

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:	Vytautas Valaitis	(parašas)

Vilnius – 2020

TURINYS

ĮVADAS	2
TEMOS AKTUALUMAS BEI NAUJUMAS	4
DARBO TIKSLAS	5
UŽDAVINIAI	5
LAUKIAMŲ REZULTATAI	5
LITERATŪRA	6

Įvadas

Šiais laikais, kai kurios, programų sistemos yra išskirstytos [Smi18]. Tokios sistemos, kaip pavadinimas sufleruoja, yra kuriamos išskirstant skaičiavimo mazgus į atskiras, savarankiškas dalis. Ne kaip monolitinės sistemos, išskirstytos sistemos gali veikti skirtingose serverinėse, kurios gali būti kitose geografinėje vietoje. Toks sistemos išskirstymas pasižymi šiomis savybėmis:

1. Pasiiekiamumas(angl. *Availability*) – sistemos gebėjimas būti pasiekiamai vartotojui bet kuriuo metu.
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 gaisrų, potvynių ir panašių nelaimių. Ši savybė taip pat užtikrina sistemos gebėjimą atsistatyti ir neprarasti duomenų po minėtų įvykių.
3. Plečiamumas(angl. *Scalability*) – didėjant vartotojų skaičiui bei programų sistemos kompleksiskumui išskirstytos sistemos užtikrina vertikalų (Padidinti mazgo techninės įrangos galingumą) bei horizontalų (Padidinti mazgų skaičių sistemoje) plečiamumą.
4. Našumas(angl. *Efficiency*) – sistemos vartotojų skaičius dažniausiai būna nepastovus, jis kinta dienos metu arba atitinkamais periodais metuose. 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 keletą atviro kodo išskirstytų sistemų padedančių apdoroti realaus laiko duomenis. Viena iš tokių išskirstytų sistemų yra Apache Kafka (toliau – Kafka) [20a].

Kafka buvo pradėta kurti kompanijos LinkedIn [20a]. 2012 metais Kafka sistema buvo perduota į Apache Software Foundation tolesniam vystymui. Šiuo metu Kafka platforma yra žinučių siuntimo sistema, kurios dizainas pasižymi lengvu plečiamumu, patvarumu, patikimumu ir greičiu. Duomenys Kafka platformoje yra išsaugomi saugiu, trukdžiams atspariu būdu. Kafka kūrėjų teigimu, šiuo metu platforma naudoja daugiau negu 80 procentų didžiausių Jungtinių Valstijų įmonių [20a]. Kafka platforma yra plačiai naudojama įvairiose srityse, tokiuose kaip žurnalistika, debesyj paslaugos, muzikos srauto paslaugos, telekomunikacijos, bankinės paslaugos ir daugelis kitų [20a].

Norint užtikrinti Kafka platformos kokybę, kūrėjai yra įgyvendinę skirtingų testų [20b]. Testai padeda atskleisti programos klaidas arba pasakyti ar naujas kodas nepaveikė seniau parašyto funkcionalumo [Whi00]. Tačiau net ir laikantis gerųjų testavimo praktikų nepavyksta išvengti programos klaidų. Net ir paskyrus daugiau resursų testavimui, netrivialiuose sistemose, tokiose kaip Kafka, pilnas sistemos testavimas yra neįmanomas [SYC⁺04]. Todėl norint atrasti subtilesnius sisteminius sutrikimus tenka naudoti kitus metodus, tokius kaip formalus specififikavimas ir verifikavimas.

Formalios specifikacijos yra matematinės technikos, skirtos apibūdinti sistemų elgseną ir padėti kuriant jos dizainą, naudojant griežtas ir veiksmingas priemones [HP95]. Turint sistemos formalią specifikaciją galima ją pasinaudoti vykdant formalų verifikavimą ir parodant, kad sistemos dizainas yra adekvatus pagal sukurta specifikaciją. Sudarinėti formalią sistemos specifikaciją galima ir nepradėjus įgyvendinti sistemos, turint tik jos dizainą. Formaliai verifikuota specifikacija suteikia informacijos apie dizaino korektiškumą ir įgalina objektyviai koreguoti sistemos dizainą dar prieš pradedant jį įgyvendinti. Formalios specifikacijos sudaromos pasinaudojant tam tikra kalbas arba įrankius. Viena iš tokių, formalaus specifikavimo kalbų, yra TLA⁺ [Lam02].

TLA⁺ yra formalios specifikacijos kalba sukurta Leslie Lamport [Lam02]. Leslie Lamport 1980 metais sukūrė laiko veiksmų logiką (angl. *Temporal Logic of Actions*) [Lam94] pasinaudodamas Amir Pnueli 1977 metais sukurta laiko logika (angl. *Temporal Logic*) [Pnu77]. 1999 metais Leslie Lamport, naudodamasis laiko veiksmų logika, sukūrė formalaus specifikavimo kalbą TLA⁺ [Lam02].

TLA⁺ kalba yra skirta kurti konkurencinių ir išskirstytų sistemų formalias specifikacijas ir šias specifikacijas verifikuoti. Naudojant TLA⁺ galima specifikuoti šias išskirstytų sistemų savybes [1702415] :

1. Gyvumas – geri dalykai galiausiai atsitinka 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 netikėtai dėl iškylusios 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 tikrintojas). TLC yra išreikštinės būsenos (explicit-state) modelio tikrintojas, kurio paskirtis yra pažingsniui pasiekti visas galimas sistemos būsenas pagal nurodytą formalią specifikaciją. Tačiau, kartais, pagal sukurta formalią specifikaciją, susidaro labai daug būsenų kurias sistema gali pasiekti, todėl tam pa nepapraktiška naudoti TLC. Tokiu atveju galime naudotis TLA⁺ specifikacijos įrodymo sistema TLAPS. TLAPS yra įrodymų sistema skirta patikrinti TLA⁺ įrodymus. Šios sistemos paskirtis yra patikrinti pateiktus teoremų įrodymus. Įrodžius teoremą laikoma, kad TLA⁺ specifikacija yra korektiška.

Tačiau parašyti formalią specifikaciją yra negana. Kartais gali nutikti taip, kad algoritmo implementacija neatitiks jos specifikacijos arba reikalavimų. Mūsų sudaryta specifikacija gali būti adekvati pagal pateiktus reikalavimus, tačiau sistemos kūrėjai šiuos reikalavimus gali įgyvendinti nekorektiškai. Kad to išvengtume turime patikrinti ar sistemos įgyvendinimas atitinka specifikaciją. Tą pasieksime atlikdami formalią verifikaciją naudodamiesi Kafka įvykiu žurnalu. Tam atlikti pasinaudosime modeliu paremtu pėdsakų tikrinimu (angl. *Model-Based Trace-Checking*) metodu [20d]. Metode aprašomi šie žingsniai:

1. Įgyvendinti programą ir paprastus testus.

2. Pridėti kodą, kuri sekto programos būseną ir įrašytų ją į failą.
3. Suskurti formalią specifikaciją parašytam kodui.
4. Paleisti 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. Antrajame žingsnyje pridėsime kodą, kuris registruos sistemos būseną ir įrašinės ją į tekstinius failus. Trečiajame žingsnyje sukursime formalią specifikaciją naudodamiesi TLA⁺. Ketvirtajame žingsnyje naudodamiesi TLC patikrinsime ar sistemos būsenos failo duomenys yra adekvatūs pagal sukurta formalią specifikaciją apdorodami būsenos duomenis ir TLC pagalba tikrindami ar tokia būsenos galima pasiekti pagal mūsų sukurta specifikaciją.

Temos aktualumas bei naujumas

Viena iš formalių specifikacijų ir TLA⁺ panaudojimo industrijoje sėkmės istorijų, yra Amazon Web Service (AWS) komandos 2014 metais išleistas straipsnis [NRZ⁺14]. Straipsnyje rašoma, kad AWS komanda naudojo TLA⁺ sudarant formalias specifikacijas dešimtyje projektų. Tuo metu AWS turėjo 7 komandas, kurios naudojos 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 sukurta Jason Gustafson [Gus04]. Pristatyme buvo parodyta, kad pritaikius TLA⁺ specifikuojant Kafka duomenų replikavimo algoritmą buvo surastos ir pataisytos 3 retais atsitinkančios programos klaidos. Betaisiant rastos ir pataisytos dar kelios klaidos.

Šiame darbe specifikuosime ir verifikuosime Kafka kūrėjų pasiūlytą Raft algoritmo implementaciją. Raft protokolas yra skirtas pasiekti susitarimą išskirstytoje sistemoje. Kad pasiekti susitarimą tarp sistemos mazgų kiekvienas mazgas turi lyderio arba sekėjo rolę. Algoritme lyderis yra atsakingas už informacijos replikavimą savo sekėjams mazgams. Lyderis kas tam tikrą laiką tarpą siunčia žinutes savo sekėjams mazgams apie savo egzistavimą. Jeigu sekėjas nesulaukia signalo iš lyderio sistemoje prasideda naujo lyderio rinkimas.

Kafka pasiūlyto Raft algoritmo implementacija būtų naudojama duomenų replikavimui. Nors Raft algoritmui jau yra sukurtų formalių specifikacijų TLA⁺ kalba, tačiau pasiūlyta implementacija skiriasi nuo iki šiol sukurtų specifikacijų, todėl reiktų sukurti atskirą formalią specifikaciją bei ją verifikuoti.

Iki šiol, kiek mums žinoma, Kafka platforma buvo specifikuota tik vieną kartą [Gus04] neakademiniam kontekste ir sukurta specifikacija atnešė naudos padedant surasti sistemos klaidas. Panašią mokslininkų sėkmę matome ir Amazon Web Service formalios specifikacijos sudarymo tyrimuose [NRZ⁺14]. Dėl papildomų Kafka formalių specifikacijų stokos ir praeityje pasisėkusio

formalaus specifikavimo išskirstytuose sistemose manome, kad papildomi tyrimai Kafka platformoje atneštu naudą surandant algoritmų klaidas arba užtikrinant, kad specifikuotose algoritmuose jų nėra. Šiuo metu Kafka sisteminių klaidų registre [20c] yra išspręstų ir neišspręstų klaidų kurių verifikavimas padėtų atskleisti naujas klaidas arba įrodyti kad klaidos ištaisytos adekvačiai. Kafka platforma turi daug naudotojų [20a], 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, kuriuose yra naudojami tokie pat algoritmai. Šiuo metu yra straipsnių, kuriuose formaliai verifikuojami išskirstytų sistemų algoritmai [Lam05], kurie yra naudojami kurti išskirstytas sistemas, todėl tikimasi, kad panašių rezultatų pavyktų pasiekti specifikuojant Kafka platformos algoritmus.

Darbo tikslas

1. Parodyti Apache Kafka duomenų replikavimo algoritmų korektiškumą.
2. Įvardinti Apache Kafka problemas susijusias su duomenų replikavimo algoritmais.

Uždaviniai

1. Išnagrinėti literatūrą susijusią su formaliais metodais, TLA⁺ specifikavimo kalba bei Kafka platformą.
2. Formaliai specifikuoti pasirinktus Kafka platformos algoritmus naudojant TLA⁺ specifikavimo kalbą.
3. Verifikuoti, ar pagal sukurtą specifikaciją Kafka platforma veikia korektiškai.
4. Esant poreikiui įrodyti specifikacijos savybes naudojant TLAPS.
5. Surasti kitas paskirstytas sistemas, kuriuose yra naudojami šiame darbe formaliai specifikuoti algoritmai.

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.

Literatūra

- [20a] Apache kafka, 2020. URL: <https://kafka.apache.org/>.
- [20b] Apache kafka mirror tests, 2020. URL: <https://github.com/confluentinc/kafka/tree/master/tests>.
- [20c] Kafka - - asf jira, 2020. URL: <https://issues.apache.org/jira/projects/KAFKA/issues/KAFKA-10635?filter=allopenissues>.
- [20d] Latex is cool, 2020. URL: <https://kafka.apache.org/>.
- [Gus04] Jason Gustafson. Hardening kafka replication. <https://kafka-summit.org/sessions/hardening-kafka-replication/>; <https://github.com/hachikuji/kafka-specification>, 2004. Pristatymas konferencijoje.
- [HP95] Gerard J Holzmann ir Doron Peled. An improvement in formal verification. *Formal Description Techniques VII*, p. 197–211. Springer, 1995.
- [Lam02] Leslie Lamport. *Specifying systems*, tom. 388. Addison-Wesley Boston, 2002.
- [Lam05] Leslie Lamport. Generalized consensus and paxos, 2005.
- [Lam94] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994-05. ISSN: 0164-0925. DOI: 10.1145/177492.177726. URL: <https://doi.org/10.1145/177492.177726>.
- [NRZ⁺14] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker ir Michael Deardeuff. Use of formal methods at amazon web services. *See <http://research.microsoft.com/en-us/um/people/lamport/tla/formal-methods-amazon.pdf>*, 2014.
- [Pnu77] A. Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, p. 46–57, 1977. DOI: 10.1109/SFCS.1977.32.
- [SYC⁺04] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid ir Daniel Jackson. Software assurance by bounded exhaustive testing. *Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, p. 133–142, 2004.
- [Smi18] Tom Smith. New research shows 63% of enterprises are adopting microservices architectures, 2018. URL: <https://www.globenewswire.com/news-release/2018/09/20/1573625/0/en/New-Research-Shows-63-Percent-of-Enterprises-Are-Adopting-Microservices-Architectures-Yet-50-Percent-Are-Unaware-of-the-Impact-on-Revenue-Generating-Business-Processes.html>.
- [Whi00] J. A. Whittaker. What is software testing? and why is it so hard? *IEEE Software*, 17(1):70–79, 2000. DOI: 10.1109/52.819971.