TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Tarkvarateaduse instituut

Raul Land 155250 IAPB

VÕRGULIIDESE PROGRAMMEERIMINE PÄRANDRAKENDUSELE

Bakalaureusetöö

Juhendaja: Martin Rebane
MSc

Tallinn 2018

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Raul Land

20.05.2018

Annotatsioon

Antud töö eesmärgiks oli ühendada ettevõttes loodav prototüüp juba olemasoleva pärandrakenduse funktsionaalsusega. Selle eesmärgi saavutamiseks otsiti sobivad vahendid, mis vastasid püstitatud nõuetele ja programmeeriti liidesed keeltes C++ ja Java. Töö käigus lisati vajaminevad raamistikud ja teegid projekti, et oleks võimalik pärandrakenduses võtta vastu HTTP päringuid ja töödelda JSON andmetüüpi, mille tugi keeles C++ vaikimisi puudub. Sõltuvalt tehtud valikutest tuli hiljem realiseerida suuremate andmete lugemine, marsruutimine ning väljaspool ASCII kooditabelist olevate sümbolite kodeerimine ja dekodeerimine keeles C++. Töö käigus valmis liides, mis on ettevõttes kasutusel ja vastab päringutele firma sees programmeeritud veebi- ja mobiilirakenduselt.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 25 leheküljel, 6 peatükki, 12 joonist, 3 tabelit.

Abstract

Implementing Network Interface for Legacy Application

The target of this thesis is to create a functioning communications link between a prototype and a legacy application in order to use functionality from the latter. The work was carried out in a company in cooperation with other co-workers to specify the exact needs for the network interface.

Project began with specifying the main requirements and based on that, proper frameworks and libraries were added in order to fulfil these requirements. This additional software included a lightweight HTTP server framework called Libmicrohttpd and a library to make JSON handling possible in C++ programming language, named JSON for modern C++. During the work, a HTTP server was programmed in language C++ and some additional work was done on the new prototype programmed in Java to make communications between the two programs possible. Due to limitations from the HTTP server framework there were problems with receiving and sending Unicode symbols and problems with transmitting larger HTTP requests. These problems were solved by adding extra functionality to the legacy application in C++. A table based routing was also implemented to satisfy the need of multiple endpoints for requests that connect to different legacy application logic. More specific HTTP server configuration was needed because the original application did not support multithreading and thus concurrent requests to the server caused an exception in the programs work. Inside the prototype application, capability to send HTTP requests was added in order to forward specific requests to the legacy application and to receive a result from it.

The finished network interface is functional and currently used in a demo environment where it serves requests from the new web application prototype and a mobile application.

The thesis is in Estonian and contains 25 pages of text, 6 chapters, 12 figures, 3 tables.

Lühendite ja mõistete sõnastik

API	*Application Programming Interface* rakendusliides, programmiliides, API-liides
ASCII	*American Standard Code for Information Interchange*, kooditabel inglise keele ja teiste klaviatuuril esinevate sümbolite esitamiseks digitaalsel kujul
daemon	„Tagaplaanil jooksev programm, mis teostab teatud ettemääratud operatsioone kindlate ajavahemike tagant või vastuseks mingitele sündmustele“ [1]
endpoint	Lõppsõlm võrgus, mis on ühendatud ainult ühe kindla protsessi või programmiga.
exception	Tõrge programmi töös
header-fail	C++ programmeerimiskeelele omane failitüüp
HTTP	*Hypertext Transfer Protocol*, protokoll andmevahetuseks internetis
IP aadress	Võrgus oleva seadme identifikaator
JSON	*JavaScript Object Notation,* formaat andmete hoidmiseks
linker	„Linkur, Mitut objekt- või laademoodulit täitmisprogrammiks ühendav programm“ [1]
pointer	„Programmeerimises muutuja, mis hoiab teise muutuja aadressi või muutujate massiivi algusaadressi“ [1].
port	Loogilise ühenduse lõpp-punkt võrkudes
socket	Süsteemisisene *endpoint* andmete saatmiseks ja lugemiseks võrgus
Unicode	*Unicode Worldwide Character Standard,* kooditabel erinevates keeltes kasutatavate sümbolite kodeerimiseks
UTF-8	*8-bit UCS/Unicode Transformation Format,* Unicode’i kooditabelis leiduvate märkide kodeerimismeetod
XML	*Extensible Markup Language*, märgistuskeel andmete struktureerimiseks

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

1.1 Taust

Bakalaureusetöö aluseks on projekt ettevõttes, mille eesmärgiks oli uuendada kindlustusfirmade poolt kasutatavat programmi. Olenemata programmi kõrgest vanusest on see aktiivses arenduses selleks, et programm oleks kooskõlas kohalike seaduste, maksusüsteemi ja kindlustusfirma enda nõuete ning poliitikaga. Samas pole võimalik programmi vanuse ja arendusvahendite tõttu moderniseerida kasutajaliidest, mis põhineb aegunud suletud lähtekoodiga raamistikul, mis oli küll uudne ja pakkus palju võimalusi omal ajal, kuid tänaseks on liiga vana ja ilma toeta. Samuti on esialgselt tegemist töölauaprogrammiga, mis töötab igal klienditeenindajal isiklikus arvutis ja töö lõpetamisel sünkroniseerib andmeid peaserveriga, mis lisab süsteemile keerukust ja suurendab ülalpidamiskulusid. Nendest probleemidest sündis põhiprojektist eraldiseisev prototüüpprojekt, mille peaeesmärgiks on töölauarakendusest teha serverirakendus ja viia kasutajaliides üle veebipõhiseks. Erinevalt originaalsest lahendusest, mis on platvormipõhine ja seega kasutatav ainult Windows operatsioonisüsteemides, on prototüüp kasutatav läbi veebilehitseja, mis võimaldab seda kasutada kõigis brauserit omavates seadmetes.

1.2 Lahendatav probleem

Uue eraldiseisva kasutajaliidese loomise tõttu tekkis vajadus andmevahetusele võrgurakenduse ja pärandrakenduse vahel. Algses programmis puudub tugi andmevahetuseks üle võrgu, seega otsustati implementeerida liidesed, et teha võimalikuks andmevahetus erinevate komponentide vahel ning samuti anda võimalus välistel programmidel seda liidest kasutada. See annaks kliendile soovi korral võimaluse kasutada meie tehtud uut kasutajaliidest ja uut API-t (*Application Programming Interface*) või luua täienisti uus kasutajaliides, mis põhineks meie API-l. Algselt oli plaan realiseerida andmevahetus *socketite* (süsteemisisene *endpoint* andmete saatmiseks ja lugemiseks võrgus) abil, kuid järeldati, et see oleks liiga madalal tasemel ja nõuaks palju

lisatööd, mis on juba teiste poolt loodud raamistikes tehtud. Seoses sellega otsustatigi liides programmeerida kasutades kolmanda osapoole loodud raamistikku. Töö eesmärgiks oli realiseerida API esialgsele programmile kasutades mõnda kolmanda osapoole raamistikku ja kasutada uut API-t, et saata andmeid demorakenduse ja pärandrakenduse vahel.

1.3 Metoodika

Lahendatav probleem on individuaalne, kuid realiseeritakse koostöös kasutajaliidese programmeerijatega. Töö käigus valitakse sobivad teegid ning raamistikud ja selle põhjal realiseeritakse toimiv lahendus. Kogu arendustöö toimus operatsioonisüsteemis Microsoft Windows 10 kasutades programme Visual Studio 2017 ja IntelliJ IDEA. Versioonihalduseks kasutasin Git-i. Prototüübi presentatsioonikiht on programmeeritud kasutades ReactJS JavaScript raamistikku ja Spring raamistikku. Originaalrakendus on programmeeritud keeltes C ja C++ ja andmeid hoitakse IBM DB2 andmebaasis. API Testimiseks kasutati programme Postman ja Apache JMeter.

1.4 Töö ülesehitus

Töö sisus antakse esmalt ülevaade üldisest projekti arhitektuurist. Sellele järgneb valitud raamistike ja teekide kirjeldused, mis valiti lahenduse programmeerimiseks ja liidese enda järkjärguline realisatsioon koos ettetulnud probleemide ülevaatega ning leitud lahendustega. Kokkuvõttes antakse hinnang tehtud tööle ja analüüsitakse, kas tehtud tööd saaks paremaks teha.

2 Süsteemi üldine kirjeldus

Programmeeritakse lahendus, kus kasutajaliides on eraldiseisev komponent, mis suhtleb originaalse pärandrakendusega läbi HTTP (*Hypertext Transfer Protocol*) päringute. Kasutades seda protokolli, toimub andmete sisestus süsteemi ja pärandrakenduse andmete kuvamine kasutajaliideses. Näitena sisestab kasutaja enda andmed ja valitud kindlustatavad objektid programmi, mille tulemusena salvestub see informatsioon pärandprogrammi andmebaasis ja tagastatakse kindlustuslepingu maksumus, mis kuvatakse kasutajaliideses (Joonis 1). Süsteem võimaldab luua uusi objekte, lugeda ja redigeerida olemasolevaid objekte.

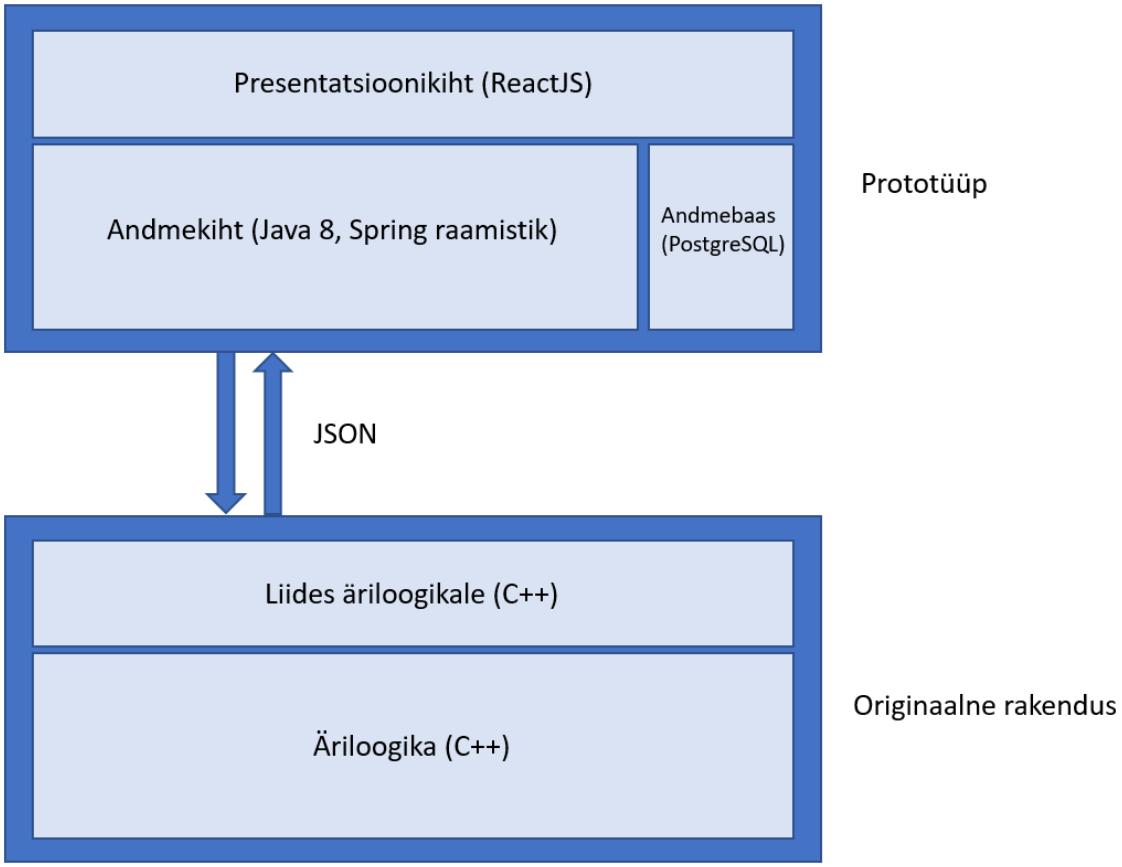
2.1 Realiseeritava prototüübi arhitektuur

Prototüübi kasutajaliides on eraldiseisev originaalsest andmekihist, mis asub pärandrakenduses. Prototüübil on ka endal andmekiht, mille poole pöördutakse, kui on vaja kasutada uut funktsionaalsust, mis ei ole otseselt seotud algse programmiga. Selline otsus tehti tingituna projektiga töötavate inimeste oskustest ja ka selle tõttu, et pärandrakendusele on aeganõudvam uut funktsionaalsust juurde lisada tema vanuse ning kasutatavate raamistike keerukuse tõttu. Samuti on programm aeglane ja ei ole programmeeritud pidades silmas paralleeltöö võimalusi. Algse rakenduse aeglusest ja mahust tingituna on eesmärk lisada juurde võimalikult vähe uusi raamistikke ja ebavajalikku funktsionaalsust, et mitte teha töökiirust veelgi aeglasemaks.

2.1.1 Projektis juba kasutusel olevad tehnoloogiad

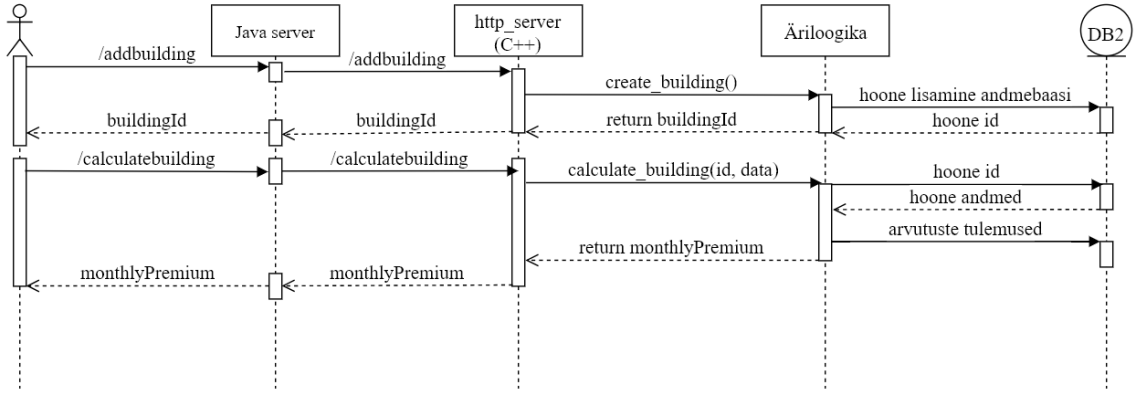
Uus kasutajaliides on programmeeritud kasutades React JavaScript raamistikku, millel puudub otsene ühendus pärandrakendusega. Kõik päringud presentatsioonikihist ja väliste teenuste poolt võetakse vastu Javas. Projekti raames lisatakse ka uut funktsionaalsust. Juhul kui see ei nõua väljakutseid pärandrakendusest või on osaliselt realiseeritav ka eraldiseisvas Java rakenduses, siis seda ka tehakse (Joonis 1). Võimalik oleks ka otse kasutajaliidesest saata päring pärandrakendusele rajatud võrguliidesele, kuid Javas ja Spring raamistikus on loodud head võimalused sissetulevate andmete

valideerimiseks, mistõttu tehakse kõik vajalikud sisestatud andmete validatsioonid prototüübi andmekihis. HTTP serverile, mis liidestab äriloogikat ülejäänud süsteemiga, saadetakse juba korrektne päring. Suunates kõik päringud Java rakendusele muutus lihtsamaks ka kogu prototüübi poolt kasutatavate *endpointide* dokumenteerimine.



Joonis 1. Prototüübi ja pärandrakenduse üldine arhitektuur.

Järgnevalt on toodud kasutusjuht, kus kindlustusprogrammi kasutaja loob uue kindlustusobjekti, näite puhul hoone, täidab andmed ehitise kohta ja valib ohud, mille eest kindlustada. Süsteem tagastab igakuise kindlustusmakse, mis kuvatakse presentatsioonikihis (Joonis 2). Sellel juhul on tegemist funktsionaalsusega, mis on ainult pärandrakenduses realiseeritud. Java rakenduse ülesanne on antud olukorras ainult päring edastada C++ programmile ja vastus tagastada kasutajaliidesele või päringu saatjale. Selline lähenemine lisab päringu saatmisele pikema viite, kuid võrreldes äriloogika töökiirusega on see viide marginaalne.



Joonis 2. Kasutusjuht - hoone lisamine ja hinnapakkumise arvutamine läbi HTTP päringute.

2.2 Äriloogika võrguliidese kirjeldus

Tulenevalt arhitektuurist on vaja andmeid saata üle võrgu erinevate komponentide vahel. Komponendid võivad olla ka samas serveris, kuid see ei pruugi sedasi alati olla. Liides põhineb HTTP protokollil, mis võimaldab saata päringuid üle võrgu. Valiti HTTP protokoll *socketite* otsese kasutamise asemel, sest see on laialdaselt kasutusel, mille tõttu on teistel programmidel lihtsam liidesega suhelda. Lisaks sellele on HTTP puhul tehtud ära palju eeltööd, mida *socketite* puhul tuleks ise programmeerida nagu näiteks kokkulepitud suhtlemise standard, mis on enamusele arendajatele üheselt mõistetav. Lähtuvalt firma plaanidest teeb valitud lahendus ka mobiilirakenduse ühendamise prototüübiga lihtsamaks.

2.2.1 Andmevahetuseks kasutatav andmetüüp

Andmete saatmiseks ja lugemiseks on kaks levinumat vormingut – XML (*Extensible Markup Language*) ja JSON (*JavaScript Object Notation).* Tehtud töös valiti andmetüübiks JSON järgmistel põhjustel:

- Presentatsioonikiht kasutab JavaScript'i, kus on andmed juba kapseldatud JSON-sse. XML-i kasutades peaks lisama juurde uue funktsionaalsuse, mis teisendab JSON-i XML-ks ja vastupidi.
- JSON on lühem ja tema kirjutamise ning lugemise jõudlus on parem [2]
- Isiklik eelistus

2.2.2 Nõuded liidesele

Kooskõlas projektijuhiga ning töö käigus täpsustati olulisemad nõuded, mida programmeeritud liides peab täitma ja millest lähtuvalt valiti ka raamistik ülesande lahendamiseks.

- Liides peab võimaldama HTTP põhiliste päringutüüpide – GET ja POST päringute vastuvõtmise.
- Programmeeritav lahendus peab toetama JSON andmetüübi töötlemist, sealhulgas teisendust sõnest vastavasse andmetüüpi ja vastupidi.
- Liides peab toetama mitut *endpointi* (lõppsõlm võrgus, mis on ühendatud ainult ühe kindla protsessi või programmiga), mille poole saavad kliendid pöörduda.
- Rootsi ja baltikumi sümbolite tugi ehk üldisemalt UTF-8 (*8-bit UCS/Unicode Transformation Format*) tugi.
- HTTP server ei tohi olemasolevat programmi märgatavalt aeglustada.
- Mitme järjestikuse päringu tekkimisel peab toimima järjekorrapõhine süsteem, kus vastatakse päringutele saabumise järjekorras.
- Peab võimaldama muuta serveri parameetreid.
- Kasutatav raamistik peab olema vabalt kasutatav ettevõtete poolt.

3 Raamistike ja teekide valikud

Tuleb valida sobivad raamistikud, mis vastaks seatud nõuetele. Põhilised probleemid, mida tuleb raamistike valikuga lahendada on HTTP protokolli käitlemine ja JSON andmetüübi lisamine C++ pärandrakendusse. Eeldati, et sisendi kontroll ja muu funktsionaalsus, mis pole otseselt seotud pärandrakenduse äriloogikaga nagu näiteks sisselogimine on juba realiseeritud kasutades JavaScript'i ja Spring raamistikku. Tänu sellele saab valida raamistikud, mis ei ole väga mahukad ja kus on olemas minimaalne funktsionaalsus, mis on vajalik ülesande täitmiseks.

3.1 HTTP server

Valikuid C++ serveri osas on mitmeid, iga üks neist omade eeliste ja puudustega. Kuna on võimalus, et prototüüp realiseeritakse tulevikus ka tervikliku lahendusena, siis omas valiku tegemisel tähtsust see, kui hästi on raamistik dokumenteeritud ja kui jätkusuutlik ta on. Teine nõue, mis tekkis otsingu ajal on seotud erinevustega C++ keeles Windows ja GNU/Linux operatsioonisüsteemidel. Leidus raamistikke, mille sisemised meetodid kasutasid ühe või teise operatsioonisüsteemi unikaalsete meetodite väljakutseid, mille tõttu nad ei toiminud mõne teise operatsioonisüsteemi peal. See tingimus vähendas valikuvõimalust veelgi, sest oli vaja raamistikku, mis töötab Windowsi operatsioonisüsteemis. Algselt kujunes üheks kandidaadiks Facebook-i poolt arendatav Proxygen HTTP server, kuid peagi selgus, et valiku tegemise hetkel ei toetanud see projekt Windowsi [3]. Leidus hulgaliselt väiksemaid projekte, kuid suur osa neist põhines Boost raamistikul või on selle olemasolu programmi jooksutamise eelduseks. Kuna Boost ise on väga mahukas, siis on selle kaasamine ülesande lahendamiseks ebaotstarbekas. Valikus oli ka Nghttp2 ja Libmicrohttpd. Nendest kahest otsustus viimane valituks, sest Nghttp2 käivitamiseks oli vajalik sertifikaatide olemasolu, millega tegelemine võtaks liiga palju lisaaega. Lisaks sellele täidab programmeeritav liides suhteliselt lihtsat ülesannet, mille tõttu on Nghttp2 pakutav HTTP/2 tugi tarbetu. Libmicrohttpd on HTTP/1.1 toega ja on mahult väike ning vajadusel on võimalik HTTPS läbi raamistiku häälestada [4].

3.2 JSON tugi C++ keeles

C++ puudub sisseehitatud JSON-i tugi, selle jaoks on vaja lisada kolmanda osapoole teek, et päringu kehast saadud JSON sõnumeid efektiivselt lugeda ja koostada. JSON-i kasutamiseks C++ programmeerimiskeeles on saadaval palju teeke, mille hulgast valisin võrdluseks kolm, mis otsingute põhjal olid populaarsemad. Testide põhjal on RapidJSON kõige kiirem ja võtab kõige vähem mälu (Tabel 1). Teised kaks teeki, PicoJSON ja JSON for modern C++ kasutavad samas suurusjärgus mälu, kuid töötlemises on PicoJSON kokkuvõttes aeglasem [5] [6] [7]. JSON-i teegi valikul oli tähtis ka UTF-8 kodeeringu tugi, kasutamislihtsus ja dokumenteeritus. Nendele kriteeriumitele vastas JSON for Modern C++, teek mis on aktiivses arenduses ja koosneb vaid ühest *header*-failist (C++ programmeerimiskeelele omane failitüüp). Lisaks omab see teek sisseehitatud meetodeid erinevate kontrollide ja võrdluste tegemiseks. Antud projekt on ka hea töökiirusega ning kasutab vähesel määral mälu võrreldes paljude teiste teekidega (Lisa 1) [8]. Kuigi tehtud valik ei ole kõige parema töökiirusega, on antud teek kasutusmugavuse arvelt parem võrreldud RapidJSON-st.

Tabel 1 Valitud JSON teekide omaduste võrdlus [8].

Teegi nimi	**Töökiirus sõne teisendamisel andmetüübiks (ms)**	**Mälukasutus (bait)**	**Objekti teisendus sõneks kiirus (ms)**
PicoJSON	140	9,739,632	80
JSON for Modern C++	72	9,897,872	88
RapidJSON	8	4,833,344	11

3.3 Kasutatud projektide litsentsid

Tegemist on ettevõtte jaoks programmeeritava projektiga, seega peab olema ka kasutataval tarkvaral litsents, mis lubab seda kasutada kasu saamise eesmärgil. Raamistik Libmicrohttpd on kaitstud GNU Lesser General Public License v2.1 (LGPL-2.1) litsentsiga ja JSON for Modern C++ MIT litsentsiga. Mõlemad litsentsid erinevad üksteisest kohustuste poolest, mida peab tegema tarkvara levitamisel, kuid kõige olulisemaks osaks on mõlema projekti puhul luba kasutada litsentsi all olevat tarkava kaubanduslikuks otstarbeks [9] [10]. Litsentside tingimustega ei minda vastuollu, sest mõlemat kasutatavat tarkvaraprojekti ei modifitseerita ja nad on ettevõtte loodud lähtekoodist eraldiseisvad.

Mõlemad valitud projektid pakuvad vajalikku funktsionaalsust ja litsentside põhjal võib neid kasutada ka kasumi teenimiseks. Sellega on kõik raamistikele esitatud nõudmised täidetud ja otsustasin antud raamistiku ja teegi projektis kasutusele võtta.

4 Võrguliidese programmeerimine

Liidese programmeerimine algas vajalike failide lisamisest projekti ja seejärel proovisin kompileerida ja jooksutada dokumentatsioonis toodud näiteid, et saada aimu üldisest ülesehitusest. Sellele järgnes järk-järgult funktsionaalsuse lisamine, et saavutada püstitatud eesmärgid. Testisin mõlemat nii JSON for modern C++ kui ka Libmicrohttpd raamistikku esialgu üksteisest eraldi, et välistada võimalikud probleemid, mis võiks tekkida kahe raamistiku ühendamisest ja et kiirendada vigade avastamise protsessi.

4.1 Esialgne demo HTTP serverist JSON toega

Pärast raamistike lähtekoodi importimist Visual Studiosse ja eraldi testimist alustasin esimese tervikliku demo programmeerimist. Selle eesmärgiks oli programmeerida minimaalne töötav server, mis suudab IP aadressile (võrgus oleva seadme identifikaator) ja pordile (loogilise ühenduse lõpp-punkt võrkudes) POST päringut tehes tagastada JSON teate, mis oli päringus kaasa pandud ehk tegemist oleks *echo serveriga*. Serveri konfigureerimisel kasutasin vaikimisi väärtuseid.

Olles juba teinud läbi näited Libmicrohttpd kodulehel, oli mul serveri käivitamiseks algne struktuur olemas. Selle põhilisteks osadeks olid *struct connection_info_struct* (Joonis 3), mis sisaldas päringus saadetud andmeid ja päringu tüüpi, funktsioonid *answer_to_connection* ja *on_complete* ning *daemon struct MHD_Daemon*,* mille abil on võimalik serverit käivitada. Selles arendusfaasis eeldasin, et kõik saadetud päringud on korrektselt vormistatud ja selle jaoks eraldi kontrolli ei teinud. Lähtudes sellest eeldusest sai koheselt *connection_info_struct* sees olevatele andmetele *const char* post_body* rakendada JSON for Modern C++ meetodit *json::parse,* mis tagastas juba JSON tüüpi objekti. Tagastame päringu saatjale kätte saadud vastuse, selleks on vaja JSON teisendada *const char** tüübiks, sest Libmicrohttpd sisene funktsioon *MHD_create_response_from_buffer* nõuab seda. Seda sai teha teisendades JSONI-i sõneks kasutades teegi meetodit *json::dump* ja selle tulemil keele spetsiifilist meetodit *c_str*, mis teisendab *std::string* tüüpi andmed *const char pointeriks* (Programmeerimises muutuja, mis hoiab teise muutuja aadressi või muutujate massiivi algusaadressi. [1]). Samuti lisasin vastuse päisesse rea „Content-Type: application/json“, et vastuvõtja saaks aru, mis vormingus andmetega on tegemist, sest teadsin, et liidest võib tulevikus hakata

kasutama ka mõni väline teenus. Antud rida päringu päises pole päringu tegemisel nõutud, kuid soovituslik [11].

```
#define MIMETYPE "application/json"
#define GET             0
#define POST            1

struct connection_info_struct
{
      int connection_type;
      std::string* post_body;
      json get_params;
};
```

Joonis 3. Sissetulevate andmete hoiustamise struktuur esialgses demos.

Testides töötavat demot oli näha, et puudus UTF-8 kodeeringu tugi ja suuremate JSON päringute saatmisel ebaõnnestus sõne teisendus JSON andmetüübiks.

4.2 Serveri struktuur

Pärast esialgset demot oli eesmärk struktureerida serveriosa loogiliselt, et see oleks kergesti kasutatav ka mujal ning arusaadav teistele arendajatele (Joonis 4). Selle tulemusena on ka server paremini eraldatud äriloogikast. Hetkeseisuga oli kõik kood ühes *.cpp* lähtekoodi failis koos *main* meetodiga.

4.2.1 *Header* failid C++ programmeerimiskeeles

Header failid on C++ omased failid laiendiga *.h*, *.hpp* või *.hxx.* Selleks, et mõista nende vajadust, on vaja mõista C++ kompilaatori üldist töökäiku – Alustatakse sellest, et kompilaator võtab kõik lähtekoodi failid ja kompileerib need ühekaupa *objekt* failideks, mis on taaskord C++ omane failitüüp. Selleks, et saada toimiv programm, ühendab kompilaator või siis spetsiaalne *linker* (mitut objekt- või laademoodulit täitmisprogrammiks ühendav programm) need *objekt* failid kokku terviklikuks programmiks. Seda protsessi nimetatakse linkimiseks. Kuna kõik failid kompileeritakse eraldi, ei tea üks lähtekoodi fail teise olemasolust [12]. Siin tuleb kasutusele *header* fail.

Header faile on vaja kolmel suuremal põhjusel:

- Nad aitavad organiseerida ja struktureerida programmikoodi.

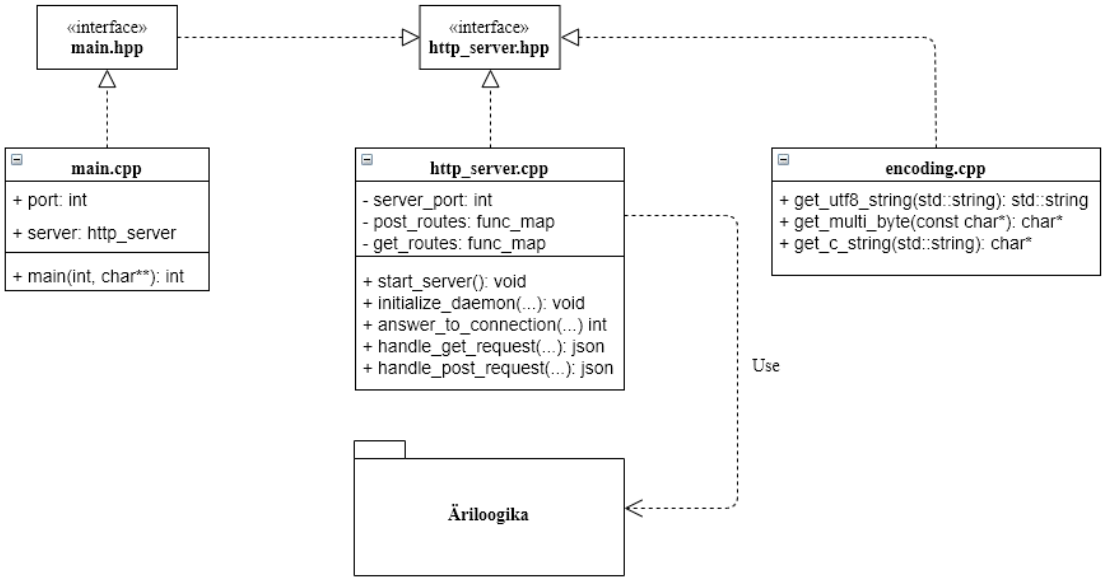
- Vähendab kompileerimisele kuluvat aega, sest hästi organiseeritud koodi, mis on jaotatud erinevatesse lähtekoodi failidesse, saab kompileerida valikuliselt ehk jätta kompileerimata failid, kus muudatusi ei ole tehtud. See ei mõjuta esmast kompileerimist, kuid arenduse käigus aitab aega säästa.

- Nendes failides saab defineerida liideseid, mis teeb võimalikuks implementatsioon hoida eraldi liidesest.

C++ keeles käib teistes failides asuvate klasside ja funktsioonide kasutamine läbi *#include* käskluste. Kui aga sedasi ühendada omavahel failid, kus on kasutusel samad *#include* failid, tekib olukord, kus kompilaator proovib sama lähtekoodi mitu korda kompileerida ja see ebaõnnestub, sest juba sama *objekt* fail on defineeritud. Ka siin on abiks sel juhul ühine *header* fail, mida mõlemad lähtekoodi failid kasutavad, mis välistab probleemi.

4.2.2 *Header* failide loomine projektis

Alustasin *header* failide loomisega, kus on defineeritud eraldi klass *http_server* koos *start_server* meetodiga. Esialgne lahendus oli klassi defineerimine otse .cpp failis, kuid kuna pärandprogramm ja server kasutasid samu importe, siis linkimisel tekkisid vead, sest *linker* proovis samu klasse korduvalt importida ja tekkis viga, et sama objekt juba on olemas. Selle tulemusena kompileerimine ebaõnnestus. Lahenduseks oli hierarhiline *header*-failide koostamine nõnda, et klassid, mis kasutasid samu importe, kasutasid ka sama *header*-faili nende ühiste importide jaoks. Samuti sai jagatud *header*-failidesse panna enda defineeritud andmetüüpide definitsioone, mida on hiljem vaja mitmes kohas, selle asemel, et need eraldi defineerida igas failis. Kogu tegevus vähendas koodi hulka, eemaldas kordused nii koodis kui ka importimisel ja selle läbi vähendab vigade tekkimise võimalust sellest, et midagi on erinevalt kahes kohas defineeritud. Hiljem lisandusid ka erinevad abistavad funktsioonid UTF-8 kodeeringu teisendamiseks, mille jaoks lõin eraldi komponendi *encoding.cpp*.



Joonis 4. HTTP serveri ja abifunktsioonide struktuur.

4.3 Suurema andmemahuga päringute töötlemine HTTP serveris

Algsest demost selgus, et saates suurema JSON-i, siis tekib programmi töös tõrge ehk *exception*. Koodi silumisel oli näha, et viga tekkis, kui jõuti töös nii kaugele, et oli vaja sisend teisendada sõnest JSON objektiks. Selle põhjuseks oli pooliku päringu sisu teisendamine, sest kogu sõnum ei olnud veel mällu loetud. Dokumentatsiooni kohaselt käivitatakse *answer_to_connection* meetodit korduvalt, kui on veel andmeid, mida sisse lugeda (Joonis 5). Sellele infole tuginedes muutsin funktsiooni, et päringu infot hoidev *struct* luuakse ainult esmasel väljakutsel ehk kui saabub uus päring, kui *con_cls pointer* on *nullptr*. Kuna *con_cls pointer* väärtustatakse päringu infoga, siis järgmistel funktsiooni väljakutsetel pole see enam *nullptr*. Pärast seda oli vaja muuta *post_body* andmetüüp selliseks, mida oleks võimalik suurendada järk-järgult, et oleks võimalik uut sissetulevaid andmeid hoiustada. Määratud suurusega andmetüüpi ei olnud otstarbekas valida, sest alati ei ole teada sissetulevate andmete suurust. Liidese kasutaja võib jätta määramata päringu päises andmete pikkuse ehk *Content-Length*, mille tõttu see lahendus ei toimiks [11]. *Const char* post_body* sai muudetud STL raamistiku andmetüübiks *std::string**. Teiseks valikuks oli ka *std::stringstream*, kuid see oli kiirel testimisel aeglasem. 100000 rea pikkune JSON suurusega 1019607 baiti võttis *std::stringi* kasutades 10.5 sekundit, aga *std::stringstreami* kasutades 11.5 sekundit. Tuvastamiseks, kas alustada JSON parsimisega, ehk kas kõik andmed on kohal, sai kasutada funktsiooni argumenti

upload_data_size, kui selle väärtus oli funktsiooni käivitamisel 0, olid kõik andmed kätte saadud.

```
static int answer_to_connection(
      void* cls,
      struct MHD_Connection* connection,
      const char* url,
      const char* method,
      const char* version,
      const char* upload_data,
      size_t* upload_data_size,
      void** con_cls
)
```

Joonis 5. Funktsiooni answer_to_connection argumendid.

4.4 UTF-8 kodeeringu tugi HTTP päringutes

Vaikimisi on serveris kasutusel *const char** andmetüüp, kus 1 bait = 1 *char*. Selle tõttu on võimalik kodeerida vaid 2^7 = 128 erinevat tähemärki ehk kõik ASCII (*American Standard Code for Information Interchange*) sümbolid. Probleem tekib kui tekst sisaldab mõnda ASCII välist sümbolit, nagu näiteks eesti või rootsi keelseid sümboleid. Neid sümboleid on võimalik kuvada UTF-8 kodeeringus. UTF-8 aga kasutab ühe sümboli kodeeringuks 1 kuni 4 erinevat baiti [13]. Serverisse tulevas *const char** baidijadas tekib olukord, kus on kõrvuti ASCII sümbolid, mis on 1 bait ja sümbolid, mis on pikkusega 1-4 baiti. Andmed saab edukalt edastatud, kuid need mitme baidi pikkused sümbolid muutuvad inimestele loetamatuks, sest programm kuvab neid 1 baidi kaupa. Samuti ei toimi sellisel tekstil tingimuslaused ja muu, mis hõlmab stringide võrdlust, kus on UTF-8 sümbolid. Kuna raamistik kasutab andmete saatmiseks ainult *const char** andmetüüpi, siis on vaja teha kaks teisendust: sissetulevad andmed on vaja teisendada UTF-8 *std::stringiks* ja väljaminevad andmed, mis on JSON-i andmetüübist teisendatud tagasi *std:stringiks*, on vaja teisendada UTF-8 vormingust taas *const char** tüübiks. Selle tarbeks lõin 2 uut funktsiooni – *get_utf8_string* ja *get_multi_byte*. Need funktsioonid kasutavad seesmiselt Microsoft Windowsile omaseid funktsioone *MultiByteToWideChar* ja *WideCharToMultiByte* (Joonis 6). Liidesest saadav baidijada tuli kõigepealt kokku koguda tervikuna, enne kui seda saab teisendada UTF-8 kodeeringusse varieeruva päringu pikkuse tõttu, sest mitme baidi pikkuste sümbolite puhul võis tekkida olukord, kus üks osa sümbolist oli ühes funktsiooni väljakutses ja ülejäänud sümbol teises

väljakutses. Pärast seda sai teisendada baidijada UTF-8 stringiks ja hoiustada *std::string* tüübis, mis toetab UTF-8 kodeeringut.

```
extern std::string get_utf8_string(std::string input)
{
	//uses WIN32 API
	int input_size = MultiByteToWideChar(CP_ACP, MB_COMPOSITE, 
input.c_str(), input.length(), nullptr, 0);
	std::wstring utf16_wstring(input_size, '\0');
	MultiByteToWideChar(CP_ACP, MB_COMPOSITE, input.c_str(), 
input.length(), &utf16_wstring[0], input_size);
	int utf8_size = WideCharToMultiByte(CP_UTF8, 0, 
utf16_wstring.c_str(), utf16_wstring.length(), nullptr, 0, nullptr, 
nullptr);
	std::string utf8_output(utf8_size, '\0');
	WideCharToMultiByte(CP_UTF8, 0, utf16_wstring.c_str(), 
utf16_wstring.length(), &utf8_output[0], utf8_size, nullptr, nullptr);
	return utf8_output;
}
```

Joonis 6. Funktsioon UTF-8 stringi teisendamiseks.

Tagasi teisendus *get_multi_byte* meetodis toimus vastupidiselt *get_utf8_string* meetodile, kuid samal põhimõttel.

4.5 GET ja POST päringute eraldi töötlemine

Liidese kasutatavas meetodis *answer_to_connection* on argumendiks *method*, mille väärtuseks on päringu tüüp. Selle põhjal lõin 2 erinevat funktsiooni – üks GET ja teine POST päringu jaoks – *handle_post_request* ja *handle_get_request*. Funktsioonide käivitused on tingimuslause sees, sest vaja oli ainult 2 tüüpi päringuid. Teine variant oleks kõik HTTP päringute tüübid lisada *map* andmestruktuuri koos vastavate funktsioonide *pointeritega*.

4.6 Mitme *endpointi* implementeerimine liideses

Aadress, kuhu päring tehakse on *answer_to_connection* funktsioonis argumendina kaasas *const char** kujul. Kui tegemist oli vaid mõne aadressiga, sobis nende marsruutimiseks ka tingimuslaused, kuid funktsionaalsuse kasvades muutuks kood halvasti loetavaks. Selle lahendamiseks lõin marsruutimise jaoks mõeldud tabeli. Siinkohal tekkis probleem, kuidas hoida funktsioone *hashmapis*. C++ programmeerimiskeeles on võimalik kätte saada funktsioonide asukohad mälus ja neid hoida *pointerites*. *Pointereid* on aga võimalik

juba tabelis hoida. Samuti on võimalik defineerida oma andmetüüpe [14]. Jõudsin lahenduseni kasutades neid kahte programmeerimiskeelele omaseid tunnuseid. Alustuseks defineerisin uue andmetüübi, mis koosneb funktsiooni mäluaadressist, mis tagastab JSON-i ja argumendiks on JSON. Defineerisin *std::unordered* mapi, kus võtmeks on *std::string* ja väärtuseks eelpool mainitud funktsiooni *pointer* (Joonis 7).

```
using pfunc = nlohmann::json(*)(nlohmann::json);
using func_map = std::unordered_map<std::string, pfunc>;
```

Joonis 7. Funktsiooni *pointeri* ja funktsioonide tabeli definitsioonid.

Pärast seda oli võimalik luua funktsioonide tabel, kus igale *endpointile* oli vastavusse viidud üks funktsioon (Joonis 8). Selle abil sai sissetulevate päringute sihtaadressi otsida tabelist, leida talle vastav funktsioon ja see käivitada. Hiljem tegin ka teise tabeli, üks POST päringute ja teine GET päringute jaoks. Praeguses lahenduses on antud võimalus rakendamata, kuid selline realisatsioon teeb võimalikuks samast *endpointist* saada erinev vastus olenevalt sellest, kas tegemist on POST või GET päring. Valitud sai *std::unordered map*, sest tegemist on *hashmapiga*, kus iga võtme otsinguks kulub konstantne võrdne aeg.

```
func_map post_routes = {
      {"/handshake", &handshake_func},
      {"/new", &echo_func},
      {"/addBuilding", &echo_func},
      {"/calculateBuilding", &echo_func},
      {"/deleteBuilding", &echo_func}
};
```

Joonis 8. Marsruutimistabel.

4.7 Päringute töötlemine järjekorra alusel

Projekti algusest oli juba teada, et pärandrakendus on loodud eesmärgiga toimida vaid töölauarakendusena. Sellest lähtuvalt oli ka programm loodud ja puudusid paralleeltöö võimalused. Teiste programmeerijate katsetest selgus, et eeldus on tõene ja programmi töö on häiritud, kui kasutada paralleeltöö metoodikat. Lisaks sellele ei paistnud olevat kiiret lahendust probleemi lahendamiseks terve programmi suure koodimahu tõttu. Tõrgete vältimiseks programmi töös ja andmebaasis tuli läheneda ka liidese programmeerimisele selliselt, et liides suhtleks pärandrakendusega sünkroniseeritult. Dokumentatsioonist selgus, et raamistik toetab kohandamist paralleeltöö osas.

4.7.1 Päringute töötlemise võimalused raamistikus

Raamistik Libmicrohttpd toetab nelja päringute töötlemise mudelit, see on defineeritav *daemonis* („tagaplaanil jooksev programm, mis teostab teatud ettemääratud operatsioone kindlate ajavahemike tagant või vastuseks mingitele sündmustele“ [1]) läbi argumentide:

- MHD_USE_THREAD_PER_CONNECTION – Peamine lõim, mis kuulab *socketit*, vajaduse põhiselt loob juurde uusi lõimesid. Üks lõim tegeleb vaid ühe päringuga.
- MHD_USE_SELECT_INTERNALLY – Üks lõim, mis kuulab *socketit*, saades sealt päringu asub seda täitma.
- MHD_USE_SELECT_INTERNALLY + MHD_OPTION_THREAD_POOL_SIZE – Lõimude kogum, kus iga lõim töötleb paralleelselt teiste lõimudega ühte või enamat päringut
- Ilma argumendita – Lõimud puuduvad

Igal lahendusel on omad eelised ja puudused, millist kasutada sõltub programmeeritava rakenduse omadustest.

MHD_USE_THREAD_PER_CONNECTION puhul luuakse iga päringu jaoks uus lõim. See toetab hästi paralleeltööd, kuid sünkroniseerimisprobleemid on kiired tekkima ja need tekkisid ka minu rakenduses, kui seda varianti katsetasin.

MHD_USE_SELECT_INTERNALLY puhul on kasutusel vaid üks lõim ja sellega programmi töös tõrkeid ei teki. Programmeerides tekkisid kahtlused selles osas, kas väga väikeste vahedega tehtud päringud jäävad ootele või lähevad kaduma, kuid testimine näitas, et kõik päringud olid töödeldud ja tagastatud nende saatmise järjekorras. Järjekord kui selline ei tekkinud C++ serveris, vaid hoopis Java Spring raamistikus, kus RestTemplate pani päringud järjekorda ootele, kui C++ serverist vastust koheselt ei tulnud.

MHD_USE_SELECT_INTERNALLY oli võimalik kasutada ka koos lõimude koguga andes *daemonile* lisaks ka argumendi MHD_OPTION_THREAD_POOL_SIZE, kuid see omakorda tekitas tõrkeid programmi töös mitme lõimu samaaegse töö tõttu. Viimane

valik ehk ilma argumendita mudel oleks nõudnud tsükli programmeerimist, mis ise perioodiliselt kutsub välja raamistiku meetodeid andmete lugemiseks *socketist*.

Kõik nimetatud viisid oleksid realiseeritavad, et lahendada probleeme projektis, kus puudub paralleeltöö võimalus, kuid selle saavutamiseks kuluks erinevaid argumente kasutades varieeruv hulk aega ja koodi, et kõrvaldada paralleeltööga tulenevad tõrked. Näitena saaks kasutada ka MHD_USE_THREAD_PER_CONNECTION mudelit, kuid see tähendaks koodi juurde kirjutamist ja asjatut ajakulu selleks, et takistada lõimude paralleelset tööd. Selle tulemusena käituks see lahendus üldjoontes sarnaselt MHD_USE_SELECT_INTERNALLY mudeliga, mille tõttu on otstarbetu kasutada esimest mudelit. Sama loogikat saab rakendada ka teisele mitme lõimuga mudelile.

Lähtudes nõuetest ja hinnates tehtava töö mahtu valisin mudeliks MHD_USE_SELECT_INTERNALLY, sest see täitis püstitatud nõuded ja vajas kõige vähem uut koodi, sest peamine loogika oli lahendatud raamistiku poole pealt.

4.8 Käivitades argumentide andmine serverile

Programmi käivitamisel saab anda kaasa parameetreid, mis on kättesaadavad arendajale *main* meetodis massiivina. Kui *port* puudub parameetrite hulgast, käivitatakse liides vaikimisi pordil 8090, vastasel juhul kasutatakse kasutaja poolt sisestatud parameetrit. Pordi muutmine on vajalik juhul kui vaikimisi *port* on juba kasutusel mõne teise programmi või protsessi poolt. Serveris on käivitatud programmeeritud lahendusest mitu protsessi, millest tekkis ka vajadus regulaarselt *porti* muuta. Sisendi lugemisel võtsin arvesse asjaolu, et kõik pordid pole meile vabalt kasutada ja seega lõin funktsionaalsuse, et kui argumendina antud *port* pole kindlas vahemikus, edastatakse teade konsooli veast ja seejärel käivitatakse server vaikimisi pordil. Selleks vahemikuks valisin pordid 8000 – 65535, mis peaksid enamusel juhtudel vabad olema.

HTTP serveri raamistiku poole pealt on paljud asjad kohandatavad ja neid saaks lisada käivitades argumentide hulka, samas leidsin, et suurem osa neist parameetritest ei vaja sageli muutmist ja sellega jäi lõplikus versioonis alles ainult võimalus *porti* muuta.

4.9 Java rakendusest päringute tegemine C++ serverisse

Lisaks C++ serverile tuli ka programmeerida Java rakenduses lahendus, mis vajadusel saadaks päringud edasi HTTP serverile pärandrakenduses. Java rakendus oli loodud kasutades Spring Boot malli. Tänu sellele oli lihtne luua uusi kontrollereid päringute vastuvõtmiseks Java rakenduses (Joonis 9).

```
@RequestMapping(value = "/addbuilding", method =
RequestMethod.POST, produces = "application/json")
public Optional<String> addBuilding(HttpEntity<String>
httpEntity) {
        return RestUtility.forwardRequest("/addbuilding",
httpEntity, HttpMethod.POST);
    }
```

Joonis 9. Päringute käitlemine Spring raamistikus.

Programmeerisin *RestUtility* klassi, mis hoiab infot C++ serveri pordi ja IP aadressi kohta ning sisaldab meetodit *forwardRequest* (Joonis 10). Selle ülesandeks on edastada päringud C++ serverile ja saadud vastus tagastada kontrollerile kasutades Spring-i *RestTemplate* klassi [15]. Valisin *RestTemplate*, sest see oli juba kaasatud projekti koos Spring raamistikuga ja näidete põhjal oli näha, et nõutav koodimaht on väike, sest enamus päringu saatmiseks vajalik on *RestTemplate* sees ära tehtud.

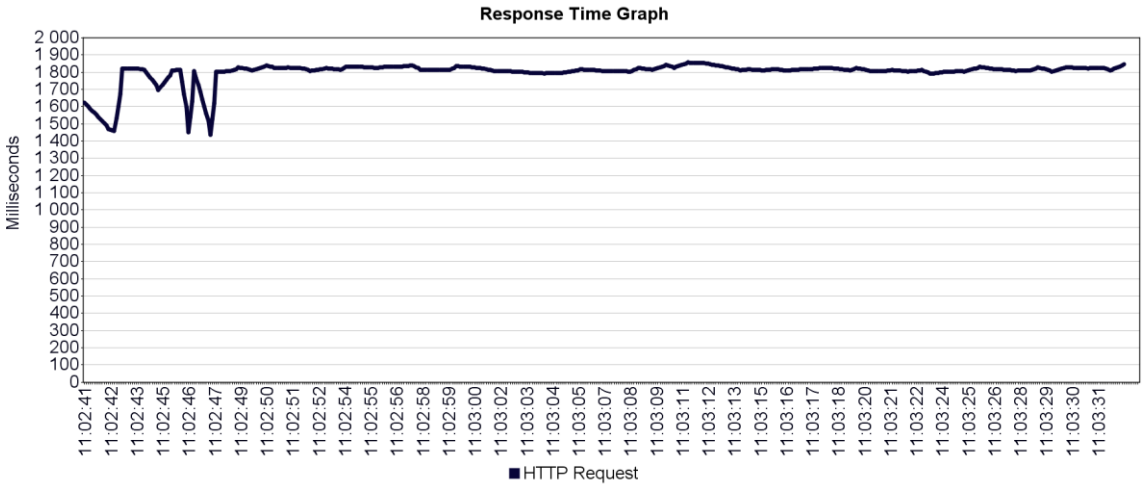
```
public static Optional<String> forwardRequest(String endpoint,
HttpEntity<String> httpEntity, HttpMethod method) {
        HttpComponentsClientHttpRequestFactory
httpRequestFactory = new
HttpComponentsClientHttpRequestFactory();
        final String url = YT_BASE_URL + serverPort + endpoint;
        RestTemplate restTemplate = new
RestTemplate(httpRequestFactory);
        restTemplate.getMessageConverters().add(0, new
StringHttpMessageConverter(Charset.forName("UTF-8")));
        try {
            ResponseEntity<String> response =
restTemplate.exchange(url, method, httpEntity, String.class);
            if (response.getStatusCode() == HttpStatus.OK) {
                return Optional.of(response.getBody());
            } else if (response.getStatusCode() ==
HttpStatus.UNAUTHORIZED) {
                throw new Exception("Unauthorized access");
            }
        } catch (Exception e) {
            return Optional.of(e.getMessage());
        }
        return Optional.empty();
    }
```

Joonis 10. Päringu edastamine C++ serverile ja vastuse tagastamine.

4.10 HTTP serveri testimine

Liidese esialgne testimine toimus Postman rakendusega. Kasutasin seda üksikute päringute tegemiseks, et testida HTTP serveri *endpointide* toimimist, UTF-8 kodeeringu ja suuremate päringute töötlemise tuge. Äriloogika testimine tehti samuti Postman rakenduses. *Echo* serveri testimiseks kasutasin Apache JMeter rakendust, sest see võimaldab kuvada testide tulemusi graafiliselt. Samuti on võimalik samal ajal teha mitu päringut korraga, et testida serveri vastupidavust ja kiirust koormuse all.

Selleks, et testida HTTP serveri võimet toime tulla suurema koormusega, testisin Apache JMeteris olukorda, kus 5 klienti teevad paralleelselt päringuid HTTP serverile. Määrasin seadetest, et 5 klienti teeksid 60 sekundi jooksul igaüks 30 päringut (Joonis 11). Testisin *echo* serverit, jättes välja äriloogika, et tulemused peegeldaksid liidese enda võimet, mitte pärandrakenduse töökiirust. Testitava päringu suuruseks võtsin 10000 rida pika JSON-i.



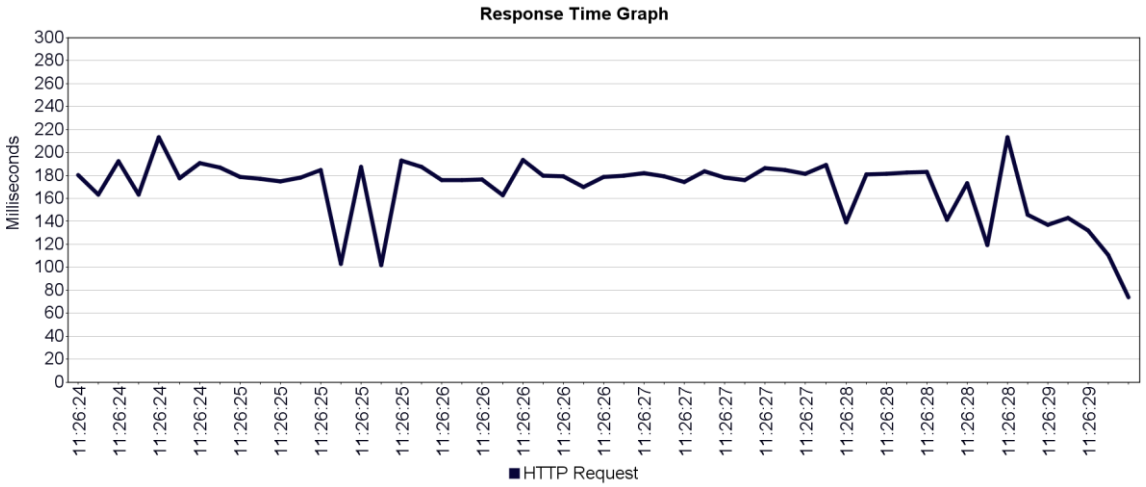
Joonis 11. JMeter testi tulemused JSON-i pikkusega 10000 rida.

Testi tulemustest saab järeldada, et märkimisväärse koormuse korral tuleb server kõikide päringutega toime ja töötab stabiilselt ilma suuremate kõikumisteta vastamiskiiruses. Tabelis 2 on toodud täpsemad testi tulemused.

Tabel 2. JMeter testi tulemuste detailandmed JSON-i pikkusega 10000 rida.

Päringute arv	150
Keskmine vastuse aeg (ms)	1798
Väikseim vastuse aeg (ms)	1109
Suurim vastuse aeg (ms)	1856
Standardhälve (ms)	91
Läbilaskevõime (päringut minutis)	165

Päringu tegemiseks kuluv aeg on sõltuv päringu sisu pikkusest, sest JSON-i suurus mõjutab selle teisendamiseks kuluvat aega ja lisaks sellele tuleb terve sõne mitu korda läbi töödelda, et seda oleks võimalik teisendada UTF-8 sõneks. Saadetav vastus tuleb samuti tagasi teisendada, mis nõuab aega. Selle väite kinnituseks tegin järgmise testi, kus tehakse sama pikal perioodil sama palju päringuid, kuid saadetav JSON on 1000 rida pikk (Joonis 12).



Joonis 12. JMeter testi tulemused JSON-i pikkusega 1000 rida.

Testi tulemused vastavad ootustele, sest vähendades JSON-i pikkust 10 korda, vähenes ka keskmine päringule vastamise aeg sama palju kordi ja läbilaskevõime suurenes 10 korda (Tabel 3).

Tabel 3. JMeter testi detailandmed JSON-i pikkusega 1000 rida.

Päringute arv	150
Keskmine vastuse aeg (ms)	170
Väikseim vastuse aeg (ms)	36
Suurim vastuse aeg (ms)	292
Standardhälve (ms)	45
Läbilaskevõime (päringut minutis)	1669

Test viidi läbi arendusmasinal protsessoriga Intel Core i7-4510U @ 2,6 GHz. Antud testi tulemused sõltuvad arvuti või serveri kiirusest, seega ei saa tulemuste põhjal HTTP serveri kiiruse kohta täpset järeldust teha, kuid saab üldistavalt kokku võtta, et töökiirus on antud rakenduseks piisav, sest eeldatavalt on server, kus programm tööle pannakse, kiirem, kui arendusarvuti ja korraga ei kasuta antud demorakendust üle 5 inimese.

5 Järeldused ja projekti edasised arengud

Programmeeritud liides on toimiv ja täidab talle seatud eesmärki – toimida demorakenduses piiratud arvu kasutajatega. Selleks, et antud pärandrakendus saada tootmisvalmis, oleks vaja vabaneda vananenud raamistikust ja minna üle modernsemale andmebaasisüsteemile, kus on lahendatud paralleeltööga seonduvad probleemid. See nõuaks ulatuslikku investeeringut ja suures osas koodi refaktoreerimist.

Edasi saaks projekti arendada kolmel erineval viisil:

- Programmeerida uus HTTP server mõne teise raamistikuga
- Jätkata sama HTTP serveri arendust
- Programmeerida kogu rakendus uuesti Javas ja võrguliides realiseerida läbi Spring raamistiku

Sellise lähenemise korral, kus tehakse ulatuslik ümberkirjutamine C++ keeles, oleks otstarbekas kasutada juba Boost raamistikku ja mõnda sellel põhinevat komplektsemat HTTP serverit. See välistaks ka suures osas selles töös käsitletud probleemid, mis võivad ette tulla sellise madala taseme raamistiku kasutusel. Tuleb pidada meeles, et Libmicrohttpd raamistik oli valitud lähtudes pärandrakenduse poolt seatud piirangutest, mis ümberkirjutamisel enam ei eksisteeriks. Siiski oleks võimalik ka antud projekti edasi arendada, kuid see nõuaks lisatööd, mille hulka kuuluks paralleeltöö sünkroniseerimise käsitlemine HTTP serveris, autentimine ja autoriseerimine. Samuti nõuaks edasiarendus arendatava HTTP serveri jaoks ulatuslikuma testimiskeskkonna loomist suurenenud keerukuse tõttu, mille eest muidu kannaks hoolt raamistiku arendaja.

Mastaapsema uuesti programmeerimise tõttu oleks mõeldav ka kogu rakenduse ümberkirjutamine Javas, kuid see nõuaks ilmselt veel rohkem aega, aga teeks võimalikuks võrguliikluse koha pealt Springi võimaluste kasutamise ja kõrvaldaks vajaduse päringute edasi saatmiseks C++ rakendusele.

6 Kokkuvõte

Eesmärgiks oli programmeerida liides ettevõtte pärandrakendusele, mis võimaldaks kasutada selle funktsionaalsust uues kasutajaliideses ja tulevikus ka mobiilirakenduses. Selle saavutamiseks leidsin sobiva raamistiku andmete edastuseks ja teegi JSON andmete lugemiseks ja kirjutamiseks programmeerimiskeeles C++. Programmeerimisvahendite valikul pidasin silmas ettevõtte piiranguid ja liidesele püsitatud nõudmisi.

Esmalt programmeerisin C++ pärandrakenduse jaoks liidese, mis võimaldab vastu võtta ja saata HTTP päringuid üle võrgu. Programmeerimise käigus lõin lisafunktsionaalsust rakenduse saadetavate ja loetavate andmete eripära silmas pidades – juurde tuli programmeerida suuremate andmete käitlemise tugi, mille täielik implementatsioon puudus kasutatavas raamistikus. UTF-8 sümbolite kodeeringu tugi vajas ka eraldi programmeerimist, mis selgus esimestel testimistel reaalsete andmetega. Sellele järgnes tehtud liidese häälestamine kasutades raamistiku pakutavaid parameetreid ja jälgides pärandrakenduse disaini poolt seatud piiranguid.

Järgmiseks lõin vajalikud *endpointid* Java rakenduses Spring raamistiku abiga, mis võtavad vastu esialgsed päringud ja vastavalt *endpointi* äriloogikale, edastavad selle varem programmeeritud C++ rakenduse liidesele või täidavad ise ülesande. Selleks, et edastada päringud pärandrakendusele programmeeritud liidesele tuli luua lisakomponent Java rakenduses, mis hoiab pärandrakendusega ühenduse loomiseks vajalikke andmeid ja sisaldab meetodit päringu edasisaatmiseks.

Programmeeritud HTTP serverile tegin ka jõudluse testimise, mille tulemusena selgus, et loodud programm tuleb toime ka suurema hulga HTTP päringutega. Testide tulemused näitavad, et päringule vastamise kiirus kahaneb lineaarselt päringus saadetavate andmete mahu kasvades.

Ülesanne lisada madala ressursikasutusega võrguliides pärandrakendusele oli edukas. Tänaseks on liides kasutuses demorakenduses, mida kasutatakse klientidele pärandrakenduse funktsionaalsuse demonstreerimiseks läbi modernse kasutajaliidese. Lisaks võrgurakendusele, kus on liides kasutusel, kasutab seda ka valminud IPad demorakendus, mis valmis pärast võrgurakenduse projekti. Tehtud tööst saab järeldada, et tegeledes pärandrakendusega seab rakenduse disain piiranguid sellele, kuidas läheneda

ülesandele ja milliseid vahendeid kasutada. Samuti võivad juurde tekkida uued nõuded tarkvara testides, kui ilmnevad mõne lahenduse puudused, mis võivad olla tingitud rakenduse disainist või kasutatava raamistiku piirangutest. Liides on funktsionaalne ja töötab mainitud rakendustes. Praeguse pärandsüsteemi kasutajaliidese täielikuks asendamiseks tuleks loodud liidest täiustada, lisades autentimise ja autoriseerimise. Eeldused selleks on käesolevas töös loodud, sest liidese realiseerimiseks kasutati Springi raamistikku, mis juba sisaldab mooduleid API-le ligipääsu piiramiseks. Kui ettevõte nii otsustab, siis on võimalik mõistliku arenduskuluga loodud süsteemi edasi arendada.

Kasutatud kirjandus

[1] H. Vallaste, „e-teatmik,“ [WWW]. Available: http://www.vallaste.ee. (05.2018).

[2] N. Nurzhan, P. Michael, R. Randall and I. Clemente, “Comparison of JSON and XML Data Interchange Formats: A Case Study,” *Caine*, no. 9, pp. 157-162, 2009.

[3] Facebook, „Proxygen: Facebook's C++ HTTP Libraries,“ 2017. [WWW]. Available: https://github.com/facebook/proxygen. (03.2018).

[4] I. Free Software Foundation, „GNU Libmicrohttpd,“ [WWW]. Available: https://www.gnu.org/software/libmicrohttpd/. (03.2018).

[5] Cybozu Labs, Kazuho Oku, „PicoJSON - a C++ JSON parser / serializer,“ 2015. [WWW]. Available: https://github.com/kazuho/picojson. (03.2018).

[6] „JSON for Modern C++ version,“ [WWW]. Available: https://github.com/nlohmann/json. (03.2018).

[7] THL A29 Limited, „RapidJSON Documentation,“ 2016. [WWW]. Available: http://rapidjson.org/. (03.2018).

[8] M. Yip, „Native JSON Benchmark,“ 2016. [WWW]. Available: https://github.com/miloyip/nativejson-benchmark. (04.2018).

[9] Free Software Foundation, Inc., „GNU LESSER GENERAL PUBLIC LICENSE Version 2.1, February 1999,“ 1999. [WWW]. Available: https://www.gnu.org/licenses/lgpl-2.1.txt. (04.2018).

[10] Wikipedia, The Free Encyclopedia, “Mit Licence,” Mai 2018. [Online]. Available: https://en.wikipedia.org/wiki/MIT_License. (01.05.2018).

[11] The Internet Society, „Hypertext Transfer Protocol -- HTTP/1.1,“ 1999. [WWW]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.13. (04.2018).

[12] „Headers and Includes: Why and How,“ [WWW]. Available: http://www.cplusplus.com/forum/articles/10627. (16.05.2018).

[13] Francois Yergeau, Alis Technologies, „UTF-8, a transformation format of ISO 10646,“ 2003. [WWW]. Available: https://tools.ietf.org/html/rfc3629#page-4. (04.2018).

[14] A. Allain, „Programs as Data: Function Pointers,“ 2017. [WWW]. Available: https://www.cprogramming.com/tutorial/function-pointers.html. (03.2018).

[15] Pivotal Software, „Class RestTemplate,“ 2018. [WWW]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html. (04.2018).

Lisa 1 – Erinevate JSON raamistike töötlemiskiirused [8]