VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
INFORMATIKOS INSTITUTAS

Formalių specifikacijų taikymas projektuojant išskirstytas sistemas

Applying Formal Specifications to Design Distributed Systems

Magistro darbo planas

Atliko:	Matas Savickis	(parašas)
Darbo vadovas:	Karolis Petrauskas, Doc., Dr.	(parašas)
Recenzentas:	Valaitis Vytautas, Asist., Dr.	(parašas)

Vilnius – 2021

TURINYS

Įvadas

Šiais laikais kai kurios programų sistemos yra išskirstytos [**mcr**]. Tokios sistemos, kaip sufleruoja pavadinimas, yra kuriamos išskirstant skaičiavimo mazgus į atskiras, savarankiškas dalis [**coulouris2005distributed**]. Ne kaip monolitinės sistemos, išskirstytos sistemos gali veikti skirtingose serverinėse, kurios gali būti įvairiose geografinėse vietose [**shirriff2006method**]. Toks sistemos išskirstymas pasižymi šiomis savybėmis:

1. Pasiekiamumas (angl. *Availability*) – vartotojas gali pasiekti sistemą bet kuriuo metu [**180327**].

2. Patvarumas (angl. *Durability*) – išskirstytos sistemos užtikrina duomenų išlaikymą ir pastovų veikimą net jeigu ir vienas iš paskirstytos sistemos mazgų nustotų veikti dėl sistemos sutrikimų, sukeltų programos klaidų arba gamtos katastrofų, tokių kaip: gaisrai, potvyniai ir panašios nelaimės. Ši savybė taip pat užtikrina sistemos gebėjimą atsistatyti ir neprarasti duomenų po minėtų įvykių [**5470366**].

3. Plečiamumas (angl. *Scalability*) – didėjant vartotojų skaičiui bei programų sistemos kompleksiškumui išskirstytos sistemos užtikriną vertikalų (padidinti mazgo techninės įrangos galingumą) bei horizontalų (padidinti mazgų skaičių sistemoje) plečiamumą [**862209**].

4. Našumas (angl. *Efficiency*) – sistemos vartotojų skaičius dažniausiai būna nepastovus, jis kinta dienos metu arba atitinkamais metų periodais. Išskirstytos sistemos padeda užtikrinti našų įrangos resursų naudojimą sumažinant įrangos galingumą (kai vartotojų skaičius yra nedidelis) bei padidinant galingumą (kai sistemos apkrova padidėja).

Šiuo metu yra sukurta keletas atviro kodo išskirstytų sistemų, padedančių apdoroti duomenis realaus laiku (ActiveMQ [**snyder2011activemq**], Apache Spark [**spark2018apache**], Apache Storm [**iqbal2015big**]). Šio tipo sistemos naudojamos apdoroti nesąžiningus piniginius sandorius finansiniame sektoriuje [**fraud**], vartotojų veiksmams interneto puslapyje sekti [**tracking**], IoT įrenginių duomenims valdyti [**iot**], bei kitais atvejais, kai didelio kiekio duomenų apdorojimas realiu laiku yra svarbu. Viena iš tokių išskirstytų sistemų yra Apache Kafka (toliau – Kafka) [**kfk**].

Kafka buvo pradėta kurti kompanijos LinkedIn [**kfk**]. 2012 metais Kafka sistema buvo perduota Apache Software Foundation tolesniam vystymui. Šiuo metu Kafka platforma yra žinučių siuntimo sistema, kuri pasižymi lengvu plečiamumu, patvarumu, patikimumu ir greičiu. Duomenys Kafka platformoje yra išsaugomi saugiu, trukdžiams atspariu būdu. Kafka kūrėjų teigimu, šiuo metu platformą naudoja daugiau negu 80 procentų didžiausių Jungtinių Amerikos Valstijų įmonių [**kfk**]. Kafka platforma yra plačiai naudojama įvairiose srityse, tokiose kaip: žurnalistika, debesijos paslaugos, muzikos srauto paslaugos, telekomunikacijos, bankinės paslaugos ir daugelis kitų [**kfk**].

Šiuo metu Kafka naudoja Apache ZooKeeper saugyklą [**junqueira2013zookeeper**], kurioje yra laikomi metaduomenys apie mazgų vieta išskirstytame tinkle ir mazgų konfigūracija. Minėti duomenys yra laikomi ne pačioje Kafka platformoje, bet atskirame ZooKeeper klasteryje. Apache

Kafka komanda nori panaikinti priklausomybę nuo Apache ZooKeeper ir valdyti metaduomenis pačioje Kafka platformoje [**metadata**]. Šiuo metu Kafka kūrėjai yra pradėje įgyvendinti savarankiškai tvarkomą metaduomenų kvorumą kurio pagrindinė dalis bus į Raft protokolą panaši susitarimo algoritmo realizacija pritaikyta pagal Kafka sistemą kuris bus naudojamas replikuoti metaduomenims [**qourum**]. Pats metaduomenų kvorumas ilgainiui pakeis Apache ZooKeeper Kafka platformoje.

Norėdami užtikrinti Kafka platformos kokybę, kūrėjai yra įgyvendinę skirtingų testų [**kfkGH**]. Testai padeda atskleisti programos klaidas arba pasakyti ar naujas kodas nepaveikė seniau parašyto funkcionalumo [**819971**]. Tačiau net ir laikantis geruųjų testavimo praktikų nepavyksta išvengti programos klaidų. Net ir paskyrus daugiau resursų testavimui sudėtinguose sistemose, tokiose kaip Kafka, pilnas sistemos testavimas yra neįmanomas [**sullivan2004software**]. Norint rasti subtilesnius sutrikimus pačioje sistemos architektūroje, tokius kaip dalinis mazgų neveikimas, lygiagrečių algoritmų klaidos naudojant keletą procesų, gedimų atsparumo ir atsistatymo po gedimo algoritmų klaidos bei kitiems kraštutiniams veikimo scenarijams [**newcombe2014use**] rasti ir išspręsti turime ieškoti kitų būdų. Vienas iš jų formalios specifikacijos.

Formalios specifikacijos yra matematinės technikos skirtos apibūdinti sistemų elgseną ir padėti kuriant jas naudojant griežtas ir veiksmingas priemones [**holzmann1995improvement**]. Turint sistemos formalią specifikaciją galima ja pasinaudoti vykdant formalų verifikavimą ir parodant, kad algoritmas yra adekvatus pagal sukurtą specifikaciją. Sudarinėti formalią sistemos specifikaciją galima ir nepradėjus įgyvendinti sistemos, turint tik jos architektūrą. Formaliai verifikuota specifikacija suteikia informacijos apie architektūros korektiškumą ir įgalina objektyviai koreguoti sistemos architektūrą dar prieš pradedant ją įgyvendinti. Formalios specifikacijos sudaromos pasinaudojant tam tikromis kalbomis arba įrankiais. Viena iš tokių formalaus specifikavimo kalbų yra TLA$^+$ [**lamport2002specifying**].

TLA$^+$ yra formalios specifikacijos kalba, kurią sukūrė Leslie Lamport [**lamport2002specifying**]. Leslie Lamport 1980 metais sukūrė laiko veiksmų logiką (angl. *Temporal Logic of Actions*) [**10.1145/177492.177726**] pasinaudodamas Amir Pnueli 1977 metais sukurta laiko logika (angl. *Temporal Logic*) [**4567924**]. 1999 metais Leslie Lamport naudodamasis laiko veiksmų logika sukūrė formalaus specifikavimo kalbą TLA$^+$ [**lamport2002specifying**].

TLA$^+$ kalba yra skirta kurti konkurencinių ir išskirstytų sistemų formalias specifikacijas ir jas verifikuoti. Naudojant TLA$^+$ galima specifikuoti šias išskirstytų sistemų savybes [**lamport2019safety**]:

1. Gyvumas – geri dalykai įvyksta programos vykdymo metu. Sistema galiausiai atliks jai paskirtą užduotį arba pateks į norimą būseną.

2. Saugumas – blogi dalykai neatsitiks programos vykdymo metu. Sistema nesustos veikti dėl netikėtai iškilusios klaidos.

Kadangi TLA$^+$ specifikacijos yra rašomos formalia kalba, tai leidžia patikrinti sukurtos specifikacijos saugumo ir gyvumo savybes.

Šias savybes mes galime patikrinti naudodamiesi TLC Model Checker (modelio tikrintoju).

TLC yra išreikštinės būsenos (explicit-state) modelio tikrintojas, kurio paskirtis yra palaipsniui pasiekti visas galimas sistemos būsenas pagal nurodytą formalią specifikaciją [**yu1999model**]. Tačiau kartais pagal sukurtą formalią specifikaciją susidaro labai daug būsenų, kurias sistema gali pasiekti, todėl tampa nepraktiška naudoti TLC. Tokiu atveju galime naudotis TLA$^+$ specifikacijos įrodymo sistema TLAPS [**cousineau2012tla+**]. TLAPS yra įrodymų sistema skirta patikrinti TLA$^+$ įrodymus. Šios sistemos paskirtis yra patikrinti pateiktus teoremų įrodymus. Įrodžius teoremą laikoma, kad TLA$^+$ specifikacija yra korektiška.

Yra ir kitų formalaus specifikavimo kalbų kurias galėtume naudoti šiame darbe. Viena iš jų Z formalaus specifikavimo kalba [**O'Regan2017**], kuri sėkmingai buvo naudota specifikuoti UNIX failų sistemai [**bowen1996formal**], bei Oxfordo universiteto paskirstytų skaičiavimų projekte [**bowen1996formal**]. Dar viena formalaus specifikavimo kalba yra VDA [**bjorner1978vienna**], kuria buvo specifikuoti bendros atminties sinchronizavimo algoritmai [**Slaats1998**]. Tačiau šiam darbui buvo pasirinkta TLA$^+$ kalba dėl jos pritaikymo išskirstytoms sistemoms naudojant būsenų mašiną [**lamport2002specifying**], esamų sėkmingo pritaikymo pavyzdžių specifikuojant išskirstytas sistemas [**newcombe2014use**; **kfkTla**; **rafttla**; **jiryu2020extreme**] bei gausaus TLA$^+$ įrankių pasirinkimo.

Tačiau parašyti formalią specifikaciją yra negana. Kartais gali nutikti taip, kad algoritmo realizacija neatitiks jo reikalavimų. Mūsų sudaryta specifikacija gali būti adekvati pagal pateiktus reikalavimus, tačiau sistemos kūrėjai šiuos reikalavimus gali įgyvendinti nekorektiškai. Norint to išvengti turime patikrinti ar sistemos įgyvendinimas atitinka specifikaciją. Yra skirtingų būdų patikrinti ar specifikacija atitinka realizacija: specifikacijos ir realizacijos rašymas ta pačia programavimo kalba [**kern1999formal**], programinio kodo generavimas naudojantis specifikacija [**houhou2017framework**], testų generavimas naudojant specifikaciją [**utting2010practical**]. Mūsų darbo atveju šie metodai netinka, nes realizuotas kodas jau bus parašytas Apache Kafka kūrėjų, o testų generavimas išskirstytos sistemos algoritmui būtų per daug sudėtingas dėl nedeterminuoto algoritmo veikimo laiko atžvilgiu [**davis2020extreme**]. Dėl šių priežasčiu rinksimės kitą formalaus verifikavimo metodą – Modeliu paremta pėdsakų tikrinimą (angl. *Model-Based Trace-Checking*) [**howard2011modelbased**]. Metode aprašomi šie žingsniai:

1. Įgyvendinti programą ir paprastus testus.
2. Pridėti kodą, kuris sektų programos būseną ir įrašytų ją į failą.
3. Sukurti formalią specifikaciją parašytam kodui.
4. Perleisti sistemos būsenos failą per įrankius skirtus patikrinti ar sistemos būsenos failo duomenys yra adekvatūs pagal formalią specifikaciją.

Šiame darbe pirmą žingsnį praleisime, nes specifikuosime jau įgyvendintus algoritmus Kafka platformoje. Antrame žingsnyje pridėsime kodą, kuris registruos sistemos būseną ir įrašinės ją į tekstinius failus. Trečiame žingsnyje sukursime formalią specifikaciją naudodamiesi TLA$^+$. Ketvirtame žingsnyje, naudodami TLC, patikrinsime ar sistemos būsenos failo duomenys yra adekvatūs pagal sukurtą formalią specifikaciją. Apdorosime būsenos duomenis ir pasinaudodami TLC tikrinsime ar tokią būseną galima pasiekti pagal mūsų sukurtą specifikaciją.

Šis metodas mums leis patikrinti algoritmo sukurtus pėdsakus po programos vykdymo išvengiant nedeterminuoto algoritmo vykdymo kuris dažniausiai kyla išskirstytose sistemose, kuomet skirtingi mazgai baigia darbą nevienodu metu. Taip pat šis metodas jau buvo sėkmingai taikytas verifikuojant kitas išskirstytų sistemų specifikacimas [**davis2020extreme**].

Temos aktualumas bei naujumas

Viena iš formalių specifikacijų ir TLA$^+$ panaudojimo industrijoje sėkmės istorijų yra Amazon Web Service (AWS) komandos 2014 metais išleistas straipsnis [**newcombe2014use**]. Straipsnyje rašoma, kad AWS komanda naudojo TLA$^+$ sudarant formalias specifikacijas dešimtyje projektų. Tuo metu AWS turėjo 7 komandas, kurios naudojosi TLA$^+$ kurdamos naujas programų sistemas. AWS sistemos specifikavimo metu buvo surasta 10 iki šiol neatrastų sisteminių klaidų, kurių atradimas ir pasiūlyti ištaisymai atskleidė tolimesnes sistemos klaidas, kurios taip pat buvo ištaisytos. Straipsnyje įvardinta ir kita, netiesioginė nauda gauta formaliai specifikuojant sistemas: pagerėjęs bendras sistemos suvokimas, padidėjęs produktyvumas ir inovacijos.

Dar viena sėkmės istorija yra 2018 metais Kafka Summit konferencijoje pristatyta Kafka TLA$^+$ formali specifikacija, kurią sukūrė Jason Gustafson [**kfkTla**]. Pristatyme buvo parodyta kaip pritaikant TLA$^+$ bei specifikuojant Kafka duomenų replikavimo algoritmą buvo surastos ir pataisytos 3 retai atsitinkančios programos klaidos. Taisant taip pat rastos ir pataisytos dar kelios klaidos.

Šiame darbe specifikuosime ir verifikuosime Kafka kūrėjų pasiūlytą Raft algoritmo [**10.1145/2723872.2723876**] realizaciją [**raftimpl**]. Raft protokolas yra skirtas pasiekti susitarimą išskirstytoje sistemoje. Norint pasiekti susitarimą tarp sistemos mazgų, kiekvienas mazgas turi turėti lyderio arba sekėjo rolę. Algoritme lyderis yra atsakingas už informacijos replikavimą savo sekėjams. Lyderis tam tikru metu praneša sekėjams apie savo egzistavimą. Jeigu sekėjas nesulaukia signalo iš lyderio, sistemoje prasideda naujo lyderio rinkimas.

Nors baziniam Raft algoritmui jau yra sukurta formalių specifikacijų TLA$^+$ kalba [**rafttla**], tačiau pasiūlyta realizacija skiriasi nuo iki šiol sukurtų specifikacijų, todėl reiktų sukurti atskirą formalią specifikaciją bei ją verifikuoti.

Kafka sutrikimų sekimo sistemoje yra aprašoma ir kitų Kafka sutrikimų [**kfkis**] , kuriuos būtų galima formaliai specifikuoti ir verifikuoti taip parodant algoritmų problemas. Ši informacija galėtų padėti sistemos kūrėjams ištaisyti Kafka sutrikimus. Sutrikimai, kuriuos būtų galima formaliai specifikuoti ir verifikuoti:

- Kafka Valdiklis neišsijungia teisingai kai įvyksta techninis gedimas [**kfkBug**]
- Lenktynių sąlyga klasėje FindCoordinatorFuture visam laikui nutraukia sąsaja su grupės koordinatoriumi [**kfkistwo**]
- Lengtyvių sąlyga gali sukelti atsilikimą kitoms aktyvioms užduotims [**kfkisthr**]

Visi šie pateikti sutrikimai buvo įvardinti kaip kritiniai ir nepradėti taisyti, todėl papildoma informacija gauta formaliai specifikuojant juos galimai padėtu išspręsti šias problemas.

Iki šiol, kiek yra žinoma, Kafka platforma neakademiniame kontekste buvo specifikuota tik vieną kartą [**kfkTla**], o sukurta specifikacija atnešė naudos padedant surasti sistemos klaidas. Panašią mokslininkų sėkmę matome ir Amazon Web Service formalios specifikacijos sudarymo tyrimuose [**newcombe2014use**]. Dėl papildomų Kafka formalių specifikacijų stokos ir praeityje pasisekusio formalaus specifikavimo išskirstytuose sistemose manome, kad papildomi tyrimai Kafka platformoje atneštų naudos surandant algoritmų klaidas arba užtikrinant, kad specifikuotose algoritmuose jų nėra. Šiuo metu Kafka sisteminių klaidų registre [**kfkissue**] yra išspręstų ir neišspręstų klaidų, kurių verifikavimas padėtų atskleisti naujas klaidas arba įrodytų, kad klaidos ištaisytos adekvačiai. Kafka platforma turi daug naudotojų [**kfk**], todėl tolimesnis kokybės užtikrinimas Kafka platformoje atneštų naudą.

Kurti specifikacijas Kafka platformose naudojamiems algoritmams gali būti naudinga ir didesnei aibei sistemų. Sėkmingai specifikavus algoritmus, naudojamus Kafka platformoje, būtų galima įrodyti adekvatumą daug didesnei išskirstytų sistemų aibei, kurioje yra naudojami tokie pat algoritmai. Šiuo metu yra straipsnių, kuriuose formaliai verifikuojami išskirstytų sistemų algoritmai [**lamport2005generalized**]. Jie yra naudojami kurti išskirstytas sistemas, todėl yra tikimasi, kad panašius rezultatus pavyks pasiekti specifikuojant Kafka platformos algoritmus.

Darbo tikslas

Parodyti Apache Kafka algoritmų korektiškumą naudojantis formaliu specifikavimu, bei įvardinti problemas specifikuotose algoritmuose.

Uždaviniai

1. Formaliai specifikuoti pasirinktus Kafka platformos algoritmus naudojant TLA$^+$ specifikavimo kalbą. Šiame žingsnyje formaliai aprašysime pasirinktus algoritmus ir jų savybes, kad galėtume patikrinti jų korektiškumą.

2. Verifikuoti ar pagal sukurtą specifikaciją Kafka platforma veikia korektiškai. Verifikacija bus atliekama naudojant modeliu paremtu pėdsakų tikrinimo (angl. *Model-Based Trace-Checking*) metodu.

3. Esant poreikiui įrodyti specifikacijos savybes naudojant TLAPS. Šios užduoties prireik, jeigu formalios specifikacijos tikrinimas TLC modelio tikrintoju užtruktų per daug laiko.

4. Surasti kitas paskirstytas sistemas, kuriose yra naudojami šiame darbe formaliai specifikuoti algoritmai. Šiuo uždaviniu parodytume, kad šis darbas būtų pritaikomas didesnei aibei paskirstytų sistemų, ne tik Apache Kafka.

Laukiami rezultatai

1. Pasirinktų Kafka algoritmų specifikacija.

2. Įrodymas apie specifikacijos adekvatumą.
3. Kafka specifikacijos ir įgyvendinimo sutapimo įvertinimas.
4. Išskirti ir specifikuoti išskirstytų sistemų šablonai taikomi kitose platformose.