VILNIAUS UNIVERSITETAS
MATEMATIKOS IR INFORMATIKOS FAKULTETAS
PROGRAMŲ SISTEMŲ STUDIJŲ PROGRAMA

„Redis Cluster“ podėlio sistemos tyrimas, taikant formalius metodus

Analysis of the “Redis Cluster” Cache System Using Formal Methods

Magistro baigiamasis darbas

Atliko:	Mantas Kontrimas	(parašas)
Darbo vadovas:	doc. dr. Karolis Petrauskas	(parašas)
Recenzentas:	prof. dr. Linas Laibinis	(parašas)

Vilnius – 2021

Santrauka

Magistriniame darbe yra analizuojama populiari podėlio sistema „Redis Cluster“ ir jos korektiškumas. Analizuojant sistemą buvo naudojami formalūs metodai – TLA$^+$ specifikavimo kalba buvo sudarytos dvi sistemos formalios specifikacijos: abstrakti specifikacija, modeliuojanti sistemos funkcionalumą, bei detali specifikacija, artima programinio kodo abstrakcijos lygiui. Specifikacijos modelio tikrinimo metu buvo vertinama, ar yra užtikrinama sistemos savybė, kad už vieną maišos lizdą yra atsakingas tik vienas pagrindinis mazgas ir jo pavaldūs mazgai. Atlikus modelio tikrinimą buvo surastos situacijos, kada ši sistemos savybė nėra užtikrinama ir sistema veikia nekorektiškai. Surastos klaidos buvo atkartotos realioje sistemoje ir šioms klaidoms buvo pateikti galimi sprendimo būdai.

Raktiniai žodžiai: Redis Cluster, išskirstytos sistemos, formalūs metodai, TLA$^+$

Summary

This master thesis analyzes the correctness of the popular cache system "Redis Cluster". The system is investigated using formal methods – in the TLA^+ language two formal specifications of the system are formulated: the abstract and detailed specifications. The abstract specification models system requirements and the more detailed specification is at the code abstraction level and models actual system algorithm and behavior. Using model checker it was checked if the system holds the property that only one master and its slaves must be responsible for the hash slot. The model checking showed that there are situations when the analyzed property is not satisfied and the system works incorrectly. Found issues were reproduced at the actual system and the ways how to fix these problems were proposed.

Keywords: Redis Cluster, distributed systems, formal methods, TLA^+

TURINYS

ĮVADAS 4

1. IŠSKIRSTYTA PODĖLIO SISTEMA 7
   1.1. „Redis Cluster“ sistema 7
      1.1.1. Sistemos našumas 8
      1.1.2. Duomenų saugojimas 8
      1.1.3. Sistemos architektūra 10
      1.1.4. Mazgų tarpusavio komunikacija 11
      1.1.5. Naujo pagrindinio mazgo rinkimas 13
      1.1.6. „Redis Cluster“ savybių apibendrinimas 14
   1.2. Kitos išskirstytos podėlio sistemos 15

2. FORMALŪS METODAI 17
   2.1. Formalios specifikavimo kalbos 18
   2.2. TLA^+ kalba 18

3. FORMALIOS SPECIFIKACIJOS 22
   3.1. Išskirstytų podėlių formalios specifikacijos 22
   3.2. Raktas-reikšmė NoSQL duomenų bazių formalios specifikacijos 23
   3.3. Apibendrinimas 25

4. „REDIS CLUSTER“ SISTEMOS FORMALIOS SPECIFIKACIJOS 26
   4.1. Išorinių veiksmų formali specifikacija 26
      4.1.1. Specifikacijos struktūra 27
      4.1.2. Sistemos komandos 29
      4.1.3. Išoriniai įvykiai 32
      4.1.4. Bendras modelio tikrinimas 33
   4.2. Žinučių apdorojimo formali specifikacija 34
      4.2.1. Specifikacijos struktūra 34
      4.2.2. Mazgo būsenos žymos 35
      4.2.3. Įvykių ciklas 36
      4.2.4. Žinučių apdorojimas 40
      4.2.5. Išoriniai įvykiai 46
      4.2.6. Blokinio administravimo komandų apdorojimas 47
      4.2.7. Modelio tikrinimo optimizavimas 56

5. „REDIS CLUSTER“ SISTEMOS TYRIMAS 58
   5.1. Specifikacijos modelis 58
   5.2. Maišos lizdų savybės invariantas 59
   5.3. Rastos klaidos ir galimi sprendimo būdai 60
   5.4. Formalių specifikacijų susiejimas 70

REZULTATAI IR IŠVADOS 73

ŠALTINIAI 75

PRIEDAI 78
   1 priedas. Išorinių veiksmų formali specifikacija 79
   2 priedas. Žinučių apdorojimo formali specifikacija 83
   3 priedas. „Redis Cluster“ sistemos UML diagramos 118

Įvadas

Šiandien serveriai ir didesnės sistemos turi aptarnauti keletą tūkstančių klientų vienu metu, o vieno naudotojo puslapio užkrovimas gali pareikalauti keleto užklausų, kurios turi būti įvykdytos per sekundės dalis [VF07; ZAA17]. Būtent todėl šiuolaikinėms sistemoms svarbu sumažinti atsako laiką į užklausas bei darbo krūvį, su kuriuo turi susidoroti serveriai [CGB15]. Podėlio (angl. *cache*) operatyviojoje atmintyje naudojimas yra vienas iš efektyviausių būdų kaip galima tai pasiekti. Podėliui reikalingą struktūrą, susidedančią iš rakto-reikšmės poros, gali pasiūlyti atvirojo kodo produktai: „Redis“, „Memcached“, „Ehcache“ ir kiti [CTW+16; LFL17]. Taip pat, siekiant dar labiau pagerinti podėlio našumą bei patikimumą, podėlis gali būti paskirstytas arba replikuotas tarp mazgų [CTW+16; PI13].

„Redis“ – tai viena iš populiariausių NoSQL duomenų bazių, operatyviojoje atmintyje laikančių raktas-reikšmė poras [Car13]. Dažniausiai ši duomenų bazė yra naudojama kaip podėlis, norint pagerinti sistemos našumą [JGO+14; LDY+18]. Be to, „Redis“ palaiko ir blokinių kūrimą (angl. *clustering*), kuris leidžia padidinti podėlio talpą, duomenis paskirstant tarp mazgų.

„Redis Cluster“ duomenis paskirsto skirtingiems mazgams, kurie yra vadinami pagrindiniais (angl. *master*) mazgais, o kiekvienas pagrindinis mazgas turi keletą pavaldžių mazgų (angl. *slave*), kurie saugo pagrindinio mazgo duomenų kopijas. Duomenys „Redis Cluster“ sistemoje yra laikomi 16384 (2^{14}) maišos lizduose (angl. *hash slots*), kurių reikšmės yra apskaičiuojamos raktui, kuriuo norima padėti reikšmę, pritaikant maišos funkciją. Kiekvienas mazgas yra atsakingas už tam tikrą maišos lizdų aibę. Blokinį sudarantys mazgai yra visi pilnai tarpusavyje sujungti komunikavimo kanalais, kuriais paskalų (angl. *gossip*) protokolo pagalba dalinasi blokinio būsena ir ilgainiui sutaria dėl kiekvieno mazgo būsenos [Red18].

Kadangi „Redis Cluster“ yra išskirstytas podėlis, tai šioje sistemoje gali egzistuoti išskirstytoms sistemos būdingos problemos [Aba12]. Išskirstytose sistemose veiksmai vyksta lygiagrečiai, asinchroniškai, keičiasi mazgų būsenos, mazgai tarpusavyje komunikuoja žinutėmis, o ir sistemoje saugoma informacija yra išskirstyta tarp keleto mazgų. Būtent dėl to yra sunku užtikrinti, kad išskirstyta sistema nuolatos veiks korektiškai: bus užtikrintas duomenų neprieštaringumas (angl. *consistency*), sistema nepaklius į aklavietes (angl. *deadlock*) arba sistema bus funkciškai teisinga ir pasižymės kitomis jai būdingomis savybėmis, pateiktomis, pavyzdžiui, reikalavimų specifikacijoje [BMP18; Mal16]. Norint tai užtikrinti, galima naudoti formalius metodus, kurie padeda ištirti sistemos korektiškumą ir jos savybes bei padeda surasti sudėtingas sistemos klaidas, kurių neaptinka kitos verifikavimo technikos [NRZ+15].

Formalūs metodai – tai matematiniu pagrindu paremti metodai, dažniausiai naudojami apra-

šant sistemos savybes ir elgseną bei tiriant dviprasmybes, sistemos nepilnumą ir prieštaringumą. Sistemos kūrimo metu formalūs metodai gali būti naudojami specifikuojant sistemos reikalavimus, o sudarytos formalios specifikacijos gali padėti surasti sistemos klaidas, projektavimo trūkumus, prieš pradedant įgyvendinti pačią sistemą. Taikant formalius metodus jau sukurtai sistemai, formalūs metodai gali padėti pagrįsti sistemos įgyvendinimo korektiškumą [Win90]. Turint formalią specifikaciją, algoritmo ar sistemos korektiškumą galima patikrinti taikant formalios specifikacijos įrodymus, modelio tikrinimą, peržiūras, animacijas ir kitus metodus [Liu16]. Formalius metodus galima taikyti visų programų sistemų kūrimo etapų metu (reikalavimų rinkimo, projektavimo, testavimo ir kitų) [WLB+09].

Taikant formalius metodus ir rašant formalias specifikacijas, yra naudojamos formalios specifikavimo kalbos. Šios kalbos yra sukurtos remiantis pasirinktu matematiniu pagrindu, pavyzdžiui, algebra, logika, aibių teorija ir kitomis matematikos sritimis, ir šias kalbas sudaro: sintaksės ir semantikos aprašymai bei semantikos taisyklės [AMI17]. Formalios specifikavimo kalbos yra skirstomos į dvi kategorijas: modeliavimo (angl. *model-oriented*), kurios apima ir būsenų mašinomis paremtas kalbas, ir savybių (angl. *property-oriented*) specifikavimo kalbas [AMI17].

TLA$^+$ kalba – viena iš formalių specifikavimo kalbų, leidžiančių specifikuoti reaktyvias, išskirstytas ir asinchronines sistemas bei aptikti tokių sistemų klaidas [LMT+02; Mos13; NRZ+15]. Ši formali specifikavimo kalba paremta veiksmų laiko logika (angl. *temporal logic of actions*, sutrumpintai – TLA), kuri gali būti naudojama tiriant lygiagrečių ir išskirstytų sistemų veikimą bei savybes [Lam94]. TLA$^+$ kalba nagrinėjamą sistemą aprašo kaip būsenų mašiną (išreikštą viena matematine formule), o modelio tikrintojas TLC tikrina visas galimas sistemos būsenas baigtiniame sistemos modelyje [LMT+02]. Taip pat egzistuoja ir kitos būsenų mašinomis paremtos formalios specifikavimo kalbos, pavyzdžiui Event-B, VDM++, kurias galima taikyti nagrinėjant išskirstytas sistemas [FTL+07; MP17; SCY15].

„Redis Cluster" sistemos korektiškumas buvo vertinamas [CTW+16], tačiau nepaisant to, kad ši sistema yra plačiai naudojama, mokslinėje literatūroje ji tirta tik fragmentiškai. Atsižvelgiant į tai, šiame magistriniame darbe yra nagrinėjamas „Redis Cluster" sistemos korektiškumas, taikant formalius metodus.

Darbo tikslas ir uždaviniai:

Darbo tikslas – naudojantis formaliais metodais, ištirti išskirstyto podėlio sistemos „Redis Cluster" korektiškumą.

Tikslui pasiekti bus sprendžiami šie uždaviniai:

1. Atlikti literatūros apžvalgą ir jos analizę;
2. Identifikuoti ir formalizuoti savybes, kuriomis turi pasižymėti išskirstyta podėlio sistema „Redis Cluster“;
3. Sudaryti sistemos „Redis Cluster“ formalią specifikaciją;
4. Ištirti sudarytos formalios specifikacijos adekvatumą;
5. Ištirti „Redis Cluster“ algoritmo korektiškumą;
6. Ištirti, kaip surastos sistemos problemos gali būti ištaisytos ar sušvelnintos.

1. Išskirstyta podėlio sistema

Šiuolaikinėms sistemoms, aptarnaujančioms keletą tūkstančių klientų, yra ypač svarbu sumažinti atsako laiką į užklausas bei darbo krūvį, su kuriuo turi susidoroti sistemos. Dažnai, norint tai pasiekti, yra naudojamas podėlis operatyviojoje atmintyje. Podėlis – tai komponentas, kuriame yra laikoma informacija raktas-reikšmė pavidalu, siekiant pagreitinti sistemos darbą, kad, pavyzdžiui, nereikėtų duomenų gavimo iš duomenų bazės ar to pačio skaičiavimo atlikti keletą kartų [CGB15; VF07; ZAA17].

Priklausomai nuo naudojimo atvejo (angl. *use case*) podėlius galima naudoti taikant kelias pagrindinės podėlio naudojimo strategijas. Pirmoji strategija – podėlis nuošalyje (angl. *cache aside*), kurios metu, įvykdžius skaičiavimą ar duomenų gavimą, duomenys yra padedami į podėlį pagal raktą ir kitą kartą, prieš vykdant skaičiavimą, programa turėdama raktą patikrina ar podėlyje dar nėra išsaugota reikšmė. Jei reikšmė jau buvo išsaugota, tai ji pagal raktą yra paimama iš podėlio, o, jei reikšmė dar nebuvo išsaugota podėlyje, tai ji yra apskaičiuojama ar gaunama ir yra padedama į podėlį. Antroji strategija – perskaitymo (angl. *read-through*), kurios metu programa kreipiasi į podėlį ir, jei podėlis neturi reikšmės pagal tą raktą, podėlis atsisiunčia trūkstamus duomenis iš duomenų bazės. Kita podėlio naudojimo strategija – rašymo per (angl. *write-through*) podėlį, kurios metu programa duomenis iš pradžių įrašo į podėlį, o po to į duomenų bazę. Sekantį kartą programai norint gauti duomenis, nebereikia kreiptis į duomenų bazę – užtenka paimti reikšmę iš podėlio, kadangi visi naujausi duomenys ir atnaujinimai taip pat bus įrašyti ir į podėlį [Cod17].

Siekiant pagerinti podėlio našumą, prieinamumą ir efektyvumą, keletas podėlių taip pat gali kooperuotis dirbti kartu ir sudaryti išskirstytą podėlį. Kaip išskirstytas podėlis nuošalyje gali būti naudojamos ir NoSQL duomenų bazės, duomenis saugančios raktas-reikšmė poromis. Tokios išskirstytos podėlio sistemos yra: „Redis“, „Memcached“, „Ehcache“ ir kitos [PI13; XHZ10; XLJ+16]. Įterpus tokį išskirstytą podėlį tarp duomenų bazės ir programos serverio, sumažėja duomenų bazės serverio apkrova bei pagreitėja visos sistemos našumas [DWL+10].

1.1. „Redis Cluster“ sistema

„Redis“ – tai viena iš populiariausių atvirojo kodo raktas-reikšmė duomenų saugyklų, kuri yra naudojama kaip duomenų bazė, podėlis ar žinučių tarpininkas. Dėl šios sistemos našumo dažniausiai daugelis naudotojų „Redis“ naudoja kaip podėlį saugoti serverio pusės (angl. *server side*) duomenis [LZL+18]. „Redis“ saugykloje saugomi duomenys gali būti įvairių tipų: simbolių eilutės, sąrašai, aibės ir išrikiuotas aibės. Taip pat nuo trečiosios „Redis“ sistemos versijos atsirado blokinių palaikymas – „Redis Cluster“, kuris leidžia podėlyje saugomus duomenis išskirstyti per

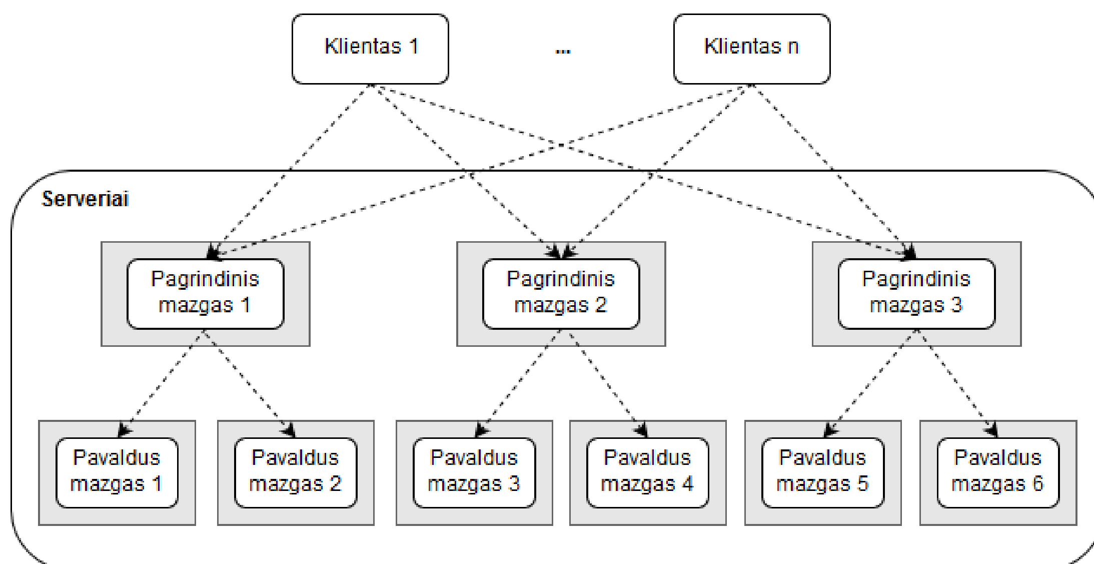
kelis mazgus, kuriuose veikia „Redis“ saugykla. Pagrindinis „Redis Cluster“ tikslas – didelis našumas (angl. *performance*) ir išplečiamumas (angl. *scalability*), išlaikant silpną, bet pagrįstą (angl. *reasonable*) duomenų neprieštaringumą ir sistemos prieinamumą [Red18].

1.1.1. Sistemos našumas

„Redis“ sistema pasižymi dideliu našumu, kadangi duomenys yra saugomi operatyviojoje atmintyje. Taip pat „Redis Cluster“ blokinio našumas nesiskiria nuo vieno „Redis“ sistemos egzemplioriaus (angl. *instance*), kadangi klientai bendrauja su konkrečiu „Redis“ mazgu, kuriame yra laikoma reikšmė pagal tam tikrą raktą. Be to, klientai, bendraudami su sistema, ilgainiui susidaro blokinio įvaizdį – žino, kuris mazgas yra atsakingas už tam tikrą aibę raktų, todėl iš karto gali kreiptis į konkretų mazgą, o ne bandyti surasti, kuris mazgas yra atsakingas už tam tikrą raktą [Red18].

1.1.2. Duomenų saugojimas

„Redis Cluster“ sistemoje yra naudojama kliento-serverio struktūra. Klientas – tai programa, kuri geba, naudojantis TCP/IP protokolu, komunikuoti su „Redis“ podėliu – serveriu, kuriame yra laikoma dalis podėlio duomenų. Iš viso sistemoje gali būti daug klientų bei daug serverių – „Redis Cluster“ mazgų, o kiekvienas iš klientų gali kreiptis į bet kurį blokinio mazgą [Red18].

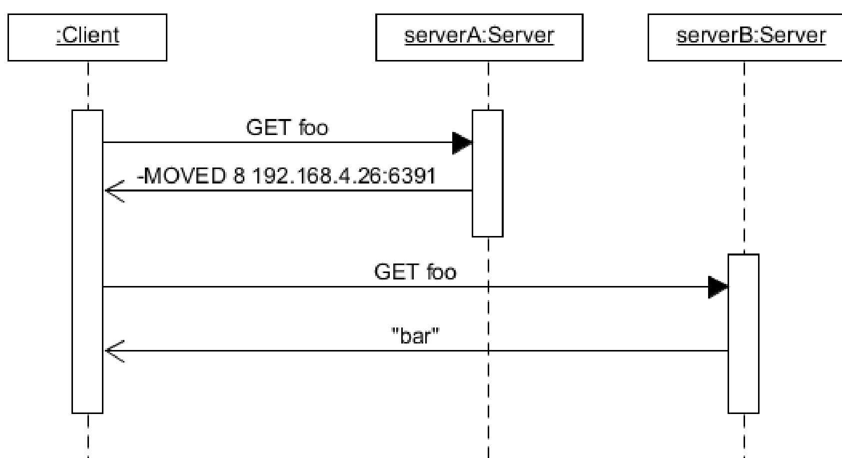


1 pav. „Redis Cluster“ kliento-serverio struktūra [CTW+16]

Iš viso „Redis Cluster“ sistemoje yra 16384 maišos lizdai (angl. *hash slots*), kurių reikšmės yra apskaičiuojamas raktui, kuriuo norima padėti reikšmę, pritaikant funkciją CRC16 moduliu 16384. Kiekvienas iš blokinio mazgų (serverių) yra atsakingas už tam tikrą maišos lizdų aibę.

Pavyzdžiui, mazgas A yra atsakingas už maišos lizdus nuo 0 iki 5000, mazgas B yra atsakingas už maišos lizdus nuo 5001 iki 10000 ir mazgas C yra atsakingas už maišos lizdus nuo 10001 iki 16383. Mazgui priskirta maišos lizdų aibė gali kisti, t.y. lizdai gali būti priskirti kitam mazgui. Pavyzdžiui, maišos lizdų aibių pertvarkymą (angl. *resharding*) galima atlikti tada, kai mazgas yra labai apkrautas. Tada dalis jam priklausančių lizdų gali būti perkelti kitam, mažiau apkrautam mazgui arba naujam mazgui, kuris ką tik buvo pridėtas prie blokinio [Red18].

Naujo mazgo pridėjimas, pašalinimas ar maišos lizdų aibės pertvarkymas nereikalauja, kad „Redis Cluster“ sistema būtų išjungta ar sustabdyta. Į blokinį naujai pridėtas mazgas nėra atsakingas už tam tikrą aibę maišos lizdų, todėl reikia atlikti maišos lizdų aibės pertvarkymą – perkelti maišos lizdus iš jau egzistuojančių mazgų į naujai sukurtą mazgą. Tuo tarpu mazgas iš blokinio gali būti pašalintas tik tada, kai jis yra tuščias [Red18].



2 pav. Kliento ir serverio komunikavimo pavyzdys, kai klientas kreipiasi į serverį A ir serveris A nėra atsakingas už raktą „foo“, todėl grąžina *MOVED* klaidos pranešimą

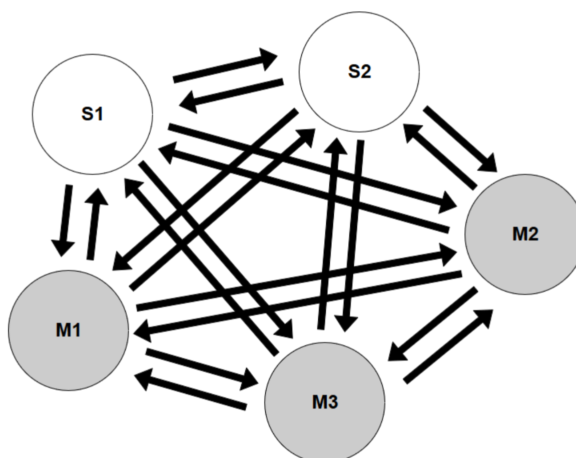
Klientas, norėdamas gauti konkretaus rakto reikšmę, gali kreiptis į bet kurį iš mazgų. Jei mazgas, į kurį buvo kreiptasi, yra atsakingas už tą raktą ir yra pagrindinis mazgas, tada jis grąžins rakto reikšmę. Tačiau, jei jis nėra atsakingas už tą raktą, jis grąžins *MOVED* klaidos pranešimą, kuriame nurodys, į kurį mazgą klientui reikia kreiptis, norint gauti reikšmę pagal tą raktą. Jeigu klientas kreipėsi į pavaldų mazgą, atsakingą už tą maišos lizdą, tai, nenorėdamas daryti dar vieno papildomo kreipimosi į pagrindinį mazgą, klientas gali vykdyti *readonly* komandą, kuri reiškia, kad klientui tinka reikšmė saugoma pavaldžiame mazge. Klientui vykdant *readonly* komandą, pavaldus mazgas grąžins peradresavimo pranešimą tik tuo atveju, jei jis nėra atsakingas už tą maišos lizdą [Red18].

Kadangi „Redis Cluster“ klientai gali siųsti užklausas į bet kurį mazgą, ir klientas ne visada

iš pirmo karto gali pataikyti į reikalingą mazgą, todėl yra patartina klientuose turėti lokalų podėlį, saugantį raktas-„Redis Cluster“ mazgo adresas reikšmes. Toks lokalus kliento podėlis pagreitins komunikavimą tarp kliento ir blokinio mazgų, kadangi klientui dažniausiai nereikės daryti kelių užklausų, norint gauti mazgo, atsakingo už raktą, adresą [Red18].

1.1.3. Sistemos architektūra

Siekiant padidinti prieinamumą, „Redis Cluster“ sistemoje yra naudojami dviejų tipų mazgai: pagrindiniai ir pavaldūs mazgai, kurie saugo pagrindinio mazgo duomenų kopijas – replikas. Visi mazgai, nepriklausomai nuo jų tipo, saugo informaciją apie kitus mazgus (visada žino blokinio būseną) ir visi mazgai yra pilnai tarpusavyje sujungti komunikavimo kanalais. Komunikavimui tarp mazgų yra naudojamas TCP/IP protokolas ir binarinis protokolas – „Redis Cluster“ magistralė (angl. *bus*). Kai blokinyje įvyksta pasikeitimas, pavyzdžiui, yra pridedamas naujas mazgas, yra suprantama, kad nebegalima pasiekti kito mazgo ir pan., blokinio būsenos pasikeitimo informacija yra paskleidžiama į visus mazgus naudojantis *Gossip* protokolu ir ilgainiui visi mazgai sutaria dėl kiekvieno mazgo būsenos [Red18].



3 pav. „Redis Cluster“ komunikavimo kanalai, kai M1, M2 ir M3 – pagrindiniai mazgai, o S1 ir S2 – pavaldūs mazgai [CTW+16]

Nepaisant replikų naudojimo, „Redis Cluster“ sistema ne visais atvejais yra pilnai prieinama. Sistema tampa neveiksni, jei dauguma pagrindinių mazgų tampa neaktyvūs. Siekiant pagerinti sistemos prieinamumą, „Redis Cluster“ sistemoje galima sukonfigūruoti automatinį replikų migravimą (angl. *replicas migration*) – pagrindiniam mazgui likus be pavaldaus mazgo yra atliekamas automatinis blokinio pavaldžių mazgų (replikų) pertvarkymas, kad pavaldaus mazgo netekęs pagrindinis mazgas vėl turėtų pavaldų mazgą, kuriame galėtų saugoti duomenų kopijas. Sukonfigūravus replikų automatinį migravimą, replikų migravimo algoritmas užtikrina, kad galiausiai kiekvienas pagrindinis mazgas bus palaikomas bent vieno pavaldaus mazgo. Algoritmo metu, visi

pavaldūs mazgai stebi, ar visi pagrindiniai mazgai turi bent po vieną pavaldų mazgą. Identifikavus tokią blokinio būseną, kai pagrindiniam mazgui trūksta pavaldaus mazgo, pradeda veikti (angl. *act*) tik tas pavaldus mazgas, kuris yra aktyvus, kuris turi mažiausią mazgo identifikatorių ir kurio pagrindinis mazgas turi didžiausią skaičių pavaldžių mazgų. Algoritmas taip pat gali būti kontroliuojamas blokinio konfigūravimo metu nustatant minimalų pavaldžių mazgų skaičių, su kuriuo turi likti pagrindinis mazgas, prieš tai kai įvyksta pavaldaus mazgo migracija [Red18].

„Redis Cluster“ neužtikrina griežto (angl. *strong*) neprieštaringumo – egzistuoja atvejų kada galima prarasti sistemoje saugomus duomenis. Sistema negali užtikrinti šios savybės, kadangi duomenų replikavimas tarp pagrindinio ir pavaldžių mazgų yra asinchroninis, t.y. pagrindinis mazgas nelaukia kol pavaldūs mazgai patvirtins, kad gauti duomenys buvo išsaugoti. Jeigu reikia, „Redis Cluster“ taip pat palaiko sinchroninį duomenų įrašymą naudojant *wait* komandą. Tačiau šios komandos naudojimas turi įtakos sistemos našumui, kadangi pagrindinis mazgas turi laukti, kol gaus atsakus iš pavaldžių mazgų, kad nauji duomenys juos pasiekė. Tačiau, pasak sistemos kūrėjų, sinchroninis replikavimas vis tiek neužtikrina griežto neprieštaringumo – gali egzistuoti sudėtingos situacijos, kurių metu naujų duomenų negavęs pavaldus mazgas bus išrinktas nauju pagrindiniu mazgu [Red18].

Kitas „Redis“ sistemos apsisaugojimo nuo duomenų praradimo būdas – duomenų saugomų atmintyje išsaugojimas į failus diske. Duomenų išsaugojimas yra dviejų tipų:

- **RDO** – tam tikru metu yra padaroma momentinė duomenų kopija (angl. *snapshot*) ir ji yra išsaugojama į failą;
- **AOF** – kiekvienas duomenų pakeitimas nuolatos yra išsaugojamas į failą.

AOF tipo duomenų saugojimo naudojimas žymiai sulėtina sistemos veikimą [CTW+16].

1.1.4. Mazgų tarpusavio komunikacija

Mazgai tarpusavyje nuolatos siunčia *ping* ir *pong* žinutes, leidžiančias nustatyti, ar visi mazgai yra dabar aktyvūs. Jei dauguma (angl. *majority*) mazgų nustato, kad vienas iš pagrindinių mazgų tapo neaktyvus (baigėsi atsako laukimo laikas (angl. *timeout*)), visi kiti mazgai vykdo balsavimą, kuris neaktyvaus pagrindinio mazgo pavaldus mazgas taps nauju pagrindiniu mazgu [CTW+16; Red18].

Nustatant mazgų būsenas ir įvykių tvarką, yra naudojama epochos (angl. *epoch*) koncepcija – skaičius, kuris papildo ir identifikuoja esamą blokinio arba mazgo būseną. Kai keletas mazgų pateikia konfliktuojančią informaciją, epochos padeda nustatyti, kuri būsena (blokinio konfigūracija) yra pati naujausia, kadangi pasikeitus blokinio būsenai pasikeičia ir epochos reikšmė. Taip

pat mazgai remdamiesi blokinio epocha gali priimti bendrą sprendimą, nes naujausia informacija visada yra su didesne epocha [Red18].

Komunikuojant tarp mazgų, yra naudojamos dvi epochos: *currentEpoch* ir *configEpoch*. Mazgui gavus pranešimą iš kito mazgo ir nustačius, kad lokali mazgo *configEpoch* epocha yra didesnė nei epocha gauta iš siuntėjo, lokali *currentEpoch* epochos reikšmė yra atnaujinama į siuntėjo siųstą epochą. Vykdant šiuos veiksmus, galiausiai visi mazgai sutars dėl bendros blokinio *currentEpoch* epochos, t.y. ji bus lygi vieno iš blokinio mazgų didžiausiai *configEpoch* reikšmei [Red18].

Siunčiant *ping* ir *pong* žinutes, pagrindiniai mazgai visada siunčia savo *configEpoch* epochos reikšmę. Pavaldūs mazgai, siunčiant *ping* ir *pong* žinutes, taip pat prideda *configEpoch* reikšmes, kurios sutampa su pagrindinio mazgo epocha [Red18].

ConfigEpoch epochos reikšmė yra atnaujinama, kai pavaldus mazgas tampa pagrindiniu mazgu. Pavaldus mazgas dalyvaudamas rinkimuose, padidina savo *configEpoch* epochos reikšmę ir bando būti išrinktas nauju pagrindiniu mazgu. Kai pavaldus mazgas yra išrenkamas, nauja unikali *configEpoch* epochos reikšmė yra sukuriama ir pavaldus mazgas tampa nauju pagrindiniu mazgu [Red18].

Kuriant naujas *configEpoch* epochos reikšmes, svarbu užtikrinti, kad jos būtų unikalios tarp visų pagrindinių mazgų. Siekiant tai užtikrinti, yra naudojamas konfliktų sprendimo algoritmas. Šio algoritmo metu, jei pagrindinis mazgas nustato, kad blokinyje egzistuoja kitas pagrindinis mazgas, kurio *configEpoch* epocha yra tokia pati ir mazgo identifikatorius yra mažesnis nei surasto mazgo, kuris turi tokią pačią epochą, tai tada mazgas padidina savo epochą vienetu. Jei egzistuoja keletas tokių mazgų, kurių epochos sutampa, tai, remiantis šiuo algoritmu, jie galiausiai visi pasirinks skirtingas *configEpoch* epochos reikšmes [Red18].

Siunčiant *ping* ar *pong* žinutes yra siunčiama ne tik mazgo epocha, bet taip pat yra pridedama informacija apie maišos lizdus, už kuriuos yra atsakingas tas mazgas. Gavęs šią žinutę iš kito mazgo, mazgas atnaujina savo maišos lizdų lentelę, saugančią informaciją, kuris mazgas yra atsakingas už kurį maišos lizdą, remiantis dvejomis taisyklėmis:

- **Pirmoji taisyklė** – jei už maišos lizdą nėra atsakingas joks mazgas ir buvo gauta žinutė, kurioje mazgas nurodo, kad jis yra atsakingas už tam tikrą maišos lizdą, tai mazgas atnaujina savo maišos lizdų lentelę ir susieja tą lizdą su tuo mazgu;
- **Antroji taisyklė** – jei maišos lizdas yra priskirtas konkrečiam mazgui ir kitas mazgas, su didesne epocha nei dabar turi lizdui priskirtas pagrindinis mazgas, nurodo, kad jis dabar yra atsakingas už tą maišos lizdą, tai mazgas atnaujina savo maišos lizdų lentelę ir susieja tą lizdą

su nauju mazgu.

Remiantis antrąja taisykle yra užtikrinamas maišos lizdų lentelės atnaujinimas visuose mazguose, kai yra išrenkamas naujas pagrindinis mazgas. Taip pat panašiai yra vykdomas maišos lizdų aibės pertvarkymas (angl. *resharding*) – atnaujinus mazgo maišos lizdų aibę, yra padidinama mazgo epocha ir, remiantis antrąja taisykle, kiti mazgai atnaujina savo maišos lizdų lenteles [Red18].

Antroji taisyklė taip pat bus naudojama tuo atveju, kai nepasiekiamas mazgas sugrįš į blokinį. Jam sugrįžus, jis pradės siųsti *ping* žinutes kitiems mazgams, kuriose nurodys, kad jis yra atsakingas už tam tikrą aibę maišos lizdų. Kiti mazgai gavę žinutes matys, kad sugrįžusio mazgo blokinio epocha yra mažesnė nei jų ir kad už tuos maišos lizdus yra atsakingas kitas naujai išrinktas pagrindinis mazgas. Nustačius tai, blokinio mazgai sugrįžusiam mazgui išsiųs *update* žinutę, kurioje nurodys, kad naujai išrinktas mazgas yra atsakingas už tuos maišos lizdus. Gavęs šią žinutę ir remiantis antrąja taisykle, sugrįžęs mazgas atnaujins savo maišos lizdų lentelę ir nebebus atsakingas už nei vieną maišos lizdą. Sugrįžęs pagrindinis mazgas, būdamas tokioje būsenoje, tampa pavaldžiu mazgu, remiantis rolės pasikeitimo taisykle: sugrįžęs pagrindinis mazgas tampa pavaldžiu mazgu to naujo pagrindinio mazgo, kuris tapo atsakingas už paskutinį maišos lizdą. Ši taisyklė išsprendžia situaciją, kai pagrindinis mazgas buvo neaktyvus labai ilgą laiko tarpą ir jo maišos lizdų aibė buvo padalinta per kelis mazgus. Sugrįžę pavaldūs mazgai elgiasi taip pat – tampa pavaldžiais mazgais to pagrindinio mazgo, kuris tapo atsakingas už paskutinį maišos lizdą seno pagrindinio mazgo [Red18].

1.1.5. Naujo pagrindinio mazgo rinkimas

Naujo pagrindinio mazgo išrinkimas yra atliekamas pavaldžių mazgų. Jiems tai padeda atlikti kiti pagrindiniai mazgai, kurie išrenka naują pagrindinį mazgą iš nepasiekiamo mazgo pavaldžių mazgų aibės. Pagrindiniams mazgams renkant naują pagrindinį mazgą, tik vienas pavaldus mazgas gali tapti nauju pagrindiniu mazgu [Red18].

Naujo pagrindinio mazgo rinkimai prasideda tada, kai pavaldus mazgas nustato, kad jo pagrindinis mazgas, kuris yra atsakingas už netuščią maišos lizdų aibę, yra neaktyvus, ir kai jam nepavyksta užmegzti kontakto su pagrindiniu mazgu per ilgesnį nei numatytą ir sukonfigūruotą laiko tarpą. Laukiamas laiko tarpas taip pat priklauso nuo mazgo rango (angl. *rank*). Pavaldaus mazgo rango reikšmė priklauso nuo to, kokio naujumo pagrindinio mazgo duomenis turi pavaldus mazgas: mazgo, kuris turi pačius naujausius pagrindinio mazgo duomenis, rangas yra lygus 0, antrojo mazgo, kuris turi šiek tiek senesnius pagrindinio mazgo duomenis, rangas yra lygus 1 ir t.t. Remiantis rangais, mazgas, kuris turi mažiausią rango reikšmę, pirmasis pradeda dalyvauti rinkimuose prieš

kitus pavaldžius mazgus. Dalyvaudamas rinkimuose, pavaldus mazgas padidina savo epochą ir kreipiasi į kitus pagrindinius mazgus. Surinkęs daugumą pagrindinių mazgų balsų, pavaldus mazgas tampa nauju pagrindiniu mazgu [Red18].

Išrinkus naują pagrindinį mazgą, jo epocha bus didesnė nei kituose pagrindiniuose mazguose esanti *currentEpoch* reikšmė, todėl mazgas, norėdamas pagreitinti blokinio mazgų būsenos atnaujinimą, išsiunčia atnaujinimo žinutes visiems blokinio mazgams. Mazgai gavę šias žinutes matys, kad skiriasi mazgo epocha, ir remiantis tuo atnaujins lokaliai saugomą blokinio būseną. Senojo pagrindinio mazgo pavaldūs mazgai gavę tokią žinutę taip pat atnaujins savo būseną bei persikonfigūruos – pradės vykdyti naujo pagrindinio mazgo duomenų replikavimą [Red18].

1.1.6. „Redis Cluster“ savybių apibendrinimas

„Redis Cluster“ yra išskirstyta sistema, kurioje veiksmai vyksta lygiagrečiai, mazgai komunikuoja tarpusavyje žinutėmis, vyksta būsenų sinchronizacija, saugomi duomenys yra išskirstyti tarp keleto mazgų. Toks sistemos pobūdis (sistemos architektūra) gali nulemti tai, kad sistema tam tikrais atvejais neveiks korektiškai – nebus užtikrintos sistemai būdingos savybės [BMP18; Mal16].

Remiantis ankstesniuose poskyriuose pateikta informacija, galima apibendrinti bendrąsias „Redis Cluster“ sistemos savybes, turinčias įtakos išskirstytos sistemos veikimui:

- „Redis Cluster“ klientai gali kreiptis į bet kurį blokinio mazgą;
- Visi mazgai saugo informaciją apie kitus mazgus ir visi mazgai yra pilnai tarpusavyje sujungti komunikavimo kanalais;
- Komunikuodami tarpusavyje, ilgainiui mazgai sutaria dėl kiekvieno mazgo būsenos;
- Sistemoje yra naudojami dviejų tipų mazgai: pagrindiniai ir pavaldūs mazgai, kurie saugo pagrindinio mazgo duomenų kopijas – replikas;
- Naujo mazgo pridėjimas, pašalinimas ar maišos lizdų aibės pertvarkymas nereikalauja, kad „Redis Cluster“ sistema būtų išjungta ar sustabdyta;
- „Redis Cluster“ sistemoje galima sukonfigūruoti automatinį replikų migravimą.

Taip pat galima išskirti ir apibendrinti „Redis Cluster“ savybes, kurios turi būti užtikrintos, kad sistema veiktų korektiškai:

- Už vieną maišos lizdą gali būti atsakingas tik vienas pagrindinis mazgas ir jo pavaldūs mazgai;

- „Redis Cluster“ sistemoje yra užtikrinamas galutinis duomenų neprieštaringumas;
- Naudojant sinchroninį duomenų įrašymą, yra užtikrinamas didesnis duomenų neprieštaringumas, tačiau ne visuomet yra užtikrinamas griežtas duomenų neprieštaringumas;
- Neaktyviam pagrindiniam ar pavaldžiam mazgui sugrįžus į blokinį, jis tampa pavaldžiu mazgu to pagrindinio mazgo, kuris tapo atsakingas už paskutinį maišos lizdą;
- Mazgas iš „Redis Cluster“ blokinio gali būti pašalintas tik tada, kai jis yra tuščias;
- „Redis Cluster“ sistema tampa neveiksni, jeigu dauguma pagrindinių mazgų tampa neaktyvūs;
- Sukonfigūruotas replikų migravimas užtikrina, kad galiausiai kiekvienas pagrindinis mazgas turės bent po vieną pavaldų mazgą;
- Pavaldaus mazgo *configEpoch* epochos reikšmė sutampa su pagrindinio mazgo *configEpoch* epochos reikšme;
- Visų pagrindinių mazgų *configEpoch* epochos reikšmės turi būti unikalios;
- Blokinio *currentEpoch* epochos reikšmė yra lygi didžiausiai blokinio mazgo *configEpoch* epochos reikšmei;
- Pavaldžiam mazgui nustačius, kad jo pagrindinis mazgas yra nepasiekiamas, yra pradedamas rinkti naujas pagrindinis mazgas;
- Pavaldus mazgas, turintis naujausius neaktyvaus pagrindinio mazgo duomenis, pirmasis pradeda dalyvauti naujo pagrindinio mazgo rinkimuose;
- Renkant naują pagrindinį mazgą, tik vienas, daugumą pagrindinių mazgų balsų surinkęs, pavaldus mazgas gali tapti nauju pagrindiniu mazgu.

1.2. Kitos išskirstytos podėlio sistemos

Egzistuoja ir kiti produktai, kuriuos galima naudoti kaip išskirstytą podėlį, pavyzdžiui, „Ehcache“. „Ehcache“ – tai taip pat atvirojo kodo Java kalba parašytas podėlis, dažnai naudojamas kuriant saityno (angl. *web*) programas. Šiame podėlyje duomenis galima laikyti tiek operatyviojoje atmintyje, tiek standžiajame diske (angl. *hard disk*) [Ehc19].

Kaip ir „Redis“, „Ehcache“ gali būti išskirstytas, kai grupė „Ehcache“ egzempliorių – mazgų, veikia kaip išskirstytas podėlis. Norint tai pasiekti, reikia naudoti „Terracotta“ serverį, kurį sukonfigūravus bus sudarytas „Ehcache“ blokinys – „Terracotta“ serveris sujungs kelis „Ehcache“

mazgus ir tų mazgų sujungti duomenys bus pasiekiami kreipiantis į „Terracotta“ serverį. Taip pat „Terracotta“ serveryje bus sukonfigūruota bendra atmintis, kuri bus pasiekiama visiems „Ehcache“ mazgams. Vienam mazgui padarius pakeitimus „Terracotta“ serverio bendroje atmintyje, pakeitimai bus matomi kitiems „Ehcache“ mazgams [DWL+10; Ehc19].

Naudojant „Ehcache“ ir norint pasiekti aukštą prieinamumą, taip pat galima sukonfigūruoti saugomų duomenų replikavimą per kelis „Ehcache“ mazgus. Sukonfigūravus replikavimą, kiekvienas iš mazgų yra lygiavertis prieš kitus mazgus, t.y. nėra pagrindinio mazgo. Be to, vykdomas duomenų replikavimas gali būti tiek sinchroninis, tiek asinchroninis [DWL+10; Ehc19].

Kitas produktas, kurį taip pat galima naudoti kaip išskirstytą podėlį, yra „Memcached“. „Memcached“ – tai raktas-reikšmė duomenų saugykla, duomenis sauganti operatyviojoje atmintyje. Duomenų saugykloje saugomi duomenys: reikšmės ir raktai, gali būti tik simbolių eilutės tipo, todėl negalima saugoti sudėtingesnės struktūros duomenų lyginant su „Redis“ sistema [Car13]. Dažniausiai „Memcached“ yra naudojamas kaip podėlis nuošalyje, siekiant sumažinti duomenų bazės apkrovą (angl. *load*) bei siekiant pagerinti visos sistemos našumą [AXF+12; DWL+10].

Kaip ir „Redis Cluster“ sistemoje, „Memcached“ išskirstyto podėlio struktūra yra vienoda – ją sudaro klientai ir serveriai. Tačiau, lyginant su „Redis Cluster“ sistema, „Memcached“ sistemos įgyvendinimas yra tiek klientuose, tiek ir serveriuose. Klientai, norėdami padėti ar perskaityti tam tikrą reikšmę išskirstytame podėlyje, apskaičiuoja maišos funkciją, kuri pagal raktą nusprendžia, į kurį serverį reikia kreiptis. Klientas, norėdamas naudojantis maišos funkcija pasirinkti serverį pagal raktą, turi žinoti, kiek išskirstytoje sistemoje yra serverių, ir kokie yra tų serverių adresai [Mem19].

Skirtingai nei „Redis Cluster“ sistemoje „Memcached“ sudarytame blokinyje mazgai tarpusavyje nekomunikuoja – nėra vykdoma jokia sinchronizacija. Taip pat nėra atliekamas mazgo duomenų replikavimas, todėl „Memcached“ išskirstytas podėlis negali užtikrinti pakankamai aukšto prieinamumo [AXF+12; Mem19].

2. Formalūs metodai

Dažnai kuriant sistemas yra reikalaujama pakankamai aukštos kokybės už priimtiną kainą ir laiką. Taip pat egzistuoja sistemos, pavyzdžiui, susijusios su telekomunikacija, medicina, kosmosu, kurios turi turėti aukšto lygio patikimumą ir saugumą. Formalūs metodai gali padėti kuriant tokio tipo sistemas [TS98]. Formalūs metodai – tai matematiniu pagrindu paremti metodai, dažniausiai naudojami aprašant sistemos savybes ir elgseną. Sistemų kūrėjai, taikydami formalius metodus, gali sistematiškai specifikuoti, kurti ir verifikuoti sistemas [Win90].

Formalius metodus galima taikyti visų programų sistemų kūrimo etapų metu, tačiau kuo ankstesnis jų taikymas gali suteikti daugiau naudos visos sistemos kūrimui [WLB+09]. Formalūs metodai gali būti naudojami analizuojant naudotojo reikalavimus, projektuojant, įgyvendinant, testuojant, palaikant ir verifikuojant sistemas [Win90]. Reikalavimų rinkimo metu formalūs metodai gali padėti nustatyti reikalavimų išbaigtumą, atsekamumą (angl. *traceability*), neprieštaringumą ir pakartotinį panaudojamumą (angl. *reusability*) [WLB+09]. Kuriant sistemas formalūs metodai gali padėti surasti projektavimo spragas, o, taikant formalius metodus sukurtai sistemai, gali padėti nustatyti, ar sistema buvo įgyvendinta korektiškai. Tuo tarpu palaikant sistemas, formalūs metodai gali padėti lengviau ir greičiau daryti sistemos patobulinimus ar sistemos optimizavimą, kadangi taikant formalius metodus galima įvertinti, ar sistema po įvykdytų pakeitimų vis dar atitinka jai keliamus reikalavimus [Win90].

Taikant formalius metodus, yra sukuriama formali specifikacija, kuri matematinėmis išraiškomis aprašo sistemą. Pati formali specifikacija nusako, ką sistema turi daryti (kokios savybės yra visada užtikrinamos, kokie turi būti įvedimo ar išvedimo duomenys ir kita), tačiau nenusako, kaip tai bus įgyvendinta [WLB+09]. Turint formalią specifikaciją taip pat galima lengviau vykdyti aiškų ir tikslų komunikavimą tiek su užsakovu, tiek tarp sistemos kūrėjų [AMI17].

Sukūrus formalią specifikaciją taip pat svarbu nustatyti, ar parašyta formali specifikacija yra adekvati – tinkamai ir pilnai aprašo nagrinėjamus sistemos aspektus, sistemos reikalavimus. Sukurtos formalios specifikacijos adekvatumas gali būti patikrintas atlikus jos validavimą. Egzistuoja keli būdai kaip tai galima atlikti: prototipų kūrimas, struktūrinė peržvalga (angl. *structured walkthrough*), specifikacijos peržiūra, žurnalų palyginimas ir kiti [MJS09; TYB+02].

Dažnai formalūs metodai taip pat turi ir įrankius, leidžiančius analizuoti formalios specifikacijos sintaksę ar semantiką [Win90]. Taip pat egzistuoja formalios kalbos, kurios turi įrankius, leidžiančius pagal formalią specifikaciją sugeneruoti programos kodą, kuris tenkins visus formalioje specifikacijoje apibrėžtus sistemos reikalavimus, savybės ir apribojimus. Tokio įrankio taikymas žymiai palengvina sistemos kūrimo pradžią [WLB+09].

2.1. Formalios specifikavimo kalbos

Taikant formalius metodus ir rašant formalias specifikacijas, yra naudojamos formalios specifikavimo kalbos. Kiekviena formali specifikavimo kalba yra nedviprasmiška, kad skaitant ja parašytą formalią specifikaciją kiekvienas skaitytojas suprastų ją vienodai, bei prasminga, kad ja parašytą formalią specifikaciją būtų galima įgyvendinti [Win90]. Specifikuojant konkrečią sistemą dažniausiai yra naudojama viena formali specifikavimo kalba, tačiau gali atsitikti taip, kad neužteks naudoti tik vienos formalios specifikavimo kalbos, kuri padengtų visus nagrinėjimus sistemos aspektus. Tokiu atveju yra patartina naudoti kelias formalias specifikavimo kalbas, kurios padės lengviau ir aiškiau specifikuoti sistemos elgseną [TS98].

Formalios specifikavimo kalbos yra paremtos įvairiomis matematikos sritimis, pavyzdžiui, algebra, logika, aibių teorija ir kitomis. Kiekviena formali specifikavimo kalba yra sudaryta iš trijų pagrindinių komponentų: sintaksės, kuri nurodo žymėjimus (angl. *notation*), kuriais yra rašoma formali specifikacija, formalios kalbos semantikos (prasmės), kuria dažniausiai skiriasi formalios specifikavimo kalbos, ir semantikos taisyklių [AMI17; Win90]. Formalios specifikavimo kalbos sintaksės domeną dažniausiai sudaro matematiniai teksto simboliai, tačiau taip pat egzistuoja formalių specifikavimo kalbų, kurių sintaksė yra paremta ir grafiniais elementais [Win90].

2.2. TLA$^+$ kalba

Viena iš tokių formalių specifikavimo kalbų, leidžiančių specifikuoti išskirstytas sistemas ar lygiagrečiai veikiančius algoritmus, yra TLA$^+$ kalba [Mos13]. TLA$^+$ formali specifikavimo kalba yra paremta aibių teorija, predikatų logika ir veiksmų laiko logika, kuri yra naudojama aprašant ir nagrinėjant lygiagrečias ir išskirstytas sistemas [LMT+02]. Ši formali specifikavimo kalba yra labai lanksti, kadangi specifikuojant sistemas galima pasirinkti norimą specifikuojamos sistemos abstrakcijos lygmenį [NRZ+15].

TLA$^+$ kalba parašyta formali specifikacija apibūdina būsenų mašiną (angl. *state machine*) ir yra išreiškiama viena matematine formule. Dažniausiai formali specifikacija turi tokią formą $\mathrm{Init} \land \Box[\mathrm{Next}]_{\mathrm{vars}} \land \mathrm{Liveness}$, kur:

- Init – pradinės būsenos predikatas;
- Next – disjunktas, nusakantis sistemos elgseną – perėjimą iš vienos būsenos į kitą;
- $\mathrm{Liveness}$ – laiko logikos formulė.

TLA$^+$ formali specifikavimo kalba taip pat turi įrankius, kurie padeda lengviau ją taikyti. Vienas iš jų – TLC modelio tikrintojas, kuris, remiantis formalia specifikacija, sugeneruoja visas

galimas pasiekti sistemos būsenas. Naudojantis TLC įrankiu, jei sistemos veikimas priklauso nuo tam tikros konfigūracijos, pavyzdžiui procesorių, mazgų skaičiaus, ir jei galima sistemos būsenų aibė nėra baigtinė, tada, atliekant modelio tikrinimą, reikia nurodyti konfigūraciją, kuri sumažintų tikrinamų būsenų aibę, t.y. padarytų ją baigtine. Tačiau TLC įrankį galima taikyti ir tuo atveju, kai būsenų aibė yra begalinė. Tada nutraukus modelio tikrinimą, bus patikrinta tiktais baigtinio ilgio atsitiktinė sistemos simuliacija, t.y. ne visi galimi variantai [LMT+02].

Rašant formalią specifikaciją TLA$^+$ kalba, galima apibrėžti sistemos saugumo ir gyvybingumo (angl. *liveness*) savybes [LMT+02]. Saugumo savybė nusako, ką sistema gali daryti, o gyvybingumo savybė nusako, ką sistema galiausiai turi padaryti. Pavyzdžiui, jei sistema gauna užklausą tai į ją galiausiai turi atsakyti [NRZ+15]. Tikrinamos gyvybingumo savybės yra skirstomos į dvi grupes: galutinumo (angl. *enventually*) ir sąžiningumo (angl. *fairness*). Galutinumo savybė nusako, kad kažkas ilgainiui įvyks, o sąžiningumo savybė nusako, kad jei kažkoks įvykis yra įvykęs, tai jis negali būti be galo ignoruojamas [Mos13].

TLC įrankis, generuodamas visas galimas sistemos būsenas, kiekvienos galimos būsenos metu tikrina saugumo savybes. Stebint šio tipo savybes yra vertinama, ar visose būsenose yra tenkinami sistemos invariantai ir taip pat, ar nėra pasiekta sistemos aklavietė. Tuo tarpu gyvybingumo savybės yra stebimos periodiškai, kadangi sudarytas būsenų grafas gali būti per didelis TLC įrankiui [LMT+02].

Nors modelio tikrinimas leidžia pilnai automatizuoti formalios specifikacijos tikrinimą (įrankis perrenka visas galimas sistemos būsenas), tačiau modelio tikrinimas turi ir vieną didelį trūkumą – būsenų sprogimo problemą [MKE18]. Ši problema pasireiškia tuo, kad su pakankamai mažomis konfigūracijomis visų galimų sistemos būsenų patikrinimą atlikti yra pakankamai lengva ir greita, tačiau, didinant formalios specifikacijos konfigūraciją, pavyzdžiui, didinant sistemos mazgų skaičių, labai greitai pradeda didėti galimų būsenų skaičius. Būtent dėl to yra sunku patikrinti didelės sistemos su daug galimų būsenų teisingumą [LMT+02]. Tokioje situacijoje reikia vykdyti formalios specifikacijos savybių įrodinėjimą. Kadangi įrodinėjimas nėra susietas su fiksuota konfigūracija, įrodinėjimo metu bus įrodyta, kad savybės yra tenkinamos su bet kokia sistemos konfigūracija. Tačiau specifikacijos savybių įrodinėjimas taip pat turi ir trūkumą – reikalauja specifikuotojo patirties ir užima daugiau žmogaus laiko ir pastangų lyginant su modelio tikrinimu [MKE18].

Įrodinėjant formalios specifikacijos formulę yra patartina naudoti hierarchinį įrodinėjimo metodą, kurio metu formalios specifikacijos formulė yra įrodinėjama suskaidant įrodymą į dalis, kurias įrodžius yra įrodoma ir pagrindinė formulė [LMT+02]. Taip pat vykdant įrodinėjimą galima naudoti kitą įrankį – TLA$^+$ įrodinėjimo sistemą (angl. *TLA$^+$ Proof System*, sutrumpintai – TLAPS) [MKE18].

Siekiant pagreitinti visų galimų būsenų tikrinimą ir taip pat susidoroti su būsenų sprogimo problema, TLC įrankis išnaudoja daugiagijiškumą bei būsenų tikrinimą galima paleisti ant kelių mašinų. Taip pat, norint neprarasti jau sugeneruotų būsenų, TLC įrankis generuojant visas būsenas sukuria kontrolinius punktus (angl. *check point*). Sustabdžius būsenų tikrinimą, jį galima vėl paleisti vykdyti nuo tam tikro kontrolinio punkto [LMT+02].

Naudojant TLA$^+$ formalią specifikavimo kalbą, galima atlikti formalios specifikacijos tobulinimą (angl. *refinement*). Norint atlikti formalios specifikacijos tobulinimą, reikia įrodyti, kad $\mathrm{Spec}(\mathrm{S}_2) \Rightarrow \mathrm{Spec}(\mathrm{S}_1)$, kur S_2 yra nauja patobulinta specifikacija, o S_1 – sena specifikacija. Kitaip tariant, patobulinta specifikacija S_2 išlaiko visas S_1 specifikacijos savybes tik tada, kai kiekvienai formulei F, jei $\mathrm{Spec}(\mathrm{S}_1) \Rightarrow \mathrm{F}$ teisinga, tai ir $\mathrm{Spec}(\mathrm{S}_2) \Rightarrow \mathrm{F}$ taip pat yra teisinga [Mos13].

TLA$^+$ formalioje kalboje specifikacijos tobulinimas yra atliekamas aprašant tobulinimo atvaizdį (angl. *refinement mapping*). Tobulinimo atvaizdį sudaro taisyklės nusakančios kaip detalios specifikacijos būsenas susieti su abstrakčios specifikacijos būsenomis – taisyklės aprašo, kaip vienos specifikacijos kintamuosius galima išreikšti kitos specifikacijos kintamaisiais. Aprašius atvaizdžio sudarymo taisykles, TLA$^+$ specifikacijos tobulinimą galima išreikšti implikacija, kad bet kokia elgsena galima detalios specifikacijos taip pat yra galima ir abstrakčioje specifikacijoje atlikus detalios specifikacijos kintamųjų atvaizdavimą. Atlikto specifikacijos tobulinimo teisingumą taip pat galima patikrinti TLC įrankiu abstrakčios specifikacijos formulę (su atvaizdžio taisyklėmis) tikrinant kaip gyvybingumo savybę [LM17].

TLA$^+$ formalia specifikavimo kalba parašytą formalią specifikaciją pakankamai lengva suprasti ir šia kalba pakankamai lengvai naudojasi inžinieriai, kadangi dažniausiai jie jau yra susipažinę su aibėmis ir paprasta diskrečiąja matematika [NRZ+15]. Taip pat egzistuoja daug sėkmingų pavyzdžių, kai TLA$^+$ formali specifikavimo kalba yra naudojama sudėtinguose ir dideliuose projektuose. Pavyzdžiui, TLA$^+$ formalią specifikavimo kalbą sėkmingai naudojasi „Amazon“, „Compaq“, „Intel“ ir „Microsoft“ kompanijos [MKE18].

„Amazon Web Services“ testuojant sistemų teisingumą naudojo standartines verifikavimo technikas (architektūros ir kodo peržiūras, statinę kodo analizę ir įvairias testavimo technikas), tačiau pastebėjo, kad jos ne visada suranda sudėtingų, lygiagrečiai veikiančių sistemų klaidas. Taip atsitiko dėl to, kad standartinės verifikavimo technikos yra paremtos žmogaus intuicija ir žmogus negali išmąstyti ir ištestuoti visų galimų sistemos atvejų. „Amazon“ siekdama išspręsti šią problemą bei norėdama susidoroti su tuo, kad išskirstytų sistemų klaidos nepatektų į produkciją, pradėjo taikyti TLA$^+$ formalią specifikavimo kalbą [NRZ+15].

TLA$^+$ formalios specifikavimo kalbos taikymas „Amazon“ kompanijai padėjo surasti tokias sistemos klaidas, kurių kitomis technikomis nebūtų radusi, bei leido atlikti kuriamų sistemų našu-

mo optimizavimą, išlaikant sistemos korektiškumą. Taip pat, pasak „Amazon“, formalaus metodo naudojimas padėjo jiems geriau suprasti kuriamų sistemų architektūrą, padėjo patikrinti architektūros korektiškumą bei taip pat padėjo dokumentuoti kuriamas sistemas, kadangi parašyta formali specifikacija yra tiksli, trumpa ir ją galima lengvai peržiūrėti ar išbandyti naudojant TLA$^+$ kalbos įrankius [NRZ+15].

3. Formalios specifikacijos

Šiame skyriuje yra apžvelgiamos šaltiniuose rastos išskirstytų podėlių bei sistemų, dirbančių su raktas-reikšmė duomenimis, formaliais metodais paremtos analizės. Aptariant šaltinius, yra siekiama išsiaiškinti: kaip buvo atliekama formaliais metodais paremta analizė, kokios savybės buvo tiriamos analizės metu ir kaip buvo modeliuojamos bei formalizuojamos tokio tipo sistemos.

3.1. Išskirstytų podėlių formalios specifikacijos

Straipsnyje [VZL08] yra analizuojamas aukšto našumo, daugiamatis išskirstytas podėlis „LambdaRAM“. Ši sistema, naudojant didelio greičio optinius tinklus, sujungia keletą išskirstytų RAM atminčių į vieną bendrą blokinį. Tokia sistemos architektūra pagreitina didelių geofizikos ir biologijos mokslo duomenų analizavimą.

„LambdaRAM“ sistemos analizei buvo naudojami formalūs metodai, siekiant išsiaiškinti ir pašalinti problemas, kurios atsiranda, kai sistemos konfigūraciją sudaro daug mazgų. Iš pradžių, remiantis programiniu kodu, buvo sukurtas sistemos modelis, kuriuo buvo siekiama supaprastinti sistemą – padaryti jos abstrakciją. Norint tai atlikti buvo padarytos prielaidos, leidusios lengviau atlikti sistemos verifikavimą. Verifikuojant sistemą buvo tiriamos dvi sistemos savybės: saugumo ir gyvybingumo. Tiriant saugumo savybę buvo norima įsitikinti, kad niekada nėra viršijamas maksimalus galimų podėlio blokų skaičius, o tiriant gyvybingumo savybę, buvo norima įsitikinti, kad kiekvienas prašomas podėlio blokas yra galiausiai suteikiamas. Saugumo savybės verifikavimas buvo atliktas ją pavertus invariantu ir patikrinus TLV įrankiu – modelio tikrintoju, kuris naudoja SMV kalbą. Tuo tarpu, tikrinant gyvybingumo savybę, ši savybė buvo paversta saugumo invariantu, kurio verifikavimas buvo atliktas vykdant įrodinėjimą. Atlikus sistemos verifikavimą, naudojant formalias metodais, buvo rasta viena sistemos klaida, susijusi su gyvybingumo savybe, apie kurią buvo pranešta „LambdaRAM“ kūrimo komandai.

Kitame straipsnyje [JaJT+03] yra analizuojami podėlio darnumo (angl. *coherence*) protokolų įgyvendinimai: „EV6“, „EV7“ ir „Itanium“, naudojant TLA^{+} formalią specifikavimo kalbą ir TLC įrankį. Podėlio darnumo protokolai yra naudojami bendros atminties daugiaprocesoriuose (angl. *multiprocessors*), kurių kiekvienas turi po lokalų privatų podėlį. Podėlio darnumo protokolas užtikrina, kad jei keletas procesoriaus podėlių turi to pačio atminties adreso duomenų kopijas, tai, pasikeitus tai atminties daliai, visos kopijos turi išlikti neprieštaringos [AB86].

Pirmasis straipsnyje [JaJT+03] nagrinėtas protokolas yra „EV6“. Nagrinėjant šį protokolą iš pradžių autoriai, stengdamiesi suprasti patį protokolą, sukūrė protokolo aprašymą. Remiantis šiuo aprašymu ir taip pat sistemos simuliatoriumi, buvo sukurta sistemos specifikacija. Siekiant dar

labiau ją supaprastinti, grupuojant žinučių skaičius buvo sumažintas nuo šešiasdešimt iki penkiolikos žinučių. Taip pat protokolui patikrinti buvo sukurtas „Alpha“ atminties modelį specifikuojantis TLA$^+$ modelis. Atminties modelį sudarė būsenų mašina, kuri gaudavo užklausas, nustatydavo reikšmes ir sugeneruodavo atsakymus. Sudarius formalią specifikaciją buvo įrodinėjami du invariantai, susiję su žinučių perdavimu, kuriame dažniausiai buvo randamos sistemos klaidos. Kadangi dar TLC įrankis nebuvo sukurtas, šie du invariantai buvo patikrini įrodymu pagalba. Atlikus įrodymą buvo rastos dvi protokolo klaidos.

Antrasis straipsnyje [JaJT+03] nagrinėtas podėlio darnumo protokolas yra „EV7“. Kūrėjai kurdami šį protokolą patys naudojo formalią specifikavimo kalbą TLA$^+$, todėl straipsnio autoriams, norint patikrinti šį protokolą, nebereikėjo rašyti formalios specifikacijos. Tačiau straipsnio autoriai prieš atlikdami modelio tikrinimą, naudojant TLC įrankiu, vis dėlto šiek tiek patobulino specifikaciją, kadangi protokolo kūrėjai, rašydami specifikaciją, neatsižvelgė į simetrijos apribojimus. Specifikacijos patobulinimas leido sumažinti galimų būsenų aibę beveik per pusę. Tikrinant modelį buvo naudojamas ne tik TLC įrankis. Taip pat protokolo klaidų ieškojimui buvo naudojami TLC įrankio sugeneruoti pėdsakai (angl. *traces*). Šie pėdsakai buvo pateikiami protokolo simuliatoriui, kuris bandydavo atkartoti TLC įrankio sugeneruotas sistemos būsenas [TYB+02]. Toks sugeneruotų pėdsakų panaudojimas taip pat padėjo rasti protokolo klaidas.

Paskutinis straipsnyje [JaJT+03] nagrinėtas podėlio darnumo protokolas yra „Itanium“. Šis protokolas yra paremtas komponentais, už kuriuos yra atsakingi kiti kūrėjai, todėl rašant formalią specifikaciją, teko abstrakčiai modeliuoti šių komponentų sąsajas (angl. *interfaces*). Verifikuojant protokolą, buvo naudojamas TLC įrankis, tačiau vis dėlto nepavyko įrodyti adekvataus sistemos teisingumo. Taip atsitiko dėl to, kad modelio tikrinimas užtruko per daug laiko, kadangi dėl abstrakčių išorinių komponentų sąsajų labai išaugo galimų būsenų skaičius. Taip pat norint sugeneruoti ir patikrinti sudėtingus atvejus, reikėjo verifikuoti konfigūraciją su daug procesorių, kas taip pat būtų lėmę galimų būsenų sprogimo problemą.

3.2. Raktas-reikšmė NoSQL duomenų bazių formalios specifikacijos

Straipsnyje [LaSG+14] yra analizuojama „Cassandra“ NoSQL duomenų bazė, kuri gali būti išskirstyta per keletą serverių ir kuri užtikrina didelį prieinamumą, tačiau pilnai neužtikrina griežto neprieštaringumo. Sistemos „Cassandra“ blokinį sudaro žiedo (angl. *ring*) forma išdėstyti mazgai, kurie yra padalinti į sritis (angl. *ranges*). Už tą pačią sritį gali būti atsakingi keli mazgai – replikos. Be to, „Cassandra“ sistemoje yra naudojama klientas-serveris struktūra: klientas gali kreiptis į bet kurį serverį, o serveris, gavęs užklausą, kreipiasi į kitą serverį, kuris yra atsakingas už tą raktą, ir

gavęs atsaką iš kito serverio, jį grąžina klientui.

Straipsnio [LaSG+14] autoriai, norėdami išanalizuoti „Cassandra“ sistemos nepriešiaringumą, iš pradžių remdamiesi programiniu kodu ir naudojant Maude formalią kalbą, paremtą perrašymo logika (angl. *rewriting logic*), sukūrė „Cassandra“ sistemos algoritmą formalizuojantį modelį. Sukurtą modelį sudarė pagrindiniai „Cassandra“ sistemos komponentai susiję su: skaidymo (angl. *partitioning*) strategija, nepriešiaringumo lygiais ir laiko žymų (angl. *timestamp*) strategija, kuri nusprendžia duomenų naujumą. Modeliuojant sistemą, taip pat buvo atsižvelgta į kliento-serverio struktūrą. Sudarytame modelyje klientai generuodavo rašymo ir skaitymo užklausas žinutėmis bei taip pat tikrindavo nepriešiaringumo pažeidimus. Tuo tarpu serveriai buvo modeliuojami lygiaverčiais, kadangi kiekvienas iš jų gali bendrauti su kiekvienu klientu. Kadangi komunikavimui tarp serverio ir kliento bei komunikavimui tarp serverių buvo naudojamos žinutės su tam tikra nedeterministiškai pasirinkta delsa (angl. *delay*), siekiant užtikrinti žinučių tvarką buvo pridėtas naujas sistemos objektas – planuotojas (angl. *scheduler*). Jis turėdamas globalų sistemos laikrodį, pagal kurį nuspręsdavo, ar žinutes jau galima siųsti ar ne, buvo atsakingas už žinučių persiuntimą.

Sukūrus „Cassandra“ sistemos modelį, buvo atliekamas modelio tikrinimas, kurio metu buvo stebima nepriešiaringumo savybė – buvo siekiama išsiaiškinti, kada sistema užtikrina griežtą ir kada užtikrina galutinį nepriešiaringumą. Analizės metu modelis buvo tikrinamas su skirtingomis konfigūracijomis: įvairiomis delsomis, nepriešiaringumo lygiais bei skirtingu skaičiumi klientų (vienu ir daug klientų). Atlikus modelio tikrinimą buvo gauti rezultatai, kad „Cassandra“ sistema visada užtikrina galutinį nepriešiaringumą, ir buvo identifikuoti atvejai, kada yra pažeidžiamas griežtas nepriešiaringumas.

Kitame straipsnyje [LNaM+15] taip pat buvo formaliai analizuojamas „Cassandra“ sistemos nepriešiaringumas, tačiau šį kartą analizė buvo kiekybinė. Analizės metu buvo siekiama išsiaiškinti kokiu lygiu iš tikrųjų yra užtikrinamas nepriešiaringumas įvairiomis sąlygomis. Tam atlikti sistema buvo modeliuojama kaip tikimybinė sistema, o formalizavimui buvo naudojama QUATEX tikimybinė laiko logika (angl. *probabilistic temporal logic*) ir statistinio modelio tikrinimo įrankis PVESTA, kuris statistinį modelio tikrinimą atlieka remiantis Monte-Carlo metodu paremtomis simuliacijomis. Kaip modelio pagrindas tapo straipsnyje [LaSG+14] naudotas modelis, o pagrindinis modelio pasikeitimas buvo: visos nedeterministinės taisyklės buvo paverstos į taisykles priklausomas nuo tikimybių. Atnaujinus modelį buvo atliktas statistinis modelio tikrinimas esant įvairioms sąlygoms: skirtingiems nepriešiaringumo lygiams ir delsai. Atlikus analizę buvo nustatyta, kad „Cassandra“ sistema, esant tam tikriems nepriešiaringumo lygiams, iš tikrųjų užtikrina didesnį nepriešiaringumą nei numatytą galutinį nepriešiaringumą.

Be to, straipsnyje [LNaM+15], siekiant nustatyti, ar statistinio modelio tikrinimo metu gauti

rezultatai atitinka realią sistemą, taip pat buvo atliktas eksperimentinis „Cassandra“ sistemos nepriештaringumo vertinimas. Modelio tikrinimo metu gauti rezultatai skyrėsi nuo eksperimentų metu gautų rezultatų apie 10-15 procentų.

3.3. Apibendrinimas

Literatūros apžvalgoje buvo apžvelgtos formaliais metodais paremtos analizės, formalios specifikacijos, nagrinėjančios panašias sistemas kaip „Redis Cluster“. Apžvalgos metu buvo rasti keli šaltiniai, kuriuose buvo nagrinėjamos išskirstytos podėlio sistemos, naudojant formalius metodus: SVL ir TLA$^+$ kalbas. Tačiau nagrinėjami išskirstyti podėliai buvo labai žemo lygio. Kadangi „Redis“ sistemoje duomenys yra saugomi raktas-reikšmė pavidalu, taip pat buvo apžvelgtos raktas-reikšmė NoSQL duomenų bazės „Cassandra“ formalios analizės, nagrinėjančios „Cassandra“ sistemos neprieštaringumą.

Ieškant ir nagrinėjant šaltinius, nepavyko rasti formalių specifikacijų, kurios nagrinėtų panašios struktūros ir savybių sistemas kaip „Redis Cluster“, todėl manome, kad sudaryta „Redis Cluster“ formali specifikacija bus pirmoji, kuri nagrinės šią sistemą.

4. „Redis Cluster“ sistemos formalios specifikacijos

Remiantis „Redis Cluster“ sistemos specifikacija ir kitais šaltiniais, buvo apžvelgta nagrinėjama „Redis Cluster“ sistema – buvo apibūdinta kokia yra sistemos architektūra ir kaip įvairiomis situacijomis veikia sistema. Galiausiai, remiantis išnagrinėtais šaltiniais, buvo identifikuotos ir suformuluotos savybės, kuriomis pasižymi „Redis Cluster“ sistema.

Tiriant šias savybes bus naudojami formalūs metodai, konkrečiai TLA$^+$ formali specifikavimo kalba. TLA$^+$ kalbą yra lengva suprasti ir taikyti, tiriant sistemas pasirinktu abstrakcijos lygiu. Taip pat, pagal rastus literatūros šaltinius galima matyti, kad ši formali specifikavimo kalba yra naudojama tiriant panašios struktūros ir pobūdžio išskirstytas sistemas kaip „Redis Cluster“ bei ši kalba yra sėkmingai naudojama praktikoje, tiriant realias problemas – TLA$^+$ kalbos taikymas „Amazon“ kompanijoje leido surasti nagrinėjamų išskirstytų sistemų klaidas.

Šiame skyriuje yra apžvelgiamos „Redis Cluster“ sistemos formalios specifikacijos parašytos TLA$^+$ formalia specifikavimo kalba. Apžvelgiant parašytas formalias specifikacijas, yra pateikiamos ir aptariamos tik esminės formalių specifikacijų dalys: parametrai (konstantos), nuo kurių priklauso specifikacijos, specifikacijos kintamieji, kurie apibūdina būsenų mašinos būseną tam tikru momentu, bei veiksmai (angl. *actions*), kurie modeliuoja sistemos būsenos pasikeitimus keičiant specifikacijos kintamųjų reikšmes. Detalias ir pilnas formalias specifikacijas su minimaliais komentarais galima rasti šio dokumento prieduose.

Tiriant „Redis Cluster“ sistemą buvo sudarytos dvi formalios specifikacijos: abstrakti ir detali specifikacijos. Abstrakčioje specifikacijoje buvo fokusuojamasi į tai, kokie reikalavimai yra keliami sistemai ir kaip sistemą bei sistemos būseną mato naudotojas, nesigilinant į sistemos įgyvendinimą. Vėliau remiantis abstrakčia specifikacija ir sistemos kodu buvo sudaryta antroji detalioji specifikacija, kuri gilinasi kaip iš tikrųjų veikia sistema, sistemos algoritmas, kaip sistema asinchroniškai vykdo komunikaciją ir nusprendžia dėl galutinės blokinio būsenos. Skirtingas abstrakcijos lygmuo leis analizuoti sistemą turint skirtingą detalumo lygmenį, o susiejus specifikacijas ir sudarius tobulinimo atvaizdį, bus galima įsitikinti, kad detalioje specifikacijoje yra išlaikomi sistemos reikalavimai apibrėžti abstrakčioje specifikacijoje.

4.1. Išorinių veiksmų formali specifikacija

Remiantis „Redis Cluster“ sistemos aprašymu ir dokumentacija [Red18], buvo specifikuoti veiksmai, kuriuos galima atlikti su kiekvienu iš blokinio mazgų. Formalioje specifikacijoje specifikuojant veiksmus nėra atsižvelgta į tai, kad tarp mazgų vyksta kažkokia komunikacija – įvykus modeliuojamam veiksmui, būsena pasikeičia keliuose modeliuojamo blokinio mazguose vienu metu

(vienu būsenų mašinos žingsniu). Pasirinktu abstrakcijos lygiu yra siekiama sumažinti specifikacijos apimtį ir labiau fokusuotis į sistemos funkcionalumą bei išoriškai matomą sistemos būseną. Be to, šalia kliento išorinių veiksmų yra modeliuojami ir įvykiai, kurie gali įvykti be jokio kliento įsikišimo, tokie kaip mazgo tapimas neaktyviu ir mazgo sugrįžimas į blokinį. Pilna formali specifikacija pateikiama priede nr. 1.

4.1.1. Specifikacijos struktūra

CONSTANTS NODE, SLOTS

VARIABLES nodeFailed, nodeSlaves, slaveOf, clusterState,
clusterSlots, clusterKnownNodes

Siekiant formalią specifikaciją padaryti lanksčią, formali specifikacija yra parametrizuota abstrakčiomis konstantomis:

- NODE – aibė visų galimų mazgų identifikatorių;
- SLOTS – aibė skaičių, kurie yra naudojami kaip galimi maišos lizdų identifikatoriai.

Formalioje specifikacijoje yra naudojami kintamieji:

- nodeFailed – kiekvieno mazgo būsena, ar mazgas tapo nepasiekiamas;
- nodeSlaves – kiekvieno mazgo pavaldūs mazgai;
- slaveOf – kiekvieno mazgo pagrindinis mazgas. Šis kintamasis yra aibė, kuri turi daugiausiai vieną elementą. Tuščia aibė reiškia, kad mazgas yra pagrindinis mazgas;
- clusterState – bendra viso blokinio būsena;
- clusterSlots – informacija, kuris pagrindinis mazgas yra atsakingas už kurį maišos lizdą;
- clusterKnownNodes – aibė mazgų, kurie yra pridėti prie „Redis Cluster“ blokinio.

$$
\begin{aligned}
&\text{TCTypeOK} \triangleq \\
&\quad \land\ \text{nodeFailed} \in [\text{NODE} \rightarrow \text{BOOLEAN}] \\
&\quad \land\ \text{nodeSlaves} \in [\text{NODE} \rightarrow \text{SUBSET NODE}] \\
&\quad \land\ \text{slaveOf} \in [\text{NODE} \rightarrow \text{SUBSET NODE}] \\
&\quad \land\ \text{clusterState} \in \{\text{"OK"}, \text{"FAIL"}\} \\
&\quad \land\ \text{clusterSlots} \in [\text{SLOTS} \rightarrow \text{SUBSET NODE}] \\
&\quad \land\ \text{clusterKnownNodes} \in \text{SUBSET NODE}
\end{aligned}
$$

Kadangi TLA$^+$ formali specifikavimo kalba nėra tipizuota, tai norint įsitikinti, kad kiekvienos būsenos pasikeitimo metu būsenų mašinoje nebuvo pereita į netinkamą būseną, formalioje specifikacijoje yra apibrėžtas predikatas $\mathrm{TCTypeOK}$. Šiame predikate yra nurodomos visos galimos specifikacijos kintamųjų reikšmės.

$$
\begin{aligned}
&\mathrm{TCInit} \triangleq \\
&\quad \land\ \mathrm{nodeFailed} = [\mathrm{n} \in \mathrm{NODE} \mapsto \textsc{false}] \\
&\quad \land\ \mathrm{nodeSlaves} = [\mathrm{n} \in \mathrm{NODE} \mapsto \{\}] \\
&\quad \land\ \mathrm{slaveOf} = [\mathrm{n} \in \mathrm{NODE} \mapsto \{\}] \\
&\quad \land\ \mathrm{clusterState} = \text{“OK”} \\
&\quad \land\ \mathrm{clusterSlots} = [\mathrm{slot} \in \mathrm{SLOTS} \mapsto \{\}] \\
&\quad \land\ \mathrm{clusterKnownNodes} = \{\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathrm{TCNext} \triangleq \\
&\quad \lor\ \exists\, \mathrm{n} \in \mathrm{NODE} : \lor\ \mathrm{ClientCommand(n)} \\
&\qquad\qquad\qquad\qquad \lor\ \mathrm{NodeAvailabilityChange(n)} \\
&\quad \lor\ \mathrm{ScheduledJobs}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{TCSpec} \triangleq\ & \mathrm{TCInit} \land \Box[\mathrm{TCNext}]_{\langle \mathrm{nodeFailed, slaveOf, nodeSlaves,} \\
& \qquad \mathrm{clusterState, clusterSlots, clusterKnownNodes} \rangle}
\end{aligned}
$$

Visa išorinių veiksmų formali specifikacija yra išreikšta formule TCSpec. Pradinės būsenos predikate TCInit specifikuota, kad specifikacijos modelio pradinė būsena yra tuščia ir blokinys dar nėra sudarytas.

Kitas formalios specifikacijos formulės TCSpec elementas – veiksmas TCNext, kuris apibūdina galimus būsenų pasikeitimus: perėjimą nuo pradinės būsenos (apibrėžtos TCInit predikatu) į kitas būsenas vykdant TCNext veiksme nurodytus specifikacijos kintamųjų reikšmių pakeitimus. TCNext veiksme yra apibrėžta, kad kažkuriam iš mazgų, kurio identifikatorius yra iš konstantos NODE aibės, įvyksta arba sistemos komanda (veiksmas $\mathrm{ClientCommand}$), arba išorinis įvykis (veiksmas $\mathrm{NodeAvailabilityChange}$). Taip pat bet kuriuo metu gali įvykti suplanuotas darbas (veiksmas $\mathrm{ScheduledJobs}$).

$$
\begin{aligned}
&\mathrm{CalculateState} \triangleq \\
&\quad \textsc{let}\ \mathrm{masterWithASingleSlot} \triangleq \\
&\qquad\quad \{\mathrm{node} \in \mathrm{clusterKnownNodes} : \land\ \mathrm{slaveOf[node]} = \{\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \land\ \exists\, \mathrm{slot} \in \textsc{domain}\ \mathrm{clusterSlots} : \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathrm{clusterSlots[slot]} = \{\mathrm{node}\}\} \\
&\qquad \mathrm{reachableMasters} \triangleq \{\mathrm{node} \in \mathrm{masterWithASingleSlot} : \mathrm{nodeFailed[node]} = \textsc{false}\} \\
&\quad \textsc{in}\ \ \textsc{if}\ ((\mathrm{Cardinality(masterWithASingleSlot)} \div 2) + 1) > \mathrm{Cardinality(reachableMasters)} \\
&\qquad\ \textsc{then}\ \text{“FAIL”} \\
&\qquad\ \textsc{else}\ \text{“OK”}
\end{aligned}
$$

Suplanuoto darbo metu yra vykdomas veiksmas $\mathrm{CalculateState}$, kurio metu yra apskaičiuojama blokinio būsena ($\mathrm{clusterState}$). Jeigu taip atsitiko, kad blokinyje nėra daugumos pagrindinių veikiančių ($\mathrm{nodeFailed}[\mathrm{node}] = \mathrm{FALSE}$) mazgų, kurie yra atsakingi už bent vieną maišos lizdą ($\mathrm{clusterSlots}[\mathrm{slot}] = \{\mathrm{node}\}$), tai tada blokinio būsena yra pakeičiama į *FAIL* reikšmę. Esant tokiai būsenai blokinys neaptarnauja kliento komandų susijusių su duomenų skaitymu arba rašymu. Kitu atveju, jei yra dauguma aktyvių pagrindinių mazgų, tai blokinio būsenos kintamajam yra priskiriama *OK* reikšmė, kas reiškia, kad blokinys yra aktyvus.

4.1.2. Sistemos komandos

$$
\begin{array}{l}
\mathrm{ClientCommand}(n) \triangleq \\
\quad \lor\ \mathrm{AddNode}(n) \\
\quad \lor\ \land\ n \in \mathrm{clusterKnownNodes} \\
\qquad \land\ \lor\ \mathrm{RemoveNode}(n) \\
\qquad\quad \lor\ \mathrm{ReplicateNode}(n) \\
\qquad\quad \lor\ \mathrm{ManualFailover}(n) \\
\qquad\quad \lor\ \mathrm{AddSlot}(n) \\
\qquad\quad \lor\ \mathrm{DelSlot}(n) \\
\qquad\quad \lor\ \mathrm{ReshardSlot}(n) \\
\qquad\quad \lor\ \mathrm{FlushSlots}(n)
\end{array}
$$

„Redis Cluster“ klientų siunčiamos įvykdyti komandos yra išoriniai veiksmai darantys įtaką mazgų tolimesniems veiksmams ir jų būsenų pasikeitimams. Dalis šių komandų gali įvykti tik tada, kai einamasis mazgas n priklauso blokiniui ($n \in \mathrm{clusterKnownNodes}$). Specifikuojant išorinius veiksmus buvo išskirtos dviejų tipų komandos: 1) komandos susijusios su blokinio mazgais; ir 2) komandos susijusios su maišos lizdais.

Mazgų komandos

„Redis Cluster“ blokiniui veikiant, iš išorės klientai gali atlikti komandas susijusias su mazgų rolių ir mazgų ryšių pasikeitimais.

$$
\begin{array}{l}
\mathrm{AddNode}(n) \triangleq \\
\quad \exists\, \mathrm{newNode} \in \mathrm{NODE} : \\
\qquad \land\ \mathrm{newNode} \notin \mathrm{clusterKnownNodes} \\
\qquad \land\ \mathrm{clusterKnownNodes}' = \mathrm{clusterKnownNodes} \cup \{\mathrm{newNode}\} \\
\qquad \land\ \textsc{unchanged}\ \langle \mathrm{nodeFailed}, \mathrm{slaveOf}, \mathrm{nodeSlaves}, \mathrm{clusterState}, \mathrm{clusterSlots} \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathrm{RemoveNode}(\mathrm{nodeToRemove}) \triangleq \\
\quad \land\ \mathrm{clusterKnownNodes}' = \mathrm{clusterKnownNodes} \setminus \{\mathrm{nodeToRemove}\}
\end{array}
$$

$$
\begin{aligned}
&\land \text{clusterSlots}' = [\text{slot} \in \textsc{domain}\ \text{clusterSlots} \mapsto \textsc{if}\ \text{clusterSlots}[\text{slot}] = \{\text{nodeToRemove}\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{then}\ \{\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{else}\ \text{clusterSlots}[\text{slot}]] \\
&\land \text{nodeFailed}' = [\text{nodeFailed}\ \textsc{except}\ ![\text{nodeToRemove}] = \textsc{false}] \\
&\land \text{slaveOf}' = [\text{node} \in \textsc{domain}\ \text{slaveOf} \mapsto \textsc{if}\ \lor \text{slaveOf}[\text{node}] = \{\text{nodeToRemove}\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \lor \text{node} = \text{nodeToRemove} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{then}\ \{\} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textsc{else}\ \text{slaveOf}[\text{node}]] \\
&\land \text{nodeSlaves}' = [\text{node} \in \textsc{domain}\ \text{nodeSlaves} \mapsto \\
&\qquad\qquad\qquad\qquad\qquad \textsc{if}\ \text{node} = \text{nodeToRemove} \\
&\qquad\qquad\qquad\qquad\qquad \textsc{then}\ \{\} \\
&\qquad\qquad\qquad\qquad\qquad \textsc{else}\ \text{nodeSlaves}[\text{node}] \setminus \{\text{nodeToRemove}\}] \\
&\land \textsc{unchanged}\ \langle \text{clusterState} \rangle
\end{aligned}
$$

Prie blokinio galima pridėti naują (veiksmas AddNode) arba pašalinti esamą (veiksmas RemoveNode) mazgą. Pridėjus naują mazgą prie blokinio, jis turi pagrindinio mazgo rolę, tačiau dar neturi maišos lizdų atsakomybių. Pašalinus mazgą iš blokinį sudarančių mazgų aibės (clusterKnownNodes), maišos lizdai, už kuriuos buvo atsakingas pašalintas mazgas, tampa niekam nepriskirti, taip pat yra ištrinama informacija apie mazgo būseną bei yra ištrinama informacija apie pagrindinius arba pavaldžius mazgus, jeigu atitinkamai pašalintas mazgas yra kažkurio mazgo pavaldus arba pagrindinis mazgas.

$$
\begin{aligned}
&\text{ReplicateNode}(n) \triangleq \\
&\quad \exists\, \text{masterToReplicate} \in \text{clusterKnownNodes} : \\
&\qquad \land n \neq \text{masterToReplicate} \\
&\qquad \land \text{slaveOf}[\text{masterToReplicate}] = \{\} \\
&\qquad \land \textsc{if}\ \text{slaveOf}[n] = \{\} \\
&\qquad\quad \textsc{then}\ \land \text{Cardinality}(\{\text{slot} \in \textsc{domain}\ \text{clusterSlots} : \text{clusterSlots}[\text{slot}] = \{n\}\}) = 0 \\
&\qquad\qquad\quad\ \land \text{nodeSlaves}' = [\text{nodeSlaves}\ \textsc{except}\ ![\text{masterToReplicate}] = @ \cup \{n\}] \\
&\qquad\quad \textsc{else}\ \exists\, \text{previousMaster} \in \text{slaveOf}[n] : \\
&\qquad\qquad\qquad \text{nodeSlaves}' = [\text{nodeSlaves}\ \textsc{except}\ ![\text{masterToReplicate}] = @ \cup \{n\}, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ![\text{previousMaster}] = @ \setminus \{n\}] \\
&\qquad \land \text{slaveOf}' = [\text{slaveOf}\ \textsc{except}\ ![n] = \{\text{masterToReplicate}\}] \\
&\qquad \land \textsc{unchanged}\ \langle \text{nodeFailed}, \text{clusterState}, \text{clusterSlots}, \text{clusterKnownNodes} \rangle
\end{aligned}
$$

Jeigu „Redis Cluster“ blokinyje egzistuoja pagrindinis mazgas ($\text{slaveOf}[\text{masterToReplicate}] = \{\}$), tai kitas mazgas gali tapti pagrindinio mazgo pavaldžiu mazgu (veiksmas ReplicateNode). Jeigu einamasis šio veiksmo mazgas n taip pat yra pagrindinis mazgas, tai jis gali tapti kito mazgo pavaldžiu mazgu tik tada, kai nėra atsakingas

už nei vieną maišos lizdą. Toks sistemos veikimas yra taikomas tada, kai yra kuriamas naujas blokinys, kadangi prie blokinio pridėjus naują mazgą, iš pradžių jis turi pagrindinio mazgo rolę. Kitu atveju, jeigu einamasis veiksmo ReplicateNode mazgas n yra pavaldus mazgas, tai mazgui nėra keliami jokie reikalavimai. Pavaldžiam mazgui tapus kito mazgo pavaldžiu mazgu, yra pašalinami ryšiai su prieš tai buvusiu pagrindiniu mazgu, ir mazgas tampa atsakingas už tuos pačius maišos lizdus, kaip ir naujai priskirtas pagrindinis mazgas.

$$\begin{array}{l}
\text{ManualFailover}(n) \triangleq \\
\quad \land\, \text{slaveOf}[n] \neq \{\} \\
\quad \land\, \exists\, \text{master} \in \text{slaveOf}[n] : \\
\qquad \land\, \text{slaveOf}' = [\text{slaveOf EXCEPT } ![n] = \{\}, \\
\qquad\qquad\qquad\qquad\qquad\quad ![\text{master}] = \{n\}] \\
\qquad \land\, \text{nodeSlaves}' = [\text{nodeSlaves EXCEPT } ![n] = (\text{nodeSlaves}[\text{master}] \setminus \{n\}) \cup \{\text{master}\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad ![\text{master}] = \{\}] \\
\qquad \land\, \text{clusterSlots}' = [\text{slot} \in \text{DOMAIN clusterSlots} \mapsto \text{IF clusterSlots}[\text{slot}] = \{\text{master}\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{THEN } \{n\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{ELSE clusterSlots}[\text{slot}]] \\
\qquad \land\, \text{UNCHANGED } \langle \text{nodeFailed}, \text{clusterState}, \text{clusterKnownNodes} \rangle
\end{array}$$

Blokinyje pagrindinį mazgą galima pakeisti vienu iš jo pavaldžiu mazgu ir nesant blokinio gedimui (veiksmas ManualFailover). Rankiniu būdu sukeitus mazgų roles, vienas iš pavaldžių mazgų tampa pagrindiniu mazgu, atsakingu už prieš tai buvusio pagrindinio mazgo maišos lizdus, o seniau buvęs pagrindinis mazgas tampa naujo pagrindinio mazgo pavaldžiu mazgu.

Duomenų komandos

$$\begin{array}{l}
\text{AddSlot}(n) \triangleq \\
\quad \exists\, \text{slotToChange} \in \text{SLOTS} : \\
\qquad \land\, \text{clusterSlots}[\text{slotToChange}] = \{\} \\
\qquad \land\, \text{clusterSlots}' = [\text{clusterSlots EXCEPT } ![\text{slotToChange}] = \{n\}] \\
\qquad \land\, \text{UNCHANGED } \langle \text{nodeFailed}, \text{slaveOf}, \text{nodeSlaves}, \text{clusterState}, \text{clusterKnownNodes} \rangle
\end{array}$$

$$\begin{array}{l}
\text{DelSlot}(n) \triangleq \\
\quad \exists\, \text{slotToChange} \in \text{SLOTS} : \\
\qquad \land\, \text{clusterSlots}[\text{slotToChange}] \neq \{\} \\
\qquad \land\, \text{clusterSlots}' = [\text{clusterSlots EXCEPT } ![\text{slotToChange}] = \{\}] \\
\qquad \land\, \text{UNCHANGED } \langle \text{nodeFailed}, \text{slaveOf}, \text{nodeSlaves}, \text{clusterState}, \text{clusterKnownNodes} \rangle
\end{array}$$

$$\begin{array}{l}
\text{FlushSlots}(n) \triangleq \\
\quad \land\, \text{clusterSlots}' = [\text{slot} \in \text{DOMAIN clusterSlots} \mapsto \text{IF clusterSlots}[\text{slot}] = \{n\} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{THEN } \{\}
\end{array}$$

$\text{ELSE } \text{clusterSlots}[\text{slot}]]$

$\land \text{ UNCHANGED } \langle \text{nodeFailed}, \text{slaveOf}, \text{nodeSlaves}, \text{clusterState}, \text{clusterKnownNodes} \rangle$

Šalia komandų susijusių su mazgais, veikiant blokiniui, klientai gali atlikti operacijas susijusias su maišos lizdų atsakomybių pakeitimais. Maišos lizdas gali būti priskirtas kažkuriam blokinį sudarančiam mazgui (veiksmas AddSlot), jeigu už maišos lizdą dar nėra atsakingas kitas blokinio mazgas ($\text{clusterSlots}[\text{slotToChange}] = \{\}$). Tuo tarpu priskirtas maišos lizdų atsakomybes galima pašalinti dvejais būdais: arba pašalinti po vieną maišos lizdo atsakomybę (veiksmas DelSlot), arba pašalinti visas atsakomybes susijusias su einamuoju mazgu n (veiksmas FlushSlots).

$\text{ReshardSlot}(\text{n}) \triangleq$

$\land \text{ slaveOf}[\text{n}] = \{\}$

$\land \exists \text{ slotToMigrate} \in \text{SLOTS} :$

$\land \text{ clusterSlots}[\text{slotToMigrate}] = \{\text{n}\}$

$\land \exists \text{ targetNode} \in \text{clusterKnownNodes} :$

$\land \text{ targetNode} \neq \text{n}$

$\land \text{ slaveOf}[\text{targetNode}] = \{\}$

$\land \text{ clusterSlots}' = [\text{clusterSlots EXCEPT } ![\text{slotToMigrate}] = \{\text{targetNode}\}]$

$\land \text{ UNCHANGED } \langle \text{nodeFailed}, \text{slaveOf}, \text{nodeSlaves}, \text{clusterState}, \text{clusterKnownNodes} \rangle$

Maišos lizdų atsakomybių perkėlimas (perskirstymas) iš vieno pagrindinio mazgo kitam pagrindiniam mazgui (veiksmas ReshardSlot) yra vykdomas atnaujinant clusterSlots kintamąjį, pagal įvykusius maišos lizdų atsakomybių pasikeitimus. Vykdant maišos lizdų perskirstymą, yra įsitikinama, kad mazgai, kurie dalyvauja maišos lizdų pertvarkyme, yra pagrindiniai mazgai.

4.1.3. Išoriniai įvykiai

Veikiant sistemai šalia klientų siunčiamų komandų gali įvykti nenumatyti išoriniai įvykiai (angl. *events*), darantys įtaką tolimesniems „Redis Cluster“ sistemos sprendimams. Formalioje specifikacijoje modeliuojant „Redis Cluster“ sistemos veikimą yra išskirti du galimi įvykti išoriniai įvykiai: 1) blokinio mazgas tampa neaktyvus (išsijungia, tampa nepasiekiamas tinkle ir pan.); ir 2) blokinio mazgas, buvęs neaktyvus, vėl prisijungia prie blokinio. Realios sistemos veikimo metu gali įvykti ir daugiau išorinių įvykių, tačiau šie du įvykiai turi didžiausią poveikį tolimesnei sistemos veiklai.

$\text{NodeFails}(\text{n}) \triangleq$

$\land \text{ nodeFailed}[\text{n}] = \text{FALSE}$

$\land \text{ nodeFailed}' = [\text{nodeFailed EXCEPT } ![\text{n}] = \text{TRUE}]$

$\land \text{ IF } \land \text{ slaveOf}[\text{n}] = \{\}$

$\land \text{nodeSlaves}[n] \neq \{\}$
$\textsc{then} \ \land \text{ElectMaster}(n)$
$\quad \land \textsc{unchanged} \ \langle \text{clusterState}, \text{clusterKnownNodes} \rangle$
$\textsc{else} \ \textsc{unchanged} \ \langle \text{slaveOf}, \text{nodeSlaves}, \text{clusterSlots}, \text{clusterKnownNodes}, \text{clusterState} \rangle$

Mazgui tapus neaktyviam (veiksmas NodeFails) yra modeliuojama, kad pasikeičia einamojo mazgo n kintamojo nodeFailed reikšmė į TRUE. Jei neaktyviu tapęs mazgas yra pagrindinis mazgas ir jis turi pavaldžių mazgų, tuo pačiu žingsniu yra išrenkamas naujas pagrindinis mazgas. Kitu atveju, jei neaktyviu tapo pavaldus mazgas, nėra vykdomi jokie papildomi veiksmai.

Realioje „Redis Cluster“ sistemoje naujo pagrindinio mazgo rinkimas vyksta vykdant balsavimą tarp blokinio mazgų. Siekiant sumažinti galimų būsenų tikrinimą, modeliuojamoje sistemoje naujo pagrindinio mazgo rinkimas vyksta pasirenkant vieną mazgą iš neaktyviu tapusio pagrindinio mazgo pavaldžių mazgų aibės.

$\text{NodeRejoins}(n) \triangleq$
$\quad \land \text{nodeFailed}[n] = \textsc{true}$
$\quad \land \text{nodeFailed}' = [\text{nodeFailed} \ \textsc{except} \ ![n] = \textsc{false}]$
$\quad \land \textsc{unchanged} \ \langle \text{slaveOf}, \text{nodeSlaves}, \text{clusterSlots}, \text{clusterKnownNodes}, \text{clusterState} \rangle$

Mazgui kurį laiką pabuvus neaktyviam, mazgas vėl gali tapti prieinamas ir pasiekiamas kitiems mazgams bei gali vėl bandyti būti blokinio dalimi (veiksmas NodeRejoins). Jeigu mazgui tapus neaktyviam buvo išrinktas naujas pagrindinis mazgas, tai sugrįžęs mazgas tampa pavaldžiu mazgu naujai išrinkto pagrindinio mazgo.

4.1.4. Bendras modelio tikrinimas

Norint įsitikinti, kad specifikacijoje nebuvo palikta klaidų, susijusių su galimų būsenų aibę neatitinkančiomis specifikacijos kintamųjų reikšmėmis, ir norint įsitikinti, kad specifikacijoje nėra aklavięčių, buvo atliktas specifikacijos modelio tikrinimas. Tikrinimo metu kiekvieno būsenos pasikeitimo metu buvo tikrinamas tipų korektiškumo invarianto TCTypeOK tenkinimas.

Modelio tikrinimui buvo pasirinktos šios konstantų reikšmės:

$$\text{NODE} = \{101, 202, 303\}, \text{SLOTS} = \{1, 2, 3, 4, 5, 6\}$$

Sudarytos specifikacijos ir apibrėžto modelio tikrinimas buvo atliekamas nešiojamuoju kompiuteriu (4 branduoliai, 16 GB RAM ir SSD diskas) bei TLA^+ Toolbox įrankiu (versija 1.7.1). Atlikus modelio tikrinimą nebuvo rasta invarianto TCTypeOK pažeidimų ar modeliuojamos sistemos aklaviečių.

4.2. Žinučių apdorojimo formali specifikacija

Pirmoji išorinių veiksmų formali specifikacija buvo sudaryta remiantis sistemos dokumentacija bei sistemos aprašymu. Išorinių veiksmų formalioje specifikacijoje modeliuojant sistemą visi veiksmai (būsenų pasikeitimai) yra atliekami vienu būsenų mašinos perėjimo žingsniu, nevykdant jokios komunikacijos tarp „Redis Cluster" blokinį sudarančių mazgų. Tai neatitinka realaus sistemos veikimo, todėl TLA^+ kalba buvo sudaryta antroji formali specifikacija. Formalioje specifikacijoje buvo modeliuojama žinučių komunikacija tarp blokinį sudarančių mazgų, t. y. vieno būsenų mašinos žingsnio metu būsena pasikeičia tik viename mazge ir informacija apie pasikeitimus pasiekia kitus mazgus, siunčiant žinutes paskalų protokolo pagalba.

Sudarant žinučių apdorojimo formalią specifikaciją, buvo remtasi atvirai prieinamu „Redis Cluster" sistemos versijos 5.0 programiniu kodu [Red19]. Galutiniame rezultate buvo gauta daugiau nei dviejų tūkstančių eilučių formali specifikacija artima programinio kodo abstrakcijos lygiui. Pilna specifikacija pateikiama priede nr. 2.

4.2.1. Specifikacijos struktūra

CONSTANTS NODE, SLOTS_COUNT,
MAX_BUFFER_LEN, CLUSTER_REQUIRE_FULL_COVERAGE
VARIABLES clusterState, eventLoop, recieveBuffer, sendBuffer

Žinučių apdorojimo formali specifikacija yra parametrizuota abstrakčiomis konstantomis:

- NODE – aibė visų galimų mazgų identifikatorių. Ši aibė, dėl modeliuojamų žinučių apdorojimo veiksmų, privalo būti sudaryta tik iš skaičių;
- SLOTS_COUNT – skaičius, kiek iš viso blokinyje yra maišos lizdų;
- MAX_BUFFER_LEN – skaičius, nusakantis maksimalų žinučių skaičių žinučių gavimo ir siuntimo buferiuose;
- CLUSTER_REQUIRE_FULL_COVERAGE – sistemos konfigūracijos parametras, nusakantis, ar visi blokinyje esantys maišos lizdai privalo būti susieti su pagrindiniais mazgais.

Taip pat formalioje specifikacijoje yra naudojami specifikacijos būsenos kintamieji:

- clusterState – kiekvieno mazgo numanoma bendra blokinio būsena:
  - Blokinio būsena: *OK* – blokinys yra aktyvus, arba *FAIL* – blokinys yra neatktyvus ir neapdoroja išorės komandų;

  - Informacija, kuris mazgas yra atsakingas už kurį maišos lizdą;
  - Šiam mazgui žinomų kitų mazgų informacija, tokia kaip: mazgo *configEpoch* reikšmė, maišos lizdų atsakomybės, mazgo dabartinės būsenos žymos (angl. *flags*), mazgo pavaldūs mazgai ir mazgo pagrindinis mazgas.
- $\mathrm{eventLoop}$ – kiekvieno mazgo įvykių ciklo (angl. *event loop*) būsena;
- $\mathrm{recieveBuffer}$ – kiekvieno mazgo gavimo buferis kiekvienam kitam blokinio mazgui;
- $\mathrm{sendBuffer}$ – kiekvieno mazgo siuntimo buferis kiekvienam kitam blokinio mazgui.

$$
\begin{aligned}
&\mathrm{TCNext} \triangleq \\
&\quad \exists\, n \in \mathrm{NODE} : \\
&\qquad \lor\ \mathrm{EventLoopMain}(n) \\
&\qquad \lor\ \mathrm{TimeActions}(n) \\
&\qquad \lor\ \mathrm{ChangeAddress}(n) \\
&\qquad \lor\ \mathrm{ClientAdminCommand}(n)
\end{aligned}
$$

Formalioje specifikacijoje TCNext veiksme kažkuriam iš „Redis Cluster“ mazgų (einamajam mazgui n) gali įvykti vienas iš veiksmų:

- Veiksmas susijęs su įvykiu ciklu (angl. *event loop*);
- Išoriniai įvykiai, lemiantys skirtingą žinučių apdorojimą ir sistemos veikimą:
  - Veiksmas susijęs su laiko pasikeitimais;
  - Veiksmas susijęs su mazgo adreso pasikeitimu;
  - Veiksmas susijęs su blokinio administravimo komandų apdorojimu.

4.2.2. Mazgo būsenos žymos

Dažniausiai žinučių apdorojimo rezultatas: einamojo mazgo blokinio būsenos kintamajame $\mathrm{clusterState}$ saugomos informacijos apie žinomų mazgų būsenos žymas (angl. *flags*) atnaujinimas. Mazgo būsenos žymos yra modeliuojamos kaip aibė, kurioje gali būti simbolių eilutės elementai, reprezentuojantys mazgo būseną:

- MYSELF – žymė, kuri niekada nesikeičia ir kuri yra priskirta tik tam mazgui, kuris yra einamasis (dabartinis) mazgas;
- MASTER – mazgas yra pagrindinis mazgas;

- SLAVE – mazgas yra pavaldus mazgas;
- PFAIL – yra manoma, kad mazgas (gali būti tiek pagrindinis, tiek pavaldus) yra nepasiekiamas, tačiau reikia, kad tai patvirtintų kiti pagrindiniai mazgai;
- FAIL – mazgas yra neaktyvus (nepasiekiamas) ir dauguma blokinio pagrindinių mazgų tai patvirtino;
- HANDSHAKE – su mazgu dar nėra užmegzta pilna komunikacija: reikia apsikeisti pirma *ping* žinute;
- MEET – mazgas nori prisijungti prie blokinio.

4.2.3. Įvykių ciklas

„Redis Cluster" sistemos mazguose visi veiksmai yra vykdomi vienoje gijoje, todėl visi veiksmai yra apdorojami įvykių cikle. Įvykių cikle gali įvykti šių tipų įvykiai:

- Laiko įvykis – įvykio įvykdymas priklauso nuo laiko (kas kažkokį laiko tarpą reikia įvykdyti tam tikrą veiksmą);
- Komunikacijos kanalo skaitymo arba rašymo įvykis – skaitymo ar rašymo įvykis, susijęs su atidarytu tinklo lizdu (angl. *socket*);
- Įvykiai vykdomi prieš vykdant kanalų skaitymo arba rašymo įvykius;
- Įvykiai vykdomi po kanalų skaitymo arba rašymo įvykių įvykdymo.

Įvykių cikle kiekvienas iš įvykių tipų yra vykdomas tam tikra tvarka. Įvykių ciklo vykdymas yra pradedamas įvykdant užregistruotus įvykius prieš kanalų skaitymą arba rašymą. Po šių įvykių seka laukimas kanalų skaitymo arba rašymo įvykių iš atidarytų tinklo lizdų. Jei per tam tikrą laukimo laiką (angl. *timeout*) neįvyksta joks skaitymo ar rašymo įvykis iš klientų ar kitų mazgų, tada yra vykdomi užregistruoti įvykiai po skaitymo arba rašymo. Jei vis dėlto per laukimo laiką kažkuris iš klientų ar mazgų nusprendžia siųsti arba priimti duomenis tinklo lizdu, tada yra surenkamas sąrašas visų kanalų, kurie nori vykdyti skaitymo arba rašymo įvykius. Surinkus kanalų skaitymo arba rašymo įvykių sąrašą, taip pat yra atliekami įvykiai po skaitymo arba rašymo. Atlikus šiuos įvykius, yra vykdomas kanalų skaitymo arba rašymo įvykių apdorojimas. Apdorojus šiuos įvykius, priklausomai nuo to, ar praėjo laiko įvykiui įvykti reikalingas laiko tarpas ar ne, yra įvykdomi laiko įvykiai. Įvykdžius arba neįvykdžius laiko įvykius, įvykių ciklas yra pradedamas vykdyti iš naujo.

Apdorojant surinktus kanalų skaitymo arba rašymo įvykius, priklausomai nuo kanalo skaitymo arba rašymo įvykio registracijos, vieno kanalo skaitymo arba rašymo įvykius galima apdoroti

skirtinga tvarka. Tai yra, jei norima galima iš pradžių apdoroti kanalo skaitymo, o vėliau kanalo rašymo įvykius arba atvirkščiai.

$$
\begin{aligned}
&\text{EventLoopMain(n)} \triangleq \\
&\quad \lor \text{BeforeSleep(n)} \\
&\quad \lor \text{ProcessFileEvents(n)} \\
&\quad \lor \text{ProcessTimeEvents(n)}
\end{aligned}
$$

Specifikuojant tarp mazgų einančių žinučių komunikaciją, yra modeliuojamas įvykių ciklas, kuriame yra apdorojami laiko (ProcessTimeEvents), mazgų kanalų skaitymo arba rašymo įvykiai (ProcessFileEvents) bei įvykiai prieš skaitymą arba rašymą (BeforeSleep) tokia pat tvarka kaip ir realioje „Redis Cluster“ sistemoje. Įvykių vykdomų surinkus kanalų skaitymo arba rašymo įvykius apdorojimas nėra specifikuotas, kadangi vykdant mazgų žinučių apdorojimą nėra užregistruojami tokio tipo įvykiai.

Įvykiai prieš skaitymą arba rašymą

Veiksme BeforeSleep yra apdorojamas tik vienas įvykis prieš skaitymą arba rašymą – ClusterBeforeSleep. Šio įvykio metu (priklausomai nuo to ar reikia) yra atliekami du veiksmai: 1) dalyvavimas pagrindinio mazgo rinkimuose; ir 2) einamojo mazgo blokinio būsenos atnaujinimas.

Pavaldus mazgas pradeda dalyvauti naujo pagrindinio mazgo rinkimuose, kai pavaldaus mazgo pagrindinis mazgas, esantis atsakingas už bent vieną maišos lizdą, tampa neaktyvus. Siekiant sumažinti specifikacijos sudėtingumą, specifikuojant naujo pagrindinio mazgo rinkimą, rinkimų algoritmas nėra pilnai specifikuotas. Realioje „Redis Cluster“ sistemoje naujo pagrindinio mazgo rinkimai priklauso nuo mazge saugomų duomenų senumo, mazgo rango, bandymų skaičiaus tapti pagrindiniu mazgu, laiko ir kitų mazgų balsų. Specifikacijoje veiksme BeforeSleep rinkimai vyksta taip: pirmasis pavaldus mazgas, pastebėjęs, kad jo pagrindinis mazgas nėra aktyvus, pasiskelbia save nauju pagrindiniu mazgu. Prieš pasiskelbiant nauju pagrindiniu mazgu, yra atnaujinama einamojo mazgo clusterState kintamojo reikšmė: pakeičiama, kad einamasis mazgas yra pagrindinis mazgas, atsakingas už neaktyvaus mazgo maišos lizdus, bei vienetu yra padidinamos mazgo *configEpoch* ir blokinio *currentEpoch* reikšmės. Atlikus šiuos einamojo mazgo būsenos pakeitimus, naujai išrinktas pagrindinis mazgas išsiunčia *pong* tipo žinutes visiems blokinio mazgams, kad jie galėtų atnaujinti būseną, kad pasikeitė einamojo mazgo rolė.

BeforeSleep veiksme atliekant blokinio būsenos atnaujinimą, vykdomas algoritmas priklauso nuo sistemos parametro: ar yra reikalaujama, kad visi blokinio maišos lizdai būtų susieti su pagrindiniais mazgais. Jeigu parametras yra nustatytas ir yra nustatoma, kad bent vienas iš maišos lizdų

nėra susietas su pagrindiniu mazgu, blokinys tampa neaktyvus ($\text{clusterState}[\text{n}].\text{clusterState} = \text{"FAIL"}$). Jeigu nėra reikalaujamas pilnas maišos lizdų padengimas, tada yra skaičiuojama ar blokinyje yra dauguma aktyvių pagrindinių mazgų, atsakingų už bent vieną maišos lizdą. Nustačius, kad nėra daugumos, blokinys taip pat tampa neaktyvus.

Veiksme BeforeSleep taip pat yra atliekamas einamojo mazgo įvykių ciklo būsenos (kintamasis eventLoop) atnaujinimas: yra sudaromas kanalų, kurie nori įvykdyti skaitymo ar rašymo įvykius, sąrašas. Kanalas tarp einamojo mazgo ir kito mazgo papuola į sąrašą priklausomai nuo to, ar einamojo mazgo siuntimo buferis kitam mazgui bei kito mazgo siutimo buferis einamajam mazgui yra tušti. Sudarius sąrašą yra pakeičiama einamojo mazgo ciklo būsena, kad toliau reikia apdoroti kanalų skaitymo bei rašymo įvykius iš sudaryto įvykių sąrašo.

Kanalų skaitymo arba rašymo įvykiai

Kadangi „Redis Cluster“ sistemos žinučių apdorojimo algoritme visuose kanalų skaitymo arba rašymo įvykiuose yra reikalaujama, kad iš pradžių būtų apdorojamas kanalo rašymo, o vėliau skaitymo įvykis, tai, skirtingai nei realioje sistemoje, nėra modeliuojamas atvirkštinis variantas.

$$
\begin{aligned}
&\text{ProcessFileEvents(n)} \triangleq \\
&\quad \lor \text{ChooseEventToProcess(n)} \\
&\quad \lor \text{ProcessCurrentEvent(n)}
\end{aligned}
$$

Apdorojant kanalų rašymo arba skaitymo įvykius yra modeliuojama, kad iš pradžių yra pasirenkamas (veiksmas ChooseEventToProcess) vienas tarp einamojo mazgo ir kito mazgo esančių kanalų, kuris nori vykdyti duomenų siuntimą arba gavimą. Pasirinktas kanalo skaitymo arba rašymo įvykis tampa einamuoju įvykiu. Jeigu kanalų skaitymo arba rašymo įvykių sąrašas yra tuščias arba visi įvykiai esantys sąraše buvo apdoroti, tada yra pakeičiama einamojo mazgo ciklo būsena, kad toliau reikia apdoroti laiko įvykius (jeigu įvyko laiko įvykis) arba įvykius prieš skaitymą arba rašymą.

Veiksme ProcessCurrentEvent yra vykdomas žinučių siuntimas ir gavimas tinklo lizdo kanalu, kurio būseną reprezentuoja recieveBuffer bei sendBuffer specifikacijos kintamieji. Šių kintamųjų tipas – seka, kurioje saugomi žinučių įrašai išlaikant jų tvarką. Veiksme ProcessCurrentEvent iš pradžių yra vykdomas žinučių siuntimas kanalu. Formalioje specifikacijoje siuntimas (veiksmas ClusterWriteHandler) yra vykdomas vienu būsenų mašinos žingsniu visas žinutes iš einamojo mazgo siuntimo buferio sekos (sendBuffer[currentNode][linkNode]) perkeliant į kito mazgo gavimo buferio (recieveBuffer[linkNode][currentNode]) sekos pabaigą.

Išsiuntus žinutes, kitu būsenos mašinos žingsniu yra vykdomas žinučių gautų tuo pačiu tinklo

lizdo kanalu apdorojimas. Norint sumažinti galimų būsenų skaičių, modeliuojamas kanalu gautų žinučių apdorojimas yra vykdomas per vieną būsenų mašinos žingsnį. Veiksmo metu rekursyviai imama po vieną žinutę iš gavimo buferio ($\mathrm{recieveBuffer[currentNode][linkNode]}$) pradžios tol, kol yra pasiekiama buferio sekos pabaiga. Baigus apdoroti vienu kanalu gautas žinutes, yra vykdomas sekančio kanalo skaitymo arba rašymo įvykio apdorojimas (veiksmas $\mathrm{ChooseEventToProcess}$).

Apdorojant gautą žinutę yra vykdomas veiksmas $\mathrm{ClusterProcessMsg}$, kuris detaliau yra aptariamas tolimesniame poskyriuje. Įvykdžius žinutės apdorojimą, pagal žinutėje gautą informaciją yra atnaujinama einamojo mazgo blokinio būsena ($\mathrm{clusterState}$) bei taip pat naujos papildomos žinutės, sukurtos pirminės žinutės apdorojimo metu, yra įdedamos į siuntimo buferių sekas ($\mathrm{sendBuffer}$). Naujai pridėtos žinutės bus išsiųstos to pačio arba kito įvykių ciklo metu, priklausomai nuo to, ar jau ankščiau einamajame įvykių cikle buvo apdoroti žinučių gavėjų kanalų skaitymo arba rašymo įvykiai.

Laiko įvykiai

Po kanalų skaitymo arba rašymo įvykių apdorojimo seka laiko įvykių vykdymas. Apdorojant žinutes periodiškai (realioje sistemoje, kas šimtą milisekundžių) yra įvykdomas tik vienas laiko įvykis – blokinio suplanuotas darbas (veiksmas $\mathrm{ClusterCronJob}$). Įvykdžius arba, jeigu laikas įvykiui dar neatėjo, neįvykdžius laiko įvykio, yra pakeičiama einamojo mazgo ciklo būsena, kad baigėsi vienas ciklas ir reikia vėl iš naujo vykdyti įvykius prieš skaitymą arba rašymą.

Vykdomas blokinio suplanuotas darbas užtikrina mazgų būsenų pasikeitimus ir nenutrūkstamą komunikaciją tarp mazgų. Šio įvykio metu realioje sistemoje vykdomi veiksmai pateikti 9 pav. esančiame priede nr. 3.

Formalioje specifikacijoje veiksme $\mathrm{ClusterCronJob}$ vykdoma:

- Nustatoma, ar žinomi mazgai, kurie ankščiau buvo bandę užmegzti komunikaciją su einamuoju mazgu ($\mathrm{"HANDSHAKE"} \in \mathrm{node.flags}$), per numatytą laukimo laiką dar kartą įvykdė komunikaciją su einamuoju mazgu. Jeigu šis laukimo laikas praėjo, tai tada toks mazgas yra pašalinamas iš einamojo mazgo žinomų mazgų sąrašo;
- Jeigu einamajam mazgui yra žinomas mazgas, su kuriuo dar nėra sukurtas tinklo lizdo komunikavimo kanalas, tada yra sukuriamas kanalas ir yra išsiunčiama žinutė. Jeigu su mazgu dar nesame pilnai užmezgę komunikavimo ($\mathrm{"MEET"} \in \mathrm{node.flags}$), tai tada yra išsiunčiama *meet* tipo žinutė, kitu atveju yra išsiunčiama *ping* tipo žinutė;
- Kas dešimtą blokinio suplanuoto darbo vykdymą (realioje sistemoje kas sekundę) yra pasirenkamas vienas atsitiktinis einamajam mazgui žinomas mazgas, su kuriuo yra pilnai už-

megztas komunikavimas. Pasirinktam mazgui yra išsiunčiama *ping* tipo žinutė;

- Žinomiems mazgams, iš kurių nebuvo gautas *pong* atsakas į *ping* žinutę daugiau nei per pusę laukimo laiko, yra uždaromas komunikavimo kanalas. Uždarytas komunikavimo kanalas bus vėl atidarytas, kai mazgas užmegs komunikaciją. Taip yra bandoma išspręsti situaciją, kai nors ir mazgas yra aktyvus, tačiau tarp mazgų įvyko komunikacijos kanalo problemos;
- Siekiant užtikrinti, kad *ping* tipo žinutės yra išsiunčiamas reguliariai be didelio uždelsimo (angl. *delay*), mazgams, kuriems dabar nėra išsiųstos *ping* žinutės ir seniau gauta *pong* žinutė yra sena, yra išsiunčiama po *ping* tipo žinutę;
- Mazgams, iš kurių *pong* žinutės nebuvo gauta daugiau nei per pilną laukimo laiką, yra uždedama *PFAIL* būsenos žyma (angl. *flag*);
- Jeigu einamasis mazgas yra pavaldus mazgas ir po įvykusių mazgų būsenų pasikeitimų jo pagrindinis mazgas tapo neaktyvus ($\text{"FAIL"} \in \text{node.flags}$), tada yra atliekamas einamojo pavaldaus mazgo persijungimas (angl. *failover*);
- Kadangi vyko einamajam mazgui žinomų mazgų būsenų pasikeitimai, yra iš naujo apskaičiuojama blokinio būsena einamojo mazgo atžvilgiu. Būsenos apskaičiavimui yra naudojamas tas pats algoritmas kaip ir apdorojant $\text{ClusterBeforeSleep}$ įvykį prieš skaitymą arba rašymą.

Įvykdžius blokinio suplanuotą darbą, yra atnaujinamas einamojo mazgo blokinio būsenos kintamasis clusterState ir naujai sukurtos žinutės yra įdedamos į einamojo mazgo siuntimo buferių sekas. Šios žinutės bus išsiųstos tik kito įvykių ciklo metu.

4.2.4. Žinučių apdorojimas

Formalioje specifikacijoje buvo pasirinkta apdoroti tik tas žinutes, kurios yra susijusios su mazgų būsenų apsikeitimu ir atnaujinimu: *ping*, *pong*, *meet*, *fail*, *update* ir *mfstart* tipo žinutės. Kiekvienoje bet kokio tipo žinutėje yra siunčiama informacija: žinutės tipas, blokinio *currentEpoch* reikšmė, siuntėjo *configEpoch* reikšmė, siuntėjo identifikatorius, siuntėjo adreso pasikeitimą nusakanti informacija, siuntėjo maišos lizdų atsakomybės, siuntėjo pagrindinis mazgas (jei siuntėjas yra pavaldus mazgas), siuntėjo būsenos žymos (angl. *flags*) ir kokia siuntėjo manymu yra viso blokinio būsena. Taip pat šalia bendrosios žinutės informacijos yra pridedama kiekvienam žinutės tipui reikalinga specifinė informacija. Formalioje specifikacijoje visos galimos žinutės įrašo reikšmės yra pateikiamos MessageType įrašo aprašyme.

Gavus bet kokio tipo žinutę, jeigu siuntėjas yra žinomas ir tarp jo ir einamojo mazgo yra užmegztas pilnas komunikavimas, yra vykdomas visiems žinučių tipams bendras žinutės apdorojimas: pagal žinutėje esančią informaciją yra atnaujinamos einamojo mazgo blokinio *currentEpoch* ir žinutės siuntėjo *configEpoch* reikšmės. Tolimesnis žinutės apdorojimas priklauso nuo kiekvieno žinutės tipo.

***Ping* žinutės apdorojimas**

Ping žinutėje papildomai bendros žinutės informacijos yra pridedami paskalų (angl. *gossip*) duomenys apie kitus mazgus: mazgų pavadinimai, jų būsenų žymos (angl. *flags*) ir adreso pasikeitimą nusakanti informacija. Mazgai iš einamojo mazgo žinomų mazgų sąrašo į paskalas yra įtraukiami tokiu principu:

- Yra bandoma įtraukti bent trijų atsitiktinai parinktų ir skirtingų mazgų paskalų informaciją. Renkant mazgus, iš einamojo mazgo mazgų sąrašo į paskalas nepakliūna tie mazgai, kurie yra galimai neaktyvus, kurie yra pradėję užmegzti komunikaciją ir kurie neturi tinklo lizdo kanalo;
- Į paskalas visada yra įtraukiami mazgai, kurie einamojo mazgo požiūriu galimai yra neaktyvus ($\text{"PFAIL"} \in \text{node.flags}$). Taip norima paskleisti žinią apie tokius mazgus visame blokinyje.

Apdorojant *ping* tipo žinutę realioje sistemoje vykdomi veiksmai pateikti 10 pav. esančiame priede nr. 3. Formalioje specifikacijoje *ping* tipo žinutė yra apdorojama ProcessPingMsg veiksme. Apdorojant vykdomi veiksmai:

1. Kadangi buvo gauta *ping* žinutė, siuntėjui yra atsakoma *pong* žinute, siekiant įrodyti siuntėjui, kad *ping* žinutės gavėjas yra aktyvus;
2. Jeigu siuntėjas yra žinomas mazgas, yra vykdomas žinutėje esančios informacijos apdorojimas:
   (a) Jei siuntėjas bando užmegzti komunikaciją ($\text{"HANDSHAKE"} \in \text{sender.flags}$), pagal mazgo būsenos žymas yra nustatoma mazgo rolė: ar siuntėjas yra pagrindinis ar pavaldus mazgas;
   (b) Jei mazgas jau yra pilnai užmezgęs komunikaciją ($\text{"HANDSHAKE"} \notin \text{sender.flags}$) ir yra nustatoma, kad pasikeitė mazgo adresas, yra atnaujinama einamojo mazgo informacija apie siuntėją: pakeičiama adreso ir komunikavimo kanalo būsenos;

(c) Patikrinama, ar siuntėjas nepakeitė savo rolės:

   i. Nustačius, kad siuntėjas iš pavaldaus mazgo tapo pagrindiniu mazgu, yra pakeičiama einamoje mazgo informacija apie siuntėją: iš seno pagrindinio mazgo pavaldžių mazgų aibės yra pašalinamas siuntėjas ir siuntėjas yra paverčiamas pagrindiniu mazgu;

   ii. Nustačius, kad siuntėjas buvęs pagrindiniu mazgu tapo pavaldžiu mazgu, yra atnaujinamos siuntėjo būsenos žymos ir yra pašalinamos atsakomybės už maišos lizdus, paliekant šių lizdų replikavimo atsakomybę.

(d) Patikrinama, ar siuntėjui, jei jis yra pavaldus mazgas, nepasikeitė pagrindinis mazgas. Pasikeitus siuntėjo pagrindiniam mazgui yra atnaujinama maišos lizdų atsakomybių informacija ir siuntėjo pagrindinio mazgo identifikatorius;

(e) Jeigu siuntėjas yra pagrindinis mazgas, yra atnaujinama informacija apie siuntėjo maišos lizdų atsakomybes (realioje sistemoje vykdomi veiksmai pateikti 11 pav. esančiame priede nr. 3). Nustatant, ar įvyko atsakomybių pasikeitimas, yra žiūrima į siuntėjo siųstą ir einamojo mazgo turimą informaciją: maišos lizdo atsakomybė yra pakeičiama, jeigu prieš tai už tą maišos lizdą nebuvo atsakingas kitas mazgas arba jeigu siuntėjo *configEpoch* reikšmė yra didesnė už prieš tai už maišos lizdą atsakingo mazgo *configEpoch* reikšmę. Atnaujinus maišos lizdų atsakomybes, taip pat yra vykdomas einamojo mazgo būsenos atnaujinimas:

   i. Jeigu einamasis mazgas yra pavaldus mazgas ir po atsakomybių pasikeitimų einamasis mazgas liko be maišos lizdų, tai tada einamasis mazgas tampa *ping* žinutės siuntėjo pavaldžiu mazgu;

   ii. Jeigu einamasis mazgas yra pagrindinis mazgas ir jis po atsakomybių atnaujinimo neliko atsakingas už bent vieną maišos lizdą, tai reiškia, kad einamasis mazgas buvo neaktyvus kažkurį laiko tarpą, todėl einamajam mazgui reikia tapti žinutės siuntėjo pavaldžiu mazgu.

(f) Jeigu siuntėjas siųstoje žinutėje informuoja, kad jis yra atsakingas už tam tikrus maišos lizdus, tačiau einamasis mazgas mato, kad už tuos maišos lizdus yra atsakingas kitas mazgas, kurio *configEpoch* reikšmė yra didesnė, tada einamasis mazgas siunčia *update* tipo žinutę *ping* žinutės siuntėjui;

(g) Vykdomas *configEpoch* reikšmės susidūrimo sprendimas, jeigu tiek einamasis mazgas, tiek siuntėjas nurodo, kad turi tokias pačias *configEpoch* reikšmes. Susidūrimo sprendi-

mas: jeigu siuntėjo mazgo identifikatorius yra didesnis, tada einamasis mazgas vienetu padidina savo *configEpoch* ir blokinio *currentEpoch* reikšmes;

(h) Vykdomas paskalose gautos informacijos apdorojimas (realioje sistemoje vykdomi veiksmai pateikti 12 pav. esančiame priede nr. 3):

i. Jeigu siuntėjas yra pagrindinis mazgas ir jis pranešė, kad paskalose esantis žinomas mazgas yra galimai neaktyvus, tai *ping* žinutės siuntėjas yra pridedamas prie sąrašo mazgų, kurie taip pat pranešė, kad tas mazgas yra neaktyvus;

ii. Pagal turimą einamojo mazgo informaciją yra nustatoma, ar paskalose esantis žinomas mazgas yra neaktyvus. Mazgas tampa neaktyviu jeigu dauguma blokinio mazgų paskalose pranešė, kad šis mazgas yra galimai nepasiekiamas. Jeigu einamasis mazgas yra pagrindinis mazgas ir jis tai nustatė, tada jis išsiunčia *fail* tipo žinutes visiems aktyviems blokinio mazgams, su kurias yra užmegzta pilna komunikacija;

iii. Nustačius, kad pagal paskalose esančius duomenis žinomo mazgo adresas pasikeitė, yra atnaujinama mazgo adreso būsena ir yra uždaromas komunikavimo kanalas;

iv. Paskalose esantiems nežinomiems mazgas yra pradedamas vykdyti komunikacijos užmezgimas: prie einamojo mazgo žinomų mazgų yra pridedamas mazgo įrašas, sukurtas pagal paskalose esančią informaciją.

***Pong* žinutės apdorojimas**

Pong tipo žinutės turinys yra toks pat kaip ir *ping* žinutės, tik skiriasi žinutės įrašo lauko type reikšmė. Žinutės sudarymui naudojamas toks pat algoritmas, tačiau egzistuoja nežymūs skirtumai *pong* tipo žinutės apdorojime.

Apdorojant *pong* tipo žinutę realioje sistemoje vykdomi veiksmai pateikti 13 pav. esančiame priede nr. 3. Formalioje specifikacijoje gauta *pong* žinutė yra apdorojama $\mathrm{ProcessPongMsg}$ veiksme. Apdorojant vykdomi veiksmai:

1. Mazgo, kuris bando užmegzti komunikaciją, rolės nustatymas;
2. Atnaujinama siuntėjo būsena, kad buvo gautas atsakas į siųstą *ping* žinutę;
3. Jei ankščiau buvo įtariama, kad siuntėjas yra galimai neaktyvus (”PFAIL” $\in \mathrm{sender.flags}$), yra atstatoma siuntėjo būsenos žyma, kad mazgas vis dėlto yra aktyvus, kadangi buvo gauta *pong* žinutė;

4. Jeigu ankščiau buvo nuspręsta, kad siuntėjas yra neaktyvus ($\text{"FAIL"} \in \text{sender.flags}$), tačiau iš jo buvo gauta *pong* žinutė, yra bandoma atstatyti siuntėjo būseną:

   (a) Jeigu siuntėjas yra pavaldus mazgas, tada iš karto yra atstatoma jo būsena;

   (b) Jeigu siuntėjas yra pagrindinis mazgas, tada, priklausomai nuo to, ar praėjo daugiau nei laukimo laikas nuo mazgo neaktyvios būsenos paskelbimo ir ar mazgas vis dar yra atsakingas už bent vieną maišos lizdą, mazgui yra sugrąžinama aktyvaus mazgo būsena. Jeigu siuntėjas netenkina šių sąlygų, mazgas einamajam mazgui lieka neaktyvioje būsenoje.

5. Patikrinama, ar siuntėjas nepakeitė rolės;
6. Patikrinama, ar siuntėjui, jeigu jis yra pavaldus mazgas, nepasikeitė pagrindinis mazgas;
7. Jeigu siuntėjas yra pagrindinis mazgas, yra atnaujinama informacija apie siuntėjo maišos lizdų atsakomybes;
8. Išsiunčiama *update* žinutė *pong* žinutės siuntėjui, aptikus maišos lizdų ir mazgų *configEpoch* reikšmių neatitikimus;
9. Vykdomas aptiktos *configEpoch* reikšmės susidūrimo problemos sprendimas;
10. Vykdomas paskalose gautos informacijos apdorojimas.

***Update* žinutės apdorojimas**

Formalioje specifikacijoje *update* tipo žinutė yra apdorojama ProcessUpdateMsg veiksme. Jeigu žinutės siuntėjas yra einamajam mazgui žinomas mazgas ir iš tikrųjų mazgo, apie kurį buvo gauta *update* informacija, *configEpoch* reikšmė yra mažesnė nei žinutėje siųsta reikšmė, apdorojant žinutę yra vykdomi veiksmai:

1. Einamojo mazgo būsenoje mazgas, apie kurį buvo gauta informacija, yra paverčiamas pagrindiniu mazgu;
2. Atnaujinama mazgo *configEpoch* reikšmė į tą, kuri buvo gauta *update* žinutės dalyje;
3. Atnaujinamos mazgo maišos lizdų atsakomybės pagal *update* žinutėje esančią informaciją.

***Meet* žinutės apdorojimas**

Meet tipo žinutės turinys yra toks pat kaip ir *ping* bei *pong* žinučių, tik skiriasi žinutės įrašo lauko type reikšmė. Žinutės sudarymui naudojamas toks pat algoritmas, tačiau skiriasi vykdomas

žinutės apdorojimas.

Apdorojant *meet* tipo žinutę realioje sistemoje vykdomi veiksmai pateikti 14 pav. esančiame priede nr. 3. Formalioje specifikacijoje *meet* tipo žinutė yra apdorojama $\mathrm{ProcessMeetMsg}$ veiksme. Apdorojant vykdomi veiksmai:

1. Jeigu siuntėjas nėra einamajam mazgui žinomas mazgas, yra atliekami veiksmai:
   (a) Einamojo mazgo blokinio būsenoje yra pridedamas naujas mazgas, sukurtas remiantis informacija gauta *meet* žinutėje;
   (b) Yra atliekamas paskalose gautos informacijos apdorojimas;
   (c) Siuntėjui yra atsakoma *pong* žinute.
2. Jeigu siuntėjas yra einamajam mazgui žinomas mazgas, yra atliekami veiksmai:
   (a) Mazgo, kuris bando užmegzti komunikaciją, rolės nustatymas;
   (b) Yra patikrinama, ar siuntėjas nepakeitė rolės;
   (c) Patikrinama, ar siuntėjui, jeigu jis yra pavaldus mazgas, nepasikeitė pagrindinis mazgas;
   (d) Jeigu siuntėjas yra pagrindinis mazgas, yra atnaujinama informacija apie siuntėjo maišos lizdų atsakomybes;
   (e) Išsiunčiama *update* žinutė *meet* žinutės siuntėjui, aptikus maišos lizdų ir mazgų *configEpoch* reikšmių neatitikimus;
   (f) Vykdomas aptiktos *configEpoch* reikšmės susidūrimo problemos sprendimas;
   (g) Vykdomas paskalose gautos informacijos apdorojimas.

***Fail* žinutės apdorojimas**

Fail tipo žinutės sudarymas vykdomas prie bendros žinutės informacijos papildomai pridedant mazgo, apie kurio neaktyvumą yra norima pranešti, identifikatorių.

Formalioje specifikacijoje *fail* tipo žinutė yra apdorojama $\mathrm{ProcessFailMsg}$ veiksme. Apdorojant žinutę, jeigu žinutės siuntėjas yra žinomas, aktyvus ($\text{"FAIL"} \notin \mathrm{sender.flags}$) mazgas, yra atnaujinama einamojo mazgo blokinio būsena pakeičiant *fail* žinutėje nurodyto mazgo būseną į neaktyvią.

Mfstart žinutės apdorojimas

Mfstart žinutė yra gaunama, kai yra pradėtas vykdyti einamojo mazgo, kuris yra pagrindinis mazgas, rankinis persijungimas (angl. *manual failover*). Šio tipo žinutės neturi papildomos informacijas ir turi tik bendrą žinutės informaciją.

Formalioje specifikacijoje *mfstart* tipo žinutė yra apdorojama $\mathrm{ProcessMFStartMsg}$ veiksme. Apdorojant žinutę, jeigu žinutės siuntėjas yra žinomas einamojo pagrindinio mazgo pavaldus mazgas ($\mathrm{sender.slaveOf} = \mathrm{cluster.myself}$), yra atnaujinama einamojo mazgo blokinio būsena pakeičiant informaciją, kad yra pradėtas rankinis persijungimas ir jį pradėjo žinutės siuntėjas.

4.2.5. Išoriniai įvykiai

Apdorojant gautas žinutes, vykdomi veiksmai kartais priklauso nuo to, ar įvyko išoriniai įvykiai. „Redis Cluster“ sistemoje žinučių apdorojimas labai priklauso nuo laiko įvykių. Taip pat šalia laiko įvykių gali įvykti kiti įvykiai (adreso pasikeitimas) lemiantys skirtingą žinutėje esančios informacijos apdorojimą.

Laiko įvykiai

„Redis Cluster“ sistemoje daug veiksmų priklauso nuo laiko, todėl specifikacijoje buvo modeliuojama laiko tėkmė. Nors ir TLA$^+$ formali specifikavimo kalba leidžia sumodeliuoti laiką ir laiko tėkmę, tačiau nenorint padaryti specifikacijos dar sudėtingesnės, laikas ir laukimo laikų pabaigos yra išreikštos tam tikru veiksmu, kuris arba įvyko, arba ne.

$$
\begin{aligned}
&\mathrm{TimeActions}(n) \triangleq \\
&\quad \lor \mathrm{CronJobTime}(n) \\
&\quad \lor \mathrm{PingPongHalfTimeout}(n) \\
&\quad \lor \mathrm{PingPongTimeout}(n) \\
&\quad \lor \mathrm{PingNotSentAndOldRecievedPong}(n) \\
&\quad \lor \mathrm{NoAnswerAfterHandshake}(n) \\
&\quad \lor \mathrm{FailTimeout}(n) \\
&\quad \lor \mathrm{MFTimeOut}(n)
\end{aligned}
$$

Žinučių apdorojimo formalioje specifikacijoje yra apibrėžti šie laiko įvykiai:

- Atėjo laikas modeliuojamame įvykių cikle vykdyti blokinio suplanuotą darbą (veiksmas $\mathrm{CronJobTime}$). Įvykus šiam laiko įvykiui yra pakeičiama įvykių ciklo būsena, bei taip pat yra atnaujinamas laiko įvykio skaitliukas: norima, kad kas dešimtą kartą vykdant blokinio suplanuotą darbą būtų išsiųsta *ping* žinutė atsitiktinai pasirinktam mazgui;

- Išsiuntus *ping* žinutę, praėjo pusė *pong* žinutės laukimo laiko (veiksmas PingPongHalfTimeout). Įvykus tokiai situacijai, vykdant blokinio suplanuotą darbą yra uždaromas komunikavimo kanalas – galbūt gavėjas yra aktyvus, tačiau yra komunikavimo kanalo problemų;
- Išsiuntus *ping* žinutę, atsako žinutė *pong* nebuvo gauta daugiau nei žinutės laukimo laikas (veiksmas PingPongTimeout). Blokinio suplanuotame darbe apdorojant tokią situaciją yra įtariama, kad mazgas yra galimai neaktyvus;
- Šiuo metu mazgui nėra dar išsiųstos *ping* žinutės ir labai seniai buvo vykdomas komunikavimas su šiuo mazgu (veiksmas PingNotSentAndOldRecievedPong). Įvykus šiam laiko įvykiui, įvykių cikle vykdant suplanuotą blokinio darbą tokiam mazgui yra išsiunčiama *ping* žinutė;
- Mazgas buvo pradėjęs užmegzti komunikaciją su einamuoju mazgu, tačiau jos netęsė daugiau nei laukimo laikas (veiksmas NoAnswerAfterHandshake). Praėjus šiam laikui, vykdant suplanuotą blokinio darbą, mazgas yra pašalinamas iš žinomų mazgų sąrašo;
- Nuo mazgo paskelbimo neaktyviu praėjo daugiau nei laukimo laikas (veiksmas FailTimeout). Praėjus šiam laikui ir pagrindiniam mazgui bandant sugrįžti į blokinį, nebegalima atstatyti mazgo būsenos į aktyvią;
- Buvo pradėtas rankinis persijungimas ir jis užsitęsė daugiau nei laukimo laikas (veiksmas MFTimeOut). Praėjus šiam laikui yra nutraukiamas dabar vykdomas rankinis persijungimas.

Adreso pasikeitimo įvykis

Žinučių apdorojime taip pat yra apdorojamas įvykis, kai blokinyje vienas iš mazgų pradeda bendrauti su kitais mazgais naudojant kitą adresą. Siekiant sumodeliuoti šią situaciją, prie mazgą reprezentuojančio įrašo yra pridėtas loginio tipo laukas hasAddressChanged, kuriame saugoma informacija, ar mazgo adresas pakito. Adreso pasikeitimo įvykio metu (veiksmas ChangeAddress) šio lauko reikšmė yra keičiama tik mazgams, kurie jau turi užmezgę komunikavimo kanalą su adresą pakeitusiu mazgu.

4.2.6. Blokinio administravimo komandų apdorojimas

$$\mathrm{ClientAdminCommand}(n) \triangleq$$
$$\lor\ \mathrm{MeetCommand}(n)$$

$\lor$ FlushSlotsCommand(n)
$\lor$ AddSlotsCommand(n)
$\lor$ DelSlotsCommand(n)
$\lor$ SetSlotCommand(n)
$\lor$ BumpEpochCommand(n)
$\lor$ ForgetCommand(n)
$\lor$ ReplicateCommand(n)
$\lor$ FailOverCommand(n)

Išorinių veiksmų formalioje specifikacijoje buvo specifikuoti veiksmai, leidžiantys konfigūruoti ir administruoti „Redis Cluster“ sistemą. Norint turėti detalesnę žinučių apdorojimo specifikaciją, remiantis išorinių veiksmų specifikacija bei „Redis Cluster“ dokumentacija, žinučių apdorojimo specifikacijoje taip pat buvo specifikuotas blokinio administravimo komandų apdorojimas. Žinučių apdorojimo formalioje specifikacijoje buvo specifikuotos tik tos komandos, kurios keičia blokinio būseną ir yra susijusios su žinutėmis, kurios keliauja tarp blokinio mazgų. Kitos komandos, kurios nebuvo specifikuotos, yra susijusios su blokinio būsenos pateikimu ar su sistemos dalimis, kurios nėra aktualios šioje specifikacijoje.

CommandsExecutionPredicate(n) $\triangleq$
$\land$ eventLoop[n].state = "PROCESSING_FILE_EVENTS"
$\land$ eventLoop[n].currentEventBeingProcessed = NULL_EventLink

Visos administravimo komandos blokinyje gali būti apdorotos tik tada, kai įvykių cikle yra vykdomas skaitymo arba rašymo įvykių apdorojimas ir kai šiuo metu nėra pasirinktas kanalas, kurį reikia apdoroti (veiksmas CommandsExecutionPredicate).

„Redis Cluster“ sistemoje blokinio administravimo komandas galima siųsti tiesiogiai blokinio mazgams TCP/IP protokolu arba galima naudotis *redis-cli* tekstine sąsaja (angl. *command line interface*). Tekstine sąsaja blokinio administravimas gali būti vykdomas: 1) interaktyviu būdu, kai yra siunčiamos komandos, kurias sugeba apdoroti blokinio mazgai; arba 2) naudojantis *redis-cli* blokinio konfigūravimo funkcijomis, kurios pagreitina ir supaprastina blokinio administravimą. Tekstinė sąsaja vykdydama konfigūravimo funkcijas iš tiesų siunčia tas pačias komandas, kurias galima siųsti tiesiogiai į blokinio mazgus TCP/IP protokolu. Tekstinė sąsaja *redis-cli*, vykdydama blokinio konfigūravimo funkcijas, kelias administravimo komandas sujungia į loginius vienetus – transakcijas.

Formalioje specifikacijoje specifikuojant blokinio administravimo veiksmus taip pat buvo įvertinta, ar verta įtraukti transakcijas į specifikaciją ir ar verta vietoje komandų specifikuoti *redis-cli* konfigūravimo funkcijas. Išnagrinėjus kaip veikia transakcijos „Redis Cluster“ sistemoje, buvo

nustatyta, kad „Redis Cluster“ sistemos transakcijos tik užtikrina, kad veiksmai esantys transakcijoje yra atliekami iš eilės ir įvykus klaidai yra toliau tęsiamas transakcijos vykdymas. Būtent dėl nesančio transakcijų atgalinio pervyniojimo (angl. *rollback*) buvo nuspręsta į specifikaciją nepridėti transakcijų mechanizmo, kuris formalią specifikaciją padarytų dar labiau sudėtingesnę. Pagrindinis „Redis Cluster“ sistemos transakcijų funkcionalumas – veiksmų vykdymas iš eilės, bus padengtas viena iš galimų būsenų mašinos būsenų aibių, kai visi komandų veiksmai specifikacijos modelyje bus atlikti iš eilės kaip ir vykdant „Redis Cluster“ transakciją.

Nagrinėjant, ar verčiau specifikuoti *redis-cli* konfigūravimo funkcijas (keleto komandų apjungimą), buvo nuspręsta specifikuoti pavienes blokinio administravimo komandas. Tokį pasirinkimą lėmė tai, kad *redis-cli* nėra vienintelė tekstinė sąsaja, kurią galima naudoti konfigūruojant blokinį, ir komandų seka, apjungta vienos *redis-cli* konfigūravimo funkcijos, gali skirtis tarp skirtingų tekstinių sąsajų implementacijų. Be to, „Redis Cluster“ sistemos naudotojų niekas neverčia naudotis tekstine sąsaja ir naudotojai gali tiesiogiai komunikuoti su blokinio mazgais TCP/IP protokolu.

***Cluster meet* komanda**

Cluster meet komanda yra naudojama, kai yra norima prie blokinio pridėti naują mazgą. Siunčiant šią komandą vienam iš blokinio mazgų, einamasis mazgas pagal komandos parametrus (IP adresą ir prievadą (angl. *port*)) prideda naują mazgą prie blokinio žinomų mazgų sąrašo. Pridėjus naują mazgą, vėliau einamasis mazgas bandys užmegzti komunikacijos kanalą su šiuo mazgu siunčiant *meet* tipo žinutę.

Norint sukurti blokinį, nebūtina siųsti *cluster meet* komandos visiems mazgams, kurie sudarys blokinį. Mazgui užmezgus komunikaciją su kitu mazgu, paskalų protokolo pagalba apie naują blokinio mazgą sužinos ir kiti mazgai. Taip pat, norint sujungti du mazgus, *cluster meet* komandą užtenka išsiųsti tik vienam iš tų mazgų.

$$
\begin{aligned}
&\mathrm{MeetCommand(n)} \triangleq \\
&\quad \land \mathrm{CommandsExecutionPredicate(n)} \\
&\quad \land \exists\, \mathrm{m} \in \mathrm{NODE} : \\
&\qquad \land \mathrm{n} \neq \mathrm{m} \\
&\qquad \land \mathrm{clusterState[n].nodes[m]} = \mathrm{NULL_ClusterNodeType} \\
&\qquad \land \mathrm{clusterState}' = [\mathrm{clusterState}\ \textsc{except}\ ![\mathrm{n}] = \\
&\qquad\qquad \mathrm{ClusterAddNode(clusterState[n],} \\
&\qquad\qquad\qquad\qquad \mathrm{CreateClusterNode(m, \{\text{“HANDSHAKE”}, \text{“MEET”}\}))]} \\
&\qquad \land \textsc{unchanged}\ \langle \mathrm{eventLoop, recieveBuffer, sendBuffer} \rangle
\end{aligned}
$$

Formalioje specifikacijoje komandos *cluster meet* apdorojimas yra išreikštas $\mathrm{MeetCommand}$

veiksmu. Šis veiksmas įvyksta tik tada, kai egzistuoja kitas mazgas, kuris dar nėra žinomas einamajam mazgui. Įvykus šiam veiksmui yra atnaujinama einamojo mazgo $\mathrm{clusterState}$ kintamojo reikšmė pridedant naują mazgą, su kuriuo vėliau reikės užmegzti komunikaciją $(\text{"HANDSHAKE"} \in \mathrm{node.flags} \land \text{"MEET"} \in \mathrm{node.flags})$.

Cluster flushslots komanda

Cluster flushslots komanda leidžia pašalinti mazgo atsakomybę už visus jam priklausančius maišos lizdus. Realioje sistemoje šią komandą galima iškviesti tik tada, kai visi maišos lizdai, už kuriuos yra atsakingas mazgas, yra tušti.

$$
\begin{aligned}
&\mathrm{FlushSlotsCommand}(n) \triangleq \\
&\quad \land \mathrm{CommandsExecutionPredicate}(n) \\
&\quad \land \mathrm{clusterState}' = \\
&\qquad [\mathrm{clusterState}\ \textsc{except} \\
&\qquad\quad ![n] = [@\ \textsc{except} \\
&\qquad\qquad !.\mathrm{nodes} = [@\ \textsc{except} \\
&\qquad\qquad\qquad ![\mathrm{clusterState}[n].\mathrm{myself}] = \\
&\qquad\qquad\qquad\quad [@\ \textsc{except} \\
&\qquad\qquad\qquad\qquad !.\mathrm{slots} = [\mathrm{slot} \in \mathrm{Slots} \mapsto \textsc{false}]]], \\
&\qquad\qquad !.\mathrm{slots} = [\mathrm{slot} \in \mathrm{Slots} \mapsto \textsc{if}\ @[\mathrm{slot}] = \mathrm{clusterState}[n].\mathrm{myself} \\
&\qquad\qquad\qquad\qquad \textsc{then}\ \mathrm{NULL_NODE} \\
&\qquad\qquad\qquad\qquad \textsc{else}\ @[\mathrm{slot}]], \\
&\qquad\qquad !.\mathrm{todoBeforeSleep} = @ \cup \{\text{“UPDATE_STATE”}\}]] \\
&\quad \land \textsc{unchanged}\ \langle \mathrm{eventLoop}, \mathrm{recieveBuffer}, \mathrm{sendBuffer} \rangle
\end{aligned}
$$

Formalioje specifikacijoje komanda *cluster flushslots* yra apdorojama $\mathrm{FlushSlotsCommand}$ veiksme. Veiksmo metu yra atnaujinama einamojo mazgo blokinio būsena pašalinant atsakomybes už visus maišos lizdus. Specifikacijoje, skirtingai nei realioje sistemoje, nėra atsižvelgiama į tai, ar maišos lizdai yra tušti – kadangi specifikacija neaprašo duomenų saugojimo, tai yra daroma prielaida, kad maišos lizdai yra tušti ir šis veiksmas gali įvykti visada.

Cluster addslots komanda

Norint mazgui pridėti atsakomybę už maišos lizdus, galima naudoti *cluster addslots* komandą. Ši komanda veikia tik tada, kai už komandos parametruose nurodytus maišos lizdus dar nėra atsakingas kitas blokinio mazgas. Įvykdžius šią komandą taip pat yra atstatoma maišos lizdo importavimo būsena.

Cluster addslots komanda dažniausiai yra naudojama, kai yra kuriamas naujas blokinys ir mazgams, kurie jau užmezgė komunikaciją, reikia paskirstyti atsakomybes už maišos lizdus. Taip pat ši komanda yra naudojama norint rankiniu būdu (angl. *manually*) sutaisyti blokinį, kai veikiant sistemai susidarė situacija, kad maišos lizdai yra blogai priskirti arba nepriskirti tam tikram mazgui, kuris už tuos maišos lizdus iš tikrųjų yra atsakingas.

$$
\begin{array}{l}
\mathrm{AddDelSlotsCmd}(n, \mathrm{isDelete}) \triangleq \\
\quad \exists\, \mathrm{slotToChange} \in \mathrm{Slots} : \\
\quad\quad \land\ \textsc{if}\ \mathrm{isDelete} \\
\quad\quad\quad \textsc{then}\ \mathrm{clusterState}[n].\mathrm{slots}[\mathrm{slotToChange}] \neq \mathrm{NULL_NODE} \\
\quad\quad\quad \textsc{else}\ \ \mathrm{clusterState}[n].\mathrm{slots}[\mathrm{slotToChange}] = \mathrm{NULL_NODE} \\
\quad\quad \land\ \mathrm{clusterState}' = \\
\quad\quad\quad [\mathrm{clusterState}\ \textsc{except} \\
\quad\quad\quad\quad ![n] = [@\ \textsc{except} \\
\quad\quad\quad\quad\quad !.\mathrm{slots} = [@\ \textsc{except}\ ![\mathrm{slotToChange}] = \textsc{if}\ \mathrm{isDelete} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textsc{then}\ \mathrm{NULL_NODE} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textsc{else}\ \ \mathrm{clusterState}[n].\mathrm{myself}], \\
\quad\quad\quad\quad\quad !.\mathrm{todoBeforeSleep} = @ \cup \{\text{“UPDATE_STATE”}\}, \\
\quad\quad\quad\quad\quad !.\mathrm{nodes} = [@\ \textsc{except} \\
\quad\quad\quad\quad\quad\quad\quad ![\mathrm{clusterState}[n].\mathrm{myself}] = \\
\quad\quad\quad\quad\quad\quad\quad\quad [@\ \textsc{except}\ !.\mathrm{slots} = \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad [@\ \textsc{except}\ ![\mathrm{slotToChange}] = \textsc{if}\ \mathrm{isDelete} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textsc{then}\ \textsc{false} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textsc{else}\ \ \textsc{true}]]], \\
\quad\quad\quad\quad\quad !.\mathrm{importingSlotsFrom} = [@\ \textsc{except}\ ![\mathrm{slotToChange}] = \mathrm{NULL_NODE}]]] \\
\quad\quad \land\ \textsc{unchanged}\ \langle \mathrm{eventLoop}, \mathrm{recieveBuffer}, \mathrm{sendBuffer} \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathrm{AddSlotsCommand}(n) \triangleq \\
\quad \land\ \mathrm{CommandsExecutionPredicate}(n) \\
\quad \land\ \mathrm{AddDelSlotsCmd}(n, \textsc{false})
\end{array}
$$

Formalioje specifikacijoje *cluster addslots* komanda yra aprašyta $\mathrm{AddSlotsCommand}$ veiksme. Šis veiksmas yra paremtas kitu veiksmu $\mathrm{AddDelSlotsCmd}$, kuris apjungia vykdomus veiksmus pridedant ar pašalinant maišos lizdų atsakomybes. Veiksmas $\mathrm{AddDelSlotsCmd}$ gali įvykti tik tada, jeigu egzistuoja maišos lizdas, už kurį einamojo mazgo požiūriu dar nėra atsakingas kitas mazgas ($\mathrm{clusterState}[n].\mathrm{slots}[\mathrm{slotToChange}] = \mathrm{NULL_NODE}$). Be to, skirtingai nei realioje sistemoje, šio veiksmo metu yra pakeičiamas tik vienas, o ne keletas maišos lizdų.

Cluster delslots komanda

Cluster delslots komanda leidžia pašalinti mazgo atsakomybę už tam tikrus maišos lizdus. Pašalinus atsakomybę už maišos lizdus, šie maišos lizdai tampa nepriskirti jokiam mazgui ir juos galima priskirti kitam mazgui naudojant *cluster addslots* komandą. *Cluster delslots* komandą galima įvykdyti tik tada, kai už tuos maišos lizdus yra atsakingas mazgas. Tačiau, įvykdžius šią komandą, blokinys gali pereiti į *FAIL* būseną, jei yra naudojamas blokinio konfigūracijos parametras reikalaujantis, kad už kiekvieną maišos lizdą turi būtų atsakingas tam tikras mazgas.

$$\begin{aligned}&\mathrm{DelSlotsCommand(n)} \triangleq \\ &\quad \land \mathrm{CommandsExecutionPredicate(n)} \\ &\quad \land \mathrm{AddDelSlotsCmd(n, \textsc{true})}\end{aligned}$$

Formalioje specifikacijoje *cluster delslots* komanda yra aprašyta $\mathrm{DelSlotsCommand}$ veiksme. Šis veiksmas yra paremtas kitu veiksmu $\mathrm{AddDelSlotsCmd}$ ir gali įvykti tik tada, kai egzistuoja maišos lizdas, už kurį yra atsakingas mazgas ($\mathrm{clusterState[n].slots[slotToChange]} \neq \mathrm{NULL_NODE}$).

Cluster setslot komanda

Norint atlikti maišos lizdų pertvarkymą (angl. *resharding*) galima naudoti *cluster setslot* komandą. Vykdant pertvarkymą galima perkelti tik vieną maišos lizdą. Maišos lizdų pertvarkymo procesas susideda iš veiksmų:

1. Maišos lizdas pereina į migruojančią (angl. *migrating*) būseną mazge, kuris dabar yra atsakingas už tą maišos lizdą;
2. Maišos lizdas pereina į importuojančią (angl. *importing*) būseną mazge, kuris perims maišos lizdą;
3. Maišos lizde laikomi duomenys yra perkeliami iš pirminio (angl. *source*) mazgo į tikslinį (angl. *target*) mazgą naudojantis *cluster getkeyinslot* (komanda leidžianti sužinoti visus raktus laikomus maišos lizde) ir *migrate* (komanda leidžianti perkelti raktus iš vieno mazgo į kitą mazgą) komandomis;
4. Maišos lizdo pertvarkymas yra užbaigiamas pranešant pirminiam ir piksliniam mazgui apie užbaigtą maišos lizdo pertvarkymo procesą.

Visus veiksmus susijusius su maišos lizdų pertvarkymu galima atlikti su komandos *cluster setslot* subkomandomis, kurias galima nusiųsti tik mazgams, kurie yra pagrindiniai mazgai:

- ***Cluster setslot <maišos lizdas> migrating <tikslinis mazgas>***: ši subkomanda pakeičia maišos lizdo būseną į migruojančią. Šią komandą galima iškviesti tik tada, kai mazgas yra atsakingas už tą maišos lizdą;
- ***Cluster setslot <maišos lizdas> importing <pirminis mazgas>***: ši subkomanda pakeičia maišos lizdo būseną į importuojančią. Šią komandą galima iškviesti tik tada, kai mazgas dar nėra atsakingas už maišos lizdą nurodytą komandoje;
- ***Cluster setslot <maišos lizdas> node <mazgas>***: ši komanda susieja maišos lizdą su komandoje nurodytu mazgu. Nusiuntus šią komandą pirminiam mazgui, jeigu visi duomenys buvo perkelti ir maišos lizdas yra tuščias, yra išvaloma (angl. *cleared*) migruojanti maišos lizdo būsena. Nusiuntus šią komandą tiksliniam mazgui, apdorojant šią komandą yra atliekami veiksmai: 1) išvaloma importuojanti maišos lizdo būsena; ir 2) yra padidinama mazgo *configEpoch* reikšmė, jei ji dar nebuvo pati didžiausia blokinyje. Įvykęs maišos lizdo atsakomybės pasikeitimas galiausiai pasieks ir kitus mazgus, tačiau norint pagreitinti procesą, šią subkomandą taip pat galima nusiųsti ir kitiems pagrindiniams mazgams (ne tik pirminiam ir tiksliniam mazgui). Apdorojant šią subkomandą mazge, kuris nedalyvavo maišos lizdų pertvarkyme, subkomandoje nurodytas maišos lizdas bus priskirtas subkomandoje nurodytam mazgui;
- ***Cluster setslot <maišos lizdas> stable***: jeigu norima atšaukti maišos lizdo pertvarkymą, ši komanda leidžia išvalyti migruojančią ir importuojančią maišos lizdo būsenas.

$$
\begin{aligned}
&\mathrm{SetSlotCommand}(n) \triangleq \\
&\quad \lor\ \mathrm{SetSlotMigrating}(n) \\
&\quad \lor\ \mathrm{SetSlotImporting}(n) \\
&\quad \lor\ \mathrm{SetSlotStable}(n) \\
&\quad \lor\ \mathrm{SetSlotNode}(n)
\end{aligned}
$$

Formalioje specifikacijoje *cluster setslot* komanda ir subkomandos yra aprašytos $\mathrm{SetSlotCommand}$ veiksme. Šis veiksmas gali įvykti tik tada, kai einamasis mazgas yra pagrindinis mazgas ($\text{"SLAVE"} \notin \mathrm{clusterState}[n].\mathrm{nodes}[n].\mathrm{flags}$).

Cluster bumpepoch **komanda**

$$
\begin{aligned}
&\mathrm{BumpEpochCommand}(n) \triangleq \\
&\quad \land\ \mathrm{CommandsExecutionPredicate}(n) \\
&\quad \land\ \textsc{let}\ \mathrm{max} \triangleq \mathrm{ClusterGetMaxEpoch}(\mathrm{clusterState}[n])
\end{aligned}
$$

$$
\begin{array}{l}
\quad\quad \text{myself} \triangleq \text{ClusterLookupNode}(\text{clusterState}[n], \text{clusterState}[n].\text{myself}) \\
\quad \textsc{in} \quad \lor \text{myself.configEpoch} = 0 \\
\quad\quad\quad \lor \text{myself.configEpoch} \neq \text{max} \\
\land \text{clusterState}' = [\text{clusterState } \textsc{except } ![n] = \\
\quad\quad\quad\quad \text{ClusterBumpConfigEpochWithoutConsensus}(@)] \\
\land \textsc{unchanged } \langle \text{eventLoop}, \text{recieveBuffer}, \text{sendBuffer} \rangle
\end{array}
$$

Cluster bumpepoch komanda yra naudojama, kai yra norima dirbtinai padidinti *configEpoch* reikšmę be pagrindinių mazgų konsensuso. *ConfigEpoch* reikšmė bus padidinta tik tada, kai dabartinė mazgo *configEpoch* reikšmė yra lygi nuliui arba mazgo *configEpoch* reikšmė yra mažesnė, nei didžiausia *configEpoch* reikšmė esanti blokinyje. Formalioje specifikacijoje ši komanda yra išreikšta veiksmu BumpEpochCommand.

***Cluster forget* komanda**

Norint pašalinti mazgą iš žinomų blokinį sudarančių mazgų sąrašo, galima naudoti komandą *cluster forget*. Šią komandą reikia nusiųsti visiems blokinį sudarantiems mazgams, kadangi užmirštas mazgas gali iš naujo būti pridėtas apdorojant paskalų informaciją esančią *ping* ir *pong* žinutėse. Norint užtikrinti, kad mazgas nebus pridėtas apdorojant paskalų informaciją, yra naudojamas juodasis sąrašas, kuriame yra saugomi mazgai, kurie jau buvo užmiršti.

$$
\begin{array}{l}
\text{ForgetCommand}(n) \triangleq \\
\quad \land \text{CommandsExecutionPredicate}(n) \\
\quad \land \exists\, m \in \text{NODE} : \\
\quad\quad \land n \neq m \\
\quad\quad \land \text{clusterState}[n].\text{nodes}[m] \neq \text{NULL_ClusterNodeType} \\
\quad\quad \land \text{“SLAVE”} \in \text{clusterState}[n].\text{nodes}[n].\text{flags} \implies \text{clusterState}[n].\text{nodes}[n].\text{slaveOf} \neq m \\
\quad\quad \land \text{clusterState}' = \\
\quad\quad\quad [\text{clusterState } \textsc{except } ![n] = \\
\quad\quad\quad\quad [@ \ \textsc{except} \\
\quad\quad\quad\quad\quad !.\text{nodes} = [x \in \text{NODE} \mapsto \\
\quad\quad\quad\quad\quad\quad\quad \textsc{if } @[x] \neq \text{NULL_ClusterNodeType} \\
\quad\quad\quad\quad\quad\quad\quad \textsc{then if } x \neq m \\
\quad\quad\quad\quad\quad\quad\quad\quad \textsc{then } \text{ProcessDeletedNodes}(@[x], \{m\}) \\
\quad\quad\quad\quad\quad\quad\quad\quad \textsc{else } \text{NULL_ClusterNodeType} \\
\quad\quad\quad\quad\quad\quad\quad \textsc{else } @[x]], \\
\quad\quad\quad\quad\quad !.\text{slots} = [\text{slot} \in \text{Slots} \mapsto \textsc{if } @[\text{slot}] = m \\
\quad\quad\quad\quad\quad\quad\quad\quad \textsc{then } \text{NULL_NODE} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textsc{else } @[\text{slot}]], \\
\quad\quad\quad\quad\quad !.\text{todoBeforeSleep} = @ \cup \{\text{“UPDATE_STATE”}\},
\end{array}
$$

$$
\begin{aligned}
&!.\text{nodesBlackList} = @ \cup \{\text{m}\},\\
&!.\text{migratingSlotsTo} = [\text{slot} \in \text{Slots} \mapsto \textsc{if}\ @[\text{slot}] = \text{m}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textsc{then}\ \text{NULL_NODE}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textsc{else}\ \ @[\text{slot}]],\\
&!.\text{importingSlotsFrom} = [\text{slot} \in \text{Slots} \mapsto \textsc{if}\ @[\text{slot}] = \text{m}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textsc{then}\ \text{NULL_NODE}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textsc{else}\ \ @[\text{slot}]]]]\\
\land\ &\textsc{unchanged}\ \langle \text{eventLoop}, \text{recieveBuffer}, \text{sendBuffer} \rangle
\end{aligned}
$$

Formalioje specifikacijoje komanda *cluster forget* yra apdorojama ForgetCommand veiksme. Šis veiksmas įvyksta tik tada, kai egzistuoja kitas žinomas mazgas ($\text{clusterState}[\text{n}].\text{nodes}[\text{m}] \neq \text{NULL_ClusterNodeType}$). Be to, jei einamasis mazgas yra pavaldus mazgas, tai mazgas, kurį norima užmiršti, negali būti einamoje mazgo pagrindinis mazgas (”SLAVE” $\in \text{clusterState}[\text{n}].\text{nodes}[\text{n}].\text{flags} => \text{clusterState}[\text{n}].\text{nodes}[\text{n}].\text{slaveOf} \neq \text{m}$).

Cluster replicate **komanda**

Cluster replicate komanda leidžia mazgui tapti komandoje nurodyto pagrindinio mazgo pavaldžiu mazgu. Apdorojus šią komandą einamasis mazgas pradeda komunikuoti su pagrindiniu mazgu dėl duomenų replikavimo. Įvykus pasikeitimams papildomai nereikia pranešti kitiems blokinio mazgams apie naują pavaldų mazgą – ši informacija kitus mazgus pasieks paskalų protokolo pagalba. Be to, šią komandą taip pat galima nusiųsti ir pagrindiniam mazgui, tačiau tokiu atveju pagrindinis mazgas neturi būti atsakingas už nei vieną maišos lizdą.

$$
\begin{aligned}
&\text{ReplicateCommand(n)} \triangleq\\
&\quad \land\ \text{CommandsExecutionPredicate(n)}\\
&\quad \land\ \lor\ \text{“MASTER”} \notin \text{clusterState}[\text{n}].\text{nodes}[\text{n}].\text{flags}\\
&\qquad\ \lor\ \text{GetNodeNumSlots}(\text{clusterState}[\text{n}].\text{nodes}[\text{n}]) = 0\\
&\quad \land\ \exists\, \text{newMaster} \in \text{NODE} :\\
&\qquad \land\ \text{n} \neq \text{newMaster}\\
&\qquad \land\ \text{clusterState}[\text{n}].\text{nodes}[\text{newMaster}] \neq \text{NULL_ClusterNodeType}\\
&\qquad \land\ \text{“SLAVE”} \notin \text{clusterState}[\text{n}].\text{nodes}[\text{newMaster}].\text{flags}\\
&\qquad \land\ \text{clusterState}' =\\
&\qquad\quad [\text{clusterState}\ \textsc{except}\\
&\qquad\qquad ![\text{n}] = [\text{ClusterSetMaster}(\text{clusterState}[\text{n}], \text{newMaster})\ \textsc{except}\\
&\qquad\qquad\qquad\qquad !.\text{todoBeforeSleep} = @ \cup \{\text{“UPDATE_STATE”}\}]]\\
&\qquad \land\ \textsc{unchanged}\ \langle \text{eventLoop}, \text{recieveBuffer}, \text{sendBuffer} \rangle
\end{aligned}
$$

Formalioje specifikacijoje komanda *cluster replicate* yra išreikšta ReplicateCommand veiksme. Šis veiksmas gali įvykti tik tada, jei egzistuoja kitas nepavaldus

("SLAVE" $\notin$ clusterState[n].nodes[newMaster].flags) bei žinomas (clusterState[n].nodes[m] $\neq$ NULL_ClusterNodeType) mazgas ir einamasis mazgas nėra pagrindinis mazgas arba, jei einamasis mazgas yra pagrindinis mazgas, tai jis neturi būti atsakingas už nei vieną maišos lizdą (GetNodeNumSlots(clusterState[n].nodes[n]) = 0).

***Cluster failover* komanda**

Cluster failover komanda yra naudoja tada, kai yra norima, kad pavaldus mazgas pradėtų rankinį pagrindinio mazgo persijungimą. Rankinis persijungimas leidžia pavaldžiam mazgui perimti pagrindinio mazgo rolę nepaisant to, ar pagrindinis mazgas yra tapęs nepasiekiamu.

Cluster failover komanda papildomai leidžia nurodyti dvi parinktis (angl. *options*):

- ***Force***: rankinis persijungimas yra pradedamas iš karto nekomunikuojant su pagrindiniu mazgu. Ši parinktis inicijuoja rolių pasikeitimą – yra reikalinga, kad dauguma pagrindinių mazgų patvirtintų rolių pasikeitimus ir būtų sugeneruota nauja *configEpoch* reikšmė naujam pagrindiniam mazgui. Šis tipas dažniausiai naudojamas, kai pagrindinis mazgas daugiau nėra pasiekiamas;
- ***Takeover***: rankinio persijungimo metu nėra atsižvelgiama į daugumą pagrindinių mazgų ir iš karto yra įvykdomas rankinis persijungimas. Šios parinkties metu yra sugeneruojama nauja *configEpoch* reikšmė, atsižvelgiant į žinomą didžiausią reikšmę blokinyje, ir pavaldžiam mazgui yra prisiskiriami pagrindinio mazgo maišos lizdai.

Formalioje specifikacijoje *cluster failover* komanda yra išreikšta FailOverCommand veiksme. Šis veiksmas gali įvykti tik tada, kai einamasis mazgas yra pavaldus mazgas. Šio veiksmo metu yra galimos trys baigtys: 1) nenaudojamos *force* ir *takeover* parinktys; 2) naudojama *force* parinktis; ir 3) naudojama *takeover* parinktis.

4.2.7. Modelio tikrinimo optimizavimas

Atliekant pirminės žinučių apdorojimo specifikacijos modelio tikrinimą, buvo pastebėta, kad yra susiduriama su būsenų sprogimo problema. TLC modelio tikrintojas sugeneruodavo labai daug būsenų ir būsenų mašinos grafo diametras kildavo labai lėtai. Norint pagreitinti modelio tikrinimą buvo atlikta pirminės formalios specifikacijos peržiūra, siekiant išsiaiškinti, kurie specifikacijos veiksmai gali daryti įtaką dideliam būsenų skaičiui ir kurios specifikacijos dalys užtrunka daugiausiai laiko. Peržiūros metu buvo rasti tokie veiksniai, lemiantys greitai pasireiškiančią būsenų sprogimo problemą, ir jie buvo pašalinti atlikus veiksmus:

- Atlikus peržiūrą buvo rasta specifikacijos klaidų, kurios leido specifikacijos veiksmams visada įvykti;
- Naudojantis TLA$^+$ Toolbox profiliavimo (angl. *profiling*) funkcionalumu buvo identifikuotos formalios specifikacijos vietos, kurios yra kviečiamos daugiausiai kartų. Taip pat profiliavimo metu buvo vertinama, kurie veiksmai užtrunka ilgiausiai – buvo rasta, kad funkcijos operatorius DOMAIN, kuris grąžina funkcijos apibrėžimo sritį, pasak profiliavimo funkcionalumo, užtrukdavo daug laiko. Dėl to formalios specifikacijos optimizavimo metu operatorius DOMAIN buvo pakeistas į atitinkamas specifikacijos konstantas. Tokį pakeitimą buvo galima įgyvendinti dėl to, kad visų funkcijų, naudojamų formalios specifikacijos kintamuosiuose, apibrėžimo sritis niekada nekinta ir yra lygi kažkuriai iš specifikacijos konstantų;
- Buvo pakeista žinučių esančių komunikavimo kanale apdorojimo eiga: vietoje to, kad būtų apdorojama po vieną žinutę vieno būsenos mašinos žingsnio metu, buvo nuspręsta įgyvendinti pakeitimą, kad vienu būsenos mašinos žingsniu būtų apdorojamos visos žinutės esančios komunikavimo kanale;
- Kadangi komunikavimo kanalai specifikacijoje yra modeliuojami sekomis, tai norint sumažinti būsenų skaičių, buvo nuspręsta pridėti galimų būsenų apribojimą, kad kiekvienoje būsenoje žinučių skaičius, esantis skaitymo arba rašymo kanaluose tarp dviejų mazgų, neturi viršyti konstantoje MAX_BUFFER_LEN nurodytos reikšmės.

5. „Redis Cluster“ sistemos tyrimas

Iš ankstesniame 1.1.6 skyriuje identifikuotų ir suformuluotų savybių, kurios turi būti užtikrintos, kad „Redis Cluster“ sistema veiktų korektiškai, buvo išsirinkta tirti vieną svarbiausių sistemos savybių – už maišos lizdą gali būti atsakingas tik vienas pagrindinis mazgas ir jo pavaldūs mazgai. Šios savybės užtikrinimas buvo vertinamas žinučių apdorojimo formalios specifikacijos modelio tikrinimo metu.

Ši savybė yra svarbi, kadangi maišos lizduose yra saugomi klientų atsiųsti duomenys ir, susidarius situacijai, kai už vieną maišos lizdą yra atsakingi keletas mazgų, klientai gali prarasti anksčiau siųstus savo duomenis arba gali nežinoti, į kurį mazgą reikia kreiptis dėl duomenų patalpinimo. Taip pat svarbu, kad visuose mazguose informacija, kuris mazgas yra atsakingas už kurį maišos lizdą, sutaptų. Tai reikalinga dėl to, kad „Redis Cluster“ klientai gali kreiptis į bet kurį blokinį sudarantį mazgą bandydami gauti maišos lizde saugomas reikšmes. Apdorojant tokią užklausą mazgas, jeigu jis nėra atsakingas už tą maišos lizdą, klientui gali grąžinti *MOVED* klaidos pranešimą su nurodymu, į kurį mazgą reikia kreiptis dėl tame maišos lizde saugomos informacijos. Esant blogai informacijai apie mazgo maišos lizdus, „Redis Cluster“ klientai gali nežinoti su kuriuo blokinio mazgu jiems iš tikrųjų reikia komunikuoti.

5.1. Specifikacijos modelis

Savybių tyrimo metu buvo naudojamas modelis, kurį sudarė trys mazgai. Ši konfigūracija buvo pasirinkta dėl to, kad, pagal „Redis Cluster“ dokumentaciją, tai yra mažiausia konfigūracija, kuri gali sudaryti blokinį. Trijų mazgų modelyje specifikacijos konstantoms buvo priskirtos tokios reikšmės:

$$\begin{gathered}\mathrm{NODE} = \{101, 102, 303\}, \mathrm{SLOTS_COUNT} = 6,\\ \mathrm{CLUSTER_REQUIRE_FULL_COVERAGE} = \mathrm{TRUE},\\ \mathrm{MAX_BUFFER_LEN} = 3\end{gathered}$$

Šiame modelyje pradinė būsena buvo aprašyta veiksme $\mathrm{InitThreeMasterNodes}$. Norint pagreitinti „Redis Cluster“ savybių tikrinimą, veiksme $\mathrm{InitThreeMasterNodes}$ specifikacijos kintamiesiems buvo priskirtos tokios reikšmės lyg blokinys jau būtų sudarytas ir mazgams nereikia daryti papildomos komunikacijos norint jį sukurti. Tai buvo pasiekta specifikacijoms kintamiesiems priskiriant:

- $\mathrm{clusterState}$ – galimų maišos lizdų aibė ($[1, 6]$) yra dalinama į tris dalis kiekvienam pagrin-

diniam mazgui priskiriant po lygią dalį maišos lizdų. Kiekvienas mazgas iš aibės NODE žino apie kitus blokinio mazgus ir yra užmezgęs komunikacijos kanalą su tais mazgais;

- $\mathrm{eventLoop}$ – įvykių ciklas pradedamas nuo įvykių prieš vykdant skaitymą arba rašymą vykdymo, laikas atlikti blokinio suplanuotą darbą dar nėra atėjęs, nėra kanalų skaitymo arba rašymo įvykių;
- $\mathrm{sendBuffer}$ – siuntimo buferis iš kiekvieno mazgo į kiekvieną mazgą yra tuščias;
- $\mathrm{recieveBuffer}$ – gavimo buferis iš kiekvieno mazgo į kiekvieną mazgą yra tuščias.

$$
\begin{aligned}
\mathrm{Constraint} \triangleq\ & \forall\, \mathrm{n}, \mathrm{m} \in \mathrm{NODE} : \\
& \quad \land\ \mathrm{Len}(\mathrm{sendBuffer}[\mathrm{n}][\mathrm{m}]) \leq \mathrm{MAX_BUFFER_LEN} \\
& \quad \land\ \mathrm{Len}(\mathrm{recieveBuffer}[\mathrm{n}][\mathrm{m}]) \leq \mathrm{MAX_BUFFER_LEN}
\end{aligned}
$$

Taip pat, norint sumažinti galimų būsenų skaičių, buvo pridėtas papildomas būsenų ribojimas (angl. *constraint*) – kiekvieno mazgo siuntimo ir gavimo buferis, kuris yra specifikuotas kaip žinučių seka, negali turėti daugiau nei trijų žinučių. Šis ribojimas buvo tikrinamas generuojant sekančias būsenų mašinos būsenas.

Šalia specifikuotų „Redis Cluster“ sistemos savybių, taip pat kiekvieno būsenų mašinos žingsnio metu buvo tikrinimas tipų korektiškumo invariantas $\mathrm{TCTypeOK}$ ir buvo tikrinama, ar nėra pasiekta modeliuojamos sistemos aklavietė. Naudojantis TLC įrankiu apibrėžto modelio tikrinimas buvo atliekamas tol, kol buvo surasta, kad yra pažeidžiama specifikuota sistemos savybė. Išanalizavus ir ištaisius klaidą formalioje specifikacijos, toliau buvo tęsiamas pakeistos specifikacijos modelio tikrinimas. Galiausiai modelio tikrinimas buvo nutrauktas praėjus dvylikai valandų – išnagrinėjus tik dalį būsenų aibės ir neradus daugiau jokių savybės pažeidimų.

Sudarytos specifikacijos ir apibrėžto modelio tikrinimas buvo atliekamas nešiojamuoju kompiuteriu (4 branduoliai, 16 GB RAM ir SSD diskas) bei TLA^{+} Toolbox įrankiu (versija 1.7.1).

5.2. Maišos lizdų savybės invariantas

$$
\begin{aligned}
& \mathrm{SlotsInvariant} \triangleq \\
& \quad \textsc{let} \\
& \quad \mathrm{isNodeAvailableForClientsReadEvents}[\mathrm{node} \in \mathrm{NODE}] \triangleq \\
& \qquad \land\ \text{“MASTER”} \in \mathrm{clusterState}[\mathrm{node}].\mathrm{nodes}[\mathrm{node}].\mathrm{flags} \\
& \qquad \land\ \mathrm{clusterState}[\mathrm{node}].\mathrm{clusterState} = \text{“OK”} \\
& \qquad \land\ \mathrm{eventLoop}[\mathrm{node}].\mathrm{state} = \text{“PROCESSING_FILE_EVENTS”} \\
& \qquad \land\ \mathrm{eventLoop}[\mathrm{node}].\mathrm{currentEventBeingProcessed} = \mathrm{NULL_EventLink}
\end{aligned}
$$

$$
\begin{array}{l}
\textsc{in}\ \ \forall\, \text{slot} \in \text{Slots} : \\
\quad \forall\, \text{n}, \text{m} \in \text{NODE} : \\
\quad\quad \land\ \text{n} \neq \text{m} \\
\quad\quad \land\ \text{isNodeAvailableForClientsReadEvents}[\text{n}] \\
\quad\quad \land\ \text{isNodeAvailableForClientsReadEvents}[\text{m}] \\
\quad\quad \land\ \text{clusterState}[\text{n}].\text{migratingSlotsTo}[\text{slot}] = \text{NULL_NODE} \\
\quad\quad \land\ \text{clusterState}[\text{m}].\text{migratingSlotsTo}[\text{slot}] = \text{NULL_NODE} \\
\quad\quad \implies \text{clusterState}[\text{n}].\text{slots}[\text{slot}] = \text{clusterState}[\text{m}].\text{slots}[\text{slot}]
\end{array}
$$

Maišos lizdų savybės užtikrinimas formalioje specifikacijoje buvo aprašytas invariantu SlotsInvariant. Šiuo invariantu yra tikrinama, kad bet kuriuose dviejuose mazguose n ir m, kurie gali priimti kliento komandas:

- Yra pagrindiniai mazgai;
- Jų požiūriu blokinys yra veikiantis ($\text{clusterState}[\text{node}].\text{clusterState} = \text{"OK"}$);
- Dabar yra vykdomas žinučių skaitymo arba rašymo etapas ($\text{eventLoop}[\text{node}].\text{state} = \text{"PROCESSING_FILE_EVENTS"}$);
- Šiuo metu nėra apdorojama jokia žinutė ($\text{eventLoop}[\text{node}].\text{currentEventBeingProcessed} = \text{NULL_EventLink}$),

informacija apie mazgus ir jų maišos lizdų atsakomybes turi sutapti ($\text{clusterState}[\text{n}].\text{slots}[\text{slot}] = \text{clusterState}[\text{m}].\text{slots}[\text{slot}]$).

Tikrinant šią savybę yra atsižvelgiama tik į pagrindinius mazgus dėl to, kad, norint gauti tikslius duomenis, „Redis Cluster“ klientai turi kreiptis į pagrindinį mazgą. Taip pat, nėra tikrinami maišos lizdai, kurie dalyvauja maišos lizdų pertvarkyme ($\text{clusterState}[\text{node}].\text{migratingSlotsTo}[\text{slot}] = \text{NULL_NODE}$). Nors ir reali sistema gali aptarnauti migruojančius maišos lizdus naudojant komandą *asking* ir klaidos pranešimą $-\text{ASK}$, tačiau, norint supaprastinti invariantą, buvo atsižvelgta tik į atvejus, kada klientai gali naudoti komandas *get* ir *set* tam tikram maišos lizdui.

5.3. Rastos klaidos ir galimi sprendimo būdai

Naudojantis TLC modelio tikrintoju, buvo rasta atvejų, kai yra pažeidžiamas maišos lizdų invariantas. Rastos klaidos, jų priežastys ir pasiūlymai, kaip būtų galima jas ištaisyti pateikti žemiau.

1 klaida: Komanda *cluster setslot* pažeidžia maišos lizdų savybę

TLC modelio tikrintojas nustatė, kad yra pažeidžiamas maišos lizdų savybės invariantas, kai pradinė būsena yra $\mathrm{InitThreeMasterNodes}$ veiksmas ir kai yra atliekami veiksmai:

1. Mazgas, kurio ID lygus 101, įvykdo veiksmą $\mathrm{BeforeSleep}$;
2. Mazgui su ID 101 yra iškviečiamas veiksmas $\mathrm{SetSlotCommand}$, kurio metu yra apdorojama komanda *cluster setslot 1 node 202* (mazgui su ID 202 yra priskiriamas 1 maišos lizdas);
3. Mazgas, kurio ID lygus 101, įvykdo veiksmą $\mathrm{BeforeSleep}$.

Maišos lizdų savybės pažeidimo priežastis – komandos *cluster setslot* subkomandą *node* galima iškviesti nurodant mazgą, kuris nedalyvauja maišos lizdų pertvarkyme. Sistema leidžia atlikti tokius veiksmus, nes viena iš subkomandos *node* paskirčių – rankiniu būdu pakeičiant maišos lizdų atsakomybes paspartinti informacijos apie įvykusį maišos lizdų pertvarkymą pasklidimą po „Redis Cluster“ blokinį. Tačiau, kaip pavyko nustatyti, toks subkomandos veikimas pažeidžia maišos lizdų savybę.

Rasta klaida buvo atkartota realioje sistemoje. Buvo sukurtas „Redis Cluster“ blokinys, kurį sudaro trys pagrindiniai mazgai: mazgas A (prievadas (angl. *port*) 6001), mazgas B (prievadas 6002) ir mazgas C (prievadas 6003). Maišos lizdai mazgams buvo padalinti po lygias dalis. Turint tokią sistemos konfigūraciją ir naudojantis tekstine sąsaja (angl. *command line interface*) *redis-cli*, maišos lizdas 5061 buvo paskirtas mazgui B siunčiant subkomandą *node* mazgui A. Mazgui A apdorojus tokią komandą susidarė situacija, kad mazgo A požiūriu už 5061 maišos lizdą yra atsakingas mazgas B, o mazgo B požiūriu už šitą maišos lizdą yra atsakingas mazgas A. Esant tokiai situacijai per *redis-cli* nusiuntus komandą *get 5061* mazgui A, yra gaunamas begalinis ciklas, kadangi mazgas A atsako, kad klientui dėl šio maišos lizdo reikia kreiptis į mazgą B, ir *redis-cli* pabandžius nusiųsti *get* komandą mazgui B, jis atsako, kad klientui reikia kreiptis į mazgą A. Klaidos atkartojimas naudojantis *redis-cli* tekstine sąsaja pavaizduotas 4 pav. Komunikuojant papildomai buvo naudojamos komandos: *cluster keyslot* – apskaičiuoja kokiame maišos lizde turi būti saugomas parametruose perduotas raktas, *cluster nodes* – į konsolę yra išvedama einamojo mazgo informacija apie blokinio mazgus ir jų maišos lizdų atsakomybes.

Norint ištaisyti šią klaidą būtų galima uždrausti kviesti *node* subkomandą, kai einamasis mazgas arba mazgas perduotas subkomandos parametruose nedalyvauja maišos lizdų pertvarkyme. Pašalintą *node* subkomandos funkcionalumą – paspartinti paskalų informacijos sklaidą apie pasikeitusius maišos lizdus, būtų galima užtikrinti šią atsakomybę perkeliant iš kliento į serverio pusę. Išsiuntus subkomandą *node* tiksliniam mazgui, kuris importuoja maišos lizdą, komandos ap-

4 pav. Pirmos klaidos atkartojimas naudojant *redis-cli*

```
127.0.0.1:6001> CLUSTER NODES
450fbc07bed0fb75f4d4e4bc1f229a49d30d789f 127.0.0.1:6001@16001 myself,master - 0 1617639726000 1
    ↪ connected 0-5460
bfe957964cadce07771aefd31e65be901e23b785 127.0.0.1:6002@16002 master - 0 1617639726602 2 connected
    ↪ 5461-10922
6df90cf369078bba453c2be920a2da15b0e9fe4d 127.0.0.1:6003@16003 master - 0 1617639726602 3 connected
    ↪ 10923-16383
127.0.0.1:6002> CLUSTER NODES
450fbc07bed0fb75f4d4e4bc1f229a49d30d789f 127.0.0.1:6001@16001 master - 0 1617639732567 1 connected
    ↪ 0-5460
bfe957964cadce07771aefd31e65be901e23b785 127.0.0.1:6002@16002 myself,master - 0 1617639731000 2
    ↪ connected 5461-10922
6df90cf369078bba453c2be920a2da15b0e9fe4d 127.0.0.1:6003@16003 master - 0 1617639732567 3 connected
    ↪ 10923-16383
127.0.0.1:6003> CLUSTER NODES
450fbc07bed0fb75f4d4e4bc1f229a49d30d789f 127.0.0.1:6001@16001 master - 0 1617639740210 1 connected
    ↪ 0-5460
bfe957964cadce07771aefd31e65be901e23b785 127.0.0.1:6002@16002 master - 0 1617639740009 2 connected
    ↪ 5461-10922
6df90cf369078bba453c2be920a2da15b0e9fe4d 127.0.0.1:6003@16003 myself,master - 0 1617639739000 3
    ↪ connected 10923-16383

127.0.0.1:6001> CLUSTER KEYSLOT bar
(integer) 5061
127.0.0.1:6001> CLUSTER SETSLOT 5061 NODE bfe957964cadce07771aefd31e65be901e23b785
OK

127.0.0.1:6001> CLUSTER NODES
450fbc07bed0fb75f4d4e4bc1f229a49d30d789f 127.0.0.1:6001@16001 myself,master - 0 1617639830000 1
    ↪ connected 0-5060 5062-5460
bfe957964cadce07771aefd31e65be901e23b785 127.0.0.1:6002@16002 master - 0 1617639830182 2 connected
    ↪ 5061 5461-10922
6df90cf369078bba453c2be920a2da15b0e9fe4d 127.0.0.1:6003@16003 master - 0 1617639830182 3 connected
    ↪ 10923-16383
127.0.0.1:6002> CLUSTER NODES
450fbc07bed0fb75f4d4e4bc1f229a49d30d789f 127.0.0.1:6001@16001 master - 0 1617639840179 1 connected
    ↪ 0-5460
bfe957964cadce07771aefd31e65be901e23b785 127.0.0.1:6002@16002 myself,master - 0 1617639839000 2
    ↪ connected 5461-10922
6df90cf369078bba453c2be920a2da15b0e9fe4d 127.0.0.1:6003@16003 master - 0 1617639840179 3 connected
    ↪ 10923-16383
127.0.0.1:6003> CLUSTER NODES
450fbc07bed0fb75f4d4e4bc1f229a49d30d789f 127.0.0.1:6001@16001 master - 0 1617639845605 1 connected
    ↪ 0-5460
bfe957964cadce07771aefd31e65be901e23b785 127.0.0.1:6002@16002 master - 0 1617639845605 2 connected
    ↪ 5461-10922
6df90cf369078bba453c2be920a2da15b0e9fe4d 127.0.0.1:6003@16003 myself,master - 0 1617639845000 3
    ↪ connected 10923-16383

127.0.0.1:6001> GET bar
-> Redirected to slot [5061] located at 127.0.0.1:6002
-> Redirected to slot [5061] located at 127.0.0.1:6001
-> Redirected to slot [5061] located at 127.0.0.1:6002
-> Redirected to slot [5061] located at 127.0.0.1:6001
...
```

dorojimo metu tikslinis mazgas taip pat galėtų išsiųsti *ping* žinutes, su pasikeitusiomis maišos lizdų atsakomybėmis, visiems blokinio mazgams.

Perkėlus siūlomą pakeitimą į žinučių apdorojimo specifikaciją ir toliau vykdant modelio tikrinimą buvo pastebėta, kad subkomanda *node* vis tiek pažeidžia maišos lizdų invariantą. Kadangi informacija apie maišos lizdo migruojančią ir importuojančią būseną yra saugoma atitinkamai pirminiame ir tiksliniame mazge, tai subkomanda *node* leidžia nesilaikyti maišos lizdų pertvarkymo proceso veiksmų tvarkos. Tokia situacija yra atkartojama modelyje pradedant nuo pradinės būsenos (aprašytos veiksmu $\text{InitThreeMasterNodes}$) ir atliekant veiksmus:

1. Mazgas, kurio ID lygus 101, įvykdo veiksmą BeforeSleep;
2. Mazgui su ID 101 yra iškviečiamas veiksmas SetSlotCommand, kurio metu yra apdorojama komanda *cluster setslot 3 importing 202* (mazgas su ID 101 nustato maišos lizdo 3 būseną į importuojančią);
3. Mazgas, kurio ID lygus 101, įvykdo veiksmą BeforeSleep;
4. Mazgui su ID 101 yra iškviečiamas veiksmas SetSlotCommand, kurio metu yra apdorojama komanda *cluster setslot 3 node 101* (mazgui su ID 101 yra priskiriamas 3 maišos lizdas).

Maišos lizdų savybė tampa pažeista, kadangi tiek mazgas su ID 101 yra atsakingas už 3 maišos lizdą ir tiek mazgas su ID 202 irgi yra atsakingas už tą patį maišos lizdą.

Šią klaidą būtų galima ištaisyti priverčiant klientus laikytis maišos lizdų pertvarkymo proceso veiksmų tvarkos. Tai būtų galima užtikrinti pradedant dalintis informacija apie maišos lizdo migruojančią būseną su visais blokinio mazgais – papildant paskalų informaciją. Tada tikslinis mazgas galėtų pakeisti maišos lizdo būseną į importuojančią tik tada, kai iš pirminio mazgo gavo informaciją, kad jis pakeitė norimo maišos lizdo būseną į migruojančią. Taip pat reiktų užtikrinti, kad subkomandą *node* būtų galima iškviesti tik tada, kai einamasis mazgas importuoja komandoje nurodytą maišos lizdą ir kai mazgas nurodytas subkomandoje *node* yra einamasis mazgas. Toks pakeitimas reikalingas norint užtikrinti, kad už maišos lizdų pertvarkymo proceso užbaigimą būtų atsakingas tik tas mazgas, kuris importuoja maišos lizdą. Kitu atveju nesant šiam pakeitimui būtų galima pasiekti situaciją, kai pirminis mazgas pakeičia maišos lizdo būseną į migruojančią ir iš karto užbaigią maišos lizdo pertvarkymo procesą gaudamas subkomandą *node*.

Įvykdžius visus siūlomus pakeitimus, maišos lizdo pertvarkymo procesas būtų vykdomas siunčiant komandas:

1. ***Cluster setslot <maišos lizdas> migrating <tikslinis mazgas>*** – ši komanda pakeistų maišos lizdo būseną į migruojančią ir ši informacija būtų pasidalinta su kitais mazgais paskalų

protokolo pagalba;

2. ***Cluster setslot <maišos lizdas> importing <pirminis mazgas>*** – po pakeitimų šią komandą būtų galima iškviesti tik tada, kai buvo gauta paskalų informacija, kad maišos lizdas yra migruojančioje būsenoje;

3. ***Migrate*** – duomenys yra perkeliami iš pirminio mazgo į tikslinį mazgą;

4. ***Cluster setslot <maišos lizdas> node <tikslinis mazgas>*** – šią komandą būtų galima išsiųsti tik mazgui, kuris importuoją maišos lizdą. Taip būtų užbaigtas maišos lizdo pertvarkymo procesas, o pirminis mazgas išvalytų maišos lizdo migruojančią būseną tik tada, kai gautų paskalų informaciją, kad tikslinis mazgas (arba tikslinio mazgo pavaldus mazgas) tapo atsakingas už migruojantį maišos lizdą.

Siūlomi pakeitimai buvo įgyvendinti žinučių apdorojimo specifikacijos CreateMessage, ClusterUpdateSlotsConfigWith, ProcessPingPongMeetMsg SetSlotNodeBug1Fix ir SetSlotImportingWithBug1Fix veiksmuose.

2 klaida: Komanda *cluster setslot <maišos lizdas> migrating <tikslinis mazgas>* pažeidžia maišos lizdų savybę

Tiriant 1 klaidos pasireiškimo aplinkybes, buvo rasta, kad yra pažeidžiamas maišos lizdų invariantas, kai komandos *cluster migrating* parametruose perduodamas tikslinis mazgas sutampa su einamuoju mazgu, kuriam yra siunčiama administravimo komanda.

Klaida buvo atkartota realioje sistemoje, kurią sudaro trys mazgai, kurie maišos lizdus yra pasidalinę lygiomis dalimis. Mazgui A išsiuntus komandą *cluster setslot 1 migrating <mazgas a>*, pasikeitė komandoje nurodyto maišos lizdo būsena į migruojančią į mazgą A. Klaidos atkartojimą naudojantis tekstine sąsaja *redis-cli* galima pamatyti 5 pav. Eilutė (*1->-2d2e1bc3c17654a516e11f5d6ab07c27ab6065cf*) nurodo, kad maišos lizdas 1 migruoją į mazgą A.

Šią klaidą galima ištaisyti pridedant papildomus patikrinimus, kad tikslinis mazgas nurodytas subkomandoje *migrating* nėra dabartinis mazgas. Siūlomi pakeitimai buvo įgyvendinti žinučių apdorojimo specifikacijos veiksme SetSlotMigratingWithBug2Fix.

5 pav. Antros klaidos atkartojimas naudojant *redis-cli*

```
127.0.0.1:6001> CLUSTER NODES
2d2e1bc3c17654a516e11f5d6ab07c27ab6065cf 127.0.0.1:6001@16001 myself,master - 0 1621800674000 1
    ↪ connected 0-5460
0b7c316db07673f9b4a710a3a8247aa8d963405f 127.0.0.1:6002@16002 master - 0 1621800675227 2 connected
    ↪ 5461-10922
b8aba4ac38ab9d82132b4896c84b8f128f3a62e8 127.0.0.1:6003@16003 master - 0 1621800675227 3 connected
    ↪ 10923-16383

127.0.0.1:6001> CLUSTER SETSLOT 1 MIGRATING 2d2e1bc3c17654a516e11f5d6ab07c27ab6065cf
OK

127.0.0.1:6001> CLUSTER NODES
2d2e1bc3c17654a516e11f5d6ab07c27ab6065cf 127.0.0.1:6001@16001 myself,master - 0 1621800693000 1
    ↪ connected 0-5460 [1->-2d2e1bc3c17654a516e11f5d6ab07c27ab6065cf]
0b7c316db07673f9b4a710a3a8247aa8d963405f 127.0.0.1:6002@16002 master - 0 1621800694360 2 connected
    ↪ 5461-10922
b8aba4ac38ab9d82132b4896c84b8f128f3a62e8 127.0.0.1:6003@16003 master - 0 1621800694360 3 connected
    ↪ 10923-16383
```

3 klaida: Komanda *cluster setslot <maišos lizdas> importing <pirminis mazgas>* pažeidžia maišos lizdų savybę

Atliekant modelio tikrinimą dalinai pakeistos formalios specifikacijos, buvo rasta, kad yra pažeidžiamas maišos lizdų invariantas, kai komandos *cluster importing* parametruose perduodamas pirminis mazgas sutampa su einamuoju mazgu.

Klaida buvo atkartota realioje sistemoje, kurią sudaro trys mazgai, kurie maišos lizdus yra pasidalinę lygiomis dalimis. Išsiuntus komandą *cluster setslot 5461 importing <mazgas a>*, komandoje nurodyto maišos lizdo būsena pasikeitė į importuojančią į mazgą A. Klaidos atkartojimą naudojantis tekstine sąsaja *redis-cli* galima pamatyti 6 pav. Eilutė (*5461-<-ea6f59852f7a0365cd0887033483e1574677d675*) nurodo, kad maišos lizdas 5461 yra importuojamas iš dabartinio mazgo A.

Šią klaidą galima ištaisyti pridedant papildomus patikrinimus, kad pirminis mazgas nurodytas subkomandoje *importing* nėra dabartinis mazgas. Siūlomi pakeitimai buvo įgyvendinti žinučių apdorojimo specifikacijos veiksme SetSlotImportingWithBug3Fix.

4 klaida: Komanda *cluster setslot <maišos lizdas> importing <pirminis mazgas>* pažeidžia maišos lizdų savybę

Tolimesnio modelio tikrinimo metu buvo rasta, kad yra pažeidžiamas maišos lizdų invariantas, kadangi komandos *cluster importing* parametruose galima perduoti maišos lizdą, už kurį nėra atsakingas komandoje nurodytas pirminis mazgas.

Klaidos atkartojimą realioje sistemoje galima pamatyti, kai buvo atkartojama trečioji klaida

6 pav. Trečios klaidos atkartojimas naudojant *redis-cli*

```
127.0.0.1:6001> CLUSTER NODES
ea6f59852f7a0365cd0887033483e1574677d675 127.0.0.1:6001@16001 myself,master – 0 1617652573000 1
    ↪ connected 0–5460
d807f0e115998ea13975cbe7ab9d076cc2f21459 127.0.0.1:6002@16002 master – 0 1617652574217 2 connected
    ↪ 5461–10922
2343a407de3d75e4a4beeb57f8983cf5346a8e6d 127.0.0.1:6003@16003 master – 0 1617652574217 3 connected
    ↪ 10923–16383

127.0.0.1:6001> CLUSTER SETSLOT 5461 IMPORTING ea6f59852f7a0365cd0887033483e1574677d675
OK

127.0.0.1:6001> CLUSTER NODES
ea6f59852f7a0365cd0887033483e1574677d675 127.0.0.1:6001@16001 myself,master – 0 1617652672000 1
    ↪ connected 0–5460 [5461-<-ea6f59852f7a0365cd0887033483e1574677d675]
d807f0e115998ea13975cbe7ab9d076cc2f21459 127.0.0.1:6002@16002 master – 0 1617652673777 2 connected
    ↪ 5461–10922
2343a407de3d75e4a4beeb57f8983cf5346a8e6d 127.0.0.1:6003@16003 master – 0 1617652673777 3 connected
    ↪ 10923–16383
```

(6 pav). Sistema priėmė komandą *cluster setslot 5461 importing <mazgas A>* nepaisant to, kad mazgas A iš tikrųjų nėra atsakingas už maišos lizdą 5461. Ši klaida gali būti ištaisyta pridedant papildomą patikrinimą, kad pirminis mazgas nurodytas subkomandoje *importing* iš tikrųjų yra atsakingas už parametruose perduotą maišos lizdą. Toks papildomas patikrinimas buvo įgyvendintas žinučių apdorojimo specifikacijos veiksme SetSlotImportingWithBug4Fix.

5 klaida: Komandos *cluster addslots* ir *cluster delslots <maišos lizdai>* pažeidžia maišos lizdų savybę

Modelio tikrinimo metu buvo rasta, kad komandų *cluster addslots* ir *cluster delslots* naudojimas pažeidžia maišos lizdų invariantą. Einamajame mazge apdorojus šias komandas, maišos lizdų pasikeitimas nepasiekė kitų mazgų.

Deja, bet šios klaidos nepavyko pilnai atkartoti realioje sistemoje – įvykus maišos lizdų pasikeitimams, dėl paskalų protokolo visada buvo sugrįžtama į pradinę būseną. Taip atsitiko dėl to, kad kitų mazgų požiūriu neįvyko jokie maišos lizdų pasikeitimai, ir einamojo mazgo *configEpoch* reikšmė buvo mažesnė ir nepakito – kitų mazgų, kurių *configEpoch* reikšmė buvo didesnė, paskalų informacija laimėjo.

Norint ištaisyti šią klaidą būtų galima įvykus maišos lizdų pasikeitimams padidinti einamojo mazgo *configEpoch* reikšmę, kad kiti mazgai priimtų šią informaciją kaip naujausią. Kitas būdas taisyti šią klaidą: užtikrinti, kad šias komandas būtų galima apdoroti tik tada, kai sistema yra reikalingoje būsenoje, kurios metu apdorojus *addslots* ir *delslots* komandas nebūtų pažeista maišos lizdų savybė.

Ši sistemos klaida nėra kritinė, kadangi „Redis Cluster“ dokumentacijoje yra nurodyta, kad šią komandą patartina siųsti tik tada, kai yra kuriamas naujas blokinys arba kai bandoma ištaisyti nenumatytas klaidas. Patys kūrėjai nurodo, kad šios komandos kvietimas nereikiamu metu gali palikti blokinį klaidingoje būsenoje, kaip ir buvo rasta modelio tikrinimo metu. Dėl to buvo nuspręsta šios klaidos netaisyti ir tolimesniame modelio tikrinime nebenaudoti veiksmų AddSlotsCommand ir DelSlotsCommand. Taip pat *cluster delslots* komanda nėra plačiai naudojama ir ji sistemoje buvo pridėta tik norint turėti pilnai užpildytą API sąsają.

6 klaida: Komanda *cluster flushslots* pažeidžia maišos lizdų savybę

Atliekant modelio tikrinimą taip pat buvo rasta, kad komanda *cluster flushslots* pažeidžia maišos lizdų savybę. Apdorojus komandą ir pašalinus visas mazgo maišos lizdų atsakomybes buvo gauta situacija, kad einamojo mazgo požiūriu už tuos maišos lizdus nėra atsakingas nei vienas mazgas, tačiau kitų mazgų požiūriu, už šiuos maišos lizdus vis dar yra atsakingas einamasis mazgas.

Rasta klaida buvo atkartota realioje sistemoje naudojantis *redis-cli* (7 pav.). Mazgui C išsiuntus komandą *cluster flushslots*, buvo gauta situacija, kad mazgo C požiūriu jis nėra atsakingas už nei vieną maišos lizdą, tačiau kitų mazgų požiūriu neįvyko jokie pasikeitimai ir mazgas C vis dar yra atsakingas už savo maišos lizdus.

Kadangi komanda *flushslots* yra atskiras *delslots* komandos atvejis (*flushslots* pašalina konkrečią aibę maišos lizdų), tai šią komandą būtų galima ištaisyti taip pat kaip ir *delslots* komandą. Deja, tačiau skirtingai nei *delslots* komandos atveju, dokumentacijoje nėra nurodoma, kada šią komandą yra patartina kviesti. Vykdant tolimesnį modelio tikrinimą buvo nuspręsta šios klaidos taip pat netaisyti – modelio tikrinimas buvo vykdomas pašalinus veiksmą FlushSlotsCommand.

7 pav. Šeštos klaidos atkartojimas naudojant *redis-cli*

```
127.0.0.1:6003> CLUSTER NODES
81d870a15ad06772ba13e0272ba000efb48ec99d 127.0.0.1:6001@16001 master - 0 1620404862328 1 connected
    ↪ 0-5460
da628276ca1761927ce3f5131608be6a27209cd4 127.0.0.1:6002@16002 master - 0 1620404862328 2 connected
    ↪ 5461-10922
3d0de3a926748019359f29821eb6e68503b74d68 127.0.0.1:6003@16003 myself,master - 0 1620404861000 3
    ↪ connected 10923-16383

127.0.0.1:6003> CLUSTER FLUSHSLOTS
OK

127.0.0.1:6003> CLUSTER NODES
81d870a15ad06772ba13e0272ba000efb48ec99d 127.0.0.1:6001@16001 master - 0 1620404890484 1 connected
    ↪ 0-5460
da628276ca1761927ce3f5131608be6a27209cd4 127.0.0.1:6002@16002 master - 0 1620404890484 2 connected
    ↪ 5461-10922
3d0de3a926748019359f29821eb6e68503b74d68 127.0.0.1:6003@16003 myself,master - 0 1620404889000 3
    ↪ connected
127.0.0.1:6002> CLUSTER NODES
81d870a15ad06772ba13e0272ba000efb48ec99d 127.0.0.1:6001@16001 master - 0 1620404895511 1 connected
    ↪ 0-5460
da628276ca1761927ce3f5131608be6a27209cd4 127.0.0.1:6002@16002 myself,master - 0 1620404894000 2
    ↪ connected 5461-10922
3d0de3a926748019359f29821eb6e68503b74d68 127.0.0.1:6003@16003 master - 0 1620404895511 3 connected
    ↪ 10923-16383
127.0.0.1:6001> CLUSTER NODES
81d870a15ad06772ba13e0272ba000efb48ec99d 127.0.0.1:6001@16001 myself,master - 0 1620404907000 1
    ↪ connected 0-5460
da628276ca1761927ce3f5131608be6a27209cd4 127.0.0.1:6002@16002 master - 0 1620404907836 2 connected
    ↪ 5461-10922
3d0de3a926748019359f29821eb6e68503b74d68 127.0.0.1:6003@16003 master - 0 1620404907836 3 connected
    ↪ 10923-16383
```

7 klaida: Komanda *cluster forget* pažeidžia maišos lizdų savybę

Vykdant modelio tikrinimą buvo rasta, kad įvykdžius komandą *cluster forget* yra pažeidžiamas maišos lizdų invariantas. Einamajame mazge apdorojus komandą ir pašalinus komandoje nurodytą pagrindinį mazgą ir taip pat pašalinus su tuo mazgu susijusias maišos lizdų atsakomybes, buvo gauta situacija, kad einamojo mazgo požiūriu už tuos maišos lizdus, kurie anksčiau buvo priskirti pašalintam mazgui, nėra atsakingas nei vienas mazgas, tačiau kitų mazgų požiūriu, už šiuos maišos lizdus yra atsakingas mazgas, kuris buvo pašalintas.

Klaida realioje sistemoje, sudarytoje iš trijų pagrindinių mazgų, buvo atkartota naudojantis *redis-cli* (8 pav.). Nusiuntus komandą *cluster forget <mazgas A>* mazgui B, mazgas A buvo pašalintas iš mazgo B žinomų mazgų aibės. Taip pat, vykdant mazgo pašalinimą, buvo ištrinta informacija apie maišos lizdus susijusius su mazgu A. Kitų mazgų A ir C požiūriu neįvyko jokie pasikeitimai – blokinį sudaro trys mazgai ir mazgas A vis dar yra atsakingas už jam priskirtus maišos lizdus.

8 pav. Septintos klaidos atkartojimas naudojant *redis-cli*

```
127.0.0.1:6002> CLUSTER NODES
fbb65420e0d310affecf0a0d177ffbe5e7e2911e 127.0.0.1:6001@16001 master - 0 1620983232352 1 connected
    ↪ 0-5460
c503d9bb3e3146a40e2ae8546ef0c7e2535481e0 127.0.0.1:6002@16002 myself,master - 0 1620983232000 2
    ↪ connected 5461-10922
270b37e7370027be3f883ced3d2cec1f8db98086 127.0.0.1:6003@16003 master - 0 1620983232352 3 connected
    ↪ 10923-16383

127.0.0.1:6002> CLUSTER FORGET fbb65420e0d310affecf0a0d177ffbe5e7e2911e
OK

127.0.0.1:6002> CLUSTER NODES
c503d9bb3e3146a40e2ae8546ef0c7e2535481e0 127.0.0.1:6002@16002 myself,master - 0 1620986590000 2
    ↪ connected 5461-10922
270b37e7370027be3f883ced3d2cec1f8db98086 127.0.0.1:6003@16003 master - 0 1620986591096 3 connected
    ↪ 10923-16383
127.0.0.1:6001> CLUSTER NODES
fbb65420e0d310affecf0a0d177ffbe5e7e2911e 127.0.0.1:6001@16001 myself,master - 0 1620986602000 1
    ↪ connected 0-5460
c503d9bb3e3146a40e2ae8546ef0c7e2535481e0 127.0.0.1:6002@16002 master - 0 1620986604005 2 connected
    ↪ 5461-10922
270b37e7370027be3f883ced3d2cec1f8db98086 127.0.0.1:6003@16003 master - 0 1620986604005 3 connected
    ↪ 10923-16383
127.0.0.1:6003> CLUSTER NODES
fbb65420e0d310affecf0a0d177ffbe5e7e2911e 127.0.0.1:6001@16001 master - 0 1620986596854 1 connected
    ↪ 0-5460
c503d9bb3e3146a40e2ae8546ef0c7e2535481e0 127.0.0.1:6002@16002 master - 0 1620986596855 2 connected
    ↪ 5461-10922
270b37e7370027be3f883ced3d2cec1f8db98086 127.0.0.1:6003@16003 myself,master - 0 1620986595000 3
    ↪ connected 10923-16383
```

Palikus sistemą tokioje būsenoje ir po tam tikro laiko nusiuntus komandą *cluster nodes* mazgui B, mazgui žinomų mazgų sąraše vėl atsirado mazgas A, kuris turėjo jam ankščiau priskirtas maišos lizdų atsakomybes. Taip atsitiko dėl to, kad mazgui B pašalinus mazgą A, mazgas A atsidūrė juodajame sąraše mazgų, kurie neturi būti pridėti atgal prie blokinio. Šio juodojo sąrašo paskirtis – apdorojus paskalų informaciją, prie žinomų mazgų sąrašo nepridėti mazgų esančių juodajame sąraše, kadangi tie mazgai ankščiau buvo pašalinti iš blokinio. Tačiau mazgai šiame juodajame sąraše yra laikomi tam tikrą laiko tarpą, kuriam praėjus mazgai vėl gali būti pridėti prie blokinio, apdorojus kitų mazgų paskalų informaciją. Būtent dėl to mazgas A vėl atsirado mazgui B žinomų mazgų sąraše, nes kito žinomo mazgo C paskalų informacijoje vis dar egzistavo mazgas A.

Rastą klaidą galima laikyti nekritine, nes „Redis Cluster“ dokumentacijoje nurodoma, kad, norint pilnai pašalinti mazgą, *cluster forget* komandą reikia nusiųsti visiems likusiems blokinio mazgams. Taip pat, kaip buvo pastebėta atkartojant klaidą realioje sistemoje, neišsiuntus komandos visiems likusiems blokinio mazgas, po tam tikro laiko blokinys sugrįžo į pradinę būseną.

Šią klaidą būtų galima ištaisyti neleidžiant apdoroti komandos *cluster forget*, jei mazgas, kurį

norima pašalinti, vis dar yra atsakingas už bent vieną maišos lizdą. Kitas būdas ištaisyti šią klaidą – pasidalinti informacija apie pašalintą mazgą su kitais blokinio mazgais. Tada komandą *cluster forget* užtektų iškviesti vieną kartą, nes mazgui, apdorojus komandą ir pašalinus mazgą iš žinomų mazgų sąrašo, informacija apie pašalintą mazgą pasiektų ir kitus blokinio mazgus. Kadangi ši klaida nėra kritinė, tai ši klaida specifikacijoje nebuvo ištaisyta ir tolimesnis modelio tikrinimas buvo atliekamas pašalinus veiksmą ForgetCommand.

5.4. Formalių specifikacijų susiejimas

Sudarytos dvi „Redis Cluster“ sistemos formalias specifikacijas yra tarpusavyje susijusios. Išorinių veiksmų specifikacija yra abstrakti – aprašo sistemos funkcionalumą (reikalavimus) ir sistemos būseną, kuri yra matoma sistemos naudotojui. Tuo tarpu žinučių apdorojimo specifikacija yra detalesnė – remiantis sistemos kodu aprašo algoritmą, kuris įgyvendina sistemos funkcionalumą. Abstrakčią ir detalesnę „Redis Cluster“ sistemos formalias specifikacijas galima susieti tarp jų apibrėžiant tobulinimo (angl. *refinement*) ryšį. Atliktas tobulinimas leis įsitikinti, kad detali žinučių apdorojimo specifikacija atitinka sistemai keliamus reikalavimus, aprašytus išorinių veiksmų specifikacijoje.

```
Abstract ≜ INSTANCE RedisCluster_abstract WITH
    NODE ← NODE,
    SLOTS ← Slots,
    nodeFailed ←
        [n ∈ NODE ↦ ∀ m ∈ NODE :
                        ∧ clusterState[m].nodes[n] ≠ NULL_ClusterNodeType
                        ∧ "FAIL" ∈ clusterState[m].nodes[n].flags],
    slaveOf ←
        [n ∈ NODE ↦ IF clusterState[n].nodes[n].slaveOf = NULL_NODE
                    THEN {}
                    ELSE {clusterState[n].nodes[n].slaveOf}],
    nodeSlaves ←
        [n ∈ NODE ↦ {m ∈ NODE : clusterState[m].nodes[m].slaveOf = n}],
    clusterSlots ←
        [s ∈ Slots ↦ {m ∈ ({clusterState[n].slots[s] : n ∈ NODE} \ {NULL_NODE}) :
                        clusterState[m].slots[s] = m}],
    clusterKnownNodes ←
        {n ∈ NODE : ∃ m ∈ NODE : clusterState[m].nodes[n] ≠ NULL_ClusterNodeType},
    clusterState ←
        IF ∀ n ∈ NODE : clusterState[n].clusterState = "FAIL"
```

THEN "FAIL"
ELSE "OK"

$\text{THEOREM}\ Refinement \triangleq TCSpec \implies Abstract!TCSpec$

Abstrakti ir detali „Redis Cluster“ sistemos specifikacijos buvo susietos žinučių apdorojimo specifikacijoje aprašant tobulinimo atvaizdį $Abstract$. Atvaizdį sudaro taisyklės:

- Detalios ir abstrakčios specifikacijų konstantos $NODE$, nusakančias mazgų aibę, sutampa;
- Detalios ir abstrakčios specifikacijų maišos lizdų aibės sutampa;
- Abstrakčios specifikacijos $nodeFailed$ kintamasis yra išreikštas funkcija, kur mazgas yra neaktyvus tada, kai visi blokinio mazgai sutarė, kad blokiniui žinomas mazgas tapo neaktyviu (”FAIL” $\in clusterState[m].nodes[n].flags$);
- Kintamasis $slaveOf$ abstrakčioje specifikacijoje nusakantis kiekvieno mazgo pagrindinį mazgą yra išreikštas funkcija, kurioje kiekvieno mazgo pagrindinis mazgas yra nustatomas pagal $clusterState[n].nodes[n].slaveOf$ reikšmę: jei ji yra $NULL_NODE$, tai reiškia, kad mazgas yra pagrindinis mazgas ir abstrakčioje specifikacijoje reikšmė yra tuščia aibe, o kitu atveju, kai reikšmė nėra lygi $NULL_NODE$, mazgas yra pavaldus mazgas;
- Kintamasis $nodeSlaves$ naudojamas abstrakčioje specifikacijoje yra išreikštas funkcija, kurioje kiekvieno mazgo pavaldūs mazgai yra nustatomi pagal tai, kas mazgų požiūriu yra jų pagrindinis mazgas ($clusterState[m].nodes[m].slaveOf = n$);
- Maišos lizdų atsakomybių kintamasis $clusterSlots$ buvo išreikštas funkcija, kur kiekvienam maišos lizdui iš pradžių yra surenkama aibė mazgų, kurie pagal kiekvieno blokinio mazgo požiūrį yra atsakingi už tą maišos lizdą. Tada sudaryta aibė yra išfiltruojama pagal tai, ar kiekvienas iš aibės mazgų pagal jų požiūrį iš tiesų yra atsakingi už tą maišos lizdą;
- Žinomų mazgų aibė laikoma abstrakčios specifikacijos $clusterKnownNodes$ kintamajame yra išreikšta aibe, kuri yra sudaroma einant per visus galimus mazgus ir žiūrint ar kiekvieno mazgo požiūriu jis žino tą mazgą ($clusterState[m].nodes[n] \neq NULL_ClusterNodeType$);
- Abstrakčios specifikacijos kintamojo $clusterState$ reikšmė priklauso nuo to, ar detalioje specifikacijoje modeliuojamame blokinyje kiekvieno mazgo požiūriu blokinys yra blogoje būsenoje.

Aprašius taisykles TLC įrankiu buvo atliktas žinučių apdorojimo specifikacijos modelio tikrinimas formulę $Abstract!TLCSpec$ tikrinant kaip gyvybingumo savybę. Taip pat norint atlikti

modelio tikrinimą teko pakeisti abstrakčios specifikacijos pradinę būseną (pridėtas naujas pradinės būsenos veiksmas ThreeMasterNodesInit), kad tiek detalios, tiek abstrakčios specifikacijų pradinės būsenos sutaptų.

Iš pradžių buvo bandoma tikrinti formalią žinučių apdorojimo specifikaciją, kurioje nėra pataisytos klaidos (1, 2, 3 ir 4 klaidos) ir iš specifikacijos nėra pašalinti veiksmai susiję su 5, 6 ir 7 klaidomis. Tokio tikrinimo metu per pirmąsias tikrinimo minutes buvo nustatyta, kad yra pažeidžiama gyvybingumo savybė ir aprašytos specifikacijos tobulinimo atvaizdžio taisyklės yra neteisingos. Taip atsitiko dėl to, kad abstrakčioje specifikacijoje yra modeliuojamas sistemos veikimas be jokios komunikacijos ir pasikeitimai yra matomi visiems blokinį sudarantiems mazgas (pavyzdžiui, kintamasis clusterSlots yra bendras ir rodo viso blokinio, o ne vieno mazgo būseną). Toks abstrakčioje specifikacijoje modeliuojamas sistemos veikimas atitinka apibrėžtą maišos lizdų savybės invariantą, todėl ir specifikacijos tobulinimo tikrinimo metu buvo nustatyta, kad yra pažeidžiama gyvybingumo savybė – sistemoje esant šioms klaidoms nėra užtikrinama, kad informacija apie maišos lizdus visuose blokinio mazguose sutaps.

Pakeistos žinučių specifikacijos su ištaisytomis klaidomis tobulinimo atvaizdžio teisingumo tikrinimas buvo toliau atliekamas nešiojamuoju kompiuteriu (4 branduoliai, 16 GB RAM ir SSD diskas) bei TLA$^+$ Toolbox įrankiu (versija 1.7.1). Modelio tikrinimas, išnagrinėjus tik dalį būsenų aibės, buvo nutrauktas praėjus dvylikai valandų ir neradus jokių gyvybingumo savybės pažeidimų.

Rezultatai ir išvados

Atlikus literatūros apžvalgą ir remiantis „Redis Cluster“ sistemos dokumentacija, sistemos specifikacija bei literatūros analizės metu rastais šaltiniais, buvo identifikuotos ir suformuluotos savybės, kuriomis pasižymi išskirstyta sistema „Redis Cluster“. Šiame sąraše identifikuota svarbi sistemos savybė, susijusi su maišos lizdais, buvo toliau tiriama taikant formalius metodus.

Nagrinėjant „Redis Cluster“ sistemos aprašymą ir atvirai prieinamą sistemos kodą, TLA$^+$ formalia specifikavimo kalba buvo sudarytos dvi formalios specifikacijos. Iš pradžių sistema buvo specifikuota abstrakčiai apibrėžiant išorinius veiksmus ir akcentuojant sistemos funkcionalumą. Vėliau buvo sudaryta detalesnė žinučių apdorojimo specifikacija, artima programinio kodo abstrakcijos lygiui. Abi specifikacijos buvo susietos apibrėžiant tobulinimo atvaizdį, kurio teisingumas buvo patikrintas atlikus modelio tikrinimą.

TLA$^+$ formalia specifikavimo kalba sudarius „Redis Cluster“ formalią specifikaciją ir atlikus modelio tikrinimą, buvo nustatyta, kad tam tikrose situacijose sistema gali veikti nekorektiškai – yra pažeidžiama sistemos maišos lizdų savybė, kad už vieną maišos lizdą yra atsakingas tik vienas pagrindinis mazgas ir jo pavaldūs mazgai. Modelio tikrinimo metu identifikuotos sistemos klaidos buvo atkartotos realioje sistemoje naudojant tekstinę sąsają *redis-cli*. Šioms klaidoms buvo pateikti galimi sprendimo būdai, kaip jas būtų galima ištaisyti.

Sudaryta „Redis Cluster“ formali specifikacija tinkamai ir pilnai aprašo nagrinėjamus sistemos aspektus. „Redis Cluster“ sistemos tyrimo metu naudota detali specifikacija yra adekvati, nes: 1) sudarant specifikaciją buvo remtasi sistemos kodu ir buvo gauta specifikacija artima programinio kodo abstrakcijos lygiui; 2) modelio tikrinimo metu rastos klaidos buvo atkartotos realioje sistemoje, kas leidžia įsitikinti, kad specifikacijoje ir realioje sistemoje yra galimi tokie patys veiksmai ir yra gaunami tokie patys rezultatai (sistemos būsena); ir 3) buvo apibrėžtas tobulinimo atvaizdis, kuris susieja detalią specifikaciją su abstrakčia specifikacija, kurioje aukštesniame abstrakcijos lygyje buvo apibrėžti sistemai keliami reikalavimai.

Iš gautų rezultatų daromos šios **išvados**:

1. Taikant formalius metodus pavyko nustatyti, kad tam tikrose situacijose „Redis Cluster“ sistemoje nėra užtikrinama maišos lizdų savybė ir sistema veikia nekorektiškai.

2. Detali sistemos specifikacija, artima programinio kodo abstrakcijos lygiui, leido surasti klaidas susijusias su blogais komandos parametrais. Rašant specifikaciją tik pagal sistemos dokumentaciją, šios klaidos galėjo būti nepastebėtos, kadangi nesiremiant programiniu kodu, būtų daroma prielaida, kad sistemoje egzistuoja korektiški komandos parametrų patikrinimai.

3. Dėl specifikacijos detalumo, atliekant modelio tikrinimą, buvo susidurta su būsenų sprogimo problema, kas neleido patikrinti visos modeliuojamos sistemos galimų būsenų aibės. Nepaisant to, atliktas modelio tikrinimas leido identifikuoti sistemos klaidas, kurias TLC įrankis aptiko per pirmąsias modelio tikrinimo minutes.
4. Turint sudarytą „Redis Cluster" formalią specifikaciją bus galima toliau tirti ir kitas „Redis Cluster" sistemos savybes, bei bus galima patikrinti, ar atlikus sistemos pakeitimus yra išlaikomas algoritmo korektiškumas ir nėra pažeidžiamos sistemos savybės.

Šaltiniai

[AB86] J. Archibald and J. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4:273–298, 1986.

[Aba12] D. Abadi. Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story. *Computer*, 45:37–42, 2012.

[AMI17] N. A. Ali, A. A. Mirghani, and A. Y. Ibrahim. Alneelain: A formal specification language. In *2017 International Conference on Communication, Control, Computing and Electronics Engineering (ICCCCEE)*, pp. 1–9, 2017.

[AXF+12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-scale Key-value Store. *ACM SIGMETRICS Performance Evaluation Review*, 40:53–64, 2012.

[BMP18] F. A. Bianchi, A. Margara, and M. Pezzè. A Survey of Recent Trends in Testing Concurrent Software Systems. *IEEE Transactions on Software Engineering*, 44:747–783, 2018.

[Car13] J. L. Carlson. *Redis in Action*. Manning Publications Co., Greenwich, CT, USA, 2013. 5, 322 p.

[CGB15] Y. Cheng, A. Gupta, and A. R. Butt. An In-memory Object Caching Framework with Adaptive Load Balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, 4:1–4:16, 2015.

[Cod17] Codeahoy. Caching Strategies and How to Choose the Right One. `https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/`, 2017. Žiūrėta 2019-06-01.

[CTW+16] S. Chen, X. Tang, H. Wang, H. Zhao, and M. Guo. Towards Scalable and Reliable In-Memory Storage System: A Case Study with Redis. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pp. 1660–1667, 2016.

[DWL+10] R. Di, T. Wang, Y. Liang, and L. Su. The Analysis and Implementation of Partition Replication-Based Distributed Cache System. In *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*, pp. 719–724, 2010.

[Ehc19] Ehcache. Clustered Cache. `https://www.ehcache.org`, 2019. Žiūrėta 2019-06-02.

[FTL+07] J. S. Fitzgerald, S. Tjell, P. G. Larsen, and M. Verhoef. Validation Support for Distributed Real-Time Embedded Systems in VDM++. In *10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, pp. 331–340, 2007.

[JaJT+03] R. Joshi, L. Lamport and J. Matthews and S. Tasiran, M. Tuttle, and Y. Yu. Checking Cache-Coherence Protocols with TLA+. *Formal Methods in System Design*, 22:125–131, 2003.

[JGO+14] Z. Ji, I. Ganchev, M. O'Droma, and T. Ding. A Distributed Redis Framework for Use in the UCWW. In *2014 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery*, pp. 241–244, 2014.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16:872–923, 1994.

[LaSG+14] S. Liu, M. R. Rahman and S. Skeirik, I. Gupta, and J. Meseguer. Formal Modeling and Analysis of Cassandra in Maude. In *Formal Methods and Software Engineering*, pp. 332–347, 2014.

[LDY+18] D. Li, M. Dong, Y. Yuan, J. Chen, K. Ota, and Y. Tang. SEER-MCache: A Prefetchable Memory Object Caching System for IoT Real-Time Data Processing. *IEEE Internet of Things Journal*, 5:3648–3660, 2018.

[LFL17] B. Li, Y. Fu, and Z. Li. The research and improvement of distributee caching system Memcached. In *2017 4th International Conference on Information, Cybernetics and Computational Social Systems (ICCSS)*, pp. 460–463, 2017.

[Liu16] S. Liu. Validating Formal Specifications Using Testing-based Specification Animation. In *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*, pp. 29–35, New York, NY, USA. ACM, 2016.

[LM17] L. Lamport ir S. Merz. Auxiliary Variables in TLA+, 2017.

[LMT+02] L. Lamport, J. Matthews, M. Tuttle, and Y. Yu. Specifying and Verifying Systems with TLA+. In *Proceedings of the 10th Workshop on ACM SIGOPS European Workshop*, pp. 45–48, 2002.

[LNaM+15] S. Liu, S. Nguyen, J. Ganhotra and M. R. Rahman, I. Gupta, and J. Meseguer. Quantitative Analysis of Consistency in NoSQL Key-Value Stores. In *Proceedings of the 12th International Conference on Quantitative Evaluation of Systems - Volume 9259*, pp. 228–243, 2015.

[LZL+18] B. Luo, W. Zhu, P. Li, and Z. Han. Distributed Dynamic Cuckoo Filter System Based on Redis Cluster. In *2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS)*, pp. 244–247, 2018.

[Mal16] D. Malhotra. Deadlock Prevention Algorithm in Grid Environment. *MATEC Web of Conferences*, 57, 2016.

[Mem19] Memcached. Memcached About. `http://memcached.org/`, 2019. Žiūrėta 2019-06-02.

[MJS09] A. Mashkoor, J. Jacquot, and J. Souquières. Transformation Heuristics for Formal Requirements Validation by Animation. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems - SafeCert 2009*, 2009.

[MKE18] A. Mashkoor, F. Kossak, and A. Egyed. Evaluating the suitability of state-based formal methods for industrial deployment. *Software: Practice and Experience*, 48:2350–2379, 2018.

[Mos13] O. Mosbahi. Combining Formal Methods for the Development of Reactive Systems. *ACM Transactions on Embedded Computing Systems*, 12:16:1–16:29, 2013.

[MP17] D. Mery and M. Poppleton. Towards an integrated formal method for verification of liveness properties in distributed systems: with application to population protocols. *Software & Systems Modeling*, 16:1083–1115, 2017.

[NRZ+15] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How Amazon Web Services Uses Formal Methods. *Communications of the ACM*, 58:66–73, 2015.

[PI13] A. Patil and R. Ingle. Leveraging Information Bus with In-Memory Caching for Service Oriented Architecture. In *2013 Third International Conference on Advances in Computing and Communications*, pp. 382–388, 2013.

[Red18] Redis. Redis Cluster Specification. `https://redis.io/topics/cluster-spec`, 2018. Žiūrėta 2019-05-19.

[Red19] Redis. Redis Version 5.0.6 Git Repository. `https://github.com/redis/redis/tree/5.0.6/`, 2019. Žiūrėta 2019-10-13.

[SCY15] N. Singh, M. Chandra, and D. Yadav. Formal specification of asynchronous checkpointing using Event-B. In *2015 International Conference on Advances in Computer Engineering and Applications*, pp. 659–664, 2015.

[TYB+02] S. Tasiran, Y. Yu, B. Batson, and S. Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *Proceedings of the 3rd IEEE International Workshop on Microprocessor Test and Verification (MTV '02)*, 2002.

[TS98] I. Traoré and A. Sahraoui. Integrating Formal Methods in the Development Process of Distributed Systems. *IFAC Proceedings Volumes*, 31:63–68, 1998.

[VF07] B. Veal and A. Foong. Performance Scalability of a Multi-core Web Server. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, pp. 57–66, 2007.

[VZL08] V. Vishwanath, L. D. Zuck, and J. Leigh. Specification and Verification of LambdaRAM- A Wide-area Distributed Cache for High Performance Computing. In *2008 6th ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pp. 187–198, 2008.

[Win90] J. M. Wing. A Specifier's Introduction to Formal Methods. *Computer*, 23:8–23, 1990.

[WLB+09] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal Methods: Practice and Experience. *ACM Computing Surveys*, 41:19:1–19:36, 2009.

[XHZ10] P. Xiang, R. Hou, and Z. Zhou. Cache and consistency in NOSQL. In *2010 3rd International Conference on Computer Science and Information Technology*, pp. 117–120, 2010.

[XLJ+16] S. Xu, S. Lee, S. Jun, M Liu, J. Hicks, and Arvind. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proceedings of the VLDB Endowment*, 10:301–312, 2016.

[ZAA17] V. Zakhary, D. Agrawal, and A. E. Abbadi. Caching at the Web Scale. *Proceedings of the VLDB Endowment*, 10:2002–2005, 2017.

Priedas nr. 1

Išorinių veiksmų formali specifikacija

```
1  ---------------- MODULE RedisCluster_abstract ----------------
2  EXTENDS Integers, FiniteSets, TLC

4  CONSTANTS NODE, SLOTS

6  VARIABLES nodeFailed, nodeSlaves, slaveOf,
7            clusterState, clusterSlots, clusterKnownNodes

9  NULL_NODE ≜ CHOOSE n : n ∉ NODE

11 TCTypeOK ≜
12   ∧ nodeFailed ∈ [NODE → BOOLEAN ]
13   ∧ nodeSlaves ∈ [NODE → SUBSET NODE]
14   ∧ slaveOf ∈ [NODE → SUBSET NODE]
15   ∧ clusterState ∈ {"OK", "FAIL"}
16   ∧ clusterSlots ∈ [SLOTS → SUBSET NODE]
17   ∧ clusterKnownNodes ∈ SUBSET NODE

19 ---------------------------------------------------------------
20 CalculateState ≜
21     LET masterWithASingleSlot ≜
22             {node ∈ clusterKnownNodes : ∧ slaveOf[node] = {}
23                                         ∧ ∃ slot ∈ DOMAIN clusterSlots :
24                                              clusterSlots[slot] = {node}}

26         reachableMasters ≜ {node ∈ masterWithASingleSlot : nodeFailed[node] = FALSE}
27     IN  IF ((Cardinality(masterWithASingleSlot) ÷ 2) + 1) > Cardinality(reachableMasters)
28          THEN "FAIL"
29          ELSE "OK"

31 ---------------------------------------------------------------
   Client commands

36 AddNode(n) ≜
37     ∃ newNode ∈ NODE :
38        ∧ newNode ∉ clusterKnownNodes
39        ∧ clusterKnownNodes' = clusterKnownNodes ∪ {newNode}
40        ∧ UNCHANGED ⟨nodeFailed, slaveOf, nodeSlaves, clusterState, clusterSlots⟩

42 AddSlot(n) ≜
43     ∃ slotToChange ∈ SLOTS :
44        ∧ clusterSlots[slotToChange] = {}
45        ∧ clusterSlots' = [clusterSlots EXCEPT ![slotToChange] = {n}]
46        ∧ UNCHANGED ⟨nodeFailed, slaveOf, nodeSlaves, clusterState, clusterKnownNodes⟩

48 DelSlot(n) ≜
49     ∃ slotToChange ∈ SLOTS :
50        ∧ clusterSlots[slotToChange] ≠ {}
51        ∧ clusterSlots' = [clusterSlots EXCEPT ![slotToChange] = {}]
52        ∧ UNCHANGED ⟨nodeFailed, slaveOf, nodeSlaves, clusterState, clusterKnownNodes⟩

54 ReshardSlot(n) ≜
55      ∧ slaveOf[n] = {}
56      ∧ ∃ slotToMigrate ∈ SLOTS :
57          ∧ clusterSlots[slotToMigrate] = {n}
58          ∧ ∃ targetNode ∈ clusterKnownNodes :
```

```
59              ∧ targetNode ≠ n
60              ∧ slaveOf[targetNode] = {}
61              ∧ clusterSlots′ = [clusterSlots EXCEPT ![slotToMigrate] = {targetNode}]
62              ∧ UNCHANGED ⟨nodeFailed, slaveOf, nodeSlaves, clusterState, clusterKnownNodes⟩

64  FlushSlots(n) ≜
65      ∧ clusterSlots′ = [slot ∈ DOMAIN clusterSlots ↦ IF clusterSlots[slot] = {n}
66                                                       THEN {}
67                                                       ELSE clusterSlots[slot]]
68      ∧ UNCHANGED ⟨nodeFailed, slaveOf, nodeSlaves, clusterState, clusterKnownNodes⟩

70  RemoveNode(nodeToRemove) ≜
71      ∧ clusterKnownNodes′ = clusterKnownNodes \ {nodeToRemove}
72      ∧ clusterSlots′ = [slot ∈ DOMAIN clusterSlots ↦ IF clusterSlots[slot] = {nodeToRemove}
73                                                       THEN {}
74                                                       ELSE clusterSlots[slot]]
75       We are not interested into this node state and we assume it is not failed
76      ∧ nodeFailed′ = [nodeFailed EXCEPT ![nodeToRemove] = FALSE]
77      ∧ slaveOf′ = [node ∈ DOMAIN slaveOf ↦ IF ∨ slaveOf[node] = {nodeToRemove}
78                                               ∨ node = nodeToRemove
79                                            THEN {}
80                                            ELSE slaveOf[node]]
81      ∧ nodeSlaves′ = [node ∈ DOMAIN nodeSlaves ↦
82                                      IF node = nodeToRemove
83                                      THEN {}
84                                      ELSE nodeSlaves[node] \ {nodeToRemove}]
85      ∧ UNCHANGED ⟨clusterState⟩

87  ReplicateNode(n) ≜
88      ∃ masterToReplicate ∈ clusterKnownNodes :
89          ∧ n ≠ masterToReplicate
90          ∧ slaveOf[masterToReplicate] = {}
91          ∧ IF slaveOf[n] = {}
92              THEN ∧ Cardinality({slot ∈ DOMAIN clusterSlots : clusterSlots[slot] = {n}}) = 0
93                   ∧ nodeSlaves′ = [nodeSlaves EXCEPT ![masterToReplicate] = @ ∪ {n}]
94              ELSE ∃ previousMaster ∈ slaveOf[n] :
95                     nodeSlaves′ = [nodeSlaves EXCEPT ![masterToReplicate] = @ ∪ {n},
96                                                       ![previousMaster] = @ \ {n}]
97          ∧ slaveOf′ = [slaveOf EXCEPT ![n] = {masterToReplicate}]
98          ∧ UNCHANGED ⟨nodeFailed, clusterState, clusterSlots, clusterKnownNodes⟩

100 ManualFailover(n) ≜
101     ∧ slaveOf[n] ≠ {}
102     ∧ ∃ master ∈ slaveOf[n] :
103         ∧ slaveOf′ = [slaveOf EXCEPT ![n] = {},
104                                      ![master] = {n}]
105         ∧ nodeSlaves′ = [nodeSlaves EXCEPT ![n] = (nodeSlaves[master] \ {n}) ∪ {master},
106                                            ![master] = {}]
107         ∧ clusterSlots′ = [slot ∈ DOMAIN clusterSlots ↦ IF clusterSlots[slot] = {master}
108                                                          THEN {n}
109                                                          ELSE clusterSlots[slot]]
110         ∧ UNCHANGED ⟨nodeFailed, clusterState, clusterKnownNodes⟩

112 ClientCommand(n) ≜
113     ∨ AddNode(n)
114     ∨ ∧ n ∈ clusterKnownNodes
115       ∧ ∨ RemoveNode(n)
116         ∨ ReplicateNode(n)
117         ∨ ManualFailover(n)
118         ∨ AddSlot(n)
```

```
119             ∨ DelSlot(n)
120             ∨ ReshardSlot(n)
121             ∨ FlushSlots(n)

123 ├──────────────────────────────────────────────────────────────────┤
       Node availability changes

128 ElectMaster(failedNode) ≜
129     ∃ newMaster ∈ nodeSlaves[failedNode] :
130        ∧ slaveOf′ = [slaveOf EXCEPT ![newMaster] = {},
131                                     ![failedNode] = {newMaster}]
132        ∧ nodeSlaves′ =
133             [nodeSlaves EXCEPT
134                    ![newMaster] = (nodeSlaves[failedNode] \ {newMaster}) ∪ {failedNode},
135                    ![failedNode] = {}]
136        ∧ clusterSlots′ = [slot ∈ DOMAIN clusterSlots ↦ IF clusterSlots[slot] = {failedNode}
137                                                        THEN {newMaster}
138                                                        ELSE clusterSlots[slot]]

140 NodeFails(n) ≜
141     ∧ nodeFailed[n] = FALSE
142     ∧ nodeFailed′ = [nodeFailed EXCEPT ![n] = TRUE]
143     ∧ IF ∧ slaveOf[n] = {}
144          ∧ nodeSlaves[n] ≠ {}
145       THEN ∧ ElectMaster(n)
146            ∧ UNCHANGED ⟨clusterState, clusterKnownNodes⟩
147       ELSE UNCHANGED ⟨slaveOf, nodeSlaves, clusterSlots, clusterKnownNodes, clusterState⟩

149 NodeRejoins(n) ≜
150     ∧ nodeFailed[n] = TRUE
151     ∧ nodeFailed′ = [nodeFailed EXCEPT ![n] = FALSE]
152     ∧ UNCHANGED ⟨slaveOf, nodeSlaves, clusterSlots, clusterKnownNodes, clusterState⟩

154 NodeAvailabilityChange(n) ≜
155     ∨ NodeFails(n)
156     ∨ NodeRejoins(n)

158 ├──────────────────────────────────────────────────────────────────┤
       Scheduled jobs

163 CalculateClusterState ≜
164     ∧ clusterState′ = CalculateState
165     ∧ UNCHANGED ⟨nodeFailed, slaveOf, nodeSlaves, clusterSlots, clusterKnownNodes⟩

167 ScheduledJobs ≜
168     ∨ CalculateClusterState

170 ├──────────────────────────────────────────────────────────────────┤
       The complete spec of the Redis Cluster system.

175 FreshStartInit ≜
176     ∧ nodeFailed = [n ∈ NODE ↦ FALSE]
177     ∧ nodeSlaves = [n ∈ NODE ↦ {}]
178     ∧ slaveOf = [n ∈ NODE ↦ {}]
179     ∧ clusterState  = "OK"
180     ∧ clusterSlots  = [slot ∈ SLOTS ↦ {}]
181     ∧ clusterKnownNodes = {}

183 ThreeMasterNodesInit ≜
184     LET numOfSlots ≜ Cardinality(SLOTS) ÷ 3
185         firstNode ≜ CHOOSE x ∈ NODE : TRUE
```

```
186             firstNodeSlots ≜ 1 .. numOfSlots
187             secondNode ≜ CHOOSE x ∈ NODE : x ≠ firstNode
188             secondNodeSlots ≜ (numOfSlots + 1) .. (2 * numOfSlots)
189             thirdNode ≜ CHOOSE x ∈ NODE : x ≠ firstNode ∧ x ≠ secondNode
190             thirdNodeSlots ≜ (SLOTS \ firstNodeSlots) \ secondNodeSlots
191         IN  ∧ nodeFailed = [n ∈ NODE ↦ FALSE]
192             ∧ nodeSlaves = [n ∈ NODE ↦ {}]
193             ∧ slaveOf = [n ∈ NODE ↦ {}]
194             ∧ clusterState  = "OK"
195             ∧ clusterSlots  = [slot ∈ SLOTS ↦ IF slot ∈ firstNodeSlots
196                                                THEN {firstNode}
197                                                ELSE IF slot ∈ secondNodeSlots
198                                                THEN {secondNode}
199                                                ELSE {thirdNode}]
200             ∧ clusterKnownNodes = NODE

202 TCInit ≜ ThreeMasterNodesInit

204 TCNext ≜
205     ∨ ∃ n ∈ NODE : ∨ ClientCommand(n)
206                    ∨ NodeAvailabilityChange(n)
207     ∨ ScheduledJobs

209 TCSpec ≜ TCInit ∧ □[TCNext]⟨nodeFailed, slaveOf, nodeSlaves,
210                              clusterState, clusterSlots, clusterKnownNodes⟩

212 THEOREM TCSpec ⟹ □TCTypeOK

214 ─────────────────────────────────────────────────────────────
```

Priedas nr. 2

Žinučių apdorojimo formali specifikacija

```
1  ---------------- MODULE RedisCluster_messaging ----------------
2  EXTENDS Integers, Sequences, FiniteSets, TLC

4  CONSTANTS NODE,                       The set of nodes in the cluster
5            SLOTS_COUNT,                Number of slots in cluster
6            MAX_BUFFER_LEN,             Maximum number of message in send and recieve buffers
7            CLUSTER_REQUIRE_FULL_COVERAGE  configuration if full coverage required. Default: TRUE

9  VARIABLES clusterState,   clusterState[n] each node has a global variable that stores state of the cluster
10           eventLoop,      eventLoop[n] each node event loop
11           recieveBuffer,  recieveBuffer[n][m] each node n recieve buffer from each node m
12           sendBuffer      sendBuffer[n][m] each node n send buffer for each node m

14 Constraint ≜ ∀ n, m ∈ NODE :
15                 ∧ Len(sendBuffer[n][m]) ≤ MAX_BUFFER_LEN
16                 ∧ Len(recieveBuffer[n][m]) ≤ MAX_BUFFER_LEN

18 NULL_NODE ≜ CHOOSE n : n ∉ NODE   NULL value for NODE

20 NodeFlags    ≜ {
21                 "MASTER",     CLUSTER_NODE_MASTER
22                 "SLAVE",      CLUSTER_NODE_SLAVE
23                 "PFAIL",      CLUSTER_NODE_PFAIL – Failure? Need acknowledge
24                 "FAIL",       CLUSTER_NODE_FAIL – The node is believed to be malfunctioning
25                 "MYSELF",     CLUSTER_NODE_MYSELF – This node is myself
26                 "HANDSHAKE",  CLUSTER_NODE_HANDSHAKE – We have still to exchange the first ping
27                 "MEET"        CLUSTER_NODE_MEET – Send a MEET message to this node
28                 }

30 Slots ≜ 1 . . SLOTS_COUNT

32 ClusterNodeType ≜
33     [
34     name : NODE ∪ {NULL_NODE},
35     flags : SUBSET NodeFlags,   flags CLUSTER_NODE_...:
36     configEpoch : Nat,   Last configEpoch observed for this node
37     slots : [Slots → BOOLEAN ],   slots[n] specifies if slot is handled by this node
38     slaves : SUBSET NODE,   set of slave nodes
39     slaveOf : NODE ∪ {NULL_NODE},   master node.
40     hasLink : BOOLEAN ,   link –TCP\IP connection
41     pingSent : BOOLEAN ,   if ping was sent
42     pingPongHalfTimeOut : BOOLEAN ,   Time property if pong has not responded after a timeout
43     pingPongTimeOut : BOOLEAN ,   Time property if pong has not responded after a timeout
44     needsToBePinged  : BOOLEAN ,   If node needs to be pinged
45     handshakeTimeOut : BOOLEAN ,   Time property if node has not responded after a hanshake
46     failTimeOut : BOOLEAN ,   Time property if node has beed failed for more than specific time
47     failReports  : SUBSET (NODE),   List of nodes signaling this as failing
48     hasAddressChanged : BOOLEAN    if address has changed
49     ]

51 BeforeSleepActions ≜ {"HANDLE_FAILOVER", "UPDATE_STATE"}

53 NULL_ClusterNodeType ≜ CHOOSE n : n ∉ ClusterNodeType

55 ClusterStateType ≜
56     [
57     myself : NODE,   This node
```

```
58      currentEpoch : Nat,
59      clusterState  : {"OK",    CLUSTER_OK − Everything looks ok
60                       "FAIL"   CLUSTER_FAIL − The cluster can't work
61                      },   state of the cluster
62      nodes : [NODE → ClusterNodeType ∪ {NULL_ClusterNodeType}],   known nodes
63      nodesBlackList : SUBSET NODE,   Nodes we don't re-add for a few seconds
64      migratingSlotsTo : [Slots → (NODE ∪ {NULL_NODE})],
65      importingSlotsFrom : [Slots → (NODE ∪ {NULL_NODE})],
66      slots : [Slots → (NODE ∪ {NULL_NODE})],
67      mfStarted : BOOLEAN ,   Manual failover has started. (part of mf_end)
68      mfEnded : BOOLEAN ,   Manual failover time limit has passed. (part of mf_end)
69      mfSlave  : NODE ∪ {NULL_NODE},   Slave performing the manual failover.
70      mfMasterOffsetSet : BOOLEAN ,   If manual failover offset was set by slave
71      mfCanStart : BOOLEAN ,   If non-zero signal that the manual failover can start requesting masters vote.
72      todoBeforeSleep : SUBSET BeforeSleepActions      Things to do in before sleep in event loop.
73      ]

    Message types

79    For PING, MEET and PONG
80  GossipMsgDataType ≜ [
81                           nodeName : NODE,
82                           flags : SUBSET NodeFlags,
83                           hasAddressChanged : BOOLEAN
84                           ]

86  FailMsgDataType ≜ [
87                        nodeName : NODE
88                        ]

90  UpdateMsgDataType ≜ [
91                          nodeName : NODE,   Name of the slots owner
92                          configEpoch : Nat,   Config epoch of the specified instance
93                          slots : [Slots → BOOLEAN ]   slots[n] specifies if slot is handled by this node
94                          ]

96  AllMessagesType ≜ (SUBSET GossipMsgDataType) ∪ FailMsgDataType ∪ UpdateMsgDataType

98  MessageType ≜ [
99                     type : {"PING",     CLUSTERMSG_TYPE_PING − Ping
100                            "PONG",    CLUSTERMSG_TYPE_PONG − Pong (reply to Ping)
101                            "MEET",    CLUSTERMSG_TYPE_MEET − Meet "let's join" message
102                            "FAIL",    CLUSTERMSG_TYPE_FAIL − Mark node xxx as failing
103                            "UPDATE",   CLUSTERMSG_TYPE_UPDATE − Another node slots configuration
104                            "MFSTART"},   CLUSTERMSG_TYPE_MFSTART − Pause clients for manual failover
105                     currentEpoch : Nat,   The epoch accordingly to the sending node.
106                     configEpoch : Nat,   The config epoch if it's a master,
107                                          or the last epoch advertised by its master if it is a slave.
108                     sender : NODE,   Name of the sender node
109                     mySlots : [Slots → BOOLEAN ],
110                     slaveOf : NODE ∪ {NULL_NODE},
111                     flags : SUBSET NodeFlags,   Sender node flags
112                     data : AllMessagesType,
113                     state : {"OK", "FAIL"},   cluster state
114                     hasSenderAddressChanged : BOOLEAN ,
115                     msgFlags : SUBSET {"PAUSED"},   Master paused for manual failover.
116                       Bug 1 fix start
117                     migratingSlotsTo : [Slots → (NODE ∪ {NULL_NODE})]
118                       Bug 1 fix end
```

```
119                         ]

121   Internal type used to stack messages that needs to be send
122 MessageToSendType ≜ [
123                         msg : MessageType,
124                         reciever : NODE
125                         ]

127 BufferType ≜ [NODE → [NODE → Seq(MessageType)]]

    Event types
132 EventLinkType ≜ [
133                     name : NODE,
134                     hasWriteEvents : BOOLEAN ,
135                     hasReadEvent : BOOLEAN
136                     ]

138 NULL_EventLink ≜ CHOOSE e : e ∉ EventLinkType

140 TimeEventType ≜ [
141                     hasOccured : BOOLEAN ,
142                     iteration : Nat
143                     ]

145 EventLoopType ≜ [
146                     state : {"BEFORE_SLEEP",
147                              "PROCESSING_FILE_EVENTS",
148                              "PROCESSING_TIME_EVENTS"},
149                     eventsBeingProcessed : [NODE → EventLinkType ∪ {NULL_EventLink}],
150                     currentEventBeingProcessed : EventLinkType ∪ {NULL_EventLink},
151                     timeEvent : TimeEventType
152                     ]

154 ├──────────────────────────────────────────────────────────────────┤
    Helpers
158 CreateClusterNode(node, flags) ≜
159     [
160     name ↦ node,
161     flags ↦ flags,
162     configEpoch ↦ 0,
163     slots ↦ [slot ∈ Slots ↦ FALSE],
164     slaves ↦ {},
165     slaveOf ↦ NULL_NODE,
166     hasLink ↦ FALSE,
167     pingSent ↦ FALSE,
168     pingPongHalfTimeOut ↦ FALSE,
169     pingPongTimeOut ↦ FALSE,
170     needsToBePinged  ↦ FALSE,
171     handshakeTimeOut ↦ FALSE,
172     failTimeOut ↦ FALSE,
173     failReports   ↦ {},
174     hasAddressChanged ↦ FALSE
175     ]

177 GetNodeNumSlots(node) ≜
178     Cardinality({slot ∈ Slots : node.slots[slot] = TRUE})

180 GetNodesFromHashMap(clusterNodes) ≜
181     {node ∈ {clusterNodes[nodeName] : nodeName ∈ NODE} : node ≠ NULL_ClusterNodeType}

183 GetClusterNodes(cluster) ≜
```

```
184      GetNodesFromHashMap(cluster.nodes)

186  GetMastersWithSingleSlot(cluster) ≜
187      {node ∈ NODE : ∧ cluster.nodes[node] ≠ NULL_ClusterNodeType
188                     ∧ "MASTER" ∈ cluster.nodes[node].flags
189                     ∧ GetNodeNumSlots(cluster.nodes[node]) > 0}

191  GetClusterSize(cluster) ≜
192      Cardinality(GetMastersWithSingleSlot(cluster))

194  ClusterLookupNode(cluster, nodeName) ≜
195      IF nodeName = NULL_NODE
196       THEN NULL_ClusterNodeType
197       ELSE cluster.nodes[nodeName]

199  CalculateClusterState(cluster) ≜
200      IF ∧ CLUSTER_REQUIRE_FULL_COVERAGE
201         ∧ ∃ slotId ∈ Slots :
202                ∨ cluster.slots[slotId] = NULL_NODE
203                ∨ "FAIL" ∈ ClusterLookupNode(cluster, cluster.slots[slotId]).flags
204       THEN "FAIL"
205       ELSE LET mastersWithSingleSlot ≜ GetMastersWithSingleSlot(cluster)
206                reachableMasters ≜
207                  {node ∈ mastersWithSingleSlot :
208                          ∨ "FAIL" ∉ cluster.nodes[node].flags
209                          ∨ "PFAIL" ∉ cluster.nodes[node].flags}
210                neededQuorum ≜ (Cardinality(mastersWithSingleSlot) ÷ 2) + 1
211            IN  IF Cardinality(reachableMasters) < neededQuorum
212                 THEN "FAIL"
213                 ELSE "OK"

215  ClusterUpdateState(cluster) ≜
216      [cluster EXCEPT !.todoBeforeSleep = @ \ {"UPDATE_STATE"},
217                      !.clusterState = CalculateClusterState(cluster)]

219  IsClusterStateDoesNotNeedToBeUpdated(cluster) ≜
220      LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
221      IN  ∨ ∧ "MASTER" ∈ myself.flags
222            ∧ cluster.clusterState = "FAIL"
223          ∨ ∧ cluster.clusterState ≠ CalculateClusterState(cluster)
224            ∧ cluster.clusterState = "OK"
225            ∧ "MASTER" ∈ myself.flags

227  NodesHashMapAddNodes(clusterNodes, addedNodes) ≜
228      [x ∈ NODE ↦ IF ∃ node ∈ addedNodes : ∧ node ≠ NULL_ClusterNodeType
229                                           ∧ node.name = x
230                    THEN CHOOSE node ∈ addedNodes : ∧ node ≠ NULL_ClusterNodeType
231                                                    ∧ node.name = x
232                    ELSE clusterNodes[x]]

234  ResetManualFailover(cluster) ≜
235      [cluster EXCEPT !.mfStarted = FALSE,
236                      !.mfEnded = FALSE,
237                      !.mfSlave  = NULL_NODE,
238                      !.mfCanStart = FALSE,
239                      !.mfMasterOffsetSet = FALSE]

241  ClusterAddNode(cluster, node) ≜
242      [cluster EXCEPT !.nodes = NodesHashMapAddNodes(@, {node})]

244  ClusterNodeRemoveSlave(master, slaveName) ≜
```

```
245       [master EXCEPT !.slaves = @ \ {slaveName}]

247   ClusterSetNodeAsMaster(cluster, node) ≜
248       IF "MASTER" ∈ node.flags
249        THEN ⟨cluster, node⟩
250        ELSE LET master ≜ ClusterLookupNode(cluster, node.slaveOf)
251                  updatedMaster ≜ IF master ≠ NULL_ClusterNodeType
252                                   THEN ClusterNodeRemoveSlave(master, node.name)
253                                   ELSE master
254                  updatedSlave ≜ [node EXCEPT !.flags = (@ \ {"SLAVE"}) ∪ {"MASTER"},
255                                                !.slaveOf = NULL_NODE]
256              IN ⟨[cluster EXCEPT !.nodes = [@ EXCEPT ![updatedMaster.name] = updatedMaster,
257                                                        ![updatedSlave.name] = updatedSlave],
258                                   !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}],
259                   updatedSlave⟩

261   ClusterNodeAddSlave(master, slave) ≜
262       IF slave.name ∈ master.slaves
263        THEN master
264        ELSE [master EXCEPT !.slaves = @ ∪ {slave.name}]

266    If this node is currently a master, it is turned into a slave.
267   ClusterSetMaster(cluster, newMaster) ≜
268       LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
269       IN CASE "MASTER" ∈ myself.flags →
270             ResetManualFailover(
271                 [cluster EXCEPT
272                     !.nodes = [@ EXCEPT ![myself.name] =
273                                   [@ EXCEPT !.flags = (@ \ {"MASTER"}) ∪ {"SLAVE"},
274                                             !.slaveOf = newMaster],
275                               ![newMaster] = ClusterNodeAddSlave(@, myself)],
276                     !.importingSlotsFrom = [slot ∈ Slots ↦ NULL_NODE],
277                     !.migratingSlotsTo = [slot ∈ Slots ↦ NULL_NODE]])
278          □ ∧ "MASTER" ∉ myself.flags
279            ∧ myself.slaveOf ≠ NULL_NODE →
280             ResetManualFailover(
281               [cluster EXCEPT
282                   !.nodes = [@ EXCEPT ![myself.name] =
283                                 [@ EXCEPT !.slaveOf = newMaster],
284                             ![myself.slaveOf] = ClusterNodeRemoveSlave(@, cluster.myself),
285                             ![newMaster] = ClusterNodeAddSlave(@, myself)]])
286          □OTHER →
287             ResetManualFailover(
288               [cluster EXCEPT
289                   !.nodes = [@ EXCEPT ![myself.name] =
290                                 [@ EXCEPT !.slaveOf = newMaster],
291                             ![newMaster] = ClusterNodeAddSlave(@, myself)]])

293 ├────────────────────────────────────────────────────────────────────────────────┤
       Actions
297   CreateMessage(cluster, msgSender, msgReceiverName, msgType, msgData) ≜
298       LET master ≜ IF ∧ "SLAVE" ∈ msgSender.flags
299                       ∧ msgSender.slaveOf ≠ NULL_NODE
300                     THEN ClusterLookupNode(cluster, msgSender.slaveOf)
301                     ELSE msgSender
302           msg ≜ [
303                   type ↦ msgType,
304                   sender ↦ msgSender.name,
305                   mySlots ↦ master.slots,
```

```
306                  slaveOf ↦ msgSender.slaveOf,
307                  flags ↦ msgSender.flags,
308                  state ↦ cluster.clusterState,
309                  currentEpoch ↦ cluster.currentEpoch,
310                  configEpoch ↦ master.configEpoch,
311                  data ↦ msgData,
312                  hasSenderAddressChanged ↦ msgSender.hasAddressChanged,
313                  msgFlags ↦ IF ∧ "MASTER" ∈ msgSender.flags
314                                ∧ cluster.mfStarted
315                             THEN {"PAUSED"}
316                             ELSE {},
317                   Bug 1 fix start
318                  migratingSlotsTo ↦ cluster.migratingSlotsTo
319                   Bug 1 fix end
320                  ]
321     IN [
322         msg ↦ msg,
323         reciever ↦ msgReceiverName
324         ]

326   Turn set of messages into a sequence
327 CreateMultipleMsgs(msgsSet) ≜
328     LET seqIds ≜ 1 . . Cardinality(msgsSet)
329         possibleSeqs ≜ [seqIds → msgsSet]
330     IN CHOOSE seq ∈ possibleSeqs : msgsSet = {seq[x] : x ∈ seqIds}

332 CreateBroadcastMessages(cluster, msgType, msgData) ≜
333     LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
334         nodesToSendMsg ≜ {node ∈ NODE :
335                             ∧ cluster.nodes[node] ≠ NULL_ClusterNodeType
336                             ∧ cluster.nodes[node].hasLink
337                             ∧ "MYSELF" ∉ cluster.nodes[node].flags
338                             ∧ "HANDSHAKE" ∉ cluster.nodes[node].flags}
339     IN CreateMultipleMsgs({CreateMessage(cluster, myself, node, msgType, msgData)
340                             : node ∈ nodesToSendMsg})

342 GetPingMsgNodes[cluster ∈ ClusterStateType,
343                 myself ∈ NODE,
344                 freshNodes ∈ Int,  accumulator
345                 gossipCount ∈ Int,  accumulator
346                 wanted ∈ Int,  accumulator
347                 maxIterations ∈ Int,  accumulator
348                 processedNodes ∈ SUBSET NODE,
349                 msgNodes ∈ SUBSET ClusterNodeType] ≜   result
350     IF ∧ freshNodes > 0
351        ∧ gossipCount < wanted
352        ∧ maxIterations > 0
353        ∧ processedNodes ≠ NODE
354      THEN LET randomNodeId ≜ CHOOSE n ∈ NODE : n ∉ msgNodes
355               randomNode ≜ ClusterLookupNode(cluster, randomNodeId)
356           IN CASE ∨ randomNode = NULL_ClusterNodeType
357                   ∨ randomNodeId = myself
358                   ∨ "PFAIL" ∈ randomNode.flags
359                       → GetPingMsgNodes[cluster,
360                                         myself,
361                                         freshNodes,
362                                         gossipCount,
363                                         wanted,
364                                         maxIterations − 1,
```

```
365                                                   processedNodes ∪ {randomNodeId},
366                                                   msgNodes]
367                    □   ∨ "HANDSHAKE" ∈ randomNode.flags
368                        ∨ (∧¬randomNode.hasLink
369                           ∧ GetNodeNumSlots(randomNode) = 0)
370                             → GetPingMsgNodes[cluster,
371                                               myself,
372                                               freshNodes − 1,
373                                               gossipCount,
374                                               wanted,
375                                               maxIterations − 1,
376                                               processedNodes ∪ {randomNodeId},
377                                               msgNodes]
378                    □   OTHER  → GetPingMsgNodes[cluster,
379                                                 myself,
380                                                 freshNodes − 1,
381                                                 gossipCount + 1,
382                                                 wanted,
383                                                 maxIterations − 1,
384                                                 processedNodes ∪ {randomNodeId},
385                                                 msgNodes ∪ {randomNode}]
386    ELSE msgNodes

388 GossipMsgData(node) ≜
389    [
390    nodeName ↦ node.name,
391    flags ↦ node.flags,
392    hasAddressChanged ↦ node.hasAddressChanged
393    ]

395 CreatePingPongMeetMsgData(cluster, sender) ≜
396    LET clusterNodes ≜ GetClusterNodes(cluster)
397         freshnodes is the max number of nodes we can hope to append at all: nodes
398         available minus two (ourself and the node we are sending the message to)
399        freshNodes ≜ Cardinality(clusterNodes) − 2
400         1/10 of the number of nodes and anyway at least 3
401        tempWantedNodes ≜ Cardinality(clusterNodes)%10
402        wantedNodes ≜ IF tempWantedNodes < 3
403                      THEN 3
404                      ELSE tempWantedNodes
405        finalWantedNodesCount ≜ IF wantedNodes > freshNodes
406                                THEN freshNodes
407                                ELSE wantedNodes
408        gossipNodes ≜ GetPingMsgNodes[cluster,
409                                      sender.name,
410                                      freshNodes,
411                                      0,
412                                      finalWantedNodesCount,
413                                      wantedNodes ∗ 3,
414                                      {},
415                                      {}]

417        failingNodes ≜ {node ∈ clusterNodes : "PFAIL" ∈ node.flags}
418    IN {GossipMsgData(node) : node ∈ (gossipNodes ∪ failingNodes)}

420 ClusterCreatePingPongMeetMsg(cluster, sender, recieverName, msgType) ≜
421    LET msgData ≜ CreatePingPongMeetMsgData(cluster, sender)
422    IN CreateMessage(cluster, sender, recieverName, msgType, msgData)

424 ClusterCreatePongMsg(cluster, sender, reciever) ≜
```

```
425       ClusterCreatePingPongMeetMsg(cluster, sender, reciever.name, "PONG")

427   ClusterCreateMeetMsg(cluster, sender, reciever) ≜
428       ClusterCreatePingPongMeetMsg(cluster, sender, reciever.name, "MEET")

430   ClusterCreatePingMsg(cluster, sender, reciever) ≜
431       ClusterCreatePingPongMeetMsg(cluster, sender, reciever.name, "PING")

433   CreateBroadcastPong(cluster) ≜
434       LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
435           msgData ≜ CreatePingPongMeetMsgData(cluster, myself)
436       IN  CreateBroadcastMessages(cluster, "PONG", msgData)

438   CreateFailMessages(cluster, failingNodeName) ≜
439       LET msgData ≜ [nodeName ↦ failingNodeName]
440       IN  CreateBroadcastMessages(cluster, "FAIL", msgData)

442   CreateUpdateMessage(cluster, recieverName, updatedNode) ≜
443       LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
444           msgData ≜ [nodeName ↦ updatedNode.name,
445                      configEpoch ↦ updatedNode.configEpoch,
446                      slots ↦ updatedNode.slots]
447       IN  CreateMessage(cluster, myself, recieverName, "UPDATE", msgData)

449   ClusterSendMFStart(cluster, recieverName) ≜
450       LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
451       IN  CreateMessage(cluster, myself, recieverName, "MFSTART", {})

453    This function implements the final part of automatic and manual failovers, where
454    the slave grabs its master's hash slots, and propagates the new configuration.
455   ClusterFailoverReplaceYourMaster(cluster) ≜
456       LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
457           oldMaster ≜ ClusterLookupNode(cluster, myself.slaveOf)
458       IN  IF ∨ "MASTER" ∈ myself.flags
459              ∨ oldMaster = NULL_ClusterNodeType
460           THEN ⟨cluster,
461                 ⟨⟩⟩
462           ELSE LET slotsToUpdate ≜ {slot ∈ Slots : oldMaster.slots[slot]}
463                    updatedClusterInfo ≜
464                      [cluster EXCEPT
465                         !.nodes = [@ EXCEPT
466                             1) Turn this node into a master.
467                             2) Claim all the slots assigned to our master.
468                            ![myself.name] = [@ EXCEPT
469                                                  !.flags = (@ \ {"SLAVE"}) ∪ {"MASTER"},
470                                                  !.slaveOf = NULL_NODE,
471                                                  !.slots = [slot ∈ Slots ↦
472                                                              IF slot ∈ slotsToUpdate
473                                                              THEN TRUE
474                                                              ELSE @[slot]]],
475                            ![myself.slaveOf] = [@ EXCEPT
476                                                  !.slaves = @ \ {myself.name},
477                                                  !.slots = [slot ∈ Slots ↦
478                                                              IF slot ∈ slotsToUpdate
479                                                              THEN FALSE
480                                                              ELSE @[slot]]]],
481                         2) Claim all the slots assigned to our master.
482                         !.slots = [slot ∈ Slots ↦
483                                     IF slot ∈ slotsToUpdate
484                                     THEN myself.name
```

```
485                                   ELSE @[slot]],
486                           3) Update state
487                           ! .todoBeforeSleep = @ \ {"UPDATE_STATE"},
488                           ! .clusterState = CalculateClusterState(cluster),
489                           5) If there was a manual failover in progress, clear the state
490                           ! .mfStarted = FALSE,
491                           ! .mfEnded = FALSE,
492                           ! .mfSlave  = NULL_NODE,
493                           ! .mfCanStart = FALSE,
494                           ! .mfMasterOffsetSet = FALSE]
495                       4) Pong all the other nodes so that they can update the state
496                         accordingly and detect that we switched to master role.
497                       msgsToSend ≜ CreateBroadcastPong(updatedClusterInfo)
498              IN ⟨updatedClusterInfo,
499                  msgsToSend⟩

501  This function is called if we are a slave node and our master serving
502  a non-zero amount of hash slots is in FAIL state.
503  The gaol of this function is:
504  1) To check if we are able to perform a failover, is our data updated?
505  2) Try to get elected by masters.
506  3) Perform the failover informing all the other nodes.
507 ClusterHandleSlaveFailover(cluster) ≜
508    LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
509        master ≜ ClusterLookupNode(cluster, myself.slaveOf)
510        manualFailover ≜ ∧ cluster.mfStarted
511                         ∧ cluster.mfCanStart
512    IN IF ∨ "MASTER" ∈ myself.flags
513          ∨ master = NULL_ClusterNodeType
514          ∨ ("FAIL" ∉ master.flags ∧ ¬manualFailover)
515          ∨ GetNodeNumSlots(master) = 0
516        THEN ⟨[cluster EXCEPT ! .todoBeforeSleep = @ \ {"HANDLE_FAILOVER"}],
517              ⟨⟩⟩
518        ELSE IF exists new master
519                  ∃ slave ∈ master.slaves : "MASTER" ∈ ClusterLookupNode(clusterState[slave], slave).flags
520              THEN ⟨[cluster EXCEPT ! .todoBeforeSleep = @ \ {"HANDLE_FAILOVER"}],
521                    ⟨⟩⟩
522              ELSE LET updatedClusterInfo ≜
523                         [cluster EXCEPT
524                             ! .todoBeforeSleep = @ \ {"HANDLE_FAILOVER"},
525                             ! .currentEpoch = @ + 1,
526                             ! .nodes = [@ EXCEPT
527                                          ! [myself.name] =
528                                            [@ EXCEPT ! .configEpoch = cluster.currentEpoch + 1]]]
529                   IN ClusterFailoverReplaceYourMaster(updatedClusterInfo)

531 ClusterHandleManualFailover(cluster) ≜
532    IF ∨ ¬cluster.mfStarted
533       ∨ cluster.mfCanStart
534       ∨ ¬cluster.mfMasterOffsetSet
535     THEN cluster
536     ELSE [cluster EXCEPT ! .mfCanStart = TRUE]

538 ProcessDeletedNodes(node, deletedNodes) ≜
539    [node EXCEPT ! .slaveOf = IF @ ∈ deletedNodes THEN NULL_NODE ELSE @,
540                 ! .slaves = @ \ deletedNodes,
541                 ! .failReports = @ \ deletedNodes]

543 ProcessNodesWithEstablishedLink(node) ≜
```

```
544        [node EXCEPT !.hasLink = TRUE,
545                     !.pingSent = TRUE,
546                     !.flags = @ \ {"MEET"}]

548   CronJobProcessNodes[nodesToProcess ∈ SUBSET NODE,
549                       cluster ∈ ClusterStateType,
550                       msgs ∈ Seq(MessageToSendType)] ≜
551        IF Cardinality(nodesToProcess) > 0
552         THEN LET nodeToProcessId ≜ CHOOSE n ∈ NODE : n ∈ nodesToProcess
553                  node ≜ ClusterLookupNode(cluster, nodeToProcessId)
554              IN  CASE ∨ node = NULL_ClusterNodeType
555                       ∨ "MYSELF" ∈ node.flags
556                          → CronJobProcessNodes[nodesToProcess \ {nodeToProcessId},
557                                                cluster,
558                                                msgs]
559                    □  ∧ "HANDSHAKE" ∈ node.flags
560                       ∧ node.handshakeTimeOut
561                          → CronJobProcessNodes
562                              [nodesToProcess \ {nodeToProcessId},
563                               [cluster EXCEPT
564                                  !.nodes =
565                                     [x ∈ NODE ↦ IF @[x] ≠ NULL_ClusterNodeType
566                                                  THEN IF x ≠ nodeToProcessId
567                                                        THEN ProcessDeletedNodes(@[x],
568                                                                                 {nodeToProcessId})
569                                                        ELSE NULL_ClusterNodeType
570                                                  ELSE @[x]],
571                                  !.slots = [slot ∈ Slots ↦ IF @[slot] = nodeToProcessId
572                                                              THEN NULL_NODE
573                                                              ELSE @[slot]],
574                                  !.migratingSlotsTo = [slot ∈ Slots ↦ IF @[slot] = nodeToProcessId
575                                                                         THEN NULL_NODE
576                                                                         ELSE @[slot]],
577                                  !.importingSlotsFrom = [slot ∈ Slots ↦ IF @[slot] = nodeToProcessId
578                                                                           THEN NULL_NODE
579                                                                           ELSE @[slot]]],
580                               msgs]
581                    □  ∧ ¬node.hasLink
582                          → LET updatedClusterInfo ≜
583                                  [cluster EXCEPT
584                                     !.nodes =
585                                        [@ EXCEPT ![nodeToProcessId] =
586                                            ProcessNodesWithEstablishedLink(@)]]
587                                myself ≜ ClusterLookupNode(updatedClusterInfo,
588                                                           updatedClusterInfo.myself)
589                            IN  CronJobProcessNodes
590                                  [nodesToProcess,
591                                   updatedClusterInfo,
592                                   IF "MEET" ∈ node.flags
593                                    THEN Append(msgs,
594                                                ClusterCreateMeetMsg(updatedClusterInfo,
595                                                                     myself,
596                                                                     node))
597                                    ELSE Append(msgs,
598                                                ClusterCreatePingMsg(updatedClusterInfo,
599                                                                     myself,
600                                                                     node))]
601                    remove link to nodes that exceed timeout so link would be reconnected automatically
```

```
602                 □ ∧ node.hasLink
603                   ∧ node.pingSent
604                   ∧ node.pingPongHalfTimeOut
605                     → CronJobProcessNodes[nodesToProcess \ {nodeToProcessId},
606                                           [cluster EXCEPT
607                                             !.nodes =
608                                                 [@ EXCEPT ![nodeToProcessId] =
609                                                            [@ EXCEPT !.hasLink = FALSE]]],
610                                           msgs]
611                   If we have currently no active ping in this instance, and the received PONG is older than half
612                   the cluster timeout, send a new ping now, to ensure all the nodes are pinged without a too big delay.
613                 □ ∨ ∧ node.hasLink
614                     ∧ ¬node.pingSent
615                     ∧ node.needsToBePinged
616                    If we are a master and one of the slaves requested a manual failover, ping it continuously.
617                   ∨ ∧ node.hasLink
618                     ∧ cluster.mfStarted
619                     ∧ "MASTER" ∈ ClusterLookupNode(cluster, cluster.myself).flags
620                     ∧ cluster.mfSlave = nodeToProcessId
621                     → LET updatedClusterInfo ≜
622                             [cluster EXCEPT
623                               !.nodes =
624                                 [@ EXCEPT ![nodeToProcessId] =
625                                     [@ EXCEPT !.pingSent = TRUE,
626                                               !.pingPongHalfTimeOut = FALSE,
627                                               !.pingPongTimeOut = FALSE,
628                                               !.needsToBePinged  = FALSE]]]
629                           myself ≜ ClusterLookupNode(updatedClusterInfo, updatedClusterInfo.myself)
630                       IN  CronJobProcessNodes[nodesToProcess \ {nodeToProcessId},
631                                               updatedClusterInfo,
632                                               Append(msgs,
633                                                      ClusterCreatePingMsg(updatedClusterInfo,
634                                                                           myself,
635                                                                           node))]
636                 □ ∧ node.pingSent   Timeout reached. Set the node as possibly failing if it is not already in this state.
637                   ∧ node.pingPongTimeOut
638                   ∧ ∨ "PFAIL" ∉ node.flags
639                     ∨ "FAIL" ∉ node.flags
640                     → CronJobProcessNodes[nodesToProcess \ {nodeToProcessId},
641                                           [cluster EXCEPT
642                                             !.nodes =
643                                                 [@ EXCEPT
644                                                     ![nodeToProcessId] =
645                                                         [@ EXCEPT !.flags = @ ∪ {"PFAIL"}]]],
646                                           msgs]
647                 □OTHER → CronJobProcessNodes[nodesToProcess \ {nodeToProcessId},
648                                              cluster,
649                                              msgs]
650     ELSE ⟨cluster, msgs⟩

652 ClusterCronJob(cluster) ≜
653     LET processNodesResult ≜ CronJobProcessNodes[NODE, cluster, ⟨⟩]

655         Abort a manual failover if the timeout is reached.
656         manualFailoverTimeout ≜ ∧ processNodesResult[1].mfStarted
657                                 ∧ processNodesResult[1].mfEnded
658         updatedClusterInfo ≜
659             [processNodesResult[1] EXCEPT
```

```
660                  !.mfStarted = IF manualFailoverTimeout THEN FALSE ELSE @,
661                  !.mfEnded = IF manualFailoverTimeout THEN FALSE ELSE @,
662                  !.mfSlave   = IF manualFailoverTimeout THEN NULL_NODE ELSE @,
663                  !.mfCanStart = IF manualFailoverTimeout THEN FALSE ELSE @,
664                  !.mfMasterOffsetSet = IF manualFailoverTimeout THEN FALSE ELSE @,
665                  !.clusterState = IF ∧ @ ≠ "FAIL"
666                                      ∧ ∀ node ∈ NODE :
667                                          processNodesResult[1].nodes[node] ≠ NULL_ClusterNodeType ⟹
668                                            "PFAIL" ∉ processNodesResult[1].nodes[node].flags
669                                      ∧ IsClusterStateDoesNotNeedToBeUpdated(processNodesResult[1])
670                                   THEN @
671                                   ELSE CalculateClusterState(processNodesResult[1])]

673        myself ≜ ClusterLookupNode(updatedClusterInfo, updatedClusterInfo.myself)
674    IN  IF "SLAVE" ∈ myself.flags
675        THEN LET handleSlaveFailoverResult ≜
676                   ClusterHandleSlaveFailover(ClusterHandleManualFailover(updatedClusterInfo))
677             IN ⟨handleSlaveFailoverResult[1],
678                 processNodesResult[2] ∘ handleSlaveFailoverResult[2]⟩
679        ELSE ⟨updatedClusterInfo,
680              processNodesResult[2]⟩

682 ProcessGossipSectionNodes[gossipMsgData ∈ SUBSET GossipMsgDataType,
683                           senderNode ∈ ClusterNodeType,
684                           cluster ∈ ClusterStateType,
685                           msgs ∈ Seq(MessageToSendType)] ≜
686    IF Cardinality(gossipMsgData) > 0
687    THEN LET gossipInfo ≜ CHOOSE info ∈ gossipMsgData : info ∈ gossipMsgData
688             node ≜ ClusterLookupNode(cluster, gossipInfo.nodeName)
689         IN CASE ∧ node = NULL_ClusterNodeType
690                 ∧ senderNode ≠ NULL_ClusterNodeType
691                 ∧ gossipInfo.nodeName ∉ cluster.nodesBlackList
692                   → ProcessGossipSectionNodes
693                       [gossipMsgData \ {gossipInfo},
694                        senderNode,
695                        [cluster EXCEPT
696                           !.nodes = [@ EXCEPT
697                                        ![gossipInfo.nodeName] =
698                                          CreateClusterNode(gossipInfo.nodeName,
699                                                            {"HANDSHAKE", "MEET"})]],
700                        msgs]
701            □ ∧ node ≠ NULL_ClusterNodeType
702              ∧ senderNode ≠ NULL_ClusterNodeType
703              ∧ "MASTER" ∈ senderNode.flags
704              ∧ node.name ≠ cluster.myself
705              ∧ ∨ "FAIL"  ∈ gossipInfo.flags
706                ∨ "PFAIL" ∈ gossipInfo.flags
707                → LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
708                      currentNodeIsMaster ≜ "MASTER" ∈ myself.flags
709                      failureReports ≜ node.failReports ∪ {senderNode.name}

711                      neededQuorum ≜ (GetClusterSize(cluster) ÷ 2) + 1
712                      numberOfFailureReports ≜ Cardinality(failureReports)
713                          Also count myself as a voter if I'm a master.
714                        + (IF currentNodeIsMaster THEN 1 ELSE 0)
715                      isNodeNeedsToBeMarkedAsFailing ≜
716                          ∧ "PFAIL" ∈ node.flags
717                          ∧ "FAIL" ∉ node.flags
718                          weak agreement from masters.
```

```
719                                        ∧ numberOfFailureReports ≥ neededQuorum

721                                    updatedClusterInfo ≜
722                                        [cluster EXCEPT
723                                            ! .nodes = [@ EXCEPT
724                                                ! [gossipInfo.nodeName] =
725                                                    [@ EXCEPT
726                                                        ! .failReports = failureReports,
727                                                        ! .flags = IF isNodeNeedsToBeMarkedAsFailing
728                                                                   THEN ((@ \ {"PFAIL"}) ∪ {"FAIL"})
729                                                                   ELSE @]],
730                                            ! .todoBeforeSleep =
731                                                IF isNodeNeedsToBeMarkedAsFailing
732                                                THEN @ ∪ {"UPDATE_STATE"}
733                                                ELSE @]
734                                IN
735                                ProcessGossipSectionNodes[gossipMsgData \ {gossipInfo},
736                                                          senderNode,
737                                                          updatedClusterInfo,
738                                                          IF ∧ isNodeNeedsToBeMarkedAsFailing
739                                                             ∧ currentNodeIsMaster
740                                                          THEN msgs ∘ CreateFailMessages(updatedClusterInfo,
741                                                                                         node.name)
742                                                          ELSE msgs]
743                        □ OTHER →
744                            LET isFailureReportNeedsToBeRemoved ≜
745                                    ∧ senderNode ≠ NULL_ClusterNodeType
746                                    ∧ "MASTER" ∈ senderNode.flags
747                                    ∧ node.name ≠ cluster.myself
748                                    ∧ "FAIL" ∉ gossipInfo.flags
749                                    ∧ "PFAIL" ∉ gossipInfo.flags

751                                hasAddressChanged ≜
752                                    ∧ ∨ "FAIL" ∈ node.flags
753                                      ∨ "PFAIL" ∈ node.flags
754                                    ∧ "FAIL" ∉ gossipInfo.flags
755                                    ∧ "PFAIL" ∉ gossipInfo.flags
756                                    ∧ gossipInfo.hasAddressChanged
757                                updatedClusterInfo ≜
758                                    [cluster EXCEPT
759                                        ! .nodes = [@ EXCEPT
760                                            ! [gossipInfo.nodeName] =
761                                                [@ EXCEPT
762                                                    ! .failReports = IF isFailureReportNeedsToBeRemoved
763                                                                     THEN @ \ {senderNode.name}
764                                                                     ELSE @,
765                                                    ! .hasLink = IF hasAddressChanged THEN FALSE ELSE @,
766                                                    ! .hasAddressChanged =
767                                                        IF hasAddressChanged
768                                                        THEN FALSE
769                                                        ELSE @]]]
770                            IN
771                            ProcessGossipSectionNodes[gossipMsgData \ {gossipInfo},
772                                                      senderNode,
773                                                      updatedClusterInfo,
774                                                      msgs]
775        ELSE ⟨cluster, msgs⟩

777    ClusterProcessGossipSection(cluster, msg) ≜
```

```
778    IF msg.data ≠ {}
779     THEN LET sender ≜ ClusterLookupNode(cluster, msg.sender)
780             IN  ProcessGossipSectionNodes[msg.data, sender, cluster, ⟨⟩]
781     ELSE ⟨cluster,
782            ⟨⟩⟩

784  This action is called when we receive a master configuration via a PING, PONG or UPDATE packet.
785  In this action we rebind the slots with newer configuration compared to ourlocal configuration,
786  and if needed, we turn ourself into a replica of the node.
787 ClusterUpdateSlotsConfigWith(cluster, sender, senderConfigEpoch, slots) ≜
788    IF ClusterLookupNode(cluster, cluster.myself) = sender
789     THEN cluster  Discarding UPDATE message about myself.
790     ELSE LET
791              slotsToUpdate ≜
792                 {slot ∈ Slots : ∧ slots[slot]
793                                 ∧ cluster.slots[slot] ≠ sender.name
794                                 ∧ cluster.importingSlotsFrom = NULL_NODE
795                                 ∧ ∨ cluster.slots[slot] = NULL_NODE  The slot was unassigned
796                                     the new node claims it with a greater configEpoch
797                                   ∨ ClusterLookupNode(cluster, cluster.slots[slot]).configEpoch
798                                          < senderConfigEpoch}
799                 updatedClusterInfo ≜
800                    [cluster EXCEPT
801                       !.nodes =
802                         [x ∈ NODE ↦
803                          IF cluster.nodes[x] ≠ NULL_ClusterNodeType
804                           THEN [cluster.nodes[x] EXCEPT
805                                   !.slots = [slot ∈ Slots ↦ IF slot ∈ slotsToUpdate
806                                                              THEN x = sender.name
807                                                              ELSE @[slot]]]
808                           ELSE cluster.nodes[x]],
809                       !.slots = [slot ∈ Slots ↦ IF slot ∈ slotsToUpdate
810                                                  THEN sender.name
811                                                  ELSE @[slot]],
812                       !.todoBeforeSleep = IF Cardinality(slotsToUpdate) > 0
813                                            THEN @ ∪ {"UPDATE_STATE"}
814                                            ELSE @,
815                        Bug 1 fix start
816                       !.migratingSlotsTo = [slot ∈ Slots ↦ IF slot ∈ slotsToUpdate
817                                                             THEN NULL_NODE
818                                                             ELSE @[slot]]]
819                        Bug 1 fix end

821                  If at least one slot was reassigned from a node to another node
822                  with a greater configEpoch, it is possible that:
823                  1) We are a master left without slots. This means that we were
824                    failed over and we should turn into a replica of the new
825                    master.
826                  2) We are a slave and our master is left without slots. We need
827                    to replicate to the new slots owner.
828                 myself ≜ ClusterLookupNode(updatedClusterInfo, cluster.myself)
829                 curMaster ≜ IF "MASTER" ∈ myself.flags
830                               THEN myself
831                               ELSE ClusterLookupNode(updatedClusterInfo, myself.slaveOf)
832           IN  IF  was slot assigned to other node from myself
833                  ∧ ∃ slot ∈ slotsToUpdate : cluster.slots[slot] = curMaster
834                  ∧ GetNodeNumSlots(curMaster) = 0
835                THEN ClusterSetMaster(updatedClusterInfo, sender.name)
```

```
836                      ELSE updatedClusterInfo

   Message processing
841  This action is called when this node is a master, and we receive from
842  another master a configuration epoch that is equal to our configuration epoch.
843  ClusterHandleConfigEpochCollision(cluster, sender) ≜
844     LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
845     IN IF  Prerequisites: nodes have the same configEpoch and are both masters.
846           ∧ sender.configEpoch = myself.configEpoch
847           ∧ "MASTER" ∈ sender.flags
848           ∧ "MASTER" ∈ myself.flags
849           ∧ sender.name ≥ myself.name  Don't act if the colliding node has a smaller Node ID.
850         THEN LET newCurrentEpoch ≜ cluster.currentEpoch + 1
851               IN [cluster EXCEPT !.currentEpoch = newCurrentEpoch,
852                                  !.nodes = [@ EXCEPT ![cluster.myself] =
853                                        [@ EXCEPT !.configEpoch = newCurrentEpoch]]]
854         ELSE cluster

856  This action is called only if a node is marked as FAIL, but we are able
857  to reach it again. It checks if there are the conditions to undo the FAIL state.
858  ClearNodeFailureIfNeeded(node) ≜
859     For slaves we always clear the FAIL flag if we can contact the node again.
860     IF ∨ "SLAVE" ∈ node.flags
861        ∨ GetNodeNumSlots(node) = 0
862     THEN ⟨[node EXCEPT !.flags = @ \ {"FAIL"}],
863           TRUE⟩
864     If it is a master and...:
865     1) The FAIL state is old enough.
866     2) It is yet serving slots from our point of view (not failed over).
867     Apparently no one is going to fix these slots, clear the FAIL flag.
868     ELSE IF ∧ "MASTER" ∈ node.flags
869             ∧ GetNodeNumSlots(node) > 0
870             ∧ node.failTimeOut
871     THEN ⟨[node EXCEPT !.flags = @ \ {"FAIL"},
872                        !.failTimeOut = FALSE],
873           TRUE⟩
874     ELSE ⟨node, FALSE⟩

876  ProcessRoleChanges(cluster, sender, msg) ≜
877     IF msg.slaveOf = NULL_NODE
878     THEN ClusterSetNodeAsMaster(cluster, sender)  Node reports it is master
879     ELSE LET  node reports it is slave
880              newMaster ≜ ClusterLookupNode(cluster, msg.slaveOf)

882              senderIsMaster ≜ "MASTER" ∈ sender.flags
883                master node changed for this slave
884              masterChanged ≜ ∧ newMaster ≠ NULL_ClusterNodeType
885                              ∧ sender.slaveOf ≠ newMaster.name
886              updatedSender ≜
887                 [sender EXCEPT
888                    !.flags = (@ \ {"MASTER"}) ∪ {"SLAVE"},
889                    !.slots = [n ∈ Slots ↦ FALSE],
890                    !.slaveOf = IF masterChanged THEN newMaster.name ELSE @]

892          IN ⟨[cluster EXCEPT
893                 !.slots = [slot ∈ Slots ↦ IF ∧ senderIsMaster
894                                              ∧ @[slot] = sender.name
895                                           THEN NULL_NODE
896                                           ELSE @[slot]],
```

```
897                   ! .todoBeforeSleep = IF senderIsMaster
898                                           THEN @ ∪ {"UPDATE_STATE"}
899                                           ELSE @,
900                   ! .nodes =
901                     IF ∧ masterChanged
902                        ∧ sender.slaveOf = NULL_NODE
903                     THEN [@ EXCEPT
904                               ! [sender.name] = updatedSender,
905                               ! [newMaster.name] = ClusterNodeAddSlave(@, sender.name)]
906                     ELSE IF ∧ masterChanged
907                     THEN [@ EXCEPT
908                               ! [sender.name] = updatedSender,
909                               ! [sender.slaveOf]    = ClusterNodeRemoveSlave(@, sender.name),
910                               ! [newMaster.name] = ClusterNodeAddSlave(@, sender.name)]
911                     ELSE [@ EXCEPT ! [sender.name] = updatedSender]],
912               updatedSender⟩

914  Procondition Sender is a known node
915  UpdateInfoAboutServerSlots(cluster, sender, msg) ≜
916    LET senderMaster ≜ IF "MASTER" ∈ sender.flags
917                          THEN sender
918                          ELSE ClusterLookupNode(cluster, sender.slaveOf)
919        slotsHasChanged ≜ ∧ senderMaster ≠ NULL_ClusterNodeType
920                          ∧ ∃ slot ∈ Slots : senderMaster.slots[slot] ≠ msg.mySlots[slot]
921    IN  IF slotsHasChanged
922          THEN LET  1) If the sender of the message is a master, and we detected that
923                      the set of slots it claims changed, scan the slots to see if we
924                      need to update our configuration.
925                   clusterWithUpdatedSlots ≜ IF "MASTER" ∈ sender.flags
926                                               THEN ClusterUpdateSlotsConfigWith(cluster,
927                                                                                 sender,
928                                                                                 msg.configEpoch,
929                                                                                 msg.mySlots)
930                                               ELSE cluster
931                     2) We also check for the reverse condition, that is, the sender
932                       claims to serve slots we know are served by a master with a
933                       greater configEpoch. If this happens we inform the sender.
934                   nodeToSendUpdate ≜
935                     IF ∃ slot ∈ Slots :
936                          ∧ msg.mySlots[slot]
937                          ∧ clusterWithUpdatedSlots.slots[slot] ≠ sender.name
938                          ∧ clusterWithUpdatedSlots.slots[slot] ≠ NULL_NODE
939                          ∧ ClusterLookupNode(clusterWithUpdatedSlots,
940                                              clusterWithUpdatedSlots.slots[slot]).configEpoch >
941                                        msg.configEpoch
942                     THEN LET slotId ≜
943                                CHOOSE slot ∈ Slots :
944                                  ∧ msg.mySlots[slot]
945                                  ∧ clusterWithUpdatedSlots.slots[slot] ≠ sender.name
946                                  ∧ clusterWithUpdatedSlots.slots[slot] ≠ NULL_NODE
947                                  ∧ ClusterLookupNode(clusterWithUpdatedSlots,
948                                                      clusterWithUpdatedSlots.slots[slot]).configEpoch >
949                                        msg.configEpoch
950                          IN  ClusterLookupNode(clusterWithUpdatedSlots,
951                                                clusterWithUpdatedSlots.slots[slotId])
952                     ELSE NULL_ClusterNodeType

954               updatedSender ≜ ClusterLookupNode(clusterWithUpdatedSlots, sender.name)
```

```
956                          updateMessage ≜ IF nodeToSendUpdate ≠ NULL_ClusterNodeType
957                                          THEN ⟨CreateUpdateMessage(clusterWithUpdatedSlots,
958                                                                    updatedSender.name,
959                                                                    nodeToSendUpdate)⟩
960                                          ELSE ⟨⟩
961                      IN  ⟨clusterWithUpdatedSlots, updatedSender, updateMessage⟩
962              ELSE ⟨cluster, sender, ⟨⟩⟩

964  ProcessPingPongMeetMsg(cluster, msg) ≜
965     LET sender ≜ ClusterLookupNode(cluster, msg.sender)
966         isKnownSender ≜ sender ≠ NULL_ClusterNodeType
967     IN  IF sender = NULL_ClusterNodeType
968          THEN ⟨cluster, ⟨⟩⟩
969          ELSE LET
970          updatedSenderIfHandShake ≜
971                   IF "HANDSHAKE" ∈ sender.flags
972                    THEN LET typeFlag ≜ IF "MASTER" ∈ msg.flags
973                                         THEN {"MASTER"}
974                                         ELSE {"SLAVE"}
975                          IN  [sender EXCEPT !.flags = ((@ \ {"HANDSHAKE"}) ∪ typeFlag)]
976                    ELSE sender

978            Update the node address if it changed
979          updatedSenderAddressInfo ≜
980                   IF ∧ msg.type = "PING"
981                      ∧ "HANDSHAKE" ∉ updatedSenderIfHandShake.flags
982                      ∧ msg.hasSenderAddressChanged
983                    THEN ⟨[updatedSenderIfHandShake EXCEPT !.hasLink = FALSE,
984                                                          !.hasAddressChanged = FALSE],
985                          TRUE⟩
986                    ELSE ⟨updatedSenderIfHandShake, FALSE⟩
987          needToUpdateClusterState ≜ updatedSenderAddressInfo[2]

989            The PFAIL condition can be reversed without external
990            help if it is momentary (that is, if it does not
991            turn into a FAIL state).
992
993            The FAIL condition is also reversible under specific
994            conditions detected by clearNodeFailureIfNeeded().
995          updatedSenderInfo ≜
996                   IF msg.type = "PONG"
997                    THEN LET senderWithUpdatedPingState ≜
998                               [updatedSenderAddressInfo[1] EXCEPT
999                                         !.pingSent = FALSE,
1000                                        !.pingPongHalfTimeOut = FALSE,
1001                                        !.pingPongTimeOut = FALSE,
1002                                        !.needsToBePinged  = FALSE]
1003                         IN  IF "PFAIL" ∈ senderWithUpdatedPingState.flags
1004                              THEN ⟨[senderWithUpdatedPingState EXCEPT
1005                                          !.flags = @ \ {"PFAIL"}],
1006                                    FALSE⟩
1007                              ELSE IF "FAIL" ∈ senderWithUpdatedPingState.flags
1008                                    THEN ClearNodeFailureIfNeeded(senderWithUpdatedPingState)
1009                                    ELSE ⟨senderWithUpdatedPingState, FALSE⟩
1010                              ELSE ⟨updatedSenderAddressInfo[1], FALSE⟩
1011         needToUpdateClusterStateAfterFailureProcessing ≜ updatedSenderInfo[2]

1013           Check for role switch: slave → master or master → slave.
1014         proccesedRoleChanges ≜ ProcessRoleChanges(cluster, updatedSenderInfo[1], msg)
```

```
1016          updatedCluster ≜ [proccesedRoleChanges[1]
1017                            EXCEPT !.todoBeforeSleep =
1018                                    IF ∧ needToUpdateClusterState
1019                                       ∧ needToUpdateClusterStateAfterFailureProcessing
1020                                    THEN @ ∪ {"UPDATE_STATE"}
1021                                    ELSE @]
1022          updatedSenderRole ≜ proccesedRoleChanges[2]

1024           Update our info about server slots
1025          updateInfo ≜ UpdateInfoAboutServerSlots(updatedCluster, updatedSenderRole, msg)
1026          updateClusterWithSlotInfo ≜ updateInfo[1]
1027          updatedSenderWithSlotInfo ≜ updateInfo[2]
1028          msgToSend ≜ updateInfo[3]

1030           If our config epoch collides with the sender's try to fix the problem
1031          myself ≜ ClusterLookupNode(updateClusterWithSlotInfo, updateClusterWithSlotInfo.myself)
1032          updatedClusterConfig ≜ IF ∧ "MASTER" ∈ myself.flags
1033                                    ∧ "MASTER" ∈ updatedSenderWithSlotInfo.flags
1034                                    ∧ msg.configEpoch = myself.configEpoch
1035                                 THEN ClusterHandleConfigEpochCollision(updateClusterWithSlotInfo,
1036                                                                      updatedSenderWithSlotInfo)
1037                                 ELSE updateClusterWithSlotInfo

1039           Get info from the gossip section
1040          proccessedGossipInfo ≜ ClusterProcessGossipSection(updatedClusterConfig, msg)

1042           Bug 1 fix start
1043           Update info about the migrating slots in the cluster
1044           Accept info only from the node who is responsible for that slot
1045          clusterWithUpdatedMigratingSlots ≜
1046             [proccessedGossipInfo[1] EXCEPT
1047                 !.migratingSlotsTo = [slot ∈ Slots ↦ IF ∧ "MASTER" ∈ updatedSenderWithSlotInfo.flags
1048                                                         ∧ updatedSenderWithSlotInfo.slots[slot] = TRUE
1049                                                      THEN msg.migratingSlotsTo[slot]
1050                                                      ELSE proccessedGossipInfo[1].migratingSlotsTo[slot]]]
1051           Bug 1 fix end
1052      IN ⟨clusterWithUpdatedMigratingSlots, (msgToSend ∘ proccessedGossipInfo[2])⟩

1054 ProcessPingMsg(cluster, msg, sender) ≜
1055      LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
1056          msgsToSend ≜ ⟨ClusterCreatePongMsg(cluster, myself, sender)⟩

1058          processedMsgInMoreDetail ≜ ProcessPingPongMeetMsg(cluster, msg)
1059       IN ⟨processedMsgInMoreDetail[1],
1060           msgsToSend ∘ processedMsgInMoreDetail[2]⟩

1062 ProcessPongMsg(cluster, msg, sender) ≜
1063      LET myself ≜ ClusterLookupNode(cluster, cluster.myself)
1064          msgsToSend ≜ ⟨⟩

1066          processedMsgInMoreDetail ≜ ProcessPingPongMeetMsg(cluster, msg)

1068          allMsgsToSend ≜ msgsToSend ∘ processedMsgInMoreDetail[2]

1070          finalClusterInfo ≜ processedMsgInMoreDetail[1]
1071       IN ⟨finalClusterInfo,
1072           allMsgsToSend⟩

1074 ProcessMeetMsg(cluster, msg, sender) ≜
1075      LET isKnownSender ≜ sender ≠ NULL_ClusterNodeType
1076      IN IF isKnownSender
1077          THEN LET
```

```
1078                    myself ≜ ClusterLookupNode(cluster, cluster.myself)
1079                    pongMsg ≜ ⟨ClusterCreatePongMsg(cluster,
1080                                                    myself,
1081                                                    sender)⟩

1083                    processedMsgInMoreDetail ≜ ProcessPingPongMeetMsg(cluster, msg)
1084                    IN ⟨processedMsgInMoreDetail[1],
1085                        pongMsg ∘ processedMsgInMoreDetail[2]⟩
1086                ELSE LET
1087                    updatedClusterWithNode ≜
1088                        ClusterAddNode(cluster, CreateClusterNode(msg.sender, "HANDSHAKE"))
1089                    processedGossipInfoResult ≜ ClusterProcessGossipSection(updatedClusterWithNode, msg)

1091                    myself ≜ ClusterLookupNode(processedGossipInfoResult[1], cluster.myself)
1092                    pongMsg ≜ ⟨ClusterCreatePongMsg(processedGossipInfoResult[1],
1093                                                    myself,
1094                                                    sender)⟩

1096                    processedMsgInMoreDetail ≜ ProcessPingPongMeetMsg(processedGossipInfoResult[1], msg)
1097                    IN ⟨processedMsgInMoreDetail[1],
1098                        pongMsg ∘ processedGossipInfoResult[2] ∘ processedMsgInMoreDetail[2]⟩

1100 ProcessFailMsg(cluster, msg, sender) ≜
1101     LET isKnownSender ≜ sender ≠ NULL_ClusterNodeType
1102         failingNode ≜ ClusterLookupNode(cluster, msg.data.nodeName)
1103     IN IF ∧ isKnownSender
1104           ∧ failingNode ≠ NULL_ClusterNodeType
1105           ∧ "FAIL" ∉ failingNode.flags
1106           ∧ "MYSELF" ∉ failingNode.flags
1107        THEN ⟨[cluster EXCEPT
1108                   !.nodes =
1109                       [@ EXCEPT ![failingNode.name] =
1110                                 [@ EXCEPT !.flags = (@ \ {"PFAIL"}) ∪ {"FAIL"}]],
1111                   !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}],
1112              ⟨⟩⟩
1113        ELSE ⟨cluster,
1114              ⟨⟩⟩

1116 ProcessUpdateMsg(cluster, msg, sender) ≜
1117     LET isKnownSender ≜ sender ≠ NULL_ClusterNodeType
1118         nodeToUpdate ≜ ClusterLookupNode(cluster, msg.data.nodeName)
1119         reportedConfigEpoch ≜ msg.data.configEpoch
1120     IN IF ∧ isKnownSender
1121           ∧ nodeToUpdate ≠ NULL_ClusterNodeType
1122           ∧ nodeToUpdate.configEpoch < reportedConfigEpoch
1123        THEN LET If in our current config the node is a slave, set it as a master.
1124                 clusterWithAddedMaster ≜ IF "SLAVE" ∈ nodeToUpdate.flags
1125                                          THEN ClusterSetNodeAsMaster(cluster, nodeToUpdate)
1126                                          ELSE ⟨cluster, nodeToUpdate⟩

1128                 updatedClusterInfo ≜
1129                     [clusterWithAddedMaster EXCEPT
1130                         !.nodes = [@ EXCEPT
1131                                     !.configEpoch = reportedConfigEpoch]]

1133                 updatedNodeToUpdate ≜ ClusterLookupNode(updatedClusterInfo, msg.data.nodeName)
1134                   clusterWithUpdatedBitmap ≜ ClusterUpdateSlotsConfigWith(updatedClusterInfo,
1135                                                                            updatedNodeToUpdate,
1136                                                                            reportedConfigEpoch,
1137                                                                            msg.data.slots)
```

```
1138                     IN ⟨clusterWithUpdatedBitmap,
1139                         ⟨⟩⟩
1140              ELSE ⟨cluster,
1141                     ⟨⟩⟩

1143 ProcessMFStartMsg(cluster, msg, sender) ≜
1144     IF ∧ sender ≠ NULL_ClusterNodeType
1145        ∧ sender.slaveOf = cluster.myself
1146     THEN LET
1147             resetManualFailover ≜ [ResetManualFailover(cluster) EXCEPT
1148                                      !.mfStarted = TRUE,
1149                                      !.mfSlave = sender.name]
1150             IN ⟨resetManualFailover,
1151                 ⟨⟩⟩
1152     ELSE ⟨cluster,
1153           ⟨⟩⟩

1155 ClusterProcessMsg(cluster, msg) ≜
1156     LET sender ≜ ClusterLookupNode(cluster, msg.sender)
1157         myself ≜ ClusterLookupNode(cluster, cluster.myself)
1158         isKnownSender ≜ sender ≠ NULL_ClusterNodeType

1160          Update our curretEpoch if we see a newer epoch in the cluster.
1161         updateCurrentEpoch ≜
1162             ∧ isKnownSender
1163             ∧ "HANDSHAKE" ∉ sender.flags
1164             ∧ msg.currentEpoch > cluster.currentEpoch

1166          Update the sender configEpoch if it is publishing a newer one.
1167         updateSenderConfigEpoch ≜
1168             ∧ isKnownSender
1169             ∧ "HANDSHAKE" ∉ sender.flags
1170             ∧ msg.configEpoch > sender.configEpoch

1172          Update the replication offset info for this node.
1173          If we are a slave performing a manual failover and our master
1174          sent its offset while already paused, populate the MF state
1175         isMFStarted ≜
1176             ∧ cluster.mfStarted
1177             ∧ "SLAVE" ∈ myself.flags
1178             ∧ myself.slaveOf = sender.name
1179             ∧ "PAUSED" ∈ msg.msgFlags
1180             ∧ ¬cluster.mfMasterOffsetSet

1182         updatedClusterInfo ≜
1183             [cluster EXCEPT
1184                 !.nodes = [@ EXCEPT
1185                               ![sender.name] = [@ EXCEPT
1186                                                    !.configEpoch = IF updateSenderConfigEpoch
1187                                                                    THEN msg.configEpoch
1188                                                                    ELSE @]],
1189                 !.currentEpoch = IF updateCurrentEpoch THEN msg.currentEpoch ELSE @,

1191                 !.mfMasterOffsetSet = IF isMFStarted THEN TRUE ELSE @]

1193         updatedSender ≜ ClusterLookupNode(updatedClusterInfo, msg.sender)
1194     IN CASE msg.type = "PING" → ProcessPingMsg(updatedClusterInfo, msg, updatedSender)
1195         □msg.type = "PONG" → ProcessPongMsg(updatedClusterInfo, msg, updatedSender)
1196         □msg.type = "MEET" → ProcessMeetMsg(updatedClusterInfo, msg, updatedSender)
1197         □msg.type = "FAIL"   → ProcessFailMsg(updatedClusterInfo, msg, updatedSender)
1198         □msg.type = "UPDATE" → ProcessUpdateMsg(updatedClusterInfo, msg, updatedSender)
```

```
1199            □msg.type = "MFSTART" → ProcessMFStartMsg(updatedClusterInfo, msg, updatedSender)

     Cluster before sleep
1204 ClusterBeforeSleep(cluster) ≜
1205     LET updatedClusterAfterFailover ≜ IF "HANDLE_FAILOVER" ∈ cluster.todoBeforeSleep
1206                                        THEN ClusterHandleSlaveFailover(cluster)
1207                                        ELSE ⟨cluster,
1208                                              ⟨⟩⟩

1210         updatedClusterState ≜ IF "UPDATE_STATE" ∈ updatedClusterAfterFailover[1].todoBeforeSleep
1211                                THEN ClusterUpdateState(updatedClusterAfterFailover[1])
1212                                ELSE updatedClusterAfterFailover[1]
1213          reset flags
1214         clusterWithResetFlags ≜ [updatedClusterState EXCEPT !.todoBeforeSleep = {}]
1215     IN ⟨clusterWithResetFlags,
1216          updatedClusterAfterFailover[2]⟩

     Cluster event loop actions
1221 ───────────────── Reading msgs ─────────────────
1222 FilterMsgsToSendForNode(msgsToSend, nodeName) ≜
1223     LET F[i ∈ 0 . . Len(msgsToSend)] ≜
1224             IF i = 0
1225             THEN ⟨⟩
1226             ELSE IF msgsToSend[i].reciever = nodeName
1227                  THEN Append(F[i − 1], msgsToSend[i].msg)
1228             ELSE F[i − 1]
1229     IN F[Len(msgsToSend)]

1231 UpdateBufferForNode(nodeName, buffer, msgsToSend) ≜
1232     LET msgsToAdd ≜ FilterMsgsToSendForNode(msgsToSend, nodeName)
1233     IN buffer ∘ msgsToAdd

1235 UpdateSenderBuffer(nodeSendBuffer, msgsToSend) ≜
1236     [node ∈ NODE ↦ UpdateBufferForNode(node, nodeSendBuffer[node], msgsToSend)]

1238 ProcessMultipleMsgs[msgsToProcess ∈ Seq(MessageType),
1239                     cluster ∈ ClusterStateType,
1240                     msgsToSend ∈ Seq(MessageToSendType)] ≜
1241     IF Len(msgsToProcess) = 0
1242      THEN ⟨cluster,
1243            msgsToSend⟩
1244      ELSE LET msgToProcess ≜ Head(msgsToProcess)
1245                processedMsg ≜ ClusterProcessMsg(cluster,
1246                                                 msgToProcess)
1247           IN ProcessMultipleMsgs[Tail(msgsToProcess),
1248                                  processedMsg[1],
1249                                  msgsToSend ∘ processedMsg[2]]

1251 ClusterReadHandler(currentNode, linkNode) ≜
1252     LET processedMsgs ≜ ProcessMultipleMsgs[recieveBuffer[currentNode][linkNode],
1253                                             clusterState[currentNode],
1254                                             ⟨⟩]

1256         updatedNodeSendBuffer ≜ UpdateSenderBuffer(sendBuffer[currentNode],
1257                                                    processedMsgs[2])

1259         updatedEventLoop ≜ [eventLoop[currentNode] EXCEPT
1260                                 !.currentEventBeingProcessed =
1261                                     [@ EXCEPT !.hasReadEvent = FALSE]]
1262     IN ∧ clusterState′ = [clusterState EXCEPT ![currentNode] = processedMsgs[1]]
```

```
1263          ∧ recieveBuffer' = [recieveBuffer EXCEPT ![currentNode][linkNode] = ⟨⟩]
1264          ∧ sendBuffer' = [sendBuffer EXCEPT ![currentNode] = updatedNodeSendBuffer]
1265          ∧ eventLoop' = [eventLoop EXCEPT ![currentNode] = updatedEventLoop]

1267   ─────────────────── Writing msgs ───────────────────
1268 ClusterWriteHandler(currentNode, linkNode) ≜
1269     LET updatedEventLoop ≜ [eventLoop[currentNode] EXCEPT
1270                                  !.currentEventBeingProcessed =
1271                                      [@ EXCEPT !.hasWriteEvents = FALSE]]
1272     IN   ∧ recieveBuffer' = [recieveBuffer EXCEPT ![linkNode][currentNode] =
1273                                  @ ∘ sendBuffer[currentNode][linkNode]]
1274          ∧ sendBuffer' = [sendBuffer EXCEPT ![currentNode][linkNode] = ⟨⟩]
1275          ∧ eventLoop' = [eventLoop EXCEPT ![currentNode] = updatedEventLoop]
1276          ∧ UNCHANGED ⟨clusterState⟩

1278   ─────────────────── Cluster time events ───────────────────
1279 ClusterTimeEvent(currentNode) ≜
1280     LET updatedInfo ≜ ClusterCronJob(clusterState[currentNode])
1281         updatedNodeSendBuffer ≜ UpdateSenderBuffer(sendBuffer[currentNode],
1282                                                    updatedInfo[2])
1283     IN   ∧ clusterState' = [clusterState EXCEPT ![currentNode] = updatedInfo[1]]
1284          ∧ sendBuffer' = [sendBuffer EXCEPT ![currentNode] = updatedNodeSendBuffer]
1285          ∧ UNCHANGED ⟨recieveBuffer⟩

1287   ─────────────────── Cluster before sleep ───────────────────
1288 ClusterBeforeSleepHandle(currentNode) ≜
1289     LET updatedInfo ≜ ClusterBeforeSleep(clusterState[currentNode])
1290         updatedNodeSendBuffer ≜ UpdateSenderBuffer(sendBuffer[currentNode],
1291                                                    updatedInfo[2])
1292     IN   ∧ clusterState' = [clusterState EXCEPT ![currentNode] = updatedInfo[1]]
1293          ∧ sendBuffer' = [sendBuffer EXCEPT ![currentNode] = updatedNodeSendBuffer]
1294          ∧ UNCHANGED ⟨recieveBuffer⟩

     Event loop processing
1299 CreateEvent(currentNode, nodeToCreateEvent) ≜
1300     LET hasMsgsToWrite ≜ Len(sendBuffer[currentNode][nodeToCreateEvent]) ≠ 0
1301         hasMsgsToRead ≜ Len(recieveBuffer[currentNode][nodeToCreateEvent]) ≠ 0
1302     IN  IF ∨ hasMsgsToWrite
1303            ∨ hasMsgsToRead
1304          THEN [
1305                name ↦ nodeToCreateEvent,
1306                hasWriteEvents ↦ hasMsgsToWrite,
1307                hasReadEvent ↦ hasMsgsToRead
1308                ]
1309          ELSE NULL_EventLink

1311 BeforeSleep(n) ≜
1312     IF eventLoop[n].state = "BEFORE_SLEEP"
1313      THEN LET eventsBeingProcessed ≜ [node ∈ NODE ↦ CreateEvent(n, node)]

1315                updatedEventLoop ≜ [eventLoop[n] EXCEPT
1316                                     !.state = "PROCESSING_FILE_EVENTS",
1317                                     !.eventsBeingProcessed = eventsBeingProcessed,
1318                                     !.currentEventBeingProcessed = NULL_EventLink]
1319            IN   ∧ ClusterBeforeSleepHandle(n)
1320                 ∧ eventLoop' = [eventLoop EXCEPT ![n] = updatedEventLoop]
1321      ELSE UNCHANGED ⟨clusterState, eventLoop, recieveBuffer, sendBuffer⟩

1323 ChooseEventToProcess(n) ≜
1324     ∧ eventLoop[n].state = "PROCESSING_FILE_EVENTS"
```

```
1325      ∧ eventLoop[n].currentEventBeingProcessed = NULL_EventLink
1326      ∧ IF ∃ node ∈ NODE : eventLoop[n].eventsBeingProcessed[node] ≠ NULL_EventLink
1327         THEN LET nodeEventsToProcess ≜ CHOOSE node ∈ NODE :
1328                         eventLoop[n].eventsBeingProcessed[node] ≠ NULL_EventLink
1329                     updatedEventLoop ≜
1330                         [eventLoop[n] EXCEPT
1331                             !.currentEventBeingProcessed =
1332                                     eventLoop[n].eventsBeingProcessed[nodeEventsToProcess]]
1333                 IN   ∧ eventLoop' = [eventLoop EXCEPT ![n] = updatedEventLoop]
1334                      ∧ UNCHANGED ⟨clusterState, recieveBuffer, sendBuffer⟩
1335         ELSE  LET updatedEventLoop ≜ [eventLoop[n] EXCEPT
1336                                           !.state = IF eventLoop[n].timeEvent.hasOccured
1337                                                         THEN "PROCESSING_TIME_EVENTS"
1338                                                         ELSE  "BEFORE_SLEEP",
1339                                           !.eventsBeingProcessed = [node ∈ NODE ↦ NULL_EventLink]]
1340                 IN   ∧ eventLoop' = [eventLoop EXCEPT ![n] = updatedEventLoop]
1341                      ∧ UNCHANGED ⟨clusterState, recieveBuffer, sendBuffer⟩

1343 ProcessCurrentElement(n) ≜
1344      ∧ eventLoop[n].state = "PROCESSING_FILE_EVENTS"
1345      ∧ eventLoop[n].currentEventBeingProcessed ≠ NULL_EventLink
1346       Due to used AE_BARRIER firstly execute write event after that read
1347      ∧ IF eventLoop[n].currentEventBeingProcessed.hasWriteEvents
1348         THEN ClusterWriteHandler(n, eventLoop[n].currentEventBeingProcessed.name)
1349         ELSE  IF eventLoop[n].currentEventBeingProcessed.hasReadEvent
1350                 THEN ClusterReadHandler(n, eventLoop[n].currentEventBeingProcessed.name)
1351                 ELSE  LET updatedEventLoop ≜
1352                             [eventLoop[n] EXCEPT
1353                                 !.currentEventBeingProcessed = NULL_EventLink,
1354                                 !.eventsBeingProcessed =
1355                                     [@ EXCEPT
1356                                         ![eventLoop[n].currentEventBeingProcessed.name] =
1357                                                 NULL_EventLink]]
1358                         IN   ∧ eventLoop' = [eventLoop EXCEPT ![n] = updatedEventLoop]
1359                              ∧ UNCHANGED ⟨clusterState, recieveBuffer, sendBuffer⟩

1361 ProcessFileEvents(n) ≜
1362      ∨ ChooseEventToProcess(n)
1363      ∨ ProcessCurrentElement(n)

1365 ProcessTimeEvents(n) ≜
1366      ∧ eventLoop[n].state = "PROCESSING_TIME_EVENTS"
1367      ∧ LET updatedEventLoop ≜ [eventLoop[n] EXCEPT !.timeEvent.hasOccured = FALSE,
1368                                                     !.state = "BEFORE_SLEEP"]
1369         IN   ∧ ClusterTimeEvent(n)
1370              ∧ eventLoop' = [eventLoop EXCEPT ![n] = updatedEventLoop]

1372 EventLoopMain(n) ≜
1373      ∨ BeforeSleep(n)
1374      ∨ ProcessFileEvents(n)
1375      ∨ ProcessTimeEvents(n)

       Time actions

1380 CronJobTime(n) ≜
1381      LET updatedEventLoop ≜ [eventLoop[n] EXCEPT !.timeEvent.hasOccured = TRUE,
1382                                                  !.timeEvent.iteration = @ + 1]
1383      IN   ∧ ¬eventLoop[n].timeEvent.hasOccured
1384           ∧ eventLoop' = [eventLoop EXCEPT ![n] = updatedEventLoop]
1385           ∧ UNCHANGED ⟨clusterState, recieveBuffer, sendBuffer⟩
```

```
1387   If we are waiting for the PONG more than half the cluster
1388   timeout, reconnect the link: maybe there is a connection
1389   issue even if the node is alive.
1390 PingPongHalfTimeout(n) ≜
1391     ∃ m ∈ NODE :
1392       LET node ≜ ClusterLookupNode(clusterState[n], m)
1393       IN   ∧ node ≠ NULL_ClusterNodeType
1394            ∧ node.pingSent
1395            ∧ ¬node.pingPongHalfTimeOut
1396            ∧ clusterState′ =
1397                 [clusterState EXCEPT ![n] =
1398                     [@ EXCEPT !.nodes = [@ EXCEPT ![node.name] =
1399                                            [node EXCEPT !.pingPongHalfTimeOut = TRUE]]]]
1400            ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1402   PING was sent and we have not got response more than cluster node time out.
1403   This causes node to be marked as failing
1404 PingPongTimeout(n) ≜
1405     ∃ m ∈ NODE :
1406       LET node ≜ ClusterLookupNode(clusterState[n], m)
1407       IN   ∧ node ≠ NULL_ClusterNodeType
1408            ∧ node.pingSent
1409            ∧ node.pingPongHalfTimeOut
1410            ∧ ¬node.pingPongTimeOut
1411            ∧ clusterState′ =
1412                 [clusterState EXCEPT ![n] =
1413                     [@ EXCEPT !.nodes = [@ EXCEPT ![node.name] =
1414                                            [node EXCEPT !.pingPongTimeOut = TRUE]]]]
1415            ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1417   If we have currently no active ping in this instance, and the
1418   received PONG is older than half the cluster timeout, send
1419   a new ping now, to ensure all the nodes are pinged without
1420   a too big delay.
1421 PingNotSentAndOldRecievedPong(n) ≜
1422     ∃ m ∈ NODE :
1423       LET node ≜ ClusterLookupNode(clusterState[n], m)
1424       IN   ∧ node ≠ NULL_ClusterNodeType
1425            ∧ ¬node.pingSent
1426            ∧ ¬node.needsToBePinged
1427            ∧ clusterState′ =
1428                 [clusterState EXCEPT ![n] =
1429                     [@ EXCEPT !.nodes = [@ EXCEPT ![node.name] =
1430                                            [node EXCEPT !.needsToBePinged = TRUE]]]]
1431            ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1433   If node has started hanshake process however hasn't
1434   continued communication for more than time out
1435 NoAnswerAfterHandshake(n) ≜
1436     ∃ m ∈ NODE :
1437       LET node ≜ ClusterLookupNode(clusterState[n], m)
1438       IN   ∧ node ≠ NULL_ClusterNodeType
1439            ∧ "HANDSHAKE" ∈ node.flags
1440            ∧ ¬node.handshakeTimeOut
1441            ∧ clusterState′ =
1442                 [clusterState EXCEPT ![n] =
1443                     [@ EXCEPT !.nodes = [@ EXCEPT ![node.name] =
1444                                            [node EXCEPT !.handshakeTimeOut = TRUE]]]]
1445            ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩
```

```
1447  If node was in FAIL state for too long
1448 FailTimeOut(n) ≜
1449     ∃ m ∈ NODE :
1450       LET node ≜ ClusterLookupNode(clusterState[n], m)
1451       IN   ∧ node ≠ NULL_ClusterNodeType
1452            ∧ "FAIL" ∈ node.flags
1453            ∧ ¬node.failTimeOut
1454            ∧ clusterState' =
1455                  [clusterState EXCEPT ![n] =
1456                      [@ EXCEPT !.nodes = [@ EXCEPT ![node.name] =
1457                                                  [node EXCEPT !.failTimeOut = TRUE]]]]
1458            ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1460  If manual failover is taking for too long
1461 MFTimeOut(n) ≜
1462     ∧ clusterState[n].mfStarted
1463     ∧ ¬clusterState[n].mfEnded
1464     ∧ clusterState' = [clusterState EXCEPT ![n] =
1465                             [@ EXCEPT !.mfEnded = TRUE]]
1466     ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1468 TimeActions(n) ≜
1469     ∨ CronJobTime(n)
1470     ∨ PingPongHalfTimeout(n)
1471     ∨ PingPongTimeout(n)
1472     ∨ PingNotSentAndOldRecievedPong(n)
1473     ∨ NoAnswerAfterHandshake(n)
1474     ∨ FailTimeOut(n)
1475     ∨ MFTimeOut(n)

      Node actions
1480 ChangeAddress(n) ≜
1481   LET node ≜ ClusterLookupNode(clusterState[n], n)
1482   IN   ∧ node ≠ NULL_ClusterNodeType
1483        ∧ ¬node.hasAddressChanged
1484        ∧ clusterState' = [clusterState EXCEPT ![n] =
1485                              [@ EXCEPT !.nodes = [@ EXCEPT ![node.name] =
1486                                                      [node EXCEPT !.hasAddressChanged = TRUE]]]]
1487        ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

      Client commands
1492 CommandsExecutionPredicate(n) ≜
1493      Client command could happen anytime during proccessing file events (socket requests)
1494     ∧ eventLoop[n].state = "PROCESSING_FILE_EVENTS"
1495     ∧ eventLoop[n].currentEventBeingProcessed = NULL_EventLink

1497  MEET < ip > < port> [bus-port] – Connect nodes into a working cluster.
1498 MeetCommand(n) ≜
1499     ∧ CommandsExecutionPredicate(n)
1500     ∧ ∃ m ∈ NODE :
1501         ∧ n ≠ m
1502         ∧ clusterState[n].nodes[m] = NULL_ClusterNodeType
1503         ∧ clusterState' = [clusterState EXCEPT ![n] =
1504               ClusterAddNode(clusterState[n], CreateClusterNode(m, {"HANDSHAKE", "MEET"}))]
1505         ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1507  FLUSHSLOTS – Delete current node own slots information.
1508  It can only be called when the database is empty. We assume that it is empty.
1509 FlushSlotsCommand(n) ≜
```

```
1510        ∧ CommandsExecutionPredicate(n)
1511        ∧ clusterState' =
1512              [clusterState EXCEPT
1513                   ![n] = [@ EXCEPT
1514                             !.nodes = [@ EXCEPT
1515                                          ![clusterState[n].myself] = [@ EXCEPT
1516                                                                          !.slots = [slot ∈ Slots ↦ FALSE]]],
1517                             !.slots  = [slot ∈ Slots ↦ IF @[slot] = clusterState[n].myself
1518                                                        THEN NULL_NODE
1519                                                        ELSE  @[slot]],
1520                             !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]]
1521        ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1523    BUMPEPOCH – Advance the cluster config epoch.
1524  ClusterGetMaxEpoch(cluster) ≜
1525     LET allConfigEpoch ≜
1526            {cluster.nodes[m].configEpoch
1527                : m ∈ {x ∈ NODE : cluster.nodes[x] ≠ NULL_ClusterNodeType}} ∪ {0}

1529          greatestConfigEpoch ≜
1530              CHOOSE max ∈ allConfigEpoch : (∀ epoch ∈ allConfigEpoch : max ≥ epoch)

1532          maxEpoch ≜ IF greatestConfigEpoch < cluster.currentEpoch
1533                       THEN cluster.currentEpoch
1534                       ELSE  greatestConfigEpoch
1535     IN   maxEpoch

1537  ClusterBumpConfigEpochWithoutConsensus(cluster) ≜
1538      LET max ≜ ClusterGetMaxEpoch(cluster)

1540           myself ≜ ClusterLookupNode(cluster, cluster.myself)

1542           updatedCluster ≜ IF ∨ myself.configEpoch = 0
1543                                 ∨ myself.configEpoch ≠ max
1544                              THEN [cluster EXCEPT !.currentEpoch = @ + 1,
1545                                                   !.nodes = [@ EXCEPT ![myself.name] =
1546                                                                [@ EXCEPT !.configEpoch =
1547                                                                   cluster.currentEpoch + 1]]]
1548                              ELSE  cluster
1549      IN   updatedCluster

1551  BumpEpochCommand(n) ≜
1552        ∧ CommandsExecutionPredicate(n)
1553        ∧ LET max ≜ ClusterGetMaxEpoch(clusterState[n])
1554              myself ≜ ClusterLookupNode(clusterState[n], clusterState[n].myself)
1555          IN   ∨ myself.configEpoch = 0
1556               ∨ myself.configEpoch ≠ max
1557        ∧ clusterState' = [clusterState EXCEPT ![n] =
1558                               ClusterBumpConfigEpochWithoutConsensus(@)]
1559        ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1561    ADDSLOTS < slot >  [slot ...] – Assign slots to current node.
1562    DELSLOTS < slot >  [slot ...] – Delete slots information from current node.
1563  AddDelSlotsCmd(n, isDelete) ≜
1564      ∃ slotToChange ∈ Slots :
1565          ∧ IF isDelete
1566              THEN clusterState[n].slots[slotToChange] ≠ NULL_NODE
1567              ELSE  clusterState[n].slots[slotToChange] = NULL_NODE
1568          ∧ clusterState' = [clusterState EXCEPT
1569                                ![n] = [@ EXCEPT
1570                                          !.slots = [@ EXCEPT ![slotToChange] =
```

```
1571                                                      IF isDelete
1572                                                       THEN NULL_NODE
1573                                                       ELSE clusterState[n].myself],
1574                                  !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"},
1575                                  !.nodes = [@ EXCEPT ![clusterState[n].myself] =
1576                                              [@ EXCEPT !.slots =
1577                                                [@ EXCEPT ![slotToChange] = IF isDelete
1578                                                                              THEN FALSE
1579                                                                              ELSE TRUE]]],
1580                                  !.importingSlotsFrom = [@ EXCEPT ![slotToChange] = NULL_NODE]
1581                                  ]]
1582        ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1584 AddSlotsCommand(n) ≜
1585     ∧ CommandsExecutionPredicate(n)
1586     ∧ AddDelSlotsCmd(n, FALSE)

1588 DelSlotsCommand(n) ≜
1589     ∧ CommandsExecutionPredicate(n)
1590     ∧ AddDelSlotsCmd(n, TRUE)

1592   "SETSLOT < slot > (importing | migrating | stable | node < node-id> ) – Set slot state.
1593 SetSlotsExecutionPredicate(n) ≜
1594     ∧ CommandsExecutionPredicate(n)
1595     ∧ "SLAVE" ∉ clusterState[n].nodes[n].flags

1597 SetSlotMigrating(n) ≜
1598     ∧ SetSlotsExecutionPredicate(n)
1599     ∧ ∃ slot ∈ Slots :
1600         ∧ clusterState[n].slots[slot] = n
1601         ∧ ∃ destNode ∈ NODE :
1602             ∧ clusterState[n].nodes[destNode] ≠ NULL_ClusterNodeType
1603             ∧ clusterState[n].migratingSlotsTo[slot] ≠ destNode
1604             ∧ clusterState' = [clusterState EXCEPT ![n] =
1605                                 [@ EXCEPT !.migratingSlotsTo =
1606                                             [@ EXCEPT ![slot] = destNode],
1607                                       !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]]
1608     ∧ UNCHANGED ⟨sendBuffer, eventLoop, recieveBuffer⟩

1610 SetSlotMigratingWithBug2Fix(n) ≜
1611     ∧ SetSlotsExecutionPredicate(n)
1612     ∧ ∃ slot ∈ Slots :
1613         ∧ clusterState[n].slots[slot] = n
1614         ∧ ∃ destNode ∈ NODE :
1615              Bug 2 fix start
1616             ∧ n ≠ destNode
1617              Bug 2 fix end
1618             ∧ clusterState[n].nodes[destNode] ≠ NULL_ClusterNodeType
1619             ∧ clusterState[n].migratingSlotsTo[slot] ≠ destNode
1620             ∧ clusterState' = [clusterState EXCEPT ![n] =
1621                                 [@ EXCEPT !.migratingSlotsTo =
1622                                             [@ EXCEPT ![slot] = destNode],
1623                                       !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]]
1624     ∧ UNCHANGED ⟨sendBuffer, eventLoop, recieveBuffer⟩

1626 SetSlotImporting(n) ≜
1627     ∧ SetSlotsExecutionPredicate(n)
1628     ∧ ∃ slot ∈ Slots :
1629        ∧ clusterState[n].slots[slot] ≠ n
1630        ∧ ∃ srcNode ∈ NODE :
```

```
1631              ∧ clusterState[n].nodes[srcNode] ≠ NULL_ClusterNodeType
1632              ∧ clusterState[n].importingSlotsFrom[slot] ≠ srcNode
1633              ∧ clusterState' = [clusterState EXCEPT ![n] =
1634                                 [@ EXCEPT !.importingSlotsFrom =
1635                                                [@ EXCEPT ![slot] = srcNode],
1636                                          !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]]
1637       ∧ UNCHANGED ⟨sendBuffer, eventLoop, recieveBuffer⟩

1639 SetSlotImportingWithBug3Fix(n) ≜
1640       ∧ SetSlotsExecutionPredicate(n)
1641       ∧ ∃ slot ∈ Slots :
1642          ∧ clusterState[n].slots[slot] ≠ n
1643          ∧ ∃ srcNode ∈ NODE :
1644               Bug 3 fix start
1645              ∧ n ≠ srcNode
1646               Bug 3 fix end
1647              ∧ clusterState[n].nodes[srcNode] ≠ NULL_ClusterNodeType
1648              ∧ clusterState[n].importingSlotsFrom[slot] ≠ srcNode
1649              ∧ clusterState' = [clusterState EXCEPT ![n] =
1650                                 [@ EXCEPT !.importingSlotsFrom =
1651                                                [@ EXCEPT ![slot] = srcNode],
1652                                          !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]]
1653       ∧ UNCHANGED ⟨sendBuffer, eventLoop, recieveBuffer⟩

1655 SetSlotImportingWithBug4Fix(n) ≜
1656       ∧ SetSlotsExecutionPredicate(n)
1657       ∧ ∃ slot ∈ Slots :
1658          ∧ clusterState[n].slots[slot] ≠ n
1659          ∧ ∃ srcNode ∈ NODE :
1660              ∧ n ≠ srcNode
1661               Bug 4 fix start
1662              ∧ clusterState[n].slots[slot] = srcNode
1663               Bug 4 fix end
1664              ∧ clusterState[n].nodes[srcNode] ≠ NULL_ClusterNodeType
1665              ∧ clusterState[n].importingSlotsFrom[slot] ≠ srcNode
1666              ∧ clusterState' = [clusterState EXCEPT ![n] =
1667                                 [@ EXCEPT !.importingSlotsFrom =
1668                                                [@ EXCEPT ![slot] = srcNode],
1669                                          !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]]
1670       ∧ UNCHANGED ⟨sendBuffer, eventLoop, recieveBuffer⟩

1672 SetSlotImportingWithBug1Fix(n) ≜
1673       ∧ SetSlotsExecutionPredicate(n)
1674       ∧ ∃ slot ∈ Slots :
1675           Bug 1 fix start
1676          ∧ clusterState[n].migratingSlotsTo[slot] = n
1677           Bug 1 fix end
1678          ∧ clusterState[n].slots[slot] ≠ n
1679          ∧ ∃ srcNode ∈ NODE :
1680              ∧ n ≠ srcNode
1681              ∧ clusterState[n].slots[slot] = srcNode
1682              ∧ clusterState[n].nodes[srcNode] ≠ NULL_ClusterNodeType
1683              ∧ clusterState[n].importingSlotsFrom[slot] ≠ srcNode
1684              ∧ clusterState' = [clusterState EXCEPT ![n] =
1685                                 [@ EXCEPT !.importingSlotsFrom =
1686                                                [@ EXCEPT ![slot] = srcNode],
1687                                          !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]]
1688       ∧ UNCHANGED ⟨sendBuffer, eventLoop, recieveBuffer⟩
```

```
1690 SetSlotStable(n) ≜
1691     ∧ SetSlotsExecutionPredicate(n)
1692     ∧ ∃ slot ∈ Slots :
1693         ∧ ∨ clusterState[n].migratingSlotsTo[slot] ≠ NULL_NODE
1694           ∨ clusterState[n].importingSlotsFrom[slot] ≠ NULL_NODE
1695         ∧ clusterState′ = [clusterState EXCEPT ![n] =
1696                             [@ EXCEPT !.importingSlotsFrom =
1697                                          [@ EXCEPT ![slot] = NULL_NODE],
1698                                        !.migratingSlotsTo =
1699                                          [@ EXCEPT ![slot] = NULL_NODE],
1700                                        !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]]
1701     ∧ UNCHANGED ⟨sendBuffer, eventLoop, recieveBuffer⟩

1703 SetSlotNode(n) ≜
1704     ∧ SetSlotsExecutionPredicate(n)
1705     ∧ ∃ slot ∈ Slots :
1706         ∧ ∃ destNode ∈ NODE :
1707             ∧ clusterState[n].nodes[destNode] ≠ NULL_ClusterNodeType
1708             ∧ LET
1709                 cluster ≜ clusterState[n]
1710                 clusterWithUpdatedMigratingSlots ≜
1711                     [cluster EXCEPT !.migratingSlotsTo = [@ EXCEPT ![slot] = NULL_NODE]]

1713                 clusterWithUpdatedImportingSlots ≜
1714                     IF ∧ destNode = n
1715                        ∧ cluster.importingSlotsFrom[slot] ≠ NULL_NODE
1716                     THEN [ClusterBumpConfigEpochWithoutConsensus(clusterWithUpdatedMigratingSlots)
1717                              EXCEPT !.importingSlotsFrom = [@ EXCEPT ![slot] = NULL_NODE]]
1718                     ELSE clusterWithUpdatedMigratingSlots

1720                 previousSlotOwnerName ≜ clusterWithUpdatedImportingSlots.slots[slot]
1721                 updatePreviousSlotOwner ≜
1722                     ∧ previousSlotOwnerName ≠ NULL_NODE
1723                     ∧ clusterWithUpdatedImportingSlots.nodes[previousSlotOwnerName]
1724                           ≠ NULL_ClusterNodeType

1726                 updatedCluster ≜ [clusterWithUpdatedImportingSlots EXCEPT
1727                                     !.nodes =
1728                                         IF updatePreviousSlotOwner
1729                                         THEN [@ EXCEPT
1730                                                  ![destNode] =
1731                                                      [@ EXCEPT !.slots =
1732                                                          [@ EXCEPT ![slot] = TRUE]],
1733                                                  ![previousSlotOwnerName] =
1734                                                      [@ EXCEPT !.slots =
1735                                                          [@ EXCEPT ![slot] = FALSE]]]
1736                                         ELSE [@ EXCEPT
1737                                                  ![destNode] =
1738                                                      [@ EXCEPT !.slots =
1739                                                          [@ EXCEPT ![slot] = TRUE]]],
1740                                     !.slots = [@ EXCEPT ![slot] = destNode],
1741                                     !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]

1743             IN  ∧ clusterState′ = [clusterState EXCEPT ![n] = updatedCluster]
1744                 ∧ sendBuffer′ = sendBuffer
1745     ∧ UNCHANGED ⟨eventLoop, recieveBuffer⟩

1747 SetSlotNodeBug1Fix(n) ≜
1748     ∧ SetSlotsExecutionPredicate(n)
1749     ∧ ∃ slot ∈ Slots :
```

```
1750            Bug 1 fix start
1751          ∧ clusterState[n].importingSlotsFrom[slot] ≠ NULL_NODE
1752          ∧ clusterState[n].nodes[n] ≠ NULL_ClusterNodeType
1753              Bug 1 fix end
1754            ∧ LET
1755                  Bug 1 fix start
1756                destNode ≜ n
1757                  Bug 1 fix end

1759                cluster ≜ clusterState[n]
1760                  Bug 1 fix start
1761                clusterWithUpdatedMigratingSlots ≜ cluster
1762                  Bug 1 fix end

1764                clusterWithUpdatedImportingSlots ≜
1765                       [ClusterBumpConfigEpochWithoutConsensus(clusterWithUpdatedMigratingSlots)
1766                            EXCEPT !.importingSlotsFrom = [@ EXCEPT ![slot] = NULL_NODE]]

1768                previousSlotOwnerName ≜ clusterWithUpdatedImportingSlots.slots[slot]
1769                updatePreviousSlotOwner ≜
1770                     ∧ previousSlotOwnerName ≠ NULL_NODE
1771                     ∧ clusterWithUpdatedImportingSlots.nodes[previousSlotOwnerName]
1772                            ≠ NULL_ClusterNodeType

1774                updatedCluster ≜ [clusterWithUpdatedImportingSlots EXCEPT
1775                                    !.nodes =
1776                                         IF updatePreviousSlotOwner
1777                                         THEN [@ EXCEPT
1778                                                    ![destNode] =
1779                                                        [@ EXCEPT !.slots =
1780                                                            [@ EXCEPT ![slot] = TRUE]],
1781                                                    ![previousSlotOwnerName] =
1782                                                        [@ EXCEPT !.slots =
1783                                                            [@ EXCEPT ![slot] = FALSE]]]
1784                                         ELSE [@ EXCEPT
1785                                                    ![destNode] =
1786                                                        [@ EXCEPT !.slots =
1787                                                            [@ EXCEPT ![slot] = TRUE]]],
1788                                    !.slots = [@ EXCEPT ![slot] = destNode],
1789                                    !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]

1791                  Bug 1 fix start
1792                updatedNodeSendBuffer ≜ UpdateSenderBuffer(sendBuffer[n],
1793                                                     CreateBroadcastPong(updatedCluster))
1794                  Bug 1 fix end

1796              IN  ∧ clusterState′ = [clusterState EXCEPT ![n] = updatedCluster]
1797                    Bug 1 fix start
1798                  ∧ sendBuffer′ = [sendBuffer EXCEPT ![n] = updatedNodeSendBuffer]
1799                    Bug 1 fix end
1800        ∧ UNCHANGED ⟨eventLoop, recieveBuffer⟩

1802  SetSlotCommand(n) ≜
1803        ∨  migrating
1804     SetSlotMigrating(n)
1805       SetSlotMigratingWithBug2Fix(n)
1806        ∨  importing
1807     SetSlotImporting(n)
1808       SetSlotImportingWithBug1Fix(n)
1809        ∨  stable
```

```
1810          SetSlotStable(n)
1811      ∨   node
1812      SetSlotNode(n)
1813          SetSlotNodeBug1Fix(n)

1815   FORGET < node-id>  – Remove a node from the cluster.
1816  ForgetCommand(n) ≜
1817      ∧ CommandsExecutionPredicate(n)
1818      ∧ ∃ m ∈ NODE :
1819          ∧ n ≠ m
1820          ∧ clusterState[n].nodes[m] ≠ NULL_ClusterNodeType
1821          ∧ "SLAVE" ∈ clusterState[n].nodes[n].flags  ⟹  clusterState[n].nodes[n].slaveOf ≠ m
1822          ∧ clusterState' = [clusterState EXCEPT ![n] =
1823                                  [@ EXCEPT
1824                                    !.nodes = [x ∈ NODE ↦
1825                                                  IF @[x] ≠ NULL_ClusterNodeType
1826                                                   THEN IF x ≠ m
1827                                                          THEN ProcessDeletedNodes(@[x], {m})
1828                                                          ELSE  NULL_ClusterNodeType
1829                                                   ELSE  @[x]],
1830                                    !.slots   = [slot ∈ Slots ↦
1831                                                  IF @[slot] = m
1832                                                  THEN NULL_NODE
1833                                                  ELSE  @[slot]],
1834                                    !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"},
1835                                    !.nodesBlackList = @ ∪ {m},
1836                                    !.migratingSlotsTo = [slot ∈ Slots ↦
1837                                                           IF @[slot] = m
1838                                                            THEN NULL_NODE
1839                                                            ELSE  @[slot]],
1840                                    !.importingSlotsFrom = [slot ∈ Slots ↦
1841                                                             IF @[slot] = m
1842                                                              THEN NULL_NODE
1843                                                              ELSE  @[slot]]]]
1844          ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1846   REPLICATE < node-id>  – Configure current node as replica to < node-id> .
1847  ReplicateCommand(n) ≜
1848      ∧ CommandsExecutionPredicate(n)
1849          If the instance is currently a master, it should have no assigned slot
1850          If DB is empty is not taken into account
1851      ∧  ∨ "MASTER" ∉ clusterState[n].nodes[n].flags
1852         ∨ GetNodeNumSlots(clusterState[n].nodes[n]) = 0
1853      ∧ ∃ newMaster ∈ NODE :
1854          ∧ n ≠ newMaster
1855          ∧ clusterState[n].nodes[newMaster] ≠ NULL_ClusterNodeType
1856          ∧ "SLAVE" ∉ clusterState[n].nodes[newMaster].flags
1857          ∧ clusterState' =
1858              [clusterState EXCEPT
1859                 ![n] = [ClusterSetMaster(clusterState[n], newMaster) EXCEPT
1860                                 !.todoBeforeSleep = @ ∪ {"UPDATE_STATE"}]]
1861          ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩

1863   FAILOVER [force | takeover] – Promote current replica node to being a master.
1864  FailOverCommand(n) ≜
1865      LET myself ≜ ClusterLookupNode(clusterState[n], clusterState[n].myself)
1866      IN
1867      ∧ CommandsExecutionPredicate(n)
1868      ∧ "MASTER" ∉ myself.flags
```

```
1869        ∧ myself.slaveOf ≠ NULL_NODE
1870        ∧ ∨  No force or takeover
1871              ∧ "FAIL" ∉ ClusterLookupNode(clusterState[n], myself.slaveOf).flags
1872              ∧ ClusterLookupNode(clusterState[n], myself.slaveOf).hasLink
1873              ∧ LET
1874                    cluster ≜ clusterState[n]
1875                    resetManualFailover ≜ [ResetManualFailover(cluster) EXCEPT !.mfStarted = TRUE]

1877                    MFMessage ≜ ClusterSendMFStart(resetManualFailover, myself.slaveOf)
1878                    updatedNodeSendBuffer ≜ UpdateSenderBuffer(sendBuffer[n],
1879                                                                ⟨MFMessage⟩)
1880                IN  ∧ clusterState′ = [clusterState EXCEPT ![n] = resetManualFailover]
1881                    ∧ sendBuffer′ = [sendBuffer EXCEPT ![n] = updatedNodeSendBuffer]
1882                    ∧ UNCHANGED ⟨eventLoop, recieveBuffer⟩
1883           ∨  force - manual failover when the master is down
1884              ∧ clusterState′ = [clusterState EXCEPT ![n] =
1885                                    [ResetManualFailover(@) EXCEPT
1886                                        !.mfStarted = TRUE,
1887                                        !.mfCanStart = TRUE]]
1888              ∧ UNCHANGED ⟨eventLoop, recieveBuffer, sendBuffer⟩
1889           ∨  takeover − we want a replica to failover without any agreement with the rest of the cluster
1890              ∧ "FAIL" ∉ ClusterLookupNode(clusterState[n], myself.slaveOf).flags
1891              ∧ ClusterLookupNode(clusterState[n], myself.slaveOf).hasLink
1892              ∧ LET
1893                    cluster ≜ clusterState[n]

1895                    resetManualFailover ≜ [ResetManualFailover(cluster) EXCEPT !.mfStarted = TRUE]

1897                    bumpConfigEpochCluster ≜
1898                        ClusterBumpConfigEpochWithoutConsensus(resetManualFailover)

1900                    updatedInfo ≜ ClusterFailoverReplaceYourMaster(bumpConfigEpochCluster)
1901                    updatedNodeSendBuffer ≜ UpdateSenderBuffer(sendBuffer[n],
1902                                                                updatedInfo[2])
1903                IN  ∧ clusterState′ = [clusterState EXCEPT ![n] = updatedInfo[1]]
1904                    ∧ sendBuffer′ = [sendBuffer EXCEPT ![n] = updatedNodeSendBuffer]
1905                    ∧ UNCHANGED ⟨eventLoop, recieveBuffer⟩

1907  ClientAdminCommand(n) ≜
1908        ∨ MeetCommand(n)
1909    bug 6 fix start
1910      ∨FlushSlotsCommand(n)
1911    bug 6 fix end
1912    bug 5 fix start
1913      ∨AddSlotsCommand(n)
1914      ∨DelSlotsCommand(n)
1915    bug 5 fix end
1916        ∨ SetSlotCommand(n)
1917        ∨ BumpEpochCommand(n)
1918    bug 7 fix start
1919      ∨ForgetCommand(n)
1920    bug 7 fix end
1921        ∨ ReplicateCommand(n)
1922        ∨ FailOverCommand(n)

1924 ├──────────────────────────────────────────────────────────────────────────┤
      The next-state action.
1928 ├──────────────────────────────────────────────────────────────────────────┤
1929  TCTypeOK ≜
```

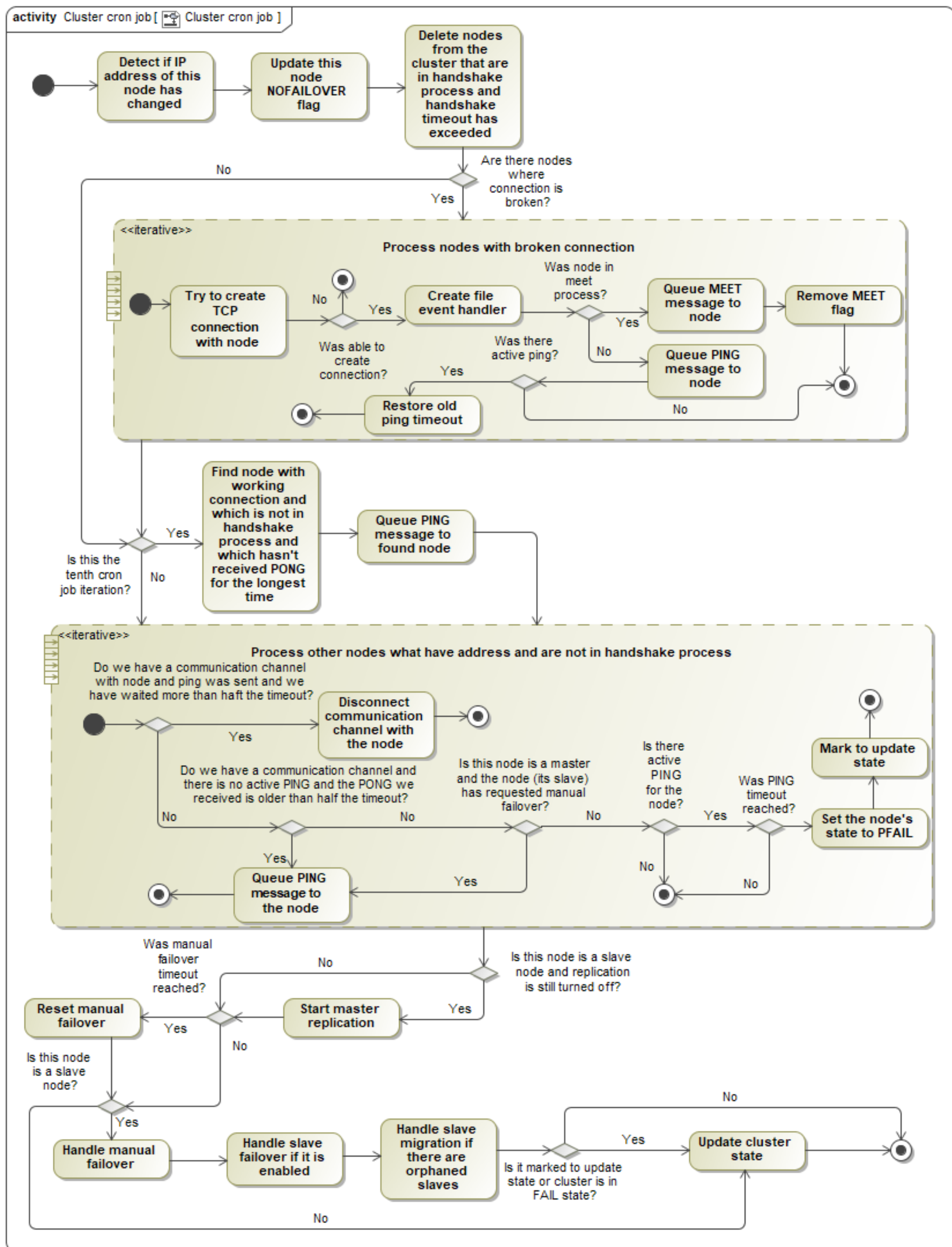
```
1930     ∧ clusterState ∈ [NODE → ClusterStateType]
1931     ∧ eventLoop ∈ [NODE → EventLoopType]
1932     ∧ recieveBuffer ∈ BufferType
1933     ∧ sendBuffer ∈ BufferType
1934 ├──────────────────────────────────────────────────────────────────────────┤
1935 TCNext ≜
1936     ∃ node ∈ NODE :
1937         ∨ EventLoopMain(node)
1938         ∨ TimeActions(node)
1939         ∨ ChangeAddress(node)
1940         ∨ ClientAdminCommand(node)
1941 ├──────────────────────────────────────────────────────────────────────────┤
1942 CreateClusterNodeWithSlots(node, flags, slotsBeingResponsible, configEpoch) ≜
1943     LET newNode ≜ CreateClusterNode(node, flags)
1944     IN  [newNode EXCEPT !.slots = [slot ∈ Slots ↦ IF slot ∈ slotsBeingResponsible
1945                                                     THEN TRUE
1946                                                     ELSE  FALSE],
1947                         !.hasLink = TRUE,
1948                         !.configEpoch = configEpoch]

1950 CreateClusterState(n, nodes, slots, currentEpoch) ≜
1951     [
1952     myself ↦ n,
1953     currentEpoch ↦ currentEpoch,
1954     clusterState   ↦ "OK",
1955     nodes ↦ nodes,
1956     nodesBlackList ↦ {},
1957     migratingSlotsTo ↦ [slot ∈ Slots ↦ NULL_NODE],
1958     importingSlotsFrom ↦ [slot ∈ Slots ↦ NULL_NODE],
1959     slots ↦ slots,
1960     mfStarted ↦ FALSE,
1961     mfEnded ↦ FALSE,
1962     mfSlave  ↦ NULL_NODE,
1963     mfMasterOffsetSet ↦ FALSE,
1964     mfCanStart ↦ FALSE,
1965     todoBeforeSleep ↦ {}
1966     ]

1968 InitThreeMasterNodes ≜
1969     LET numOfSlots ≜ SLOTS_COUNT ÷ 3
1970         possibleConfigEpochs ≜ 1 . . Cardinality(NODE)
1971         firstNode ≜ CHOOSE x ∈ NODE : TRUE
1972         firstNodeSlots ≜ 1 . . numOfSlots
1973         firstNodeConfigEpoch ≜ CHOOSE x ∈ possibleConfigEpochs : TRUE
1974         secondNode ≜ CHOOSE x ∈ NODE : x ≠ firstNode
1975         secondNodeSlots ≜ (numOfSlots + 1) . . (2 ∗ numOfSlots)
1976         secondNodeConfigEpoch ≜ CHOOSE x ∈ possibleConfigEpochs : x ≠ firstNodeConfigEpoch
1977         thirdNode ≜ CHOOSE x ∈ NODE : x  ≠ firstNode ∧ x ≠ secondNode
1978         thirdNodeSlots ≜ (Slots \ firstNodeSlots) \ secondNodeSlots
1979         thirdNodeConfigEpoch ≜ CHOOSE x ∈ possibleConfigEpochs :  ∧ x ≠ firstNodeConfigEpoch
1980                                                                     ∧ x ≠ secondNodeConfigEpoch
1981     IN
1982      ∧ clusterState =
1983         [n ∈ NODE ↦
1984             CreateClusterState(n,
1985                                 [m ∈ NODE ↦
1986                                     CASE
1987                                         m = firstNode →
```

```
1988                                              CreateClusterNodeWithSlots(m,
1989                                                                           IF n = m
1990                                                                            THEN {"MYSELF", "MASTER"}
1991                                                                            ELSE {"MASTER"},
1992                                                                           firstNodeSlots,
1993                                                                           firstNodeConfigEpoch)
1994                                       □ m = secondNode →
1995                                              CreateClusterNodeWithSlots(m,
1996                                                                           IF n = m
1997                                                                            THEN {"MYSELF", "MASTER"}
1998                                                                            ELSE {"MASTER"},
1999                                                                           secondNodeSlots,
2000                                                                           secondNodeConfigEpoch)
2001                                       □ m = thirdNode →
2002                                              CreateClusterNodeWithSlots(m,
2003                                                                           IF n = m
2004                                                                            THEN {"MYSELF", "MASTER"}
2005                                                                            ELSE {"MASTER"},
2006                                                                           thirdNodeSlots,
2007                                                                           thirdNodeConfigEpoch)],
2008                                 [slot ∈ Slots ↦  CASE slot ∈ firstNodeSlots → firstNode
2009                                                   □slot ∈ secondNodeSlots → secondNode
2010                                                   □slot ∈ thirdNodeSlots → thirdNode],
2011                                 Cardinality(NODE))]
2012      ∧ eventLoop = [n ∈ NODE ↦ [
2013                                   state ↦ "BEFORE_SLEEP",
2014                                   eventsBeingProcessed ↦ [m ∈ NODE ↦ NULL_EventLink],
2015                                   currentEventBeingProcessed ↦ NULL_EventLink,
2016                                   timeEvent ↦ [
2017                                                hasOccured ↦ FALSE,
2018                                                iteration ↦ 0
2019                                               ]
2020                                  ]]
2021      ∧ recieveBuffer = [n ∈ NODE ↦ [m ∈ NODE ↦ ⟨⟩]]
2022      ∧ sendBuffer = [n ∈ NODE ↦ [m ∈ NODE ↦ ⟨⟩]]

2024 TCInit ≜ InitThreeMasterNodes
2025 ├──────────────────────────────────────────────────────────────────────┤
     Properties invariants.

2030   Only single master node and it slave nodes must be responsible for a single cluster slot
2031 SlotsInvariant ≜
2032     LET
2033     isNodeAvailableForClientsReadEvents[node ∈ NODE] ≜
2034         ∧ "MASTER" ∈ clusterState[node].nodes[node].flags
2035          Cluster is not down and can process request
2036         ∧ clusterState[node].clusterState = "OK"
2037          Client can execute command request between any file event
2038         ∧ eventLoop[node].state = "PROCESSING_FILE_EVENTS"
2039         ∧ eventLoop[node].currentEventBeingProcessed = NULL_EventLink
2040     IN  ∀ slot ∈ Slots :
2041            ∀ n, m ∈ NODE :
2042               ∧ n ≠ m
2043               ∧ isNodeAvailableForClientsReadEvents[n]
2044               ∧ isNodeAvailableForClientsReadEvents[m]
2045                slot is not migrating
2046               ∧ clusterState[n].migratingSlotsTo[slot]  = NULL_NODE
2047               ∧ clusterState[m].migratingSlotsTo[slot] = NULL_NODE
```

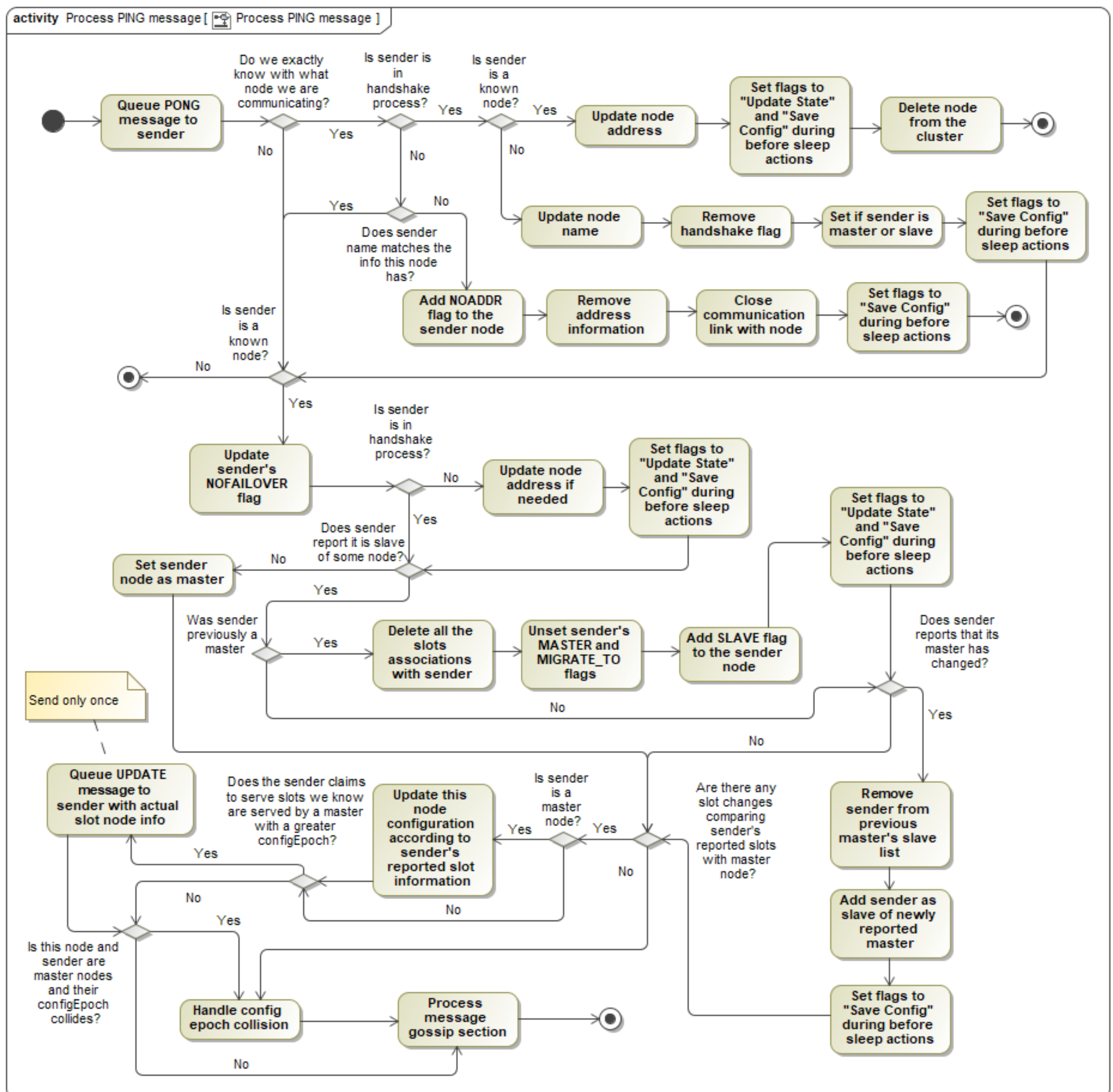
```
2048            ⟹
2049             Two different nodes should report the same node as responsible for this slot
2050            clusterState[n].slots[slot] = clusterState[m].slots[slot]

2052 ├──────────────────────────────────────────────────────────────────────────┤
     The complete spec of the Redis Cluster system.

2057 TCSpec ≜ TCInit ∧ □[TCNext]_⟨clusterState, eventLoop, recieveBuffer, sendBuffer⟩

2059 THEOREM TCSpec ⟹ □TCTypeOK
2060 THEOREM TCSpec ⟹ □SlotsInvariant

2062 ├──────────────────────────────────────────────────────────────────────────┤
     Refinement mapping.
2066 Abstract ≜ INSTANCE RedisCluster_abstract WITH
2067       NODE ← NODE,
2068       SLOTS ← Slots,
2069       nodeFailed ←
2070           [n ∈ NODE ↦ ∀ m ∈ NODE :
2071                             ∧ clusterState[m].nodes[n] ≠ NULL_ClusterNodeType
2072                             ∧ "FAIL" ∈ clusterState[m].nodes[n].flags],
2073       slaveOf ←
2074           [n ∈ NODE ↦ IF clusterState[n].nodes[n].slaveOf = NULL_NODE
2075                         THEN {}
2076                         ELSE {clusterState[n].nodes[n].slaveOf}],
2077       nodeSlaves ←
2078           [n ∈ NODE ↦ {m ∈ NODE : clusterState[m].nodes[m].slaveOf = n}],
2079       clusterSlots ←
2080           [s ∈ Slots ↦ {m ∈ ({clusterState[n].slots[s] : n ∈ NODE} \ {NULL_NODE}) :
2081                          clusterState[m].slots[s] = m}],
2082       clusterKnownNodes ←
2083           {n ∈ NODE : ∃ m ∈ NODE : clusterState[m].nodes[n] ≠ NULL_ClusterNodeType},
2084       clusterState ←
2085           IF ∀ n ∈ NODE : clusterState[n].clusterState = "FAIL"
2086            THEN "FAIL"
2087            ELSE "OK"

2089 ├──────────────────────────────────────────────────────────────────────────┤
2090 THEOREM Refinement ≜ TCSpec ⟹ Abstract!TCSpec
2091 └──────────────────────────────────────────────────────────────────────────┘
```

Priedas nr. 3

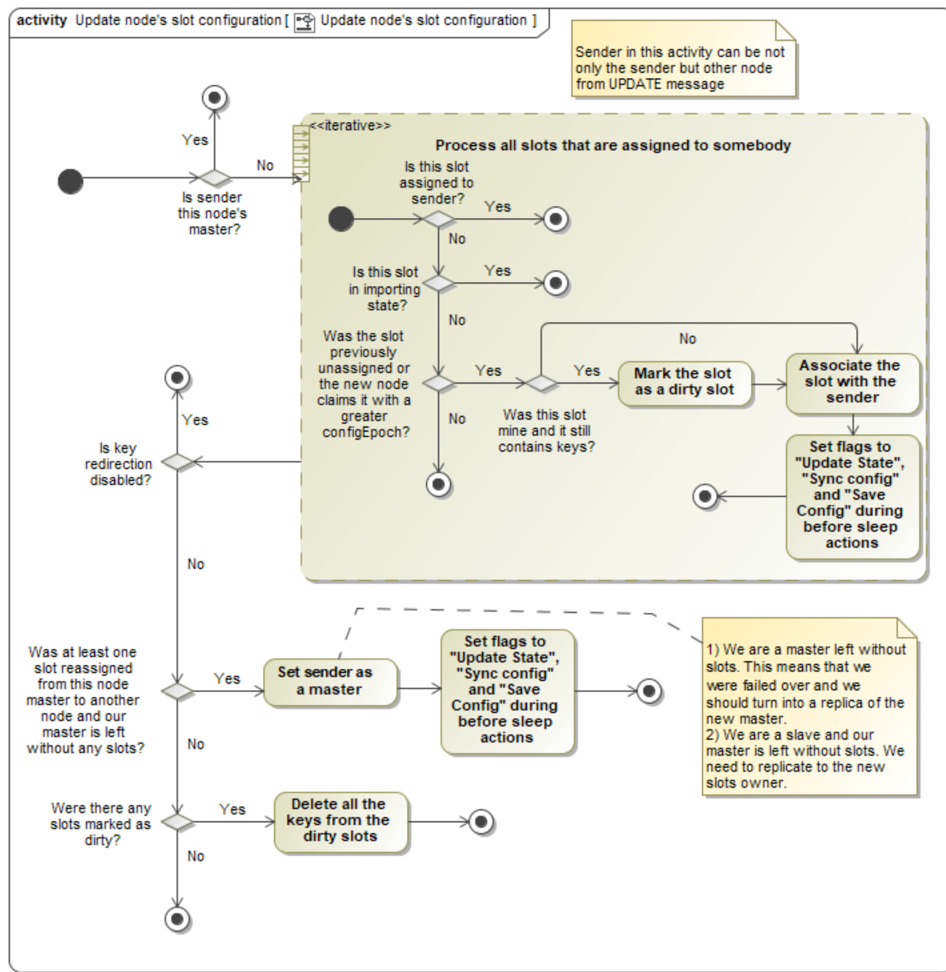
„Redis Cluster“ sistemos UML diagramos



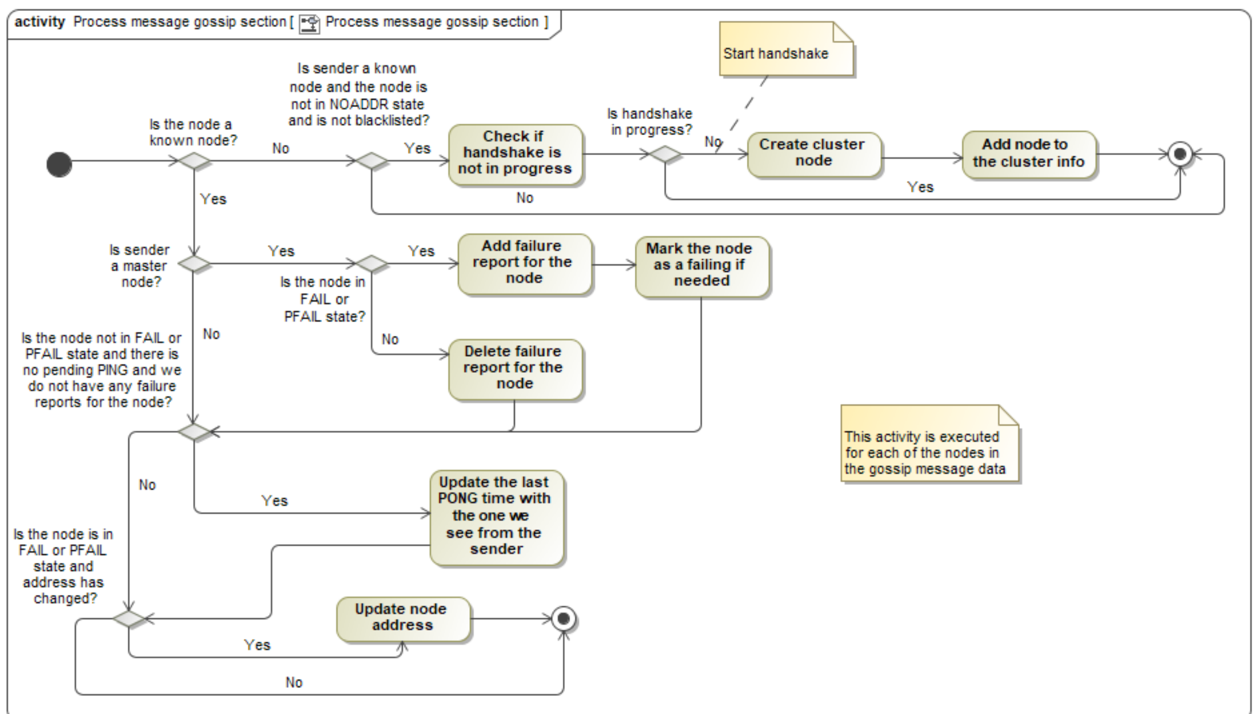
9 pav. „Redis Cluster“ blokinio suplanuoto darbo veiklos diagrama



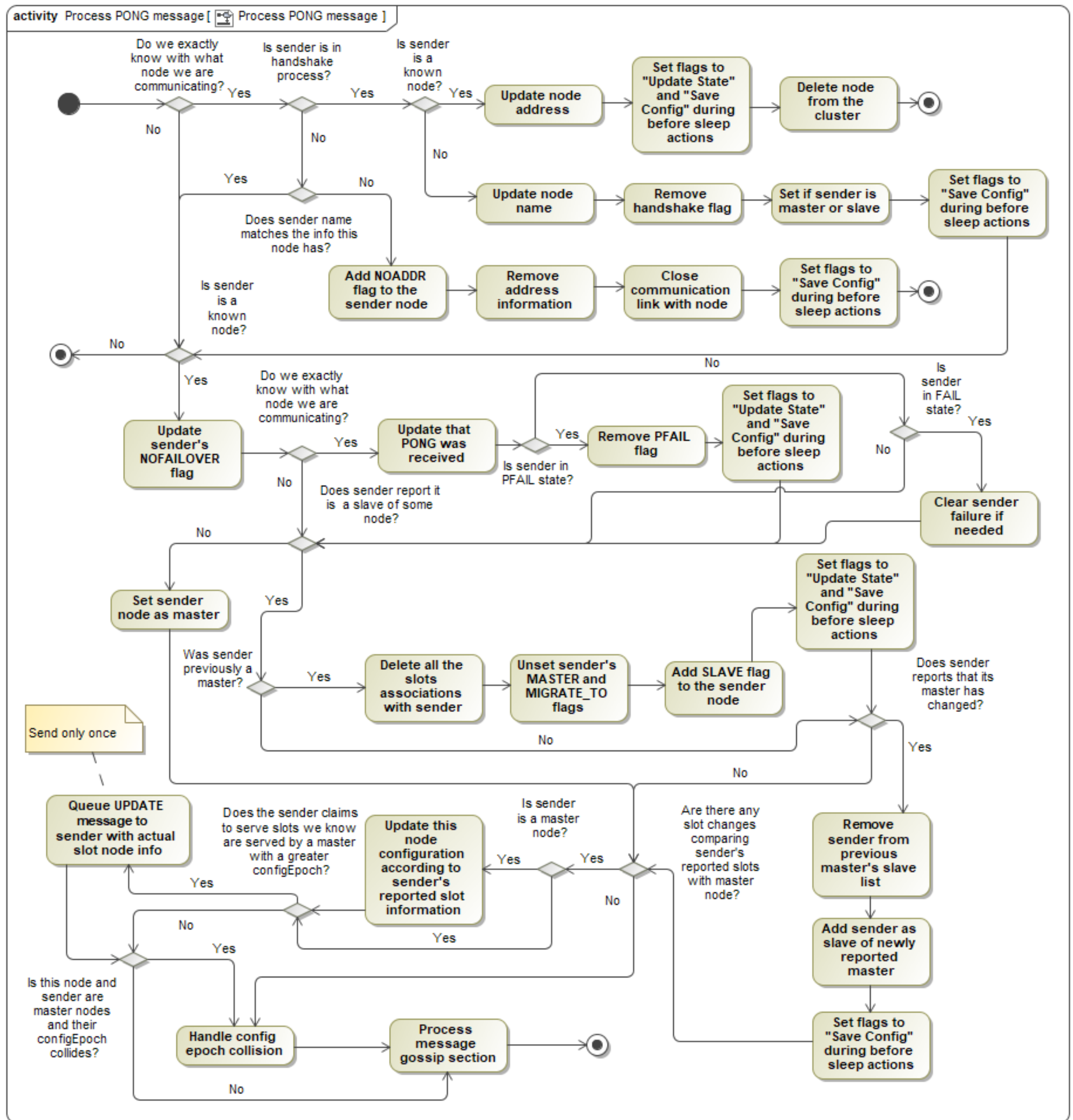
10 pav. *Ping* tipo žinutės apdorojimo veiklos diagrama



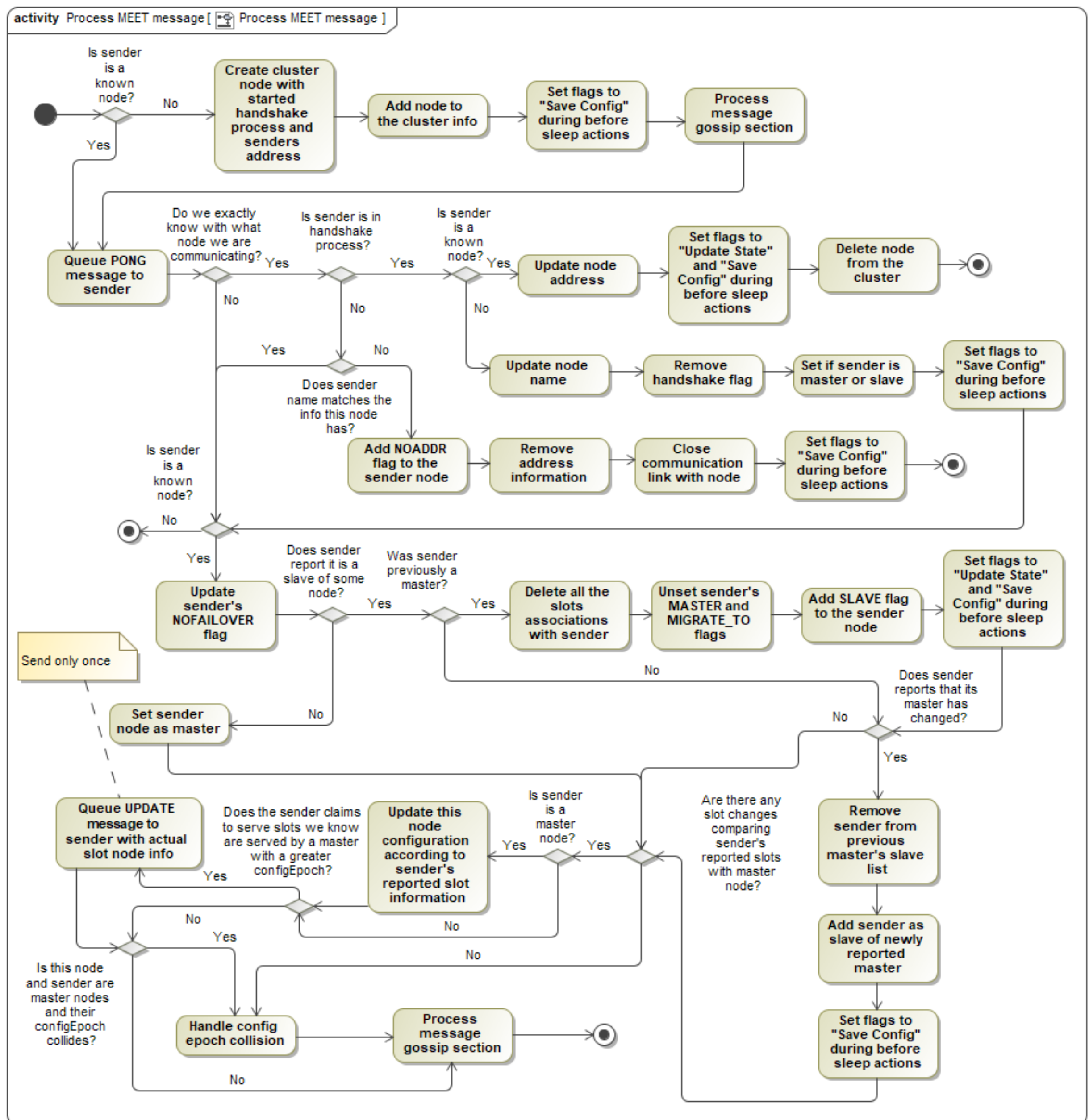
11 pav. Maišos lizdų atsakomybių atnaujinimo veiklos diagrama



12 pav. Paskalose esančio mazgo informacijos apdorojimo veiklos diagrama



13 pav. *Pong* tipo žinutės apdorojimo veiklos diagrama



14 pav. *Meet* tipo žinutės apdorojimo veiklos diagrama