T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Infotehnoloogia eriala

Rauno Moisto

XML päringute paralleeltöötlus programmeerimiskeeles PHP

Bakalaureusetöö (6 EAP)

Juhendaja: Mait Kommusaar
Juhendaja: Vambola Leping

TARTU 2015

XML päringute töötlus programmeerimiskeeles PHP

Lühikokkuvõte:

Selle lõputöö eesmärk on kiirendada autootsingu päringut CRBMS [1] süsteemis paralleel-töötluse läbi. Enne konkreetsete probleemide lahendamist oli vaja leida parim töövahend. Selleks osutus ZeroMQ.

Võtmesõnad:

PHP, paralleeltöötlus, XML päringud, pthreads, Gearman, ZeroMQ

XML request parallel processing in PHP

Abstract:

The purpose of this thesis is to speed up vehicle requests in CRBMS [1]. This will be achieved by parallel processing. Before jumping to actually solving the problem the most suitable method for parallel processing was chosen. After some consideration it was decided to use ZeroMQ.

The figure (*Joonis 2*) illustrates the proposed solution the best. Where work is now sent out to ZeroMQ workers the work used to be done in the same process that handles receiving the replies. Processing replies from suppliers can take up to a few seconds each to process. The problem was that during the processing new replies arrived and had to wait for the previous process to complete.

However the solution presents various new problems such as possible memory leaks, maintaining a persistent database connection, timeouts, inter-process data flow and error handling.

Keywords:

PHP, parallel processing, XML requests, pthreads, Gearman, ZeroMQ

Sisukord

Sissejuhatus

Lõputöö eesmärk on kiirendada autopäringut CRBMS (*Car Rental Broker Management System*) [1] süsteemis XML päringute paralleeltöötluse läbi. Päringu sisuks on info selle kohta, kes, millal ja kus autot rentida soovib. Vastuseks on info sobivate autode kohta. Infot saab selle kohta, millised autod on saadaval ning milliste hindadega. Päringud saadetakse kõikidele autode pakkujaile juba praegu paralleelselt, nende vastused aga töödeldakse jadas. Täpsemalt punktis 1.1.

Lõputöö koosneb kahest osast:

- Paralleeltöötluse vahendite võrdlus, parima leidmine antud kontekstis.
- Praktiline osa, mille eesmärk on valitud vahendite abil päringute töötlust kiirendada.

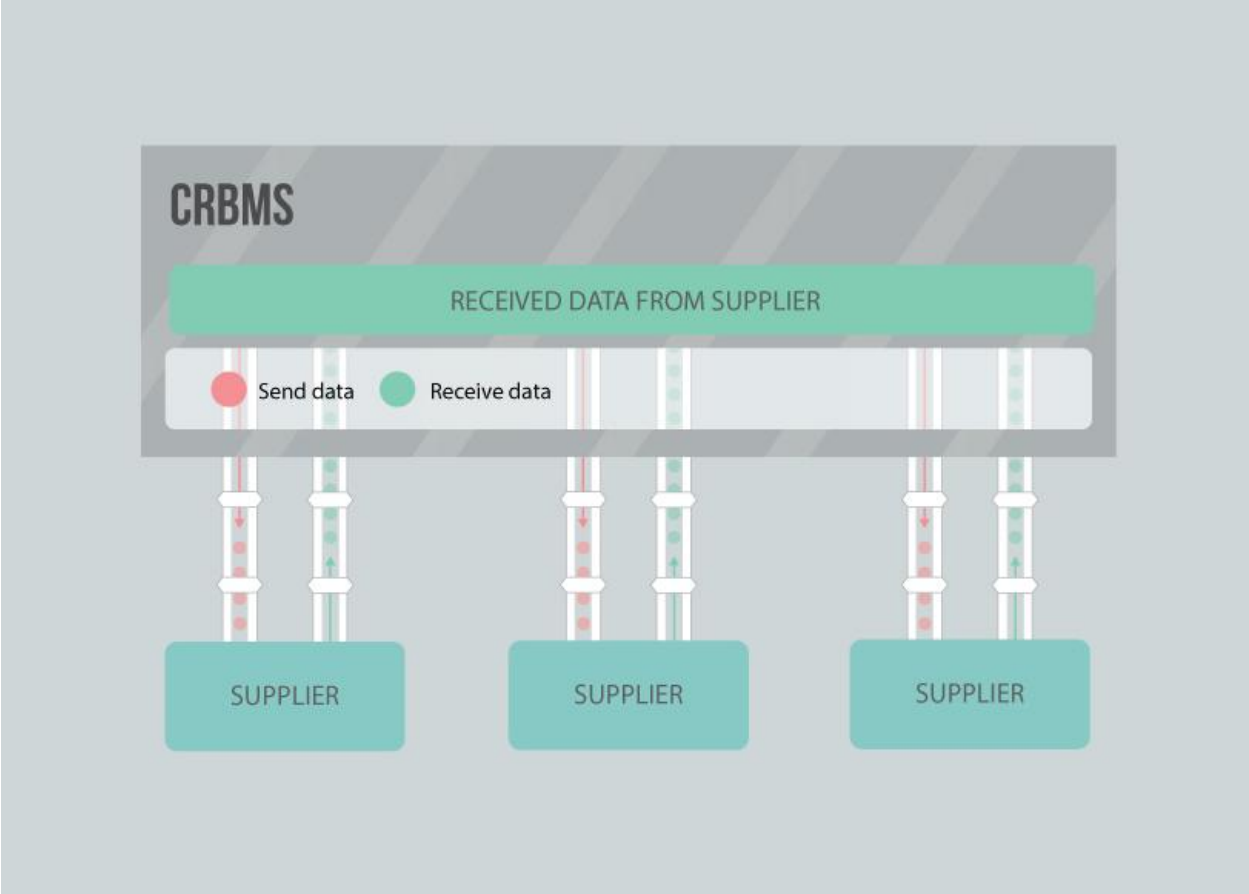
Vahendite võrdlusel peab muuhulgas arvestama, et vahend oleks ühilduv PHP programmeerimis-keelega ja et see oleks skaleeruv mitmele füüsilisele masinale.

Praktilise osa eesmärk on katsetada erinevaid paralleeltöötluse võimalusi ning hinnata ajavõitu efektiivsemast protsessoriaja kasutusest. Tulemustest selguvad konkreetsed soovitused CRBMS jaoks koos põhjendustega. Antud töö eesmärgiks ei ole otseselt CRBMS arendamine. CRBMS on keerukas süsteem, mille autopäring on üks selle keerukamaid komponente. Tulemused on tugevalt mõjutatud paljudest välistest faktoritest nagu vahemälu (*cache*) kasutus, hetke koormus testserverites, autode pakkujate (*supplier*) päringutele vastamise kiirus jms. Kuigi selle töö raames on testimise eesmärgil implementeeritud ka CRBMS juures kasutatav kood siis see vajab enne kasutusele võttu kindlasti täiendusi ja selle jõudlustulemused on eeltoodud põhjustel raskesti mõõdetavad.

1. Arhitektuur

1.1. Kasutusel olev arhitektuur

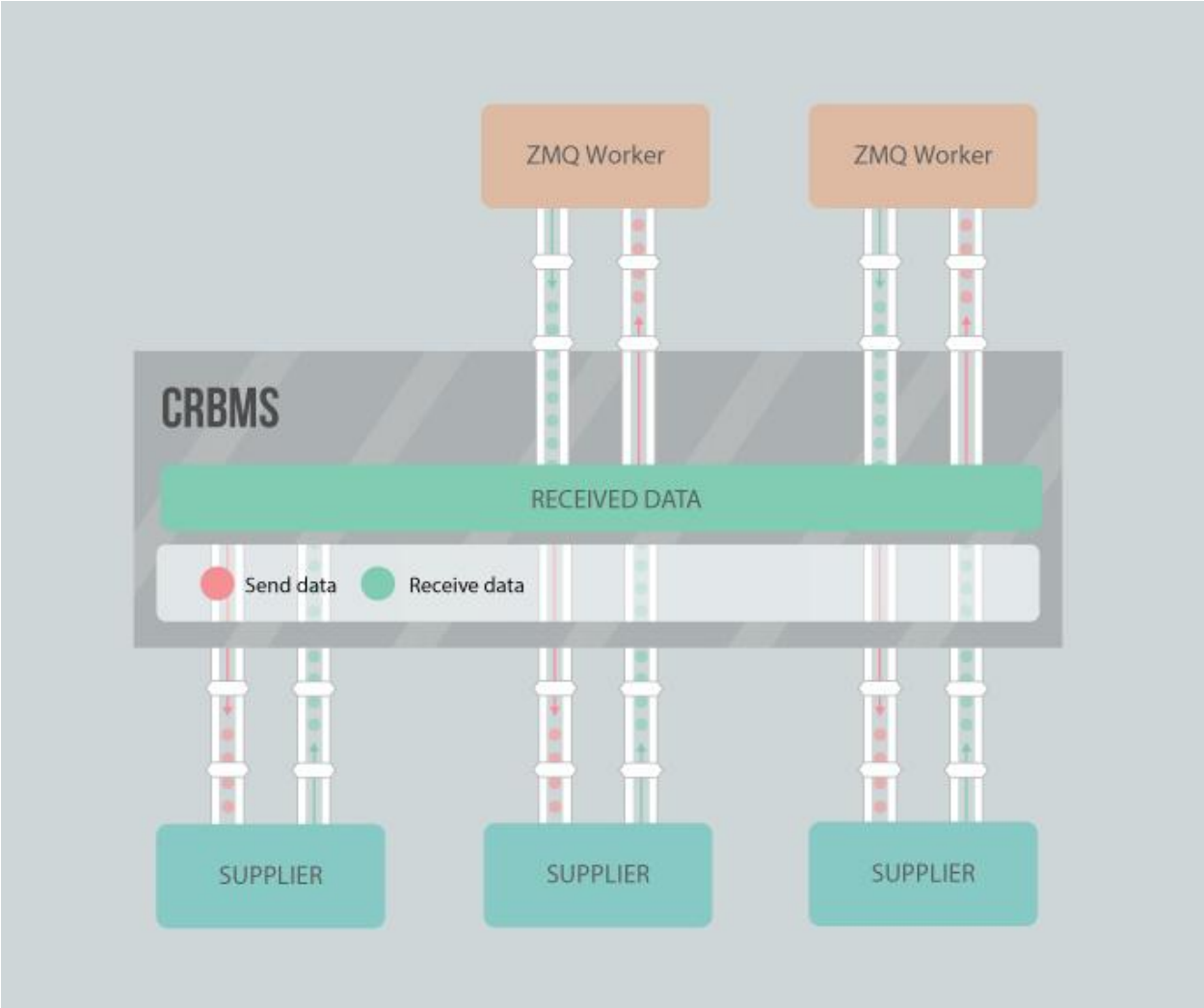
Hetkel kasutuses olevas arhitektuuris teeb veebibrauser päringu veebiserverile, mis koostab XML päringu ja saadab selle edasi CRBMS'le. Edasi koostab CRBMS sõltuvalt autode pakkujast kas päringu enda andmebaasi (*offline prices*) või XML päringu autode pakkuja juurde (*online prices*). Päringud saadetakse välja paralleelselt, kuid töödeldakse jadas. Veebiserver saab vastuse kätte siis, kui CRBMS on lõpetanud kõikide päringute töötluse ja koostanud vastuse. See protsess võib aega võtta äärmuslikel juhtudel üle poole minuti, mille jooksul võib klient arvata, et veebileht on katki ja ta lahkub.



Joonis 1: CRBMS praegu

1.2. CRBMS kihis paralleeltöötlus

Kõige loogilisem lahendus on muuta CRBMS kiiremaks. Üks viis selleks on autode pakkujatelt saabuvaid vastuseid töödelda paralleelselt, mitte jadas. Selle arhitektuuri implementeerimisest on juttu selle töö edasistes peatükkides. Antud implementatsiooni põhiprobleem on autode pakkujate pool päringutele vastamiseks kuluv aeg. Näiteks, kui pakkuja vastab CRBMS'le alles 30 sekundi pärast või ei vasta üldse, siis pole CRBMS'i pool kiirest vastuse töötlusest praktilist kasu. Täpsemalt punktis 4.2. Joonisel olev „ZMQ Worker" on selgitatud hiljem.



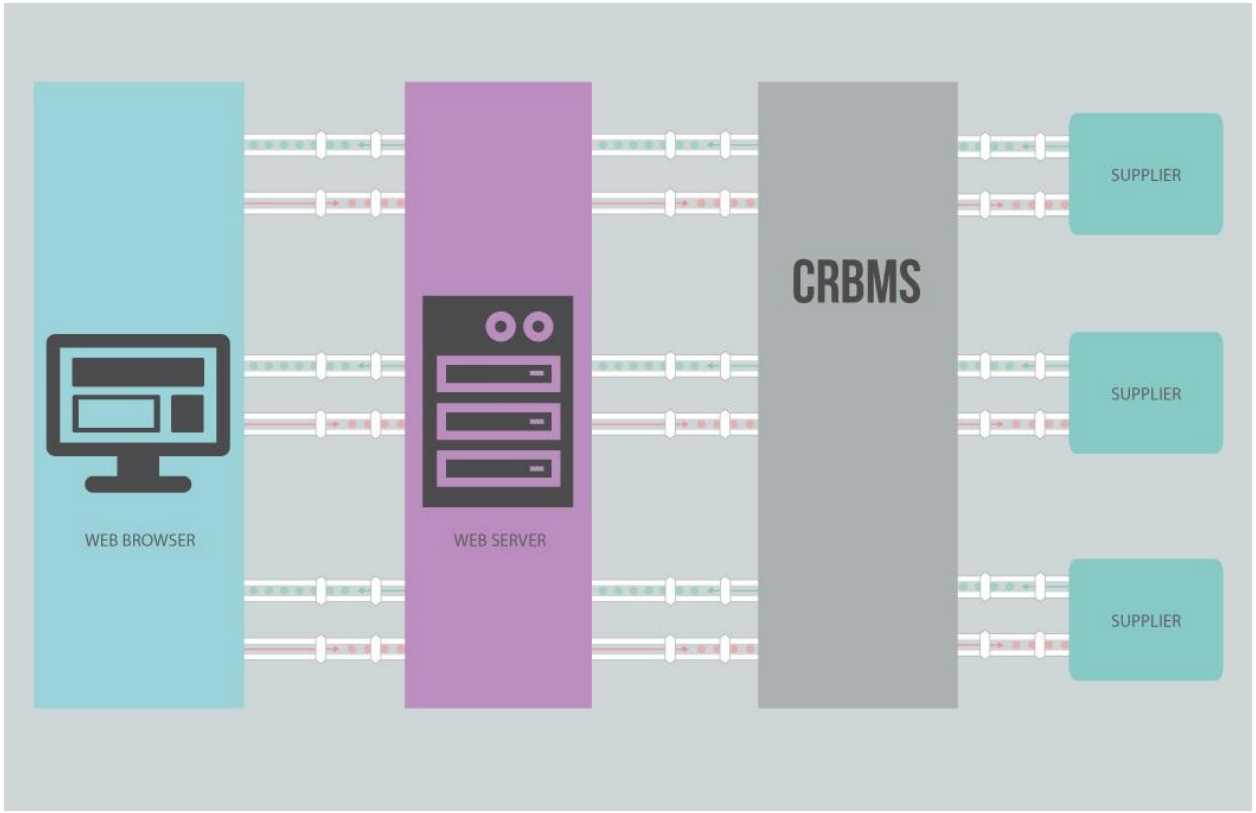
Joonis 2: CRBMS kihis paralleeltöötlus

1.3. Veebibrauseri kihis paralleeltöötlus

Kõige lihtsamini implementeeritav lahendus on iga autode pakkuja kohta teha CRBMS'le eraldi päring. Sellise lähenemisega on kogu autode otsingu protsess paralleelne juba veebibrauseri kihist alates, saates välja iga autode pakkuja kohta eraldi päringu. Iga päringu korral küsib CRBMS autosid ainult ühelt pakkujalt. Veebibrauser saab esimesed autod kätte peaaegu kohe peale esimeselt pakkujalt vastuse saamist.

See lähenemine lahendab esialgse probleemi täielikult, kuid sellega kaasneb uusi probleeme. Esiteks kulutab see kokkuvõttes oluliselt rohkem protsessoriaega, kui praegune lahendus. Nii veebibrauser, veebiserver kui ka CRBMS töötlevad ühe päringu asemel korraga kuni 40 päringut. Kuigi CRBMS'i vaatest on iga uus päring ligi 40 korda väiksem, kui varem, siis ikkagi kasvab koormus oluliselt. Eeldusel, et kasutusel olev riistvara ei ole ostetud suure reserviga, tuleb riistvara juurde osta. Teine probleem on statistika kogumine, sest praegu loetakse päringute arvu, arvestamata nende sisu. See tähendab, et varem kogutud statistika ei ole uuega võrreldav.

Antud lahendus on ühe kliendi vaatest implementeeritud, kuid ei ole kasutuses. Nimetatud implementatsiooni saab praegu kasutada ainult demo ja testimise eesmärkidel.

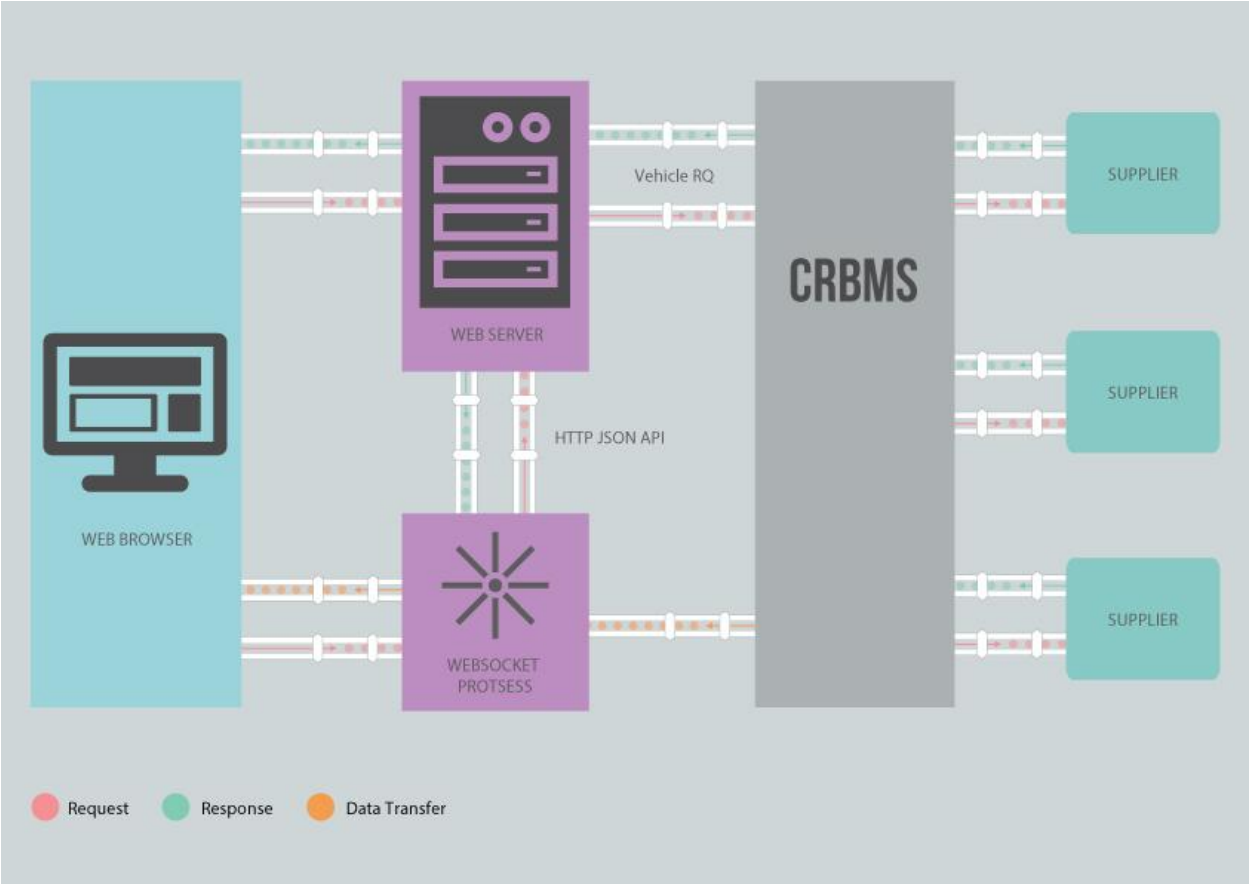


Joonis 3: paralleeltöötlus alates veebibrauserist

1.4. CRBMS ja veebiserveri kihis paralleeltöötulus

Kõige efektiivsem ja samas kõige keerulisem lahendus on kombineerida kaks eelnevat lahendust. Selles arhitektuuris töötleb CRBMS kogu autode otsingu päringut korraga ja töötleb autode pakkujate vastuseid paralleelselt, nagu punktis 1.2 kirjeldatud. Erinevus seisneb selles, et veebiserverile ei vastata enam alles siis, kui kõik vastused on olemas ja töödeldud. Selle asemel saadetakse vastus veebiserverile kohe, kui see on vastu võetud ja töödeldud, sõltumata teistest päringutest. See eeldab praegusest teistsugust veebiserveri implementatsiooni, sest vastuseid saadetakse ühe asemel mitu. Samuti on vaja need vastused edastada veebibrauserile. Kuna HTTP protokoll ei luba veebiserveril brauseriga ühendust luua, peab kasutama kas *WebSocket*'eid või peab brauser *pollima*. Selle võimaldamiseks on vaja veebiserveri kõrvale luua pidevalt käiv server.

See arhitektuur lahendab probleemi ja annab sama efekti, mis punktis 1.3 kirjeldatu, vältides seal tekkivaid probleeme. Ainus probleem on implementatsiooni keerukus. Sellise lahenduse valmis tegemine ja töökindluse tagamine oleks palju tööd, millest hetkel pole peaaegu midagi olemas.



Joonis 4: paralleeltöötlus

2. Paralleeltöötlusvahendite võrdlus

2.1. Lõimed (*Threads*)

Paralleeltöötlusest rääkides ei saa jätta mainimata selle kõige lihtsamat vormi - lõimed. Käivitades programmi erinevaid osi erinevates lõimedes on võimalik sama protsessi raames kasutada kõiki protsessoreid ja nende kõiki tuumasid korraga. PHP pthreads laiendusest (*extension*) on 2014. aastal avaldatud mitu uut versiooni. [2] Enne seda oli avaldatud ainult *beta* versioone, mida ei saa usaldusväärseteks pidada. See lahendus üksi pole piisav, sest ei skaleeru mitmele füüsilisele masinale.

2.2. Alamprotsessid (*Fork*)

Enne kasutatavate lõimede saadavale tulekut saavutati sama tulemus PHP alamprotsesside loomisega. Selleks on PHP funktsioon pcntl_fork. [3] Ka see lahendus ei skaleeru mitmele füüsilisele masinale. Prefiksiga pcntl PHP funktsioonid ei tööta Windows operatsioonisüsteemis.

2.3. Gearman

Gearman [4] lahendab eelnevate võimaluste suurima puudujäägi - see skaleerub mitmele serverile. Eristatakse töölisi (*worker*), tööservereid (*job server, message broker*) ja kliente (*client*). Tööserverid on protsessid, mis vahendavad kõiki töid. Kliendid tegelevad tööserveritele tööde saatmisega ning töölised tööserverilt tööde vastu võtmisega. Gearmani puudustena saab välja tuua Windowsi toe puudumise ning sõltuvuse tööserverist, mille kadumisel tööd teha ei saa (*single point of failure*).

2.4. ZeroMQ

ZeroMQ [5] (teiste nimedega zmq, 0MQ ja ØMQ) on selle töö kirjutamise ajal vähe tuntud paralleeltöötlust võimaldav vahend, mis lahendab kõik eelnevalt väljatoodud probleemid. ZeroMQ võimaldab suhtlust protsesside vahel väga madalatel tasemetel võimalikult madala latentsiga. Puudub keskne kontrollpunkt, kliendid ja töölised suhtlevad üksteisega otse. Info edastamiseks on ZeroMQ *socket*'id, mida saab kasutada nii protsessi piires (INPROC), serveri piires (IPC) kui ka serverite vahel (TCP). Erinevalt Gearmanist ja viimasele sarnanevatest vahenditest puudub protsess, mida käima panna. Selle asemel kasutatakse seda teegina (*software library*). ZeroMQ on vabavaraline, avatud lähtekoodiga ning kirjutatud programmeerimiskeeles C++. ZeroMQ on saadavatel väga paljudes keeltes, kuid baasteek on loodud C keele jaoks. Kõigi

teiste, sealhulgas PHP jaoks, on kirjutatud teegid (ZeroMQ nimetab neid *language binding*'teks) C teegi peale. [6]

2.5. Võrdlustabel

	Mitme protsessori kasutus	Mitme serveri kasutus	*Single point of failure*	Windowsi tugi
pthreads	jah	ei	ei	jah
fork	jah	ei	ei	ei*
Gearman	jah	jah	jah	ei
ZeroMQ	jah	jah	ei	jah

* uusi protsesse saab käivitada ka näiteks curl [7] abil http päringute tegemisega, kuid see pole optimaalne

Tabel 1: paralleeltöötlusvahendite võrdlus

3. Paralleeltöötuslus ZeroMQ abil

3.1. Server

Ülelihtsustatud PHP ZeroMQ server, mis trükib sellele saadetud andmed ekraanile:

```
<?php
$socket = new ZMQSocket(new ZMQContext(), ZMQ::SOCKET_PULL);
$socket->bind("tcp://127.0.0.1:5555");
while (true) {
    $res = $socket->recv();
    echo $res . PHP_EOL;
}
```

Selline server kuulab TCP porti 5555 ja ootab sisendit *revc* käsu juures. Kui sisendit pole siis programm ootab. [8]

3.2. Klient

Ülelihtsustatud PHP ZeroMQ klient, mis saadab sõnumi eelnevalt kirjeldatud serverile:

```
<?php
$socket = new ZMQSocket(new ZMQContext(), ZMQ::SOCKET_PUSH);
$socket->connect("tcp://127.0.0.1:5555");
$socket->send("Hello, world!");
```

Üks huvitav ja väga kasulik ZeroMQ omadus on, et kliendi võib käivitada ka enne kui serveri ja sõnum jõuab ikkagi kohale. Sõnumeid hoitakse järjekorras (*message queue*) kuni tekib sobiv server. Märkimisväärne on ka see, et teoreetiliselt pole mingit vahet kumb pool kutsub välja *bind* ja kumb *connect* käsu. Kuigi nende segi ajamisel süsteem üldjuhul töötab, on see siiski väär-kasutus ja seda tuleks vältida. [6]

3.3 Päringumustrid

Eelnev näide kasutab PUSH-PULL mustrit. Selline muster sobib ühesuunaliseks andmete liigutamiseks. Klient ei oota vastust ning server seda ei saada. Sellise mustri kasutamisel saadetakse andmed üldjuhul mitteblokeerivas režiimis, mis tähendab, et kui peale *send* käsu välja kutsumist järgneb veel koodi, käivitatakse see koheselt. Andmete saatmine toimub tagataustal teises ZeroMQ lõimes. [6]

Teine levinud muster on REQ-REP (lühendid sõnadest *request* ja *reply*). Selle mustri puhul saadab klient päringu serverile ning ootab vastuse ära, et vastusena saadud andmetega midagi

teha. See sobib ka siis kui serveri poolt saadetud andmed pole olulised vaid on vaja enne jätkamist ära oodata millal server töö lõpetab. [6]

ZeroMQ pakub ka teisi mustreid, näiteks PUB-SUB (lühendid sõnadest *publish* ja *subscribe*), kuid need pole antud kontekstis olulised. [6]

4. Tulemused

4.1 Ideaaljuht

4.1.1 Server

```php
<?php
if (!isset($argv[1])) {
    die("No dsn specified!" . PHP_EOL);
}
$socket = new ZMQSocket(new ZMQContext(), ZMQ::SOCKET_REP);
$socket->bind($argv[1]);
while (true) {
    $res = $socket->recv();

    /* Time to do some really intense work */
    $start = microtime(true);
    $rn = 2;
    while (true) {
        if ($rn < 1000000) {
            $rn = $rn * $rn;
        } else {
            $rn = $rn / 2;
        }
        if (microtime(true) - $start >= 1) {
            break; // stop after 1s
        }
    }
    $socket->send($res . " said " . $rn, ZMQ::MODE_DONTWAIT);
}
```

Server käivitatakse käsurealt, andes argumendiks DSN (*Data Source Name*) aadressi. DSN on näiteks “tcp://127.0.0.1:5603”. Sama serveri piires võib kasutada ka IPC (*Inter-process Communicaiton*) protokolli, kuid see pole saadaval Windows keskkonnas. Järgnevas kliendi näites kasutatakse 8 töölist. Näite kasutamiseks tuleks käivitada kõik 8 järgnevas näites toodud aadressidega serverit.

4.1.2 Klient

```
<?php
class MyMessage extends Thread {
    private $job;
    public function __construct($job) {
        $this->job = $job;
    }
    public function run() {
        $servers = array(
            "tcp://127.0.0.1:5603",
            "tcp://127.0.0.1:5604",
            "tcp://127.0.0.1:5605",
            "tcp://127.0.0.1:5606",
            "tcp://127.0.0.1:5607",
            "tcp://127.0.0.1:5608",
            "tcp://127.0.0.1:5609",
            "tcp://127.0.0.1:5610"
        );
        $socket = new ZMQSocket(new ZMQContext(), ZMQ::SOCKET_REQ);
        foreach($servers as $server) {
            $socket->connect($server);
        }
        $socket->send("Client" . $this->job);
        $message = $socket->recv();
        if ($message) {
            echo $message . PHP_EOL;
        }
    }
}
$start = microtime(true);
$jobs = array();
$messages = 16;
for ($i = 0; $i < $messages; $i++) {
    $jobs[$i] = new MyMessage($i);
    $jobs[$i]->start();
}
foreach ($jobs as $i => $job) {
    $res = $job->join();
}
echo "Time for " . $messages . " messages: " . (microtime(true) - $start) . "s" . PHP_EOL;
```

Selle kliendi võib käivitada nii käsurealt kui läbi veebiserveri. Enne käivitamist tuleb veenduda, et serverite nimekiri vastab reaalselt käivitatud serverite nimekirjale.

Antud näide võimaldab ideaaljuhul tehtava töö ajakulu vähendada kahekordselt protsessori tuumade arvu kahekordistamisega. Siin näites tegeleb server suvalise numbri arvutamisega. Ideaaltingimuste saavutamiseks kestab numbri arvutamine alati ühe sekundi. Konkreetses katses on kasutusel 8 töölist ja 16 tööd. Eeldusel, et arvutil, millel see näide käivitatakse, on vähemalt 8 protsessori tuuma, võtab see töö aega 2 sekundit. Kui tööliste või tuumade arv on väiksem, läheb aega kauem. Kui kasutusel on vaid üks tööline või üks tuum läheb aega 16 sekundit. Ajakulu ei

ole siiski tööde arvu ja tööliste või tuumade arvu suhe. 8 töölisega 17 töö tegemiseks läheb aega vähemalt 3 sekundit. Sellisel juhul saab üks tööline 3 tööd ja ülejäänud 2.

Näites on kasutusel käesoleva töö algul kirjeldatud lõimed (*pthreads*). Iga päringu jaoks on kasutusel oma lõim, et päringud üksteise järel ootama ei peaks. Lõimede kasutus pole paralleelsete päringute tegemiseks kohustuslik. Sama tulemuse saavutamiseks ilma lõimedeta tuleb kliendi poolel luua server, mis vastuseid kuulab, päringud kõik korraga ära saata ning seejärel loodud serveriga vastuseid ootama hakata. Selline lahendus tähendab ühe lisa-aadressi kasutamist.

Erinevaid tuumade, tööliste ja tööde arvude kombinatsioone katsetades on huvitav jälgida protsessori koormust tuumade lõikes. Selleks sobib Windows keskkonnas väga hästi Task Manager. Toodud näide on just sel põhjusel tahtlikult protsessorit koormav. Tüüpilistes näidetes kasutatakse *sleep* [9] funktsiooni töö simuleerimiseks, kuid see ei koorma protsessorit.

4.2 CRBMS

Joonisel 2 on kujutatud üks osa ühest autopäringust. Selles osas toimub autode pakkujatele hinnapäringute tegemine. Päringud saadetakse laiali paralleelselt, selle taha ootama ei jääda. Päringute vastu võtmine käib tsüklis nii kaua kuni kõik vastused on kohale jõudnud. Iga kohale jõudnud vastuse peale käivitatakse andmete töötlus. Probleem tekib siis, kui vastuseid on palju ja need saabuvad kõik samal ajal, sest enne järgmise vastuse vastu võtmist peab eelmise vastuse töötlemise ära ootama. Nimetatud töötlus on ajakulukas protsess, mis suurte päringute puhul võib aega võtta mitu sekundit.

Joonisel kujutatud lahenduse järgi ei töödelda vastuseid kohe vaid saadetakse koheselt edasi töölisele (“ZMQ Worker”). Töölisele edasi saatmine on väga kiire protsess ja koheselt on võimalik uusi vastuseid vastu võtta. Tööline töötleb vastuse ning vastab peaprotsessile, kui on valmis. Kui kõik töölised on lõpetanud, jätkub protsess nagu see seni on käinud ning algse autopäringu tegija saab vastuse.

Testsüsteemis, kus kogu päringu ajakulu on keskmiselt ligikaudu 7 sekundit, langes see ligikaudu 6 sekundini. Ajavõit tulenes otseselt sellest, et kahte korraga saabunud vastust töödeldi paralleelselt. Üks neist võttis aega sekundi, teine kaks korda kauem.

4.3 Probleemid

PHP protsessid on algselt loodud käivituma korraks, mitte igavesti käima. Pakutud lahenduse juures on vaja iga koodimuudatuse järel töölised kinni panna ning uuesti käivitada. Töölise protsess hoiab koodi mälus ja seda pole võimalik uuendada ilma töölise sulgemise ja uuesti käivitamiseta.

Töölised võivad töö ootamatult lõpetada kui peaks tekkima mingi tõsine probleem (PHP *Fatal error*).

Probleemne on ka ühendus andmebaasiga, mis MySQL puhul võib aeguda 8 tunniga. [10] Teoorias on võimalik seda kontrollida ning taastada, kuid praktikas on see tülikas ja ebausaldusväärne.

Lisaks võib keeruliste programmide käivitamisel esineda probleeme mälukasutusega (*garbage collection* ei rakendu õigesti).

Eelnimetatud probleemid lahendab pea täielikult see, kui tööline on maksimaalselt lihtne nagu punktis 3.1.1 toodud näide ning tegelik töötlus käivitatakse alamprotsessina. Selleks sobib hästi PHP *exec* [11] käsk. Ka siin tekib probleem - andmete ette andmine *exec* käsuga käivitatud alamprotsessile. Kogu XML stringi ette andmine käsureale pole mõeldav paljude erisümbolite kasutamise tõttu (sh jutumärgid, väiksem ja suurem kui märgid, ampersandid). Valikus on välise ühiskasutatava mälu kasutamine (näiteks Memcache, MySQL või Redis) ja andmete vahetu edastamine ZeroMQ abil. Viimase lahenduse implementeerimine on eelnevatest keerukam, kuid peaks andma eelise nii kiiruse, usaldusväärsuse kui ka kättesaadavuse osas.

Lisaks valmistavad probleeme erinevad aegumised (*timeout*). Vaikimisi ootab *recv* käsk igavesti, mis ei ole alati soovitav. Aegumiste haldust ZeroMQ otseselt ei paku, kuid seda saab ise teha. Selleks tuleb *recv* käsk käivitada mitteblokeerivas režiimis (*ZMQ::MODE_DONTWAIT*). Sel juhul tuleb käsu tagastusväärtust (*return value*) tsüklis kontrollida. Tsüklile kuluv aeg korrutatud tsüklite arvuga annabki aegumise aja.

Tööjaotus (*load balancing*) võib olla probleemne. Kui kliendi poolel välja kutsuda mitu *connect* käsku nagu punktis 3.1.2 siis alustatakse alati samast töölisest. Kui töölistele mitu päringut saata siis iga päringuga valitakse nimekirjast järgmine. Selliselt saab kõige enam koormust nimekirjas esimesena olev server. Üks võimalik lahendus on nimekiri igal pöördumisel suvaliselt järjestada. Selleks sobib PHP funktsioon *shuffle* [12].

Kokkuvõte

Antud töö raames uuritud vahendid on kõik väga huvipakkuvad ja neid tasub edasi uurida. Lisaks neile on paralleeltöötlusvahendeid veel. Siin välja toodud võimalused on neist silmapaistvaimad populaarsuse, lihtsuse või funktsionaalsuse poolest.

Lõputöö raames valmis lisaks välja toodule veel palju koodinäiteid. Muuhulgas erinevad katsetused lõimede ja Gearmaniga. Põhiosa tööle kulunud ajast kuluski erinevate koodinäidete välja mõtlemisele ja nende katsetamisele. Kuigi sellele on palju aega kulunud pole see aeg piisav. Isegi põhjalikult uuritud ZeroMQ võimaldab veel palju enamat kui selle töö raames vaadeldu.

Bakalaureusetööks ettenähtud piiratud aja tõttu ei ole lahendus CRBMS jaoks täielik ega lõplik. Näiteks on siin vaatluse alla võetud vaid üks osake autopäringust. Kindlasti leidub veel kohti, mida saaks paralleeltöötluse abil kiirendada. Arendustöö jätkub kindlasti peale selle töö lõppu.

Kasutatud materjalid

[1] Car Rental Broker Management System Overview - Car Rental Broker CRBMS

http://www.rentalbooking.co.uk/ - viimati vaadatud 12.05.2015

[2] PECL :: Package :: pthreads

http://pecl.php.net/package/pthreads - viimati vaadatud 12.05.2015

[3] PHP: pcntl_fork - Manual

http://php.net/manual/en/function.pcntl-fork.php - viimati vaadatud 12.05.2015

[4] gearman [Gearman Job Server]

http://gearman.org/ - viimati vaadatud 12.05.2015

[5] Code Connected - zeromq

http://zeromq.org/ - viimati vaadatud 12.05.2015

[6] ØMQ - The Guide

http://zguide.zeromq.org/page:all - viimati vaadatud 12.05.2015

[7] PHP: cURL - Manual

http://php.net/manual/en/book.curl.php - viimati vaadatud 12.05.2015

[8] PHP: 0MQ messaging - Manual

http://www.php.net/manual/en/book.zmq.php - viimati vaadatud 12.05.2015

[9] PHP: sleep - Manual

http://php.net/manual/en/function.sleep.php - viimati vaadatud 12.05.2015

[10] MySQL :: MySQL 5.0 Reference Manual :: 5.1.4 Server System Variables

https://dev.mysql.com/doc/refman/5.0/en/server-system-variables.html#sysvar_wait_timeout - viimati vaadatud 12.05.2015

[11] PHP: exec - Manual

http://php.net/manual/en/function.exec.php - viimati vaadatud 12.05.2015

[12] PHP: shuffle - Manual

http://php.net/manual/en/function.shuffle.php - viimati vaadatud 12.05.2015

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks.

Mina Rauno Moisto (sünnikuupäev: 29.04.1991)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose XML päringute paralleeltöötlus programmeerimiskeeles PHP, mille juhendajad on Mait Kommusaar ja Vambola Leping,

   1.1 reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

   1.2 üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2 olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3 kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, 13.05.2015