TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Krista Norak 175255IDDR

ANDMEEDASTUSKIHI LOOMINE TTÜ ISEJUHTIVA AUTO KASUTAJALIIDESE JAOKS

Diplomitöö

Juhendaja: Raivo Sell
PhD

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: Krista Norak

24.05.2018

Annotatsioon

Käesoleva diplomitöö eesmärk oli luua andmeedastuskiht TTÜ isejuhtiva auto ja veebiserveri vahel, võimaldades seeläbi sõiduki kasutajaliidese loomise, mis on vajalik saavutamaks isejuhtiva sõiduki usaldusväärsema ja turvalisema liiklemise – liides võimaldab teostada sõiduki algoritmide väljundi ja sõiduandmete järelvalvet.

Töö käigus leiti lahendus kahele probleemile. Lisaks põhiprobleemile (andmeedastus), leiti analüüsi käigus, kuidas saavutada soovitud pilti liideses ehk milliseid tehnoloogilisi vahendeid tarkvara arenduses kasutada. Saadud info põhjal loodi käesoleva töö eesmärgiks olev andmeedastuskiht, mis edastab korrektselt veebiserverisse liidese loomiseks vajalikke andmeid.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 23 leheküljel, 4 peatükki, 5 joonist, 1 tabelit.

Abstract

Creating a Data Forwarding Layer for the TTÜ Self-Driving Car's User Interface

The purpose of this diploma thesis was to create a data forwarding layer between the TUT self-driving car and a web server, therefore enabling the creation of a user interface for the self-driving vehicle which is necessary for ensuring better reliability and safer driving of the car – the user interface enables supervision of the vehicle's driving data and the output of the algorithms.

The diploma thesis is divided into two parts – the analysis and technical implementation section. In the analysis section, the initial objective was set and an overview of the software currently used on the TTÜ's self-driving car was done. Furthermore, the author investigated similar solutions on self-driving vehicles around the world. Different options for creating the desired user interface were compared and data specification was done. As a result of the analysis, the author found the best ways for displaying data in the user interface and what data and in what form needed to be transported from the car to the web server.

In the technical implementation section, the author described the practical part of the current diploma thesis and created an overview of the developed software. Additionally, the author describes the testing process in this section.

Two problems were solved in the thesis. In addition to the main problem (data forwarding), the author discovered in course of the analysis how to accomplish the desired image in the user interface – which technological tools should be used in software development. Based on the collected data, the goal of the present thesis – the data forwarding layer which forwards necessary data correctly to the web server – was developed.

The thesis is in Estonian and contains 23 pages of text, 4 chapters, 5 figures, 1 tables.

Lühendite ja mõistete sõnastik

iseauto	Isejuhtiv auto
GPS	*Global Positioning System*, globaalne positsioneerimisseade
Lidar	*Light Detection and Ranging*, laserskaneerimise seade
ROS	*Robot Operating System*, robotiarendusplatvorm
Autoware	Tarkvarapakett sõiduki juhtimiseks
Sõlm	*Node* – ROS'i modulaarne osa
Teema	*Topic* – tarkvaraline vahend ROS sõnumite haldamiseks [1]
Sõnum	*Message* – informatsioonikandja ROS'i siseses suhtluses [1]
Jälgima	*Subscribe* – ROS'i teema jälgimine
Avaldama	*Publish* – ROS'i teema avaldamine
Bag	.bag faili formaadis salvestatud ROS sõnumite andmed
RViz	*ROS Visualization* - ROS'i raamistikule mõeldud 3D visualiseerimise keskkond [2]
WViz	*Web Visualization* - ROS'i tööriist, mille eesmärk on kuvada veebis sarnane 3D visualisatsioon nagu RViz'is [3]
Sõltuvuspakid	*Dependencies* – pakid, mille olemasolu on vastava programmi käitamiseks vajalikud
Ros3djs	ROS'ile mõeldud 3D visualiseerimise *JavaScript*'i teek
JSON	*JavasScript Object Notation* – Andmevahetusvorming
UDP	*User Datagram Protocol* – kasutajadatagrammi protokoll
Yolo	*You Only Look Once* – objektituvastusprogramm
UI	*User Interface* - kasutajaliides

Sisukord

Jooniste loetelu

Tabelite loetelu

1 Sissejuhatus

Isejuhtivate sõidukite loomine on tänapäeval väga aktuaalne teema. Kõige olulisem nende loomisel, on turvalisuse tagamine. Hiljutised sündmused maailmas seoses isejuhtivate sõidukite avariidega, suurendavad inimeste kartust ja umbusku nende turvalisusesse. Suurendamaks sõiduki usaldusväärsust, on vaja täiendavat järelvalvet sõiduki kohta. Selleks on vajalik liides, mille kaudu on võimalik jälgida auto algoritmide väljundit ja sõiduandmeid. Tagamaks, et liides kuvab korrektseid andmeid, on oluline, et oleks olemas toimiv andmeedastuskiht sõiduki ja veebiserveri vahel.

Käesoleva diplomitöö eesmärgiks on luua andmeedastuskiht TTÜ isejuhtiva auto ja veebiserveri vahel, et võimaldada isejuhtiva sõiduki kasutajaliidese loomine, misläbi saavutatakse sõiduki algoritmide väljundi ja sõiduandmete järelevalve, tagades auto usaldusväärsema ja turvalisema liiklemise. Töö raames plaanib autor leida analüüsi käigus mis andmeid on vaja liidese loomiseks ja mis on parim viis nende edastamiseks ning luua vastav tarkvara, mis oleks hõlpsasti kasutatav ning paindlik tuleviku lisaarenduste tarvis.

Diplomitöö on jaotatud kahte ossa: analüüs ja tehniline teostus. Töö analüüsi osas koostatakse lähteülesanne ning ülevaade TTÜ isejuhtiva auto olemasolevast tarkvarast ning seal kasutusel olevatest tehnoloogiatest. Lisaks uurib autor sarnaseid mujal maailmas isejuhtivatel sõidukitel kasutusel olevaid lahendusi. Võrreldakse erinevaid võimalusi soovitud liidese loomiseks ning koostatakse andmete spetsifikatsioon. Analüüsi tulemusena leitakse, mis on parim viis andmete kuvamiseks liideses ning mis andmeid ja mis kujul on vaja saada isejuhtivast autost veebiserverisse.

Tehnilise teostuse osas kirjeldab autor töö praktilist osa ning loob ülevaate arendatud tarkvarast. Lisaks kirjeldab autor selles osas testimisprotsessi.

2 Analüüs

2.1 Lähteülesanne

2017. aasta kevadel asus TTÜ koostöös Silberautoga ehitama Eesti esimest isejuhtivat autot ehk iseautot [4]. Loodav sõiduk on mõeldud TTÜ linnaku piires liiklemiseks. Joonis 1 kujutab iseauto disaini. Planeeritav tähtaeg sõiduki valmimiseks on 2018. aasta sügis.

Joonis 1. Iseauto disain [5]

Sellise sõiduki loomisel on oluline veenduda, et auto töö on turvaline ning jätkusuutlik. Seega tuleks võimaldada iseauto järelevalve. Soovitud eesmärgi saavutamiseks on vaja liidest, mille kaudu jälgida auto algoritmide väljundit ja sõiduandmeid. Selleks, et tagada liidese usaldusväärsus, on vaja, et seal kasutatav info oleks korrektne. Seega on vaja andmeedastuskihti iseauto ja veebiserveri vahel. Käesoleva diplomitöö eesmärk on saada tööle andmete edastamine iseautolt veebiserverisse, kust toimub hiljem andmete kuvamine soovitud kujul (liides).

2.1.1 Tarkvara iseauto peal

Isejuhtiva auto operatsioonisüsteemiks on Ubuntu 16.04, mille peal jookseb ROS (*Robot Operating System*) ja Autoware (sõiduki juhtimist võimaldav tarkvarapakett), mis on kogu auto töö aluseks. Järgnevalt kirjutab autor lähemalt autos kasutusel olevast tarkvarast.

2.1.2 ROS

Robot Operating System ehk ROS on raamistik, mis on mõeldud robotite tarkvara loomiseks ehk see on nö vahelüli roboti ja arvuti vahel. See on avatud lähtekoodiga ning hõlmab endas mitmeid vahendeid ja teeke, mis aitavad arendajatel kirjutada robotitele koodi. ROS on oma olemuselt modulaarne, koosnedes mitmetest sõlmedest (*nodes*) ja teenustest (*services*) [6].

Iseautol on kasutusel ROS Kinetic versioon, mis on 2016. aastal avalikustatud kümnes ROS'i distro ehk üks uuemaid ROS'i versioone [7].

ROS'il on partnervõrk (*peer-to-peer network*) nimega Computation Graph, mis võimaldab protsessidel ühiselt andmeid töödelda. Sõlmed suhtlevad omavahel sõnumite kaudu. Igal sõnumil on kindel struktuur ning koosneb kindlate andmetüüpidega väljadest. Sõnumeid edastatakse läbi teemade (*topic*s), mis on vahend, mida arendaja kasutab antud sõnumi avaldamiseks või jälgimiseks. Kõiki neid nimesid haldab ROS Master, mis töötab nagu DNS server ehk see pakub ainult informatsiooni nimede kohta ning ei korralda ise sõlmede omavahelist suhtlust. Sõnumi saatmiseks avaldatakse (*publish*) sõlmes teema ning kui mõnel sõlmel on seda infot vaja, jälgitakse (*subscribe*) vastava nimega teemat. Ühte teemat võivad jälgida ja avaldada paralleelselt mitu erinevat sõlme. Konkreetsete päringute tegemiseks, kui ei piisa vaid teemade jälgimisest, kasutatakse teenuseid (*services*). Klient saadab päringsõnumi teenust pakkuvale sõlmele (*node*) ning jääb vastust ootama. Päringul ja vastusel on mõlemal kindel struktuur [8].

ROS võimaldab salvestada sõnumite andmeid .bag faili formaadis. Saadud andmeid on võimalik kasutada roboti arendus- ja testimistöödes. Tänu oma nimelaiendile, kutsutakse sellist faili *bag*'iks ning selle loomiseks kasutatakse enamasti vahendit *rosbag* [9]. *Bag'*i on võimalik taasesitada, simuleerides nii reaalset testolukorda. Raskesti saadav info nagu näiteks sensoritelt tulev info, on sel juhul taaskasutatav ning arendustöid saab efektiivsemalt teha. Iseauto puhul näiteks salvestatakse testsõitudel andmed *bag'*i.

Enamasti ei salvestata kõiki ROS'is liikuvaid sõnumeid, vaid valitakse neid, mida on soov hiljem uurida või kasutada. Lisaks uute arenduste hõlbustamiseks aitab see ka siluda olemasoleva koodi vigasid.

2.1.3 Autoware

Autoware on tarkvarapakett, mis sisaldab mitmeid isejuhtiva sõiduki juhtimiseks vajalikke tööriistasid (näiteks teekonna planeerimine, lokalisatsioon, simulatsioon ja muud). Pakett põhineb ROS'il ning on välja töötatud Jaapani ülikooli Nagoya University ja Tier IV poolt [10].

2.1.4 RViz

RViz on ROS'i raamistikule mõeldud 3D visualiseerimise keskkond. See aitab visualiseerida andmeid kaameralt ja laseritelt, kuvada kaarte ja palju muud. Selle eesmärk on kuvada ruumiliselt see maailm, mida robot nii-öelda „näeb". Vaade on konfigureeritav, et kasutaja saaks näha andmeid just sellisel kujul nagu tal vaja on. RViz on vabavaraline tarkvara ning seda kasutatakse laialdaselt robotite arendamisel [2].

2.2 Analoogsed olemasolevad lahendused

Isejuhtivate sõidukite projekte on maailmas mitmeid. Autor uuris, kas ja kuidas on lahendatud teiste isejuhtivate autode kasutajaliidesed. Uuritud iseautode nimedeks on EZ10, Parkshuttle, Olli, WEpod, Navya, Novus drive, Sedric, Waymo ja CityMobil2.

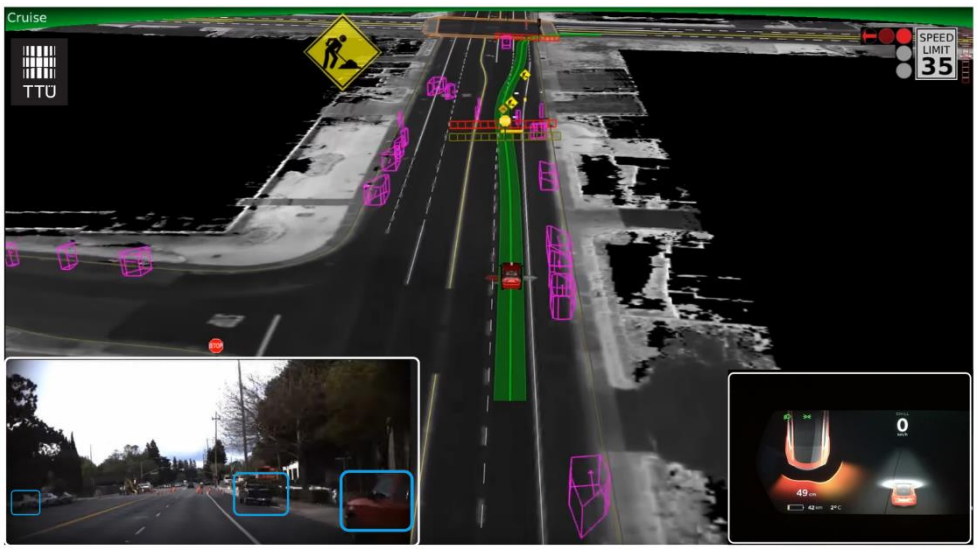
Üldiselt on tehtud isejuhtivatele sõidukitele mobiilirakendused, millega broneerida koht isejuhtiva sõidukil, valida sihtpunkt ja maksta sõidu eest. Vähe on informatsiooni selle kohta, et oleks tehtud kasutajaliides, kust jälgida sõiduki parameetreid ja algoritmide väljundeid, kuid on mõned erandid. Üheks nendest on Volkswageni Groupi arendusprojekt Sedric, kus tuuleklaasiks plaanitakse teha läbipaistev OLED ekraan, kust saab soovi korral näha täiendatud reaalsuse (*augmented reality*) andmeid [11]. Lisaks leidis autor EZ10 puhul liidese, mis koosneb Google Maps kaardist, kuhu on lisatud auto parameetrid ja kaardi sätted [12]. Google isejuhtiva auto Waymo sisse on disainitud ekraan, kust reisija saab jälgida, kuidas auto ümbritsevat keskkonda näeb ja mis liigutust kavatseb järgmisena teha. Vaadet üritatakse teha ka kasutajasõbralikumaks, muutes ja lisades detaile elementidele, et reisijatel oleks mugav ja arusaadav nägemus sõiduki liikumisest [13].

2.3 Arendatav lahendus

Käesoleva töö raames läheb arendamisele andmeedastuskiht iseauto ja veebiserveri vahel. Enne kui saab aga hakata seda ehitama, tuleb välja selgitada, milliseid andmeid liidese loomiseks vaja on ja kuidas neid saada. Antud peatüki eesmärgiks on välja selgitada, kuidas hakatakse tulevikus looma iseauto kasutajaliidest ning selle info põhjal järgnevates peatükkides kirjeldada andmeedastuskihis osalevaid andmeid ja nende liikumist.

Iseauto kasutajaliides on jaotatud kolmeks osaks: kaart, video ja parameetrite kuvamine (Joonis 1). Video ja parameetrite kuvamiseks tuleb otse-edastada veebiserverisse kaamerapilt ning info parameetrite kohta. Nende edastamine toimiks hästi, kui jooksutada veebiserveri peal eraldi ROS ning saata vastavad andmed iseauto ROS'i pealt veebiserveri ROS'i peale, kasutades UDP protokolli. Selle, et need andmed oleksid kuvatud nii nagu UI visioonil (Joonis 2), tagab liidese enda kood.

Keerulisemaks ülesandeks on kaardi kuvamine. Selleks, et liides suudaks kuvada esialgsele visioonile vastavat kaarti, tuleb leida veebi väljund RViz'ile. Soovitud tulemuseni jõudmiseks on mitu võimalust. Järgnevalt tutvustatakse erinevaid variante RViz'i väljundi kuvamiseks ning tuuakse välja, milline neist on parim ja miks.



Joonis 2. Esialgne visioon liidesest

2.3.1 WViz

WViz on ROS'i tööriist, mille eesmärk on kuvada veebis sarnane 3D visualisatsioon nagu RViz'is. Sinna on lisatud ka mehhanismid, mis lubavad kasutajal robotit kontrollida. Kasutajatel pole vaja ROS'i, et näha visualisatsiooni ning saata robotile käske. Esmapilgul tundub, et WViz on potentsiaalne viis kaardi kuvamiseks liideses, kuna dokumentatsioonis on kirjas, et see toetab *Point Cloud 2* kuvatüüpi, mis on ühtlasi ka iseauto kaardi tüübiks [3].

Lähemal uurimisel selgus, et WViz ja selle sõltuvuspakid (*dependencies*) on väga vanad ja ei toeta osalt ROS Kineticut. Seetõttu on väga raske panna seda uuema ROS'iga nagu näiteks iseauto peal oleva ROS Kineticuga ühilduma. Lisaks on probleemiks asjaolu, et WViz lubab kasutajatel robotit kontrollida. Iseauto turvalisuse tagamiseks on äärmiselt oluline, et läbi kasutajaliidese poleks kuidagi võimalik autot juhtida või muul moel selle tegevust häirida.

2.3.2 Ros3djs

Ros3djs on 3D visualiseerimise *JavaScript*'i teek, mida kasutatakse koos teiste ROS'i *JavaScript*'i teekidega. See loodi Robot Web Tools'i (vabavaralised tööriistad veebipõhiste robotrakenduste loomiseks) raames [14], [15]. *Point Cloud 2* tüüpi kaardi kuvamiseks on ros3djs teegis olemas klass PointCloud2. Selle eesmärk on jälgida etteantud ROS'i teemat (*topic*) ning kuvada seal olevad punktid. Lisaks ei ole kasutajal võimalik läbi kuvatud kaardi robotile käske saata [16].

Katse kuvada kaarti kasutades ros3djs teegi PointCloud2 objekti ebaõnnestus. Esialgu konsoolis olnud vead vananenud klasside kohta said lokaalselt korda tehtud, kuid ka siis ei olnud veebis näha kaarti. Lisaks ei suutnud antud *JavaScript*'i teek taluda suuri andmemahtusid, mida oli näha, kui jooksutati ROS *bag*'i, mis edastas sadu tuhandeid punkte. Probleem võib olla ka selles, et ros3djs poolt kasutatav *rosbridge_server* ei suuda edastada korrektselt *PointCloud2* tüüpi ROS'i sõnumeid. Sellegipoolest, mõeldes ette, jõuti arusaamale, et isegi kui saaks ros3djs teegiga *PointCloud2* tüüpi kaardi kuvamise tööle, siis tuleks sinna lisada veel vektorkaart ning auto jälgimine. Kuna aga sellist võimalust antud teek hetkeseisuga ei paku, ei ole ros3djs iseauto kasutajaliidese loomiseks mõistlik vahend.

2.3.3 RViz'i vaate otseedastamine

Tänu tehtud uurimisele, jõudis autor arusaamani, et hetkel ei eksisteeri vahendit, millega saaks saavutada sama pildi nagu RViz programmis, st nii *PointCloud2* kaardi kui ka vektorkaardi kuvamine ja auto jälgimise realiseerimine. Seega on kõige mõistlikum variant kuvada RVizi enda vaadet. Selle jaoks tuleb veebiserveri ROS'i peal jooksutada RViz ning kasutades sobilikku vahendit, otseedastada ekraanipilti videona veebiliidesesse. Katsetuse käigus ilmnes, et ekraani voogedastamine ja vastuvõtmine kasutades VLC Media Player'it toimis.

2.4 Süsteeminõuded

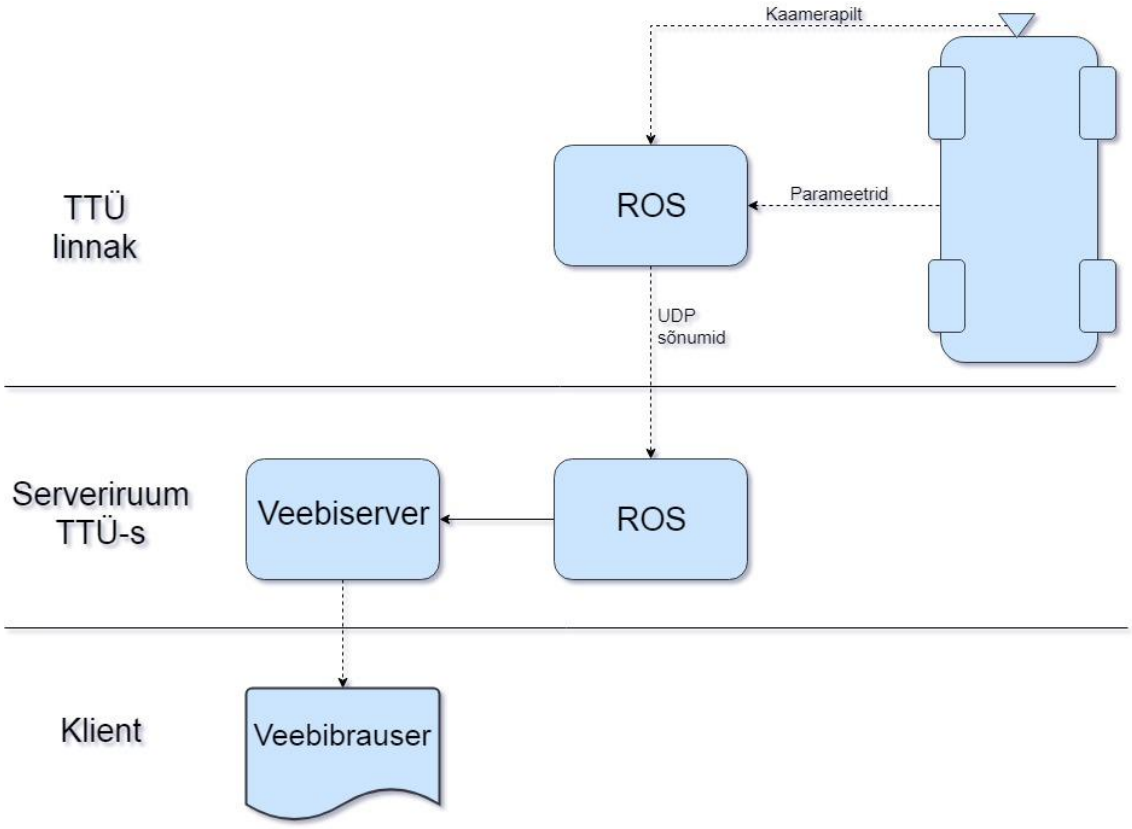
Järgnevalt täpsustab autor nõuded, mis käesoleva diplomitöö raames peavad täidetud saama.

Funktsionaalsed nõuded:

- Kaamerapildi edastamine veebiserverisse
- Auto asukoha edastamine veebiserverisse
- Iseauto kiiruse edastamine veebiserverisse
- Parkimisandurite näitude edastamine veebiserverisse
    - Vastava ROS sõnumi tüübi loomine
    - ROS sõnumit edastava teema loomine
- Aku laetuse taseme kohta info edastamine veebiserverisse
    - Vastava ROS sõnumi tüübi loomine
    - ROS sõnumit edastava teema loomine
- Andmete vastuvõtmine veebiserveris

2.5 Planeeritav süsteemi arhitektuur

Joonis 3 esitab planeeritavat süsteemi arhitektuuri andmete edastamiseks veebiserverisse. Lisatud on ka kliendipoolne osa, parema ülevaate saavutamiseks.



Joonis 3. Planeeritav süsteemi arhitektuur

2.6 Andmete spetsifikatsioon

Spetsifikatsiooni eesmärgiks on kirjeldada, mis andmeid on vaja auto pealt, kuidas neid veebiserveris kasutatakse ja mis kujul andmed läbi võrgu liiguvad.

Nagu eelnevalt peatükis 2.3 mainitud, on kasutajaliidesel kolm vaadet: kaart, kaamera ja parameetrid (Joonis 2), mistõttu on käesolev peatükk grupeeritud nende vaadete alusel kolmeks osaks.

Andmete edastamiseks läbi võrgu autost veebiserverisse tuleks kasutada UDP protokolli, kuna sellega saavutatakse suurem info edastuskiirus ehk andmed jõuavad kohale väiksema viivitusega, tagades serveris võimalikult värske informatsiooni. Seda tänu sellele, et vastupidiselt TCP protokollile, ei kontrollita UDP puhul pidevalt, et kogu info ikka kohale jõuab, saavutades seega madalama ribalaiuse (*bandwidth*), lisa- (*overhead*)

ja latentsusaja (*latency*) [17]. Auto ROS'i peal hakkab jooksma sõlm (*node*), mis kogub auto pealt andmeid ja saadab need UDP protokolliga veebiserverisse, kus on lahti vastav port, mille kaudu kuulatakse saabuvaid UDP sõnumeid.

Andmete saatmine üle võrgu toimuks JSON'i kujul. Video võiks saata eraldi voona, kuna see tuleb ilmselt erineva kiirusega kui teised andmed ning edastada oleks mõistlikum *byte* kujul.

Tabel 1 koondab endas kõik andmed, mida on hetkeseisuga (tulevikus võib tekkida vajadus saada rohkem infot auto parameetrite kohta) vaja auto pealt saada veebiserverisse.

Tabel 1. Andmed, mida on vaja auto pealt

Nr	**Andmed autolt**	**ROS sõnumi tüüp**	**Mille jaoks vajalik?**
1	Auto asukoht (*Lidar/GPS*)	*geometry_msgs/PoseStamped*	Auto jälitamine kaardil
2	Video kaamerast	*sensor_msgs/Image*	Kaamerapildi kuvamine
3	Kiirus	*geometry_msgs/TwistStamped*	Parameetrite kuvamine
4	Aku laetus	*Hetkel auto ROS'is ei eksisteeri*	Parameetrite kuvamine
5	10x ultraheli andurite näit	*Hetkel auto ROS'is ei eksisteeri*	Parkimisandurite kuvamine graafiliselt

Kaart

Kaardi kuvamiseks on vaja *PointCloud2* kaarti, vektorkaarti ning auto asukohta. Esimesed kaks saab laadida otse veebiserverist, auto asukoha kohta on vaja infot auto pealt. Auto asukoha määrab Lidar *(light detection and ranging)* või GPS (*Global Positioning System*). ROS sõnumi tüüp, mida antud juhul jälgima peaks, on *geometry_msgs/PoseStamped*, kus on määratud auto positsioon ja orientatsioon koos ajatempliga.

Kaamera

Kaamerapilt tuleb saada auto pealt ning enne veebiserverisse saatmist tuleb see kokku pakkida. Kuna auto peale tuleb mitu kaamerat, siis tuleks ROS'i sõlm teha nii, et teema nimi, mida jälgitakse on konfigureeritav (*launch* failis anda teema nimi parameetrina

kaasa). Siis saab valida, millist kaamera pilti edastatakse. Serveris saab lasta video läbi *Yolo* (objektituvastusprogramm), mis tekitab tuvastatud objektide ümber kastikesed nagu on näha UI visioonil (Joonis 2).

Parameetrid

Parameetrite edastamine peaks olema paindlik, et oleks hiljem võimalik neid lihtsalt lisada. Esialgu on kuvatavad parameetrid: kiirus, aku laetus, 10x ultraheli andurite näit (parkimisandurid graafiliselt). Kiirusel on auto ROS'is olemas *geometry_msgs/TwistStamped* tüüpi teema, mida saab jälgida, kuid aku laetuse ja ultraheli andurite kohta sõnumeid ROS'is hetkel ei eksisteeri.

3 Tehniline teostus

3.1 ROS'i sõlm iseauto peal

Andmeedastuskihi loomise esimeseks sammuks pärast analüüsi oli ROS'i sõlme loomine iseauto peale. Enne koodi kirjutamist, tuli otsustada, kas kasutada arenduskeeleks *Python*'it või *C++*'i. Üldine lähenemine ROS projektide puhul on, et suuremate ja kriitilisemate arenduste jaoks kasutatakse *C++*'i ning *Python*'it kasutatakse väiksemate arenduste puhul ja olukordades, kus selle jõudlus ei erine oluliselt *C++* 'i jõudlusest. Kuna antud arendus ei ole iseauto sõiduks kriitilise tähtsusega ja *Python*'i kasutamine ei vähenda sõlme jõudlust ning lisaks on autoril selle keelega rohkem kogemust, otsustati *Python*'i kasuks. Käesoleva sõlme lähtekood on toodud Lisas 1.

ROS'i sõlm loodi TTÜ iseauto projekti „iseauto/nodes" repositooriumisse. Vastavalt analüüsile, oli vaja, et jälgitav kaamera teema oleks lihtsasti muudetav, et saaks kergesti valida, millise kaamera pilti näidata. Selleks lisas autor sõlme *launch* faili parameetri nimega „camera_topic", mille väärtus on valitud kaamera pilti edastava teema nimi:

```
<launch>
   <node name="server_gateway" pkg="server_gateway" type="server_gateway.py">
      <param name="camera_topic" type="string" value="/camera/image_raw"/>
   </node>
</launch>
```

Sõlme *Python*'i failis määras autor ära, milliseid teemasid antud sõlm jälgib: „/current_pose“, „/current_velocity“, CAMERA_TOPIC, „/current_battery“, „parking_sensor1“. Kuna parkimisandurite ehk ultraheli andurite näitu hetkel auto ROS'i peal ei avaldat ning neid on ka sõiduki peal 10 asemel hetkel ainult üks, siis pidi autor ise looma sobiva teema nime ning autor pani käesoleva sõlme jälgima vaid ühte vastavasisulist teemat. Vastav koodiosa:

```
rospy.Subscriber("/current_pose", PoseStamped, pose_callback)
rospy.Subscriber("/current_velocity", TwistStamped, velocity_callback)
rospy.Subscriber(CAMERA_TOPIC, Image, image_callback)
rospy.Subscriber("/current_battery", BatteryInfoStamped, battery_callback)
rospy.Subscriber("/parking_sensor1", ParkingSensorStamped,
parking_sensor_callback)
```

Igale jälgijale (*Subscriber*) tegi autor *callback* funktsioonid, mis võtavad vastavast ROS sõnumist vajaliku info ning saadavad need läbi veebi UDP protokolliga veebiserverisse. Kaamerapilti edastas autor *byte* kujul. Kõiki ülejäänud andmeid aga JSON'i formaadis. Järgnevalt on toodud auto positsiooni jälgija *callback* funktsiooni realisatsioon programmikoodis:

```
def pose_callback(data):
    message = json.dumps({'msg_name': 'current_pose',
                          'time':
datetime.fromtimestamp(data.header.stamp.to_sec()).__str__(),
                          'pose': {
                               'position': {
                                    'x': data.pose.position.x,
                                    'y': data.pose.position.y,
                                    'z': data.pose.position.z
                               },
                               'orientation': {
                                    'x': data.pose.orientation.x,
                                    'y': data.pose.orientation.y,
                                    'z': data.pose.orientation.z,
                                    'w': data.pose.orientation.w
                               }
                           }
                          })
    send_udp_message(message, 11201)
```

Kaamerapildi edastamisel pidi arvestama, et üks ROS'i *Image* tüüpi sõnum sisaldab väga suures mahus andmeid. Seega pakkis autor pildi andmed enne saatmist kokku. Esialgu üritati saata kogu pakitud kaamerapilti ühe *byte* tüüpi UDP sõnumina veebiserverisse. Nähes kohe esimeste katsete käigus, et kaamerapildi ühes tükis edastamine ei toimi, kuna UDP sõnum on sel juhul liiga suur, otsustas autor saata kokku pakitud kaamerapildi serverisse osade kaupa. Selleks, et serverisse saabunud tükkidest oleks võimalik tervik pilt kokku panna, lisatakse iga osa külge, mis ID-ga pildi juurde see kuulub ning mitmes osa pildist see on. Lisaks saadetakse enne tükkide edastamist UDP sõnum, mis sisaldab pildi metaandmeid, ID-d ning saadetavate osade hulka. Pildi ID on globaalne muutuja, mis sõlme käivitamisel saab väärtuseks 1 ning mida suurendatakse ühe võrra pärast iga pildi edastamist. Kaamerapildi tüki edastamisel saadetakse ID *unsigned int* tüübina ehk see hõivab *byte* kujul saadetavast sõnumist neli esimest baiti. Tüki järjekorranumbri tüübiks on *unsigned short* ehk sellele kuuluvad järgmised 2 baiti sõnumist. Järgnevalt on näha kirjeldatud lahenduse realisatsiooni koodis.

```
def image_callback(message):
    global IMAGE_ID
    compressed_msg = zlib.compress(message.data)
    length = len(compressed_msg)
    chunk_size = 6400
    chunks_amount = length / chunk_size + 1
    send_image_info(chunks_amount, message)
    chunk_counter = 1
    for i in range(0, length, chunk_size):
        chunk = struct.pack(">I", IMAGE_ID) + struct.pack(">H",
chunk_counter) + compressed_msg[i:i+chunk_size]
        send_udp_message(chunk, 11200)
        chunk_counter += 1
    IMAGE_ID = IMAGE_ID + 1

def send_image_info(chunks_amount, message):
    message = json.dumps({'msg_name': 'image info',
                          'image_id': IMAGE_ID,
                          'height': message.height,
                          'width': message.width,
                          'encoding': message.encoding,
                          'is_bigendian': message.is_bigendian,
                          'step': message.step,
                          'amount_of_chunks': chunks_amount
                         })
    send_udp_message(message, 11200)
```

Erinevat tüüpi informatsiooni edastavad sõnumid saadetakse erinevatele portidele. See tagab, et hiljem kasutajaliidest luues oleks mugav panna liidese erinevad osad just vajalikku infot kuulama. Pordi number, kuhu sõnum saata, määratakse igas *callback'*is. IP aadress, kuhu UDP sõnum saadetakse, määratakse koodifaili alguses. Allpool on toodud kood, kus on näha meetodit, mis saadab etteantud sõnumi soovitud porti:

```
def send_udp_message(message, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.sendto(message, (UDP_IP, port))
```

Analüüsi käigus selgus, et hetkel puuduvad iseauto ROS'i peal teemad, mis edastaks infot aku laetuse ja parkimisandurite näidu kohta. Seega pidi autor välja mõtlema nii nende teemade nimetused kui ka sõnumi tüübi. Mõlema puhul on vaja, et edastaks infot koos ajatempliga. Aku laetuse sõnumi puhul on vaja edastada protsent, parkimisandurite ehk ultraheliandurite puhul kaugus objektist. Tagamaks paindlikkuse, tuleks need väärtused saata *float* andmetüübina. Selleks kirjutas autor eraldi ROS sõnumid *iseauto_msgs/BatteryInfoStamped* ja *iseauto_msgs/ParkingSensorStamped*, mis koosnevad ajatempliga päisest ning vastavalt *BatteryInfo* või *ParkingSensor* sõnumist.

ParkingSensor sõnum koosneb sensori nimest (*string*) ning kaugusest (*float*). Tänu sellele, et sõnumis on ka sensori nimi, saab sõlmes jälgida kümne erineva ultraheliandurи poolt avaldatavaid teemasid ning kasutada kõigi jaoks sama *callback* funktsiooni.

3.2 Andmete vastuvõtmine veebiserveris

Veebiserveriks on virtuaalserver, millel on 16 GB mälu, 4 tuuma ning 200 GB kettamahtu. Operatsioonisüsteemiks on Ubuntu 16.04, kuna sellel töötavad ka iseauto peal kasutusel olevad arendused.

Andmete vastuvõtmiseks kirjutas autor uue sõlme, mis jookseb veebiserveri ROS'i peal. Selle eesmärgiks on kuulata paralleelselt kõiki porte, millele iseauto pealt UDP sõnumeid saadetakse ning saabuvat infot avaldada veebiserveri ROS'i. Valminud sõlme lähtekood on toodud Lisas 2.

Selleks, et antud sõlm saaks tegeleda kõigi saabuvate sõnumitega paralleelselt, kasutas autor lõimimist (*threading*). Erinevatele sõnumitüüpidele loodi eraldi lõimed:

```
thread1 = threading.Thread(target=start_image_udp_server)
thread2 = threading.Thread(target=start_velocity_udp_server)
thread3 = threading.Thread(target=start_pose_udp_server)
thread4 = threading.Thread(target=start_battery_udp_server)
thread5 = threading.Thread(target=start_parking_sensor_udp_server)
thread1.start()
thread2.start()
thread3.start()
thread4.start()
thread5.start()
```

Iga lõim paneb serveri kuulama saabuvaid UDP sõnumeid vastava pordi peal, misjärel iga saabuv sõnum saadetakse vastavasse *callback* funktsiooni, kus seda töödeldakse vastavalt vajadusele. Allpool on toodud iseauto kiirust kuulav funktsioon ning saabuva sõiduki kiirust edastava sõnumi *callback* funktsioon:

```
def start_velocity_udp_server():
    for data in udp_server(port=11202):
        udp_velocity_callback(data)

def udp_velocity_callback(data):
    json_data = json.loads(data)
    publish_velocity_to_ROS(json_data)
```

Informatsiooni edastamiseks serveri ROS'is, tuli luua uus vastavat tüüpi ROS'i sõnum, täita see saabunud andmetega ning seejärel avaldada ROS'is. Järgnevalt on näha kiiruse avaldamise funktsioon:

```
def publish_velocity_to_ROS(velocity):
    ros_velocity = TwistStamped()
    ros_velocity.twist.linear.x = velocity['twist']['linear']['x']
    ros_velocity.twist.linear.y = velocity['twist']['linear']['y']
    ros_velocity.twist.linear.z = velocity['twist']['linear']['z']
    ros_velocity.twist.angular.x = velocity['twist']['angular']['x']
    ros_velocity.twist.angular.y = velocity['twist']['angular']['y']
    ros_velocity.twist.angular.z = velocity['twist']['angular']['z']
    velocity_publisher.publish(ros_velocity)
```

Kõige suuremaks väljakutseks käesoleva sõlme loomisel osutus kaamerapildi kokku panemine saabuvatest osadest. Sõlm peab ootama ära kõik tükid ning kaamerapildi infot edastava sõnumi ning ainult siis, kui need kõik on olemas, ehitama nende abil üles korrektse pildi ja avaldama selle ROS'i. Selleks lõi autor *dictionary*, kuhu lisatakse väärtusi, kus võtmeks on pildi ID ning väärtuseks *ImageChunks* klassi objekt. *ImageChunks* on autori loodud klass, kuhu salvestatakse kogu vajalik info kaamerapildi taasloomiseks saabuvatest sõnumitest ehk seal on väljad pildi metaandmete jaoks, loend saabunud pildi tükkidest, oodatavate tükkide arv ning kaamerapildi taasloomiseks vajalikud meetodid. Tükkide lisamisel kasutatava *Chunks* klassi loomine oli vajalik selleks, et oleks hiljem võimalik loend sorteerida tükkide järjekorranumbrite põhjal. Järgnevalt on toodud *ImageChunks* klassi realisatsioon koodis.

```
class ImageChunks:
   def __init__(self):
       self.amount_of_chunks = -1
       self.chunks = []

   def is_ready(self):
       return self.amount_of_chunks == len(self.chunks)

   def add_chunk(self, seq, new_chunk):
       chunk = Chunk(seq, new_chunk)
       self.chunks.append(chunk)

   def set_metadata(self, json_msg):
       self.amount_of_chunks = json_msg['amount_of_chunks']
       self.height = json_msg['height']
       self.width = json_msg['width']
       self.encoding = json_msg['encoding']
       self.is_bigendian = json_msg['is_bigendian']
       self.step = json_msg['step']


   def get_full_image(self):
       imageList = sorted(self.chunks, key=lambda x: x.seq)
       image = b''
       for x in imageList:
           image += x.data
       return zlib.decompress(image)
```

Kuna nii pildi infot edastav sõnum kui ka tükid saadetakse samasse porti, siis tuleb andmete saabumisel esmalt eristada, kumba sõnumiga tegemist on. Selleks kontrollitakse, kas tegemist on JSON või *byte* tüüpi sõnumiga, sest pildi info saadetakse JSON'ina ning tükid *byte* kujul. Mõlemal juhul kontrollitakse, kas eelnevalt mainitud piltide *dictionary*'is on vastava pildi ID'ga võti juba olemas. Kui ei ole, siis see lisatakse ning ühtlasi käivitatakse meetod, mis kustutab anud võti-väärtus paari *dictionary'*ist teatud aja möödumisel. See tagab, et vanu või katkiseid (üks pildi osa näiteks ei jõua kohale) andmeid ei hoitaks serveris alles. Lisaks kontrollitakse nii pildi info kui ka tüki sõnumi saabumisel, kas kõik pildi loomiseks vajalikud sõnumid on saabunud ning kui on, siis avaldatakse kaamerapilt ROS'i. Kirjeldatud funktsionaalsust teostab allpool toodud koodiosa.

```
def udp_image_callback(data):
    global images
    try:
        json_data = json.loads(data)
        image_id = json_data['image_id']

        if (image_id not in images):
            images[image_id] = ImageChunks()
            delete_after_five_seconds(image_id)
        images[image_id].set_metadata(json_data)

        if(images[image_id].is_ready()):
            publish_image_to_ROS(images[image_id])
            del images[image_id]
    except ValueError:
        image_id = struct.unpack(">I", data[:4])[0]
        seq = struct.unpack(">H", data[4:6])[0]
        chunk = data[6:]

        if (image_id not in images):
            images[image_id] = ImageChunks()
            delete_after_five_seconds(image_id)
        images[image_id].add_chunk(seq, chunk)

        if(images[image_id].is_ready()):
            publish_image_to_ROS(images[image_id])
            del images[image_id]

def publish_image_to_ROS(image):
    ros_image = Image()
    ros_image.data = image.get_full_image()
    ros_image.height = image.height
    ros_image.width = image.width
    ros_image.encoding = image.encoding
    ros_image.is_bigendian = image.is_bigendian
    ros_image.step = image.step
    image_publisher.publish(ros_image)
```

3.3 Testimine

Tagamaks, et loodud andmeedastuskiht toimib vastavalt vajadustele, testis autor oma loodud ROS'i sõlmesid. Selleks käivitas autor esmalt oma loodud sõlme, mis hakkab jooksma iseauto peal, ning avaldas ROS'i teemasid, mida andmeedastuskiht jälgib. Kõik teemad, mis iseauto ROS'i peal juba eksisteerivad, sai autor avaldada taasesitades TTÜ iseauto testsõitudel loodud *rosbag'*i. Teised teemad tuli avaldada terminalis käsuga „*rostopic pub /topic package/msg_type '<message>'* ". Allpool on näidatud, kuidas toimus aku laetuse taseme kohta infot saatva teema „/current_battery" avaldamine läbi käsurea:

```
rostopic pub /current_battery iseauto_msgs/BatteryInfoStamped '{header: auto,
batteryInfo: {percentage: 62.0}}'
```

Esmalt tuli jälgida, et käivitatud sõlme töö käigus ei tekiks konsooli ühtegi viga. Kuna alguses üritas autor edastada kaamerapilti ühes tükis, siis tekkis konsooli viga „*Message too long*". Lähemal uurimisel selgus, et UDP sõnumitel on vastavalt operatsioonisüsteemile määratud lubatud suurim UDP paki suurus ning kaamerapilt oli sellest suurem. Lahendamaks probleemi, otsustas autor jaotada kaamerapildi mitmeks osaks ja saata need eraldi UDP sõnumitena serverisse.

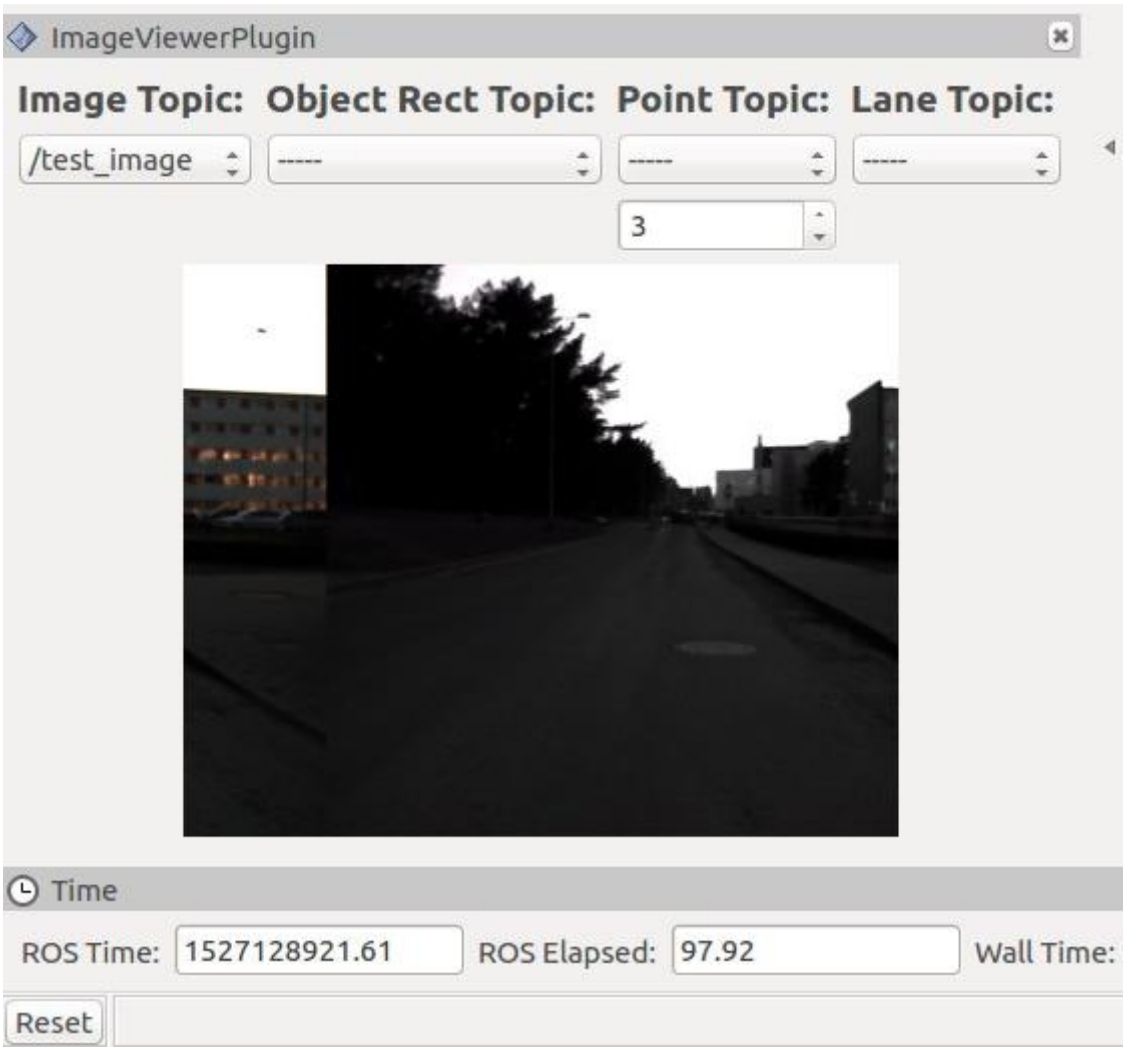
Järgmiseks, kui sõlme töö käigus enam ühtegi veateadet ei ilmunud, kirjutas autor testimise eesmärgil ajutise *Python'*i koodijupi, mis võttis vastu saabuvaid UDP sõnumeid. Seeläbi sai jälgida, kas ja mis kujul saadetud info kohale jõuab. Esialgu edastas autor andmeid ROS'i sõlmelt kohalikku masinasse (*localhost*) ehk IP aadressiks määrati 127.0.0.1. Joonis 4 kuvab vastuvõetud UPD sõnumit aku laetuse taseme kohta, mis sisaldab korrektset infot, see tähendab samu andmeid, mida eelnevalt käsurealt avaldatud teema edastas.

```
krista@krista-VirtualBox:~/udp receiver$ python "/home/krista/udp receiver/receiver test.py"
2018-05-22 14:54:55,977 udp server      INFO   Listening on udp 127.0.0.1:5005
2018-05-22 14:55:01,514 udp server     DEBUG   '{"battery percentage": 62.0, "msg name": "current battery", "time": "2018-05-22 14:55
:01.436583"}'
```

Joonis 4. Vastuvõetud UDP sõnum aku laetuse kohta.

Olles veendunud, et UDP sõnumite saatmine ja kuulamine toimib, käivitas autor kohalikus masinas ROS'i sõlme, mis tegeleb iseauto poolt saadetud andmete vastuvõtmisega ning nende avaldamisega serveri ROS'is.

Kõigepealt testiti, kas käesolevas sõlmes saabunud kaamerapildi tükkidest kokku pandud pildi avaldamine toimib. Selle käigus tulid välja mitmed vead koodis. Näiteks ei saadetud esialgu serverisse pildi metaandmeid, mis on aga tegelikult vajalikud soovitud kaamerapildi avaldamiseks ROS'is. Oli ka probleem, et omavahel võrreldi int ja tuple tüüpi muutujaid, mistõttu ebaõnnestus samaväärsuse tuvastamine. Pärast pikka testimist, silumist ja paranduste tegemist, õnnestus jõuda soovitud tulemuseni. ROS'is avaldati edukalt Image tüüpi kaamerapildi teemat ning selle põhjal jooksis RViz'is samasugune kaamerapilt, mis algselt *rosbag*'is oleva kaamerapildi teema põhjalgi. Joonis 5 kujutab kuvatõmmist RViz'ist, kui seal parajasti jooksis serveri ROS'i poolt avaldatud teema põhjal loodud kaamerapilt.



Joonis 5. UDP sõnumitest avaldatud teema põhjal kuvatud kaamerapilt RVizis

Olles läbinud eelpool toodud testi edukalt, kontrollis autor, et ka teised andmed korrektselt serveri sõlme jõuaksid ning et toimiks edukalt ka nende avaldamine ROS'is. Kuna tegemist on tunduvalt väiksemate sõnumitega, siis sai neid testida käsuga *rostopic echo <topic_name>*, mis trükib konsooli vastava teema iga kord, kui toimub selle avaldamine. Selle testi puhul ilmnes vaid üks viga koodis: teatud väljade valimine JSON'ist toimus valesti. Kiire muudatus koodis parandas käesoleva vea, misjärel täheldati, et kõikide saabuvate sõnumite kuulamine ja ROS'is avaldamine toimis korrektselt.

Läbitud testidest oli näha, et andmeedastuskiht toimib, kui seda katsetada kohalikus masinas. Järgnevalt testis autor andmete kuulamist ja ROS'is avaldamist reaalselt kasutusel olevas Ubuntu serveris. Selleks paigaldas autor esmalt serverisse ROS'i ning käesoleva diplomitöö raames loodud serveri sõlme. Lisaks tuli suurendada lubatud UDP sõnumi suurust ning avada pordid, kuhu saabuvad iseauto sõlme pealt UDP sõnumid. Kasutades käsureal sama käsku, mis eelmises testis, nägi autor, et andmed jõuavad serverisse ning need avaldatakse serveri ROS'i. Eesmärk saada andmed iseauto pealt veebiserverisse on seega täidetud, mistõttu võib antud testi lugeda läbituks. Kaheldav oli aga serveri töö stabiilsus. Kohati tekkis anomaaliaid, kus näiteks server ei tegelenud pärast teatud arvu kaamerapildi loomist enam ülejäänute saabuvate piltide UDP sõnumitega. Kuna lokaalselt testides toimis edastamine probleemideta, võib järeldada, et käesoleva probleemi põhjusteks on serveri seadistused. Järgmised sammud enne liidese ehitamist oleks seega serveri konfiguratsiooni lihvimine, et tagada serveri suurem stabiilsus.

Kokkuvõtlikult arvab autor, et diplomitöö raames loodud andmeedastuskiht läbis testid edukalt.

4 Kokkuvõte

Käesoleva diplomitöö eesmärk oli luua andmeedastuskiht TTÜ isejuhtiva auto ja veebiserveri vahel, võimaldades seeläbi usaldusväärse isejuhtiva sõiduki kasutajaliidese loomise.

Töö esimeses osas tutvustas autor diplomitöö aluseks olevat TTÜ isejuhtivat autot ning seal kasutusel olevat tarkvara: ROS, Autoware, RViz. Kirjeldati mujal maailmas isejuhtivatele sõidukitele loodud kasutajaliideseid ning võrreldi erinevaid võimalusi soovitud liidese loomiseks. Lisaks püstitati nõuded andmeedastuskihile ning täpsustati edastatavad andmed ning nende edastamise viis.

Diplomitöö teises ehk tehnilise teostuse osas kirjeldas autor tarkvara loomise ja testimise protsessi ning näitas, et analüüsi põhjal valmis soovitud lahendus. Valminud andmeedastuskihi poolt TTÜ iseautolt veebiserverisse saadetavad andmed on auto positsioon, kiirus, aku laetuse tase, parkimisanduri näit ning kaamerapilt. Tulevikus on võimalik vajadusel lisada veel muid edastatavaid andmeid.

Autori hinnangul vastab diplomitöö tulemus töö alguses püstitatud nõuetele. Töö käigus realiseeritud lahendus on hea alus TTÜ isejuhtiva auto kasutajaliidese ehitamiseks. Loodud andmeedastuskiht tagab, et tulevases iseauto kasutajaliideses kuvatakse korrektseid andmeid. Lisaks on tänu antud töö analüüsi osale teada, mis tehnoloogiatega oleks kõige mõistlikum kasutajaliidest luua.

Kasutatud kirjandus

[1] R. Vellerind, „Avatud robotiarendusplatvormi ROS võimekuse loomine,“ 2017. [Võrgumaterjal]. Available: http://mobile.dspace.ut.ee/bitstream/handle/10062/56559/Vellerind_BA2017.pdf?sequence=1&isAllowed=y. [Kasutatud 09 05 2018].

[2] Open Source Robotics Foundation, „rviz,“ 15 06 2016. [Võrgumaterjal]. Available: http://wiki.ros.org/rviz. [Kasutatud 25 04 2018].

[3] Open Source Robotics Foundation, „Wviz,“ 31 08 2012. [Võrgumaterjal]. Available: http://wiki.ros.org/wviz. [Kasutatud 25 04 2018].

[4] A. Vill, „Linnaleht,“ 01 06 2017. [Võrgumaterjal]. Available: https://www.linnaleht.ee/808551/eesti-esimene-oma-isesoitja-alustab-juba-tuleval-suvel. [Kasutatud 25 03 2018].

[5] M. Kingsepp, „TTÜ Iseauto blogi - I etapp,“ Tallinna Tehnikaülikool, 15 09 2017. [Võrgumaterjal]. Available: http://iseauto.ttu.ee/blog/page/2/. [Kasutatud 09 05 2018].

[6] Open Source Robotics Foundation, „About ros,“ 2016. [Võrgumaterjal]. Available: http://www.ros.org/about-ros/. [Kasutatud 24 04 2018].

[7] Open Source Robotics Foundation, „ROS Kinetic,“ 08 01 2018. [Võrgumaterjal]. Available: http://wiki.ros.org/kinetic. [Kasutatud 24 04 2018].

[8] Open Source Robotics Foundation, „ROS Concepts,“ 21 06 2014. [Võrgumaterjal]. Available: http://wiki.ros.org/ROS/Concepts. [Kasutatud 24 04 2018].

[9] Open Source Robotics Foundation, „ROS Bags,“ 02 05 2015. [Võrgumaterjal]. Available: http://wiki.ros.org/Bags. [Kasutatud 25 04 2018].

[10] S. Kato, „Autoware wiki,“ 26 03 2018. [Võrgumaterjal]. Available: https://github.com/CPFL/Autoware/wiki. [Kasutatud 25 04 2018].

[11] A. Noakes, „Volkswagen unveils Sedric, its first fully autonomous vehicle,“ 03 09 2017. [Võrgumaterjal]. Available: https://arstechnica.com/cars/2017/03/volkswagen-unveils-sedric-its-first-fully-autonomous-vehicle/. [Kasutatud 25 04 2018].

[12] EasyMile, „EasyMile's solution An Autonomous Road Transport Solutions,“ [Võrgumaterjal]. Available: https://ruter.no/globalassets/kollektivanbud/moter/2017-01-12-autonomous-transport/easymile_oslo_ruter_light.pptx?id=11194. [Kasutatud 25 04 2018].

[13] D. Etherington, „Waymo focuses on user experience, considers next steps,“ 31 10 2017. [Võrgumaterjal]. Available: https://techcrunch.com/2017/10/31/waymo-self-driving-ux/. [Kasutatud 25 04 2018].

[14] R. Toris, J. Kammerl, D. Lu, J. Lee, O. C. Jenkins, S. Osentoski, M. Wills ja S. Chernova, „Robot Web Tools,“ [Võrgumaterjal]. Available: http://robotwebtools.org/. [Kasutatud 25 04 2018].

[15] R. Toris, „Ros3djs,“ 27 03 2015. [Võrgumaterjal]. Available: http://wiki.ros.org/ros3djs. [Kasutatud 25 04 2018].

[16] Robot Web Tools, „Class: PointCloud2,“ 04 01 2018. [Võrgumaterjal]. Available: http://robotwebtools.org/jsdoc/ros3djs/current/ROS3D.PointCloud2.html. [Kasutatud 25 04 2018].

[17] M. Rouse ja G. Lawton, „TechTarget,“ [Võrgumaterjal]. Available: https://searchnetworking.techtarget.com/definition/UDP-User-Datagram-Protocol. [Kasutatud 18 04 2018].

Lisa 1 – Iseauto ROS sõlme lähtekood

```
#!/usr/bin/env python

#
# Sends data needed for the interface to web server via UDP
#

import rospy
import socket
import json
import zlib
import struct
from datetime import datetime

from geometry_msgs.msg import PoseStamped
from geometry_msgs.msg import TwistStamped

from sensor_msgs.msg import Image

from iseauto_msgs.msg import BatteryInfoStamped
from iseauto_msgs.msg import ParkingSensorStamped

UDP_IP = "193.40.231.233"
IMAGE_ID = 1

def pose_callback(data):
    message = json.dumps({'msg_name': 'current_pose',
                          'time':
datetime.fromtimestamp(data.header.stamp.to_sec()).__str__(),
                          'pose': {
                               'position': {
                                    'x': data.pose.position.x,
                                    'y': data.pose.position.y,
                                    'z': data.pose.position.z
                               },
                               'orientation': {
                                    'x': data.pose.orientation.x,
                                    'y': data.pose.orientation.y,
                                    'z': data.pose.orientation.z,
                                    'w': data.pose.orientation.w
                               }
                           }
                          })
    send_udp_message(message, 11201)
```

```
def velocity_callback(data):
    message = json.dumps({'msg_name': 'current_velocity',
                          'time':
datetime.fromtimestamp(data.header.stamp.to_sec()).__str__(),
                          'twist': {
                              'linear': {
                                  'x': data.twist.linear.x,
                                  'y': data.twist.linear.y,
                                  'z': data.twist.linear.z
                              },
                              'angular': {
                                  'x': data.twist.angular.x,
                                  'y': data.twist.angular.y,
                                  'z': data.twist.angular.z
                              }
                          }
                          })
    send_udp_message(message, 11202)

def image_callback(message):
    global IMAGE_ID
    compressed_msg = zlib.compress(message.data)
    length = len(compressed_msg)
    chunk_size = 6400
    chunks_amount = length / chunk_size + 1
    send_image_info(chunks_amount, message)
    chunk_counter = 1
    for i in range(0, length, chunk_size):
        chunk = struct.pack(">I", IMAGE_ID) + struct.pack(">H",
chunk_counter) + compressed_msg[i:i+chunk_size]
        send_udp_message(chunk, 11200)
        chunk_counter += 1
    IMAGE_ID = IMAGE_ID + 1

def send_image_info(chunks_amount, message):
    message = json.dumps({'msg_name': 'image info',
                          'image_id': IMAGE_ID,
                          'height': message.height,
                          'width': message.width,
                          'encoding': message.encoding,
                          'is_bigendian': message.is_bigendian,
                          'step': message.step,
                          'amount_of_chunks': chunks_amount
                          })
    send_udp_message(message, 11200)

def battery_callback(data):
    message = json.dumps({'msg_name': 'current_battery',
                          'time':
datetime.fromtimestamp(data.header.stamp.to_sec()).__str__(),
```

```
                        'battery_percentage': data.batteryInfo.percentage
                    })
    send_udp_message(message, 11203)

def parking_sensor_callback(data):
    message = json.dumps({'msg_name': 'parking_sensor',
                        'sensor_name': data.parkingSensor.sensorName,
                        'time':
datetime.fromtimestamp(data.header.stamp.to_sec()).__str__(),
                        'sensor_distance': data.parkingSensor.distance
                    })
    send_udp_message(message, 11204)

def send_udp_message(message, port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.sendto(message, (UDP_IP, port))

rospy.init_node("server_gateway", anonymous=True)
CAMERA_TOPIC = rospy.get_param('~camera_topic')
rospy.Subscriber("/current_pose", PoseStamped, pose_callback)
rospy.Subscriber("/current_velocity", TwistStamped, velocity_callback)
rospy.Subscriber(CAMERA_TOPIC, Image, image_callback)
rospy.Subscriber("/current_battery", BatteryInfoStamped, battery_callback)
rospy.Subscriber("/parking_sensor1", ParkingSensorStamped,
parking_sensor_callback)

def main():
    rospy.spin()

if __name__ == "__main__":
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```

Lisa 2 – Serveri ROS sõlme lähtekood

```
#!/usr/bin/env python

#
# Listens to UDP messages and publishes them as ROS messages
#

import rospy
import socket
import json
import zlib
import struct
import threading
import time

from geometry_msgs.msg import PoseStamped
from geometry_msgs.msg import TwistStamped

from sensor_msgs.msg import Image

from iseauto_msgs.msg import BatteryInfoStamped
from iseauto_msgs.msg import ParkingSensorStamped

images = dict()

class ImageChunks:
    def __init__(self):
        self.amount_of_chunks = -1
        self.chunks = []

    def is_ready(self):
        return self.amount_of_chunks == len(self.chunks)

    def add_chunk(self, seq, new_chunk):
        chunk = Chunk(seq, new_chunk)
        self.chunks.append(chunk)

    def set_metadata(self, json_msg):
        self.amount_of_chunks = json_msg['amount_of_chunks']
        self.height = json_msg['height']
        self.width = json_msg['width']
        self.encoding = json_msg['encoding']
        self.is_bigendian = json_msg['is_bigendian']
        self.step = json_msg['step']


    def get_full_image(self):
        imageList = sorted(self.chunks, key=lambda x: x.seq)
```

```
        image = b''
        for x in imageList:
            image += x.data
        return zlib.decompress(image)

class Chunk:
    def __init__(self, seq, data):
        self.seq = seq
        self.data = data

def udp_server(host='193.40.231.233', port=1234):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

    sock.bind((host, port))
    while True:
        (data, addr) = sock.recvfrom(128*1024)
        yield data

def start_image_udp_server():
    for data in udp_server(port=11200):
        udp_image_callback(data)

def start_velocity_udp_server():
    for data in udp_server(port=11202):
        udp_velocity_callback(data)

def start_pose_udp_server():
    for data in udp_server(port=11201):
        udp_pose_callback(data)

def start_battery_udp_server():
    for data in udp_server(port=11203):
        udp_battery_callback(data)

def start_parking_sensor_udp_server():
    for data in udp_server(port=11204):
        udp_parking_sensor_callback(data)

def udp_image_callback(data):
    global images
    try:
        json_data = json.loads(data)
        image_id = json_data['image_id']

        if (image_id not in images):
            images[image_id] = ImageChunks()
            delete_after_five_seconds(image_id)
        images[image_id].set_metadata(json_data)

        if(images[image_id].is_ready()):
```

```
            publish_image_to_ROS(images[image_id])
            del images[image_id]
    except ValueError:
        image_id = struct.unpack(">I", data[:4])[0]
        seq = struct.unpack(">H", data[4:6])[0]
        chunk = data[6:]

        if (image_id not in images):
            images[image_id] = ImageChunks()
            delete_after_five_seconds(image_id)
        images[image_id].add_chunk(seq, chunk)

        if(images[image_id].is_ready()):
            publish_image_to_ROS(images[image_id])
            del images[image_id]

def deleting_thread(image_id):
    time.sleep(5)
    if image_id in images:
        del images[image_id]

def delete_after_five_seconds(image_id):
    thread6 = threading.Thread(target=deleting_thread, args=[image_id])
    thread6.start()

def udp_velocity_callback(data):
    json_data = json.loads(data)
    publish_velocity_to_ROS(json_data)

def udp_pose_callback(data):
    json_data = json.loads(data)
    publish_pose_to_ROS(json_data)

def udp_battery_callback(data):
    json_data = json.loads(data)
    publish_battery_to_ROS(json_data)

def udp_parking_sensor_callback(data):
    json_data = json.loads(data)
    publish_parking_sensor_to_ROS(json_data)

def publish_image_to_ROS(image):
    ros_image = Image()
    ros_image.data = image.get_full_image()
    ros_image.height = image.height
    ros_image.width = image.width
    ros_image.encoding = image.encoding
    ros_image.is_bigendian = image.is_bigendian
    ros_image.step = image.step
    image_publisher.publish(ros_image)
```

```
def publish_velocity_to_ROS(velocity):
    ros_velocity = TwistStamped()
    ros_velocity.twist.linear.x = velocity['twist']['linear']['x']
    ros_velocity.twist.linear.y = velocity['twist']['linear']['y']
    ros_velocity.twist.linear.z = velocity['twist']['linear']['z']
    ros_velocity.twist.angular.x = velocity['twist']['angular']['x']
    ros_velocity.twist.angular.y = velocity['twist']['angular']['y']
    ros_velocity.twist.angular.z = velocity['twist']['angular']['z']
    velocity_publisher.publish(ros_velocity)

def publish_pose_to_ROS(pose):
    ros_pose = PoseStamped()
    ros_pose.pose.position.x = pose['pose']['position']['x']
    ros_pose.pose.position.y = pose['pose']['position']['y']
    ros_pose.pose.position.z = pose['pose']['position']['z']
    ros_pose.pose.orientation.x = pose['pose']['orientation']['x']
    ros_pose.pose.orientation.y = pose['pose']['orientation']['y']
    ros_pose.pose.orientation.z = pose['pose']['orientation']['z']
    ros_pose.pose.orientation.w = pose['pose']['orientation']['w']
    pose_publisher.publish(ros_pose)

def publish_battery_to_ROS(data):
    ros_battery = BatteryInfoStamped()
    ros_battery.batteryInfo.percentage = data['battery_percentage']
    battery_publisher.publish(ros_battery)

def publish_parking_sensor_to_ROS(data):
    ros_parking_sensor = ParkingSensorStamped()
    ros_parking_sensor.parkingSensor.distance = data['sensor_distance']
    parking_sensor_publisher.publish(ros_parking_sensor)

rospy.init_node("server", anonymous=True)
image_publisher = rospy.Publisher("test_image", Image, queue_size=10)
velocity_publisher = rospy.Publisher("test_velocity", TwistStamped,
queue_size=10)
pose_publisher = rospy.Publisher("test_pose", PoseStamped, queue_size=10)
battery_publisher = rospy.Publisher("test_battery", BatteryInfoStamped,
queue_size=10)
parking_sensor_publisher = rospy.Publisher("test_parking_sensor",
ParkingSensorStamped, queue_size=10)

thread1 = threading.Thread(target=start_image_udp_server)
thread2 = threading.Thread(target=start_velocity_udp_server)
thread3 = threading.Thread(target=start_pose_udp_server)
thread4 = threading.Thread(target=start_battery_udp_server)
thread5 = threading.Thread(target=start_parking_sensor_udp_server)
thread1.start()
thread2.start()
thread3.start()
thread4.start()
thread5.start()
```

```
def main():
    rospy.spin()

if __name__ == "__main__":
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```