud kolme erinevat tüüpi komponente: komponendid konstandi genereerimiseks (*"Konstant"*), komponendid kõigi sisendargumentide summeerimiseks (*"Liida"*), ning komponendid sisendargumendi näitamiseks (*"Vastus"*, *"Näita arvu"*).



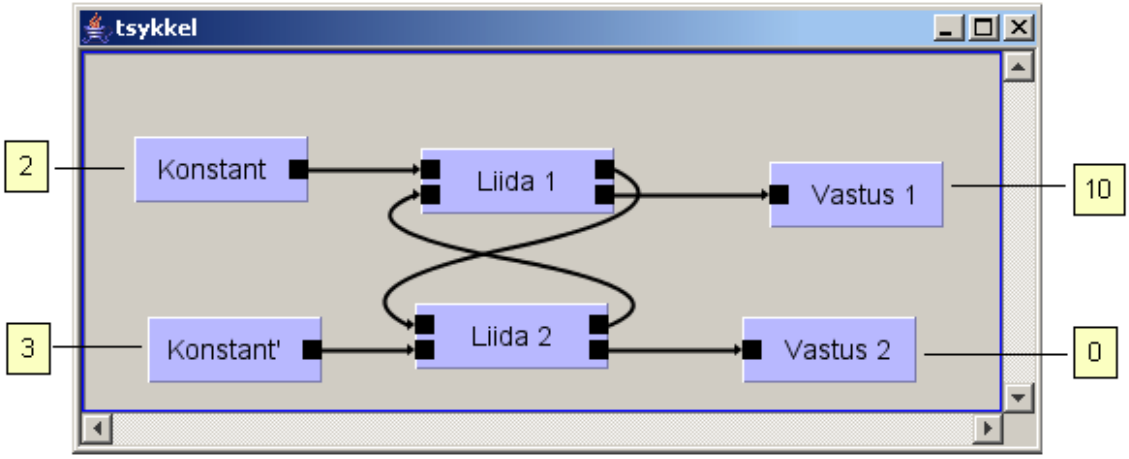
Joonis 4: Triviaalne tsükliteta graaf. Komponendi "Liida" sisenditeks on kaks konstanti, ning oma töö tulemuse väljastab ta iseseisvale neljandale komponendile.



Joonis 5: Varjatud sisemise tsükliga graaf. Tekib pisut segane olukord: otsest tsüklit graafis ei paista olevat, kuid komponent "Liida" peaks oma tööks vajalikud neli argumenti saama kõik samalt komponendilt "Konstant". Trianaga kaasa tulev kümnendmurdude genereerimise komponent oskab küll dubleerida oma töö tulemuse mitmesse eraldi väljundisse (kaks komponenti "Näita arvu" näitavad mõlemad, et said argumendiks arvu 7, ehk konstandi algse väärtuse), kuid komponent "Liida" ei tööta ootuspäraselt, ning lõppvastuseks saadakse summa 28 asemel arv 0. Triana lubab sellise töövoo küll koostada, kuid ei interpreteeri seda sisuliselt õigesti, ja seega tasub taolisi käitumiselt halvasti-defineeritud töövoogude erijuhte vältida.



Joonis 6: Joonise 5 graafiga sisult ekvivalentne varjatud tsüklita graaf. Sisemise tsükli vältimiseks on "Liida"-komponentide sisendid eraldatud. Taolist graafi interpreteerib Triana töövoomootor korrektselt, saadeks ootuspärase vastuse 28. Kõiki selliseid töövooge, mida saab mingil analoogsel viisil tsükliteta suunatud graafina esitada, saab ka ühel või teisel moel (sõltuvalt konkreetselt töövoogude loomise keskkonnast/mootorist) levinud töövoomootoritel realiseerida.



Joonis 7: Tsükliga graaf. On selge, et antud graafiga kujutatud juht ei mahu töövoo definitsiooni piiridesse, kuna graafis sisaldub otsene tsükkel graafiteooria mõttes, ning ei ole üheselt selge, millistes järjekordades komponente käivitada ning milliste sisendargumentidena. Triana küll loeb selle taaskord legaalseks töövooks ning jõuab käivitamisel ka mingite tulemusteni, kuid need on küllaltki meelevaldsed, ning mõnel teisel töövoomootoril võib taolise töövoo käitumine olla hoopis teistsugune või töövoog võib olla üldse mitte käivitatav. Siin kasutatud definitsioonist lähtuvalt ei võimaldata töövoogu esitada graafina, mis sisaldab otseseid tsükleid.

Taoline töövoogude tsükliteta suunatud graafidena esitamine on üld-aktsepteeritav ([1] , [2]). Projektid, kus kasutatakse töövoogude jaoks mõnda alternatiivset mudelit (näiteks *Fraunhofer Resource Grid*, mille raames üritatakse luua Petri võrkudel põhinev töövoogude-keskkond; [1]), on esialgu arendamis-järgus ning pigem eksperimentaalsed.

Traditsioonilistes programmeerimiskeeltes on konventsiooniks see, et kui keelelised konstruktsioonid ei defineeri teistsugust käitumist, siis toimub lausete täitmine programmi-tekstis suunaga ülevalt alla. Töövoogude puhul asendub see graafi läbimisega vastavalt kaarte suundadele kõigist võimalikest sisenditest kõikide võimalike väljunditeni ("sisendiks" nimetatakse graafiteoorias tippu, millesse sisenevate kaarte arv on 0; "väljundiks" vastupidi tippu, millest väljuvate kaarte arv on 0). See aitab selgitada ka tsüklite puudumise nõuet – kui graafis esineksid tsüklid, kaoks üks-ühesus graafi läbimisel, kuna jääks selgusetuks, kas ja kui siis mitu korda mingit tsüklit tuleb läbida. Piltlikult öeldes võiks töövoomootor töövoo täitmisel komponentide-vahelises tsüklis "keerutada" kuitahes palju kordi, tegemist oleks tavaprogrammeerimisest "lõputu tsükli" nime all tuntud nähtusega ning programm ei jõuakski kunagi konkreetse lõpptulemuseni.

2.3 Töövoo ortogonaalsus oma komponentide suhtes

Võrreldes veelkord töövooge tavaprogrammeerimisega, on suureks erinevuseks see, et programmitekstis on kõik funktsionaalsed tegevused siiski kirjeldatud selle sama programmikoodiga, ehk nad on programmi sisemine ja lahutamatu osa. Töövoogude puhul on vastupidi kõik funktsionaalsed tegevused viidud töövoost enesest väljapoole (iseseisvatesse komponentidesse), kirjeldades ära vaid viisi, kuidas ühe või teise komponendiga suhelda. Komponendid on töövoo seisukohalt sisuliselt nn. "mustad kastid" (*black box*) – nende käivitamisel antakse neile ette mingid sisendandmed ja saadakse vastu mingi väljund, seejuures teadmata mitte midagi sellest, milliseid operatsioone see komponent ette antud andmetega teeb või kuidas leitakse väljastatav tulemus.

Nii saavutatakse töövoo täielik ortogonaalsus oma komponentide sisu suhtes. Traditsioonilise programmeerimiskeele koodis on need kaks aspekti tahes-tahtmata omavahel segatud ning teineteisest lahutamatud. See tähendab, et suvalise funktsionaalset külge puudutava muudatuse sisse-viimiseks tuleb muuta programmi koodi, mis defineerib ühtlasi nii funktsionaalse kui struktuurilise aspekti. Töövoogude puhul on need aspektid teineteisest lahutatud nii, et üksikute komponentide käitumise ümber-defineerimist on võimalik teostada nii, et töövoogu endasse ei tule sisse viia ühtegi muudatust. Seda selgitab hästi järgnev näide.

Käesoleva peatüki alguses toodud Ruby-keeles kirjutatud programmilõik omistab esimesel real muutujale **a** väärtuseks süsteemi kella-aja tundide-osa väärtuse. Mitte usaldades oma arvuti kella õigsust ja soovides nüüd muuta programmi käitumist nii, et kellaaega küsitaks hoopis interneti vahendusel mõnest serverist, tuleb traditsioonilise programmeerimise juhul hakata muutma programmi koodi. Seevastu realiseerides sama funktsionaalsuse töövoo ja vastavate komponentide abil ning soovides sisse viia samasugust muudatust, jääb töövoog ise täielikult muutumatuks. Kuna kellaaja leidmise tegevus sisaldub vastavas iseseisvas komponendis, tuleb muudatuse realiseerimiseks muuta vaid selle eraldi-seisva komponendi käitumist − töövoo jaoks oli ja jääb see komponent "mustaks kastiks", mille sisust ei tea töövoog midagi ning on seega muudatusest isoleeritud.

Tekib õigustatud küsimus: millist kasu annab asjaolu, et töövoog ise jäi muutumatuks − komponendi koodi oli ju sellegipoolest vaja muuta ja tööd muudatuse sisse viimiseks oli tegelikult täpselt sama palju? Kui nii töövoogu ennast kui ka tema komponente haldab ja arendab üks ja seesama isik või töögrupp, siis ei annagi see mingit positiivset efekti; tavaliselt aga sellistel puhkudel töövoogude kasutamine võrreldes tavaprogrammeerimisega ennast ka ei õigusta. Pigem on töövoogude tüüpiliseks rakendusalaks olukorrad, kus ühe töövoo komponendid on hajutatud laiali suurte geograafiliste distantside taha, neid haldavad täiesti erinevad institutsioonid ning klientidele (töövoogude koostajatele) tehakse nad kättesaadavaks interneti abiga. Kuna muudatus puudutas vaid ühe komponendi sisulist poolt, siis kõik selle komponendi "kliendid", kes seda oma töövoo osana millekski kasutavad, jäävad sellest muudatusest puutumata (nende töövood töötavad edasi, ilma et nad isegi teaks, et ühe komponendi sees midagi muudeti).

3. TÖÖVOOGUDE IDEE REALISATSIOONID

Siiani on töövoogude mõistega ümber käidud kui üldistatud, abstraktse ideega. Kuivõrd tegelik elu seisneb siinkohal siiski reaalsete tarkvaralahenduste väljaarendamises ning eksisteerivate lahenduste kasutamises, tulebki nüüd minna konkreetseks ning vaadelda, millist tarkvara tänaseks töövoogude ideele baseerudes loodud on ning kes ja kuidas seda kasutab (saaks kasutada).

Esmalt tuleb mõista, et mingit ühtset standardit töövoogude koostamisel, salvestamisel ega käivitamisel ei eksisteeri. Kui näiteks interneti-lehekülje loomisel mistahes abivahendit kasutades on tulemuseks standardne *html*-keelne fail, mida saab omakorda vaadata mistahes interneti-*browseriga*, siis töövoogude näol on tegemist täiesti teise äärmusega. Praktiliselt iga töövoogude ideel põhinev lahendus realiseerib iseseisvalt ja teiste analoogsete tarkvaralahendustega mitte-ühilduvalt kõik vajalikud tarkvarakomponendid. Et võimaldada normaalne ja mugav töövoogudega töötamine, peaks iga rakendus endas sisaldama vähemalt järgmisi tarkvaralisi komponente:

- töövoo mootor - tarkvara olulisim osa ehk platvorm, mis oskab loodud töövoogu käivitada
- keel koostatud töövoo salvestamiseks (edaspidi **töövookeel**)
- võimalused töövoo koostamiseks otsese töövoo-keelse süntaksi kirjutamisest kõrgemal ja kasutajasõbralikumal tasandil
- võimalused töövoo visualiseerimiseks graafina
- võimalused uute komponentide lisamiseks töövoogude koosseisus kasutamiseks

Nagu peatselt lähemalt seletatakse, on taoline jaotus mõneti tinglik, ning komponentide piirid terviklikus rakenduses tihti hägusad.

3.1 Töövoo keeled ja mootorid

Töövoo visuaalne esitus suunatud graafina on küll inimese jaoks hea ja ülevaatlik, kuid ei sisalda endas töövoo käivitamiseks kogu vajalikku informatsiooni töövoo komponentide kohta ega ole ratsionaalne viis töövoogu digitaalselt salvestada. Töövoogude salvestamiseks on erinevate rakenduste raames välja töötatud erinevaid valdkonna-spetsiifilisi keeli (*domain-specific language*), mis on omavahel erinevad ja reeglina mitte-ühilduvad. Töövookeelte ühise joonena saab välja tuua,

et teadaolevalt baseeruvad nad kõik standardsel XML andmeformaadil, ning on seega küll inimloetavad, kuid tavaliselt liiga keerukad ja detailirohked, et töövoo koostajal oleks mõistlik otse töövoo-keelset süntaksi kirjutada.

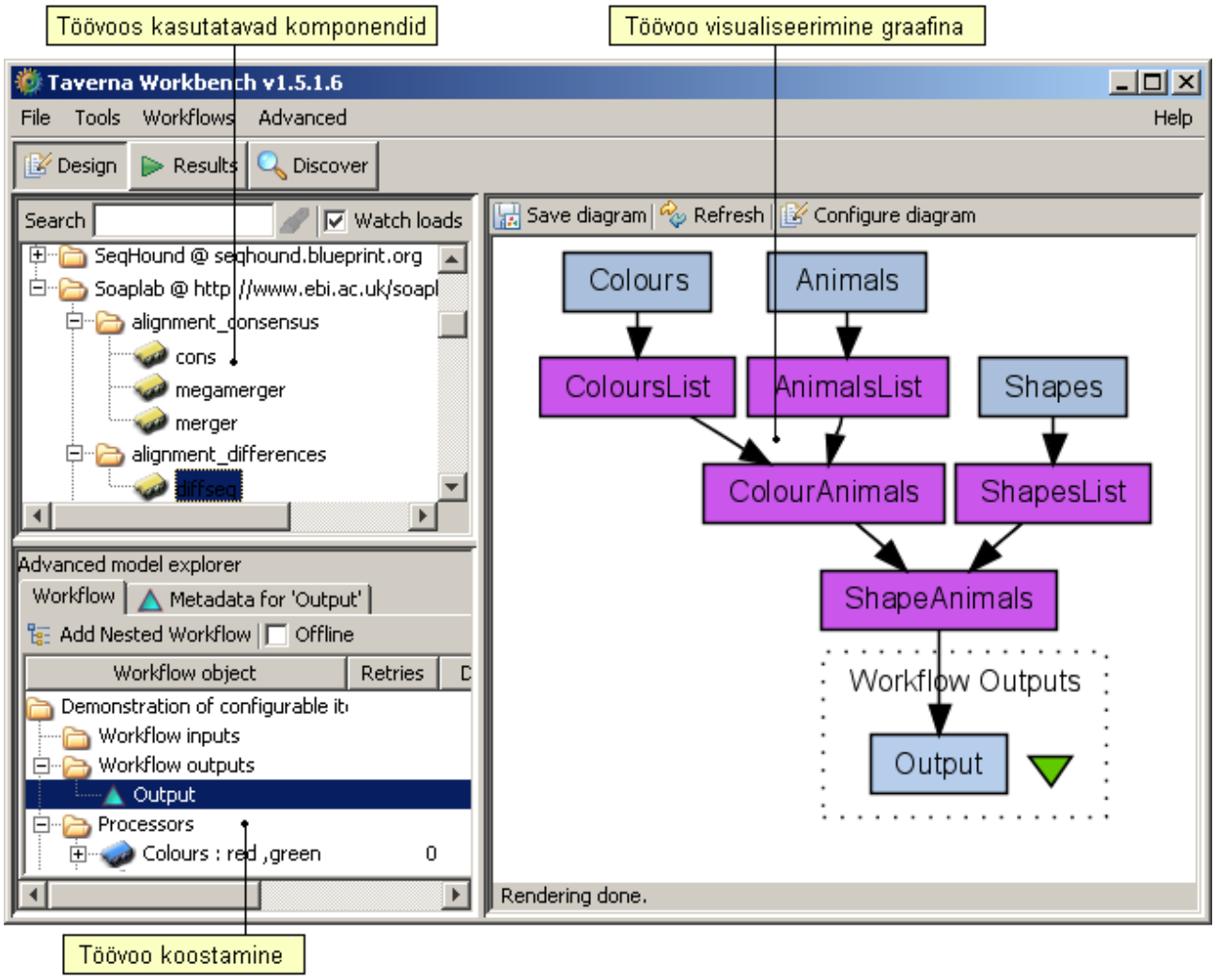
Olles juba leidnud mitmeid paralleele töövoogude ja traditsioonilise programmeerimise vahel, võib ka töövoogude idee erinevaid realisatsioone otseselt võrrelda erinevate programmeerimiskeelte paljususega, seades seejuures töövoo keelele vastavusse programmeerimiskeele ning töövoo mootorile programmeerimiskeele kompilaatori või interpretaatori.

Programmeerimiskeelne kood on kõige harilikum tekstifail, mis vastavalt konkreetse programmeerimiskeele süntaktilistele ning semantilistele reeglitele kirjeldab üks-üheselt mingit tegevust või protseduuri. Programmeerimiskeelne tekst on arvuti protsessorile otseseks täitmiseks täiel määral mõistmatu, kujutades endast vaid kokkuleppelist vahekihti, mis ühendab endas inimesele mugavaks kasutamiseks vajaliku kõrgetasemelise abstraktsuse, varjates liigsed detailid, ning arvutile täitmiseks antavuse eelduseks oleva üks-ühesuse, mis saavutatakse rangelt defineeritud süntaktiliste reeglite abil. Programmi käivitamisel "tõlgitakse" see tekst vastava programmeerimiskeele kompilaatorite või interpretaatorite abil juba protsessorile mõistetavaks masinkoodiks. Analoogne protsess toimib ka töövoogude puhul. Kompilaatori või interpretaatori rollis on siin **töövoo mootor**, mis oskab selles konkreetses töövoo-keeles kirjeldatud töövoogu käivitada ning loodetavasti saab hakkama kõigi töövoos kirjeldatud komponentidega suhtlemisega. Selliselt vaadelduna on töövoog sisuliselt üks arvutiprogrammi erijuht, ning töövoo mootor vastavalt platvorm, millel on võimalik see programm käivitada.

3.2 Integreeritud keskkonnad

Kuna töövoogude vallas ei esine ühilduvust erinevate tarkvaralahenduste vahel ning enamasti on kogu lahendus alates töövoo-keele väljatöötamisest kuni töövoomootori ja visualiseerimisvahendite loomiseni kirjutatud ühe ja sama osapoole poolt, ei ole olnud otsest põhjust eraldi tähelepanu pöörata ka tarkvara modulaarsusele. Sellistel lähtetingimustel muutub tarkvara lõpliku formaadi kujundamisel peamiseks kaalutluseks kasutajasõbralikkus, mis motiveerib koostama ühtset integreeritud töökeskkonda, kus kõik asjasse puutuvad tegevused on koondatud kokku ühtsesse rakendusse. Tarkvaraarenduses on taoliseid paljusid funktsioone ühendavaid keskkondi loodud juba ammu ning nimetatud üldnimega **integreeritud arenduskeskkond** (*integrated development environment* e. *IDE*); muudes eluvaldkondades (eriti näiteks inseneriteaduste suundadel) nimetatakse konkreetse kitsa eriala spetsiifiliste probleemide lahendamiseks

loodud keskkondi inglise keelse terminiga *problem solving environment* e. *PSE* (kanoniseeritud eestindus puudub). Ka töövoogudega töötamiseks loodud tarkvaralahendused on enamasti seda tüüpi, ning nende klassifitseerimiseks kasutatavana võib tihti kohata mõlemat ülaltoodud mõistet.



Joonis 8: Taverna tööpink. Näide töövoogude integreeritud arenduskeskkonnast. Tüüpiline töövoogudetarkvara sisaldab endas kõiki ülal eristatud tarkvarakomponente, mis on omavahel tihedalt integreeritud monoliitseks mugavaks "tööpingiks" või töövoogude arenduskeskkonnaks.

3.3 Töövoogude koostamine töövoogude arenduskeskkonnas

Olles defineerinud töövoo kui suunatud tsükliteta graafi, on loomulik, et töövoo koostamise protsess seisnebki töövoo komponentide ja nende-vaheliste seoste põhjal graafi koostamises. Mõnede töövookeskkondade (näiteks Triana) puhul kehtib see võrdlus eriti üks-üheselt, kuivõrd töövoo

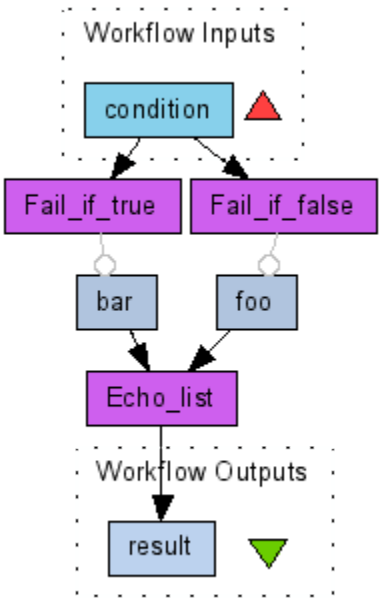
koostamiseks tulebki töövoo komponente sümboliseerivaid kastikesi mööda ekraani hiirega ringi lohistada ning omavahel noolekestega ühendada. (See on hea näide töökeskkonna tihedast integreeritusest: töövoogude koostamise ning visualiseerimise funktsioonid on omavahel täielikult ühendatud; seevastu Taverna töövookeskkonnas on nad teineteisest sõltumatud (seda võib näha ka jooniselt 8)).

Töövoo alamkomponentidega ühendumise tehniliste detailide eest hoolitseb töövoo mootor, säästes sellest töövoo koostajat. Kõige märgatavama efekti annab see just hajusate komponentide puhul, kuna võrgusuhtluse korraldamine programmeerimiskeele tasandil on paratamult keerukas ning aega- ja oskusi nõudev. Ühesõnaga tegutsetakse soovitud funktsionaalsusega "programmi" koostamisel taolise töövoo-graafina tavaprogrammeerimise suhtes tunduvalt kõrgemal (abstraktsemal) tasandil (näiteks hiljem pikemalt jutuks tuleva Triana töövookeskkonna reklaamlauseks on *"Create programs without programming."* – "Koosta programme ilma programmeerimata."). Need olulised lihtsustused aitavad hajustehnoloogiad muuta kättesaadavaks ka vähesema informaatika-alase haridusega kasutajaskonnale, näiteks füüsika ja geenitehnoloogia vallas tegutsevatele teadlastele.

Nagu eelnevalt mainitud, esinevad komponendid töövoo koosseisus "mustade kastide" abstraktsioonina: töövoo koostaja teab (kolmandate osapoolte loodud komponentide puhul näiteks dokumentatsioonist lähtuvalt), milliseid sisendandmeid ning millisel kujul on vajalik komponendile tema tööks ette anda, ning millises formaadis väljundi komponent oma töö järel tagastab. Komponendi sisemine toimeloogika ei ole seejuures tähtsust-omav ning tihti ka kasutajale üldse mitte teada. Kahe komponendi omavahel ühendamise eelduseks on see, et esimesena käivitatava komponendi väljundi formaat ühtib talle järgneva komponendi poolt eeldatava sisendi formaadiga, või et töövoo mootor oskab formaate ise konverteerida. Kui töövoo koostaja tahab näiteks kasutada teineteise järel kahte kolmandate osapoolte koostatud hajuskomponenti (s.t. ta ei saa ise muuta ei ühe komponendi väljund-formaati ega teise komponendi sisend-formaati), kuid andmeformaadid ilma teatud vaheteisendusteta ei ühildu, on loomulikuks lahenduseks kahe komponendi vahele ise luua kolmas komponent, mille ülesandeks on teostada vajalik formaaditeisendus. Paremal juhul pakub sellise komponendi koostamiseks võimaluse töövoo-keskkond ise.

Kõigis tänapäeva programmeerimiskeeltes on programmeerija käsutuses suhteliselt standardne hulk keelelisi konstruktsioone, mis võimaldavad dünaamiliselt reageerida erinevatele olukordadele programmi täitmisel ning vastavalt sellele kujundada edasist käitumist (kõige ilmsem näide nendest konstruktsioonidest oleks klassikaline *"if-then-else"*-lause). Ekvivalentset käitumist saavutada võimaldavad üldiselt ka töövookeeled, aga realiseeritud võivad need võimalused olla

üsna erinevalt. Näiteks realiseerib Taverna töövookeskkond tsükleid varjatud kujul: kui üks komponent tagastab oma väljundina ühe andme-elemendi asemel terve elementide seeria, itereeritakse vaikimisi üle kogu selle seeria, käivitades järgnevat komponenti üks kord iga seerias sisalduva elemendi kohta. Töövoo koostajal on võimalik itereerimise strateegiaid ka omalt poolt ümber seadistada, kasutades näiteks ühe variatsioonina hulkade korrutamise põhimõtet. Viimane on võimalik olukorras, kus komponent C eeldab oma sisendina kahte andme-elementi, millest ühe väljastab eelnev komponent A ning teise komponent B, ning A ja B väljastavad ühe elemendi asemel vastavalt *n*-elemendilise ning *m*-elemendilise andmete seeria. Sellisel juhul käivitatakse komponent C eraldi iga võimaliku sisendandmete kombinatsiooni jaoks, valides iga kord kummastki seeriast täpselt ühe elemendi, ehk kokku *m*n* korda. Ka tingimuslaused (nn. *if-then-else*) realiseeritakse tavaprogrammeerimisest pisut erinevalt. Tuues näite taaskord Tavernast, tuleb seal töövoog tingimuse rakendada soovimise punktis jagada kaheks paralleelseks haruks, millest üldjuhul peaks täitmisele kuuluma mõlemad. Et tingimus rakenduks, tuleb nüüd kumbagi haru alustada spetsiaalse komponendiga, mis kontrollib nn. *then*-harus soovitud tingimuse tõesust ja *else*-harus sama tingimuse väärust. Komponendi ülesandeks kummalgi juhul on kontrollimise ebaõnnestumise puhul vastava haru edasine rakendumine blokeerida.



Joonis 9: Tingimuslauset sisaldav töövoog Tavernas. Parem haru komponendiga *foo* käivitatakse juhul, kui tingimus osutus tõeseks; kui tingimus on vale, täidetakse vastupidi vasak haru komponendiga *bar*. Tingimusi kontrollivate komponentide *Fail_if_true* ja *Fail_if_false* rolliks on töövoo täitmise blokeerimine harus, mis vastavalt tingimusele ei kuulu täitmisele (seega mistahes juhul täidetakse kahest võimalikust harust täpselt üks).

4. TÖÖVOO KOMPONENDID. VEEBITEENUSEID INTEGREERIV TÖÖVOOG

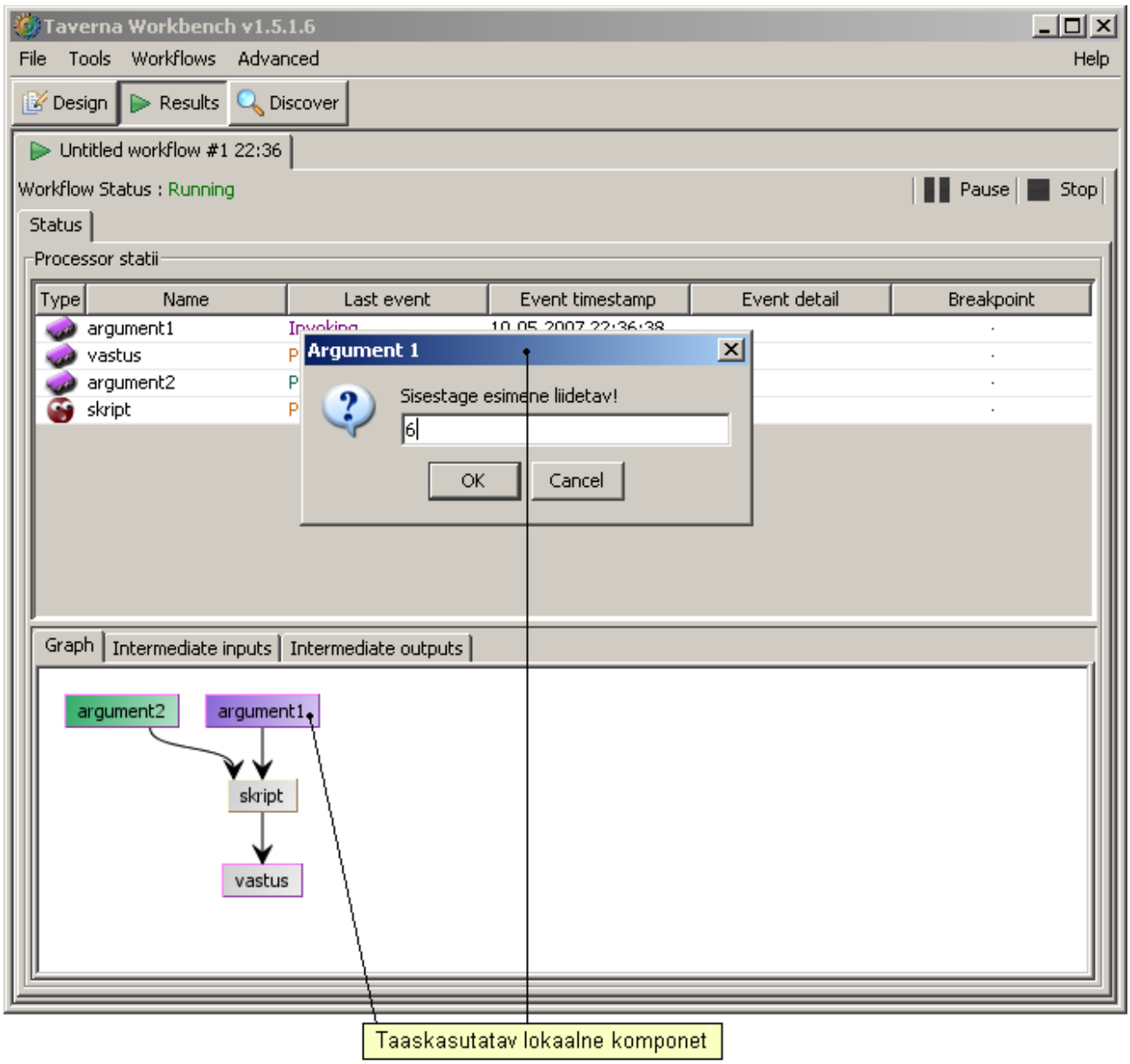
4.1 Komponentide klassifikatsioon

Oluline on mõista, et teoreetilisel tasandil ei sea töövoogude idee mingeid otseseid piiranguid sellele, millist tüüpi on tema komponendid ja kuidas töövoo mootor nendega sidet peab – oluline ongi vaid see, et komponendid oleksid mingil moel töövoomootorile kättesaadavaks tehtud. Et töövoo mootori kasutamine ennast praktikas õigustaks, peaks uue komponendi lisamine ja kirjeldamine olema kasutaja jaoks piisavalt lihtsaks tehtud. (Suhteliselt selge on, et töövoo-keskkonna abiga saavutatav kasutusmugavus ja lihtsus komponentidest töövoo koostamisel muutuksid tarbetuks, kui triviaalseimagi uue komponendi mootoriga ühendamine nõuaks kasutajalt päevade-pikkust pingelist programmeerimist). Reaalsuses tähendab see aga seda, et töövoomootori ja komponentide vaheline suhtlemine saab toimuda piiratud arvu hästi-defineeritud ja laialt levinud protokollide vahendusel. Kõige loomulikumaks näiteks oleksidki siinkohal veebiteenuste standard-protokollid *SOAP* ning *XML-RPC*. Iga uue suhtlusprotokolli võimaldamine ja toetamine nõuab töövoomootori valmistajatelt tõsist tööd, ning kuna tegelikkuses sobilikke standardseid suhtlusprotokolle ka väga palju ei leidu, on reaalsed töövookeskkonnad ette nähtud sobituma küllaltki piiratud arvu eri tüüpi komponentidega.

Töövooge võib põhimõtteliselt kasutada ka nii, et kõik töövoo poolt kasutatavad komponendid on **lokaalsed**, s.t. asuvad samas arvutis, millel käivitatakse töövoog ise. Taolise tegevuse praktiline kasu võiks olla see, kui tegemist on kolmandate osapoolte poolt spetsiaalselt sellele töövoomootorile kirjutatud taaskasutatavate komponentidega, mis pakuvad vajalikku funktsionaalsust ning integreeruvad töövoomootoriga ilma mingi lisavaevata. Sellisel juhul osutub töövoomootor kasulikuks, varjates töövoo koostaja eest suure hulga tarbetut keerukust, millega tuleks tegeleda siis, kui hakata ise kirjutama programmi, mis seda komponenti kasutaks (paljudel juhtudel osutuks see tõenäoliselt keerulisemaks, kui sama komponendi ise uuesti kirjutamine). Juuresoleval joonisel on demonstreeritud dialoogiakende kasutamist taaskasutatavate lokaalsete komponentidena Taverna töövookeskkonnas. Määrata tuleb vaid dialoogiakna pealkiri ja aknas näidatav tekst (tegelikult on needki pigem vabatahtlikud), ning antud näites suunata akna abil saadud väärtus vajalikku järgmisesse komponenti. Suvalises programmeerimiskeeles nõuaks see siiski teatavat kasutajaliidese-teekide tundmist. Antud töövoos on kasutatud ka teist tüüpi kohalikku

komponenti, mis kujutab endast töövoo looja poolt kirjutatud skripti (ehk lühikest programmijuppi), mis töövoo koosseisus temasse suunatud sisendandmetega töövoo koostaja poolt kirjeldatud viisil manipuleerides leiab teatud väljundi, ning edastab selle järgmisesse komponenti. Taolised programmeeritavad kohalikud komponendid on väga vajalikud eelpool-käsitletud olukorras, kus kahe järjestikkuse hajusa komponendi andmeformaadid teineteisega üks-üheselt ei sobi. Nende omavahel integreerimiseks lülitatakse sellisel juhul kahe hajusa komponendi vahele töövoogu skript, mis teostab vajaliku andmeteisenduse (või mis-iganes muu loogilise vahe-operatsiooni).

Kuigi lokaalsed komponendid on töövoogude lahutamatu osa, on nende roll töövoogude kasutamisel siiski pigem sekundaarne ja toetav. Töövoogude ning töövoomootorite peamised eelised hakkavad ilmnema siis, kui neid kasutada erinevate **hajusate komponentide** integreerimiseks ühtseks automaatseks protsessiks. Nagu hiljem täpsemalt selgitatakse, ei ole siinkohal komponentide hajusalt paiknemise põhjuseks reeglina mitte koormuse jagamise ja jõudluse suurendamise vajadus (töö jagamine funktsioonilt samastesse harudesse, mida jooksutavad paralleelselt erinevad füüsilised arvutid). Pigem on komponendid funktsionaalsuselt unikaalsed ja iseseisvad, ning nende hajutatuse põhjustab kuulumine erinevate institutsioonide haldamise alla ja/või seotus erinevate paratamatult hajali paiknevate andmekogudega. Taolised taaskasutatavad komponendid, mille väljatöötaja ja kasutaja on tihti erinevad isikud, tekitavad aga koheselt küsimuse, kuidas peaks toimuma töövoo ja hajusa komponendi omavaheline suhtlus, ning panevad otsima ühiste üldtuntud standardite järele.



Joonis 10: Taaskasutatav kohalik komponent. Näiteks Taverna töövookeskkonnaga tulevad kaasa mugavad valmis komponendid dialoogiakendena töövoo jooksmisel kasutajaga interaktiivselt suhtlemiseks: teadete ja hoiatuste näitamine, kasutajalt täiendavate andmete küsimine poole töövoo täitmise pealt jne.. Konkreetselt on käivitatud töövoog, mis enne *BeanShell* skripti käivitamist küsib kasutajalt dialoogiakende abil skripi jaoks sisendargumente.

4.2 Veebiteenuste standardid

Teaduslike töövoogude ning töövoomootorite tekkimise ajaks oldi ärisektoris hajusate komponentide vahelise suhtlemise standardiseerimise probleemiga juba pikalt tegeleda jõutud, ning sõelale olid jäänud mõningad üldtuntud protokollid ja standardid. Teaduslike töövoogude olemusega sobis hästi *SOAP* ja *XML-RPC* protokollidel baseeruvate veebiteenuste tehnoloogia, mis muuhulgas ei kohustanud hajusa suhtluse kaht osapoolt vähimalgi määral kasutama sama

programmeerimiskeelt või platvormi. (Tegelikult tuleb veebiteenuste ja töövoomootorite arengu vahel kindlasti näha ka vastupidist seost: veebiteenuste näol oli tekkinud hea baastehnoloogia hajusarvutuste uudse suuna arenguks, kuid ilma töövoomootorite abita oli veebiteenuste põhjal terviklike arvutusprotsesside koostamine liiga aeganõudev ning keerukas).

Töövoo hajusate komponentidega üle interneti suhtlemise eest hoolitseb töövoomootor (töövoomootoritest tuleb lähemalt juttu järgmises peatükis). Antud töös vaatluse all olevad veebiteenuseid integreerivad töövoomootorid kasutavad selleks ülalnimetatud standardseid protokolle, samas varjates töövoo koostaja eest kõik hajussuhtlust puudutavad detailid. Veebiteenuste standardi hulka kuulub ka veebiteenuste kirjeldamise keel **WSDL** (w*eb service definition language*), mis sisaldab endas kogu vajalikku meta-informatsiooni, et teenuse klient (antud juhul töövoomootor) oskaks kirjeldatava teenusega suhelda sellest eelnevalt mitte midagi teades (tarvilik on kirjeldada, millisel hulgal ning millist tüüpi argumente tuleb kliendi poolt teenusele ette anda, milline on teenuse poolt antava vastuse formaat, ning muid suhtlus-spetsiifilisi detaile).

Teaduslikest töövoogudest ning töövoomootoritest rääkides on standardsed *wsdl*-failid ning kogu veebiteenuste tehnoloogia üldtuntus äärmiselt suure tähtsusega. Nimelt kuna veebiteenused on väga laialt levinud standard, on paljudes programmeerimiskeeltes hea tugi mingi valmis-kirjutatud algoritmi või programmi veebiteenusena publitseerimiseks, mida **hajusa teenuse pakkuja** saab ära kasutada oma programmi hõlpsasti üle interneti kättesaadavaks muutmiseks. Programmi tööks vajaliku sisendi ning programmi väljundi kirjeldamiseks genereeritakse ideaaljuhul (sõltuvalt konkreetsest programmeerimiskeelest/abivahenditest) ka vajalik *wsdl*-fail, ilma et programmi koostaja selleks mingit erilist lisatööd peaks tegema, ning see fail muudetakse rakendusserveril läbi interneti kättesaadavaks. Taoliselt luuaksegi töövoomootoritega integreeruvaid uusi hajusaid komponente.

Mugavatest vahenditest uute veebiteenustel baseeruvate komponentide loomiseks tuleks kindlasti esile tõsta tarkvaralahendust SoapLab, mis võimaldab olemasolevaid käsurealt töötavaid tööriistu mähkida veebiteenuste liideskihti [6]. See lähenemine ei sõltu mähitava programmi realiseerimisel kasutatud programmeerimiskeelest, samuti ei ole vaja hakata täiendama programmi enda koodi, mis muudab SoapLabi kasulikuks abivahendiks vanast ajast jäänud arhailise-võitu ja ebamugavate käsureatööriistade uuel viisil taaskasutamiseks näiteks töövoogude koosseisus. Programmeerimiskeele-spetsiifilistest tehnoloogiatest väärib märkimist moodsa skriptimiskeele Ruby baasil töötav veebiraamistik Ruby on Rails, mille alamraamistik ActionService võimaldab loetud koodiridadega luua veebiteenuseid praktiliselt suvalisest Ruby-keelsest programmist [5, lk 411] (tehnoloogia leiab kasutamist ka käesoleva töö praktilises osas; vastav programmikood

sisaldub töö lisas 4). Java platvormil pakub veebiteenuste loomisel head tuge Apache Axis raamistik [7], Perli programmeerimiskeele baasil on selleks moodul SOAP::Lite [8].

Teisest küljest oskavad töövoomootorid neile ette antud *wsdl*-failide põhjal vastavate hajusate komponentidega suhelda, seega **hajusa teenuse klient** (töövoo koostaja ehk näiteks teadlane, kes tahab mitmeid hajusaid komponente omavahel ühendatuna kasutades midagi arvutada) ei pea tänu töövoomootorile ja standard-protokollide kasutamisele ise tähelepanu pöörama keeruka võrgusuhtluse protseduuri organiseerimisele, saades keskenduda oma arvutuse sisulisele küljele. Oma töövoo koostamise keskkonda uue komponendi lisamiseks piisab reeglina (sõltuvalt töövoomootorist) vaid sisestada teenust kirjeldava *wsdl*-faili võrguaadress.

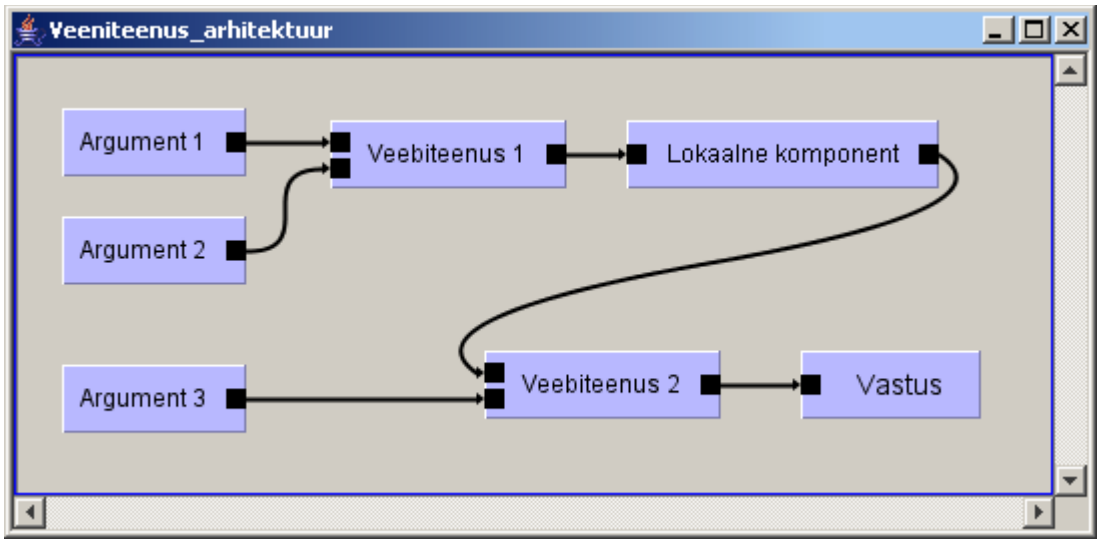
4.3 Veebiteenuseid integreeriva töövoo arhitektuur

Andmaks selgemat ettekujutust veebiteenuste baasil koostatud töövoo toimimisest, tuuakse näitena alljärgneval joonisel kujutatud töövoog, üritades muuseas ka veidi seletada veebiteenuste kasutamise võimalikke eesmärke.

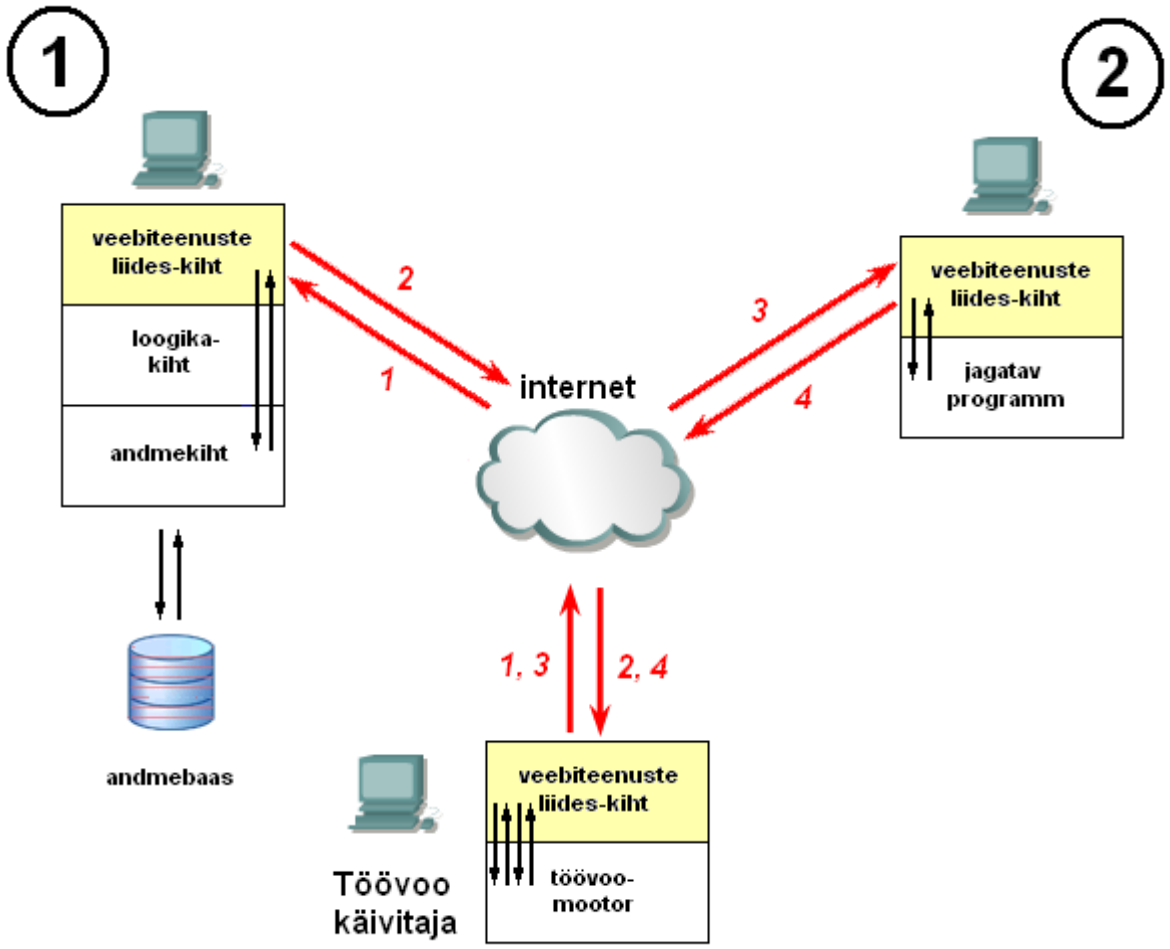
Töövoog koosneb kahest teenusest: esimese teenuse sisuks on teatud andmebaasipäring kahe etteantud argumendi järgi. Teine hajuskomponent kujutab endast veebiteenusena publitseeritud tööriista andmete analüüsiks, kusjuures lisaks andmebaasist saadud päringutulemustele on töövoo koostajal vajalik tööriistale ette anda veel täiendav argument. Samuti ei sobi andmeteenuse vastuse formaat kasutamiseks otse tööriista sisendandmetena, vaid seda tuleb eelnevalt pisut töödelda, milleks on töövoo koostaja kirjutanud lühikese programmi-jupi ja realiseerinud selle kohaliku komponendina.

Antud töövoo käivitamisel analüüsib töövoomootor esmalt sõltuvusi suunatud graafi komponentide vahel. Leides, et teiste komponentide töö sõltub komponendi "Veebiteenus 1" töö tulemustest, pöördutakse esmalt üle võrgu selle teenuse poole. Töövoomootor teeb siinkohal suurema osa tööst kasutaja eest automaatselt ära, koostades teenust kirjeldava wsdl-faili alusel teenusele arusaadava *SOAP*-formaadis sõnumi, asetades vajalikud argumendid sõnumis õigetele kohtadele ning saates see sõnum üle interneti veebiteenuse suunas. Kuna tegemist on andmeteenuse tüüpi teenusega, tehakse serverist, kus asub veebiteenus, omakorda päring andmebaasiserverisse, kuid see protsess jääb töövoo käivitaja eest täielikult nähtamatuks − kogu teenuse sisemine toimeloogika varjatakse veebiteenuste liides-kihi taha. Server koostab vastuse ning saadab selle töövoo käivitaja arvutis töötavale töövoomootorile tagasi. Töövoomootor loeb saadud *SOAP*-vastusest välja olulised andmed, teisendab need kohaliku komponendi abiga ning pöördub

analoogsel viisil teise veebiteenuse poole, laskmaks saadud andmeid teises serveris asuva kasuliku tööriista abil analüüsida. Kõik kordub analoogselt, kusjuures veebiteenuste liideskihi taga paiknevat tööriista ennast töövoo käivitaja taaskord ei näe. See asjaolu muudab töövoogude tehnoloogia suurepäraseks vahendiks **andmete või programmide piiratud ja kontrollitud jagamiseks** kasutamisest huvitatud osapooltega, mida käsitletakse pikemalt töövoogude otstarbekatele rakendustele pühendatud peatükis. Kogu kirjeldatud protsessi on üle interneti toimuva andmevahetuse perspektiivist kujutatud ka järgneval joonisel.



Joonis 11: Veebiteenuseid integreeriv töövoog.



Joonis 12: Töövoo käivitamine interneti andmevahetuse perspektiivist. Numereeritud nooltega on kujutatud *SOAP*-päringute/-vastuste liikumine üle interneti, kusjuures numbrid markeerivad kronoloogilist järjekorda.. Kuna esimese teenuse iseloomuks on andmeteenus, pöördutakse sealsest serverist omakorda andmebaasiserverisse, kuid töövoo käivitaja seda ei näe.

4.4 Veebiteenustel ja Grid-teenustel põhineva hajusarvutuse võrdlus

Kui antud töös keskendutakse töövookeskkondadele, mis on ette nähtud integreerima veebiteenustena publitseeritud komponente (sellest kohe ka pikemalt), siis tuleb ära mainida ka teine väga levinud töövoogude rakendus, milleks on Grid-teenustestest automaatsete protsesside koostamine. Hajustehnoloogiast üldisemalt kõnelenud peatükis toodi välja kaks peamist motivatsiooni hajustehnoloogiate kasutamiseks: esiteks mahuka andmetöötluse puhul koormuse horisontaalne jagamine, võimaldamaks sama ülesande lahendamiseks kasutada rohkem füüsilist ressurssi, ning teiseks töö vertikaalne etappideks jaotamine vastavalt unikaalset funktsionaalsust pakkuvatele teenustele, mis paiknevad hajutatult kasutaja tahtest sõltumatuna. Kui veebiteenused kalduvad sobivat pigem teise probleemi lahendamiseks, siis Grid-teenuste abil teostatakse pigem just ülimahukaid arvutusi ning töödeldaks hiiglaslikke andmemahte. Mahukamate ja keerukamate

protsesside efektiivne läbiviimine nõuab märksa keerukamaid standardeid, milleks on teenuse-põhise Gridi jaoks kujunenud *Global Grid Forum*i poolt välja töötatud *Open Grid Services Achitecture* (*OGSA*) spetsifikatsioon [4]. Selle raames kasutatakse küll samuti veebiteenuste tehnoloogiat, kuid teenused võivad oma funktsioonidelt olla märksa keerukamad. Kui käesolevas töös vaatluse all olevas veebiteenuste-põhises andmete hajustöötluses on iga teenus üldiselt iseseisev, oodates koos iga väljakutsega teatud algandmeid ning olles valmis nende põhjal vastuseks väljastama leitud tulemuse, siis Grid-teenuste hulka kuuluvad ka mitut liiki tugiteenused, mis tegelevad näiteks ressursside haldamise, klientide autoriseerimise ning informatsiooni väljastamisega teiste teenuste oleku kohta.

Tuues siinkohal ühe lihtsa võrdluse, siis veebiteenuste-põhisele lähenemisele tüüpiline viis teenuselt arvutustulemuste tagasi saamiseks oleks teenuse otsene vastus esitatud *SOAP*-päringule, kasutades selleks *HTTP* protokolli (otseselt võrreldav interneti-*browser*iga mingile internetileheküljele navigeerimisega: saadetakse serverile päring sooviga see lehekülg saada ning oodatakse vastust, kusjuures selle saamise aeg on eeldatavasti suurusjärgus sekundikümnendikud kuni sekundid). Grid-teenuste puhul oleks selleks tüüpiliselt eraldi teenus, mille poole saaks pöörduda palvega kopeerida arvutustulemused kuhugi kliendile kättesaadavasse serverisse, kasutades selleks näiteks GridFTP-d. Suurte andmemahtude ja pikkade arvutusaegade (mahukad Grid-arvutused võivad kesta mitmeid päevi) puhul on esimene variant loomulikult välistatud.

Keerukamad standardid, suuremad mahud ja ajaliselt pikemad protsessid esitavad loomulikult suuremaid nõudmisi ka töövoomootoritele. Sisseehitatud toetusega Grid-teenustel baseeruvateks töövookeskkondadeks on näiteks GridNexus ([2]) ja Java CoG Kit koosseisus olev töövoomootor Karajan; samuti toetab muuhulgas Grid-teenuseid ka universaalne Triana töövookeskkond.

4.5 Konkreetsed veebiteenuste-põhised lahendused

Taverna

Taverna näol on tegemist mitmete Suurbritannia ülikoolide bioinformaatika ja arvutiteaduste instituutide koostööna arendatava stabiilsesse arengufaasi jõudnud rakendusega, mis on üheks osaks laiemast myGrid projektist. Algselt oli peamiseks eesmärgiks luua keskkond, mille abil oleks briti geenitehnoloogidel ning bioinformaatikutel hõlbus veebiteenustena publitseeritud tööriistadest koostada terviklikke automaatseid protsesse (töövooge) oma katseandmete töötlemiseks ning analüüsiks [3].

Loodud töövoogude salvestamiseks kasutab Taverna spetsiaalset XML-põhist töövookeelt SCUFL; rakendus ise on kirjutatud keeles Java. Taverna teab automaatselt paljude bioinformaatikule erialaselt kasulike veebiteenuste asukohti, mida saab koheselt kasutama hakata. Keskkonna tugevateks külgedeks on selge ja stabiilne kasutajaliides, kasutajasõbralikkus nii installeerimisel kui töö käigus ja head võimalused uute hajusate komponentide lisamiseks wsdl-failide aadresside sisestamise läbi või teiste võimalike liideste kaudu. Samuti väga heaks omaduseks on lokaalsed programmeeritavad BeanShell-tüüpi komponendid, mis võimaldavad töövoo mingis etapis kergesti realiseerida andmeformaatide konverteerimisi või lihtsamat aritmeetikat, kasutades seejuures skriptimiskeelena Javat.

Triana

Triana on töövookeskkond, mille loomist alustati algselt gravitatsioonilaineid uuriva teadusprojekti *GEO 600* raames, eesmärgiga luua toetav töövahend arvutuste automatiseerimiseks. Nüüdseks areneb Triana edasi sõltumatu vabavaralise projektina, mida arendatakse Cardiffi ülikoolis. [1]

Töövoogude salvestamiseks võimaldab Triana kasutada töövookeelt BPEL4WS, kuid on laiendatav töövoogude laadimiseks ja salvestamiseks spetsiaalsete *plug-in*'ite lisamisega. Triana baseerub samuti keelel Java. Javas on realiseeritud ka töövookeskkonnaga kaasa tulevad kohalikud komponendid, mida saab ise juurde kirjutada: vajalik on luua kindlat liidest realiseeriv Java klass, kirjeldada see kindlas formaadis XML-failiga ning muuta mõlemad failid töövookeskkonnale kättesaadavaks. Mähkides komponendi vastavasse Java-keelsesse "ümbrisesse", saab täiendavaid komponente kirjutada ka teistes keeltes.

Triana hajusaid komponente võib samuti kirjutada Javas ning nendega suhelda läbi RMI tehnoloogia. Antud töö seisukohalt oluline on aga see, et toetatakse ka töövoomootori liidestamist hajusate komponentidega läbi veebiteenuste, mis toimub (sarnaselt Tavernaga) teenuseid kirjeldava wsdl-faili aadressi sisestamise. Triana pakub tuge ka Grid-teenustega suhtlemiseks.

5. TÖÖVOO NÄIDE: ILMAJAAM

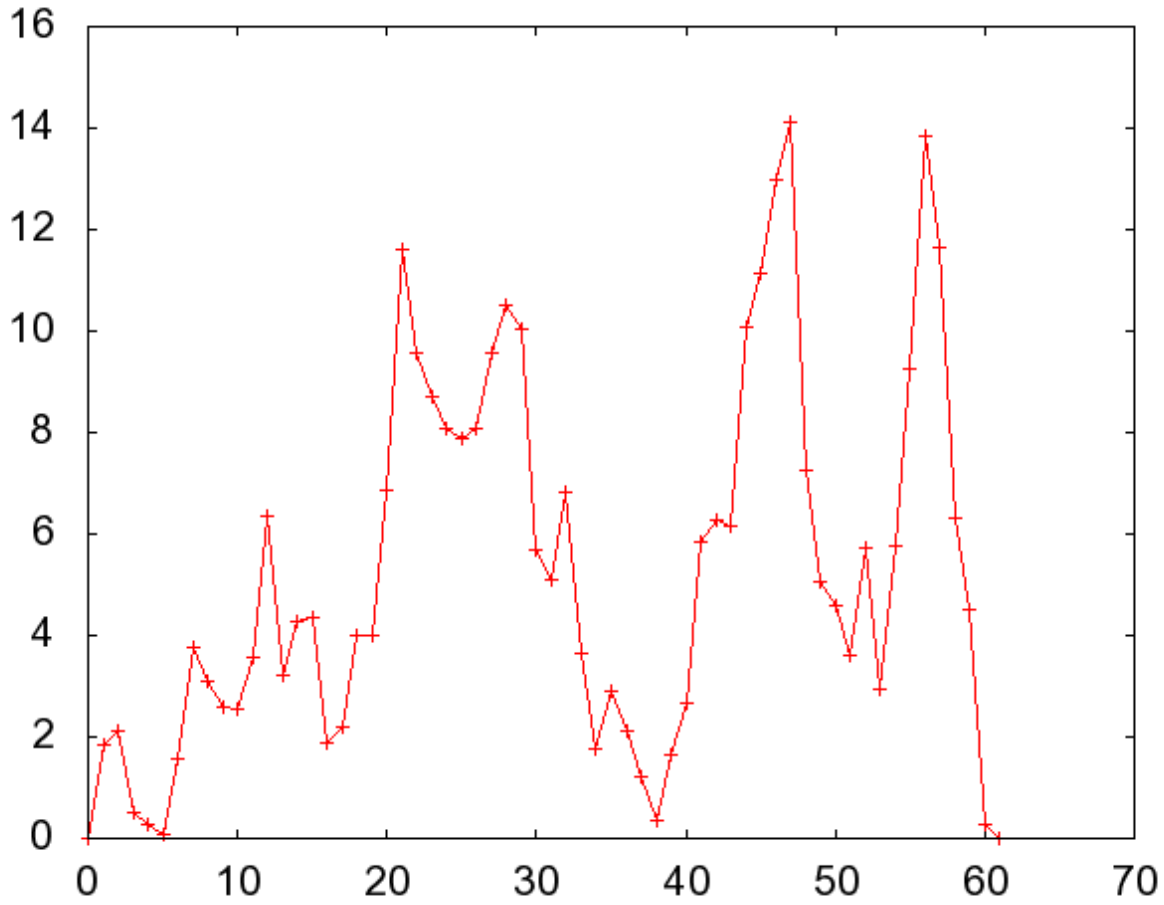
Näide töövoogude võimalikust kasutamisest on realiseeritud Taverna töövookeskkonnas ning integreerib automaatseks arvutusprotsessiks neli sõltumatut veebiteenust. Pärast soovitud perioodi alguse- ja lõpu-kuupäevade seadistamist esimese komponendi argumentideks ning töövoo käivitamist leiab töövoomootor kasutaja edasise sekkumiseta selle perioodi jaoks graafikud õhutemperatuuri ööpäevaste ning ööpäeva keskmiste kõikumiste kohta. Algandmed (õhutemperatuurid igal täistunnil) saadakse Tartu Ülikooli Keskkonnafüüsika Instituudi ilmajaamalt, mis asub Tartu Ülikooli füüsikahoone katusel ning mille mõõtmistulemused on internetis kättesaadavad lehekülje http://meteo.physic.ut.ee vahendusel.

Vajalikud veebiteenused on realiseeritud ühe Ruby On Rails raamistikule kirjutatud rakenduse osadena. Rakenduse kood sisaldub töö lisas 4. Järgnevas teenuste loetelus on iga teenuse juures viidatud ka failidele, milles sisalduv kood seda teenust defineerib. Antud juhul on rakendus üles seatud samasse arvutisse, kus töötab ka töövoomootor (tuleb rõhutada, et üldjuhul see muidugi nii ei ole), mis muudab rakenduse kättesaadavaks virtuaalselt baas-aadressilt http://localhost:3001/. Rakenduse ja töövoomootori samas arvutis käivitamine ei tulene siin mingitest sisulistest piirangutest − teenused saab sisuliste muudatusteta üles seada ka suvalisse võrguserverisse.

Veebiteenus 1: Andmepäring http://meteo.physic.ut.ee leheküljelt.

wsdl-faili asukoht:	http://localhost:3001/data_miner/service.wsdl
failid teenuse koodiga:	data_miner_controller.rb,
	data_miner_service.rb,
	data_miner_api.rb
komponendid töövoograafil:	temperatuurid_internetist

Teenus sooritab sisendparameetreid kasutades *HTTP Post* tüüpi päringu, mis võltsib aadressil http://meteo.physic.ut.ee/et/archive.php asuva veebivormi täitmist. Veebiteenus edastab oma vastusena ilmajaama serverile tehtud päringu otsese väljundi. (Sisuliselt toimub veebivormina realiseeritud kasutajaliidese konverteerimine veebiteenuseks.)



Joonis 13: Töövoo tulemus. Töövoo täitmise lõppedes on üheks tulemiks graafik, kuhu on valitud perioodi jaoks skitseeritud õhutemperatuuride ööpäevased keskmised Tartus. Antud graafik kujutab perioodi 1. märts 2007 kuni 1. mai 2007 (x-telje ühikuteks päevad perioodi algusest, y-teljel keskmised õhutemperatuurid (°C); kuna genereeritavatele graafikutele võimalikult korrektse välimuse andmine ei ole antud töö seisukohast oluline, ei ole sellele hetkel liialt tähelepanu pööratud).

Veebiteenus 2: Ööpäevaste keskmiste temperatuuride arvutamine.

wsdl-faili asukoht:	http://localhost:3001/average_temperatures/service.wsdl
failid teenuse koodiga:	average_temperatures_controller.rb,
	average_temperatures_service.rb,
	average_temperatures_api.rb
komponendid töövoograafil:	keskmiste_arvutamine

Teenus võtab andmeteenuselt saadud väljundi, eraldab temperatuurid kuupäevade alusel gruppidesse, arvutab iga grupi keskmise ning tagastab leitud väärtused graafiku-teenusele sobiva andmeseeria formaadis.

Veebiteenus 3: Andmete formaadi konverteerimine.

wsdl-faili asukoht:	http://localhost:3001/format_converter/service.wsdl
failid teenuse koodiga:	format_converter_controller.rb, format_converter_service.rb, format_converter_api.rb
komponendid töövoograafil:	formaadi_konverteerimine

Teenus eraldab andmeteenuselt saadud vastusest vaid temperatuurid (eemaldades kuupäevad ning kella-ajad), ning esitab nad graafikuid skitseeriva teenuse jaoks sobiva andmeseeriana.

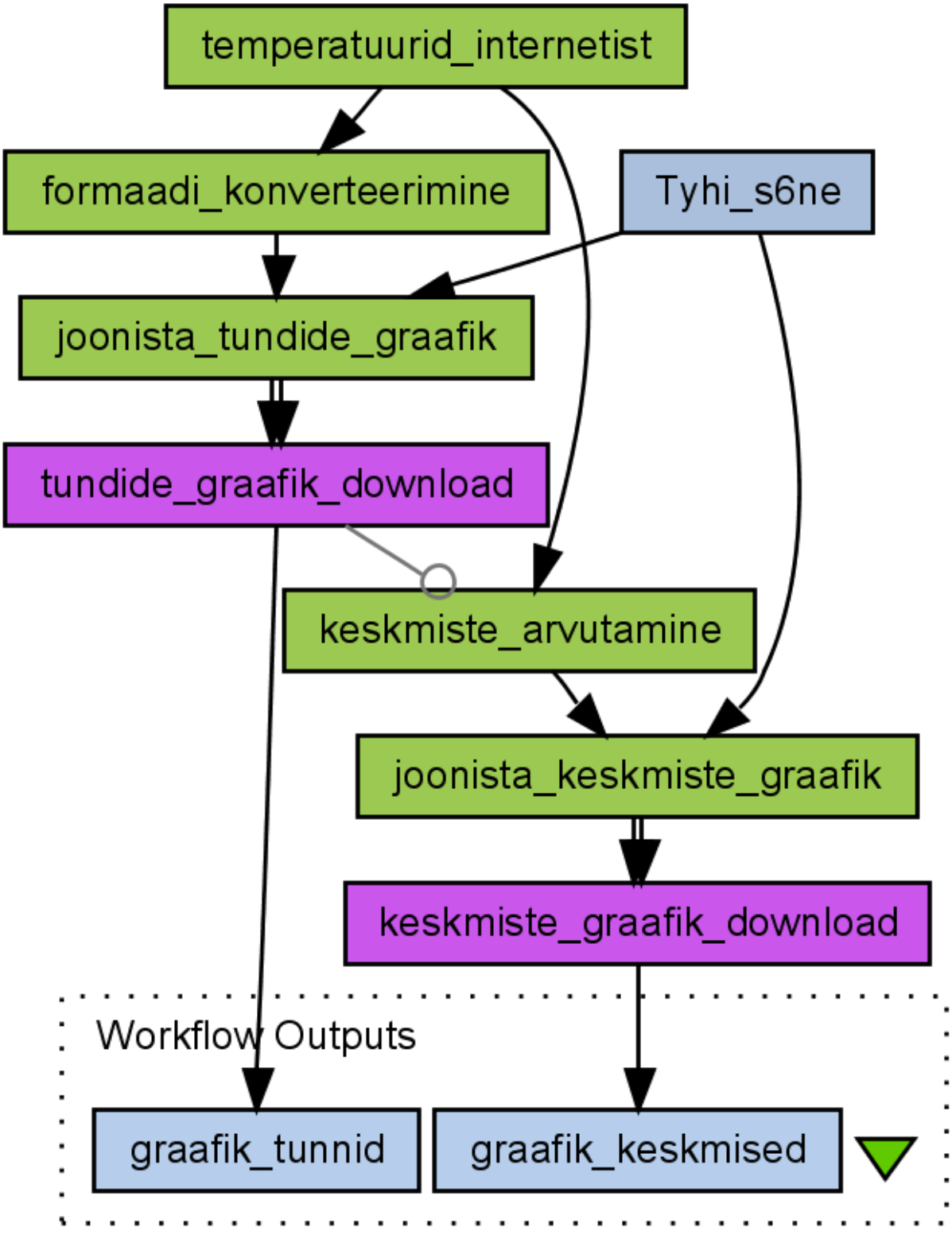
Veebiteenus 4:Graafikute skitseerimine.

wsdl-faili asukoht:	http://localhost:3001/plot/service.wsdl
failid teenuse koodiga:	plot_controller.rb, plot_service.rb, plot_api.rb
komponendid töövoograafil:	joonista_tundide_graafik, joonista_keskmiste_graafik

Teenus ootab sisendina kahte jäigalt fikseeritud formaadiga andmeseeriat, mille vastavad elemendid sisaldavad graafiku järjekordse punkti x-koordinaati ning y-koordinaati. Kui x-koordinaate defineerivat andmeseeriat ei esitata, eeldab teenus, et igale y-andmeseeria elemendile vastavaks x-koordinaadi väärtuseks on elemendi järjekorra number andmeseerias (antud juhul ongi seda kasutatud, sisestades x-koodinaatide andmeseeria kohale tühja sõne – komponent "Tyhi_s6ne").

Selle näite varal on hea seletada ka töövoo ortogonaalsust oma komponentide suhtes ning sellest tulenevat kasu. Töövoogudele loomulik kasutusstsenaariumi oleks antud juhul näiteks selline, et töövoo looja leidis internetist kaks avalikku veebiteenust, mida oli võimalik tema töövoos edukalt rakendada: andmeteenus õhutemperatuuride pärimiseks ning universaalne graafikute skitseerimise teenus. Andmeteenuse formaadi konverteerimisega graafikuteenusele vajalikule kujule ning ööpäevaste keskmiste temperatuuride leidmisega tegelevad komponendid tuli tänu nende spetsiifikale töövookoostajal ilmselt ise realiseerida. Töövoo jätkuva toimimise eelduseks on

see, et avalike veebiteenuste sisendid ja väljundid jäävad samaks − kuidas aga muutub nende teenuste sisu, ei ole oluline. Näiteks kui ühel hetkel peaks Keskkonnafüüsika Instituudi ilmajaam lõpetama oma andmete avalikustamise internetis, võib temperatuure tagastava andmeteenuse haldaja muuta teenust nii, et andmete lähteallikana kasutatakse mõnda teist ilmajaama. Analoogselt võidakse muuta graafikute skitseerimise teenust, hakates näiteks GnuPloti asemel graafikute genereerimiseks kasutama mõnda teist tarkvaralahendust. Siinjuures on oluline, et selliste muudatuste tegemisel ei pea töövoogu ennast muutma, s.t. kui töövoo koostaja ja avalike veebiteenuste haldajad ei ole samad isikud, ei pea töövoo koostaja teenustega toimunud muudatustest isegi teadlik olema.



Joonis 14: Graafina esitatud töövoog. Näitena koostatud töövoog visualiseerituna tsükliteta suunatud graafiks Taverna töövookeskkonnas.

6. TÖÖVOOGUDE OTSTARBEKAD KASUTUSALAD

Ilma pikemate seletusteta peaks olema mõistetav, et ei töövoogude ega mistahes teise sel määral abstraktse tehnoloogiaga ei saa iialgi ulatuslikult asendada traditsioonilist programmeerimist. Kõik taolised katsed on viinud järeldustele, et erinevatest abivahenditest võib olla kasu töö lihtsustamiseks ja tootlikuse tõstmiseks mingis kitsas ja spetsiifilises etapis, kuid sammukenegi väljaspool oma loomulikku nišši jäävad nad kõik traditsioonilise programmeerimise kõrval abituteks ja ebapiisavateks. Järgnevalt üritataksegi piiritleda seda produktiivset nišši veebiteenuseid integreerivate töövoogude jaoks.

Eelnevalt sai aksiomaatiliselt väidetud, et puhtalt lokaalsete komponentide baasilt ei ole mõtekas töövooge koostada. Eelnevalt demonstreeritud Taverna keskkonna poolt pakutud võimalus kasutajaga dialoogiakende abil suhtlemiseks selleks ridagi programmeerimata võib tekitada entusiasmi, kuid kahjuks on see pisut ennatlik. Enamasti on töövoomootoritega vaikimisi kaasas olevate taaskasutatavate lokaalsete komponentide arv äärmiselt piiratud. Vähegi keerulisema programmi koostamisel tuleks hakata ise juurde kirjutama hulganisti komponente, samas osutub uute komponentide loomine, ühendamine ja töövookeskkonnaga liidestamine tavaliselt suhteliselt keerukaks ja ebamugavaks. Nendel põhjustel ei suuda ainult lokaalsetest komponentidest koostatav töövoog ei arenduskiiruselt, lihtsuselt ega ammugi mitte paindlikkuselt võistelda sama funktsionaalsuse loomisega traditsioonilise programmeerimise vahendite abil. Seega eeldab edasine arutlus, et töövoo koosseisus kasutatakse muuhulgas ka hajusaid komponente ehk veebiteenuseid.

Töövoogude olemust kirjeldavas peatükis toodud näited (nt. komponent, mis tagastab hetke kellaaega) on loomulikult vägagi utreeritud. Reaalsus on see, et mingi programmi realiseerimine traditsioonilise programmeerimiskeele vahendite asemel hajusatest komponentidest koosneva töövoona on paratamatult kompromiss, mille käigus tuuakse ohvreid jõudluses, arendamise kiiruses, süsteemi stabiilsuses ning mujal (nn. *overhead* ehk administratiivsed jm. lisakulud). Nende allikad on erinevad:

- **Lisatöö võrguliikluseks kõlbuliku sõnumiformaadi koostamisel/lugemisel.** Veebiteenustena realiseeritud hajusate komponentide poole pöördumisel kulub võrreldes lokaalse programmimooduli väljakutsumisega täiendavat ressurssi *SOAP*-formaadis sõnumi kodeerimise/dekodeerimise peale nii kliendi kui ka serveri poolel.

- **Võrgu latentsus ning piiratud andme-edastuskiirus.** hajusate komponentidega suhtlemise juures.
- **Enda (töövoona realiseeritud) "programmi" töö sõltuvusse seadmine töövoo koostaja poolt mitte-kontrollitavatest asjaoludest.** Tänapäevaks on interneti infrastruktuurist kui sellisest küll saanud midagi, mis on piisavalt stabiilne ning mille peale võib suhteliselt kindel olla, küll aga ei saa seda öelda iga serveri kohta eraldivaadatuna. Kui mõne töövoos kasutatava teenuse asukohaks olev server juhtub , välistab see ka loodud töövoo
- **Täiendav arendusaeg alamprogrammide veebiteenustena publitseerimisel.** Kuigi veebiteenuste baasil töövoo koostamise juures seisneb idee üldiselt selles, et töövoo koostaja kasutab kolmandate osapoolte poolt valmis kirjutatud taaskasutatavaid veebiteenuseid ja pääseb ise nende programmeerimisest, ei tohiks tähelepanuta jätta seda, et keegi on need teenused siiski kunagi loonud. Kuigi tänapäeval leidub palju häid vahendeid valmis programmide veebiteenustena publitseerimiseks, nõuab see siiski teatavat lisatööd ja oskusi.
- **Seotus töövookeskkonna platvormi külge.** Kõigil töövookeskkondadel on omad väga konkreetsed piirangud. Kui ühel hetkel on töövoogu vaja lisada mingit funktsionaalsust, näiteks mingit uut tüüpi liidestega komponente, mille jaoks töövookeskkond otsest tuge ei paku, osutub omapoolne laiendamine tõenäoliselt üsnagi keerukaks. Traditsioonilises programmeerimiskeeles realiseeritud programm on sellisel juhul alati paindlikum.
- **Töövookeskkonna enda tarbitav ressurss.** Graafilise töökeskkonna käimashoidmine ning haldus nõuab samuti teatud ressurssi, mis töövookeskkonna asemel näiteks käsureaskripti kasutamisel jääks tarbimata, kuid tänapäeva arvutite võimsuse juures on see suhteliselt tühine osa.

Olles välja toonud vastuargumendid, üritatakse järgnevalt konstrueerida olukordi, kus töövoogude ja hajusate veebiteensute kasutamisest saadav kasu kaaluks üles kulud. Tinglikult võiks veebiteenuste kasutamise eesmärgid jagada kaheks – juurdepääsu võimaldamine tsentraalsele andmekogule või juurdepääsu võimaldamine võrguserveris asuvale tööriistale –, kuigi esineb ka vahepealseid variante.

Võrguserveris töötavate tööriistade juhul on tihemini põhjuseks, miks tööriista mitte iga kasutaja arvutisse kopeerida ning seal lokaalselt käivitada, tööriista autoriõiguse omaniku soov selle kasutamist piirata või jätta selle sisu avalikustamata. Samuti vabastab tööriista töötamine serveris ning sellele juurdepääsu võimaldamine veebiteenuste abil vajadusest tööriista installeerida iga kasutaja enda arvutisse, mis võib muutuda oluliseks argumendiks, kui tööriist vajab oma tööks

mingeid keerukaid abivahendeid või litsentseeritud ja kallist lisatarkvara.

Tänapäeva teaduse ülitiheda konkurentsi oludes toimub pidev võitlus publikatsioonide, grantide ning lihtsalt uute ideedega esimesena välja tulemise au nimel. Ei ole haruldased olukorrad, kus teadlased soovivad käimasolevatest projektidest küll avalikustada teatud töövilju, kuid teha seda nii, et konkurendid pääsevad neile ligi vaid rangete piirangutega. Näiteks võib tegemist olla uudse analüüsitööriistaga: see soovitakse autorite poolt enne vastava publikatsiooni ilmumist osaliselt avalikuks teha nii, et teistel oleks võimalik seda testida ja sellega töötada, andes ise tööriistale ette sisendandmeid ning saades vastu reaalseid arvutustulemusi, kuid seejuures ei soovita avalikustada tööriista sisemist toimealgoritmi.

Üks populaarne lahendus sellele probleemile on traditsiooniliselt olnud oma tööriista peitmine veebivormi taha: luuakse tavaline internetilehekülg, mis on ligipääsetav mistahes internetikasutajale. Leheküljel asuva vormi täitmisega annab kasutaja ette algandmed, mis antakse pärast kasutajalt vastava korralduse saamist edasi serveris käivitatavale programmile lähteparameetriteks. Käivitatud tööriista töö lõppedes genereeritakse tulemuste põhjal teine internetilehekülg, kust kastutaja näeb enda sisestatud lähteandmetele vastavat programmi väljundit. Seejures ei saa ta teada midagi vormi taha peidetud tööriista sisemisest toimeloogikast (mille varjamine oligi veebivormi kasutamise eesmärk, selle asemel et lubada kasutajal tööriist enda arvutisse alla laadida). Nii lahendatakse küll juurdepääsu limiteerimise probleem, kuid selle lähenemise kitsaskohtadeks on vormi käsitsi täitmise ebamugavus ning oht teha seejuures endalegi märkamatult viga. Samuti on selliselt esitatud tööriista puhul äärmiselt halvad võimalused selle kasutamiseks mingi pikema automatiseeritud arvutusprotsessi alamosana. Veebiteenuste abil on võimalik kergesti realiseerida sarnane juurdepääsu limiteerimine, lastes programmil töötada serveris ning saates kasutajale vaid tema väljundi, kuid vormide kasutamisega võrreldes on siin suuri eeliseid. Tänu tugevalt standardiseeritud veebiteenuste liideskihile on antud juhul (erinevalt veebivormidest) hõlbus automatiseerida ligipääsu taolisele tööriistale, kasutades seda mõne teise programmi osana või koostades töövoomootoril töövooge, mis kasutavad veebiteenust oma komponendina.

Tsentraalsete võrguandmebaaside loomise vajadus tekib tihti: tegemist võib olla liiga suurte andmemahtudega, et igal teadlasel oleks mõttekas neid oma arvutisse kopeerida; samuti võib tsentraliseerituse tingida vajadus pidevalt kõige värskemate andmete järele, aga paljude koopiate kasutamisel tekiksid probleemid andmete sünkroniseeritusega (osutub lihtsamaks andmeid iga kord ühest tsentraalsest serverist uuesti pärida, kui omada paljusid lokaalseid koopiaid ning pidevalt hoolitseda selle eest, et kõik koopiad andmebaasidest sisaldaksid samu ja ühtlasi ka kõige värskemaid andmeid). Veebiteenused võivad siin omada mugava "värava" rolli andmebaasile ligi

pääsemiseks, olles eriti kasulikud juhul, kui teostatavad andmepäringud ei ole triviaalsed. Samuti võib ka andmebaaside puhul kasutada veebiteenuseid selleks, et piirata ja kontrollida ulatust, mil määral kasutaja andmebaasile ligi pääseb. Kasutajale seatavaks piiranguks oleks antud juhul see, et kuna ta ei pääse andmebaasile ligi otse, võimaldab see piirata andmebaasi kasutamist vaid teatud laadi päringute lubamisele või limiteeritud mahuga vastuste tagastamisele.

Samuti esineb tihti olukordi, kus keskse andmekogu valdajaks on üks institutsioon või teadusasutus, kuid nende andmete kasutamisest on huvitatud kümned või sajad teadusprojektid üle maailma. Selle olukorra ratsionaalseks lahenduseks olekski keskne andmeserver, mille ette luuakse juurdepääsu kontrollimise ning turvalisuse kaalutlustel veebiteenuste liideskiht, läbi mille saavad kolmandad osapooled andmebaasist päringuid teha. Tuntud konkreetseks näiteks selle olukorra kohta on KEGG-projekti bioinformaatika vallast [9]. Näiteks on projekti tsentraalse andmebaasi peale loodud veebiteenus, mis annab vastuse bioinformaatikas tihtiesinevale probleemile: on tuvastatud mingi hulk proteiine ning soovitakse teada, milliste proteiinidega need leitud proteiinid omakorda reageerivad. Sellele küsimusele vastuse leidmine ei ole teadusprojektides mitte eesmärk omaette, vaid mingi pikema arvutusprotsessi möödapääsmatu alametapp; töövoomootori abiga on võimalik see protsess automatiseerida, koostades töövoog, mis iga kord vajaduse tekkides pöördub vastava veebiteenuse poole, ilma et teadlane peaks sellega käsitsi tegelema.

Töövoomootorite efektiivse kasutamise näite teaduse vallast annab see, kui kujutada ette töörühma, mis koosneb mitmete teadusasutuste paljudest teadlastest, kelle vahel on jagatud vastutusalad projekti lõikes või kes hoolitsevad erinevate tööetappide realiseerimise eest. Nad peavad erinevatest geograafilistest punktidest reaalajas ligi pääsema projekti raames loodud andmekogudele või värskeimatele versioonidele arendatavatest tööprogrammidest. Kui terviklik arvutusprotsess kujutab endast keerukat protseduuri, mis integreerib mitmete selliste tööriistade tööd ning kasutab andmeid mitmetest võrguandmebaasidest, oleks ilmselt mõttekas luua tööriistadele ja andmebaasidele veebiteenuste liidesed ning see protseduur realiseerida töövoona, muutes ta töövoomootori abiga automaatseks [3]. Sellega garanteeritakse protseduuri üks-üheselt samane korratavus igal käivitamisel suvalise projektiliikme poolt, välditakse võimalikke inimlikest eksimustest tulenevaid vigu ning hoitakse kokku teadlaste aega, kes peavad kogu projekti vältel sama arvutusprotsessi käivitama näiteks kümneid või sadu kordi.

KOKKUVÕTE

Interneti kiire arenguga on tekkinud uued võimalused sidusama ja mugavama koostöö arendamiseks ka teaduses, ning seda ka teineteisest kuitahes suurte geograafiliste distantsidega lahutatud teadusrühmade vahel. Kui hetkel füüsika vallas tuntuimaks hajustehnoloogiaks olev Grid on raskekoeline, massiivne ning mitte kuigi kasutajasõbralik ning adresseerib peamiselt arvutusvõimsuse jagamise probleemi, siis paralleelselt arenev veebiteenustel põhinev andmete hajustöötlus on märksa kergemakaalulisem ja lihtsam. Samuti on veebiteenuste ja neid integreerivate töövoogude tehnoloogia tüüpiline kasutamismotiiv Gridiga võrreldes mõnevõrra erinev, võimaldades pigem veebipõhist juurdepääsu unikaalsetele tööriistadele ja andmekogudele.

Töövooge käsitletakse antud kontekstis kui suunatud tsükliteta graafe, mille tippudeks on mingid alamprogrammid ning mille kaared defineerivad alamprogrammide automaatse käivitamise järjekorda, moodustades programmi-laadse tervikliku automaatse arvutusprotsessi. Töövooge luuakse ning käivitatakse selleks ettenähtud spetsiaalses töövookeskkonnas, mis pakuvad ka graafilise kasutajaliidese töövoo koostamiseks või vähemalt visualiseerimiseks suunatud graafina. Selle keskkonna osa, mis vastutab otsese töövoo käivitamisprotsessi eest, nimetatakse töövoo mootoriks. Erinevad töövookeskkonnad lubavad töövoo koosseisus kasutada erinevat tüüpi komponente (alamprogramme); antud töös vaadeldi lähemalt töövookeskkondi, mis on ette nähtud veebiteenustena realiseeritud komponentide integreerimiseks.

Töövookeskkonna roll seejuures on veebiteenuste kasutamise muutmine lihtsamaks, varjates lõppkasutaja (teadlase) eest liigse tehnilise keerukuse ja muutes hajustehnoloogiad nii kättesaadavaks ka kasutajatele, kellel puuduvad sügav programmeerimisoskus ning veebiteenuste tehnoloogia süvitsi-tundmine. Ühe teadusprojekti raames loodud universaalseid tööriistu saavad pärast seda, kui neile on loodud veebiteenuste liideskiht ning see muudetud kättesaadavaks mõnel veebiserveril, uuesti ära kasutada teised projektid, kus vajatakse arvutusprotsessi mingis etapis sama funktsionaalsust. Komponentidest koostatud töövoog muudab mitmetest etappidest koosneva andmetöötlusprotsessi automaatseks tervikuks, hoides kokku teadlaste aega ning vältides võimalikke inimlikke näpuvigu, mis võiksid esineda nende tööetappide teineteise järel käsitsi teostamisel.

Väga levinud on taoline projektide- ja teadusasutuste-vaheline koostöö näiteks bioinformaatikas ja geenitehnoloogias. Füüsika vallas pole tööriistadele ja andmekogudele veebiteenuste abil juurdepääsu loomine ning nende baasil töövoogude koostamine käesolevaks

hetkeks kriitilist massi saavutanud, et olla teadlastele ringkondades piisavalt laialt teadvustatud ja tunnustatud kui usaldusväärse töövahend. Probleeme, mida selle tehnoloogia abil saaks lihtsustada, jätkub aga ka füüsika osas kindlasti.

KASUTATUD MATERJALID

1. Churches, D., Gombas, G., Harrison, A., Maassen, J., Robinson, C., Shields, M., Taylor, I., Wang, I. Programming scientific and distributed workflow with Triana services. *Concurrency and Computation: Practice and Experience,* 2005*,* **18**, 10, 1021 - 1037.

2. Brown, J. L., Ferner, C. S., Hudson, T. C., Stapleton, A. E., Vettera, R. J., Carland, T., Martin, A., Martin, J., Rawls, A., Shipman, W. J., Wood, M. GridNexus: A Grid Services Scientific Workflow System. *International Journal of Computer & Information Science*, 2005, **6**, 2.

3. Oinn, T., Greenwood, M., Addis, M., Alpdemir, M. N., Ferris, J., Glover, K., Goble, C., Goderis, A., Hull, D., Marvin, D., Li, P., Lord, P., Pocock, M. R., Senger, M., Stevens, R., Wipat, A., Wroe, C. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience,* 2005*,* **18**, 10, 1067 - 1100.

4. Gannon, D., Plale, B., Christie, M., Fang, L., Huang, Y., Jensen, S., Kandaswamy, G., Marru, S., Pallickara, S. L., Shirasuna, S., Simmhan, Y., Slominski, A., Sun, Y. Service Oriented Architectures for Science Gateways on Grid Systems. *International Conference on Service Oriented Computing 2005* (artiklite kogumik). Benatallah, B., Casati, F., Traverso, P., Springer-Verlag Berlin Heidelberg, 2006, 21 - 32.

5. Thomas, D., Hansson, D., Breedt, L., Clark, M., Davidson, J. D., Gehtland, J., Schwarz, A. *Agile Web Development With Rails, 2nd Edition*. Pragmatic Bookshelf, 2006, 411 - 427.

6. Senger, M., Rice, P., Oinn, T. Soaplab − A Unified Sesame Door To Analysis Tools. *Proc UK e-Science programme All Hands Conference 2003*. Nottingham, UK, 2003.

7. Graham, S., Simeonov, S., Boubez, T., Daniels, G., Davis, D., Nakamura, Y., Neyama, R. *Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI*. Sams Publishing, 2005.

8. Snell, J., Tidwell, D., Kulchenko, P. *Programming Web Services with SOAP*. O'Reilly, 2002.

9. Kanehisa, M., Goto, S., Hattori, M., Aoki-Kinoshita, K.F., Itoh, M., Kawashima, S., Katayama, T., Araki, M., and Hirakawa, M. From genomics to chemical genomics: new developments in KEGG. *Nucleic Acids Res*., 2006, **34**, 354-357.

SUMMARY

With the rapid development in the infrastructure of the Internet, new technlogical solutions to improve the co-operation between different scientific institutions separated by any geographical distances have emerged. The best-known distributed computing technology known today in the world of physics is Grid, which mostly addresses the problem of distributing computational power. As opposed to massive and not too user-friendly Grid, the web service based solutions developing in parallel are more light-weight and less complex. Also, the motive behind the use of web service based distributed computing lies more in granting the Internet-based access to unique analysis tools and databases through well-defined interfaces.

The workflows are usually represented as directed acyclic graphs, with vertexes representing the sub-programs and edges the relations between these sub-programs, forming an executable algorithm similar to a traditional computer program. Workflows are created and executed on specific workbenches or integrated development environments (IDEs), usually including a graphical user interface (GUI) which provides means to create workflows as directed graphs or at least visualize them as such. The part of the environment responsible for invoking the workflow is called workflow engine. Different implementations of workflow environments support different interfaces with the workflow's components; in this specific work, the main attention is on workflow environments used to integrate components with web service based interfaces.

In creating complete computational processes out of web service based components, one important role of the workflow environment is to save the end-user (e.g. a scientist willing to calculate something) from having to deal with the complex technical details of web communication, making the technology accessible to users with less experience in programming and only limited knowledge on web services technology. Universal analysis tools created as a part of one project can be re-used in another, after the web service interfaces are provided to them and made available on a publicly accessible web server. A workflow integrates its components into a completly automatical computational process, saving scientists' time and providing a better alternative to the error-prone manual invocation of necessary sub-programs.

In the field of bio-informatics, this form of co-operation amongst scientific workgroups sharing re-usable tools and database access with each other as public web services is widely-spread. Among scientists in physics, the web service based workflows technology is yet to gain popularity.

LISA 1. TERMINITE ÜHTLUSTATUD EESTINDUSED

workflow - töövoog

workflow engine - töövoomootor

workflow node, workflow component - töövoo komponent

service-oriented architecture (SOA) - teenusepõhine arhitektuur

workflow definition language - töövookeel

web service - veebiteenus

integrated development environment (IDE) - integreeritud arenduskeskkond

LISA 2. ILMAJAAMA NÄITE TÖÖVOO TAVERNA (SCUFL) KOOD

```
<?xml version="1.0" encoding="UTF-8"?>
<s:scufl xmlns:s="http://org.embl.ebi.escience/xscufl/0.1alpha" version="0.2"
log="0">
  <s:workflowdescription lsid="urn:lsid:net.sf.taverna:wfDefinition:7aa07209-
f6e9-426f-9678-f3da434d1b10" author="" title="plot" />
  <s:processor name="tundide_graafik_download">
    <s:local>org.embl.ebi.escience.scuflworkers.java.WebImageFetcher</s:local>
  </s:processor>
  <s:processor name="joonista_tundide_graafik">
    <s:arbitrarywsdl>
      <s:wsdl>http://localhost:3001/plot/service.wsdl</s:wsdl>
      <s:operation>Plot</s:operation>
    </s:arbitrarywsdl>
  </s:processor>
  <s:processor name="temperatuurid_internetist">
    <s:defaults>
      <s:default name="start_year">2007</s:default>
      <s:default name="start_month">3</s:default>
      <s:default name="start_day">1</s:default>
      <s:default name="end_year">2007</s:default>
      <s:default name="end_month">5</s:default>
      <s:default name="end_day">1</s:default>
    </s:defaults>
    <s:arbitrarywsdl>
      <s:wsdl>http://localhost:3001/data_miner/service.wsdl</s:wsdl>
      <s:operation>MineTemperaturesData</s:operation>
    </s:arbitrarywsdl>
  </s:processor>
  <s:processor name="formaadi_konverteerimine">
    <s:arbitrarywsdl>
      <s:wsdl>http://localhost:3001/format_converter/service.wsdl</s:wsdl>
      <s:operation>ConvertFormat</s:operation>
    </s:arbitrarywsdl>
  </s:processor>
  <s:processor name="Tyhi_s6ne" boring="true">
    <s:stringconstant />
  </s:processor>
  <s:processor name="keskmiste_arvutamine">
    <s:arbitrarywsdl>
      <s:wsdl>http://localhost:3001/average_temperatures/service.wsdl</s:wsdl>
      <s:operation>FindAverageTemperatures</s:operation>
    </s:arbitrarywsdl>
  </s:processor>
  <s:processor name="joonista_keskmiste_graafik">
    <s:arbitrarywsdl>
      <s:wsdl>http://localhost:3001/plot/service.wsdl</s:wsdl>
      <s:operation>Plot</s:operation>
    </s:arbitrarywsdl>
  </s:processor>
  <s:processor name="keskmiste_graafik_download">
    <s:local>org.embl.ebi.escience.scuflworkers.java.WebImageFetcher</s:local>
  </s:processor>
  <s:link source="Tyhi_s6ne:value" sink="joonista_keskmiste_graafik:data_x" />
```

```
  <s:link source="Tyhi_s6ne:value" sink="joonista_tundide_graafik:data_x" />
  <s:link source="formaadi_konverteerimine:return"
sink="joonista_tundide_graafik:data_y" />
  <s:link source="joonista_keskmiste_graafik:return"
sink="keskmiste_graafik_download:base" />
  <s:link source="joonista_keskmiste_graafik:return"
sink="keskmiste_graafik_download:url" />
  <s:link source="joonista_tundide_graafik:return"
sink="tundide_graafik_download:base" />
  <s:link source="joonista_tundide_graafik:return"
sink="tundide_graafik_download:url" />
  <s:link source="keskmiste_arvutamine:return"
sink="joonista_keskmiste_graafik:data_y" />
  <s:link source="temperatuurid_internetist:return"
sink="formaadi_konverteerimine:source" />
  <s:link source="temperatuurid_internetist:return"
sink="keskmiste_arvutamine:source" />
  <s:link source="keskmiste_graafik_download:image" sink="graafik_keskmised" />
  <s:link source="tundide_graafik_download:image" sink="graafik_tunnid" />
  <s:sink name="graafik_tunnid" />
  <s:sink name="graafik_keskmised" />
  <s:coordination name="FindAverageTemperatures_BLOCKON_Get_image_from_URL">
    <s:condition>
      <s:state>Completed</s:state>
      <s:target>tundide_graafik_download</s:target>
    </s:condition>
    <s:action>
      <s:target>keskmiste_arvutamine</s:target>
      <s:statechange>
        <s:from>Scheduled</s:from>
        <s:to>Running</s:to>
      </s:statechange>
    </s:action>
  </s:coordination>
</s:scufl>
```

LISA 3. ILMAJAAMA NÄITES KASUTATUD KESKKONDADE KONFIGURATSIOONID

Töövoo käivitamise keskkond

Operatsioonisüsteem: Windows XP Pro, Service Pack 2

Java Runtime Environment: versioon 1.5.0_11

Java interpretaator: Java HotSpot Client VM 1.5.0_11

Taverna: versioon 1.5.1

Veebiteenuseid sisaldava rakenduse ligipääsu baas-aadress: http://localhost:3001/

Veebiteenuste serveri keskkond

Operatsioonisüsteem: Windows XP Pro, Service Pack 2

GnuPlot: versioon 4.2

Ruby: versioon 1.8.5

Ruby On Rails: versioon 1.2.2

ActionWebService alamraamistik: versioon 1.2.2

ActionPack alamraamistik: versioon 1.13.2

Ruby rakenduserver: Mongrel 1.0.1, 32-bitise Windowsi binaarne versioon

port: 3001

LISA 4. ILMAJAAMA NÄITES KASUTATUD TEENUSTE KOOD

Kasutades lähtekohana Ruby On Railsi genereeritava standardse projekti skeletti, defineerivad serveri käitumise järgnevad failid:

app/controllers/abstract_service_controller.rb

```
class AbstractServiceController < ApplicationController

  before_filter :debug_request, :set_soapaction_header


  def set_soapaction_header
    request.env["HTTP_SOAPACTION"] = soapaction_name
  end

  def debug_request
    logger.debug(" ===== sain requesti")
    logger.debug(" ===== method : " + request.method.to_s)
    logger.debug(" ===== path : " +  request.path.to_s)
    logger.debug(" ===== xml_http_request ? " + request.xml_http_request?.to_s)
    logger.debug(" ===== content_type : " + request.content_type.to_s)
    logger.debug(" ===== accepts mime type : " + request.accepts.to_s)
    logger.debug(" ===== formatted_post?  " + request.formatted_post?.to_s)
    logger.debug(" ===== parameters : " + request.parameters.to_s)
    logger.debug(" ===== post_format ? " + request.post_format.to_s)
    logger.debug(" ===== all request's @env keys: ")
    request.env.each do |key, value|
      logger.debug(" ---------------- #{key} => #{value}")
    end
    logger.debug(" ===== xml data : " + request.raw_post.to_s)
  end

end
```

app/controllers/data_miner_controller.rb

```
class DataMinerController < AbstractServiceController

  require "app/services/data_miner_service.rb"

  wsdl_service_name 'DataMiner'

  def soapaction_name
    "/data_miner/api/MineTemperaturesData"
  end
```

```
  def mine_temperatures_data(start_year, start_month, start_day, end_year,
                             end_month, end_day)

     DataMinerService::query_for_temperatures(start_day, start_month,
         start_year, end_day, end_month, end_year)
  end

end
```

app/controllers/average_temperatures_controller.rb

```
class AverageTemperaturesController < AbstractServiceController

  require "app/services/average_temperatures_service.rb"

  wsdl_service_name 'AverageTemperatures'

  def soapaction_name
    "/average_temperatures/api/FindAverageTemperatures"
  end

  def find_average_temperatures(source)
    answer = AverageTemperaturesService.find_average_temperatures(source)
    logger.debug(" === FOUND FOLLOWING AVERAGE TEMP.-s :" + answer)
    answer
  end

end
```

app/controllers/format_converter_controller.rb

```
class FormatConverterController < AbstractServiceController

  require "app/services/format_converter_service.rb"

  wsdl_service_name 'FormatConverter'

  def soapaction_name
    "/format_converter/api/ConvertFormat"
  end

  def convert_format(source)
    a = FormatConverterService::convert(source)
    logger.debug("=============" + source)
    a[:y]

  end
end
```

app/controllers/plot_controller.rb

```
class PlotController < AbstractServiceController

  require "app/services/plot_service.rb"
```

```
  wsdl_service_name 'Plot'

  def soapaction_name
    "/plot/api/Plot"
  end

  # If no x-axis values given, use just 0;1;2;..
  def plot(data_x, data_y)
    if (data_x == "")
      logger.debug(" -- using replacement x-axis values")
      a = data_y.split(";").each_index {|i| data_x << i.to_s + ";" }
    end
    PlotService::generate_plot(data_x, data_y)
  end

end
```

app/services/data_miner_service.rb

```
require 'net/http'
require 'uri'

class DataMinerService

  # instead of temperature, can easily be set to query for anything
  # ("9" => "1" basically means "output data for temprerature => true".. see the
query form
  #  at meteo.physic.ut.ee)
  def self.query_for_temperatures(begin_day, begin_month, begin_year, end_day,
end_month, end_year)

    uri_string = "http://meteo.physic.ut.ee/et/archive.php?do=data"
    query_params = { "begin[year]" => begin_year.to_s,
                     "begin[mon]" => begin_month.to_s,
                     "begin[mday]" => begin_day.to_s,
                     "end[year]" => end_year.to_s,
                     "end[mon]" => end_month.to_s,
                     "end[mday]" => end_day.to_s,
                     "9" => "1" }

    url = URI.parse(uri_string)

    req = Net::HTTP::Post.new(uri_string)

    req.set_form_data(query_params, "&")
    res = Net::HTTP.new(url.host, url.port).start { |http| http.request(req) }

    #remove the first line with the title:
    res.body.split("\n",2)[1]
  end

end
```

app/services/average_temperatures_service.rb

```
class AverageTemperaturesService
```

```
  def self.find_average_temperatures(source)

    lines = source.split("\n")
    output = String.new

    previuos_date = String.new
    temperatures_this_date = Array.new

    lines.each do |line|
      if (previuos_date == line.split[0])
        # .. still the same date
        temperatures_this_date << line.split(",")[1].lstrip.to_f
      else
        # new date => calculate the average of the previous date
        unless temperatures_this_date.length == 0
          output << format("%.2f", self.average_value(temperatures_this_date))
          output << ";"
        end

        # .. and reset
        temperatures_this_date = Array.new
        previuos_date = line.split[0]
      end

    end

    # .. and for the last date

    output << format("%.2f", self.average_value(temperatures_this_date))

  end

  private

  def self.average_value(array)
        sum = 0.0
        array.each {|t| sum = sum + t}
        (sum / array.length.to_f)
  end

end
```

app/services/format_converter_service.rb

```
class FormatConverterService

  def self.convert(source)
    x = String.new
    y = String.new

    lines = source.split("\n")

    i = 0
    lines.each do |line|
      elements = line.split(",")
      x << i.to_s.rstrip.lstrip << ";"
      y << elements[1].rstrip.lstrip << ";"
      i = i + 1
    end
```

```
    a = {:x => x, :y => y}
  end

end
```

app/services/plot_service.rb

```
class PlotService

  TEMP_FILES_DIR = "./tmp/gen_files/"
  GENERATED_GRAPHS_DIR  =  "./public/graphs/"

  # refactor to a conf-file..
  PUB_CATALOG_URL = "http://localhost:3001/graphs/"

  def self.generate_plot(string_stream_x, string_stream_y)
    data_x = PlotService.string_to_array(string_stream_x)
    data_y = PlotService.string_to_array(string_stream_y)

    filename_prefix = self.find_next_unused_plotname
    script_filename = "commands.plt"

    # Generate source data file for the plot
    PlotService.generate_datafile(data_x, data_y, filename_prefix)

    # Generate the plot
    PlotService.generate_gnuplot_script(filename_prefix, script_filename)
    PlotService.run_gnuplot(script_filename, filename_prefix)

    "#{PUB_CATALOG_URL}#{filename_prefix}.png"
  end


  private

  # Generates the file for GnuPlot input data
  def self.generate_datafile(x, y, filename)
    file = File.new("#{TEMP_FILES_DIR}#{filename}.dat", "w")

    xy = Array.new
    x.each_index{|i| xy << [x.at(i), y.at(i)] }

    file_stream = String.new
    xy.each do |pair|
      file_stream << pair[0].to_s.ljust(10) << pair[1].to_s << "\n"
    end

    file << file_stream
    file.close
  end

  def self.find_next_unused_plotname
    i = 1
    while File.exist?("#{TEMP_FILES_DIR}plot#{i.to_s}.dat")
      i = i + 1
    end
    "plot#{i}"
  end
```

```
  # Generates the GnuPlot script file with appropriate customizations
  #   Options include:
  #     x_range, y_range - arrays for start/end values on x & y axis
  #                        in format [start, end]
  #     graph_label - adds label to the graph
  #     x_label, y_label - adds label to the corresponding axis
  #     grid - define whether to draw grids on this plot
  def self.generate_gnuplot_script(dest_filename, command_filename,
        options = {})

    g_label = options[:graph_label]
    x_label = options[:x_label]
    y_label = options[:y_label]
    rx = options[:x_range]
    ry = options[:y_range]

    stream = String.new
    stream << "set terminal png giant font \'Arial\' enhanced" << "\n"
    stream << "set output \'#{TEMP_FILES_DIR}#{dest_filename}.png\'" << "\n"
    stream << "set key off" << "\n"
    stream << ((x_label==nil) ? "unset xlabel"  : "set xlabel \"#{x_label}\"" )
<< "\n"
    stream << ((y_label==nil) ? "unset ylabel"  : "set ylabel \"#{y_label}\"" )
<< "\n"
    stream << ((g_label==nil) ? "unset label"  : "set label \"#{g_label}\"" ) <<
"\n"
    stream << "set xrange [#{rx[0]}:{rx[1]}]" << "\n" unless rx == nil
    stream << "set yrange [#{ry[0]}:{ry[1]}]" << "\n" unless ry == nil
    stream << "set grid" << "\n" if (options[:grid] == true)
    stream << "plot \'#{TEMP_FILES_DIR}#{dest_filename}.dat\' with linespoints"
<< "\n"
    stream << "exit" << "\n"

    file = File.new("#{TEMP_FILES_DIR}#{command_filename}", "w+")
    file << stream
    file.close

  end


  def self.run_gnuplot(script_filename, data_filename)
    system("wgnuplot #{TEMP_FILES_DIR}#{script_filename}")
    FileUtils.cp("#{TEMP_FILES_DIR}#{data_filename}.png",
"#{GENERATED_GRAPHS_DIR}#{data_filename}.png")
  end


  def self.string_to_array(string)
    string.split(';').collect { |el| el.to_f }
  end

end
```

app/apis/data_miner_api.rb

```
class DataMinerApi < ActionWebService::API::Base

  api_method :mine_temperatures_data,
```

```
            :expects => [{"start_year" => :int},
              {"start_month" => :int},
              {"start_day" => :int},
              {"end_year" => :int},
              {"end_month" => :int},
              {"end_day" => :int}],
            :returns => [:string]

end
```

app/apis/average_temperatures_api.rb

```
class AverageTemperaturesApi < ActionWebService::API::Base

  api_method :find_average_temperatures,
            :expects => [{"source" => :string}],
            :returns => [:string]

end
```

app/apis/format_converter_api.rb

```
class FormatConverterApi < ActionWebService::API::Base

  api_method :convert_format,
            :expects => [{"source" => :string}],
            :returns => [:string]
end
```

app/apis/plot_api.rb

```
class PlotApi < ActionWebService::API::Base

  api_method :plot,
            :expects => [{"data_x" => :string},
              {"data_y" => :string}],
            :returns => [:string]

end
```