VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ BAKALAURO STUDIJŲ PROGRAMA

Raft Algoritmo Ra Praplėtimų Tyrimas

Investigation of Raft Algorithm Ra Extensions

Kursinis darbas

Atliko: 3 kurso 1 grupės studentas
Edvardas Dlugauskas (parašas)

Darbo vadovas: dr. Karolis Petrauskas (parašas)

Vilnius – 2021

TURINYS

Įvadas

Mūsų amžiuje, esminėms visuomenės sistemoms persikeliant į virtualią erdvę, informacinių sistemų stabilumas tampa vis svarbesnis. IT sistemos turi atlaikyti aparatinės įrangos gedimus, nes vieno bito apvertimas sistemos kūrėjams gali kainuoti labai daug. Neseniai atliktas tyrimas parodė, kad apie 8% visų serverių gali tikėtis per metus patirti bent vieną sutrikimą [VN10].

Išskirstytos sistemos yra patraukli alternatyva lygiagrečioms sistemoms, leidžianti efektyviai panaudoti šiuolaikiškus IT resursus ir užtikrinti sistemos gyvybingumą ir korektiškumą [Ong14; ST17]. Viena iš esminių išskirstytų sistemų problemų yra konsensuso pasiekimas. Išskirstytos sistemos veikimo metu gali būti situacijų, kai kelių mazgų būsenos yra skirtingos. Pavyzdžiui, kai vienas iš mazgų trumpam laikui tampa neprieinamas ir negali atnaujinti savo būsenos, o vėliau vėl tampa prieinamas. Mazgų būsenos gali išsiskirti, ir iš skirtingų mazgų klientui gali būti grąžinami skirtingi rezultatai. Tai pažeistų sistemos vientisumą. Todėl būtina sinchronizuoti būsenas tarp mazgų ir tuo pačiu metu leisti sistemai veikti toliau jei pakankamai maža dalis mazgų neveikia ar yra nepasiekiama. Tam reikia išspręsti konsensuso problemą – kaip mazgai gali pasiekti susitarimą dėl jų bendros būsenos, net kai dalis mazgų gali būti nepasiekiama ar jos duomenys pasenę?

Formaliai, išskirstyta sistema yra skaičiavimų elementų (toliau – mazgų) rinkinys, kuris yra naudojamas kaip vientisa sistema [ST17]. išskirstytose sistemose konsensuso algoritmai yra reikalingi būsenų sinchronizavimui tarp sistemos mazgų. Konsensuso algoritmas turi užtikrinti, kad sistema būtų prieinama vieno ar daugiau mazgų gedimo ar neprieinamumo atvejais [Ong14].

Šiame darbe išnagrinėti Raft konsensuso algoritmo praplėtimai naudojami Ra bibliotekoje. Raft yra konsensuso algoritmas sukurtas palengvinti konsensuso algoritmų įgyvendinimo naštą sistemų architektams ir leisti lengvai praplėsti ir pritaikyti Raft algoritmą specifinėms reikmėms [Ong14].

Šio darbo tikslas yra patikrinti, ar Raft algoritmas, praplėstas Ra naudojamais mazgų neveikimo aptikimo praplėtimais, vis dar atitinka Raft algoritmo savybes ir užtikrina efektyvų mazgų neveikimo aptikimą [Ong14; Ra20].

Šio darbo uždaviniai yra:

- Išnagrinėti Raft algoritmą ir biblioteką Ra.
- Pateikti esamos literatūros apie Raft algoritmą ir Ra įgyvendinimo apžvalgą.
- Išanalizuoti Ra padarytus praplėtimus ir jų įtaką Raft algoritmui.
- Pateikti formalią specifikaciją Ra mazgų neveikimo arba nepasiekiamumo (toliau – gedimo) aptikimui.

Šiame darbe yra pristatomos Raft algoritmo ir Ra bibliotekos apžvalgos, pabrėžiant Ra įgyvendintų praplėtimų įtaką. Taip pat yra pateikiamos ir palyginamos Raft ir Ra gedimų aptikimo algoritmų formalios specifikacijos ir pateikiamos rekomendacijos ateities darbams.

Specifikacijoms naudojama formalių specifikacijų rašymo kalba TLA^+, kuri leidžia formaliai sumodeliuoti algoritmų specifikacijas naudojant matematikos ir logikos taisykles [Lam02].

Industrijoje TLA$^+$ dažnai naudojama apibrėžti išskirstytas sistemas [NRZ+14]. Dėl to, kad TLA$^+$ kalba yra formali, yra galimas modelio tikrinimas, leidžiantis patikrinti tokias aprašytos sistemos charakteristikas kaip gyvybingumas ir saugumas [Lam02].

Šis darbas yra paskirstytas į tris dalis. Pirmoje dalyje yra apžvelgiamas Raft algoritmas ir detaliau išnagrinėjami kelių jo dalių, susijusių su mazgų neveikimo aptikimu, veikimo principai. Antroje dalyje yra aprašomos Raft algoritmo bibliotekoje Ra naudojamos kalbos Erlang ypatybės ir išnagrinėjami Ra padaryti praplėtimai, yra detaliau išnagrinėjami Ra naudojami mazgų neveikimo aptikimo metodai. Trečioje dalyje yra pateikiamos Raft ir Ra gedimo aptikimo algoritmų formalių specifikacijų apžvalgos.

1. Raft algoritmo apžvalga

Raft yra algoritmas, naudojamas konsensusui pasiekti išskirstytoje sistemoje. Bazinis Raft algoritmas yra formaliai specifikuotas TLA^+ kalba. Algoritmo korektiškumas yra formaliai įrodytas remiantis specifikacija [Ong14] ir pagrįstas naudojimu industrijoje – Raft naudojamas etcd[1], Consul[2] ir CockroachDB[3] projektuose.

Raft buvo sukurtas kaip lengviau suprantama alternatyva tuo metu labiausiai paplitusiam Paxos[4] konsensuso algoritmui [DJ16; Ong14]. Raft autoriai, atlikus vartotojų tyrimą, parodė, kad Raft algoritmas yra lengviau suprantamas negu Paxos, iš ko seka mažesni įgyvendinimo kaštai, mažesnis galimų klaidų skaičius ir paprastesnis bazinio algoritmo praplėtimas [Ong14]. Raft leidžia užtikrinti sistemos gyvybingumą ir saugumą kol veikia daugiau negu pusė išskirstytos sistemos mazgų.

Raft algoritmo kontekste sistema laikoma būsenų mašina, kuri priima iš klientų užklausas su komandų įrašais. Kiekvienas įrašas yra norimos sistemai įvykdyti komandos aprašymas. Įvykdant komandą, sistema pakeičia savo vidinę būseną ir grąžina klientui rezultatą – savo atnaujintą būseną ar jos dalį. Atitinkamai, išskirstyta sistema susideda iš mazgų, kurie irgi yra būsenų mašinos, turinčios savo įrašų žurnalus ir vidines būsenas. Toks išskirstytos sistemos išdėstymas leidžia pasiekti aukštą gedimų toleravimo ir greičio lygį [Ong14; ST17].

Raft algoritmas remiasi lyderio išrinkimu – vienas iš mazgų tampa lyderiu, ir yra laikoma, kad lyderio žurnalas yra tiesos šaltinis. Visos klientų užklausos yra siunčiamos lyderiui, kuris persiunčia įrašus kitiems mazgams (sekėjams). Kai dauguma mazgų turi informaciją apie naują įrašą, mazgai užfiksuoja įrašą savo žurnale ir įvykdo įraše nurodytą komandą. Jei nustoja veikti sekėjas, sistema tęsia darbą, kol yra veikiančių mazgų dauguma. Nustojus veikti lyderiui, nesulaukę lyderio gyvumo patvirtinimo, kiekvienas mazgas po atsitiktinio laiko tarpo, pasirinkto iš nustatyto intervalo, pradeda rinkimus. Rinkimai prasideda kai vienas iš mazgų padidina savo termino (kadencijos) reikšmę ir išsiunčia kitiems mazgams prašymo balsuoti užklausas. Išsirinkus naują lyderį sistemą gali tęsti darbą [Ong14].

Formuojant Raft formalią specifikaciją, buvo padarytos sekančios prielaidos apie sistemą ir sistemos tinklą [Ong14]:

- Sistema yra asinchroninė ir neturi tokios sąvokos kaip „laikas".
- Žinutės gali turėti neapibrėžtą skaičių perėjimų kol pasieks savo gavėją. Išsiunčiant žinutę yra įgalinamas žinutės gavimas, tačiau nėra garantuojamas joks gavimo savalaikiškumas.
- Mazgai gali nustoti veikti ir save perkrauti atsistatydami dalį savo būsenos iš ilgalaikės atminties.
- Tinklas gali pakeisti žinučių tvarką ir pamesti (pašalinti) žinutę.

[1] https://etcd.io/
[2] https://www.consul.io/docs/internals/consensus.html
[3] https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html
[4] http://lamport.azurewebsites.net/pubs/paxos-simple.pdf

Sistemos mazgai turi laikiną atmintį, kurioje saugo dažnai naudojamą informaciją, ir ilgalaikę atmintį. Ilgalaikė atmintis turi informacijos apie sistemos konfigūraciją, kuri yra reikalinga korektiškam Raft algoritmo veikimui – mazgai išsaugo savo dabartinį terminą ir už ką jame balsavo. Taip pat ilgalaikėje atmintyje saugomi užfiksuoti įrašai [Ong14].

Raft specifikacija garantuoja, kad Raft algoritmas tenkina sekančias savybes [Ong14]:

1. Lyderio išrinkimas: duotam lyderio terminui galiausiai bus išrinktas ne daugiau kaip vienas lyderis.

2. Lyderio žurnalo uždarumas modifikacijoms: lyderis niekada nekeičia ir netrina įrašų savo žurnale, jis gali tik pridėti naujų įrašų.

3. Žurnalų įrašų atitikimas: jeigu dviejų mazgų žurnalai turi įrašą su tuo pačiu indeksu ir terminu, tai tie įrašai yra lygūs ir visi abiejų žurnalų įrašai iki to indekso yra lygūs.

4. Lyderio žurnalo pilnumas: jeigu žurnalo įrašas buvo užfiksuotas, tai visų sekančių terminų lyderiai turės tą užfiksuotą įrašą savo žurnale.

5. Sistemos kaip būsenos mašinos saugumas: jei mazgas įvykdė žurnalo įrašą, joks kitas mazgas niekada neįvykdys kitokio įrašo su tuo pačiu indeksu.

Toliau detaliau apžvelgiame, kaip Raft užtikrina, kad galiausiai būtų išrinktas mazgų lyderis, kaip lyderis perduoda įrašus kitiems mazgams, ir kaip Raft elgiasi nutikus mazgų tinklo trūkiams.

1.1. Raft lyderio išrinkimas

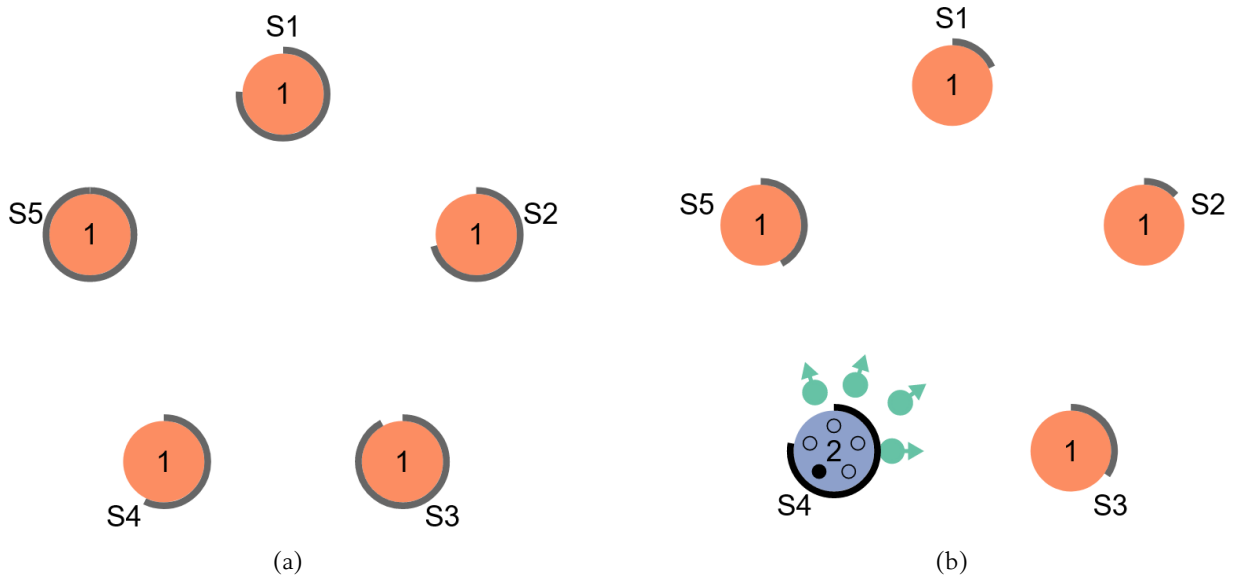
Pagal Raft, mazgas–lyderis periodiškai siunčia savo sekėjams tuščias įrašo pridėjimo užklausas (`AppendEntries`), taip užtikrinant savo „širdies plakimą" (angl. *heartbeat*) [Ong14]. Jeigu sekėjas kažkurį laiką negauna užklausų iš lyderio, sekėjas pradeda naujus rinkimus ir pasisiūlo kaip kandidatas į naują lyderį. Nuosekliai auganti reikšmė, nurodanti dabartinio lyderio kadencijos numerį, vadinamą *terminu* (angl. *term*), o laikas, po kurio sekėjas pradės naujo termino rinkimus vadinamas *rinkimų laiku* (angl. *election timeout*). Rinkimų laikas yra atsitiktinė reikšmė iš pasirinkto laiko intervalo, kas padeda greičiau išrinkti naują lyderį kai yra daugiau negu vienas kandidatas [Ong14].

Rinkimų pradžioje, sekėjas tampa kandidatu, balsuoja šiame rinkimų termine už save, ir išsiunčia visiems kitiems mazgams prašymo balsuoti užklausas (`RequestVote`). Mazgas per rinkimų terminą gali balsuoti tik už vieną mazgą. Mazgai balsuoja už pirmą mazgą, kuris šiame termine atsiuntė prašymą balsuoti, arba už save, jei jie ką tik pradėjo rinkimus. Mazgai nebalsuoja už kitą mazgą, jei tas mazgas yra pasenęs, tai yra to mazgo terminas yra mažesnis, o jei terminai lygūs, kandidato žurnalas turi mažiau įrašų. Galiausiai arba kandidatas laimi rinkimus, arba kitas mazgas–kandidatas laimi rinkimus, arba lyderis nėra išrenkamas ir pradedamas naujas balsavimas [Ong14].

Geresniam algoritmo supratimui, galima panagrinėti tokį Raft lyderio išrinkimo situacijos pavyzdį: yra penki mazgai pavadinimais *S1*, *S2*, *S3*, *S4* ir *S5*. Kiekvienas iš jų ką tik startavo ir

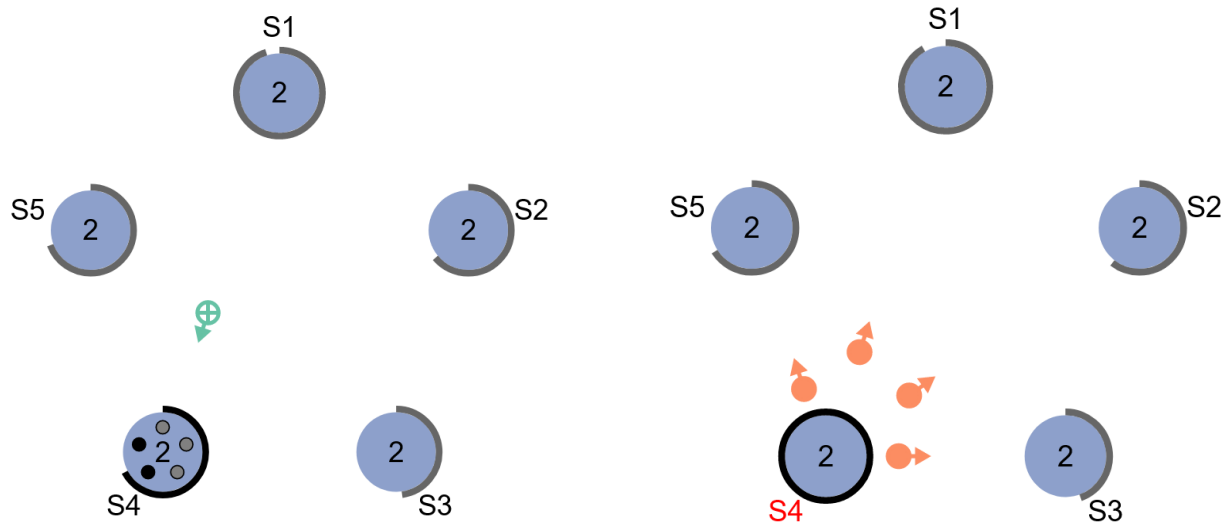
jokio lyderio kol kas nėra. Kiekvienas iš mazgų priskiria sau atsitiktinį rinkimų laiką iš intervalo, nustatyto sistemos konfigūracijoje. Rekomenduojama, kad apatinė šio intervalo riba būtų kelis kartus didesnė negu tikėtinas žinučių uždelsimas tinkle.

Vizualizavimo pagalbai panaudosime RaftScope [Ong20] programą, kuri yra laisvai prieinama pagal MIT licenciją. 1a pav. mazgus simbolizuoja apskritimai, o dabartinį mazgo terminą – skaičius apskritime. Likęs rinkimų laikas pavaizduotas kaip pilkas apskritimų kraštas. Matome, kad visų serverių rinkimų laikai yra skirtingi ir pirmas tą laiką pasieks serveris *S4*.



1 pav. Kai tarp mazgų nėra lyderio, mazgas, kurio rinkimų laikas yra ankščiausiai (*S4*), pirmas inicijuoja naujus rinkimus ir tampa kandidatu.

Kai ateina rinkimų laikas, mazgas tampa kandidatu ir yra pradedami nauji lyderio rinkimai. *S4* padidina savo terminą, balsuoja šiame rinkimų termine už save ir išsiunčia kitiems mazgams (*S1*, *S2*, *S3*, *S5*) prašymą balsuoti už jį. Prie šios užklausos yra pridedama informacija apie *S4* žurnalą, kad mazgai galėtų nuspręsti, ar kandidatas *S4* yra tinkamas lyderis ir ar jam reikia atiduoti savo balsą. Mazgui *S4* reikia surinkti tris iš penkių balsų, taip užtikrinus daugumos pritarimą. 1b pav. surinkti balsai pavaizduoti kaip užtušuoti juodi apskritimai viduje mazgo.



(a) (b)

2 pav. Gavus daugumą balsų, mazgas–kandidatas (*S4*) tampa lyderiu.

2a pav. *S4* jau surinko du balsus – vieną iš savęs, kitą iš kito mazgo. Gavus trečią, paskutinį reikalingą, balsą, *S4* tampa lyderiu ir išsiunčia kitiems mazgams savo širdies plakimo užklausą (žr. 2b pav.). Gavus lyderio užklausą, kiti mazgai iš naujo nustato rinkimų laiką.

1.2. Raft įrašų perdavimas

Išrinktas lyderis aptarnauja klientus; jei užklausas iš klientų gauna mazgas kuris nėra lyderis, jis persiunčia užklausą lyderiui. Iš kliento lyderis gauna užklausą su nusakytu veiksmu, kurį reikia atlikti išskirstytai sistemai kaip būsenų mašinai. Gautą žinutę lyderis prideda į savo žurnalą kaip naują įrašą, tada asinchroniškai išsiunčia įrašo pridėjimo užklausas sekėjams. Kai dauguma sekėjų sėkmingai prideda įrašą į savo žurnalą, įrašas yra pažymimas kaip užfiksuotas (angl. *committed*), įvykdomas atitinkamas įrašo veiksmas mazgų būsenų mašinoms ir lyderis grąžina rezultatą klientui [Ong14].

Jei dalis mazgų nustojo veikti ar lėtai grąžina atsakymus, lyderis pakartotinai siunčia įrašų pridėjimo užklausas tiems sekėjams, kurie dar nepatvirtino įrašo gavimo. Galiausiai visi mazgai turės visus žurnalo įrašus. Šis pakartotinio informacijos siuntimo procesas gali vykti net jau lyderiui išsiuntus atsakymą klientui. Tai užtikrina sistemos gyvybingumą kai mažesnioji dalis mazgų yra nepasiekiami [Ong14].

1.3. Raft mazgų tinklo trūkių neigiamų pasekmių sumažinimas

Pagal klasikinį Raft, nutikus mazgų tinklo trūkiui, tai yra kai dalis mazgų tampa nepasiekiama kitai daliai mazgų, įvyksta papildomas lyderio išrinkimas – atsijungusi dalis mazgų nepasiekia buvusio lyderio ir, praėjus rinkimų laikui, pradeda naują terminą (padidina kadencijos numerį). Kaip pasekmė, susijungus šioms mazgų dalims atgal į vieną tinklą, didesnė dalis mazgų gauna užklausas iš mažesnės dalies mazgų, kuriuose nurodytas terminas yra didesnis. Tada didesnės dalies mazgų lyderis atsistatydina, ir prasideda nauji rinkimai [Ong14]. Per trūkio laiką nauji įrašai

galėtų būti užfiksuoti tik didesnėje dalyje mazgų, nes mažesnėje nebūtų galima gauti daugumos patvirtinimo. Jeigu per trūkio laiką buvo užfiksuota naujų įrašų, tai reikštų, kad lyderiu galėtų būti išrinktas tik vienas iš tos didesnės mazgų grupės narių. Vienas iš šitų mazgų yra trūkio pabaigoje esantis lyderis. Taigi, yra vykdomi papildomi rinkimai, nors galėtų būti paliktas esamas lyderis.

Šioms klasterių apjungimo pasekmėms pašalinti Raft autoriai siūlo padaryti algoritmo praplėtimą: įvesti dar vieną mazgų būseną tarp „sekėjo" ir „kandidato" būsenų – „pirminis balsavimas" (angl. *pre-vote*). Šioje būsenoje kandidatas pradeda lyderio rinkimus tik sužinojęs, kad dauguma mazgų balsuotų už jį kaip lyderį. Mazgo terminas šioje būsenoje nėra padidinamas. Šis sprendimas išsprendžia trikdžius daliai mazgų prisijungiant atgal į didesnį mazgų klasterį, nes mažesnėje dalyje mazgų niekada nebus padidintas kadencijos numeris [Ong14].

2. Raft algoritmo įgyvendinimas Ra

Šioje dalyje yra apžvelgiamas Raft algoritmo įgyvendinimas Ra. Ra yra atviro kodo Raft algoritmo biblioteka, naudojama atviro kodo žinučių brokeryje RabbitMQ[5]. Ra biblioteka yra naudojama RabbitMQ komandos išskirstytoms eilėms įgyvendinti, tačiau yra laisvai prieinama ir gali būti naudojama kaip bendra Raft biblioteka. Ra yra įgyvendinta Erlang[6] kalba.

Nors didžioji dalis Raft algoritmo savybių yra įgyvendintos pagal Raft specifikaciją, dėl Erlang kalbos ypatybių ir konkrečių naudojimo scenarijų Ra skiriasi nuo specifikacijos, pavyzdžiui, žurnalo perdavimo ir gedimų aptikimo algoritmais [Ra20].

Pirmoje dalyje apžvelgsime Erlang kalbos ypatybes, kurios nulėmė Ra pasirinktus įgyvendinimo sprendimus. Antroje dalyje apžvelgsime Raft praplėtimus, padarytus Ra bibliotekoje. Trečioje dalyje yra detaliau išnagrinėjamas Ra naudojamas klaidų aptikimo mechanizmas ir palyginamas su Raft pasiūlytų širdies plakimo mechanizmu naudojant `AppendEntries` užklausas.

2.1. Išskirstytas programavimas Erlang

Erlang yra funkcinė programavimo kalba leidžianti kurti išskirstytas, atsparias gedimams, ir didelio prieinamumo reikalaujančias sistemas. Pavadinimas „Erlang" gali būti naudojamas kaip sinonimas „Erlang/OTP", kur OTP (angl. *Open Telecom Platform*, šiame darbe naudojame angliška santrumpą) yra karkasas skirtas supaprastinti skirtingų sistemų bendrų bruožų įgyvendinimą; OTP yra prieinamas kartu su Erlang kalba ir yra standartinės bibliotekos, kaip kalbos C++ bibliotekos `std`, atitikmuo [Erl20b]. Šiame darbe autoriai pavadinimą „Erlang" naudojame tik Erlang programavimo kalbai vadinti.

Išskirstytas programavimas Erlang yra pagrįstas aktorių modeliu [Agh85; FF12]. Pagrindinis Erlang vykdymo vienetas yra mazgas (angl. *node*) – tai yra Erlang virtualios mašinos vienetas (išskirstytos sistemos vienetus taip pat vadinsime mazgais). Erlang mazgas gali turėti tūkstančius jame vykdomų procesų (angl. *process*), o procesai komunikuoja vienas su kitu žinučių pagalba (angl. *message*). Vienam iš procesų nustojus veikti, kiti procesai neįtakojami ir tęsia savo darbą. Nenumatytoms klaidoms suvaldyti procesai gali būti prižiūrimi prižiūrėtojo (angl. *supervisor*), kuris gautų pranešimą nenumatytai klaidai įvykus ir prižiūrimam procesui nustojus veikti [Erl20a].

Vienam iš mazgų sudarant ryšį su kitu mazgu (pavyzdžiui, panaudojus `net_kernel:connect/1`), yra apsikeičiama žinomais mazgais ir taip sudaromas vienas mazgų tinklas. Erlang mazgų sistema yra labai lanksti ir leidžia vienam iš mazgų deleguoti kitam mazgui laisvai pasirinktą funkciją vykdymui (angl. *arbitrary code execution*). `net_kernel` modulio pagalba vienas iš mazgų gali prenumeruoti mazgų statusų pakeitimų žinučių gavimą. Tokiu atveju, pasikeitus žinomo mazgo statusui, mazgas gaus pranešimą su nauju atitinkamo mazgo statusu [Erl20a; Heb13].

Erlang mazgai palaiko ryšį tarp vienas kito „širdies plakimo" (angl. *heartbeat*) mechanizmo pagalba. `ticktime` yra širdies plakimo užklausos siuntimo intervalas padaugintas iš keturių ir žymi laiką, po kurio mazgas bus laikomas nepasiekiamu. Kai vienas iš mazgų negauna iš kito mazgo

[5]https://www.rabbitmq.com/

[6]https://www.erlang.org/

jo gyvavimo patvirtinančio pranešimo per `ticktime` sekundžių, tas stebimas mazgas laikomas nepasiekiamu [Heb13].

2.2. Ra įgyvendinti Raft algoritmo praplėtimai

Dėl Erlang kalbos ypatybių, konkretaus naudojimo scenarijaus ir siekio optimizuoti Raft algoritmo veikimo laiką ir naudojamų duomenų kiekį, Ra išsiskiria nuo specifikacijos kelių savybių įgyvendinimu [Ra20]. Šiame skyriuje yra pateikiama Ra žurnalo perdavimo algoritmo ir Ra mazgų negyvybingumo aptikimo mechanizmų apžvalgos.

2.2.1. Ra žurnalo perdavimo algoritmas

Ra nenaudoja žurnalo įrašų perdavimo užklausų mazgų gyvybingumui užtikrinti kai visų mazgų žurnalai yra sinchronizuoti. Kai vienas iš mazgų yra pasenęs, tai yra, to mazgo paskutinis sinchronizuotas įrašas yra ankstesnis negu paskutinis įrašas, siųstas lyderio tam mazgui, `AppendEntries` užklausa yra daroma, bet rečiau, negu pasiūlyta Raft specifikacijoje. Jei skirtumas tarp paskutinio sinchronizuoto įrašo ir paskutinio siųsto įrašo yra labai didelis, mazgo apkrovai sumažinti užklausos nėra siunčiamos [Ra20]. Žurnalų įrašų perdavimas yra vykdomas asinchroniškai, o atsakymai yra grupuojami [Ra20].

Šiuo praplėtimo atveju Ra autoriai išnaudojo Erlang kalbos galimybes, nes Erlang leidžia palaikyti ryšį tarp mazgų išreikštinai nesiunčiant papildomų užklausų. Nesiunčiant papildomų užklausų taip pat yra atlaisvinamas ryšio kanalas, leidžiant pačiam Erlang efektyviai palaikyti ryšius ir daryti širdies plakimo užklausas [Ra20]. Šiame darbe šis praplėtimas detaliau nenagrinėjamas.

2.2.2. Ra gedimų aptikimo algoritmas

Ra nenaudoja Raft aprašyto gedimo aptikimo algoritmo. Ra naudoja gimtąją Erlang mazgų stebėjimo infrastruktūrą stebėti ar mazgai nenustojo veikti ir kartu pasitelkia Aten[7] biblioteka mazgų susisiekimo klaidoms aptikti [Ra20]. Aten yra adaptyvus kaupiamasis gedimų aptikimo algoritmas (angl. *accrual failure detection algorithm*), kuris atsižvelgia į širdies plakimų istoriją ir pagal tai nusprendžia, ar yra didelė tikimybė nesulaukti sekančio gyvumo patvirtinimo. Tai yra naudinga siekiant sumažinti klaidingai teigiamų gedimo rezultatų skaičių didelio tinklo apkrovos metu [SPT+07].

Gimtieji Erlang monitoriai leidžia greitai aptikti mazgų neveikimą procesui sustojus veikti nenumatytos klaidos metimo atveju (angl. *crash*). Stebėtojui yra išsiunčiama `'DOWN'` žinutė [Erl20c]. Erlang monitoriai užtikrina, kad sekėjai greitai sužino apie lyderio neveikimą, ir taip iš dalies pakeičia Raft širdies plakimo mechanizmą [Ra20]. Aten biblioteka leidžia aptikti tinklo paskirstymo problemas, kaip mazgų nepasiekiamumą per tinklą. Kai Aten įtaria, kad vienas iš mazgų yra nepasiekiamas, yra išsiunčiama `nodedown` žinutė mazgo stebėtojui. Atitinkamai, jei mazgas „atsigauna" ir atrodo vėl veikiantis, yra išsiunčiama `nodeup` žinutė [Ate20]. Kartu Erlang

[7] https://github.com/rabbitmq/aten

monitoriai ir Aten biblioteka, anot Ra autorių, užtikrina mazgų gyvumą ir pakeičia Raft aprašytą širdies plakimo mechanizmą [Ra20].

Ra bibliotekoje visi mazgai kurie naudoja Raft algoritmą yra įgyvendinti kaip būsenų mašinos Erlang modulio `gen_statem` [8] pagalba. Pirmiausia būsenų mašina prenumeruoja gauti pranešimus apie mazgų statusų pokyčius naudojant `net_kernel:monitor_nodes/1`[9] funkciją. Kai vienas iš mazgų tampa neprieinamas, procesas gauna `'DOWN'` pranešimą ir įvykdo atitinkamą Raft algoritmo logiką. Neprieinamumo tikrinimo laiką `ticktime` naudotojas nustato pagal savo poreikį konfigūracinio parametro pagalba [Ra20].

Taigi, Ra naudoja Erlang monitorius kiekvieno iš mazgų būsenoms stebėti. Tai leidžia palengvinti įgyvendinimo kaštus, nes didžioji dalis širdies plakimo mechanizmo algoritmo yra paimama iš Erlang OTP bibliotekos. Mazgui netikėtai nustojus veikti, klaida bus aptikta iškarto ir stebėtojui bus išsiųsta `'DOWN'` žinutė, o mazgui lėtai atsakant ar ryšiui nutrūkus, nepasiekiamumą padės nustatyti širdies plakimo mechanizmas paremtas adaptyviu kaupiamuoju gedimų aptikimo algoritmu.

[8] http://erlang.org/doc/man/gen_statem.html

[9] http://erlang.org/doc/man/net_kernel.html#monitor_nodes-1

3. Ra gedimų aptikimo algoritmo formali specifikacija

Ra bibliotekos Raft algoritmo įgyvendinimo korektiškumo užtikrinimo sudėtingumui sumažinti, šio darbo metu buvo praplėsta Raft algoritmo formalios specifikacijos dalis susijusi su gedimų aptikimo algoritmu. Taip pat sukurta ir Ra bibliotekos gedimų aptikimo algoritmo formali specifikacija. Šioje dalyje yra pateikiamos šių dviejų formalių gedimų aptikimo algoritmo specifikacijų (toliau – specifikacijų), parašytų TLA$^+$ kalba, aprašymai. Specifikacijos yra pateikiamos kaip darbo priedai. Specifikacijų išeities tekstai yra viešai pasiekiami adresu: `https://github.com/EdvardasDlugauskas/Coursework-2020`. Šių dviejų algoritmų vienodos elgsenos įrodymas nėra pateikiamas ir lieka būsimiems darbams.

Pagrindinis Raft ir Ra gedimo aptikimo algoritmų formalių specifikacijų kūrimo tikslas yra formaliai aprašyti Ra mazgų gedimo aptikimo mechanizmą, kas vėliau padėtų įrodyti Ra įgyvendinimo Raft specifikacijos atitikimą ir Raft savybių išlaikymą. Raft autoriai pateikė Raft formaliosios specifikacijos pradinį kodą, tačiau jame gedimų aptikimo algoritmas yra suabstrahuotas kaip vienas žingsnis – sekėjo rinkimo laiko pasibaigimas ir perėjimas į sekančią būseną, pradiniam kode tai nurodo funkcija `Timeout` [Ong14]. Detalesniam Raft ir Ra gedimo aptikimo mechanizmų palyginimui šio darbo autoriai, remiantis Raft specifikacija, sukūrė detalesnę Raft gedimo aptikimo algoritmo specifikaciją, kuri yra pateikta nr. 1 priede. Ra gedimų aptikimo algoritmo specifikacija yra sukurta panašiu detalumo lygiu siekiant palengvinti šių dviejų specifikacijų palyginimą. Pilna Ra gedimų aptikimo algoritmo specifikacija yra pateikta nr. 2 priede.

Šių specifikacijų rašymo metu buvo laikomasi tokių pat prielaidų kaip Raft formalioje specifikacijoje (žr. 1 skyrių).

Pagal Raft, pasikeitus lyderiui, naujojo lyderio terminas bus didesnis, kas privers sekėją padidinti ir savo vidinį terminą, pagal kurį galima ignoruoti senesnių lyderių pavėlavusias užklausas (nes juose nurodytas terminas bus senesnis). Tai leidžia supaprastinti specifikaciją iki vieno lyderio–sekėjo ryšio tyrimo. Specifikacijų sritis ir yra dviejų mazgų – lyderio ir sekėjo – ryšio palaikymas ir sekėjo lyderio gedimų aptikimas. Šis lyderio–sekėjo ryšys yra nepalaikomas kai sekėjas nustato, kad lyderis nustojo veikti arba nėra pasiekiamas. Dėl to papildomai apribojame specifikacijų aprašytą sritį:

- Modeliuojamas tik vienas lyderio–sekėjo ryšys.
- Lyderiui nustojus veikti ir sekėjui tai nustačius, sekėjas pereiną į sekančią būseną („pirminis balsavimas” arba „kandidatas”) ir tolimesnis elgesys nėra aprašomas specifikacijose.
- Pagal Raft specifikaciją, `AppendEntries` užklausa be žurnalo įrašų sekėjo būsenoje įtakoja tik `commitIndex` reikšmę [Ong14]. Todėl kitos perduodamos reikšmės nėra įtrauktos į specifikacijas.

Formali Raft gedimo aptikimo mechanizmo specifikacija, kuri yra pateikta priede nr. 1, aprašo lyderio ir vieno iš jo sekėjų ryšį, kai sekėjas gauna iš lyderio `AppendEntries` užklausas. Specifikacija apibrėžia visas galimas šio ryšio būsenas. Yra apibrėžiama pradinė būsena (`Init`), kurioje lyderis yra veikiantis bet dar neišsiuntė nė vienos užklausos, ir apibrėžiami visi galimi

perėjimai iš vienos būsenos į kitą (`Next`). Tokių perėjimų yra keletas: lyderis išsiunčia žinutę; lyderis padidina savo paskutinio fiksuoto įrašo indeksą; tinklas pameta (ištrina) žinutę; sekėjas gauna žinutę ir atnaujina savo `commitIndex` reikšmę; lyderis nustoja veikti, po ko negalės atlikti jokių veiksmų; sekėjas nustato kad lyderis nepasiekiamas.

$$
\begin{aligned}
\text{Next} \triangleq\ & \lor \text{SendMessage} \\
& \lor \text{IncrementIndex} \\
& \lor \text{DropMessage} \\
& \lor \text{ReceiveMessage} \\
& \lor \text{CrashLeader} \\
& \lor \text{Timeout}
\end{aligned}
$$

Prie kiekvieno perėjimo aprašytos sąlygos, kurias tenkinant šis perėjimas gali įvykti. Pavyzdžiui, lyderis gali siųsti užklausas tik jei jis yra veikiantis.

$$
\begin{aligned}
&\text{SendMessage} \triangleq \\
&\quad \land \text{leaderState} = \text{“ALIVE”} \\
&\quad \land \text{messages}' = \text{Append}(\text{messages}, \text{leaderIndex}) \\
&\quad \land \text{UNCHANGED } \langle \text{leaderState}, \text{nodeIndexes}, \text{isTimeout} \rangle
\end{aligned}
$$

Formali Ra bibliotekos gedimo aptikimo mechanizmo specifikacija yra pateikta priede nr. 2 ir aprašo tą patį lyderio–sekėjo ryšį. Ra atveju širdies plakimo žinutės neneša papildomos informacijos kaip `commitIndex`, o nustojus veikti lyderiui yra išsiunčiama `nodedown` žinutė, kurią gavus sekėjas gali nelaukti kol pasibaigs rinkimų laikas. Taigi, galimi perėjimai iš vienos būsenos į kitą šiuo atveju yra: lyderis išsiunčia širdies plakimo žinutę; sekėjas gauna širdies plakimo žinutę; tinklas pameta (ištrina) širdies plakimo žinutę; lyderis nustoja veikti, yra išsiunčiama `nodedown` žinutė; sekėjas gauna `nodedown` žinutę ir iš karto pasižymi, kad lyderis yra nepasiekiamas; tinklas pameta (ištrina) `nodedown` žinutę; sekėjas nustato kad lyderis nepasiekiamas. Analogiškai su Raft specifikacija, prie kiekvieno perėjimo parašytos jam reikalingos sąlygos.

$$
\begin{aligned}
\text{Next} \triangleq\ & \lor \text{DropHeartbeat} \\
& \lor \text{SendHeartbeat} \\
& \lor \text{ReceiveHeartbeat} \\
& \lor \text{DropNodedown} \\
& \lor \text{ReceiveNodedown} \\
& \lor \text{CrashLeader} \\
& \lor \text{Timeout}
\end{aligned}
$$

Raft gedimų aptikimo specifikacijos korektiškumui patikrinti buvo panaudotas invariantas `TypeOK`, užtikrinantis teisingus specifikacijoje naudojamų kintamųjų rėžius. Taip pat buvo panaudota savybė `LeaderFailureDetected` – jei lyderis yra neveikiantis, kažkurioje sekančioje būsenoje bus aptiktas lyderio nepasiekiamumas.

$$
\begin{aligned}
\text{TypeOK} \triangleq\ & \wedge \text{leaderState} \in \{\text{“ALIVE”}, \text{“CRASHED”}\} \\
& \wedge \text{messages} \in \text{Seq}(\text{Nat}) \\
& \wedge \text{leaderIndex} \in \text{Nat} \\
& \wedge \text{followerIndex} \in \text{Nat} \\
& \wedge \text{isTimeout} \in \text{boolean}
\end{aligned}
$$

$$
\text{LeaderFailureDetected} \triangleq \text{leaderState} = \text{“CRASHED”} \leadsto \text{isTimeout} = \text{TRUE}
$$

Raft gedimo aptikimo specifikacijai buvo sukurtas TLA$^+$ modelis ir apribota tikrinamų būsenų aibė: lyderio indeksas turi būti mažesnis 10, o žinučių, vienu metu esamų tinkle, skaičius – mažesnis nei 5. Savybėms patikrinti buvo panaudotas TLC modelio būsenų tikrinimo įrankis, integruotas į TLA$^+$ Toolbox darbo aplinką. Iš viso buvo patikrinta virš 80 tūkstančių unikalių būsenų, kiekvienai iš kurių galiojo aprašytos kintamųjų rėžių ir lyderio gedimo aptikimo savybės. Ra gedimo aptikimo specifikacijai buvo sukurtos ir patikrintos atitinkamos `TypeOK` ir `LeaderFailureDetected` savybės, buvo apribota tikrinamų būsenų aibė – širdies plakimo užklausų gali būti ne daugiau kaip 13. Visos modelio rastos būsenos (iš viso unikalių apie 100 tūkstančių) atitiko apibrėžtas savybes.

Palyginus specifikacijas, galima teigti, kad, išskyrus `commitIndex` perdavimą sekėjui, Ra algoritmas atlieka tuos pačius veiksmus kaip Raft algoritmas, bet papildomai turi `nodedown` žinučių perdavimo mechanizmą. Mūsų nuomone, tai parodo, kad Ra įgyvendinimas atitinka Raft specifikaciją, tačiau formaliam įrodymui reikia ištirti `commitIndex` nebuvimo širdies plakimo žinutėse pasekmes ir formaliai įrodyti dviejų algoritmų ekvivalentiškumą. Tai yra paliekama ateities darbams.

Rezultatai ir išvados

Mūsų nuomone, Raft algoritmo įgyvendinimas Ra yra gerokai palengvinamas tuo, kad gimtosios Erlang funkcijos ir bibliotekos yra skirtos būtent tokių išskirstytų sistemų būsenų stebėjimui, todėl įgyvendinimo kaštai yra dar labiau sumažinami. Visa tai gerai komponuoja su Raft autorių motyvacija [Ong14] – padaryti išskirstytų sistemų konsensuso algoritmo įgyvendinimą kuo paprastesnį, sumažinti galimų klaidų skaičių, leisti kuo lengviau papildyti esamą funkcionalumą naujais praplėtimais. Mūsų nuomone, Ra biblioteka yra ne tik puikus techninio Raft algoritmo įgyvendinimo, bet ir Raft paprastumo ir aiškumo dvasią išlaikančios bibliotekos pavyzdys.

Šiame darbe buvo pasiekti rezultatai:

1. Pateiktos formalios Raft ir Ra gedimų aptikimo algoritmų specifikacijos parašytos TLA$^+$ kalba. Išnagrinėti jų skirtumai.

Šio darbo padarytos išvados:

1. Mūsų nuomone, tarp Raft ir Ra gedimo aptikimo specifikacijų yra daug panašumų, tačiau yra ir skirtumų, kurių poveikis algoritmo veikimui neakivaizdus.

2. Ra bibliotekos Raft mazgų neveikimo aptikimo specifikacijos atitikimui formaliai įrodyti reikia ištirti `commitIndex` reikšmės nebuvimo širdies plakimo žinutėse padarinius ir formaliai įrodyti jų gedimų aptikimo algoritmų Raft savybių atitikimo ekvivalentiškumą, galimai remiantis šiame darbe pateiktomis formaliomis Raft ir Ra algoritmų specifikacijomis.

Literatūra

[Agh85] Gul Abdulnabi Agha. ACTORS: a model of concurrent computation in distributed systems, 1985-06-01. url: https://dspace.mit.edu/handle/1721.1/6952 (tikrinta 2020-05-14). Accepted: 2004-10-20T20:10:20Z.

[Ate20] Aten. Rabbitmq/aten, 2020-04-22. url: https://github.com/rabbitmq/aten (tikrinta 2020-05-06). original-date: 2018-01-31T13:51:46Z.

[DJ16] Ongaro Diego ir Ousterhout John. Designing for understandability: the raft consensus algorithm - YouTube. 2016-10-28. url: https://www.youtube.com/watch?v=vYp4LYbnnW8 (tikrinta 2020-04-07).

[Erl20a] Erlang. Erlang – distributed erlang. 2020-05-13. url: http://erlang.org/doc/reference_manual/distributed.html (tikrinta 2020-05-13).

[Erl20b] Erlang. Erlang – introduction. 2020-05-13. url: http://erlang.org/doc/system_architecture_intro/sys_arch_intro.html (tikrinta 2020-05-13).

[Erl20c] Erlang. Erlang – processes. 2020-05-06. url: http://erlang.org/doc/reference_manual/processes.html#monitors (tikrinta 2020-05-06).

[FF12] Audrianne Farrugia and Adrian Francalanza. Towards a formalisation of erlang failure and failure detection:6, 2012.

[Heb13] Fred Hebert. *Learn You Some Erlang for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, 1 edition leid., 2013-01-13. isbn: 978-1-59327-435-1.

[Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[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.

[Ong14] Diego Ongaro. Consensus: bridging theory and practice:258, 2014.

[Ong20] Diego Ongaro. Ongardie/raftscope, 2020-06-07. url: https://github.com/ongardie/raftscope (tikrinta 2020-06-11). original-date: 2014-05-21T09:36:26Z.

[Ra20] Ra. Rabbitmq/ra, 2020-02-21. url: https://github.com/rabbitmq/ra (tikrinta 2020-02-22). original-date: 2017-02-22T17:07:57Z.

[SPT+07] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler ir Theo Ungerer. A new adaptive accrual failure detector for dependable distributed systems. *Proceedings of the 2007 ACM symposium on Applied computing*, p. 551–555, 2007.

[ST17] Maarten van Steen ir Andrew S. Tanenbaum. *Distributed systems*. Pearson Education, London, third edition (version 3.01 (2017)) leid., 2017, p. 582. isbn: 978-90-815406-2-9 978-1-5430-5738-6. OCLC: 1006750554.

[VN10] Kashi Venkatesh Vishwanath ir Nachiappan Nagappan. Characterizing cloud computing hardware reliability. *Proceedings of the 1st ACM symposium on Cloud computing*, p. 193–204, 2010.

Priedas nr. 1

Raft algoritmo gedimo aptikimo specifikacija TLA$^+$ kalba

MODULE *RaftHeartbeat*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

Is leader *ALIVE* or *CRASHED*
VARIABLE *leaderState*

A collection of heartbeat (*AppendEntries*) messages the leader has sent.
A single message is abstracted to represent the leader's index
VARIABLE *messages*

A representation of the *commitIndex* and term, leader increases index monotonically.
VARIABLE *leaderIndex*
VARIABLE *followerIndex*
$nodeIndexes \triangleq \langle leaderIndex, followerIndex \rangle$

Indicates whether the follower timed out after not hearing from
the leader for the specified amount of time.
VARIABLE *isTimeout*

$vars \triangleq \langle leaderState, messages, nodeIndexes, isTimeout \rangle$

The leader crashes and doesn't recover
$CrashLeader \triangleq$
$\land\ leaderState = \text{“ALIVE”}$
$\land\ leaderState' = \text{“CRASHED”}$
$\land$ UNCHANGED $\langle messages, nodeIndexes, isTimeout \rangle$

The leader sends the follower an *AppendEntries* message
$SendMessage \triangleq$
$\land\ leaderState = \text{“ALIVE”}$
$\land\ messages' = Append(messages, leaderIndex)$
$\land$ UNCHANGED $\langle leaderState, nodeIndexes, isTimeout \rangle$

Helper function to remove a message from a sequence of messages
$RemoveMessage(i, seq) \triangleq$
$[j \in 1 \,..\, Len(seq) - 1 \mapsto$ IF $j < i$ THEN $seq[j]$ ELSE $seq[j+1]]$

The network drops a message
$DropMessage \triangleq$
$\land\ Len(messages) \geq 1$
$\land\ \exists\, i \in 1 \,..\, Len(messages) :$
$\quad messages' = RemoveMessage(i, messages)$
$\land$ UNCHANGED $\langle leaderState, nodeIndexes, isTimeout \rangle$

The leader increments its index
$IncrementIndex \triangleq$
$\land\ leaderState = \text{“ALIVE”}$

$\land\ leaderIndex' = leaderIndex + 1$
$\land$ UNCHANGED $\langle leaderState,\ messages,\ followerIndex,\ isTimeout\rangle$

The follower receives a message from the leader.

$$
\begin{array}{l}
ReceiveMessage \triangleq \\
\quad \land\ Len(messages) \geq 1 \\
\quad \land\ \exists\, i \in 1\,..\,Len(messages) : \\
\qquad ((\text{LET}\ message \triangleq messages[i] \\
\qquad \text{IN}\quad followerIndex' = \text{IF}\ message > followerIndex \\
\qquad\qquad\qquad\qquad\qquad \text{THEN}\ message \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE}\ followerIndex) \\
\qquad \land\quad messages' = RemoveMessage(i,\ messages)) \\
\quad \land\ \text{UNCHANGED}\ \langle leaderState,\ leaderIndex,\ isTimeout\rangle
\end{array}
$$

The follower times out

$$
\begin{array}{l}
Timeout \triangleq\ isTimeout' = \text{TRUE} \\
\qquad \land\quad \text{UNCHANGED}\ \langle leaderState,\ messages,\ nodeIndexes\rangle
\end{array}
$$

Initial state of model

$$
\begin{array}{l}
Init \triangleq\ \land\ leaderState = \text{“ALIVE”} \\
\qquad\quad \land\ messages = \langle\rangle \\
\qquad\quad \land\ leaderIndex = 0 \\
\qquad\quad \land\ followerIndex = 0 \\
\qquad\quad \land\ isTimeout = \text{FALSE}
\end{array}
$$

Next state function

$$
\begin{array}{l}
Next \triangleq\ \lor\ SendMessage \\
\qquad\quad \lor\ IncrementIndex \\
\qquad\quad \lor\ DropMessage \\
\qquad\quad \lor\ ReceiveMessage \\
\qquad\quad \lor\ CrashLeader \\
\qquad\quad \lor\ Timeout
\end{array}
$$

$Spec \triangleq\ Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars}(Next)$

Invariant that helps make sure we haven't stepped out of bounds

$$
\begin{array}{l}
TypeOK \triangleq\ \land\ leaderState \in \{\text{“ALIVE”},\ \text{“CRASHED”}\} \\
\qquad\quad \land\ messages \in Seq(Nat) \\
\qquad\quad \land\ leaderIndex \in Nat \\
\qquad\quad \land\ followerIndex \in Nat \\
\qquad\quad \land\ isTimeout \in \text{BOOLEAN}
\end{array}
$$

Properties of the system

$LeaderFailureDetected \triangleq\ leaderState = \text{“CRASHED”} \rightsquigarrow isTimeout = \text{TRUE}$

THEOREM $Correctness \triangleq\ Spec \Rightarrow \Box LeaderFailureDetected$

Priedas nr. 2

Ra bibliotekos gedimo aptikimo specifikacija TLA$^+$ kalba

MODULE *RaHeartbeat*

EXTENDS *Naturals*, *FiniteSets*, *Sequences*, *TLC*

Is leader *ALIVE* or *CRASHED*
VARIABLE *leaderState*

Helper variable tracking the number of times the leader node has failed
VARIABLE *nodedownIndex*

A collection of 'nodedown' messages the leader has sent.
In *Erlang*, when a monitored node crashes, the 'nodedown' message is sent to the follower.
The value of a message is not a part of the implementation and
represents the number of times the node has failed.
VARIABLE *nodedownMessages*

$nodedownInfo \triangleq \langle nodedownMessages, nodedownIndex \rangle$

Heartbeat messages the leader has sent
The value of a message is not a part of the implementation and
represents the number of times a heartbeat message was sent.
VARIABLE *heartbeatMessages*

Helper variable tracking the number of sent heartbeat messages
VARIABLE *heartbeatIndex*

$heartbeatInfo \triangleq \langle heartbeatMessages, heartbeatIndex \rangle$

Whether the follower timed out
VARIABLE *isTimeout*

$vars \triangleq \langle leaderState, nodedownInfo, heartbeatInfo, isTimeout \rangle$

The leader crashes and doesn't recover
$CrashLeader \triangleq$
$\quad \land leaderState = \text{“ALIVE”}$
$\quad \land leaderState' = \text{“CRASHED”}$
Erlang sends the *NODEDOWN* message if the leader was monitored
$\quad \land nodedownMessages' = Append(nodedownMessages, nodedownIndex)$
$\quad \land nodedownIndex' = nodedownIndex + 1$
$\quad \land$ UNCHANGED $\langle heartbeatInfo, isTimeout \rangle$

Helper function to remove a message from a sequence of messages
$RemoveMessage(i, seq) \triangleq$
$\quad [j \in 1 \,..\, Len(seq) - 1 \mapsto$ IF $j < i$ THEN $seq[j]$ ELSE $seq[j + 1]]$

The network drops a heartbeat message
$DropHeartbeat \triangleq$

$\land\ Len(heartbeatMessages) > 0$
$\land\ \exists\, i \in 1\,..\,Len(heartbeatMessages) :$
$\quad heartbeatMessages' = RemoveMessage(i,\ heartbeatMessages)$
$\land$ UNCHANGED $\langle leaderState,\ heartbeatIndex,\ nodedownInfo,\ isTimeout\rangle$

The follower receives a heartbeat message from the leader.

$ReceiveHeartbeat\ \triangleq$
$\land\ Len(heartbeatMessages) > 0$
$\land\ \exists\, i \in 1\,..\,Len(heartbeatMessages) :$
$\quad heartbeatMessages' = RemoveMessage(i,\ heartbeatMessages)$
$\land$ UNCHANGED $\langle leaderState,\ heartbeatIndex,\ nodedownInfo,\ isTimeout\rangle$

The leader sends a heartbeat message to the follower.

$SendHeartbeat\ \triangleq$
$\land\ leaderState =$ "ALIVE"
$\land\ heartbeatMessages' = Append(heartbeatMessages,\ heartbeatIndex)$
$\land\ heartbeatIndex' = heartbeatIndex + 1$
$\land$ UNCHANGED $\langle leaderState,\ nodedownInfo,\ isTimeout\rangle$

The network drops a nodedown message.

$DropNodedown\ \triangleq$
$\land\ Len(nodedownMessages) > 0$
$\land\ \exists\, i \in 1\,..\,Len(nodedownMessages) :$
$\quad nodedownMessages' = RemoveMessage(i,\ nodedownMessages)$
$\land$ UNCHANGED $\langle leaderState,\ nodedownIndex,\ heartbeatInfo,\ isTimeout\rangle$

The follower receives a nodedown message from the leader which causes it time out immediately.

$ReceiveNodedown\ \triangleq$
$\land\ Len(nodedownMessages) > 0$
$\land\ \exists\, i \in 1\,..\,Len(nodedownMessages) :$
$\quad nodedownMessages' = RemoveMessage(i,\ nodedownMessages)$
$\land\ isTimeout' =$ TRUE
$\land$ UNCHANGED $\langle leaderState,\ nodedownIndex,\ heartbeatInfo\rangle$

$Timeout\ \triangleq$
$\land\ isTimeout =$ FALSE
$\land\ isTimeout' =$ TRUE
$\land$ UNCHANGED $\langle leaderState,\ heartbeatInfo,\ nodedownInfo\rangle$

Initial state of model

$Init\ \triangleq\ \land\ leaderState =$ "ALIVE"
$\land\ nodedownIndex = 0$
$\land\ nodedownMessages = \langle\rangle$
$\land\ heartbeatIndex = 0$
$\land\ heartbeatMessages = \langle\rangle$
$\land\ isTimeout =$ FALSE

Next state function

$$
\begin{aligned}
Next \triangleq\ & \lor DropHeartbeat \\
& \lor SendHeartbeat \\
& \lor ReceiveHeartbeat \\
& \lor DropNodedown \\
& \lor ReceiveNodedown \\
& \lor CrashLeader \\
& \lor Timeout
\end{aligned}
$$

$$Spec \triangleq Init \land \Box[Next]_{vars} \land \mathrm{WF}_{vars}(Next)$$

Invariant that helps make sure we haven't stepped out of bounds

$$
\begin{aligned}
TypeOK \triangleq\ & \land leaderState \in \{\text{“ALIVE”}, \text{“CRASHED”}\} \\
& \land nodedownIndex \in Nat \\
& \land nodedownMessages \in Seq(Nat) \\
& \land heartbeatMessages \in Seq(Nat) \\
& \land heartbeatIndex \in Nat \\
& \land isTimeout \in \text{BOOLEAN}
\end{aligned}
$$

Properties of the system

$$LeaderFailureDetected \triangleq leaderState = \text{“CRASHED”} \leadsto isTimeout = \text{TRUE}$$

$$\text{THEOREM } Correctness \triangleq Spec \Rightarrow \Box LeaderFailureDetected$$