Aktorite mudelil põhinev paralleelprogrammeerimine

Sander Sõnajalg
sander_s@ut.ee

Programmeerimiskeelte semantika uurimisseminar, 2008
Arvutiteaduse Instituut, Tartu Ülikool

Kokkuvõte

Aktorid on mudel paralleelprogrammeerimiseks, mis vastandub traditsioonilisele lõimedega programmeerimise lähenemisele, kus lõimed vahetavad teineteisega infot ühistesse mälupesadesse info salvestamise ja sealt lugemise teel. Aktorid on ühe programmi iseseisvalt ja paralleelselt täidetavad protsessid, mis erinevalt lõimedest mingeid ressursse omavahel ei jaga. Suhtlus aktorite vahel toimub sõnumite vahetamise teel. Sellise mudeli põhjal kirjutatud kood on üldiselt intuitiivsem ja lihtsam ning propageerib sündmuste-põhist lähenemist paralleelse tarkvara realiseerimisel.

Kuna sõnumite vahetamine on lihtsasti realiseeritav nii ühe arvuti piires, üle asutuse-sisese kohtvõrgu kui üle avaliku interneti, sobib aktorite mudel nii paralleelset andmetöötlust nõudva lokaalse tarkvara realiseerimiseks (näiteks veebiserverid) kui ka hajusprogrammeerimise ülesannete lahendamiseks (näiteks mahukad teadusarvutused, mida hajutatakse jõudluse tõstmise eesmärgil).

1. Sissejuhatus

Läbi arvutiteaduse lühikese ajaloo on pidevalt püsinud aktuaalsena küsimus, kuidas panna loogilise terviku moodustava programmi erinevaid osasid töötama teineteisega paralleelselt ning teha seda võimalikult otstarbekalt. Päevakorrast välja langeda ei taha need küsimused just seetõttu, et kuigi erinevaid lahendusi ja meetodeid küll eksisteerib, on nad valdavalt üsna keerukad, ebamugavad ja veaohtlikud. Oludes, kus programmeerijate ja testijate aeg muutub võrreldes riistvaraliste ressurssidega üha kallimaks, paneb see otsima lihtsamaid lahendusi ja lähenemisnurki.

Traditsiooniline viis mitmete protsesside paralleelseks käivitamiseks ühe arvuti piires on kasutada otseselt operatsioonisüsteemi lõimesid (*system threads*). Koostöös toimivate protsesside üksteisega suhtlemine implementeeritakse ühiste muutujate kaudu, mis asuvad protsesside vahel ühiselt kasutatavates mälupesades (*shared memory*). Sellise mudeli kasutamisel tuleb vigade vältimiseks ühiste ressursside kasutamist kuidagi reguleerida, mis toob kaasa palju raskesti-lahendatavaid ja veaohtlikke praktilisi probleeme, näiteks nn. „surnud ringi tekkimine“ (*deadlock*): üks protsess ootab oma töö jätkamiseks teise taga ning teine jällegi esimese taga. Kui paralleelselt täidetav ülesanne asub aga näiteks teises masinas (ning võimalik et ka teises egograafilises paigas), ei saa temaga enam mälupesade jagamise põhimõttel suhelda, ning kasutada tuleb erinevaid üle interneti suhtlemiseks mõeldud andme-edastusprotokolle (veebiteenused,

JXTA, MPI, jms). Ühesõnaga puuduvad ühtsed liidesed lokaalsete ja mitte-lokaalsete paralleelsete protsesside vahel.

Teine mudel paralleelprogrammeerimise realiseerimiseks (või ka teoreetiliseks kirjeldamiseks) on aktorid (*actors*) – teineteisega paralleelselt töötavad, sõnumite vahetamise abil suhtlevad agendid. Aktorit võib selles mõttes võrrelda lõimega, et tegu on vähima iseseisvalt käivitatava programmiühikuga. Aktorid suhtlevad teineteisega sõnumite vahetamise teel (*message passing*), kusjuures ei ole põhimõttelist vahet, kas kaks teineteisega koostöös toimivat aktorit on käivitatud sama protsessori samas lõimes või kahes interneti abil ühendatud masinas, mis asuvad teine teisel pool maakera. Et nii samas arvutis kui erinevates masinates käivitatud aktorid suhtlevad üksteisega ühtviisi sõnumite abil, eksisteerib loomulikul viisil ühtne liides kommunikatsiooniks nii kohalike kui hajusalt paiknevate aktorite vahel. Et aktorid ei jaga teineteisega ühiseid ressursse, jääb ära ka hulk keerukust ressurssidele ligipääsuõiguste haldamiseks (lukustusmehhanismid). Kuigi sõnumi-edastuse mudel suhtlemisel ühe arvuti piires on jagatud mälu mudelist mõnevõrra aeglasem, annab aktoritel põhinev lähenemine üldiselt võidu programmeerija tööajas, tarkvara paremas disainis ja koodibaasi kvaliteedis ning hallatavuses.

Aktoritel põhineva paralleelprogrammeerimise vahendite realisatsioonidest on ilmselt tänini kõige mõjukamaks spetsiaalne paralleelprogrammeerimiseks loodud keel Erlang (mida peetakse selles suhtes nö. *industry standard*'iks). Viimasel ajal on tekkinud erinevaid aktoritega programmeerimist hõlbustavaid raamistikke pea kõigile populaarsetele programmeerimiskeeltele, kuigi ilmselt mitmeidki neist võib pidada veel liiga „noorteks“, et nende peale rajada suuri töökriitilisi tarkvarasüsteeme.

Autori panuseks antud referaadi näol on viidatud teadusartiklite ja muude materjalide põhjal aktorite mudelist koherentse ja süstematiseeritud üldpildi koostamine, mis võrdleb konkreetsete implementatsioonide eripärasid ning toob nende üldistamise teel välja aktorite olulisemad universaalsed põhi-ideed.

2. Paralleelprogrammeerimine ning selle olulisus

Riistvara ja tarkvara areng pöördub üha enam suunaga paralleelsuse ning hajutatuse poole: ilmunud on mitmetest tuumadest koosnevad protsessorid, mitut protsessorit sisaldavad masinad, paljudest masinatest koosnevad arvutusklastrid teadusarvutustes jne. Arvutuslikult mahuka ülesande täitmine võidakse jagada paralleelselt teostatavateks alamülesanneteks puhtalt aja kokkuhoiu mõttes: see võimaldab jagada koormus ära paljude protsessorite vahel, vähendades tunduvalt ülesande lahendamiseks kuluvat koguaega (mis võib teatud juhtudel olla kriitilise tähendusega). Teisest küljest võib hajutamise ja paralleliseerimise põhjustada süsteemi funktsionaalsetest nõuetest tulenev eripära. Selle juhu lihtsa (kuigi pisut kunstliku) näitena võib ette kujutada näiteks reaalajas toimivat oksjoni-rakendust: paljud interneti abil ühendatud kasutajad on oma arvutis käivitanud *desktop*-rakendused, mis omavahel suhtlevad. Keegi kasutajatest algatab oksjoni mingile kaubale, mida soovib müüa. Kõik ülejäänud kasutajad näevad seda pakkumist ning saavad oksjonist reaalajas osa võtta ning oma pakkumisi esitada. Seda süsteemi võib vaadelda kui hajusat programmi, mis justkui jookseb kõikjal ja samas mitte kusagil: kõik ühendatud kasutajad saavad osa programmi funktsionaalsusest, samas süsteemi kui terviku toimimine ei sõltu otseselt ühestki konkreetsest kasutajast.

Paralleelprogrammeerimiseks (*concurrent programming, parallel programming*) nimetatakse tehnikat programmeerimises, kus jagatakse mingi terviklik ülesanne ära väiksemateks alamosadeks, mida on võimalik täita üksteisega samaaegselt (paralleelselt). Alamülesanded võivad olla teineteisega tihedalt seotud: nad võivad kasutada ühiseid lähteandmeid, töö käigus teineteisega informatsiooni vahetada või kasutada oma töö juba lõpetanud alamülesannete tagastatud lõpptulemusi. Nimetame selliseid iseseisvalt käivitatavaid, kuid oma töös teineteisest loogiliselt sõltuvaid programmiüksusi edaspidi (mõnevõrra tinglikult) **protsessideks** (*processes*).

3. Erinevad mudelid paralleelprogrammeerimise realiseerimiseks.

Tänapäeva arvutite/operatsioonisüsteemide puhul on ainsaks otseseks võimaluseks kuidas panna arvuti

täitma mitut erinevat ülesannet üksteisega paralleelselt (ja mitte jadana üksteisele järgnevalt) **operatsioonisüsteemi lõimede** (*system threads*) kasutamine. Kui protsessid käivitatakse erinevates lõimedes, kontrollib operatsioonisüsteem vastavalt oma paremale äranägemisele automaatselt ise, millises lõimes asuvale protsessile kui palju protsessori tööaega võimaldada. Iga natukese aja tagant hetkel aktiivse protsessi töö ajutiselt peatatakse, et võimaldada natuke aega töötada mõnel teisel protsessil jne. Kuna korraga järjest ühele protsessile antavad tööajad on väikesed, tekibki (ka ühe-tuumalise protsessoriga) mulje, et paljud protsessid jooksevad arvutis "paralleelselt". Protsesside *de facto* paralleelne täitmine leiab aset muidugi vaid juhul, kui arvutis on rohkem kui üks tuuma, kuid asju suuremast plaanist vaadates tekib näiv paralleelse täitmise efekt ka ühe tuuma puhul. Viimasel juhul toimub põhimõtteliselt protsesside paralleelse täitmise emuleerimine operatsioonisüsteemi poolt paralleelsust mitte-toetaval riistvaral. Näiteks kui kasutaja on oma töölaual mitmes aknas avanud mitu programmi, on paralleelsuse-illusioon tema jaoks täielik, olgugi et protsessor võib olla ühe-tuumaline ja programme täidetakse tegelikult korda-mööda. Sarnase emulatsiooniga puutume hiljem uuesti kokku kergekaaluliste aktorite juures, kus sama muster kordub taseme võrra kõrgemal: väikesel hulgal operatsioonisüsteemi lõimedel emuleeritakse tunduvalt suurema hulga aktorite paralleelset käivitamist.

Esialgu võib lihtsuse huvides eeldada, et paralleelselt käivitatavad protsessid jooksevad igaüks erinevas operatsioonisüsteemi lõimes. Kuna paralleelprogrammeerimise eesmärgiks on ositi paralleelne, kuid ühtse tervikuna toimiv integreeritud süsteem, siis omavad mõtet vaid sellised lahendused, kus paralleelsed protsessid saavad töö käigus omavahel jooksvalt suhelda. Peamine erinevus erinevate paralleelprogrammeerimise mudelite vahel tulenebki viisist, kuidas see küsimus on otsustatud lahendada. Laias laastus esineb kaks selgelt eristuvat lähenemisnurka:

1) suhtlemine ühiste mälupesade jagamise teel (*shared memory communication*)
2) suhtlemine sõnumite vahetamise teel (*message passing communication*)

3.1. Suhtlemine jagatud mälu abil

Traditsioonilisemaks (kuid ilmselt mitte lihtsamaks) viisiks paralleelseid protsesse omavahel infot vahetama panna on leida mingi viis, kuidas mitu protsessi saaksid töötada kasutades ühiseid muutujaid (*shared variables*). Praktikas tähendab see mingite ühiste ressursside – ennekõike mälu – muutmist nende protsesside vahel ühiselt kasutatavaks. Kogu suhtluse moodustabki andmete vastastikkune lugemine ja kirjutamine jagatud mälupesadesse: kui üks protsess tahab mingil hetkel kasutada mingeid teisele protsessile teada olevaid andmeid, loeb ta lihtsalt vastavas ühiskasutatavas mälupesas asuva jagatud muutuja väärtuse.

Sellist lähenemist komplitseerib peamiselt see, et igal hetkel tuleb mingil viisil garanteerida, et kaks protsessi ei üritaks samaaegselt muuta sama mälupesa sisu. Tavaliselt lahendatakse see probleem mingi ühiste ressursside lukustamise mehhanismi abil: igal konkreetsel hetkel on õigus teatud mälupiirkonnaga manipuleerimiseks mitte rohkem kui ühel lõimel. Praktikas seostub aga lukustamisega terve rida probleeme, mis muudavad selle lähenemise paralleelprogrammeerimisele küllaltki raskesti-mõistetavaks ning veaohtlikuks.

Vaatamata oma probleemidele leiab mälu jagamisel baseeruv paralleelprogrammeerimine väga laialdast kasutamist: sellele lähenemisele on rajatud lõimedega programmeerimise standardteegid nii programmeerimiskeeltes Java, C# kui ka mitmetes teisetes keeltes.

3.2. Suhtlemine sõnumite vahetamise teel

Teine võimalus paralleelprogrammeerimise implementeerimiseks on hoida paralleelsete protsesside ressursid (mälu) teineteisest rangelt eraldi, kuid lasta neil omavahel suhelda üksteisele sõnumite edastamise teel. Selliseid paralleelseid, eraldatud ressurssidega ning omavahel sõnumite abil suhtlevaid agente nimetataksegi **aktoriteks** (*actors*).

Sellise lahenduse puhul jääb ära kogu ressursside lukustamisega seonduv keerukus ning üldiselt on sellel lähenemisel põhinev paralleelprogrammeerimine tunduvalt loomulikum, lihtsamini mõistetav ning kood paremini loetav. Pudelikaelaks saab siin aga sõnumite vahetamise protsess kui selline: võrreldes jagatud mälul põhineva kommunikatsiooniga on ühe masina piires sõnumite vahetamine tunduvalt aeglasem.

Ilmselt sellest lähtuvalt on ka enamus tänapäeval levinud vahevara (*middleware*) kasutanud paralleelprogrammeerimist vajavate komponentide realiseerimiseks valdavalt siiski jagatud mälu mudelit.

Järgnevalt ära toodud tabel üritab neid kahte mudelit erinevatest aspektidest lähtudes võrrelda.

	Jagatud mälu mudel	**Aktorite mudel**
Vähim iseseisvalt käivitatav programmiühik	(Enamasti) lõim	Aktor (enamasti ühe operatsioonisüsteemi lõime kohta mitmeid aktoreid)
Jagatud ressursid	Ühised muutujad jagatud mälupesades	Puuduvad
Protsesside-vaheline suhtlus	Jagatud mälupiirkondade sisu muutmine	Sõnumite vahetamine aktorite vahel
Eelised	Kiirus	1) Intuitiivsem ja loetavam kood 2) Propageerib „kergemate“ *(light-weight*) protsesside kasutamist, mis ei põhine otseselt aeglastel operatsioonisüsteemi lõimedel
Nõrgad kohad	1) Keerukad lukustamis-mehhanismid 2) Pole otseselt laiendatav juhtudele, kus ühist mälu füüsiliselt ei ole	Jõudlus
Realisatsioonid	Paralleelprogrammeerimise standard-teegid Javas, C#-s jne.	Erlang, Scala Actor Framework Functional Java jt.

Joonis 1: Jagatud mälu ja aktorite mudelite võrdlus

4. Aktorite mudeli alusprintsiibid

On oluline rõhutada, et aktorite mudel on pigem abstraktne idee või kontseptsioon kui mingi konkreetne, formaalselt defineeritav objekt. Oma erinevates implementatsioonides võib see idee olla detailides omandanud mõnevõrra erinevaid kujusid; erinevate implementatsioonide aktorid võivad olla pisut varieeruvate omaduste ja võimalustega. Seetõttu on aktoritele mistahes formaalse definitsiooni andmisest suurema praktilise väärtusega idee selgitamine, kasutades näidetena reaalselt eksisteerivaid implementatsioone mis on ennast praktikas tõestada suutnud. Eelnevalt toome siiski ära definitsiooni, mille annab mõistele „aktor“ paralleelprogrammeerimise kontekstis [Agha1985]:

Aktorid on arvutuslikud agendid, mis seavad igale vastuvõetud sõnumile vastavusse järgmise kolmiku:

- lõplik hulk uusi sõnumeid, mis omakorda saadetakse teistele aktoritele
- uus sisemine olek (mis määrab muuhulgas ära viisi, kuidas see aktor edaspidi saabuvatele uutele sõnumitele reageerib)
- lõplik hulk uusi aktoreid, mis sõnumi saamise tulemusena luuakse

Definitsioon 1: Aktorid.

Kuna taoline abstraktne definitsioon annab küllaltki vähe aimu sellest, milline üks aktor tegelikult programmikoodis välja näeks, on hea vaadata kohe kõrvale mõnda reaalset näidet. Et aktorite idee tuleb

süntaktilises mõttes väga selgelt ja kergesti-loetavalt välja näiteks programmeerimiskeelele Scala kirjutatud Scala Actor Framework'is (edaspidi lühendatud SAF), siis kasutataksegi näidete toomisel edaspidi just seda implementatsiooni. Järgnev on näide lihtsast loendurina töötavast aktorist [Haller2006-2].

```
class Counter extends Actor {
  override def run: unit =
    loop(0)

  def loop(value: int): unit = {
    Console.println("Value: " + value)
    receive {
      case Incr() =>
        loop(value + 1)
      case Value(p) =>
        p ! value
        loop(value)
      case _ =>
        loop(value)
    }
  }
}
```

Kood 1: Scalas laiendatakse aktori defineerimiseks raamistiku baasklassi `Actor`. Meetodi `run` üle-katmisega defineeritakse esialgne käitumine aktori algsel käivitamisel; klassi ülejäänud sisu üle otsustamises jäävad programmeerijale vabad käed. Antud aktor töötab kui loendur, käivitades lõpmatu tsükli, mis alustab lugemist nullist ja uute sõnumite saabudes: a) sõnumile `Incr()` reageerib oma sisemise loenduri väärtuse suurendamisega ühe võrra; b) sõnumile `Value(p)` reageerib aktorile `p` sõnumiga oma loenduri väärtuse saatmisega; c) kõik teised sõnumid töötleb küll ära, kuid midagi sisulist nendele reageerimisel ei tee.

Eelnevat näidet silmas pidades toome järgnevalt välja ja püüame analüüsida olulisemaid omadusi, mis aktorite mudelile iseloomulikud on.

4.1. Asünkroonsus teadete edastamisel

Teadete vahetamine aktorite vahel toimub **asünkroonselt** (*asynchronous message passing*), s.t. saatja saadab teate ära ega jää ootama mingit vastust ega kinnitust selle kohta, et teade on kohale jõudnud, vaid jätkab oma teiste tegevustega. (Kui näiteks kaks aktorit asuvad erinevates, võrguühendusega ühendatud arvutites, ei garanteeri iseenesest miski, et saadetud teade ka reaalselt kohale jõuab. Seetõttu on seda sõnumite vahetamise skeemi vahel ka nimetatud nn. „saada ja palveta“ mudeliks (*send-and-pray*).) Asünkroonsus tähendab ühtlasi seda, et saatja ei saa sõnumi adressaadilt küsida, kas see on sõnumi saatmise hetkel uutele sõnumitele reageerimiseks üldse valmis (või on näiteks alles hõivatud eelmise sõnumi töötlemisega). See loob vajaduse kasutada aktorite juures puhvreid ehk nn. „kirjakaste“ (*mailbox*) töötlemata sõnumite hoidmiseks.

Nagu ülaltoodud näidiskoodist näha, näeb programmeerimine aktorite abil üldiselt välja täpselt nagu programmeerimine ikka; lisanduvad lihtsalt kolm aktorite-spetsiifilist tegevust: sõnumite saatmine, sõnumitele vastamine ja uute aktorite loomine. Neid on nimetatud ka aktoritega programmeerimise „primitiivideks“ ehk fundamentaalseteks ehitusblokkideks (*concurrency primitives*). Vaatleme kahte sõnumi-edastust puudutavat tegevust lähemalt:

1) **uue sõnumi saatmine teisele aktorile**
   - SAFs on selleks meetod “`!`“ – ülaltoodud näites tähendab rida “`p ! value`“ aktorile `p` muutuja `value` väärtuse saatmist.
   - Sõnumi saatnud aktor jätkab vastust ootamata koheselt oma tegevust (antud juhul lihtsalt käivitab uuesti meetodi `loop` ja on valmis reageerima uutele sõnumitele).

2) **sobiva sõnumi töötlemine kirjakastist**
   - SAFs selleks meetod `receive`. Ülaltoodud näites tegelebki meetod `loop` peamiselt sõnumitele reageerimisega.

- Alustades kõige esimesena saabunud (kõige vanemast) sõnumist, proovitakse meetodi `receive` täitmisel kõik aktori kirjakastis olevad sõnumid ühekaupa läbi, üritades igatühte neist järjest sobitada mõne `receive`-blokis sisalduva näidisega (`case Xxx`). Esimese sobitumise puhul valitakse see sõnum töötlemiseks. Antud näites töödeldakse igal `receive` kutsel alati kirjakasti kõige vanem sõnum, sest viimane näidis "_" sobitub kõigega.

4.2. Sündmuste-põhine lähenemine

Inimese jaoks kõige lihtsamini-loetavad ja paremini-mõistetavad programmid on vaieldamatult need, mis kasutavad samu termineid ja jäljendavad samu skeeme, milles inimene ise on harjunud mõtlema. Ilmselt just sellepärast on näiteks algajatele programmeerijatele tunduvalt lihtsaminimõistetav imperatiivne programmeerimine, kus asjad juhtuvad üksteise *järel* ja teatud *tingimustel* (*if-then-else* laused jne), kui deklaratiivne programmeerimine, kus tuleb mõelda pigem matemaatilistes terminites nagu funktsioonide määramis- ja tõesuspiirkonnad jne. (nt. Prolog). Esimene mõtteviis neist kahest on lihtsalt inimesele tunduvalt loomupärasem ja omasem. Sarnase võrdluse võib tuua ka paralleelprogrammeerimise kohta jagatud mälu mudeli või aktorite mudeli abil.

Nimelt paralleelprogrammeerimises oleks inimesele ilmselt kõige loomupärasem võtta süsteemi analüüsimisel aluseks mingid **sündmused**, mis süsteemi osadega toimuvad. Näiteks kui kaks protsessi üritasid kumbki erinevast otsimispuu harust leida õiget vastust mingile otsimisülesandele, võiks sündmuseks olla see, kui üks protsess leiab oma harust vastuse. See protsess, mis tegeleb vastuse otsimisega teisest harust, peaks antud sündmusele reageerima sellega, et lihtsalt lõpetab oma tegevuse (kuna on selge, et tema harus õiget vastust enam olla ei saa).

Jagatud mälu mudelil põhinevad paralleelprogrammeerimise teegid kipuvad reaalsel koodikirjutamise tasandil sündmuste-põhisest mõtlemisest küllaltki kaugele minema. Erinevad protsessid mitte ei *oota* ise millegi neile vajaliku juhtumist et siis oma tegevusega edasi minna, vaid on lihtsalt passiivses olekus („magavad") ja programmeerija peab mingi sündmuse toimudes käima ise vastavaid lõimesid eksplitsiitselt „äratamas" (*wake*) või „magama panemas" (*sleep*). Selline lähenemine muudab programmeerimise tülikamaks ning loogiliste seoste mõistmise ja kirjeldamise süsteemi erinevate osade vahel raskemaks. Palju intuitiivsem oleks selline programmeerimise mudel, kus protsessid saaksid koodi tasemel otseselt deklareerida, millised mujal süsteemis toimunud sündmustest neid huvitavad ja kuidas nad soovivad neile reageerida. Seda kontseptsiooni on nimetatud ka **sündmuste-põhiseks programmeerimiseks** (*event-driven programming*). Nagu kohe näeme, asub aktorite mudel võrreldes traditsioonilise lõimede-põhise paralleelprogrammeerimisega sündmuste-põhise programmeerimise kontseptsioonile tunduvalt lähemal.

Hea piltlik näide sündmuste-põhisest töövoost on toodud ühes aktoritest kirjutavas veebi-blogis[2] (*Tsitaat 1*).

> „Kui Bob astub Alice'i kontorituppa ja palub Alice'il koostada ja talle tuua eelmise kuu majandustulemuste aruanne, ei jää ta uksele seisma ja ootama, kuni Alice selle tehud saab. Samuti ei hakka ta iga natukese aja tagant kontrollimas käima, kas Alice on aruande koostamise juba lõpetanud, ja kõige viimane asi mis tal tuleks pähe teha, oleks minna magama ja iga natukese aja tagant üles ärgata, et kontrollida, kas aruanne on vahepeal valmis saanud. Selle asemel läheb Bob tagasi oma tuppa ja jätkab oma muu töö tegemist, kuna teab, et kui Alice on aruande valmis saanud, siis tuleb ta ja toob selle talle ise. Alice ja Bob on iseseisvad paralleelsed töötajad, seega ei oota nad teineteise taga."
>
> *Tsitaat 1:* Piltlik näide sündmuste-põhisest lähenemisest.

Selle näite põhjal võiks sündmuste-põhise lähenemise eesmärkidena välja tuua kaks peamist nõudmist, mida peaks olema paralleelsete protsesside („töötajate") programmeerimisel lihtne koodis kirjeldada:

1) Protsessid peaksid saama kergesti ja otseselt deklareerida oma huvi mingi sündmuse vastu, et saada

2 http://apocalisp.wordpress.com/2008/07/28/threadless-concurrency-with-actors ; autor pseudonüümiga Apocalisp.

sellest teavitatud ja sellele adekvaatselt reageerida kui see aset leiab.
2) Kui üks paralleelne protsess ootab mingit sündmust, ei peaks see tal takistama oma muu tööga jätkamist.

Kuna mingis muus vormis interaktsioone aktorite vahel ei saa toimuda, siis on loomulik võrdsustada „sündmuse“ mõiste uue teate saabumisega aktori kirjakasti. Kood 2 näitab, kuidas antud stsenaariumist lähtudes ilmselt programmeeritaks Bobi käitumine tavalise imperatiivse programmeerimise mõtlemisest lähtudes. Kui Bob tahab aruannet, siis ta justkui kutsub vastavat Alice'i funktsiooni (olgugi et ta teeb seda sõnumi saatmise abil), ja jääb ise ootama, kuni see tagastatab väärtuse (sisuliselt: kuni Alice saadab talle aruande).

Seevastu sündmuste-põhise programmeerimise stiilis lahendus antud stsenaariumile on toodud koodis 3: Bob annab Alice'ile asünkroonselt teada, et ta sooviks saada aruannet, ja jätkab ise oma muu tööga. Pärast iga tööülesande lõpetamist vaatab Bob uuesti oma kirjakasti, ja kui Alice on vahepeal aruande valmis saanud ja talle ära saatnud, jätkab Bob aruandlusega tegelemist (ignoreerime siinkohal lihtsuse mõttes teadete saabumis-järjekorda: selleks, et Bob hakkaks pärast aruande saabumist kindlasti *kohe* aruandlusega tegelema, mitte ei võtaks lihtsalt kirjakastist kõige vanemat talle saadetud sõnumit, tuleks implementeerida ka sõnumite prioriteetide-tasemed).

```
class Bob extends Actor {
  override def run(): unit = startNewTask()


  def startNewTask(): unit = {
    Console.println("[Bob] ready for new work!")

    // Bob ootab uusi töökorraldusi (ehk sõnumeid kolleegidelt)
    receive {

      // Bob saab näiteks oma sekretärilt meeldetuletuse, et ta peaks üle vaatama
      // eelmise kuu aruandluse
      case TPSRequiringTask() =>

        // Bob saadab Alice'ile palve koostada ja saata aruanne
        alice ! AskTPSReport(this)

        // Bob jääb ootama, kuni Alice talle aruande saadab
        receive {
          case TPSReportReceived(report) =>
            parseTPSReport(report)
            // Järgmiseks ülesandeks on Bob valmis alles pärast Alice'ilt aruande
            // saamist ja selle töötlemist.
            startNewTask()
        }

      // Teised teated Bobil (näiteks teised tööülesanded)
      case Other1 => ..
      case Other2 => ..
      ..
    }
  }
}
```

Kood 2: Imperatiivsele programmeerimisele tüüpiline lähenemine: Bob jääb ootama Alice'i vastust (kommentaarid koodis).

```
class Bob extends Actor {
  override def run(): unit = startNewTask()

  def startNewTask(): unit = {
    Console.println("[Bob] ready for new work!")

    // Bob ootab uusi töökorraldusi (ehk sõnumeid kolleegidelt)
    receive {

      // Bob saab näiteks sekretärilt oma meeldetuletuse, et ta peaks üle vaatama
      // eelmise kuu aruandluse
      case TPSRequiringTask() =>

        // Bob saadab Alice'ile palve koostada ja saata aruanne
        alice ! AskTPSReport(this)
        // Bob EI JÄÄ aruannet ootama, vaid läheb oma muu tööga edasi
        startNewTask()


      // Bob jätkab tööd aruandlusega siis, kui näeb oma kirjakastist,
      // et Alice on talle aruande saatnud
      case TPSReportReceived(report) =>
        parseTPSReport(report)
        startNewTask()

      // teised teated Bobil (näiteks teised tööülesanded)
      case Other1 => ..
      case Other2 => ..
      ..
    }
  }
}
```

Kood 3: Sündmuste-põhisele programmeerimise põhimõtetega kooskõlas olev lähenemine: Bob ei jää vastust ootama, vaid „deklareerib oma huvi“ vastava sündmuse vastu (aruande saabumine Alice'ilt) ja jätkab oma muud tööd.

4.3. Lõime-põhised vs. kergekaalulised aktorid

Kuigi põhimõtteliselt aktorite mudel ise selle tehnika kasutamiseks vähimalgi määral ei kohusta, peetakse tänapäeval loodavate aktorite-raamistike juures väga oluliseks seda, et aktorid oleksid realiseeritud nö. kergekaalulistena. **Kergekaalulisteks aktoriteks** (*light-weight actors*, allikas [Haller2006-1] ka *thread-less actors*) nimetatakse aktoreid, mis ei ole otseselt seotud all-oleva platvormi lõimedega.

Naiivne viis aktoritega programmeerimise raamistiku realiseerimiseks oleks käivitada iga aktor eraldi lõimes. Sellise lähenemise korral oleks aktorite realiseerimine üsna lihtne: saaksime otseselt toetuda olemasolevatele lõimedega programmeerimise vahenditele. Teatavasti on aktorite raamistiku loomise juures peamiseks väljakutseks realiseerida õigete semantiliste omadustega aktorite mudeli kaks peamist primitiivi: asünkroonne sõnumite saatmine ning vastuvõtmine. Näeme, et kui iga aktor asub eraldi lõimes, on seda teha üsna lihtne. Nimelt selleks, et panna aktorit ootama mingit uut sõnumit, on vaja see lõim lihtsalt ajutiselt peatada. Sõnumi saatmisel, vastupidi, vaatab saatja, kas sõnumit vastuvõtva aktori lõim on aktiivne või mitte. Kui adressaat on hõivatud, siis lisatakse sõnum lihtsalt aktorile postkasti; kui adressaat aga on inaktiivne, tuleb vastav lõim lisaks ka „üles äratada“ (*wake*). Aktori inaktiivseks muutumise hetkel ümbritseva konteksti salvestamise ja säilitamise eest vastutab all-olev platvorm (näiteks Java lõimede puhul Java Virtual Machine), ja aktori äratamisel jätkub töö sealt, kust enne pooleli jäi. Kahjuks aga ei skaleeru selline lahendus kuigi hästi. Näiteks Java lõimed on otseselt seotud omakorda operatsioonisüsteemi lõimedega, mis aga tähendab, et uute aktorite loomine oleks sellise implementatsiooni korral arvutuslikult kallis (operatsioonisüsteemi lõimed on ressursside mõttes „kallid“). Samuti oleks piiratud paralleelsete aktorite maksimaalne koguarv – nii Java kui C# suudavad toime tulla maksimaalselt umbes 2000 paralleelse lõimega [Armstrong2003].

Seetõttu peetakse aktorite mudeli realisatsioonide juures oluliseks, et aktorid ei oleks üks-ühele seotud lõimedega, vaid kuitahes palju aktoreid võiks joosta koos samas lõimes. Samas saab lõimes paiknevatest aktoritest igal hetkel olla aktiivne korraga vaid üks. See nõudmine komplitseerib olukorda tunduvalt, kuna üles kerkib uus keeruline küsimus: kuidas pidada meeles inaktiivsete aktorite konteksti?

Vaatleme näiteks koodi 4.

```
val x = receive { case a => a }
val y = receive { case b => f(b) }
g(x, y)
```

Kood 4: Blokeerumisel teise `receive`'i juures peab meeles pidama eelnevat ja järgnevat konteksti.

Kui aktor jõuab teise reani ja jääb ootama teadet muutuja `b` väärtusega, tuleb ta muuta inaktiivseks ja jätta meelde ümbritsev kontekst (näiteks muutuja `x` väärtus). Kui toimub „sündmus" ehk saabub sobiv teade muutuja `b` väärtusega, muudetakse aktor taas aktiivseks ja täitmine jätkub sealt, kust enne pooleli jäi – arvutatakse `g(x,y)`, kasutades kontekstis meeles peetud `x` väärtust.

Näiteks Scala Actor Frameworki realiseerimisel on see probleem lahendatud sulundite (*closures*) kasutamise abil. Mõnevõrra lihtsustades võib öelda, et sulundid võimaldavad põhimõtteliselt teatud osa programmikoodi „kokku pakkida" ja mingisse muutujasse salvestada, ning hiljem seda muutujat käivitada kui funktsiooni, kusjuures käivitatakse varasemalt kokku pakitud kood. Iga kord kui aktor `receive` meetodi juures blokeerub (leidmata oma kirjakastist jätkamiseks sobivat kirja), salvestab ta oma `receive` meetodi sisu sulundina oma isendimuutujasse. Seejärel viskab ta teatud erindi, andmaks märku, et on muutunud inaktiivseks ja teised aktorid võivad oma tööd jätkata. Kuna sulundisse salvestatakse just `receive` meetodi sisu, nõuab selline lähenemine muuseas ka pisut teistsugust (aga sisult ekvivalentset) koodi kirjutamise stiili: kõik sõnumi töötlemisele järgnevad tegevused tuleb tõsta meetodi receive sisse, sõnumitöötlemise koodi järele. Näiteks kood 4 tuleks ümber kirjutada viisil, mis näidatud koodis 5.

Kui selle aktori kirjakasti saabub uus jätkamiseks sobiv teade, võetakse salvestatud sulund (mis

```
receive {
  case a =>
    val x = a
    receive {
      case b =>
        val y = f(b)
        g(x, y)
    }
}
```

Kood 5: Eelmise koodiga ekvivalentne kood, mis on ümber kirjutatud nii, et toetada `receive` käskude juures peatumise puhul edasiste tegevuste salvestamist sulundina.

sisaldab endas aktori „olekut" ehk on funktsioon tegevustest, mida see aktor edasi pidi tegema) ja käivitatakse see, andes argumendiks saabunud teate [Haller2006-2]. Selline lähenemine on ühtlasi väga heas kooskõlas eelpoolkirjeldatud sündmuste-põhise lähenemisega paralleelprogrammeerimisele (muidu inaktiivsed aktorid reageerivad saabuvatele teadetele).

Lõpetame aktorite mudeli SAF realisatsiooni vaatlemise analüüsi sellega, et vaatleme, millised Scala programmeerimiskeel keelelised vahendid selle raamistiku realiseerimisel oluliseks osutusid:

- **näidise-sobitus** – osutus väga kasulikuks selleks, et saabunud teadete hulgast postkastist leida sobiv.
- **kõrgemat järku funktsioonid, sulundid** – osutus hädavajalikuks selleks, et pidada meeles inaktiivsete aktorite olekut, milles nad inaktiivseks muutumise hetkel olid.

Mõlemad need võimalused on tüüpilised just funktsionaalsetele keeltele ning puuduvad näiteks Javas (samas, hetkel väljatöötamisel oleva keele uue versiooni Java7[3] lõppvariandis on sulundid juba tõenäoliselt olemas). Paneme tähele, et ka esimesed silmatorkavamad aktorite mudeli implementatsioonid põhinevad enamikus funktsionaalsetel keeltel: nii Erlang kui Scala on funktsionaalsed (lähemalt peatükkis „Aktorite mudeli realisatsioonid"). Samas Java keelele korralik kergekaaluliste aktorite raamistik pikka aega üleüldse puudus. Siit võib teha järelduse, et kergekaaluliste aktorite implementeerimine esitab suuri nõudmiseid ka

3 Arenduse ametlik lehekülg: https://jdk7.dev.java.net/ ;
Loetelu ettepanekutest keele laiendamiseks: http://tech.puredanger.com/java7

programmeerimiskeelele ja selle võimalustele/väljendusrikkusele, ja on loomulikum ennekõike just funktsionaalsetes keeltes.

5. Aktorite mudel hajusprogrammeerimises

Senises käsitluses on enamasti vaikimisi eeldatud, et programmi paralleliseerimine toimub ühe arvuti piires. Tegelikult aga osutub, et aktorite mudel – erinevalt jagatud mälu mudelist – on täiesti naturaalsel viisil laiendatav ka üldisemale olukorrale, kus üksteisega seotud paralleelsed protsessid võivad käivituda geograafiliselt kuitahes hajutatud arvutivõrgu erinevatel masinatel. See üldistatavus tuleneb otseselt sellest, et aktorite mudel ei aseta programmi käivitavale infrastruktuurile ühte väga olulist piirangut, mida teeb jagatud mälu mudel: füüsiline juurdepääs ühisele mäluressursile. Aktorite mudeli ainus nõue on see, et aktorid saaksid *mingil viisil* (jättes detailid konkreetse implementatsiooni täpsustada) teineteisele sõnumeid saata. Et aga tänapäeval on valdav enamus arvuteid nii-ehk-naa interneti abil ühendatud ning enamus asutustes toimivad ka ülikiired majasisesed kohtvõrgud, siis on selge, et see nõue on ühise füüsilise mäluressursi nõudega võrreldes tunduvalt leebem.

Et aktorid ei sõltu teineteisest mitte mingil muul moel kui teineteisele saadetavate sõnumite kaudu, ja et rakenduste-vaheliseks sõnumite edastamiseks üle interneti eksisteerivad mitmed stabiilsete lahendused, on äärmiselt loomulik idee paigutada paralleelsed aktorid eri masinatesse ja panna nad omavahel suhtlema kohtvõrgu või interneti vahendusel. Sellisel juhul räägime sisuliselt juba **hajusprogrammeerimisest** (*distributed programming*). Motiiviks võiks siin olla näiteks täiendavate arvutusvõimsuste kaasamine (erinevad aktorid koormavad erinevaid arvuteid), või mingite teenuste muutmine kättesaadavaks just seal, kus neid vaja on (nt. panga-automaadid linnatänavatel). Enamus tänapäevastest aktorite-raamistikest ka toetavad aktorite käivitamist erinevates masinates: kindlasti teevad seda näiteks Erlang, Scala Actor Framework ja SALSA.

Detailides võivad erinevad implementatsioonid pisut erineda, kuid üldised ideed on sarnased. Hajusate aktorite toetamiseks tuleb raamistikul üldjoontes lahendada neli peamist probleemi [Haller2007]:

1) sõnumi-edastus üle võrgu
2) sõnumite konverteerimine bitijadadeks (ehk serialiseerimine)
3) aktorite identifikaatorite haldamine
4) võrguühenduste haldamine

Et oleks võimalik tuvastada, kuhu ühele või teisele aktorile mõeldud teade füüsiliselt edastada tuleb, peavad aktorid olema tähistatud **globaalselt unikaalsete identifikaatorite** (*globally unique identifiers*) abil, mis võimaldavad iga teate jaoks leida üles selle adressaadi. Näiteks Scalas on identifikaatoriteks paarid, mille esimene komponent on arvuti globaalselt unikaalne nimi ja teine komponent on aktori lokaalselt unikaalne nimi selles arvutis [Haller2007]. Et programmeerimiskeele siseselt on sõnumid enamus raamistike korral selle programmeerimiskeele objektid ja seega pigem hierarhilise struktuuriga, aga üle interneti saab saata vaid bitte ja baite, tuleb sõnumid ka enne saatmist konverteerida bitijadadeks ehk **serialiseerida** (*serialize*).

Üldiselt ei pea aktor teadma, kas teine aktor, kellega ta tahab suhelda (sõnumit saata või vastu võtta), asub temaga samas arvutis või mitte; samas on siiski tihti kasulik sellega arvestada, kuna sõnumiedastus üle mistahes võrgu on kordades aeglasem, kui sõnumiedastus ühe arvuti piires. Seega peaks aktoritel põhinevat hajusat süsteemi realiseerides üritama seda kindlasti optimeerida just erinevates masinates asuvate aktorite vaheliste sõnumiedastuste arvu suhtes.

Kuna sõnumiedastust kohaliku masina piires on võimalik ka kiiremini korraldada kui oleks vastavate teadete saatmine võrguprotokolli abil ise-endale (*localhost*-i), implementeerib näiteks Scala Actor Framework kohalikele aktoritele ja mujal paiknevatele aktoritele teate saatmist mõnevõrra erinevalt. Need sisemised detailid on aga peidetud raamistiku välise liidese taha ja raamistikku kasutav programmeerija sellest midagi teadma ei pea.

6. Aktorite mudeli realisatsioonid

6.1. Programmeerimiskeel Erlang

Erlang on programmeerimiskeel, mille firma Ericsson lõi spetsiaalselt paralleelprogrammeerimise ülesannete lahendamiseks ning mis on eriti rakendatav kõrge veakindluse-vajadusega reaalaja-süsteemides. Olnud alguses vaid firma-siseseks kasutamiseks, avaldati Erlang 1998. aastal ka *open-source* versioonina.

Keel ise on funktsionaalne ja dünaamilise tüüpimisega. Paralleelprogrammeerimise vahendid on keelde sisse ehitatud ning realiseerivad eksplitsiitselt ja otseselt aktorite mudeli ideid. Nagu aktorite-põhistele paralleelprogrammeerimise vahenditele kombeks, on ka Erlangis aktorid implementeeritud kergekaalulistena (puudub üks-ühene seos operatsioonisüsteemi lõimedega). Olles üks esimesi, edukamaid ja „puhtamaid“ aktorite kontseptsiooni implementatsioone, tuuakse Erlangi tänase päevani aktoritest rääkides tihti välja esimese ja tähtsaima näitena.

6.2. Scala Actor Framework (programmeerimiskeelele Scala)

Scala on uudne ja omapärane programmeerimiskeel, mis ühendab endas mitut programmeerimise paradigma: funktsionaalprogrammeerimist ja objekt-orienteeritud programmeerimist. Silma paistab Scala veel sellega, et jookseb (peamiselt) Java Virtual Machine'il ja on ühilduv Javaga, ehk eksisteerivaid Java teeke on võimalik Scalas otseselt kasutada. See loob tugevad eeldused Scala populaarsuse jätkuvaks tõusuks, mis annab tulevikuperspektiivi ka Scala aktorite-teegile Scala Actor Framework (SAF). Tähelepanu võiks juhtida just sellele, et kuna Scalas on olemas mitmeid funktsionaalprogrammeerimisest tuntud tehnikaid ja omadusi mis Javal puuduvad (nt. näidise-sobitus ja kõrgemat järku funktsioonid), siis sobib Scala aktorite realiseerimiseks tunduvalt paremini kui Java ise (kood tuleb selgem, loetavam ja lihtsam). Tänu ühilduvusele Javaga võikski tulevikus mõelda just suurte Java-süsteemide paralleelprogrammeerimist kasutavate moodulite realiseerimisele SAF-i abil.

Muus osas sarnaneb SAF küllaltki paljus Erlangile (autorid on seda eeskuju keele disainimisel ka tunnistanud). Nii nagu Erlangiski, eksisteerivad ka SAF-is eksplitsiitselt primitiivid sõnumite saatmiseks ja vastuvõtmiseks. Erindite leidliku kasutamise abil on leitud võimalused, standardset JVM baitkoodi muutmata realiseerida kergekaalulised aktorid. Scala süntaksi võib näha ka ülalpool toodud näidetes, seega ei hakata sellel siinkohal enam pikemalt peatuma.

6.3. Programmeerimiskeel SALSA

Nagu Erlang, nii on ka SALSA (*Simple Actor Language System and Architecture*) spetsiaalselt ühe valdkonna (ehk paralleelprogrammeerimise) ülesannete lahendamiseks mõeldud programmeerimiskeel (*domain-specific language*). SALSA lähtekood tõlgitakse spetsiaalse eeltöötleja (*preprocessor*) abil enne kompileerimist täiesti tavaliseks Java koodiks, mis on algsest SALSA koodist tunduvalt madala-tasemelisem, pikem ja seega programmeerijale käsitsi muutmiseks praktiliselt kõlbmatu. Saadud Java lähtekood kompileeritakse siis juba standardsete Java tööriistadega edasi tavaliseks JVM baitkoodiks ning käivitatakse JVM-il nagu iga teinegi Java programm. Palju kriitikat SALSAle on toonud naiivse-võitu implementatsioon, kus igale aktorile seatakse otseselt vastavusse üks JVM-i lõim, mis koos aktori deaktiveerimisega blokeeritakse (ühesõnaga SALSA aktorid ei ole kergekaalulised). Samuti ei ole paljudele meeltmööda viis, kuidas SALSA transleerib ühe programmeerimiskeele lähtekoodi ümber teise keele lähtekoodiks.

6.4. Vahendid Java-le

Pikka aega oli Javale tema küllaltki vähe-ekspressiivsete ja piiratud süntaktiliste võimaluste poolest korraliku aktorite-raamistike loomine suureks väljakutseks. Viimastel aastatel on siiski paar lahendust selle väljakutse ka vastu võtta suutnud.

Kilim on raamistik/tööriist, mis on kergekaaluliste aktorite implementeerimiseks võtnud ette üsna keeruka tee. Välja töötatud ja implementeeritud on keerukas tüübisüsteem aktorite poolt vahetatavate sõnumite tüüpimiseks. Selle tulemusena on väidetavalt saavutatu seni tuntud lahendustest kiireim aktorite-vaheline sõnumiedastus (nt. 3 korda kiirem kui Erlangis) [Srinivasan2008]. Lisaks Kilimi kooditeegi importimisele oma programmi, lisandub Kilimi kasutamisel tavapärasesse töövoogu veel üks samm: Java standard-kompilaatoriga baitkoodiks kompileeritud programm tuleb järel-töödelda spetsiaalse tööriista Kilim *weaver*'i abil. Just baitkoodi modifikatsioonide tõttu on paljud avaldanud kahtlusi Kilimi lähenemisnurga perspektiivikuses, eriti just potentsiaalsete ühilduvusprobleemide tõttu teiste Javale mõeldud tööriistadega.

Functional Java[4] on teine äärmiselt põnev projekt, mis üritab Java keele uuemaid võimalusi kasutades implementeerida erinevaid funktsionaalprogrammeerimisest tuntud võtteid. Ühe moodulina sisaldab see endas ka aktorite-raamistikku, mis implementeerib kergekaalulised aktorid Javale, kasutamata selleks baitkoodi manipulatsioone, nagu seda tegi Kilim. Samas ei näe kood, mida tuleb aktoritega programmeerimisel kirjutada, kummagi lahenduse puhul välja kaugeltki nii ilus ja selge, kui Erlangi või Scala Actor Frameworki puhul.

6.5. Vahendid teistes programmeerimiskeeltes

Aktorite raamistikke on viimastel aastatel hoogsalt juurde tekkinud, ja enamikele tänapäeval kasutatavatele programmeerimiskeeltele on praeguseks hetkeks vähemalt mingisugusel tasemel vahendid olemas. Võrreldes Java või C#-ga, on dünaamilisemates ja väljendusrikkamates keeltes nagu Python või Ruby kirjutatud aktorite raamistikud tunduvalt elegantsemad ja kasutajasõbralikumad. Aktoritega programmeerimiseks Python'is on olemas raamistikud Kamaelia[5], Eventlet[6] ja Parley[7]; Ruby'le implementeerivad aktorid raamistikud Dramatis[8] ja Revactor[9].

7. Praktilisi näiteid

Kõige triviaalsemaks näiteks olukorrast, kus paralleelprogrammeerimine kasulikuks osutub, on selliste arvutuslikult intensiivsete probleemide lahendamine, kus "pudelikaelaks" osutub mingi kergesti paralleliseeritav, kuid arvutuslikult kulukas tegevus, näiteks fikseeritud lahenditehulga pime läbiproovimine (*blind search*). Olgu meil näiteks kasutada arvutusklaster, mis koosneb kümnest arvutist, igasühes neist neli tuuma. Et tahame probleemi võimalikult kiiresti lahendada ja oma ressursse maksimaalselt ära kasutada, siis on selge, et peaksime looma vähemalt samapalju paralleelseid protsesse, kuipalju on meie käsutuses protsessorite tuumasid (40).

Vaatleme näiteks krüptoloogia vallast tuntud diskreetse logaritmi leidmise ülesannet: võimalike lahendite hulk on väga suur, kuid õige on neist vaid üks. Õige vastuse peale sattudes saame koheselt teada, et oleme lahendi leidnud, kuid proovimise teel selleni jõudmine võtab lihtsalt väga kaua aega. Aktoritega on taolise probleemi lahendamine väga lihtne: iga tuuma jaoks luuakse üks aktor, mille järel lepivad aktorid kokku, kes millist võimalike lahendite hulga osa proovima hakkab. Kui mõni aktoritest leiab õige lahendi, saadab ta koheselt kõigile teistele vastava teate, et vastus on leitud ja edasine otsimine mõttetu. Sellise stsenaariumi puhul tuleb ka silmas pidada, et selleks, et aktorid saaksid saabuvatele teadetele adekvaatselt reageerida, tuleb nende töö ära jaotada väiksemateks (näiteks mõne sekundi pikkusteks) etappideks. Pärast iga sellise etapi lõpetamist käib aktor kontrollimas oma kirjakasti, ja kui leiab sealt teate, et mõni teine

4 http://functionaljava.org/
5 http://www.kamaelia.org
6 http://wiki.secondlife.com/wiki/Eventlet
7 http://osl.cs.uiuc.edu/parley/
8 http://dramatis.mischance.net/
9 http://revactor.org/

aktoritest on ülesande juba lahendanud, siis uut arvutusetappi aktor enam ei käivita. (Kui aktor panna lihtsalt naiivselt lahendama arvutusülesannet, mille teostamine võtab tal aega näiteks kolm päeva, ja lasta sellel arvutusel enne järgmiste tegevuste teostamist lõpuni joosta, siis ei „näe“ aktor oma kirjakastis olevat teadet ja arvutab ikkagi edasi, kuigi see on mõttetu.)

Mõnevõrra keerukama näitena võib vaadelda sama arvutusklastri võimsuse ärakasutamist näiteks male või mõne teise *minimax* algoritmiga realiseeritava kahe mängijaga strateegiamängu mängimiseks [Russell2003]. Algoritmi idee on vaadata läbi teatud arv **n** käiku ette (otsinguruumiks on siin seega kõikvõimalikud legaalsed viisid teha järgmised **n** käiku), ja leida selline käik, mis teatud meetrikate alusel viib meid **n** käigu pärast kõige paremasse seisu, arvestades et vastane käib alati vastu oma parima võimaliku vastukäigu. Aktoreid tuleb siin taas luua vähemalt samapalju, kui on tuumasid. Erinevus on aga selles, et ükski aktor ei tea otsinguruumi oma osa läbivaatamise käigus, mida ta täpselt otsib: otsitavaks on selline käik, mille puhul meetrikate väärtus on globaalselt parim. Selleks, et olla kursis, milliseid senini parimaid väärtusi on teised aktorid enda otsinguruumi osadest selleks hetkeks juba leidnud, peavad aktorid teineteisega pidevalt sõnumeid vahetama. Aktorite suhtlemine reaalajas on äärmiselt oluline: kui teada igal hetkel senini parimaid lahendeid, saab algoritmi tunduvalt optimeerida, kasutades otsingupuu kärpimist mitte-lootustandvatest harudest (*alpha-beta pruning*).

Loomulikult on kõik sellised ülesanded lahendatavad ka teiste paralleelprogrammeerimise mudelite abil, kuid aktorite mudeli eksplitsiitne sõnumi-edastus ja sündmuse-põhine lähenemisviis muudavad selle siinkohal kõige loomulikumaks ning kergemini-mõistetavaks lahenduseks.

Lõpetuseks toome veel ühe elulise näite praktilise tarkvara-arenduse vallast. YAWS (ehk pikemalt Yet Another Web Server) on Erlangis kirjutatud veebiserver, mis kasutab erinevate klientide paralleelseks teenindamiseks Erlangi kergekaalulisi lõimi. Veebiserver peab tüüpiliselt suutma hallata suurt arvu paralleelseid kasutajasessioone. Spetsiaalselt korraldatud eksperimendis ([Armstrong2003]) võrreldi Apache'i veebiserveri ja YAWSi vastupanuvõimet *denial-of-service*-tüüpi rünnaku tingimustes. Nimelt käivitati 16-st arvutist koosnenud klastri ühes arvutis emb-kumb server, ja hakati ülejäänud arvutitest avama järjest uusi (kuigi väga aeglaseid) paralleelseid sessioone. Kui otseselt operatsioonisüsteemi lõimesid kasutanud Apache jooksis kinni juba umbes 4000 paralleelse sessiooni juures, siis sessioonide haldamiseks Erlangi kergekaalulisi aktoreid kasutanud YAWS suutis edukalt hakkama saada korraga kuni 80 000 paralleelse sessiooniga.

8. Kokkuvõte

Nii nagu on tihti juhtunud läbi kogu infotehnoloogia arengu, võib ka aktorite mudeli viimasel ajal lainena kasvanud populaarsust otseselt seostada trendide ja arengutega arvutisüsteemide riistvaras. Nii mitmetuumaliste protsessorite jõudmine mass-kasutusse kui kohtvõrkude ja avaliku interneti kiiruse ja kvaliteedi radikaalne tõus on avaldanud üha tugevamat survet leida lihtsamaid lahendusi paralleelprogrammeerimise realiseerimiseks. Nii nagu paljudes teisteski programmeerimise alam-valdkondades, nii nihkub tööjõu kallinedes ja riistvara üha kiiremaks ja odavamaks muutudes tasapisi ka paralleelprogrammeerimises optimaalne punkt selles suunas, et arendajate tööviljakuse nimel on mõistlik ohverdada pisut loodava tarkvara jõudlusest. Eelistatakse üha abstraktsemaid ja kõrgema-tasemelisemaid vahendeid, mis jäävad küll kiiruse poolest klassikalisematele võib-olla natukene alla, kuid suurendavad tuntavalt arendajate tööviljakust. Sellega võib osaliselt seletada ka aktorite mudeli üha laialdasemat kasutamist.

Enamus populaarsetest programmeerimiskeeltest pakuvad oma standard-teekides välja vahendeid paralleelprogrammeerimiseks lõimede ja jagatud mälu abil. See lahendus on küll mõnevõrra kiirem, kuid programmeerijale keerukas, veaohtlik ja teatud olukordades ka mitte-skaleeruv. Alternatiivse lahendusena on aastaks 2008 peaaegu kõigile keeltele kättesaadavad ka erinevad raamistikud, mis võimaldavad paralleel-programmeerimisele läheneda tunduvalt lihtsamal viisil ehk aktorite mudeli põhjal. Loodud on ka spetsiaalseid valdkonna-spetsiifilisi keeli, mis ongi ette nähtud just paralleelprogrammeerimise eesmärkideks (Erlang, SALSA).

Rusikareegel on see, et mida ekspressiivsem ja dünaamilisem on keel ja mida rohkem on selles erinevaid funktsionaalprogrammeerimisele tüüpilisi võimalusi, seda hõlpsam on selles ka implementeerida elegantseid ja efektiivseid vahendeid paralleelprogrammeerimiseks aktorite mudelil. Ilmselt just seetõttu on

ka rohkem raamistikke ja vahendeid loodud just moodsatele ülipaindlikele keeltele nagu Pyhton ja Ruby. Samas näiteks Java jaoks seevastu, mis eelpool-nimetatud omadustega kindlasti just ei hiilga, oli pikka aega korralik kergekaaluliste aktorite raamistik üleüldse puudu. Viimase paari aastaga on ka Javale mõned lahendused siiski tekkinud (Kilim, FunctionalJava), ja vahendite loend tõotab täieneda, kuna selge tendents on Java keele muutumisele iga uue versiooniga süntaktiliselt üha võimsamaks (oluline verstapost oli geneerilise programmeerimise konstruktsioonide (*generics*) lisamine Java viiendasse versiooni ning hetkel väljatöötamisel olevasse seitsmendasse versiooni lisatakse suure tõenäosusega juba ka sulundid).

Olles praegu veel enamuses väga uued ja osaliselt eksperimentaalsed, selguvad lähimate aastate jooksul tänaste või uute aktorite-raamistike seast ilmselt ka „võitjad“ selliste suurte ja ärisektorile oluliste programmeerimiskeelte jaoks nagu Java ja C#. Kindlasti on tänaseks eksisteerivatel lahendustel veel arenguruumi kasutusmugavuse, stabiilsuse ja jõudluse parandamise osas. Isegi kui mitmete lahendustega võiks üldjoontes rahule jääda, kulub mõningase inertsiga töötaval ärisektoril ilmselt veel mõni aasta aega, enne kui neid tehnoloogiaid korralikult kasutama õpitakse, usaldama hakatakse ja suurtes projektides rakendada julgetakse.

Viited

[Agha1985] Gul A. Agha. ACTORS - A Model of Concurrent Computation in Distributed Systems.

[Armstrong2003] Joe Armstrong. Concurrency Oriented Programming in Erlang.

[Haller2006-1] Philipp Haller. An Object-Oriented Programming Model for Event-Based Actors.

[Haller2006-2] Philipp Haller ja Martin Odersky. Event-Based Programming Without Inversion Of Control. Proceedings of JMCL '06, Oxford, August 2006.

[Haller2007] Philipp Haller ja Martin Odersky. Actors That Unify Threads And Events.

[Russell2003] Stuart J. Russell, Peter Norvig. Artificial Intelligence: A Modern Approach (2nd Edition).

[Srinivasan2008] Sriram Srinivasan, Alan Mycroft. Kilim: Isolation-Typed Actors for Java.