Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jan Kos

Algoritmi soglasja v porazdeljenih krmilnikih programsko določenih omrežij

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Mojca Ciglarič

Ljubljana, 2018

Zahvala

Zahvaljujem se mentorici izr. prof. dr. Mojci Ciglarič za strokovno pomoč, vodenje in nasvete pri izdelavi magistrskega dela. Iskreno se zahvaljujem tudi svoji družini in punci, ki so me ves čas podpirali, me spodbujali takrat, ko sem to najbolj potreboval, in mi vedno stali ob strani.

Jan Kos, 2018

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
IP	Internet Protcol	internetni protokol
SDN	Software Defined Networks	programsko določena omrežja
API	Application Programming Interface	aplikacijski programski vmesnik
TE	Traffic Engineering	prometno načrtovanje
SPOF	Single Point of Failure	kritična točka odpovedi
HA	High Availability	visoka razpoložljivost
FT	Fault-Tolerance	odpornost na napake
CAP	Consistency, Availability, Partition tolerance	konsistentnost, razpoložljivost, particijska toleranca
ONF	Open Networking Foundation	fundacija za odprta omrežja
BFD	Bidirectional Forwarding Detection	dvosmerno odkrivanje napak
ICMP	Internet Control Message Protocol	internetni protokol za izmenjavo kontrolnih sporočil
ATM	Asynchronous Transfer Mode	asinhroni prenosni način
GSMP	General Switch Management Protocol	splošni protokol za upravljanje stikal
IETF	Internet Engineering Task Force	delovna skupina za internetno tehniko

kratica	**angleško**	**slovensko**
MPLS	Multiprotocol Label Switching	večprotokolna komutacija z zamenjavo label
MPLS-TE	MPLS-Traffic Engineering	prometno načrtovanje MPLS
ForCES	Forwarding and Control Element Separation	ločitev krmilnega in posredovalnega elementa
PCE	Path Computation Element	element za izračun poti
PCC	Path Computation Client	element, ki zahteva izračun poti
BGP	Border Gateway Protocol	protokol mejnih usmerjevalnikov
BGP-LS	BGP-Link State	stanje povezav BGP
TCAM	Ternary Content-Addressable Memory	ternarni vsebinsko naslovljiv pomnilnik
BCAM	Binary Content-Addressable Memories	binarni vsebinsko naslovljiv pomnilnik
NOS	Network Operating System	omrežni operacijski sistem
LLDP	Link Layer Discovery Protocol	protokol za odkrivanje na povezavni plasti
REST	Representational State Transfer	slog spletne storitve
IBN	Intent-Based Networking	mreženje na osnovi namena
DPI	Deep Packet Inspection	temeljito pregledovanje paketov
ToS	Type of Service	vrsta storitve
WAN	Wde Area Networks	prostrana omrežja
RTBH	Remotely Triggered Black Hole	oddaljeno prožena črna luknja
NFV	Network Functions Virtualization	virtualizacija omrežnih funkcij
CO	Central Office	funkcijska lokacija
CORD	Central Office Rearchitected as a Datacenter	preoblikovanje funkcijske lokacije v podatkovni center
BNG	Broadband Network Gateway	širokopasovni robni prehod
OLT	Optical Line Termination	optično dostopovno vozlišče
SLA	Service-Level Agreement	dogovor na ravni storitve

kratica	angleško	slovensko
GPS	Global Positioning System	sistem globalnega pozicioniranja
BFT	Byzantine Fault-Tolerance	bizantinska odpornost na napake
PoW	Proof-of-Work	dokazilo o delu
RPC	Remote Procedure Call	oddaljen klic procedure
MTBF	Mean Time Between Failure	povprečen čas med zaporednima odpovedma
2PC	Two-Phase Commit	protokol dvofaznega potrjevanja

Povzetek

Naslov: Algoritmi soglasja v porazdeljenih krmilnikih programsko določenih omrežij

Programsko določena omrežja z logično centralizacijo nadzora in abstrakcijo omrežnih virov obljubljajo večjo prilagodljivost, cenovno učinkovitost in lažje upravljanje omrežne infrastrukture, vendar njihovo širšo uporabo ovirajo različni izzivi. V magistrskem delu raziščemo problem zagotavljanje visoke razpoložljivosti krmilnikov programsko določenih omrežij. Gre za slabo raziskan problem, ki je hkrati ključnega pomena pri uvajanju programsko določenih omrežij v produkcijska okolja. Najprej predstavimo koncept programsko določenih omrežij. Pri tem raziščemo, kje ima njihova uporaba največji potencial in identificiramo različne izzive, ki otežujejo njihovo uvedbo. Izpostavimo problem zagotavljanja visoke razpoložljivosti in skalabilnosti omrežja zaradi centralizacije krmilne ravnine ter kot rešitev navedemo implementacijo krmilnika v obliki porazdeljenega sistema odpornega na napake. V nadaljevanju preučimo omejitve pri načrtovanju in implementaciji teh sistemov. Osredotočimo se na algoritme soglasja kot ključno komponento za zagotavljanje visoke razpoložljivosti. Nato predstavimo strategije pri zasnovi porazdeljenih krmilnikov in raziščemo, katere odprtokodne implementacije omogočajo njihovo visoko razpoložljivost. Krmilnik ONOS izberemo kot potencialno najbolj primeren za produkcijsko uporabo in analiziramo njegovo arhitekturo. Z izbranim krmilnikom in simulatorjem programsko določenih omrežij Mininet vzpostavimo pilotno okolje. Razvijemo testno ogrodje, s katerim simuliramo scenarije, ki vključujejo različne odpovedi vozlišč krmilnika

in komunikacijskih kanalov, ter pri tem analiziramo obnašanje sistema. Na podlagi rezultatov analize ovrednotimo izbrano implementacijo krmilnika z vidika zagotavljanja visoke razpoložljivosti in podamo predloge za izboljšanje razpoložljivosti rešitve.

Ključne besede

programsko določena omrežja, visoka razpoložljivost, porazdeljeni sistemi

Abstract

Title: Consensus algorithms in distributed SDN controllers

Software defined networks (SDN) promise greater flexibility, cost efficiency and easier management of network infrastructure by logically centralizing the control and abstracting network resources, but their wider use is inhibited by various challenges. In this master's thesis, we explore the problem of ensuring high-availability of SDN controllers. It is a mostly unexplored problem which is also crucial with the implementation of software defined networks in production environments. Firstly, we present the concept of software defined networking. Doing so, we explore where their use has the greatest potential and identify various challenges which make their implementation difficult. We emphasize the problem of ensuring high-availability and the scalability of the network because of the centralized control plane and list as a solution the implementation of a controller in the form of a fault-tolerant distributed system. Following this, we study the limitations in design and the implementation of these systems. We focus on consensus algorithms as a key component in ensuring high availability. Following this, we present strategies with developing distributed controllers and explore which open-source implementations allow for their high availability. We choose the ONOS controller as the potentially most suitable for production use and analyze its architecture. With the chosen controller and the simulator of software defined networks Mininet, we establish a pilot environment. We develop a test framework for simulating scenarios that include various controller node and communication channel failures and analyze system behavior while do-

ing so. Based on the results of the analysis we evaluate the chosen controller implementation based on ensuring high availability and give suggestions for improving the availability of the solution.

Keywords

software-defined networks, high availability, distributed systems

Poglavje 1

Uvod

Razvoj interneta nas je pripeljal v digitalno družbo, kjer je skoraj vse povezano in dostopno kjerkoli in kadarkoli. Priča smo eksponentnemu povečevanju informacijskih potreb. Količina letnega prometa IP (angl. *Internet Protocol*) na globalni ravni naj bi že v letu 2016 presegla zetabajt [1]. V razmahu so oblačne storitve, ki zahtevajo fleksibilno infrastrukturo za avtomatizirano izstavljanje poljubnih virov na zahtevo. Vse to prinaša nove izzive tudi na področju omrežne infrastrukture, medtem ko so tradicionalna omrežja vedno bolj kompleksna in težavna za upravljanje [2]. Sestavljena so iz avtonomnih naprav, ki vsaka zase tesno združujejo upravljavsko, krmilno in podatkovno ravnino. Upravljavska ravnina (angl. *management plane*) omogoča konfiguracijo in nadzor naprave, krmilna ravnina (angl. *control plane*) zajema številne kontrolne in usmerjevalne protokole, s pomočjo katerih izračuna in namesti ustrezna pravila v posredovalne tabele, medtem ko podatkovna ravnina (angl. *data plane*) skrbi za učinkovito posredovanje paketov na podlagi teh pravil. Uveljavljanje novih politik tako zahteva uporabo nizkonivojskih nestandardiziranih ukazov na množici naprav, ki nimajo globalnega pogleda na omrežje, hitro in samodejno odzivanje na spremembe v vse bolj dinamičnih okoljih pa predstavlja velik izziv [3]. V zadnjem času se zato pojavlja nova paradigma programsko določenih omrežij (angl. *Software-Defined Networks, SDN*), ki z logično centralizacijo nadzora in abstrakcijo

omrežnih virov obljublja večjo prilagodljivost, cenovno učinkovitost in lažje upravljanje omrežne infrastrukture.

Ključni element programsko določenih omrežij predstavlja krmilnik omrežja. Gre za programsko opremo, ki teče na običajnih strežnikih in preko južnih programskih vmesnikov (angl. *southbound APIs*) določa delovanje podatkovne ravnine v množici omrežnih naprav [4]. Najbolj razširjen protokol za komunikacijo med krmilnikom in posredovalnimi napravami, je protokol OpenFlow, ki omogoča krmiljenje v proaktivnem ali reaktivnem načinu s pomočjo tokovnih pravil (angl. *flow rules*). Proaktivni način omogoča vnaprejšnjo namestitev tokovnih pravil, medtem ko v reaktivnem načinu pravila predstavljajo odziv na dejanski promet v omrežju. Krmilnik preko severnih programskih vmesnikov (angl. *northbound APIs*) aplikacijam zagotavlja abstrakcijo omrežnih virov in globalen pogled na omrežje. Krmilna ravnina je tako logično centralizirana, omrežne naprave pa so zadolžene le še za posredovanje prometa na podlagi pravil, ki jih določa osrednji krmilnik. Takšen pristop omogoča hitrejšo implementacijo omrežnih funkcionalnosti v obliki aplikacij, z globalnim pogledom na omrežje poenostavi reševanje problemov, kot sta prometno načrtovanje (angl. *traffic engineering, TE*) ali uveljavljanje različnih politik na nivoju celotnega omrežja, in zato predstavlja velik potencial zlasti za ponudnike oblačnih in telekomunikacijskih storitev. [5]

Centralizacija krmilne ravnine pa ima poleg težav s skalabilnostjo tudi vpliv na razpoložljivost celotnega omrežja, saj osrednji krmilnik predstavlja kritično točko odpovedi (angl. *single point of failure, SPOF*). V pričujočem delu podrobneje raziščemo možnosti za zagotavljanje visoke razpoložljivosti (angl. *high availability, HA*) krmilnikov programsko določenih omrežij. Gre za slabo raziskan problem, ki je hkrati ključnega pomena pri uvajanju programsko določenih omrežij v produkcijska okolja [4, 6]. O visoki razpoložljivosti govorimo takrat, kadar je razpoložljivost, torej odstotek časa, ko je storitev dostopna z razumno zakasnitvijo, blizu 100% [7]. To lahko zagotovimo z implementacijo krmilnika v obliki porazdeljenega sistema, ki lahko nemoteno deluje tudi v primeru odpovedi posameznih instanc krmil-

nika.

Temeljni problem porazdeljenih sistemov, odpornih na napake (angl. *fault-tolerant distributed systems*), predstavlja problem soglasja (angl. *consensus problem*), ki v popolnoma asinhronem sistemu nima rešitve [7], medtem ko rešitve v delno sinhronem sistemu vodijo v kompromis med varnostjo (angl. *safety*) in živostjo (angl. *liveness*). Omejitve pri implementaciji praktičnih sistemov ponazarja teorem CAP, ki pravi, da v primeru razdelitve omrežja ne moremo hkrati zagotoviti konsistentnosti in razpoložljivosti podatkov [8, 9]. V kontekstu krmilnikov so to podatki o stanju omrežja. Teorem PACELC razširja teorem CAP z navedbo, da je tudi v primeru normalnega delovanja potreben kompromis, in sicer kompromis med nizko zakasnitvijo ter strogo konsistentnostjo [10]. Različne implementacije tako slonijo bodisi na modelu stroge ali eventuelne konsistentnosti kot tudi na kombinaciji obeh modelov [11, 12, 13]. Strogo konsistentno stanje krmilnikov lahko zagotovimo z uporabo algoritmov soglasja (angl. *consensus algorithms*) [14, 15]. Najbolj razširjen je algoritem Paxos, medtem ko Raft predstavlja enostavnejšo in bolj razumljivo alternativo [16], ki je vedno bolj zanimiva tudi za uporabo v praktičnih sistemih.

Glavni prispevek magistrskega dela je analiza obnašanja programsko določenega omrežja v primeru različnih odpovedi. V ta namen predstavimo ključne strategije pri zasnovi porazdeljenih krmilnikov in raziščemo, katere odprtokodne implementacije omogočajo njihovo visoko razpoložljivost. Kot potencialno najbolj primeren krmilnik za uporabo v produkcijskem okolju izberemo krmilnik ONOS, analiziramo njegovo arhitekturo in vzpostavimo pilotno okolje. Za potrebe predvidene analize razvijemo testno ogrodje z razširitvijo simulacijskega okolja Mininet. Z uporabo razvite rešitve simuliramo testne scenarije za različne primere odpovedi, ter pri tem analiziramo obnašanje sistema. Na podlagi rezultatov analize ovrednotimo izbrano impelmentacijo krmilnika z vidika zagotavljanja visoke razpoložljivosti in podamo predloge za izboljšanje razpoložljivosti rešitve.

Struktura preostanka pričujočega dela je naslednja: v poglavju 2 predsta-

vimo koncept programsko določenih omrežij, pri čemer raziščemo, kje ima njihova uporaba največji potencial in identificiramo različne izzive, ki otežujejo širšo uporabo. Kot ključno oviro pri uvedbi programsko določenih omrežij v produkcijska okolja izpostavimo problem zagotavljanja visoke razpoložljivosti in skalabilnosti omrežja zaradi centralizacije krmilne ravnine ter kot rešitev navedemo implementacijo krmilnika v obliki porazdeljenega sistema odpornega na napake. Omejitve pri načrtovanju in implementaciji teh sistemov preučimo v poglavju 3. Osredotočimo se na algoritme soglasja kot ključno komponento za zagotavljanje visoke razpoložljivosti. V poglavju 4 predstavimo ključne strategije pri zasnovi porazdeljenih krmilnikov, raziščemo, katere odprtokodne implementacije omogočajo njihovo visoko razpoložljivost, in analiziramo arhitekturo izbranega krmilnika. V poglavju 5 opišemo vzpostavitev pilotnega okolja, ki vključuje razvoj testnega ogrodja. Nato v poglavju 6 predstavimo različne testne scenarije in analizo obnašanja na podlagi rezultatov teh testov. Zaključimo s sklepnim poglavjem (poglavje 7), kjer izpostavimo ključne ugotovitve.

Poglavje 2

Programsko določena omrežja

Različne ideje in tehnologije za programsko upravljanje omrežij se postopoma razvijajo že zadnjih dvajset let, medtem ko se je izraz programsko določena omrežja (angl. *Software-Defined Networks, SDN*) širše uveljavil v kontekstu projekta OpenFlow, ki so ga predstavili na Stanfordski univerzi leta 2008 [17]. Od tod izvira tudi osnovna definicija te paradigme, ki predvideva fizično ločitev krmilne in podatkovne ravnine v omrežju, pri čemer je krmilna ravnina logično centralizirana in določa delovanje podatkovne ravnine v množici omrežnih naprav. Zanimanje za programsko določena omrežja narašča od leta 2011, ko so potencial prepoznali Google, Facebook, Yahoo, Microsoft, Verizon in Deutsche Telekom ter ustanovili organizacijo ONF (angl. *Open Networking Foundation*) za promocijo in uveljavitev SDN z razvojem odprtih standardov. ONF še danes bdi nad standardom OpenFlow, ki predstavlja najbolj razširjeno izvedbo programsko določenih omrežij.

V zadnjem času se pojavlja vse več alternativnih definicij programsko določenih omrežij, ki nekoliko odstopajo od prvotne ideje. Temeljijo predvsem na uporabi obstoječih vmesnikov za upravljanje in nastavljanje omrežnih naprav. Namesto centralizacije krmilne ravnine ima krmilnik vlogo orkestratorja omrežnih storitev. To pomeni, da skrbi predvsem za koordinacijo aktivnosti pri avtomatiziranem nastavljanju in upravljanju naprav, ki zagotavljajo želene omrežne storitve. Takšne rešitve so v praksi širše uporabne in

združljive z obstoječo infrastrukturo. Kljub temu lahko v določenih primerih uporabe prava programsko določena omrežja prinašajo številne prednosti. V nadaljevanju dela se bomo osredotočili na arhitekturo in koncepte, ki izhajajo iz osnovne definicije programsko določenih omrežij in standarda OpenFlow.

2.1 Definicija

Programsko določena omrežja predstavljajo nov pristop k realizaciji računalniških omrežij, ki ima naslednje ključne lastnosti [18, 19, 5]:

1. **Ločitev krmilne in podatkovne ravnine** (angl. *Decoupling of control and data planes*)
   Naloga podatkovne ravnine je medpomnjenje (angl. *buffering*), razvrščanje (angl. *scheduling*) in posredovanje (angl. *forwarding*) paketov ter prilagajanje podatkov v zaglavju. V ta namen podatkovna ravnina vključuje fizične vmesnike za sprejem in oddajo, posredovalne tabele in pripadajočo logiko za ustrezno obdelavo prejetih paketov glede na različne podatke iz zaglavij, kot so naslovi MAC ali IP, identifikatorji VLAN in značke MPLS. Za izračun in namestitev ustreznih pravil v posredovalne tabele so zadolženi različni kontrolni in usmerjevalni protokoli krmilne ravnine. V nasprotju s tradicionalnimi omrežji, kjer je krmilna ravnina vsebovana v vsaki izmed omrežnih naprav in tesno združena s podatkovno ravnino, je krmilna ravnina programsko določenih omrežij fizično ločena. Implementirana je v obliki zunanje entitete, ki jo imenujemo krmilnik SDN. Ločitev nam omogoča neodvisen razvoj kontrolnih funkcij in logično centralizacijo nadzora. Funkcija omrežnih naprav je pri tem poenostavljena.

2. **Logično centraliziran nadzor** (angl. *Logically centralized control*)
   Nadzor nad omrežjem je logično centraliziran s pomočjo osrednjega krmilnika SDN, ki opravlja nalogo krmilne ravnine za množico omrežnih naprav. Z globalnim pogledom na omrežje ima krmilnik možnost učin-

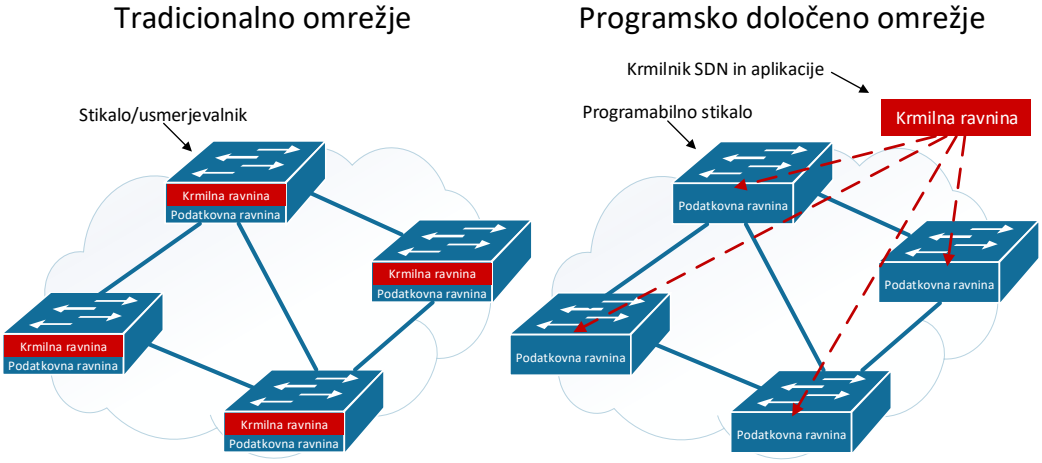
kovitejše izrabe virov in poenostavi reševanje problemov, kot je prometno načrtovanje ali uveljavljanje politik na ravni celotnega omrežja. Ravno tako je poenostavljen razvoj novih omrežnih funkcionalnosti in samodejno odzivanje na spremembe v omrežju. Kljub temu da je krmilna ravnina v osnovi centralizirana, arhitektura z namenom večje operativne učinkovitosti dopušča tudi delegacijo določenih kontrolnih funkcij omrežnim napravam. To je smiselno predvsem za funkcije, kot je procesiranje sporočil ICMP ali hitro zaznavanje izpadov s protokolom BFD (angl. *Bidirectional Forwarding Detection*), saj njihova centralizacija ne prinaša dodane vrednosti.

3. **Programabilnost omrežnih storitev** (angl. *Programmability of network services*)
   Omrežne storitve so programabilne s pomočjo aplikacij, ki tečejo na krmilniku SDN. Za razvoj novih omrežnih storitev nista potrebna uvedba dragih namenskih naprav ali poseg v strojno opremo, saj jih lahko razvijamo neodvisno, in sicer v obliki programskih modulov za osrednji krmilnik. Pri tem se programerju ni treba ukvarjati s podrobnostmi fizične arhitekture, saj krmilnik skrbi za abstrakcijo porazdeljenega stanja, posredovanja in konfiguracije [20]. Abstrakcija posredovanja omogoča, da programer opredeli želeno stanje posredovalnih tabel brez specifičnih ukazov strojne opreme. Abstrakcija porazdeljenega stanja aplikacijam zagotavlja globalen pogled na omrežje, kar pomeni, da programer namesto kompleksnega porazdeljenega problema rešuje preprostejši logično centraliziran problem. Abstrakcija konfiguracije pa pomeni, da lahko aplikacija zahteva želeno obnašanje omrežja, za izvedbo in optimizacijo v realnem času pa poskrbi krmilnik. Uporaba aplikacijskih vmesnikov pri tem omogoča enostavnejšo integracijo z ostalimi sistemi in prilagajanje omrežnih storitev potrebam aplikacij v realnem času.

4. **Odprti vmesniki** (angl. *Open interfaces*)

Velika želja pri implementaciji programsko določenih omrežij je uporaba odprtih, dobro dokumentiranih in standardiziranih aplikacijskih programskih vmesnikov tako za komunikacijo med aplikacijami in krmilnikom kot tudi za upravljanje podatkovne ravnine. Na ta način naj bi se izognili zaklepanju uporabnikov, spodbudili razvoj inovativnih omrežnih funkcij in znižali stroške izdelave strojne opreme. Praktične rešitve za implementacijo določenih funkcionalnosti kljub temu vključujejo lastniške razširitve vmesnikov. Lastniki omrežij se morajo odločiti za kompromis med odprtostjo vmesnikov in dodano vrednostjo, ki jo ta rešitev ponuja.

Visokonivojski pogled na arhitekturo programsko določenega omrežja v primerjavi s tradicionalnim omrežjem lahko vidimo na sliki 2.1. Krmilna rav-



Slika 2.1: Visokonivojska arhitektura programsko določenih omrežij [15]

nina je fizično ločena in centralizirana v obliki krmilnika SDN, ki s pomočjo aplikacij določa delovanje podatkovne ravnine celotnega omrežja.

2.2 Zgodovina

Raziskovalci so potrebo po lažjem upravljanju omrežij in hitrejši uvedbi novih storitev prepoznali davno pred rojstvom protokola OpenFlow in samega

izraza ″programsko določena omrežja″ [17]. Prve poskuse ločitve krmilne in podatkovne ravnine lahko zasledimo že v pobudah iz sredine devetdesetih let, kot sta npr. Open Signaling in DCAN (angl. *Devolved Control of ATM Networks*) [21].

Namen projekta DCAN je bil razvoj infrastrukture za prilagodljivo upravljanje in nadzor omrežij ATM (angl. *Asynchronous Transfer Mode*) s predpostavko, da je potrebno krmilne in upravljavske funkcije fizično ločiti iz množice omrežnih naprav. Nadzor naj bi prevzela zunanja entiteta, ki ima podobno vlogo kot krmilnik v programsko določenih omrežjih. Ločitev krmilne programske opreme od posredovalne strojne opreme so predlagali tudi raziskovalci v okviru pobude Open Signaling, saj so ugotovili, da je hitro uvajanje novih omrežnih storitev težavno ravno zaradi zaprtosti ter vertikalne integracije stikal in usmerjevalnikov. Ključni element njihovega predloga je bila vrsta odprtih vmesnikov za nadzor stikal ATM. Na temeljih tega predloga je pod okriljem organizacije IETF (angl. *Internet Engineering Task Force*) nastala tudi specifikacija protokola GSMP (angl. *General Switch Management Protocol*) [22]. Protokol osrednjemu krmilniku omogoča vzpostavljanje in sproščanje povezav na stikalih ATM ter opravljanje množice drugih krmilnih funkcij, ki so sicer v domeni porazdeljenih protokolov na tradicionalnih usmerjevalnikih [21]. Leta 1996 je podjetje Ipsilon Networks predstavilo produkt IP Switch [23] kot rešitev za prenos prometa IP po takratnih omrežjih ATM. IP Switch na zunaj deluje kot usmerjevalnik IP, pri čemer uporablja GSMP za identifikacijo tokov ter dinamično vzpostavitev navideznih zvez (angl. *virtual circuit*) na stikalih ATM. Uporabljen koncept tokov, kjer podatkovni tok predstavlja skupino paketov med dvema končnima točkama, je podoben konceptu mikrotokov v arhitekturi OpenFlow. Glavna omejitev rešitve IP Switch je velikost tabel navideznih zvez, saj vsak tok zahteva uporabo svoje navidezne zveze. Za uporabo v hrbteničnih omrežjih so se uveljavile bolj skalabilne rešitve z uporabo protokola MPLS (angl. *Multiprotocol Label Switching*). [21, 5, 17]

V sredini devetdesetih let je nastala tudi iniciativa aktivnih omrežij (angl.

Active Networks) s konceptom programabilnih stikal. Ta omogočajo prenos ukazov za obdelavo paketov bodisi s pomočjo zunajpasovnega protokola ali pa skupaj s podatki v samih paketih. To naj bi omogočalo dinamično prilagajanje obnašanja omrežja v realnem času, vendar se tehnologija ni obnesla predvsem zaradi varnostnih težav in težav z zmogljivostjo.

Pomembnejšo vlogo pri razvoju programsko določenih omrežij, kot jih poznamo danes, imajo poleg projekta OpenFlow tudi projekti ForCES (angl. *Forwarding and Control Element Separation*), PCE (angl. *Path Computation Element*), 4D in Ethane.

Organizacija IETF je leta 2003 ustanovila delovno skupino ForCES za razvoj istoimenskega projekta, ki definira arhitekturno ogrodje in pripadajoč protokol za standardizirano izmenjavo informacij med krmilno in podatkovno ravnino v omrežnem elementu ForCES [24]. Osnovna ideja je, tako kot pri protokolu OpenFlow, ločitev krmilne in podatkovne ravnine, vendar sta krmilni in podatkovni element vsebovana v sami omrežni napravi. Pristop omogoča centralizacijo nadzora in dopušča umik kontrolnega elementa v drugo napravo. Protokol ni prišel v širšo uporabo, delovna skupina ForCES pa je svoje delo zaključila leta 2014.

Leta 2005 je organizacija IETF začela tudi s standardizacijo arhitekture PCE [25], ki omogoča centralni izračun poti v omrežjih MPLS ter GMPLS za potrebe prometnega načrtovanja. Arhitektura ločuje elemente za izračun poti PCE in odjemalce PCC (angl. *Path Computation Client*). Vlogo centralnega krmilnika oziroma elementa za izračun poti PCE z globalnim pogledom na omrežje ima lahko ena izmed omrežnih naprav ali zunanji strežnik. Za namen komunikacije med elementom PCE in odjemalci PCC so v okviru arhitekture razvili in standardizirali protokol PCEP (angl. *Path Computation Element (PCE) Communication Protocol*) [26]. Ta krmilniku omogoča namestitev tunelov MPLS-TE (angl. *Multiprotocol Label Switching-Traffic Engineering*) na podlagi centralno izračunanih poti. Za sam izračun optimalnih poti pa krmilnik potrebuje podatke o topologiji omrežja in lastnosti povezav. Do teh podatkov lahko pride z izmenjavo sporočil preko notranjega usmerjevalnega

protokola (angl. *Interior Gateway Protocol, IGP*), ki ga uporablja omrežje. Druga možnost je uporaba protokola BGP (angl. *Border Gateway Protocol*) oziroma njegove razširitve BGP-LS (angl. *Border Gateway Protocol-Link State*) [27] za izvoz topologije oziroma prenos podatkov o stanju povezav iz različnih notranjih usmerjevalnih protokolov. Delovna skupina za razvoj PCE je še vedno aktivna, arhitektura pa je zanimiva predvsem zato, ker prinaša nekatere prednosti programsko določenih omrežij z uporabo obstoječe infrastrukture MPLS.

V okviru projekta 4D [28] so avtorji leta 2005 predstavili ideje za omrežno arhitekturo, ki popolnoma odstopa od tradicionalnih omrežij. Namesto uporabe avtonomnih omrežnih naprav so predlagali dekompozicijo omrežja v 4 ravnine, pri čemer so vse krmilne in upravljavske funkcije implementirane v obliki zunanjega neodvisnega sistema. Ta sistem naj bi imel popoln nadzor nad omrežnimi napravami in njihovimi posredovalnimi tabelami. Arhitektura naj bi tako omogočala enostavno definicijo politik na nivoju celotnega omrežja. Projekt 4D tako kot ForCES predstavlja pomemben korak v smer programsko določenih omrežij, kot jih poznamo danes, vendar tudi ta ni doživel dejanskih izvedb, ki bi dokazale njegovo uporabno vrednost.

Izhajajoč iz idej projekta 4D so raziskovalci na univerzah Stanford in Barkley leta 2007 predstavili konkretno rešitev za kontrolo dostopa v poslovnem omrežju, imenovano Ethane [29]. Le-ta omogoča avtentikacijo uporabnikov in njihovo izolacijo v primeru kršenja politik. Rešitev temelji na uporabi centralnega krmilnika, ki nadzoruje delovanje podatkovne ravnine v množici omrežnih naprav z uporabo tokovnih pravil. Stikala neznan promet posredujejo krmilniku, ta pa v tokovne tabele namesti ustrezno tokovno pravilo.

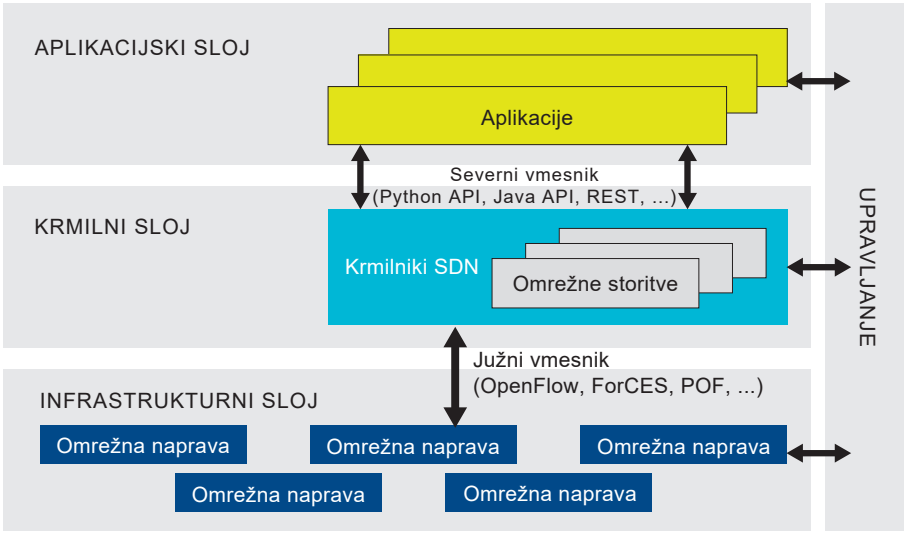
Ethane je služil kot osnova za razvoj projekta OpenFlow, ki so ga prvič predstavili leta 2008 [30]. OpenFlow definira standardni protokol za komunikacijo med krmilno in podatkovno ravnino ter zgradbo stikala, ki vsebuje eno ali več tokovnih tabel. Tokovne tabele vsebujejo tokovna pravila, ki se nanašajo na določeno podmnožico omrežnega prometa. Delovanje podatkovne ravnine v stikalu je tako v celoti odvisno od tokovnih pravil, ki jih

namesti centralni krmilnik. Projekt je bil sprva namenjen izvajanju poskusov za razvoj novih protokolov v akademskih okoljih, vendar je z demonstracijo praktičnih primerov uporabe pritegnil tudi zanimanje industrije. Revija *MIT Technology Review* je leta 2009 OpenFlow označila kot eno izmed desetih najbolj prodornih tehnologij leta, pri čemer so pristop opisali z izrazom ″programsko določena omrežja″ [31]. Leta 2011 so potencial med drugim prepoznali Google, Facebook, Yahoo, Microsoft, Verizon in Deutsche Telekom ter ustanovili organizacijo *Open Networking Foundation* [32] za promocijo in uveljavitev programsko določenih omrežij z razvojem odprtih standardov. Organizacija je prevzela skrb za standardizacijo protokola OpenFlow, ki se je uveljavil kot *de facto* standard za komunikacijo med krmilno in podatkovno ravnino v programsko določenih omrežjih. Danes je standardiziranih že več različic protokola, ki ga podpirajo tako omrežne naprave različnih proizvajalcev kot različne implementacije odprtokodnih in lastniških krmilnikov SDN. ONF šteje že več kot 150 članov, med katerimi so predvsem večji ponudniki telekomunikacijskih ter oblačnih storitev in proizvajalci omrežne opreme. Leta 2016 je združitev z ONF napovedala tudi neprofitna organizacija *Open Networking Lab (ON.LAB)*, pod okriljem katere se med drugim razvija odprtokodni krmilnik programsko določenih omrežij ONOS.

Prve komercialno uspešne rešitve, ki temeljijo na konceptu programsko določenih omrežij, so bile večinoma namenjene virtualizaciji omrežij v navideznih strežniških okoljih. Primer takšnih rešitev sta VMware NSX in Juniper Contrail. V zadnjem času vedno večje zanimanje za programsko določena omrežja kažejo tudi ponudniki oblačnih in telekomunikacijskih storitev. Kljub velikemu zanimanju pa uporabo v produkcijskih okoljih otežujejo različni izzivi, ki jih bomo obravnavali v nadaljevanju tega dela. Google B4 [33] je ob tem ena od redkih realizacij programsko določenih omrežij, ki deluje v produkcijskem okolju. Rešitev temelji na uporabi protokola OpenFlow in s pomočjo centralne aplikacije za prometno načrtovanje omogoča skoraj 100% izkoristek nekaterih relativno dragih povezav med Googlovimi podatkovnimi centri po svetu.

2.3 Arhitektura

Ključne komponente programsko določenih omrežij, to so aplikacije, krmilniki SDN in omrežne naprave, lahko razdelimo v tri sloje, kot prikazuje slika 2.2.



Slika 2.2: Komponente programsko določenih omrežij [34]

2.3.1 Infrastrukturni sloj

Infrastrukturni sloj programsko določenih omrežij sestavlja množica omrežnih naprav [19]. V primerjavi s tradicionalnimi omrežnimi napravami so te poenostavljene, saj ne izvajajo kompleksnih krmilnih funkcij. Skrbijo za obdelavo in posredovanje paketov s pomočjo dobro definiranih ukazov oziroma pravil, ki jih določa zunanji krmilnik preko t. i. južnega vmesnika (angl. *southbound API*). Južni vmesnik definira nabor ukazov podatkovne ravnine in protokol za komunikacijo ter predstavlja ključno orodje za ločitev krmilne in podatkovne ravnine.

Ukazi za procesiranje so običajno podani v obliki tokovnih pravil, ki so nameščena v eno ali več tokovnih tabel. Pravila za posamezen podatkovni tok določajo pripadajočo akcijo, kot je npr. posredovanje prometa na enega ali več izhodnih omrežnih vmesnikov, sprememba polj v zaglavju ali skok na

naslednjo tokovno tabelo. V tem primeru govorimo o tokovno osnovanem procesiranju, kjer podatkovni tok predstavlja skupino paketov med dvema ali več končnimi točkami. Množica izvornih in ciljnih končnih točk je lahko določena s fizičnimi vmesnikom, naslovi MAC ali IP, vrati UDP oziroma TCP in drugimi polji v zaglavju paketov, odvisno od konkretne izvedbe. Ob sprejemu paketa na vhodnem vmesniku naprava poskrbi za razčlenitev polj v zaglavju, dejanske vrednosti polj primerja s pravili v tokovni tabeli, poišče prvo ujemanje in izvede pripadajočo akcijo. Običajno je določena tudi privzeta akcija, ki se izvede v primeru, ko v tabeli, ni ujemajočega pravila. Z namestitvijo ustreznih pravil v tokovne tabele je mogoče implementirati različne omrežne storitve, kot so požarne pregrade, tunelski prehodi ali izenačevalniki bremena. Gre za neke vrste programiranje podatkovne ravnine, zato naprave imenujemo tudi programabilna stikala.

Najbolj razširjen južni vmesnik je OpenFlow, ki poleg namestitve podatkovnih tokov omogoča tudi posredovanje paketov krmilniku za reaktivni način delovanja. Alternativo predstavlja ForCES, vendar v praksi ni doživel širše uporabe. Protokol OVSDB (angl. *Open vSwitch Database Management Protocol*) je v osnovi mišljen kot komplement protokolu OpenFlow za upravljanje stikal Open vSwitch. Dejansko gre za protokol, ki omogoča manipulacijo oddaljenih podatkovnih struktur. Te lahko odvisno od implementacije predstavljajo tudi vsebino posredovalnih tabel. OVSDB poleg stikala Open vSwitch uporabljajo še nekatere druge rešitve. Protokola MPLS-TP [35] in BGP Flowspec [36] sta ravno tako namenjena upravljanju oddaljene podatkovne ravnine v omrežnih napravah. Eden izmed novejših pristopov, ki naj bi omogočil bolj fleksibilno procesiranje paketov, je protokol POF (angl. *Protocol-Oblivious Forwarding*) [37]. Namesto podpore za ujemanje polj poznanih protokolov v zaglavju paketov namreč omogoča ujemanje pravil s pomočjo odmikov in dolžin poljubnih polj, vendar zahteva uporabo prilagojene strojne opreme. Različne omejitve obstoječih protokolov pa naslavljajo še nekatere druge pobude, kot so ROFL, OpFlex, OpenState in PAD. V širšem smislu med južne vmesnike štejemo tudi protokole, ki omogočajo in-

terakcijo s krmilno ali upravljavsko ravnino omrežnih naprav, kot so PCEP, I2RS, NETCONF, OF-config in XMPP.

Vlogo omrežnih naprav programsko določenih omrežij imajo lahko bodisi fizične naprave, kjer je funkcionalnost podatkovne ravnine implementirana v obliki namenske strojne opreme, ali t. i. navidezna stikala (angl. *virtual switch*), ki so v celoti implementirana s pomočjo programske opreme. Tako fizične kot navidezne naprave poleg funkcij krmilne ravnine vključujejo tudi agenta za komunikacijo preko južnega vmesnika.

Navidezna stikala so uporabna predvsem za virtualizacijo omrežij v strežniških okoljih, kjer zagotavljajo povezljivost, ločevanje prometa ter druge omrežne funkcije za navidezne stroje (angl. *virtual machines*) in sistemske vsebnike (angl. *containers*). Navidezna omrežja so lahko tesno povezana s fizičnim omrežjem, vedno bolj pa je razširjena tudi uporaba navideznih prekrivnih omrežij (angl. *overlay virtual networks*). V tem primeru navidezna stikala v strežnikih poskrbijo tudi za enkapsulacijo prometa z različnimi tunelskimi protokoli, fizična omrežna infrastruktura pa zagotavlja zgolj transport preko protokola IP. Na voljo je več odprtokodnih in komercialnih implementacij navideznih stikal OpenFlow, med katerimi so najbolj razširjeni Open vSwitch [32], Indigo [38] in NEC PF1000 [39], ki se uporabljajo tudi v komercialnih rešitvah. Številne druge implementacije se uporabljajo predvsem za prototipiranje in simulacijo omrežij.

Za implementacijo tokovnih tabel navidezna stikala uporabljajo različne podatkovne strukture, kot so zgoščevalne table (angl. *hash tables*) ali iskalna drevesa (angl. *search trees*) [40], vendar je prepustnost programskih rešitev v praksi omejena na nekaj 10 Gb/s. Fizične naprave za procesiranje prometa s hitrostmi do 100 Gb/s na posamezen vmesnik in skupno več kot 3Tb/s v ta namen izkoriščajo namensko strojno opremo. Kompleksna tokovna pravila, ki omogočajo ujemanje velikega števila polj z uporabo nadomestnih znakov (angl. *wildcards*) zahtevajo uporabo večje količine relativno dragega ternarnega vsebinsko naslovljivega pomnilnika (angl. *Ternary Content-Addressable Memory, TCAM*). Ta se sicer uporablja tudi v tradicionalnih omrežnih na-

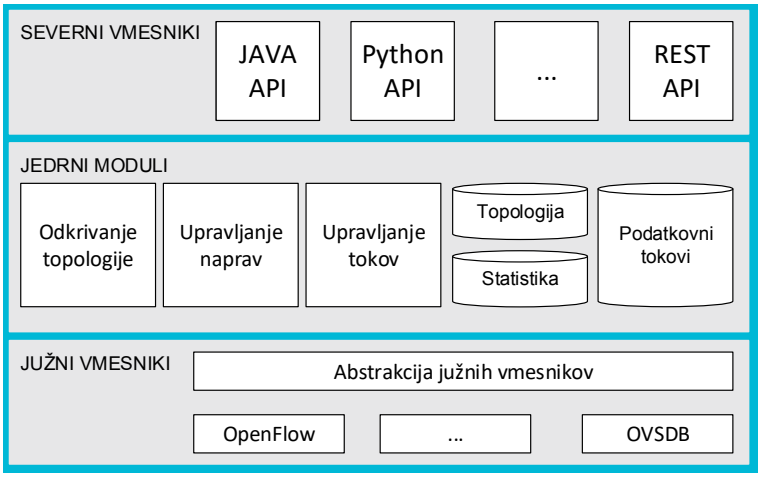
pravah za iskanje najboljšega ujemanja (angl. *longest-prefix match*) naslovov IP, implementacijo usmerjanja na osnovi prometnih filtrov (angl. *policy based routing*) ter seznamov za kontrolo dostopa (angl. *access control lists*), vendar je zaradi visoke cene običajno omejene velikosti, ki zadostuje zgolj za nekaj tisoč zapisov. Proizvajalci se zato poslužujejo različnih tehnik za optimizacijo poizvedovanja. Preprostejša pravila, ki zahtevajo zgolj natančno ujemanje določenih polj, se lahko preslikajo v kombinacijo zapisov običajnih posredovalnih tabel druge in tretje plasti. Za iskanje natančnega ujemanja pri tem zadostuje uporaba cenejšega binarnega vsebinsko naslovljivega pomnilnika (angl. *Binary Content-Addressable Memories, BCAM*) ali celo običajnega pomnilnika z naključnim dostopom (angl. *Random-access memory, RAM*) v kombinaciji z naprednimi algoritmi in podatkovnimi strukturami, kot so Bloomovi filtri [41]. To omogoča cenovno učinkovitejšo izvedbo naprav s podporo za večje število tokovnih pravil. Učinkovita uporaba strojne opreme za implementacijo tokovnih pravil v splošnem predstavlja velik izziv, zato se zmožnosti fizičnih naprav lahko precej razlikujejo.

Večina proizvajalcev omrežne opreme, kot so Cisco, HP, NEC, IBM, Juniper in Extreme, je vsaj nekaterim običajnim stikalom dodala tudi podporo za protokol OpenFlow. Takšna stikala omogočajo tako uporabo tradicionalnih omrežnih protokolov kot tudi delovanje v načinu OpenFlow oziroma kombinacijo obeh načinov. Logiko za oddaljeno krmiljenje podatkovne ravnine lahko namestimo tudi na določena generična stikala (angl. *white-box switches*), ki omogočajo namestitev različnih omrežnih operacijskih sistemov, kot je Cumulus Linux [42]. Omrežni operacijski sistem PICA8 PicOS [43] temelji na operacijskem sistemu Linux in z uporabo odprtokodnega agenta Open vSwitch omogoča interakcijo s podatkovno ravnino preko protokola OpenFlow ali OVSDB. V zadnjem času je na voljo tudi več namenskih naprav OpenFlow, ki so optimizirane za večje število tokovnih pravil. Večina trenutno dostopnih naprav omogoča od nekaj tisoč do nekaj deset tisoč tokovnih pravil, medtem ko nekatera namenska stikala za tokovno procesiranje sprejmejo tudi do milijon tokovnih pravil [44]. Pri tem se je potrebno zavedati,

da je največje število tokovnih pravil odvisno tudi od kompleksnosti pogojev za ujemanje teh pravil.

2.3.2 Krmilni sloj

Krmilni sloj je sestavljen iz enega ali več krmilnikov SDN, ki predstavljajo ključni element programsko določenih omrežij [19]. Gre za programsko opremo, ki teče na običajnih strežnikih in skupaj z aplikacijami opravlja nalogo logično centralizirane krmilne ravnine za množico omrežnih naprav. Krmilnik deluje kot neke vrste posrednik med aplikacijami, ki zahtevajo želeno obnašanje omrežja, in omrežnimi napravami, s pomočjo katerih zagotovi ustrezne omrežne vire. Preko t. i. severnih programskih vmesnikov (angl. *northbound*



Slika 2.3: Tipična zgradba krmilnika SDN [5]

APIs) aplikacijam zagotavlja abstrakcijo omrežnih virov in globalen pogled na omrežje. Pri tem jih obvešča o različnih omrežnih dogodkih, kot so spremembe topologije zaradi odpovedi povezav ali priklop nove končne naprave, in skrbi za izvedbo ter optimizacijo aplikacijskih zahtev z namestitvijo ustreznih tokovnih pravil v omrežne naprave. V ta namen poleg severnih vmesnikov na eni strani in gonilnikov za implementacijo različnih južnih vmesnikov na drugi strani, vključuje različne module za osnovne omrežne funkcije

ter pripadajoče shrambe podatkov, kjer vzdržuje podatke o stanju omrežja. Tipično zgradbo krmilnika lahko vidimo na sliki 2.3. Krmilniki imajo v arhitekturi programsko določenih omrežij podobno vlogo kot tradicionalni operacijski sistemi v računalniških sistemih, zato jih pogosto imenujemo tudi omrežni operacijski sistemi (angl. *network operating systems, NOS*). Njihove distribucije velikokrat že vsebujejo različne aplikacije za konkretne primere uporabe, čeprav v splošnem predstavljajo ogrodje za razvoj poljubnih omrežnih funkcij.

Osnovne omrežne funkcije, ki jih zagotavlja krmilnik, vključujejo odkrivanje omrežne topologije, izračun najkrajših poti v omrežju, upravljanje tokovnih pravil in sledenje omrežnim statistikam [45]. Odkrivanje topologije zajema tako odkrivanje omrežnih naprav in povezav med njimi, kakor tudi odkrivanje povezav s končnimi napravami, ki se priključujejo v omrežje. Krmilnik podatke o omrežnih napravah in njihovih zmožnostih pridobiva preko južnega vmesnika od naprav samih. Povezave med njimi lahko odkriva s pošiljanjem paketov LLDP (angl. *Link Layer Discovery Protocol*) preko vmesnika ene naprave in prejemom istega paketa na vmesnik neke druge naprave ali pa iz zunanjih virov. Na podlagi zbranih podatkov gradi in hrani topologijo celotnega omrežja, iz katere izračuna tudi optimalne poti med različnimi vozlišči. Odkrivanje povezav izvaja periodično ter ob različnih dogodkih, ki vplivajo na spremembo topologije, kot je dodajanje nove naprave ali obvestilo o spremembi stanja vmesnika obstoječe naprave. Ob vsaki spremembi poskrbi za posodobitev poti in ostalih povezanih podatkov ter o spremembi obvesti aplikacije, ki te podatke uporabljajo. Krmilnik vzdržuje tudi bazo tokovnih pravil, ki izhajajo iz zahtev aplikacij in trenutnega stanja omrežja, ter skrbi za sinhronizacijo teh pravil z vsemi omrežnimi napravami. Ob tem beleži različne statistične podatke posameznih tokov, ki jih pridobi iz omrežnih naprav. Poskrbeti mora tudi za ustrezno prioritizacijo tokovnih pravil različnih aplikacij. Poleg omenjenih funkcij lahko krmilnik vključuje tudi nekatere varnostne mehanizme za implementacijo varnostnih politik na nivoju celotnega omrežja ter druge storitve, ki so skupne različnim aplikaci-

jam.

Prvi krmilnik SDN za krmiljenje naprav OpenFlow imenovan NOX [46], so razvili v podjetju Nicira. Napisan je v programskem jeziku C++, njegova programska koda pa je sedaj prosto dostopna. Pozneje so avtorji z namenom lažje uporabe v raziskovalne in izobraževalne namene razvili tudi krmilnik POX, ki je napisan v programskem jeziku Python in omogoča preprostejši razvoj modulov. Nicira je v sodelovanju s podjetjema Google in NTT razvila še lastniški krmilnik Onix [47], ki je služil kot osnova za implementacijo omrežja Google B4 [33] ter komercialne rešitve Nicira NVP za virtualizacijo omrežij v strežniških okoljih [48]. Niciro je prevzel VMware, ki NVP naprej razvija pod imenom VMware NSX. Danes so na voljo številne implementacije tako odprtokodnih kot komercialnih krmilnikov SDN. Večinoma gre za monolitne aplikacije napisane v različnih programskih jezikih. Predstavniki odprtokodnih krmilnikov so med drugim Beacon [49], njegov naslednik Floodlight [50], Trema [51], Ryu [52], OpenContrail [53], OpenDaylight [54], ONOS [55], OpenMUL [56] in RuNOS [57]. Pomembnejšo vlogo ima javanski krmilnik OpenDaylight, ki se razvija pod okriljem organizacije Linux Foundation in predstavlja tudi osnovo za nekatere lastniške rešitve. Poleg protokola OpenFlow na južnem vmesniku podpira različne protokole, kot so NETCONF, BGP in PCEP, za upravljanje tradicionalne omrežne infrastrukture. V zadnjem času je vedno več pozornosti zlasti s strani telekomunikacijskih ponudnikov deležen tudi krmilnik ONOS. Tako OpenDaylight kot ONOS veljata za dominantni rešitvi na področju krmilnikov OpenFlow, vendar se osredotočata na različne primere uporabe. OpenContrail predstavlja nekoliko drugačno rešitev, ki na južnem vmesniku uporablja protokole BGP, XMPP ter NETCONF za upravljanje navideznih omrežnih naprav. Proizvajalci, kot so NEC, HP in IBM ponujajo tudi svoje lastniške implementacije krmilnikov OpenFlow. Pri tem je najbolj aktiven NEC s svojo rešitvijo ProgrammableFlow Controller. Večina ostalih proizvajalcev ponuja zaprte rešitve ali pa OpenFlow podpirajo le delno [5].

V nasprotju s standardiziranimi južnimi vmesniki, kjer prevladuje pro-

tokol OpenFlow, vsak krmilnik ponuja svoj lasten nestandardiziran severni vmesnik za razvoj aplikacij. Večinoma gre za aplikacijske programske vmesnike v programskem jeziku, v katerem je razvit sam krmilnik. Različne aplikacije so tako nameščene kot programski moduli krmilnika. Določene funkcionalnosti krmilniki izpostavljajo tudi preko vmesnikov REST (angl. *Representational State Transfer*) za lažjo integracijo z zunanjimi sistemi. Organizacija ONF je z namenom standardizacije severnih vmesnikov, ki bi omogočili prenosljivost aplikacij med različnimi krmilniki, ustanovila namensko delovno skupino [58]. Rešitev se za zdaj kaže predvsem v standardiziranem modelu in vmesnikih z uporabo mreženja na osnovi namena (angl. *Intent-Based Networking, IBN*) [59]. V praksi se severni vmesniki še vedno razvijajo in sproti prilagajajo potrebam različnih namenov uporabe.

2.3.3 Aplikacijski sloj

Aplikacijski sloj sestavljajo aplikacije oziroma programska oprema, ki služi implementaciji krmilne logike različnih omrežnih storitev [19]. Sem spadajo tako osnovne funkcionalnosti, kot so povezljivost končnih točk na drugi plasti, usmerjanje, dodeljevanje naslovov IP in različni uporabniški vmesniki za upravljanje, kakor tudi naprednejše storitve, kot so požarne pregrade ali izenačevalniki bremena [5]. Želeno obnašanje omrežja lahko aplikacije dosežejo z uporabo severnega vmesnika krmilnika, preko katerega npr. zahtevajo namestitev tokovnih pravil za posredovanje paketov med dvema točkama po optimalni poti, pošiljanje paketa preko določenega vmesnika omrežne naprave, delitev prometa po različnih poteh ali celo njegovo preusmerjanje aplikacijski logiki za potrebe avtentikacije, temeljitega pregledovanja paketov (angl. *Deep Packet Inspection, DPI*) in podobnih nalog povezanih z uveljavljanjem varnostnih politik. Običajno se aplikacije odzivajo na različne dogodke, o katerih so ravno tako obveščene preko severnega vmesnika s pomočjo povratnih funkcij (angl. *callback functions*). To so večinoma spremembe omrežne topologije zaradi odpovedi povezav ali priklopa novih naprav. Odzivajo se lahko tudi na dogodke iz zunanjih virov, kot so usmerjevalni strežniki, sistemi

za nadzor omrežja ali druge poslovne aplikacije.

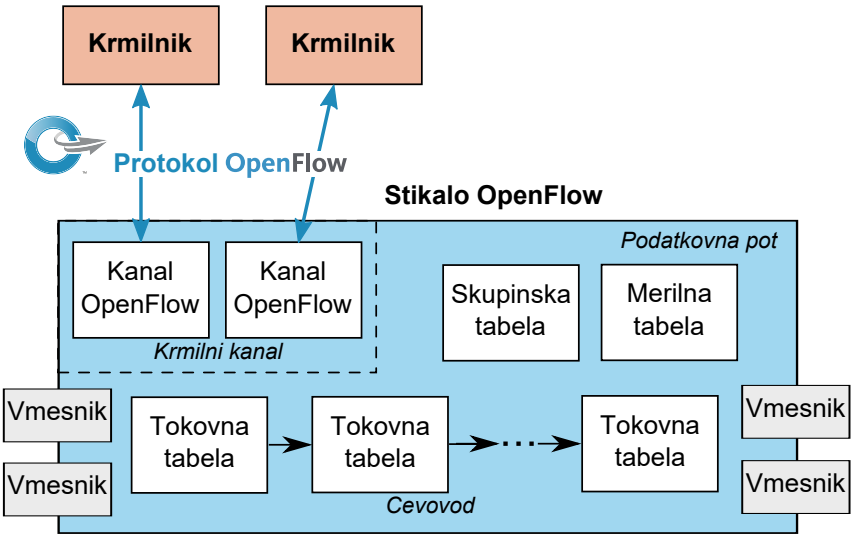
2.4 OpenFlow

OpenFlow [60] je odprt standard, ki definira protokol za komunikacijo med krmilno in podatkovno ravnino ter abstraktni model stikala oziroma obnašanje podatkovne ravnine. Omogoča neposredni dostop in upravljanje podatkovne ravnine s pomočjo tokovnih pravil tako v fizičnih kot navideznih omrežnih napravah.

Standardiziranih je že 6 glavnih različic standarda z oznakami od 1.0 do 1.5, pri čemer ima vsaka izmed njih še nekaj manjših izdaj. V pripravi je že sedma glavna različica z oznako 1.6, vendar večina programske in strojne opreme trenutno podpira različico 1.3 ali starejšo. Različica 1.3 je tudi različica s podaljšano podporo (angl. *long term support*).

2.4.1 Stikalo OpenFlow

Zgradbo stikala OpenFlow lahko vidimo na sliki 2.4. Stikalo je sestavljeno



Slika 2.4: Ključne komponente stikala OpenFlow [60]

iz omrežnih vmesnikov za sprejem in oddajo paketov, ene ali več tokovnih in skupinskih tabel ter krmilnega kanala OpenFlow za komunikacijo s krmilnikom. Komunikacija poteka z uporabo sporočil, ki jih definira protokol OpenFlow. Sporočila omogočajo dodajanje, brisanje in posodabljanje tokovnih pravil ter druge operacije za nadzor podatkovne ravnine.

Stikalo se v vlogi odjemalca povezuje na krmilnik preko protokola TCP na vratih 6653. Opcijsko sta lahko vlogi odjemalca in strežnika tudi zamenjani. Možna je uporaba varnega kanala s protokolom TLS, vendar ta od različice 1.1 ni več obvezna, zato je podpora v nekateri programski in strojni opremi pomanjkljiva [61]. Krmilni kanal je običajno vzpostavljen preko fizično ločenega upravljavskega omrežja, čeprav standard dopušča tudi možnost znotrajpasovnega krmiljenja (angl. *in-band control*). To pomeni, da komunikacija s krmilnikom poteka kar preko programsko določenega omrežja OpenFlow, ki ga sicer upravlja krmilnik. Takšna postavitev je bolj kompleksna, saj je potrebno že pred vzpostavitvijo krmilnega kanala kakor tudi ob vsaki spremembi tokovnih tabel zagotoviti vsaj osnovno povezljivost stikala s krmilnikom.

Vsaka vrstica tokovne tabele predstavlja tokovno pravilo. Kot lahko vi-

Tabela 2.1: Komponente tokovnega pravila [60]

Primerjalna polja	Prioriteta	Števci	Set ukazov	Časovne omejitve	Piškotki	Zastavice

dimo v tabeli 2.1, so tokovna pravila sestavljena iz:

- **Primerjalnih polj**, ki sestavljajo pogoj za ujemanje tokovnega pravila. Prva različica standarda omogoča ujemanje 12-terke (angl. *12-tuple*) polj vhodnega paketa, ki določajo:
  - vhodni vmesnik prejetega paketa,
  - izvorni naslov MAC,
  - ponorni naslov MAC,

  - tip okvirja,
  - identifikator VLAN,
  - prioriteto VLAN,
  - izvorni naslov IP,
  - ponorni naslov IP,
  - različico protokola IP,
  - polje IP ToS (angl. *Type of Service*)
  - izvorna vrata TCP/UDP ali tip ICMP in
  - ciljna vrata TCP/UDP ali kodo ICMP.

  Pogoj je lahko sestavljen iz enega ali več omenjenih polj, pri čemer je možna uporaba nadomestnih znakov. Z vsako novo različico standarda se nabor podprtih polj močno povečuje. Osnovna podpora za protokol IPv6 je bila dodana z različico 1.2. Različica 1.5 dopušča primerjavo več kot 40 polj iz zaglavij poznanih protokolov, kot so značke MPLS, operacijska koda ARP in različne zastavice TCP. Poleg tega je možno tudi ujemanje na podlagi različnih metapodatkov, določenih v predhodnih fazah procesiranja.

- **Prioritete**, ki določa vrstni red primerjave tokovnih pravil.
- **Števcev**, ki so namenjeni zbiranju statistike posameznega podatkovnega toka, kot je število ujemajočih paketov, število ujemajočih oktetov in trajanje podatkovnega toka.
- **Seta ukazov**, ki se izvedejo v primeru ujemanja pravila. Ukazi omogočajo skok na naslednjo tabelo in dodajanje akcij v set akcij, ki se izvede ob koncu cevovodnega procesiranja vsakega paketa oziroma z eksplicitnim ukazom *Apply-Actions*. Set akcij za posamezen paket je na začetku prazen in se pri procesiranju prenaša med tokovnimi tabelami. Lahko vsebuje največ eno akcijo posameznega tipa. Te se

izvedejo v naprej določenem vrstnem redu. Sem spadajo akcije, kot je sprememba polja TTL za pakete IP in MPLS, dodajanje in odstranjevanje značk MPLS, PBB ali VLAN, uveljavljanje različnih politik QoS, procesiranje paketa preko določene skupine v skupinski tabeli in posredovanje paketa na izhodni vmesnik. Izhodni vmesnik je lahko pri tem eden izmed fizičnih omrežnih vmesnikov stikala, logični vmesnik, ki predstavlja npr. agregacijsko skupino ali tunelski vmesnik, ali eden izmed rezerviranih vmesnikov, ki omogočajo izvedbo nekaterih drugih posredovalnih akcij. Rezervirani vmesniki so npr.:

  - **ALL:** Rezerviran vmesnik za posredovanje paketa na vse omrežne vmesnike stikala razen vmesnika, preko katerega je stikalo paket prejelo.
  - **CONTROLLER:** Omogoča posredovanje paketa krmilniku s pomočjo sporočila *PACKET_IN* za potrebe reaktivnega krmiljenja.
  - **LOCAL:** Predstavlja vmesnik lokalnega omrežnega sklada stikala. Uporablja se za znotrajpasovno krmiljenje.
  - **NORMAL:** Namenjen je posredovanju paketa običajnemu cevovodu za procesiranje v hibridnih stikalih, ki omogočajo tudi uporabo tradicionalnih omrežnih protokolov.
  - **FLOOD:** V hibridnih stikalih omogoča poplavljanje paketa s pomočjo običajnega cevovoda za procesiranje, pri čemer se lahko obnašanje razlikuje glede na konkretno implementacijo običajnega cevovoda.

  V kolikor je set ukazov ujemajočega pravila prazen oziroma ne vsebuje akcije za posredovanje na skupino, vmesnik ali drugo tabelo, se paket privzeto zavrže.

- **Časovne omejitve**, ki določa čas, v katerem mora prispeti vsaj en paket, ki se ujema s tem pravilom, če gre za omejitev *idle_timeout*, oziroma mora krmilnik pravilo obnoviti, če gre za omejitev *hard_timeout*,

sicer stikalo to pravilo odstrani iz tokovne tabele.

- **Piškotkov**, ki predstavljajo poljubne vrednosti, namenjene krmilniku za potrebe filtriranja, oziroma izbero tokov za posodobitev ali izbris in ne vplivajo na samo procesiranje.

- **Zastavic**, ki določajo zahteve tokov pri procesiranju. Zastavica *OFPFF_SEND_FLOW_REM* npr. zahteva, da stikalo obvesti krmilnik, v kolikor iz tokovne tabele pobriše pravilo, na katero se zastavica nanaša.

Vsako tokovno pravilo enolično identificirata njegova prioriteta in seznam primerjalnih polj. Običajno tokovna tabela vsebuje tudi pravilo *table-miss* s prioriteto 0 in brez pogojev za ujemanje. To pravilo se izvede v primeru, da ni drugega ujemajočega pravila. V primeru reaktivnega krmiljenja pravilo vključuje akcijo za posredovanje paketa krmilniku, v splošnem pa se lahko izvede tudi katera koli druga akcija npr. posredovanje paketa običajnemu cevovodu za procesiranje ali skok na drugo tabelo. V kolikor krmilnik ne namesti specifičnega pravila *table-miss*, se paketi, ki se ne ujemajo z nobenim pravilom v tabeli, privzeto zavržejo.

Podpora za večje število tokovnih tabel in skoke med njimi je na voljo od različice 1.1 dalje. Uporaba ene same tabele lahko namreč vodi v eksplozijo števila potrebnih pravil za izvedbo ortogonalnega procesiranja paketov npr. na podlagi pravil za kontrolo dostopa, pravil za ločevanje prometa različnih najemnikov in posredovalnih pravil, kjer je končni rezultat kartezični produkt vseh omenjenih pravil. Uporaba več tabel tako omogoča zmanjšanje števila potrebnih zapisov in prinaša večjo fleksibilnost pri procesiranju. Število tabel in njihove zmožnosti so odvisne od konkretne implementacije stikala.

Različica 1.1 je prinesla tudi možnost uporabe skupinskih tabel. Kot

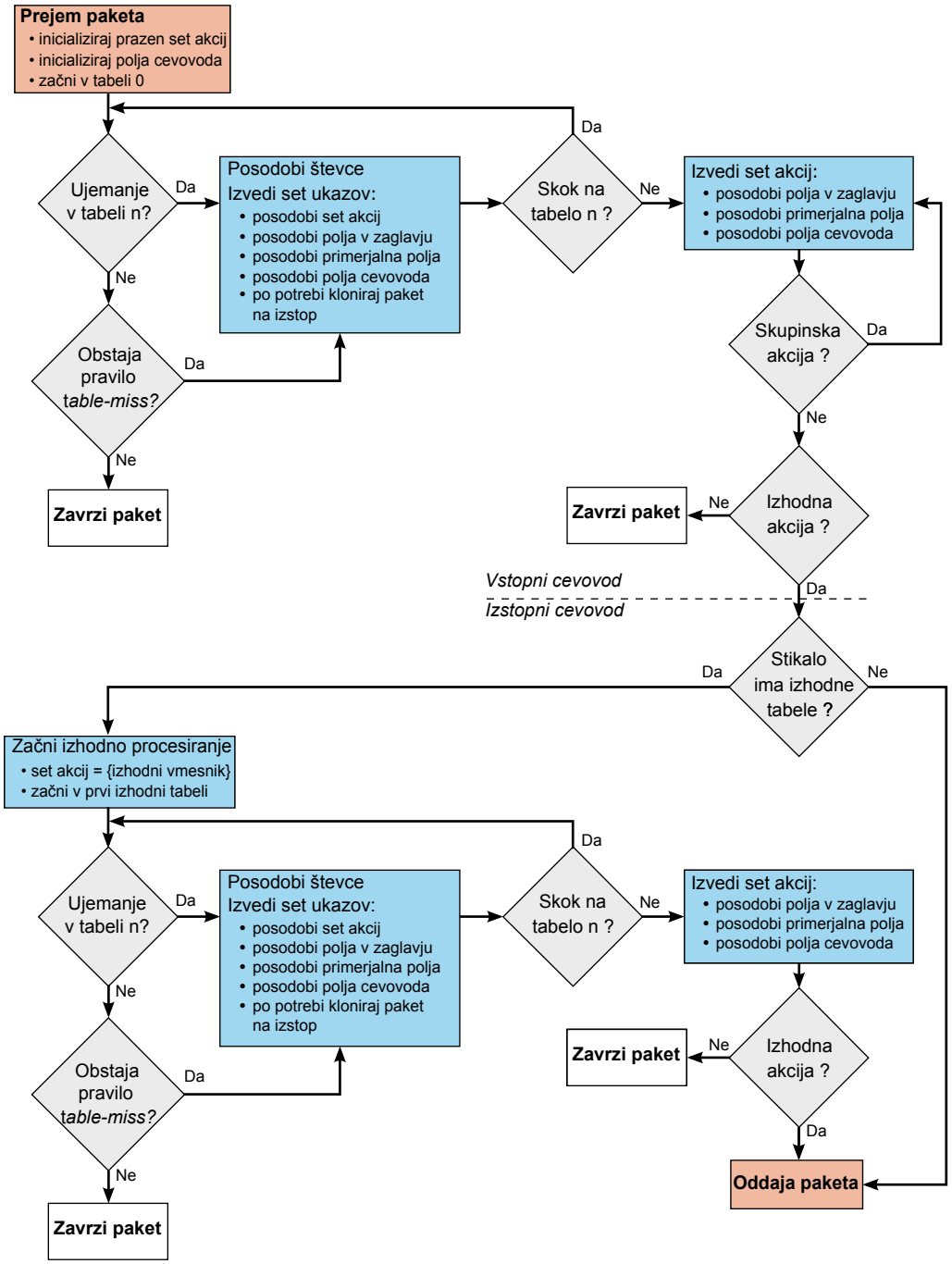
Tabela 2.2: Polja zapisov skupinskih tabel [60]

Identifikator skupine	Tip skupine	Števci	Seznam akcijskih košev

prikazuje tabela 2.2, so zapisi skupinskih tabel sestavljeni iz:

- **Identifikatorja skupine**, ki je 32 bitno nepredznačeno celo število in enolično določa skupino.
- **Tipa skupine**, ki določa semantiko, in sicer:
  - **all:** Določa izvedbo vseh košev v skupini. Uporablja se za implementacijo razpršenega (angl. *broadcast*) posredovanja in posredovanja več prejemnikom (angl. *multicast*).
  - **select:** Določa izvedbo enega izmed košev v skupini. Procesiranje paketa se izvede s setom akcij koša, ki je izbran na podlagi izbirnega algoritma, npr. rezultata enosmerne zgoščevalne funkcije nad določeno skupino polj. To omogoča izvedbo deljenja prometa med več enakovrednih poti oziroma izhodnih vmesnikov, pri čemer se lahko ob izpadu posameznega vmesnika uporabi preostanek delujočih vmesnikov brez posredovanja krmilnika. Algoritem lahko upošteva tudi različne uteži košev, vendar konfiguracija in stanje izbirnega algoritma nista del standarda OpenFlow.
  - **indirect:** Določa izvedbo edinega koša v skupini, saj lahko skupina vsebuje zgolj en akcijski koš. Namenjena je enolični identifikaciji seta akcij, na katerega lahko kaže več tokovnih ali skupinskih zapisov. Vsebinsko gre za enak konstrukt kot skupina *all* z enim samim košem.
  - **fast failover:** Določa izvedbo prvega živega koša v skupini in s tem omogoča spremembo posredovanja brez posega krmilnika. Skupina potrebuje mehanizem za preverjanje živosti (angl. *liveness*) posameznih košev.
- **Števcev**, ki so namenjeni zbiranju statistike posamezne skupine.
- **Seznama akcijskih košev**, kjer vsak koš vsebuje set akcij s pripadajočimi parametri.

Merilne tabele so na voljo od različice 1.3 dalje in vsebujejo različne merilne zapise, ki so sestavljeni iz enega ali več merilnih pasov. Omogočajo



Slika 2.5: Poenostavljen diagram poteka procesiranja paketa [60]

implementacijo različnih enostavnih operacij za zagotavljanje kakovosti storitev, v kombinaciji s čakalnimi vrstami posameznih vmesnikov pa tudi implementacijo kompleksnih ogrodij kakovosti storitev, kot je DiffServ.

Ob prejemu paketa na enem izmed omrežnih vmesnikov stikalo vedno prične s procesiranjem v prvi tokovni tabeli. Nadaljnji koraki so odvisni od ukazov, ki jih določa ujemajoče pravilo v tej tabeli. Poenostavljen diagram poteka procesiranja paketa v stikalu OpenFlow je prikazan na sliki 2.5.

Stikalo tekom procesiranja povečuje različne števce za potrebe beleženja statistike tako na nivoju celotnih tokovnih, merilnih in skupinskih tabel, kot tudi na nivoju posameznih tokov, skupin, meril, merilnih pasov, vmesnikov, čakalnih vrst in akcijskih košev. Zahtevani so zgolj nekateri tipi števcev, medtem ko je prisotnost ostalih števcev odvisna od implementacije stikala.

2.4.2 Protokol OpenFlow

Protokol OpenFlow definira tri tipe sporočil, in sicer sporočila krmilnik proti stikalu (angl. *controller-to-switch*), asinhrona (angl. *asynchronous*) in simetrična (angl. *symmetric*) sporočila, pri čemer ima vsak tip več podtipov sporočil.

Sporočila krmilnik proti stikalu

Sporočila krmilnik proti stikalu so sporočila, pri katerih komunikacijo prične krmilnik:

- Sporočili *FEATURES_REQUEST* in *FEATURES_REPLY* krmilniku omogočata, da identificira stikalo in pridobi osnovne podatke o njegovih zmožnostih, kot je število tokovnih tabel.

- Nastavitvena sporočila omogočajo poizvedbo in nastavljanje določenih parametrov stikala.

- Sporočila za spremembo stanja omogočajo dodajanje, brisanje in posodabljanje zapisov v tokovnih, skupinskih in merilnih tabelah. Najbolj

pogosto je sporočila tipa *FLOW_MOD*, ki je namenjeno manipulaciji tokovnih pravil.

- Sporočila za branje stanja omogočajo branje vrednosti števcev, nastavitev in zmožnosti stikala. Večina teh sporočil je sestavljenih iz sekvence večdelnih sporočil, ki omogočajo prenos večje količine podatkov.

- Sporočilo *PACKET_OUT* omogoča, da krmilnik stikalu posreduje vsebino paketa, ki ga želi poslati iz določenega vmesnika stikala. Običajno gre za vračanje paketa, ki ga je stikalo poslalo v obdelavo aplikacijski logiki krmilnika s paketom *PACKET_IN*. Sporočilo mora vsebovati tudi navodila za procesiranje posredovanega paketa. V kolikor stikalo omogoča medpomnjenje paketov, se lahko namesto vsebine paketa posreduje zgolj identifikator medpomnilnika, kjer stikalo hrani paket, ki ga je predhodno posredovalo krmilniku s sporočilom *PACKET_IN*. Krmilnik lahko vsebino paketa generira tudi samostojno in jo posreduje stikalu npr. za potrebe odkrivanja topologije.

- S sporočilom *BARRIER_REQUEST* krmilnik zahteva, da stikalo izvede vsa predhodno poslana sporočila za spremembo stanja preden odgovori s sporočilom *BARRIER_REPLY*. Na ta način krmilnik izvede sinhronizacijo v primeru, da želi poslati sporočila za spremembo stanja, ki so odvisna od uspešne izvedbe predhodnih sprememb.

- Z različico protokola 1.2 ali novejšo, ki omogoča povezovanje z več krmilniki hkrati, lahko krmilnik spremeni svojo vlogo s sporočilom *ROLE_REQUEST*, na katerega stikalo odgovori z *ROLE_REPLY*. Več o tem si bomo pogledali v poglavju 4 o porazdeljenih krmilnikih v kontekstu pristopov za povezovanje stikala na več kot eno instanco krmilnika.

- Sporočila za konfiguracijo asinhronih sporočil krmilniku omogočajo nastavitev filtrov za tipe asinhronih sporočil, ki jih želijo prejemati od posameznega stikala.

Asinhrona sporočila

Asinhrona sporočila so sporočila, pri katerih komunikacijo prične stikalo brez poizvedovanja krmilnika:

- Sporočilo *PACKET_IN* je namenjeno posredovanju vsebine paketa aplikacijski logiki krmilnika. V primeru reaktivnega krmiljenja stikalo na ta način krmilniku posreduje vse pakete, za katere ni ujemajočega pravila v tokovni tabeli. Krmilnik odgovori z namestitvijo ustreznega pravila s sporočilom *FLOW_MOD* in paket vrne z uporabo sporočila *PACKET_OUT*. V kolikor stikalo omogoča medpomnjenje paketov, lahko stikalo krmilniku pošlje le del paketa, ki je potreben za procesiranje, in identifikator medpomnilnika, ki ga krmilnik uporabi pri odgovoru. Tudi v primeru proaktivnega krmiljenja lahko poljuben promet npr. za potrebe avtentikacije ali odkrivanja topologije preusmerimo krmilniku z uporabo tokovnega pravila, ki pakete posreduje na rezerviran vmesnik *CONTROLLER*. Ujemajoči paketi se ravno tako posredujejo krmilniku z uporabo sporočila *PACKET_IN*.

- S sporočilom *FLOW_REMOVED* stikalo obvesti krmilnik o brisanju tokovnega pravila, ki je označeno z zastavico *OFPFF_SEND_FLOW_REM*. Brisanje je lahko posledica potekle časovne omejitve ali eksplicitnega brisanja s sporočilom *FLOW_MOD*.

- Sporočila *PORT_STATUS* omogočajo obveščanje krmilnika o spremembi stanja vmesnikov bodisi zaradi spremembe konfiguracije ali izpada povezave.

Simetrična sporočila

Simetrična sporočila so sporočila, ki jih lahko pošiljata tako krmilnik kot stikalo:

- Sporočilo *HELLO* se uporablja za vzpostavitev povezave in dogovor o uporabljeni različici protokola OpenFlow.

- Sporočilo *ECHO_REQUEST* zahteva odgovor s sporočilom *ECHO_REPLY* in se pošilja periodično za preverjanje živosti povezave med stikalom in krmilnikom. Omogoča tudi meritev zakasnitve in pasovne širine.

- Sporočilo *ERROR* je namenjeno obveščanju o napakah.

- Rezervirana sporočila *EXPERIMENTER* omogočajo lastniške razširitve protokola.

2.5 Primeri uporabe

Zasledimo lahko vedno več primerov uporabe programsko določenih omrežij, pri čemer večina aplikacij spada v eno izmed sledečih kategorij: centralizirano prometno načrtovanje, varnost, nadzor omrežja, omrežja podatkovnih centrov ter fiksna in mobilna dostopovna omrežja [4].

2.5.1 Centralizirano prometno načrtovanje

Prometno načrtovanje oziroma optimizacija omrežnih poti z namenom učinkovitejše izrabe omrežnih virov je eden izmed problemov, ki je najlažje rešljiv s pomočjo osrednje točke, ki ima pregled nad celotnim omrežjem. Tradicionalni usmerjevalni protokoli večinoma temeljijo na izbiri najkrajših poti, ki z vidika razpoložljivih kapacitet ali drugih parametrov niso nujno optimalne. Poleg tega so povezave v prostranih omrežjih (angl. *wide area networks, WAN*) relativno drage in zahtevajo prekomerno zagotavljanje kapacitet (angl. *overprovisioning*) z namenom maskiranja različnih odpovedi. Ves promet namreč obravnavajo enakovredno, medtem ko imajo poslovne aplikacije zelo različne zahteve po pasovni širini in zakasnitvah pri prenosu. Upravljavci omrežij se zato poslužujejo različnih tehnik za prometno načrtovanje, pri čemer tradicionalne tehnologije, kot je MPLS-TE, nimajo globalnega pogleda na omrežje in ne omogočajo optimalne rešitve. Aplikacije za prometno

načrtovanje izkoriščajo globalen pogled na omrežje in fleksibilnost programsko določenih omrežij za optimizacijo omrežnih poti ter dinamično dodeljevanje pasovne širine. Pri tem upoštevajo razpoložljivost kapacitet in zakasnitev posameznih povezav kakor tudi dejanske potrebe in prioritete različnih poslovnih aplikacij.

Google je z implementacijo centralne aplikacije za prometno načrtovanje v njihovem prostranem omrežju B4 [33] dosegel skoraj 100% izkoriščenost nekaterih povezav oziroma več kot 75% povprečno izkoriščenost vseh povezav med njihovimi podatkovnimi centri. To jim prinaša znatne prihranke v primerjavi z uporabo tradicionalnih prostranih omrežij, kjer so povezave izkoriščene zgolj 30-40% in zahtevajo ustrezno večje kapacitete. Hkrati so dosegli tudi hitrejšo konvergenco omrežja v primeru izpadov posameznih povezav. Pri tem so namesto dragih namenskih usmerjevalnikov za prostrana omrežja uporabili običajna stikala, namenjena uporabi v podatkovnih centrih, ki jih krmilijo s pomočjo protokola OpenFlow in prilagojenega lastniškega krmilnika Onix. Vzporedno z aplikacijo za prometno načrtovanje uporabljajo tudi aplikacijo za usmerjenje po najkrajših poteh, ki usmerjevalne poti BGP in ISIS izmenjuje z ostalo omrežno infrastrukturo s pomočjo odprtokodne implementacije usmerjevalnih protokolov Quagga. Aplikacija zagotavlja osnovno povezljivost IP, ki služi prenosu tunelskega prometa aplikacije za prometno načrtovanje. Hkrati omogoča posredovanje prometa po najkrajši poti v primeru težav z aplikacijo za prometno načrtovanje. Gre za eno izmed prvih in verjetno največjih implementacij programsko določenih omrežij z uporabo protokola OpenFlow v produkcijskem okolju. Microsoft razvija podobno rešitev v okviru projekta SWAN [62], na voljo pa so tudi različne komercialne rešitve, ki omogočajo optimizacijo prostranih omrežij v poslovnih okoljih, npr. Nuage SD-WAN [63].

2.5.2 Varnost

Programsko določena omrežja odpirajo različne možnosti tudi na področju implementacije varnostnih rešitev. Večina rešitev izkorišča prednosti cen-

tralnega krmilnika za uveljavljanje varnostnih politik na nivoju celotnega omrežja z namestitvijo tokovnih pravil, ki dovoljujejo oziroma preprečujejo določen tip prometa na robnih napravah, kamor so priključeni končni uporabniki oziroma strežniki. S preusmerjanjem prometa aplikacijski logiki lahko aplikacije uporabnika oziroma napravo prehodno tudi overijo. Med drugim so možne implementacije različnih sistemov za zaznavanje in preprečevanje vdorov ali preprečevanje porazdeljenih napadov za prekinitve storitev (angl. *Distributed Denial of Service, DDoS*). V ta namen pridejo do izraza predvsem različne omrežne statistike, ki jih beleži centralni krmilnik, saj aplikacijam omogočajo identifikacijo anomalij v omrežju, ter možnost filtriranja neželenega prometa čim bližje izvora. Filtriranje z uporabo tokovnih pravil je pri tem lahko bolj selektivno kot običajno oddaljeno proženje črnih lukenj (angl. *Remotely Triggered Black Hole, RTBH*) z uporabo protokola BGP.

2.5.3 Omrežja podatkovnih centrov

Nekatere aplikacije za uporabo v podatkovnih centrih so namenjene optimizaciji posredovanja na nivoju fizične infrastrukture, predvsem pa srečujemo rešitve za virtualizacijo omrežij [48] v navideznih strežniških okoljih. VMware NSX [64] je komercialna rešitev, ki se v ta namen že uporablja v produkcijskih okoljih. Poleg povezljivosti navideznih strojev, ki je z uporabo navideznih prekrivnih omrežij v celoti ločena od fizične omrežne infrastrukture, nudi tudi različne napredne omrežne storitve. To so med drugim izenačevalniki bremena, navidezna zasebna omrežja, veriženje storitev in zagotavljanje varnosti po principu mikrosegmentacije, kjer uveljavljanje varnostnih politik poteka na nivoju posameznih delovnih bremen, npr. navideznih strojev in ne celotnega omrežnega segmenta. Varnostne politike so pri tem izvedene z namestitvijo tokovnih pravil na navidezna stikala, preko katerih ta bremena komunicirajo. Podobne koncepte za izstavitev in upravljanje navidezne omrežne infrastrukture z integracijo v oblačno platformo OpenStack uporabljata tudi odprtokodni aplikaciji SONA za krmilnik ONOS in NetVirt za krmilnik OpenDaylight. Tovrstne rešitve naj bi predstavljale manjkajoč

člen pri implementaciji programsko določenih okolij [65], kjer je virtualizacija računskih in shranjevalnih virov že uveljavljena praksa, medtem ko so bile možnosti virtualizacije omrežnih virov s tradicionalnimi pristopi omejene.

2.5.4 Fiksna in mobilna dostopovna omrežja

Za ponudnike telekomunikacijskih storitev je uporaba konceptov programsko določenih omrežij zanimiva predvsem v kombinaciji s komplementarno tehnologijo virtualizacije omrežnih funkcij (angl. *Network Functions Virtualization, NFV*). Gre za tehnologijo, kjer funkcionalnost namenskih omrežnih naprav realiziramo s pomočjo programske opreme v obliki navideznih strojev, sistemskih vsebnikov ali druge tehnologije, ki lahko teče na običajnih strežnikih [66]. SDN pri tem služi za veriženje storitev in centralno upravljanje celotne infrastrukture. V okviru projekta CORD (Central Office Re-architected as a Datacenter) [67] organizacije ONF se razvija agilna platforma za dostavo telekomunikacijskih storitev s ciljem preoblikovanja funkcijskih lokacij (angl. *Central Office, CO*) ponudnikov storitev v podatkovne centre, kjer se uporabljajo zgolj generična stikala OpenFlow v arhitekturi listov in hrbtenic (angl. *Leaf-and-Spine*) ter običajni strežniki. Platforma združuje uporabo orkestratorja XOS, oblačne platforme OpenStack za izstavljanje računskih virov in krmilnika programsko določenih omrežij ONOS skupaj z različnimi aplikacijami za upravljanje fizične omrežne infrastrukture ter virtualiziranih omrežnih funkcij, ki tečejo na oblačni infrastrukturi. Na ta način naj bi platforma omogočila nudenje storitev, ki jih sicer omogočajo širokopasovni robni prehodi (angl. *Broadband Network Gateway, BNG*), optična dostopovna vozlišča (angl. *Optical Line Termination, OLT*) in druge namenske omrežne naprave v fiksnih dostopovnih omrežjih. Pri razvoju in testiranju posameznih komponent omenjenih rešitev sodelujejo tudi večji ponudniki storitev, kot so AT&T, NTT, Comcast, Telefónica in Deutsche Telekom. Podobne rešitve so zanimive tudi za uporabo v mobilnih omrežjih in so ravno tako predvidene v okviru projekta CORD. Poleg tega se predlogi, kot je SoftRAN [68], ukvarjajo z optimizacijo uporabe frekvenčna spektra, dode-

ljevanja radijskih virov in učinkovitejšega prehoda uporabnikov med baznimi postajami z uporabo centraliziranih aplikacij za nadzor radijskega omrežja.

2.6 Izzivi

Uporaba pravih programsko določenih omrežij v produkcijskih okoljih je kljub velikemu potencialu in vedno večjemu zanimanju še vedno dokaj redka, saj njihovo uvedbo ovirajo različni izzivi. Eden izmed razlogov je prav gotovo velika mera sprememb, s katerimi se je potrebno soočiti tako z vidika uporabljene strojne in programske opreme kot tudi z vidika potrebnih znanj za implementacijo. Problem lahko nekoliko omili uporaba različnih strategij za postopno uvajanje programsko določenih omrežij tam, kjer ta prinašajo največ koristi, pri čemer interakcija s preostalo omrežno infrastrukturo poteka transparentno z uporabo tradicionalnih omrežnih protokolov. Kljub temu so lahko začetni vložki zelo veliki. Tudi Google se je realizacije konceptov programsko določenih omrežij lotil postopoma [33]. Začeli so z razvojem krmilne platforme in aplikacije za usmerjanje po najkrajših poteh, s pomočjo katere so lahko v svojem prostranem omrežju posamezne usmerjevalnike nadomestili s stikali OpenFlow v arhitekturi listov in hrbtenic. Pozneje so rešitev nadgradili z aplikacijo za prometno načrtovanje, o kateri smo nekaj več povedali v prejšnjem poglavju. Trenutno v okviru projekta Espresso podobne koncepte uvajajo tudi v delu omrežja, ki skrbi za komunikacijo z zunanjim svetom [69].

V preteklosti je bil eden izmed večjih izzivov tudi dostopnost strojne opreme s podporo oddaljenega upravljanja podatkovne ravnine. Danes so na voljo programabilna stikala različnih proizvajalcev, vendar se njihove zmožnosti precej razlikujejo. Kljub temu da med južnimi vmesniki dominira protokol OpenFlow, ki je dobro definiran, se ta še vedno razvija in dopušča številne implementacijske razlike. Kot smo omenili pri predstavitvi infrastrukturnega sloja programsko določenih omrežij, so že razlike v kapacitetah tabel za tokovna pravila lahko precejšnje. Poleg tega prihaja tudi

do velikih razlik pri številu pravil, ki jih lahko stikalo namesti na časovno enoto ter podprtih opcij, ki so odvisne od različice protokola. Z možnostjo uporabe več povezanih tabel, ki sicer omogočajo bolj fleksibilno procesiranje, le-te običajno niso več popolnoma generične, temveč predstavljajo različne fizične tabele stikala, zato se mora krmilnik zavedati njihove strukture. Vse to je potrebno vzeti v obzir tako pri izbiri strojne opreme kot tudi pri razvoju same rešitve. Dodaten izziv predstavlja tudi pomanjkanje standardizacije na področju severnih vmesnikov, ki onemogoča prenosljivost aplikacij. Določeni meri vertikalne integracije se tako ni mogoče izogniti.

Resnejši pomisleki pri uvedbi programsko določenih omrežij se nanašajo predvsem na skalabilnost in razpoložljivost omrežja zaradi centralizacije krmilne ravnine. V nasprotju s tradicionalnimi omrežji, ki so visoko decentralizirana in kot taka odporna na odpovedi posameznih komponent, saj se lahko omrežne naprave avtonomno odločajo o izbiri alternativnih poti v omrežju, če te obstajajo, je razpoložljivost programsko določenih omrežij odvisna tudi od razpoložljivosti osrednjega krmilnika, ki tako predstavlja kritično točko odpovedi. Poleg tega mora zmogljivost krmilnika zadoščati za izvedbo krmilnih funkcij celotnega omrežja. Za zagotavljanje skalabilnosti in visoke razpoložljivosti sistema je kljub logični centraliziranosti modela pomembna fizična porazdeljenost krmilnika [4, 6]. Z implementacijo krmilnika v obliki porazdeljenega sistema lahko zagotovimo nemoteno delovanje tudi v primeru odpovedi posameznih instanc krmilnika ali razdelitev v omrežju. Obenem lahko izkoristimo računsko moč večjega števila fizičnih strežnikov. Temeljni problem pri implementaciji porazdeljenih sistemov, odpornih na napake, predstavlja problem soglasja [7]. V nadaljevanju si bomo pogledali več o algoritmih soglasja, ki ta problem rešujejo in tako predstavljajo ključno komponento za zagotavljanje visoke razpoložljivosti krmilnikov, s tem pa tudi samih programsko določenih omrežij.

Poglavje 3

Porazdeljeni sistemi

3.1 Osnovne lastnosti

Porazdeljeni sistemi so sestavljeni iz vozlišč (angl. *nodes*), običajno so to komponente programske oziroma strojne opreme, ki so povezana preko omrežja in komunicirajo ter usklajujejo svoje delovanje zgolj z izmenjavo sporočil [7]. Vozlišča so lahko pri tem prostorsko ločena s poljubno razdaljo. To zajema celo vrsto sistemov, ki uporabljajo množico omrežnih računalnikov za dosego skupnega cilja. Vsi ti sistemi imajo za posledico določene pomembne skupne lastnosti, in sicer:

- **Sočasnost izvajanja** (angl. *concurrency*): Programska koda se izvaja sočasno na vseh vozliščih. Sočasnost omogoča povečanje zmogljivosti z dodajanjem novih vozlišč, vendar zahteva koordinacijo pri uporabi deljenih virov.

- **Odsotnost globalne ure** (angl. *no global clock*): Sinhronizacija ur posameznih vozlišč je zaradi zakasnitve pri izmenjavi sporočil in omejene natančnosti ur računalniških sistemov mogoča samo do določene mere natančnosti. Posamezna vozlišča nimajo dostopa do globalne ure, s katero bi lahko natančno določila zaporedje dogodkov, ki se zgodijo na preostalih vozliščih, kar otežuje koordinacijo.

- **Neodvisnost odpovedi** (angl. *failures independence*): Vsak računalniški sistem lahko odpove, zato je treba pri načrtovanju sistema to upoštevati. Po eni strani večje število vozlišč povečuje verjetnost, da vsaj eno izmed njih odpove na tak ali drugačen način, vendar so odpovedi neodvisnih vozlišč med seboj neodvisne. S pravilno zasnovo sistema, ki omogoča delovanje tudi v primeru napak na določenem številu vozlišč, je mogoče to lastnost izkoristiti za izboljšanje zanesljivosti celotnega sistema. Pri tem je potrebno upoštevati tudi možnosti napak, ki so specifične za porazdeljene sisteme, npr. napake in zakasnitve na komunikacijskih povezavah, preko katerih vozlišča izmenjujejo sporočila.

Glavni cilj implementacije storitev v obliki porazdeljenih sistemov je običajno povečanje zmogljivosti in razpoložljivosti. Zaželena je tudi skalabilnost celotnega sistema.

3.1.1 Skalabilnost

Skalabilnost je zmožnost sistema, da se prilagaja povečanim zahtevam po obdelavi na predvidljiv način, pri tem pa sistem ne postane prezapleten, predrag ali nemogoč za upravljanje. Poznamo različne dimenzije skalabilnosti, in sicer:

- **Obremenitvena**: Porazdeljen sistem mora dovoljevati preprosto povečevanje oziroma zmanjševanje števila virov, da zadosti večji ali manjši obremenitvi.
- **Geografska**: Sistem mora ohranjati uporabnost in odzivnost ne glede na oddaljenost uporabnikov in virov.
- **Administrativna**: Dodajanje kapacitet naj ne bi bistveno povečalo stroškov administracije.

Porazdeljeni sistemi porabljajo del svojih virov za zagotavljanje skalabilnosti. V praksi za vsak sistem obstaja omejitev največje rasti za vsako od omenjenih dimenzij.

3.1.2 Povečevanje zmogljivosti

Zmogljivost je določena s količino uporabnega dela, ki ga sistem opravi glede na porabljen čas in sistemske vire. Zmogljivostne metrike so odvisne od konkretne domene uporabe. Odvisno od konteksta lahko povečanje zmogljivosti pomeni npr. večjo prepustnost ali zmanjšanje odzivnega časa. Z uporabo zmogljivejše strojne opreme je mogoče zmogljivosti sistema povečati tudi v primeru, ko storitev teče na enem samem fizičnem vozlišču. To imenujemo vertikalno skaliranje (angl. *scale-up*) storitev. V določeni meri je uporaba zmogljivejše strojne opreme preprostejša od implementacije porazdeljenega sistema, vendar moči strojne opreme ne moremo povečevati v nedogled. Z večanjem zahtev v določeni točki uporaba zmogljivejše strojne opreme postane enostavno predraga ali pa sploh ni več mogoča. Takrat je smiselno horizontalno skaliranje (angl. *scale-out*) oziroma povečanje zmogljivosti z dodajanjem novih vozlišč, ki zahteva implementacijo porazdeljenega sistema.

3.1.3 Povečevanje razpoložljivosti

Razpoložljivost je odstotek časa, ko je storitev dostopna z razumno zakasnitvijo [7]. Odvisna je od zanesljivosti celotnega sistema, torej od verjetnosti za pojav napake in časa, ki je potreben za vzpostavitev normalnega delovanja. Želeno razpoložljivost lahko definiramo s številom devetk, kot vidimo v tabeli 3.1, ki za različne nivoje razpoložljivosti prikazuje, kakšen je lahko

Tabela 3.1: Nivoji razpoložljivosti storitve

Razpoložljivost		**Nedostopnost/leto**	**Nedostopnost/teden**
90%	(”one nine”)	36,5 dni	16,8 ure
99%	(”two nines”)	3,65 dni	1,68 ure
99.9%	(”three nines”)	8,76 ure	10,1 min
99.99%	(”four nines”)	52,56 min	1,01 min
99.999%	(”five nines”)	5,26 min	6,05 s
99.9999%	(”six nines”)	31,5 s	604,8 ms

največji dovoljen čas nedostopnosti storitve na letni in tedenski ravni.

O visoki razpoložljivosti govorimo takrat, kadar je razpoložljivost blizu 100%. Konkretne zahteve se lahko v različnih produkcijskih sistemih razlikujejo in so navadno podane v obliki dogovora na ravni storitev (angl. *Service-Level Agreement*). Običajno gre za najmanj 99,9% razpoložljivost. Z uporabo enega samega vozlišča je takšen nivo razpoložljivosti zelo težko ali nemogoče doseči, saj smo omejeni z zanesljivostjo posameznih komponent strojne in programske opreme. Poleg tega lahko največji dovoljen čas nedostopnosti storitve hitro presežemo že z rednim vzdrževanjem. V tem primeru je nujna implementacija storitve v obliki porazdeljenega sistema odpornega na napake (angl. *fault-tolerant distributed system*). Takšen sistem lahko nemoteno deluje tudi v primeru določenih napak npr. sesutja posameznih vozlišč ali razdelitev v omrežju. Mera in tip dopuščenih napak sta pri tem odvisna od njegove zasnove, ki največkrat predstavlja kompromis med razpoložljivostjo, zmogljivostjo in količino potrebnih strojnih virov za izvedbo.

Ker povečanje zanesljivosti sistema temelji na predpostavki, da so odpovedi posameznih vozlišč med seboj neodvisne, je pomembno, da tudi na fizičnem nivoju sistem ne vsebuje kritičnih točk odpovedi. Vozlišča morajo predstavljati najmanj fizično ločene strežnike. Ravno tako je potrebno poskrbeti za ustrezno redundanco napajanja, hlajenja in drugih sistemov, od katerih je odvisno delovanje vseh vozlišč. V določenih primerih je za večjo mero neodvisnosti zaželena tudi geografska porazdeljenost vozlišč.

3.2 Temeljni modeli

Porazdeljeni sistemi so namenjeni uporabi v realnem svetu, kjer morajo v najrazličnejših okoliščinah zagotavljati pravilno delovanje s čim večjo mero zmogljivosti in razpoložljivosti. Pri tem jih omejujejo različni fizični dejavniki, ki jih moramo za dosego želenih rezultatov upoštevati pri njihovem načrtovanju. Temeljni modeli nam pomagajo pri razumevanju in sprejema-

nju različnih načrtovalskih odločitev znotraj teh omejitev. Predstavljajo formalni opis skupnih lastnosti, ki so ključnega pomena za obnašanje sistema, pri čemer nudijo abstrakcijo nepotrebnih podrobnosti. S temeljnimi modeli eksplicitno izpostavimo vse pomembne predpostavke sistema in ugotavljamo, kaj je pod danimi predpostavkami sploh izvedljivo ali neizvedljivo. Izvedljivost določene zasnove sistema lahko preverimo s tem, da preverimo, ali v danem sistemu držijo vse relevantne predpostavke. Jasno izražene predpostavke predstavljajo tudi osnovo za matematično dokazovanje lastnosti. Za razumevanje omejitev pri načrtovanju porazdeljenih sistemov sta pomembna predvsem model interakcije in model napak.

3.2.1 Model interakcije

Obnašanje in stanje preprostega programa je odvisno od algoritma, ki določa zaporedje korakov za dosego določenega cilja. Program se izvaja na enem vozlišču, medtem ko so porazdeljeni sistemi sestavljeni iz množice vozlišč. Njihovo stanje in obnašanje opisujejo porazdeljeni algoritmi, ki določajo zaporedje korakov za vsako izmed vozlišč, vključno z izmenjavo sporočil, ki služijo prenosu informacij in koordinaciji aktivnosti. Pri analizi in načrtovanju porazdeljenih sistemov smo na te interakcije še posebej pozorni. Modeli interakcije se osredotočajo predvsem na omejitve, ki izhajajo iz lastnosti komunikacijskih kanalov in dejstva, da vozlišča nimajo dostopa do globalne ure ali skupnega pomnilnika.

Lastnosti komunikacijskih kanalov

Komunikacijski kanali so lahko izvedeni na različne načine običajno s prenosom sporočil preko računalniškega omrežja. Pomembni so njihova zakasnitev (angl. *latency*), pasovna širina (angl. *bandwidth*) in trepetanje (angl. *jitter*). Zakasnitev je čas od oddaje sporočila na eni strani do prejema na drugi strani. Minimalna zakasnitev je pogojena s svetlobo hitrostjo in oddaljenostjo vozlišč. Poleg tega je odvisna od časa, ki je potreben za procesiranje sporočila tako na poti kot tudi v omrežnem skladu enega in drugega vozlišča,

preden se sporočilo dostavi ciljnemu procesu. Na zakasnitev lahko ravno tako vpliva obremenitev omrežja še posebej v okoljih, kjer je prenosni kanal deljen. Trepetanje je varianca zakasnitve, pasovna širina pa predstavlja količino podatkov, ki jo lahko prenesemo v določeni časovni enoti.

Globalni čas in ure

Vsak računalnik v porazdeljenem sistemu ima svojo interno uro, ki jo lokalni procesi uporabljajo za določanje trenutnega časa. Procesi, ki tečejo na različnih računalnikih, lahko svoje dogodke sicer označijo s časovnimi žigi (angl. *timestamps*), vendar obstaja velika verjetnost, da časovna žiga za dogodka, ki sta se v istem trenutku zgodila na dveh različnih računalnikih, ne bosta enaka. Ure v posameznih napravah namreč ne tečejo s popolnoma enako frekvenco, zato sčasoma pride do odstopanja med njimi, tudi če v določeni točki vse ure v porazdeljenem sistemu uskladimo. Odstopanje frekvence ure od idealne referenčne ure imenujemo drsenje ure (angl. *clock drift*). Stopnje drsenja (angl. *drift rate*) pa se od naprave do naprave razlikujejo.

Poznamo različne pristope za usklajevanje ur računalniških sistemov. Z uporabo satelitskih sprejemnikov GPS (angl. *Global Positioning System*) lahko računalnik določi čas z natančnostjo mikrosekunde. Potrebno je upoštevati, da sprejem signala GPS znotraj stavb ni mogoč, predvsem pa uporaba sprejemnikov GPS za vsak računalniški sistem ni stroškovno smiselna in praktična za uporabo. Običajno računalniki svojo uro periodično usklajujejo preko omrežja z izmenjavo sporočil z namenskimi časovnimi strežniki, ki točen čas pridobivajo iz zunanjih virov, kot so cezijevi oscilatorji ali sprejemniki GPS. Na natančnost takšnega usklajevanja vplivajo predvsem zakasnitve pri izmenjavi sporočil oziroma njihova varianca.

Sinhronost sistema

V porazdeljenih sistemih težko določimo časovne meje za izvedbo operacij, dostavo sporočil ali drsenje ure. Za potrebe njihove analize običajno operi-

ramo z dvema preprostima modeloma, ki imata nasprotne časovne predpostavke, in sicer:

- **Sinhron model** (angl. *Synchronous model*)
  V sinhronem porazdeljenem sistemu veljajo naslednje predpostavke [70]:

  - Poznamo zgornjo in spodnjo časovno mejo za izvedbo vsakega koraka procesa.
  - Poznamo zgornjo in spodnjo časovno mejo za prenos vsakega sporočila preko komunikacijskega kanala.
  - Vsak proces ima lokalno uro z omejeno stopnjo drsenja.

  Reševanje problemov v sinhronem sistemu je zaradi časovnih predpostavk lažje. Možno je npr. zaznavanje odpovedi z uporabo časovnikov. V realnih sistemih sicer lahko predvidimo časovne meje za izvedbo procesnih korakov in prenos sporočil ter stopnjo drsenja ure, vendar težko določimo točne vrednosti. V kolikor vrednosti mej ne moremo garantirati, bo sistem, katerega zasnova temelji na teh mejah, nezanesljiv. Modeliranje algoritmov z uporabo modela sinhronega sistema je kljub temu koristno, saj nam poda osnovno idejo o obnašanju algoritma v realnem sistemu, če bo le-ta večino časa deloval sinhrono.

- **Asinhron model** (angl. *Asynchronous model*)
  Asinhron porazdeljen sistem ne vsebuje nobenih časovnih predpostavk. To pomeni, da ne poznamo mej za hitrost izvajanja procesa, zakasnitev pri prenosu sporočil ali stopnje drsenja ure. Pojem časa pravzaprav ne obstaja in se nanj ne moremo zanašati. Določanje zaporedja dogodkov z uporabo realnega časa ali zaznavanje napak z uporabo časovnikov tako ni mogoče. Z uporabo asinhronega modela lahko modeliramo porazdeljene sisteme, kot je internet, kjer nobena od teh lastnosti v resnici ni zagotovljena. Izkaže se, da je določene probleme mogoče rešiti tudi brez

časovnih predpostavk. Takšne rešitve so bolj robustne, saj so odporne na različne časovne težave v realnih sistemih.

Za realne sisteme ne moremo predpostavljati, da so popolnoma sinhroni, saj so podvrženi napakam in nedoločenosti z vidika časa, ki je potreben za izvajanje določenih operacij in dostavo sporočil. Po drugi strani je reševanje mnogih problemov v popolnoma asinhronih sistemih nemogoče. Modeliranje realnih sistemov je zato smiselno z uporabo delno sinhronih modelov [71], ki predpostavljajo, da se sistem vsaj nekaj časa obnaša sinhrono, pri tem pa dopuščajo intervale asinhronosti.

Vrstni red dogodkov

Večkrat nas zanima ali se je določen dogodek na enem vozlišču zgodil pred, po oziroma istočasno kot nek dogodek na drugem vozlišču. S pravilnim vrstnim redom operacij je običajno določena pravilnost delovanja sistema, vrstni red dogodkov pa je v pomoč tudi pri reševanju konfliktnih zahtev. Izvajanje porazdeljenega sistema lahko z dogodki in njihovim vrstnim redom opišemo tudi brez natančnega poznavanja globalne ure z uporabo logičnega časa oziroma logičnih ur (angl. *logical clocks*) [72], ki temeljijo na uporabi števcev operacij ter izmenjavi sporočil.

3.2.2 Model napak

V porazdeljenih sistemih lahko pride do odpovedi (angl. *failures*) tako vozlišč oziroma procesov kot tudi komunikacijskih kanalov med njimi. O odpovedi govorimo takrat, ko vozlišče ali komunikacijski kanal zaradi napačnega stanja (angl. *error*) ne deluje več tako, kot bi moral v skladu s specifikacijo. Vzrok napačnega stanja je napaka (angl. *fault*). Z modelom napak definiramo in klasificiramo napake z namenom analize njihovih učinkov in načrtovanja sistemov, ki lahko delujejo pravilno tudi takrat, ko se napake pojavijo. To so t. i. robustni sistemi oziroma sistemi odporni na napake. Glede na vedenje ločimo več tipov napak, in sicer:

- **Izpustitvene napake** (angl. *omission failures*) so napake, pri katerih komponenta ne izvede zadane naloge. Poseben primer izpustitvene napake procesa oziroma vozlišča je njegovo sesutje (angl. *crash*). To pomeni, da je proces prenehal z delovanjem in ne bo izvajal nadaljnjih korakov. Če predpostavimo, da lahko procesi odpovejo zgolj s sesutjem, je načrtovanje sistema poenostavljeno. Ostali procesi morajo odpoved zaznati in ukrepati s porazdelitvijo opravil na druga vozlišča. Zaznavanje sesutja je mogoče z uporabo časovnikov, ki se po določenem času iztečejo, vendar potrebujemo vsaj delno sinhron sistem. V asinhronem sistemu namreč ne moremo ločiti med sesutjem procesa in njegovim počasnim odzivom oziroma morebitnimi zakasnitvami pri komunikaciji. Izpustitvene napake komunikacijskih kanalov so napake, pri katerih komunikacijski kanal sporočila ne dostavi ciljnemu vozlišču.

- **Bizantinske napake** (angl. *Byzantine failures*) predstavljajo najhujši tip napak, ki lahko vodijo v poljubno napačno stanje sistema. Bizantinska napaka procesa je napaka, pri kateri lahko proces samovoljno izpusti določene korake procesiranja ali samovoljno izvede napačne. Proces lahko pri tem na zahtevo odgovori z napačnimi podatki, ali pa napačne podatke shrani. Tovrstne napake so največkrat povzročene zlonamerno in jih je težko zaznati zgolj na podlagi odgovora enega samega vozlišča. Do bizantinskih napak lahko pride tudi na komunikacijskih kanalih, če se poslano sporočilo npr. okvari med samim prenosom ali večkrat dostavi ciljnemu sistemu.

- **Časovne napake** (angl. *timing failures*) so napake do katerih pride zaradi neusklajenosti ur ali nepričakovanih zakasnitev pri izmenjavi sporočil oziroma izvedbi procesnih korakov. Običajno so posledica preobremenitve omrežja ali računalniškega sistema in so relevantne za sinhrone sisteme, kjer veljajo časovne predpostavke.

Maskiranje napak

Komponente porazdeljenih sistemov so običajno sestavljene iz množice preprostejših komponent. Poznavanje lastnosti napak nam omogoča, da s pravilno zasnovo sistema maskiramo napake posameznih komponent in zgradimo sistem, ki je na te napake odporen. Pri maskiranju lahko napake bodisi skrijemo ali pa jih prevedemo na bolj sprejemljiv tip napake.

Bizantinske napake komunikacijskih kanalov lahko v primeru, ko te niso zlonamerne, spremenimo v eno izmed izpustitevnih napak npr. tako, da z uporabo kontrolnih vsot zavržemo okvarjena sporočila, sporočila dostavljena v napačnem vrstnem redu pa pred predajo procesu razvrstimo na podlagi sekvenčnih številk. Poleg tega lahko občasne izpustitvene napake pri komunikaciji skrijemo z uporabo protokolov za potrditev prejema in ponovno pošiljanje. Iz temeljnega rezultata [73], ki je povezan s problemom dveh generalov, lahko vidimo, da zanesljiva komunikacija preko nezanesljivega kanala v osnovi sicer ni mogoča, lahko pa to lastnost zagotovimo že z relativno šibkimi predpostavkami o zanesljivosti omrežja. V praksi lahko implementacijo porazdeljenih algoritmov precej poenostavimo s predpostavko, da je na voljo zanesljiv komunikacijski kanal, ki ga zagotovimo z uporabo protokola TCP (angl. *Transmission Control Protocol*) preko omrežja, kjer so napake zgolj občasne. Po potrebi pri tem z uporabo protokola TLS zagotovimo, da je kanal tudi varen. Porazdeljeni algoritmi se tako osredotočajo predvsem na maskiranje odpovedi samih procesov s tehnikami, kot je replikacija, medtem ko morajo z vidika napak komunikacijskih kanalov zagotoviti predvsem pravilno delovanje v primeru razdelitev v omrežju. Do teh lahko še vedno pride, če komunikacijski kanal v celoti odpove, medtem ko vozlišča oziroma procesi še vedno delujejo.

3.3 Nezmožnost soglasja v asinhronem sistemu

Varnost (angl. *safety*) in živost (angl. *liveness*) sta temeljni lastnosti pravilnosti (angl. *correctness*) porazdeljenih sistemov. Njun koncept je prvi

vpeljal Leslie Lamport v [74]. Neformalno ju lahko definiramo kot:

- lastnost **varnosti**, ki zagotavlja, da se v sistemu ne bo zgodilo nič slabega in
- lastnost **živosti**, ki zagotavlja, da se bo sčasoma zgodilo nekaj dobrega.

Vsak porazdeljen sistem zagotavlja določeno mero varnosti in živosti. Njuno razmerje oziroma kompromis med živostjo in varnostjo je predmet številnih raziskav zlasti od predstavitve t. i. rezultata nezmožnosti FLP (angl. *FLP impossibility result*) [75], v katerem so Fischer, Lynch in Paterson dokazali, da v asinhronem sistemu noben deterministično algoritem ne more garantirati soglasja že ob odpovedi enega samega vozlišča. To velja tudi ob predpostavki, da lahko vozlišča odpovejo zgolj s sesutjem in da se sporočila ne morejo izgubiti.

Problem soglasja (angl. *consensus problem*) je temeljni problem porazdeljenih sistemov, odpornih na napake, pri katerem se mora množica procesov oziroma vozlišč z izmenjavo sporočil odločiti za neko skupno vrednost, ki je potrebna za izvajanje porazdeljenega algoritma. Lahko gre za odločitev o izvedbi oziroma razveljavitvi transakcije v porazdeljeni podatkovni bazi, določitev vrstnega reda operacij, replikacijo dnevniškega zapisa, izbiro zmagovalca volitev ali poljubne druge vrednosti, ki je relevantna za konkreten primer uporabe. Določeno število vozlišč je lahko pri tem nedostopnih ali v katerem od drugih napačnih stanj, odvisno od tega, kakšen tip napak dopušča konkretna rešitev. Na ta način lahko vozlišča tudi ob prisotnosti napak vzdržujejo konsistentno stanje in delujejo kot skladen sistem. Formalno mora algoritem soglasja za vsako izvedbo izpolnjevati naslednje pogoje [7]:

- **Soglasnost** (angl. *Agreement*): Odločitev vseh pravilno delujočih procesov je enaka.
- **Veljavnost** (angl. *Validity*): Če vsi pravilno delujoči procesi predlagajo enako vrednost, se mora vsak pravilno delujoč proces zanjo tudi odločiti.

- **Končnost** (angl. *Termination*): Sčasoma vsak pravilno delujoč proces doseže odločitev.

Soglasnost in veljavnost predstavljata lastnosti varnosti, medtem ko je končnost lastnost živosti.

Dokaz nezmožnosti soglasja v asinhronem sistemu [75] izhaja iz problema detekcije napak v sistemu, kjer sta zakasnitev sporočil in čas za izvedbo operacij lahko poljubna. Proces bi moral v določenem trenutku, ko čaka na odgovor drugega procesa, bodisi čakati še naprej, dokler ne bi prejel odgovora, ali pa domnevati, da je drugi proces v napačnem stanju, in nadaljevati brez njegovega odgovora. V prvem primeru bi se lahko zgodilo, da se algoritem ne bi nikoli končal, če bi drugi proces dejansko odpovedal, medtem ko bi drug primer lahko privedel do tega, da bi se različni procesi odločili za različne vrednosti. Varnosti in živosti algoritma soglasja tako pod predpostavkami asinhronega sistema ni mogoče zagotoviti, kar pa ne pomeni, da je problem v praksi nerešljiv. Rešitev posledično vodi v kompromis med živostjo in varnostjo v situacijah, ko predpostavljene časovne meje ne držijo.

Eden izmed pristopov za reševanje tega problema v praksi je namreč predpostavka o delno sinhronem sistemu, ki je dovolj šibkejši od sinhronih sistemov, da ga lahko uporabimo kot model realnega sistema, a hkrati zagotavlja minimalne časovne predpostavke za rešljivost problema soglasja z dovolj veliko verjetnostjo. Za rešljivost problema soglasja v delno sinhronem sistemu je ključna ločitev varnosti in živosti algoritma [71]. Praktični algoritmi soglasja tako zagotavljajo, da varnost ni kršena niti v času asinhronosti sistema, torej takrat, ko so zakasnitve izven predvidenih mej, medtem ko je končnost zagotovljena zgolj v primeru, da se procesi odzivajo znotraj določenih predvidenih mej. To je mogoče zagotoviti z različnimi tehnikami, kot je maskiranje napak, detekcija napak z uporabo časovnikov ter randomizacija. Detekcija napak z uporabo časovnika je v delno sinhronem sistemu mogoča s pomočjo lokalne ure, ki sicer ni nujno usklajena s preostalimi urami, vendar lahko služi izteku maksimalnega predvidenega časa za odgovor, po katerem proces domneva, da je drugi proces odpovedal in njegovih sporočil ne upošteva več.

Uporabo teh tehnik si bomo v razdelku 3.6 pogledali na praktičnem primeru algoritma soglasja Raft.

Število napak, ob katerih lahko sistem še vedno doseže soglasje, je odvisno od njegove zasnove. Minimalno število vozlišč, s katerimi lahko zgradimo sistem, ki dopušča določeno število napak, je pogojeno z modelom sistema in tipom dopuščenih napak. V delno sinhronem sistemu potrebujemo za odpornost na f bizantinskih napak najmanj $3f+1$ vozlišč oziroma najmanj $2f+1$ vozlišč, če predpostavimo, da lahko vozlišča odpovedo zgolj s sesutjem [71]. Takšnim sistemom pravimo, da so *f-odporni* (angl. *f-resilient*). V nadaljevanju se bomo osredotočili na sisteme, ki dopuščajo zgolj odpovedi s sesutjem. Takšni sistemi so v praksi največkrat uporabljeni za splošno namensko zagotavljanje visoke razpoložljivosti storitev. Algoritmi, ki zagotavljajo bizantinsko odpornost na napake (angl. *Byzantine fault tolerance, BFT*), so namreč bistveno kompleksnejši tako za implementacijo kot samo izvajanje. Uporabni so za posebne namene, kot so misijsko kritični (angl. *mission critical*) sistemi npr. sistemi za kontrolo letenja ali sistemi ničelnega zaupanja (angl. *zero-trust*). Primer sistemov ničelnega zaupanja so omrežja porazdeljenih kripto valut, ki temeljijo na uporabi verige blokov (angl. *blockchain*), kjer ključ za zagotavljanje bizantinske odpornosti predstavlja dokazilo o delu (angl. *Proof-of-Work, PoW*).

3.4 Teorem CAP in PACELC

Teorem CAP, imenovan tudi Brewerjev teorem, izhaja iz domnev profesorja Erica Brewerja o omejitvah pri implementaciji porazdeljenih sistemov v kontekstu spletnih storitev. Eric je svoje domneve prvič predstavil leta 2000 na Simpoziju principov porazdeljenega računalništva [76], medtem ko sta jih pozneje formalno dokazala Seth Gilbert in Nancy Lynch [77]. Kratica CAP predstavlja tri lastnosti, in sicer [76]:

- **Konsistentnosti** (angl. *consistency,* ***C***): Konsistentnost podatkov vseh vozlišč je ekvivalentna eni sami kopiji podatkov. To pomeni, da

branje podatka iz poljubnega vozlišča vedno vrne rezultat zadnjega pisanja ali napako.

- **Razpoložljivost** (angl. *availability,* ***A***): Sistem mora na vsako zahtevo sčasoma odgovoriti, pri čemer odgovor ni nujno rezultat zadnjega pisanja.

- **Particijska toleranca** (angl. *partition tolerance,* ***P***): Sistem mora delovati pravilno tudi ob prisotnosti razdelitev v omrežju, torej takrat, ko so zaradi napak komunikacijskih kanalov vozlišča ločena na dve ali več skupin, kjer je komunikacija mogoča zgolj znotraj posamezne skupine ne pa tudi med vozlišči v različnih skupinah.

Teorem pravi, da v porazdeljenih sistemih z deljenimi podatki ni mogoče istočasno garantirati vseh treh lastnosti, zato je pri implementaciji potreben kompromis. Pri tem je pomembno, da konsistentnost v kontekstu teorema CAP predstavlja strogo linearizabilno konsistentnost, medtem ko v splošnem poznamo tudi šibkejše modele konsistentnosti, kot je eventuelna konsistentnost. Model stroge konsistentnosti edini zagotavlja, da porazdeljen sistem deluje ekvivalentno sistemu, ki teče na enem samem vozlišču, pri čemer ločimo še:

- zaporedno konsistentnost (angl. *sequential consistency*), kjer je zaporedje operacij nad podatki enako na vseh vozliščih, in

- linearizabilno konsistentnost (angl. *linearizable consistency*), kjer je zaporedje operacij nad podatki ne samo enako na vseh vozliščih temveč tudi skladno z zaporedjem zahtev v globalnem realnem času.

Poleg tega je potrebno poudariti, da teorem CAP podobno kot rezultat FLP predpostavlja asinhron sistem. Če poskušamo teorem CAP umestiti v širši kontekst porazdeljenih sistemov, lahko vidimo, da gre pravzaprav za poseben primer kompromisa med živostjo in varnostjo nezanesljivih porazdeljenih sistemov [8]. Konsistentnost je v tem primeru lastnost varnosti, medtem ko je razpoložljivost lastnost živosti.

Teorem CAP ponazarja pomembne omejitve pri praktični implementaciji različnih porazdeljenih sistemov, vendar v resnici omejuje manjši del njihove zasnove [9]. Odpornost sistema na razdelitve v omrežju je lastnost, ki je v primeru porazdeljenega sistema nujna, saj pri komunikaciji preko omrežja vedno obstaja možnost, da pride do razdelitev. Pri implementaciji se moramo tako odločiti, ali bo sistem poleg odpornosti na razdelitve vedno zagotavljal strogo konsistentnost podatkov ali razpoložljivost. Kompromis je v resnici potreben le takrat, ko se sistem obnaša asinhrono, torej takrat, ko so v omrežju dejansko prisotne razdelitve. V praksi to pomeni, da lahko glede na obnašanje v primeru razdelitev zgradimo dva tipa porazdeljenih sistemov, in sicer:

- Sisteme tipa **CP** (konsistentnost in particijska toleranca), ki v primeru razdelitev žrtvujejo razpoložljivost, zato da zagotovijo strogo konsistentnost podatkov. V to kategorijo običajno spadajo vsi sistemi, ki za izvajanje operacij zahtevajo večinsko sklepčnost (angl. *majority quorum*), kar pomeni, da skupine, ki v danem trenutku ne vsebujejo več kot polovice vozlišč, niso razpoložljive.

- Sisteme tipa **AP** (razpoložljivost in particijska toleranca), ki v primeru razdelitev žrtvujejo strogo konsistentnost podatkov. Več skupin vozlišč lahko vrača in spreminja podatke, kljub temu da med seboj ne morejo komunicirati, zato lahko pride do divergence stanja. Sistem lahko omejuje operacije, ki so dovoljene v času razdelitve, da omeji stopnjo divergence. Po odpravi razdelitve mora poskrbeti za uskladitev vseh skupin in odpravo morebitnih konfliktov. V ta namen običajno poleg samega stanja vozlišča hranijo tudi zgodovino operacij in njihove časovne žige.

Ker so razdelitve v omrežju zlasti znotraj podatkovnih centrov relativno redke, lahko oba tipa večino časa zagotavljata tako strogo konsistentnost kot razpoložljivost.

Teoretično lahko v kontekstu teorema CAP govorimo tudi o sistemih tipa

CA, vendar bi tak sistem deloval pravilno zgolj na enem samem vozlišču, kjer razdelitve niso mogoče. Če smatramo, da obstaja tudi porazdeljen sistem tipa CA in v primeru razdelitev zgolj ne zagotavlja razpoložljivosti, gre pravzaprav za sistem tipa CP.

3.4.1 Teorem PACELC

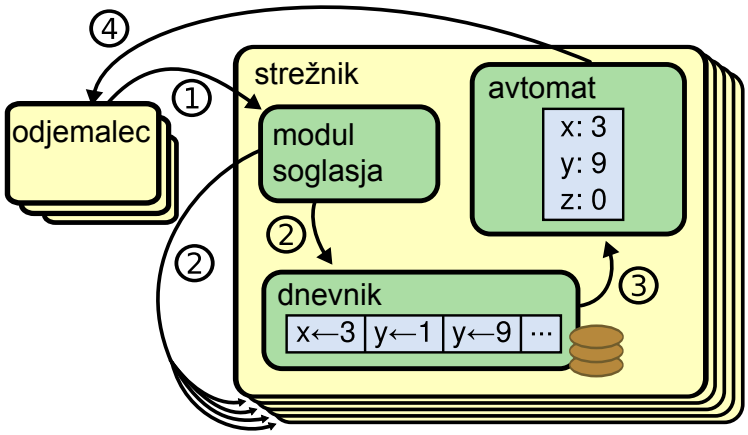
Teorem PACELC [10] predstavlja razširitev teorema CAP za primer normalnega delovanja. Zagotavljanje stroge konsistentnosti podatkov v porazdeljenem sistemu omejuje njegovo zmogljivost, saj sta za vsako operacijo potrebna bodisi izmenjava sporočil z večino vozlišč za dosego soglasja ali vsaj posredovanje zahteve vodilnemu vozlišču, kar ni optimalno z vidika zakasnitev in izrabe računskih virov. Problem predstavlja predvsem replikacija v prostranih omrežjih, kjer so lahko zakasnitve relativno velike. Teorem PACELC poleg omejitev teorema CAP, ki se nanašajo na razdelitve v omrežju, navaja, da je kompromis potreben tudi v primeru normalnega delovanja, in sicer kompromis med nizko zakasnitvijo ter strogo konsistentnostjo. Z uporabo šibkejših konsistentnih modelov lahko zmanjšamo zakasnitev, saj si lahko dovolimo branje in pisanje s pomočjo manjše množice vozlišč. Nekateri sistemi omogočajo tudi izbiro parametrov, s katerimi določimo, koliko vozlišč je potrebnih za branje in pisanje glede na želeno razmerje med zakasnitvijo in konsistentnostjo podatkov.

3.5 Algoritmi soglasja

Replikacija avtomatov (angl. *state machine replication*) z uporabo algoritmov soglasja je uveljavljen pristop za implementacijo strogo konsistentnih storitev, odpornih na napake [78]. Algoritmi soglasja omogočajo, da skupina vozlišč oziroma strežnikov deluje kot skladen sistem tudi ob odpovedi posameznih komponent. V tem primeru gre za izvajanje identičnega zaporedja operacij na poljubnem determinističnem avtomatu vsakega strežnika. Pristop omogoča reševanje številnih problemov porazdeljenih sistemov, kot je

izbira vodje, upravljanje članstva v skupinah, deljenje in podvajanje podatkov, odkrivanje storitev, zaklepanje deljenih virov ali upravljanje konfiguracije.

Implementacija repliciranega avtomata je mogoča s pomočjo repliciranega dnevnika (angl. *replicated log*) [16], kot lahko vidimo na sliki 3.1. Vsak



Slika 3.1: Arhitektura repliciriranega avtomata [16]

strežnik hrani dnevnik z nizom ukazov oziroma operacij, ki jih mora avtomat izvršiti v določenem vrstnem redu. Vsi dnevniki vsebujejo iste ukaze v istem vrstnem redu, zato je izvajanje avtomatov na vseh vozliščih enako. Ker v ta namen uporabljamo deterministične avtomate, so tudi izhodi in končna stanja enaka.

Naloga algoritma soglasja je, da poskrbi za konsistentno stanje vseh dnevnikov. Modul soglasja sprejema zahteve odjemalcev in z izmenjavo sporočil z moduli soglasja preostalih strežnikov zagotovi, da sčasoma vsi dnevniki vsebujejo iste zahteve v istem vrstnem redu. Šele ko posamezno zahtevo v svoj dnevnik zapiše in potrdi večina strežnikov, lahko strežniki zahtevo tudi izvedejo. Modul nato odgovor posreduje odjemalcu. Množica strežnikov tako na zunaj deluje kot en sam visoko razpoložljiv avtomat, ki zagotavlja poljubno storitev. Večina praktičnih algoritmov soglasja ima naslednje lastnosti:

- Zagotavljajo varnost za vse nebizantinske napake, vključno z razdelitvami v omrežju, nepredvidenimi zakasnitvami, izgubo paketov in podvojenimi ali napačno razvrščenimi sporočili.

- Zagotavljajo razpoložljivost dokler deluje in med seboj komunicira več kot polovica vseh strežnikov, saj za izvedbo operacij zahtevajo večinsko sklepčnost. Pri tem predpostavljajo, da lahko posamezni strežniki odpovedo kvečjemu s sesutjem, dopuščajo pa njihovo obnovitev in ponovno pridružitev gruči. Za odpornost na f napak potrebujemo *2f+1* strežnikov, kar je tudi minimum za dosego soglasja v delno sinhronem sistemu. Gruča petih strežnikov lahko tako nemoteno deluje ob odpovedi največ dveh strežnikov.

- Za zagotavljanje konsistentnega stanja niso odvisni od realnega časa. Neusklajene ure ali zelo velike zakasnitve lahko v najslabšem primeru povzročijo nerazpoložljivost sistema.

- Izvedba operacije je mogoča takoj, ko jo potrdi večina strežnikov v gruči, zato manjšina počasnejših strežnikov ne vpliva na skupno zmogljivost sistema.

V splošnem sta v uporabi dva pristopa za dosego soglasja v porazdeljenih sistemih [16]. To sta simetrični in asimetrični pristop. Pri simetričnem pristopu imajo vsi strežniki enako vlogo, zato lahko klienti zahteve pošljejo poljubnemu strežniku, ki prične z njihovo replikacijo na preostale strežnike. Pri asimetričnem pristopu ima v vsakem trenutku eden izmed strežnikov vodilno vlogo, kar pomeni, da lahko sprejema in razvršča zahteve odjemalcev, medtem ko jih preostali strežniki zgolj potrjujejo in izvršujejo. To omogoča dekompozicijo problema na dva dela, in sicer na protokol za izbiro vodje oziroma volitve in protokol normalnega delovanja, pri čemer je lahko protokol normalnega delovanja precej poenostavljen. Izbira vodje je potrebna zgolj ob samem zagonu in ob morebitnem izpadu vodilnega strežnika. Ker vse zahteve potujejo čez vodilni strežnik, je mogoče enostavno razvrščanje zahtev

v skladu z realnim časom. V praksi tak pristop omogoča večjo učinkovitost sistema, zato ga uporablja večina praktičnih implementacij.

Algoritem Paxos [79] je v zadnjih desetih letih postal sinonim na področju algoritmov soglasja in predstavlja izhodišče za večino praktičnih implementacij repliciranih avtomatov. Uporabljajo ga številni Googlovi porazdeljeni sistemi, med drugim tudi storitev za zaklepanje deljenih virov Chubby [80], ki omogoča koordinacijo v porazdeljeni shrambi Bigtable [81]. V nasprotju z večino drugih shranjevalnih sistemov, kjer Paxos služi predvsem hrambi metapodatkov in koordinaciji aktivnosti, shrambi Megastore [82] in Spanner [83] uporabljata Paxos za primarno replikacijo večje količine podatkov. Paxos služi tudi zagotavljanju konsistentnega stanja porazdeljenih krmilnikov SDN v omrežju Google B4 [33]. Glavna težava Paxosa je njegova kompleksnost. Avtor Leslie Lamport je podrobno razlago algoritma [79] sprva podal v obliki fiktivne zgodbe, ki je bila marsikomu nerazumljiva. Pozneje je objavil poenostavljeno različico [84], vendar implementacija celovite rešitve na osnovi Paxosa še vedno predstavlja velik izziv za razvijalce. Podrobno je namreč definiral algoritem, ki omogoča soglasje za posamezno vrednost npr. en zapis repliciranega dnevnika, medtem ko je za kompozicijo množice osnovnih instanc Paxosa v t. i. multi-Paxos podal le smernice. Številne podrobnosti so tako prepuščene razvijalcem. Poleg tega enostavna kompozicija Paxosa v multi-Paxos ni optimalna za uporabo v praktičnih sistemih. Implementacije zato vsebujejo kopico kompleksnih razširitev ter optimizacij in tako precej odstopajo od prvotne definicije algoritma, kar povečuje verjetnost napak. Konkretne rešitve, kot je Chubby, so večinoma lastniške in niso javno dokumentirane. Algoritem Viewstamped Replication [85] temelji na podobnih idejah kot Paxos, čeprav gre za neodvisno delo Briana Okija in Barbare Liskov iz približno istega časa. Zab [86] je nekoliko novejša alternativa, ki predstavlja jedro odprtokodne in zelo razširjene koordinacijske storitve Apache ZooKeeper. Nedavno se je skupini algoritmov soglasja pridružil še algoritem Raft [16], katerega glavno vodilo pri zasnovi je bila razumljivost. Avtorja Diego Ongaro in John Ousterhout sta ga prvič predstavila

leta 2014 v članku z naslovom "In Search of an Understandable Consensus Algorithm" [16]. Doktorska disertacija Diega Ongara [87] vsebuje še nekoliko bolj podrobno razlago in dopolnitev z enostavnejšim mehanizmom za spremembo članstva gruče. Raft obravnava vse potrebne vidike za implementacijo praktičnega sistema, vključno z interakcijo odjemalcev, spremembo članstva gruče in zgoščevanja dnevniških zapisov. Kljub enostavnejši zasnovi je njegova učinkovitost primerljiva z ostalimi algoritmi, zato smo priča vedno večjemu številu odprtokodnih implementacij, ki se uporabljajo tudi v realnih sistemih. Več o samem algoritmu Raft in njegovih implementacijah si bomo pogledali v nadaljevanju.

3.6 Raft

Raft [16] je torej algoritem soglasja za upravljanje repliciranega dnevnika, ki omogoča implementacijo repliciranega avtomata poljubne storitve, za katero želimo zagotoviti visoko razpoložljivost. Problem soglasja rešuje z uporabo asimetričnega pristopa tako, da s pomočjo volitev najprej izbere vodilni strežnik, ki ima popolno odgovornost za upravljanje dnevnika. Vodilni strežnik sprejema zahteve odjemalcev, jih posreduje preostalim strežnikom in strežnike tudi obvesti, kdaj lahko zahtevo izvedejo. V primeru, da vodja odpove ali ne more komunicirati s preostalimi strežniki, sledi nov krog volitev.

Raft tako problem soglasja deli na dva relativno neodvisna podproblema, in sicer volitve za izbiro vodilnega strežnika ter proces normalnega delovanja. Z omejitvami pri glasovanju ter preverjanjem skladnosti in omejitvami pri potrjevanju zapisov ves čas zagotavlja naslednje lastnosti varnosti:

- V vsakem mandatu je izvoljen največ en vodilni strežnik.
- Vodja nikoli ne prepisuje ali briše obstoječih zapisov, temveč jih v dnevnik zgolj dodaja.
- Vsak potrjen zapis bo prisoten tudi v dnevniku vseh novih vodilnih strežnikov.

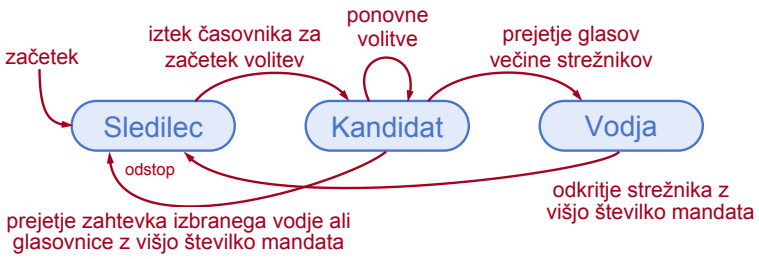
- Če katerikoli strežnik izvede zahtevo, ki je zapisana na določenem indeksu dnevnika, noben drug strežnik ne bo za ta indeks dnevnika izvedel drugačne zahteve. To je tudi ključna lastnost za zagotavljanje konsistentnega stanja.

3.6.1 Vloge in stanje strežnikov

Gručo Raft običajno sestavlja liho število strežnikov, ki imajo lahko med izvajanjem algoritma eno izmed treh vlog, in sicer:

- **Vodja** (angl. *leader*) je zmagovalec volitev, ki je odgovoren za replikacijo dnevnika in interakcijo z odjemalci.
- **Sledilec** (angl. *follower*) je pasiven strežnik, ki ne pošilja nobenih zahtev, temveč zgolj odgovarja na zahteve vodje ali kandidatov.
- **Kandidat** (angl. *candidate*) v procesu volitev kandidira za vodjo.

Tekom normalnega delovanja je vodja natanko en strežnik, medtem ko imajo preostali strežniki vlogo sledilcev. Prehajanje med različnimi vlogami lahko



Slika 3.2: Prehajanje strežnikom med različnimi vlogami [87]

vidimo na sliki 3.2. Vsak strežnik začne v vlogi sledilca in po izteku časovnika prične s postopkom volitev. Če prejme večino glasov preostalih strežnikov v gruči, postane vodja. V nasprotnem primeru postane sledilec, ki ohrani svojo vlogo dokler ne odpove ali prejme zahteve od nove vodje. To se lahko zgodi v primeru razdelitev v omrežju, če večina preostalih strežnikov še vedno lahko komunicira in tako izvoli novo vodjo.

Vsak strežnik v trajnem pomnilniku hrani:

- številko trenutnega mandata (atribut *currentTerm*),
- identifikator kandidata, za katerega je oddal glas na volitvah trenutnega mandata (atribut *votedFor*) in
- seznam dnevniških zapisov (atribut *log[]*), ki vsebujejo ukaz za izvedbo ter številko mandata, v katerem je strežnik ukaz prejel.

Vsak strežnik v začasnem pomnilniku hrani:

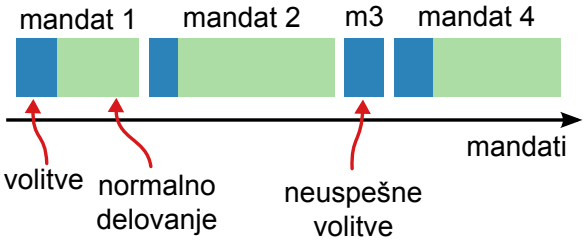
- indeks zadnjega potrjenega zapisa v dnevniku (atribut *commitIndex*) in
- indeks zadnjega zapisa v dnevniku, katerega ukaz je že izvedel na svojem avtomatu (atribut *lastApplied*).

Vodja poleg tega za vsakega sledilca hrani še:

- indeks naslednjega zapisa, ki ga bo poslal dotičnemu sledilcu in je na začetku kandidature za eno večji od indeksa zadnjega zapisa vodje (atribut *nextIndex[]*)
- indeks zadnjega uspešno repliciranega zapisa, ki je na začetku kandidature postavljen na 0 (atribut *matchIndex[]*).

3.6.2 Mandati

Raft deli čas na mandate (angl. *terms*) poljubnih dolžin, kot prikazuje slika 3.3. Vsak mandat se začne s procesom volitev, kjer eden ali več kandidatov poskuša postati vodja. Mandati so predstavljeni v obliki celega števila, ki se monolitno povečuje z vsako kandidaturo. Imajo vlogo logične ure [72] in omogočajo identifikacijo zastarelih podatkov. Številka mandata se prenaša v vseh sporočilih. Če strežnik prejme sporočilo z višjo številko kandidature, mora takoj posodobiti svojo številko mandata. V primeru, da gre za vodjo



Slika 3.3: Delitev na mandate [16]

ali kandidata, ob tem tudi preide v vlogo sledilca. V kolikor strežnik prejme zahtevo z zastarelo številko mandata, jo avtomatsko zavrne in v odgovoru posreduje številko trenutnega mandata.

3.6.3 Sporočila

Strežniki komunicirajo z uporabo oddaljenih klicev procedur (angl. *Remote Procedure Call, RPC*). V ta namen Raft definira zgolj dva tipka klicev oziroma štiri različna sporočila. To sta zahtevek in odgovor za pripenjanje novega ukaza v dnevnik (*AppendEntries RPC*) ter zahtevek in odgovor za glasovanje (*RequestVote RPC*).

AppendEntries RPC

Zahtevke za pripenjanje novega ukaza v dnevnik vedno pošilja trenutni vodja, sledilci pa morajo nanje odgovarjati. Uporabljajo se tudi kot mehanizem za preverjanje živosti vodje. Vsak zahtevek vsebuje:

- številko trenutnega mandata (*term*),
- identifikator vodje (*leaderId*),
- indeks zadnjega zapisa, kateremu sledijo poslani zapisi (*prevLogIndex*),
- številko mandata zadnjega zapisa, kateremu sledijo poslani zapisi (*prevLogTerm*),

- seznam zapisov, ki jih mora sledilec pripeti v svoj dnevnik (*entries[]*) in
- indeks zadnjega potrjenega zapisa v dnevniku vodje (*leaderCommit*).

Sledilec lahko odgovori s pozitivnim ali negativnim odgovorom ter številko njegovega mandata.

RequestVote RPC

Zahtevek za glasovanje pošlje kandidat vsem ostalim strežnikom ob začetku volitev. Zahtevek vsebuje:

- številko mandata kandidata (*term*),
- identifikator kandidata (*candidateId*),
- indeks zadnjega zapisa v kandidatovem dnevniku (*lastLogIndex*) in
- številko mandata zadnjega zapisa v kandidatovem dnevniku (*lastLogTerm*).

Tudi odgovor na zahtevek za glasovanje je lahko pozitiven ali negativen in vključuje številko mandata.

3.6.4 Volitve

Vsak strežnik ima ob zagonu vlogo sledilca, v kateri ostane dokler prejema zahtevke od trenutnega vodje. Vodja vsem sledilcem periodično pošilja zahtevke *AppendEntries* tudi takrat, ko nima novih zapisov za pripenjanje, da ohrani svojo avtoriteto. Če sledilec v naprej določenem časovnem intervalu ne prejme nobenega zahtevka vodje, predvideva, da je prišlo do njegovega sesutja ali izolacije in začne s procesom volitev. To stori tako, da poveča številko mandata, ponastavi števec za iztek časovnika volitev in glasuje sam zase, pri čemer vsem ostalim strežnikom pošlje zahtevek za glasovanje. S tem tudi preide v vlogo kandidata. V kolikor dobi pozitiven odgovor večine

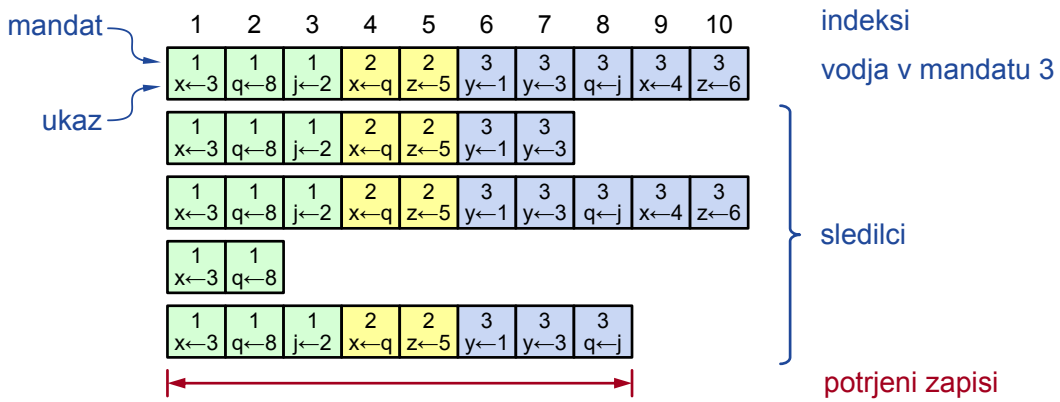
preostalih strežnikov, preide v vlogo vodje in začne s pošiljanjem zahtevkov *AppendEntries*.

Strežniki zahtevo za glasovanje zavrnejo v primeru, da je številka mandata kandidata manjša od njihove številke mandata. Če sta številki mandata enaki, zahtevo zavrnejo v primeru, da je indeks zadnjega zapisa v kandidatovem dnevniku manjši od indeksa zadnjega zapisa v njihovem dnevniku. Na ta način nikoli ne glasujejo za drug strežnik s starejšo različico dnevnika. Na volitvah lahko zmaga le strežnik z najbolj popolnim dnevnikom. Ta omejitev pri glasovanju skupaj z omejitvijo pri potrjevanju zapisov zagotavlja, da je vsak potrjen zapis prisoten tudi v dnevniku nove vodje.

Strežniki zahtevo za glasovanje zavrnejo tudi v primeru, če so že glasovali bodisi za sebe ali nek drug strežnik, od katerega so dobili zahtevo za glasovanje pred iztekom lastnega časovnika za začetek volitev. V kolikor se časovnik za začetek volitev izteče v približno istem času na več kot enem strežniku, se lahko zgodi, da noben od kandidatov ne dobi večine glasov. V tem primeru volitve niso uspešne, zato kandidati po določenem času začnejo z novim krogom volitev. Zaradi uporabe randomizacije pri izbiri vrednosti za iztek časovnika, so takšne situacije dokaj redke in relativno hitro razrešene. Vrednost je ob vsaki ponastavitvi števca izbrana naključno na intervalu [T, T2], pri čemer je T parameter, ki ga lahko prilagajamo in vpliva na razpoložljivost, kot bomo videli v razdelku 3.6.6.

3.6.5 Normalno delovanje

Po uspešnem krogu volitev začne vodja sprejemati zahteve odjemalcev. Vsaka zahteva vsebuje ukaz za izvedbo na repliciranem avtomatu, ki ga vodja skupaj s številko mandata pripne v svoj dnevnik in z zahtevkom *AppendEntries* razpošlje vsem sledilcem. Strukturo dnevnikov lahko vidimo na sliki 3.4. Sledilci najprej preverijo skladnost zahtevka. To pomeni, da indeks in številko mandata zadnjega zapisa v svojem dnevniku primerjajo z indeksom in številko mandata iz zahtevka, ki se nanaša zapis, kateremu sledijo novo poslani zapisi. Če se tako indeks kot številka mandata ujemata, so vsi



Slika 3.4: Struktura dnevnikov [16]

predhodni zapisi dnevnika vodje in sledilca enaki, zato sledilec nove zapise pripne v svoj dnevnik in vodji pošlje pozitiven odgovor. Večina zahtevkov je skladnih, saj vodja v seznamu *nextIndex[]* za vsakega sledilca hrani indeks naslednjega zapisa. Do neskladja lahko pride samo v primeru odpovedi in menjave vodje. Takrat lahko sledilcu določeni zapisi manjkajo ali pa dnevnik vsebuje odvečne nepotrjene zapise, zato je indeks v njegovem dnevniku bodisi manjši od indeksa v zahtevku ali pa se za isti indeks številki mandata ne ujemata. V tem primeru sledilec zahtevek zavrne. Vodja zmanjšuje indeks naslednjega zapisa, ki ga pošlje sledilcu, dokler se indeks in številka mandata v zahtevku ne ujemata z indeksom in številko mandata sledilca. Sledilec takrat zahtevek sprejme in v dnevnik doda vse manjkajoče zapise ter prepiše zapise, ki se ne ujemajo z zapisi vodje. Na ta način je zagotovljena skladnost dnevnikov tudi ob menjavi vodje. V primeru nedosegljivosti sledilca bodisi zaradi njegovega sesutja ali zgolj začasne razdelitve v omrežju, vodja poskuša s ponovnim pošiljanjem zahtevka za pripenjanje zapisa, dokler ne dobi potrditve.

Ko vodja prejme pozitivne odgovore najmanj polovice sledilcev, lahko zapis v dnevniku potrdi s povečanjem indeksa zadnjega potrjenega zapisa, saj je ta varno shranjen na več kot polovici strežnikov. Ukaz potrjenega zapisa nato izvede s pomočjo avtomata in odgovor vrne odjemalcu. Sledilce

o potrditvi obvesti z naslednjim zahtevkom za pripenjanje novega zapisa, ki vedno vsebuje indeks zadnjega potrjenega zapisa v dnevniku. Šele takrat lahko ukaz izvedejo tudi sledilci.

Vse zahteve odjemalcev so enolično označene z identifikatorjem odjemalca in zaporedno številko zahteve, ki ju strežniki ravno tako zabeležijo v dnevniški zapis z namenom preverjanja duplikatov. Odjemalec lahko v primeru odpovedi vodje zahtevo enostavno ponovi s poljubnim strežnikom v gruči. Če ciljni strežnik ni vodja, zahtevo bodisi zavrne in odjemalca preusmeri na novega vodjo ali novemu vodji zahtevo posreduje. Vodja preveri, če je zahteva že pripeta v dnevnik, potrjena in izvedena. Za uspešno izvedene zahteve samo vrne odgovor, potrjene pred tem še izvede, medtem ko nepotrjene zahteve potrdi in izvede šele s potrditvijo ene izmed zahtev v trenutnem mandatu. Če je v prejšnjem mandatu prišlo do sesutja vodje še preden je zahtevo v svoj dnevnik zapisala več kot polovica strežnikov, se lahko zgodi, da dnevnik novega vodje ter posledično tudi vseh sledilcev v novem mandatu te zahteve ne vsebuje. V tem primeru vodja prične z običajnim postopkom njene replikacije.

Ko se odjemalec prvič poveže z enim od strežnikov, se ustvari nova seja, ki predstavlja logično povezavo med odjemalcem in vsemi strežniki gruče Raft. Podatki o seji se hranijo v repliciranem avtomatu, zato lahko odjemalec prehaja med različnimi strežniki, ne da bi izgubil sejo. Da odjemalec ohrani sejo, mora gruči periodično pošiljati sporočila za preverjanje živosti. S tem v repliciranem avtomatu posodablja časovni žig za svojo sejo. Seja poteče, če odjemalcu časovnega žiga seje ne uspe posodobiti v naprej določenem časovnem intervalu.

3.6.6 Čas in razpoložljivost

Rekli smo, da algoritmi soglasja za zagotavljanje konsistentnega stanja niso odvisni od realnega časa, kar velja tudi za algoritem Raft. Varnost algoritma namreč ni kršena niti v primeru napačnega delovanja ur ali nepredvidenih zakasnitev pri procesiranju oziroma dostavi sporočil. To je ključno za pra-

vilno delovanje sistema, saj lahko časovne zamike poleg drsenja ure, okvar ali povečane obremenitve sistema povzročijo tudi dogodki, kot je sproščanje pomnilnika (angl. *garbage collection*) v nekaterih programskih jezikih ali živa selitev navideznega stroja, če se algoritem izvaja v navideznem okolju. Kot smo videli iz rezultata nezmožnosti FLP (razdelek 3.3) in teorema CAP (razdelek 3.4), algoritem v delno sinhronem sistemu ne more garantirati tako varnosti kot živosti, zato Raft žrtvuje razpoložljivost v situacijah, ko se sistem obnaša asinhrono torej izven predvidenih časovnih mej. Če npr. prenos sporočil traja dlje od časovne omejitve za začetek volitev, volitve niso uspešne, brez vodje pa napredovanje algoritma ni mogoče.

Časovne meje so najbolj kritične pri samem procesu volitev. Če je $t_{odgovor}$ povprečen čas za odgovor na oddaljene klice procedur, $t_{volitve}$ časovna omejitev za začetek volitev in t_{MTBF} povprečen čas med zaporednima odpovedma (angl. *Mean Time Between Failure, MTBF*), lahko Raft izvoli in vzdržuje stabilnega vodjo dokler velja:

$$t_{odgovor} << t_{volitve} << t_{MTBF} \tag{3.1}$$

Čas $t_{odgovor}$ mora biti za velikostni razred manjši od $t_{volitve}$ zato, da lahko vodja zanesljivo dostavlja sporočila za preverjanje živosti in s tem ohrani svojo avtoriteto. Ravno tako mora biti $t_{volitve}$ za velikostni razred manjši od t_{MTBF}, da lahko algoritem napreduje. Vrednosti $t_{odgovor}$ in t_{MTBF} sta lastnosti sistema, medtem ko lahko omejitev $t_{volitve}$ prilagajamo. Tipični strežniki imajo t_{MTBF} vsaj nekaj mesecev, zato druge neenakosti ni težko zagotoviti. Čas $t_{odgovor}$ je odvisen od časa, ki je potreben za dostavo zahteve in odgovora preko omrežja ter časa, ki je potreben za procesiranje zahteve. Klici oddaljenih procedur algoritma Raft v večini primerov zahtevajo, da prejemnik vsaj določene informacije shrani v trajni pomnilnik, zato se lahko vrednosti $t_{odgovor}$ tudi med strežniki v lokalnem omrežju gibljejo od 0.5ms do 20ms odvisno od hitrosti trajnega pomnilnika. Pri večjih razdaljah med posameznimi strežniki, npr. pri replikaciji med geografsko porazdeljenimi strežniki, je potrebno upoštevati ustrezno večjo zakasnitev pri prenosu sporočil v obe smeri. Običajne vrednosti za $t_{volitve}$ so tako med 10ms in 1000ms. Manjši kot

je $t_{volitve}$, manjši je lahko čas nedostopnosti ob izpadu vodje, saj se proces volitev začne hitreje. Po drugi strani lahko premajhna vrednost povzroči nepotrebne menjave vodstva in nerazpoložljivost sistema v primeru povečanih zakasnitev pri izmenjavi sporočil ali procesiranju zahtev.

Drug časovni parameter, od katerega je odvisno delovanje algoritma, je interval za periodično pošiljanje sporočil za preverjanje živosti. Ta mora biti strogo manjši od časovne omejitve za začetek volitev in je običajno enak njeni polovici. S pogostejšim pošiljanjem sporočil za preverjanje živosti se lahko izognemo neželenim menjavam vodstva v primeru občasne izgube paketov, vendar zmanjšanje intervala dodatno obremenjuje centralno procesne enote in omrežje. Intervali manjši od $t_{odgovor}$ v večini primerov niso smiselni.

3.6.7 Implementacije

LogCabin je prosto dostopna referenčna implementacija Rafta, ki je avtorjem služila kot testna platforma pri zasnovi samega algoritma. Algoritem in njegovo referenčno implementacijo so razvili za potrebe koordinacije v shranjevalnem sistemu RAMCloud. Danes so na voljo tudi številne druge odprtokodne in lastniške implementacije, ki se uporabljajo v različnih porazdeljenih sistemih.

Facebook že dlje časa testira rešitev Facebook HydraBase [88], razširitev porazdeljene shrambe Apache HBase, ki jo trenutno uporabljajo za shranjevanje tekstovnih sporočil za instantno sporočanje. Uporaba algoritma Raft za replikacijo podatkov znotraj posameznih podatkovnih regij naj bi jim omogočala potencialno povečanje razpoložljivosti sistema iz trenutnih 99,99% na 99,999%.

CoreOS etcd [89] je odprtokodna porazdeljena podatkovna baza tipa ključ-vrednost (angl. *key-value*), ki uporablja lastno implementacijo algoritma Raft v programskem jeziku Go. Med drugim služi kot zaledni sistem za odkrivanje storitev ter shranjevanje stanja in konfiguracije v priljubljenem sistemu za orkestracijo sistemskih vsebnikov Kubernetes [90]. Odkrivanju storitev in upravljanju konfiguracije je namenjena tudi rešitev HashiCorp

Consul [91], ki ravno tako temelji na lastni prosto dostopni implementaciji Rafta.

Ena izmed bolj razširjenih implementacij algoritma Raft je tudi Atomix Copycat [92], ki je na voljo v obliki odprtokodne knjižnice za programski jezik Java in predstavlja jedro koordinacijskega ogrodja Atomix za razvoj porazdeljenih sistemov odpornih na napake. Copycat v osnovi omogoča implementacijo poljubnega repliciranega avtomata z razširitvijo javanskega razreda *StateMachine*, medtem ko samo ogrodje že implementira funkcionalnost za upravljanje članstva v skupinah, skupinsko komunikacijo in različne podatkovne ter koordinacijske primitive. Tako ponuja strogo konsistentne porazdeljene različice nekaterih običajnih podatkovnih struktur, kot so slovar (angl. *map*), množica (angl. *set*) in vrsta (angl. *queue*), kakor tudi primitive za zaklepanje virov in izbiro vodje v skupini. Atomix za vse operacije v osnovi zagotavlja linearizabilno strogo konsistentnost, omogoča pa tudi izbiro zaporedne stroge konsistentnosti za hitrejše branje, ki se lahko izvede na poljubnem strežniku brez izmenjave sporočil z najmanj polovico preostalih strežnikov. Ogrodje podpira tudi transakcije z uporabo protokola dvofaznega potrjevanja (angl. *Two-Phase Commit, 2PC*), pri čemer ponuja različne nivoje izolacije. Poleg aktivnih strežnikov oziroma replik, ki sodelujejo v replikaciji podatkov z algoritmom Raft, lahko v gručo Atomix dodamo še pasivne in rezervne replike. V primeru, da katera od aktivnih replik odpove, jo lahko Atomix samodejno nadomesti s pasivno ali rezervno repliko. Rezervne replike ne sodelujejo pri replikaciji podatkov, medtem ko replikacija podatkov na pasivne replike poteka asinhrono z uporabo protokola Gossip. To omogoča hitrejšo promocijo pasivnih replik v aktivne. Pasivne replike lahko poleg tega sodelujejo tudi pri branjih, kjer je dovoljena zaporedna konsistentnost. Poleg javanskega vmesnika je na voljo tudi samostojen agent, ki je lahko tako v vlogi strežnika kot odjemalca gruče Atomix in omogoča oddaljeno upravljanje preko vmesnika REST. Ogrodje Atomix predstavlja ključno komponento porazdeljenega jedra krmilnika programsko določenih omrežij ONOS in se skupaj s krmilnikom razvija pod okriljem organizacije

ONF.

Poglavje 4

Porazdeljeni krmilniki

V poglavju 2 smo temeljito analizirali koncepte programsko določenih omrežij. Pri tem smo raziskali, kje ima njihova uporaba največji potencial (razdelek 2.5), ter identificirali različne izzive, ki otežujejo širšo uporabo (razdelek 2.6). Kot ključno oviro pri uvedbi programsko določenih omrežij v produkcijska okolja smo izpostavili problem zagotavljanja visoke razpoložljivosti in skalabilnosti omrežja zaradi centralizacije krmilne ravnine ter prišli do spoznanja, da je kljub logični centraliziranosti modela nujno potrebna fizična porazdeljenost krmilnika. To pomeni, da krmilnik sestavlja večje število instanc oziroma fizičnih strežnikov, ki so povezani v gručo in na zunaj delujejo kot ena sama logično centralizirana krmilna ravnina. Porazdeljeni krmilniki, kot sta OpenDaylight in ONOS, v ta namen izkoriščajo uveljavljenje mehanizme za gradnjo porazdeljenih sistemov [93], ki smo jih skupaj z omejitvami pri načrtovanju teh sistemov spoznali v poglavju 3. Najprej si bomo pogledali nekaj ključnih strategij pri zasnovi porazdeljenih krmilnikov in raziskali, katere implementacije odprtokodnih krmilnikov OpenFlow omogočajo njihovo visoko razpoložljivost. Nato bomo analizirali arhitekturo izbranega krmilnika, ki ga bomo v nadaljevanju uporabili za vzpostavitev pilotnega okolja in izvedbo testov.

4.1 Zasnova

Arhitekture porazdeljenih krmilnikov programsko določenih omrežij so prilagojene potrebam konkretnih rešitev. V splošnem jih lahko razdelimo glede na različne strategije pri deljenju podatkov, povezovanju stikal s krmilniki, koordinaciji in tipu krmiljenja [11, 12].

4.1.1 Deljenje podatkov

Podatki, ki so pomembni za delovanje krmilnika, so predvsem podatki o stanju omrežja [94]. Z vidika deljenja podatkov v osnovi ločimo plosko (angl. *flat*) oziroma horizontalno in hierarhično oziroma vertikalno arhitekturo krmilnikov.

V horizontalni arhitekturi imamo samo en nivo enakovrednih instanc krmilnikov, pri čemer ima vsaka instanca globalen pogled na omrežje, ki ga skupaj z ostalimi instancami vzdržuje v porazdeljeni shrambi. V ta namen instance komunicirajo preko t. i. vzhodno-zahodnih vmesnikov (angl. *east-west APIs*). Vsaka instanca posodablja stanje za del omrežja, ki ga direktno nadzoruje. Ključni gradnik predstavljajo algoritmi soglasja, ki zagotavljajo strogo konsistentno stanje sistema tudi ob prisotnosti napak in s tem omogočajo, da porazdeljen krmilnik deluje ekvivalentno krmilniku na enem samem vozlišču tudi v primeru odpovedi posameznih instanc ali razdelitev v omrežju. V primeru odpovedi instance lahko preostali krmilniki prevzamejo nadzor nad njenimi omrežnimi napravami, saj se kopija celotnega stanja nahaja na vseh pravilno delujočih instancah v gruči. Kot smo spoznali v poglavju 3, zagotavljanje konsistentnega stanja zahteva kompromis z vidika zakasnitev sistema. V primeru porazdeljenih krmilnikov to omejuje odzivnost krmilne ravnine, zato se nekatere rešitve vsaj za določen tip podatkov poslužujejo tudi šibkejših modelov konsistentnosti [13].

V hierarhični arhitekturi imamo več nivojev krmilnikov, pri čemer ima globalen pogled na omrežje zgolj korenski krmilnik, medtem ko krmilniki na nižjih nivojih hranijo le stanje za svoj del omrežja. Koordinacija med krmil-

niki je precej poenostavljena, saj vedno poteka preko skupnega krmilnika na višjem nivoju. Posamezen krmilnik lahko vse operacije znotraj dela omrežja, ki ga nadzoruje, izvaja popolnoma neodvisno od ostalih, kar omogoča večjo skalabilnost sistema. Poleg tega je lahko nivo abstrakcije omrežja na korenskem krmilniku bistveno višji, medtem ko se s podrobnostmi posameznega dela omrežja ukvarjajo le pripadajoči lokalni krmilniki. Po drugi strani mora krmilnik za vse operacije, ki zahtevajo globalen pogled na omrežje, komunicirati s krmilnikom na višjem nivoju, ki še vedno predstavlja kritično točko odpovedi. Ravno tako je razpoložljivost posameznega dela omrežja odvisna od razpoložljivosti krmilnika, ki ta del direktno nadzoruje. Za zagotavljanje visoke razpoložljivosti so posamezni krmilniki v hierarhični arhitekturi običajno tudi horizontalno porazdeljeni. Takšen pristop uporablja Google v svojem programsko določenem omrežju B4 [33], kjer ima vsaka fizična lokacija svoj horizontalno porazdeljen lokalni krmilnik, ki zagotavlja visoko razpoložljivost in direktno nadzoruje stikala na določeni lokaciji, medtem ko globalni krmilnik komunicira z množico lokalnih krmilnikov in skrbi za globalno optimizacijo prometa. Tudi globalni krmilnik je pri tem horizontalno porazdeljen in sicer med več podatkovnih centrov na različnih fizičnih lokacijah. Posebnost arhitekture lokalnih krmilnikov je v tem, da osnovno povezljivost med lokacijami še vedno zagotavljajo z uporabo tradicionalnih omrežnih protokolov, medtem ko je globalni krmilnik namenjen zgolj optimizaciji omrežnih poti.

4.1.2 Povezovanje stikal s krmilniki

Če želimo s fizično porazdelitvijo krmilnika zagotoviti visoko razpoložljivost programsko določenega omrežja, moramo poskrbeti, da lahko s podatkovno ravnino vsakega stikala po potrebi upravlja več kot ena fizična instanca krmilnika. To lahko dosežemo s selitvijo naslova IP, preko katerega krmilnik komunicira s stikali. Stikalo je v tem primeru še vedno povezano na eno samo instanco krmilnika, vendar se ob njeni odpovedi ponovno poveže na naslov, ki ga med tem preselimo na drugo delujočo instanco. Bolj fleksibilno alter-

nativo za mehkejšo izvedbo nadomestnega načina delovanja (angl. *failover*) predstavlja možnost hkratnega povezovanja stikala na več kot eno instanco krmilnika, ki so jo v podjetju Nicira sprva implementirali v obliki lastniških razširitev protokola OpenFlow. Z različico protokola 1.2 so te razširitve postale del standarda OpenFlow [60]. Stikalo tako vzpostavi krmilni kanal z vsemi instancami krmilnika v gruči, ki so lahko v vlogi gospodarja (angl. *master*), sužnja (angl. *slave*) ali v enakovredni (angl. *equal*) vlogi. Instance krmilnika, ki so v vlogi gospodarja ali v enakovredni vlogi, imajo poln bralno-pisalni (angl. *read/write*) dostop, kar pomeni, da lahko spreminjajo vsebino tokovnih tabel, hkrati pa prejemajo tudi vsa asinhrona sporočila z omrežnimi dogodki, medtem ko imajo instance v vlogi sužnja zgolj bralni dostop. Koordinacija selitve stikal med instancami je v domeni same implementacije krmilnika in poteka s pomočjo sporočil *ROLE_REQUEST* in *ROLE_REPLY*, s katerimi lahko instanca spremeni svojo vlogo. Krmilnik mora pri tem zagotoviti, da vlogo s polnim bralno-pisalnim dostopom od posameznega stikala zahteva največ ena instanca v gruči, sicer lahko pride do konfliktov pri spreminjanju tokovnih pravil. Stroga konsistentnost preslikave med stikali in pripadajočimi instancami krmilnikov z vlogo gospodarja je tako ključna za pravilno delovanje sistema.

4.1.3 Koordinacija

Z vidika koordinacije ločimo pristope, ki temeljijo na uporabi vodje, in pristope, kjer imajo vse instance enakovredne vloge. V hierarhični arhitekturi ima korenski krmilnik implicitno vlogo vodje, medtem ko so pristopi v horizontalnih arhitekturah lahko različni. Rekli smo, da večina praktičnih algoritmov soglasja koordinacijo zagotavlja z uporabo vodje. V primeru odpovedi vodje to sicer pomeni, da morajo krmilniki najprej izvoliti novo vodjo, kar lahko vpliva čas, ki je potreben za vzpostavitev normalnega delovanja, vendar tak pristop zahteva manj režijskega dela pri sami replikaciji podatkov. Za večjo učinkovitost sistema nekatere implementacije krmilnikov delijo podatke med več neodvisnih instanc repliciranih avtomatov, kar pomeni, da

ima lahko vsaka instanca krmilnika vlogo vodje za določen del podatkov.

4.1.4 Zunajpasovno in znotrajpasovno krmiljenje

Kot smo omenili v poglavju 2, lahko krmiljenje stikal OpenFlow poteka bodisi zunajpasovno ali znotrajpasovno. Znotrajpasovno krmiljenje v primeru porazdeljenih krmilnikov predstavlja poseben problem, saj lahko pride do cikličnih odvisnosti med omrežno povezljivostjo, algoritmom soglasja, ki za napredovanje potrebuje povezljivost s preostalimi vozlišči, ter krmilno logiko omrežja, ki temelji na algoritmu soglasja in zagotavlja omrežno povezljivost [93]. Odpoved povezave lahko povzroči začasno razdelitev v omrežju in privede do nerazpoložljivosti krmilnike ravnine, ki razdelitve v omrežju tako ne more odpraviti s preusmeritvijo prometa po alternativni poti. Avtorji [93] predlagajo razširitev algoritma Raft, ki v primeru odpovedi direktne povezave med dvema krmilnikoma omogoča posredno komunikacijo preko ostalih članov v gruči. V splošnem se težavam najlažje izognemo z uporabo zunajpasovnega krmiljenja, vendar moramo v tem primeru zagotoviti tudi redundanco na nivoju krmilnega omrežja. Pri uporabi znotrajpasovnega krmiljenja si lahko pomagamo z vnaprejšnjo namestitvijo alternativnih poti, ki jih omogočajo skupinske tabele OpenFlow, ali pa osnovno povezljivost zagotovimo neodvisno od centralizirane krmilne ravnine s pomočjo tradicionalnih omrežnih protokolov, kot to počne Google med posameznimi podatkovnimi centri v omrežju B4.

4.2 Implementacije

Onix [47] je bil prvi krmilnik namenjen uporabi v večjih produkcijskih okoljih. Problem skalabilnosti in visoke razpoložljivosti rešuje s horizontalno porazdeljeno arhitekturo in uporabo uveljavljenih mehanizmov za gradnjo porazdeljenih sistemov, vendar gre za lastniško rešitev, ki je kljub drugačnim prvotnim namenom ostala zaprtega tipa. Ravno tako njegov poznejši razvoj za potrebe krmiljenja omrežja Google B4 in komercialne rešitve VMware

NSX ni javno dokumentiran.

V poglavju 2 smo že našteli nekatere predstavnike odprtokodnih krmilnikov SDN. Ugotovili smo, da je večina implementacij fizično centraliziranih in v osnovi ne zagotavlja visoke razpoložljivosti krmilne ravnine. Izjema so krmilniki OpenDaylight [54], ONOS [55], OpenMUL [56] in RuNOS [57]. Več poskusov zagotavljanja visoke razpoložljivosti krmilne ravnine lahko zasledimo v okviru projektov, kot so HyperFlow [95], DISCO [96] in Ravana [97]. Večinoma gre za prototipe rešitev, ki temeljijo na obstoječih centraliziranih krmilnikih in uporabi porazdeljene shrambe podatkov. HyperFlow je aplikacija za krmilnik NOX, ki omogoča sinhronizacijo stanja krmilnikov s pomočjo porazdeljenega datotečnega sistema WheelFS. Tudi Ravana temelji na obstoječem krmilniku Ryu, vendar zagotavlja strogo konsistentno stanje vseh krmilnikov gruči z replikacijo avtomata na osnovi algoritma Viewstamped Replication. Več informacij o aktivnem razvoju katere od omenjenih razširitev nismo zasledili. Ravno tako implementacije teh razširitev niso dostopne v obliki izvorne kode ali izvršljivih datotek, zato smo se osredotočili na trenutno dostopne odprtokodne rešitve.

Ugotovili smo, da krmilnika OpenMUL in RuNOS ne omogočata horizontalnega skaliranja krmilne ravnine. Njuno arhitekturo sestavljata največ dve instanci krmilnika, pri čemer je aktivna samo ena, medtem ko je druga v pripravljenosti, da prevzame nalogo krmilne ravnine v primeru odpovedi aktivne instance. Sočasno delovanje obeh instanc ni predvideno. Hkrati takšna rešitev v primeru odpovedi komunikacijskih kanalov ne garantira, da je aktivna samo ena instanca, zato lahko razdelitve v omrežju povzročijo nekonsistentno stanje, ki vodi v napačno delovanje krmilne ravnine in nerazpoložljivost omrežja. Težava je namreč v zaznavanju odpovedi. Kot smo videli v poglavju 3, za dosego soglasja v delno sinhronem sistemu ob prisotnosti f nebizantinskih napak potrebujemo najmanj $2f+1$ vozlišč. Če bi želeli garantirati varnost sistema, ki ga v tem primeru predstavlja porazdeljena krmilna ravnina, bi odpoved ene od dveh instanc povzročila njegovo nerazpoložljivost. Instanca, ki je v pripravljenosti, zato v primeru nezmožnosti ko-

munikacije z drugo instanco v gruči prevzame aktivno vlogo, čeprav je lahko nezmožnost komunikacije tudi posledica razdelitve v omrežju in ne samo sesutja aktivne instance. V skladu s teoremom CAP gre torej za rešitev tipa AP, ki v primeru razdelitev žrtvuje strogo konsistentnost, da zagotovi razpoložljivost krmilne ravnine, vendar to še ne zagotavlja razpoložljivosti programsko določenega omrežja. Nekonsistentno delovanje obeh instanc lahko privede do tveganih stanj (angl. *race condition*) pri prevzemu nadrejene vloge krmilnika za posamezno omrežno napravo, kar pomeni, da nadrejeno vlogo hkrati poskušata prevzeti obe instanci v gruči, zato nadzor stikala ni mogoč. Verjetnost za pojav razdelitev lahko sicer zmanjšamo z večjo redundanco krmilnega omrežja, vseeno pa se aplikacije na tak mehanizem izvedbe nadomestnega načina delovanja ne morejo zanesti pri spreminjanju stanja, ki zahteva strogo konsistentnost, npr. pri definiciji varnostnih politik.

Poleg omenjenih pomanjkljivosti je tudi obseg funkcionalnosti obeh rešitev znatno manjši od obsega funkcionalnosti krmilnikov OpenDaylight in ONOS, ki veljata za dominantni rešitvi [6, 5] in sta potencialno tudi najbolj primerni za uporabo v produkcijskih okoljih. Njun razvoj podpira zelo velika odprtokodna skupnost v sodelovanju z različnimi proizvajalci omrežne opreme in ponudniki storitev. Oba krmilnika sta napisana v programskem jeziku Java in omogočata visoko modularnost, saj so posamezne storitve implementirane v obliki modularnih gradnikov OSGi, ki jih lahko dinamično odstranjujemo in dodajamo med samim delovanjem. Tako OpenDaylight kot ONOS problem koordinacije rešujeta z uporabo algoritma soglasja Raft, ki zahteva večinsko sklepčnost za izvedbo operacij. Za zagotavljanje razpoložljivosti v primeru izpada ene instance tako potrebujemo gručo najmanj treh strežnikov, vendar s tem lahko zagotovimo pravilnost tudi v primeru razdelitev krmilnega omrežja. Porazdeljeno jedro krmilnika ONOS predstavlja razširitev koordinacijskega ogrodja Atomix, medtem ko OpenDaylight uporablja lastno implementacijo algoritma Raft na temelju ogrodja Akka. Obe rešitvi za večjo učinkovitost omogočata deljenje podatkov med več instanc algoritma Raft. OpenDaylight deli podatke glede na vsebinske sklope, medtem ko so po-

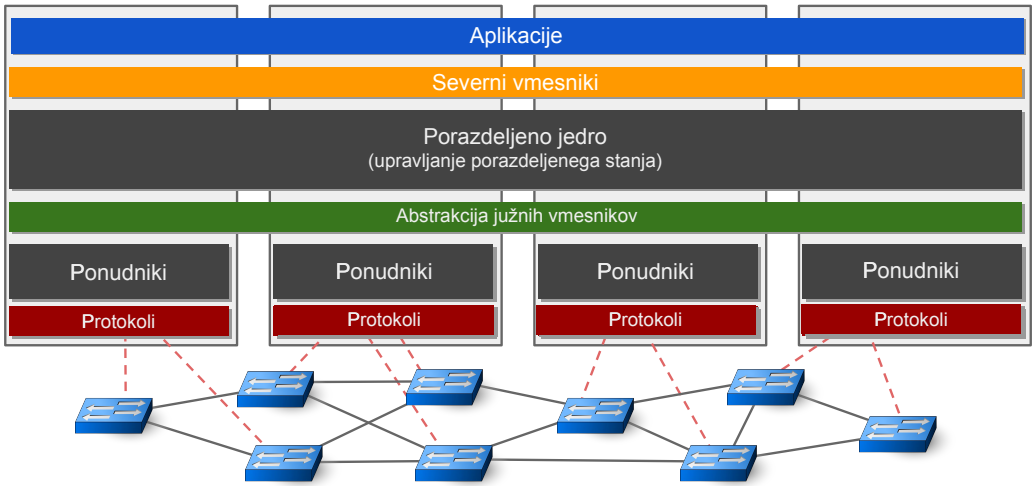
datki pri krmilniku ONOS z namenom večje skalabilnosti razdeljeni glede na ključ posamezne vrednosti. Krmilnik ONOS poleg tega omogoča eventuelno konsistentno shrambo podatkov, pri čemer sta hitrejše branje in pisanje pomembnejša od stroge konsistentnosti. Za vzpostavitev pilotnega okolja in izvedbo testov smo izbrali krmilnik ONOS. V tej fazi z vidika zagotavljanja visoke razpoložljivosti sicer nismo prepoznali večjih razlik v primerjavi s krmilnikom OpenDaylight, smo pa ocenili, da arhitektura krmilnika ONOS omogoča večjo skalabilnost, kar naj bi bil tudi eden izmed ključnih atributov pri njegovi zasnovi. Poleg tega krmilnik ONOS cilja na ponudnike telekomunikacijskih storitev, kjer smo prepoznali velik potencial za uporabo programsko določenih omrežij (poglavje 2). Arhitekturo krmilnika ONOS si bomo podrobneje ogledali v nadaljevanju.

4.3 ONOS

ONOS (angl. *Open Network Operating System*) je torej odprtokoden krmilnik programsko določenih omrežij, katerega glavna cilja pri razvoju sta horizontalna skalabilnost in visoka razpoložljivost, saj je namenjen uporabi v omrežjih ponudnikov storitev, ki običajno predstavljajo kritično infrastrukturo. Kljub temu da gre v primerjavi z ostalimi krmilniki za relativno novo rešitev, prva različica je bila izdana konec leta 2014, je že pritegnil pozornost nekaterih telekomunikacijskih ponudnikov in proizvajalcev omrežne opreme, ki finančno pomagajo in sodelujejo pri samem razvoju v okviru organizacije ONF.

4.3.1 Arhitektura

Visokonivojski pregled arhitekture krmilnika ONOS lahko vidimo na sliki 4.1. Po zgledu krmilnika Onix je ONOS zgrajen okoli porazdeljenega jedra (angl. *distributed core*), ki skrbi za sinhronizacijo stanja in koordinacijo vseh instanc v gruči. Porazdeljeno jedro hrani in preko severnih vmesnikov aplikacijam izpostavlja stanje omrežja na način, ki je neodvisen od protokolov



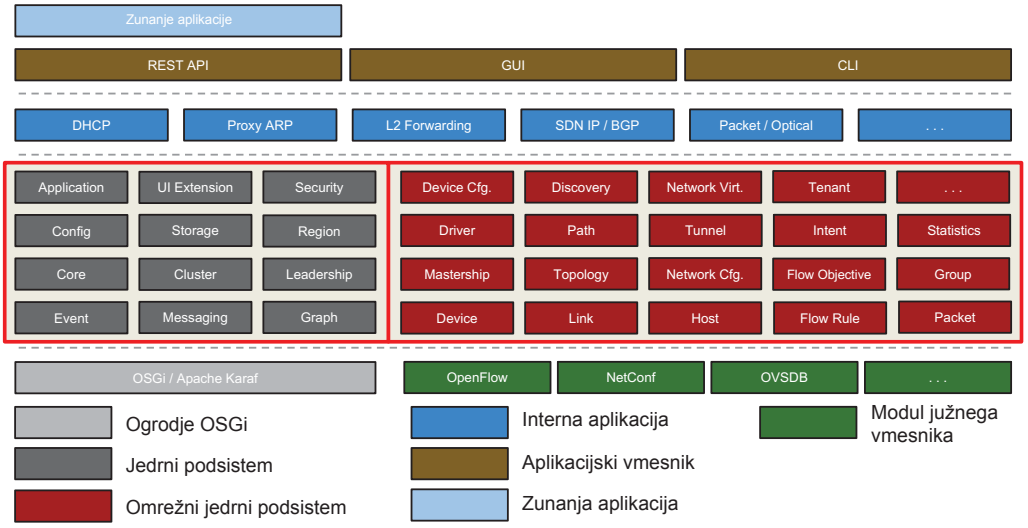
Slika 4.1: Večnivojska arhitektura krmilnika ONOS [98]

(angl. *protocol-agnostic*) na južni strani. Neodvisno od protokolov lahko aplikacije tudi podajo omrežne zahteve bodisi v obliki visokonivojskih tokovnih pravil ali v obliki namere (angl. *intent*) za komunikacijo med dvema točkama v omrežju. Poleg tega jedro aplikacijam nudi različne porazdeljene podatkovne in koordinacijske primitive za hranjenje lastnega stanja. Za interakcijo z omrežnimi napravami skrbijo t. i. ponudniki (angl. *providers*), ki s pomočjo gonilnikov (angl. *drivers*) implementirajo različne protokole južnih vmesnikov. Sprva je bil poudarek ONOSa predvsem na protokolu OpenFlow, medtem ko so pozneje dodali tudi ponudnike za druge protokole, kot so NETCONF, OVSDB, REST, SNMP in TL1. Ponudniki v vsaki instanci krmilnika procesirajo dogodke določene podmnožice omrežnih naprav. Pri tem skrbijo za preslikavo med abstraktnim omrežnim stanjem in zahtevami jedra ter specifičnimi ukazi oziroma sporočili konkretnega južnega vmesnika, kot je OpenFlow.

4.3.2 Storitve

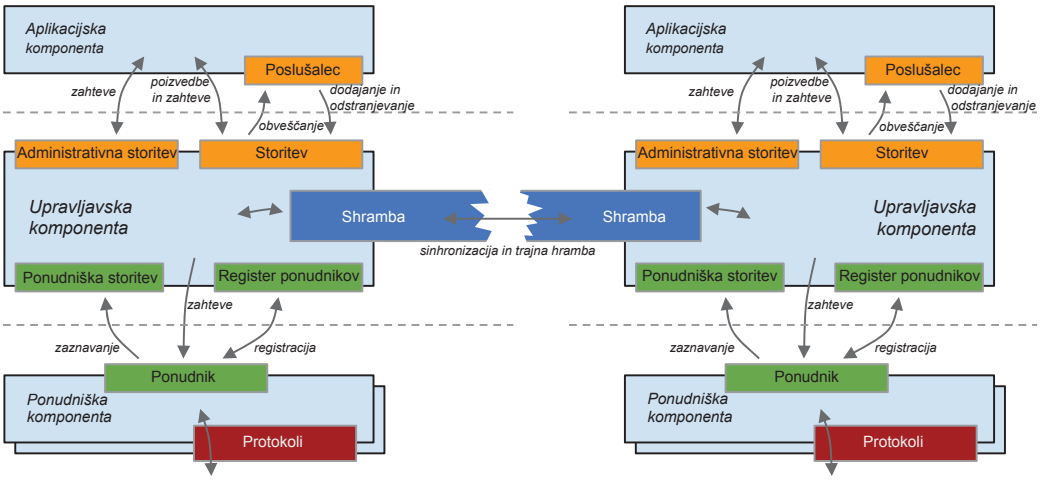
Krmilnik ONOS je sestavljen iz množice storitev oziroma podsistemov, ki predstavljajo posamezno enoto funkcionalnosti. Sem spadajo različni omrežni

podsistemi za upravljanje omrežnih naprav (angl. *device subsystem*), povezav (angl. *link subsystem*), končnih točk (angl. *host subsystem*), topologije (angl. *topology subsystem*), poti (angl. *path service*), tokovnih pravil (angl. *flow rule subsystem*) in paketov (angl. *packet subsystem*), kot tudi jedrni podsistemi za upravljanje gruče (angl. *cluster subsystem*), konfiguracije (angl. *configuration subsystem*), shranjevalni podsistem (angl. *storage subsystem*) in nekateri drugi, kot lahko vidimo na sliki 4.2.



Slika 4.2: Storitve oziroma podsistemi krmilnika ONOS [98]

Vsak podsistem je sestavljen iz ene ali več ključnih komponent, ki ležijo na različnih nivojih krmilnika in so med seboj povezane z enim ali več aplikacijskih vmesnikov, kot prikazuje slika 4.3. V jedru običajno ležita upravljavec (angl. *manager*) in shramba (angl. *store*) posameznega podsistema. Upravljavec prejema informacije od ponudnikov na južni strani, jih shranjuje v pripadajoči shrambi, ki je lahko bodisi strogo ali eventuelno konsistentna, ter jih servira aplikacijam in drugim storitvam preko vmesnika na severni strani. Komunikacija lahko poteka sinhrono s pomočjo poizvedb oziroma zahtev preko storitvenega vmesnika ali asinhrono z implementacijo poslušalca, ki omogoča odzivanje na določene dogodke npr. na pojav nove omrežne na-



Slika 4.3: Struktura podsistemov ONOS [98]

prave ali prispetje paketa za obdelavo v aplikacijski logiki. Vsi podsistemi pri tem ne vključujejo komponent na vseh nivojih. Sistem za upravljanje topologije npr. ne komunicira neposredno s ponudniki na južni strani, temveč za izgradnjo topološkega grafa omrežja uporablja informacije podsistemov za upravljanje naprav in povezav. Podobno storitev za izračun poti operira le nad grafom omrežja topološkega podsistema.

4.3.3 Predstavitev stanja omrežja

Stanje omrežja je v porazdeljenem jedru predstavljeno neodvisno od protokolov s pomočjo različnih konstruktov.

Topologija omrežja

Globalni pogled na omrežje je shranjen s pomočjo naslednjih tipov:

- **Naprava** (angl. *device*): Omrežna naprava, ki ima več vmesnikov in je enolično določena z identifikatorjem *DeviceId*. Naprave predstavljajo notranja vozlišča omrežnega grafa.

- **Vmesnik** (angl. *port*): Omrežni vmesnik naprave, ki skupaj z identifikatorjem naprave tvori točko povezave (angl. *connect point*). Točka povezave predstavlja začetek ali konec povezave v omrežnem grafu.

- **Končna točka** (angl. *host*): Končna točka v omrežju, ki ima določen naslov IP, naslov MAC, identifikator VLAN in točko povezave. Končne točke predstavljajo liste v omrežnem grafu.

- **Povezava** (angl. *link*): Usmerjena povezava med dvema napravama oziroma točkama povezave.

- **Robna povezava** (angl. *edge link*): Poseben primer povezave, ki predstavlja povezavo med napravo in končno točko.

- **Pot** (angl. *path*): Seznam ene ali več povezav, vključno z robnimi povezavami.

- **Topologija** (angl. *topology*): Posnetek prehodnega grafa omrežja, ki je uporaben za izračun poti s poljubnim algoritmom.

Krmiljenje omrežja

Krmiljenje omrežja omogočajo tipi:

- **Tokovno pravilo** (angl. *flow rule*): Visokonivojska predstavitev tokovnega pravila, ki je podana s parom pogoja za ujemanje in pripadajoče akcije oziroma kompozita akcij.

- **Namera** (angl. *intent*): Visokonivojska zahteva za želeno obnašanje omrežja, npr. povezljivost dveh končnih točk v omrežju. Podsistem za mreženje na osnovi namena jih prevede v ustrezna tokovna pravila, ki odražajo želeno obnašanje omrežja.

- **Vloga** (angl. *role*): Predstavlja vlogo posamezne instance krmilnika za določeno omrežno napravo in ima lahko vrednost:

- *NONE*, če omrežna naprava nima vzpostavljenega krmilnega kanala s konkretno instanco krmilnika,
- *MASTER*, če ima instanca krmilnika vlogo gospodarja za to napravo, ali
- *STANDBY*, če ima naprava vzpostavljen krmilni kanal z instanco, vendar le-ta ni njen gospodar.

Omrežni paketi

Za potrebe preusmerjanja prometa aplikacijski logiki sta na voljo tudi tipa:

- **odhodni paket** (angl. *outbound packet*), ki je namenjen predstavitvi paketa za pošiljanje s sporočilom OpenFlow *PACKET_OUT*, in
- **dohodni paket** (angl. *inbound packet*), ki je namenjen predstavitvi paketa prejetega s sporočilom OpenFlow *PACKET_IN*.

4.3.4 Porazdeljeno delovanje

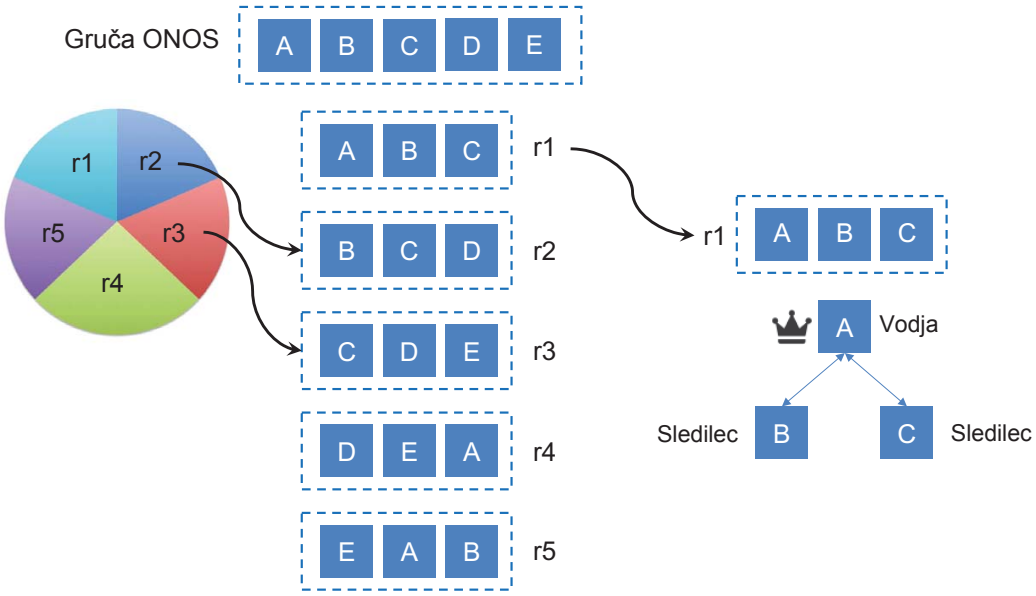
Porazdeljeno jedro krmilnika ONOS od različice 1.4 dalje temelji na koordinacijskem ogrodju Atomix, ki smo ga omenili v okviru implementacij algoritma Raft (razdelek 3.6.7). Številne razširitve avtorji krmilnika postopoma vključujejo tudi v samo ogrodje, ki se sedaj kot ključna komponenta krmilnika ravno tako razvija pod okriljem organizacije ONF. Koordinacija starejših različic je temeljila na uporabi porazdeljene shrambe Hazelcast [99], vendar le-ta v primeru razdelitev ni garantirala pravilnega delovanja, saj gre za rešitev tipa AP.

Krmilnik ONOS je sestavljen iz gruče ene ali več instanc (angl. *instance*) oziroma vozlišč (angl. *node*) krmilnika. Običajno je število vozlišč z namenom zagotavljanja visoke razpoložljivosti liho in večje ali enako 3, saj algoritem Raft, ki je osnova za koordinacijo gruče, za izvedbo operacij zahteva večinsko sklepčnost. Gruča treh instanc krmilnika je tako razpoložljiva, dokler delujeta in med seboj komunicirata vsaj 2 instanci. Vsaka instanca ima

enolični identifikator *NodeID*, ki je privzeto enak naslovu IP, preko katerega instanca komunicira s preostalo gručo. Za potrebe porazdeljenega delovanja jedro nudi tako strogo kot tudi eventuelno konsistentno shrambo.

Strogo konsistentna shramba

Strogo konsistentna shramba temelji na replikaciji podatkov z uporabo algoritma soglasja Raft, pri čemer so vse instance tako v vlogi strežnikov kot tudi odjemalcev Raft. Podatki so za večjo učinkovitost pri branju in pisanju razdeljeni v več razdelkov oziroma med več instanc algoritma, kot prikazuje slika 4.4. Privzeto število razdelkov je enako številu instanc v gruči, kar



Slika 4.4: Delitev podatkov med več instanc algoritma Raft [98]

pomeni, da ima lahko vsaka instanca vlogo vodje za določeno podmnožico podatkov. Implementacija poleg tega omogoča branje z zaporedno strogo konsistentnostjo iz poljubnega vozlišča, običajno je to kar lokalni strežnik Raft, če ta za vodjo zaostaja manj kot za en interval preverjanja živosti. Zahteva za branje se posreduje vodji samo, če je striktno zahtevana linearizablnost, ali če strežnik zaostaja za vodjo več kot za en interval preverjanja

živosti, medtem ko se zahteva za pisanje vedno posreduje vodji in replicira po običajnem postopku.

Strogo konsistentna shramba je različnim storitvam krmilnika na voljo v obliki porazdeljenih podatkovnih struktur, in sicer porazdeljenega slovarja, množice, vrste, atomarnega števca in atomarne vrednosti. Poleg tega je na voljo tudi storitev za upravljanje članstva v skupinah z možnostjo preverjanja živosti in izbire vodje. Izbira vodje je ravno tako izvedena z replikacijo avtomata in jo lahko aplikacije uporabijo za lastne potrebe, npr. če mora aplikacija določeno operacijo izvesti zgolj na eni izmed živih instanc krmilnika. Če vodja določene skupine odpove oziroma ne more komunicirati z gručo, mu po določenem času poteče seja, zato ga gruča samodejno izključi iz skupine in izbere novo vodjo z višjo številko mandata. Vodje skupin in pripadajoče številke mandatov so pri tem neodvisne od vodij in mandatov algoritma Raft.

Storitev za izbiro vodje se uporablja tudi interno v okviru storitve za upravljanje vloge gospodarja (angl. *mastership role service*), ki mora za vsako omrežno napravo med instancami krmilnika, ki so v vlogi pripravljenosti (vrednost *STANDBY*), izbrati vodjo oziroma instanco z vlogo gospodarja (vrednost *MASTER*). Instanca v vlogi gospodarja lahko edina zahteva bralno-pisalni dostop do te omrežne naprave. Instanca lahko sama odstopi od vloge gospodarja, če izgubi povezavo z določeno omrežno napravo, ali če omrežna naprava na sporočilo za spremembo vloge *ROLE_REQUEST* odgovori z napako. V primeru, da instanca ne more komunicirati s preostankom gruče, gruča po poteku njene seje samodejno izvoli novo vodjo oziroma novo instanco z vlogo gospodarja za to napravo. Stroga konsistentnost pri izbiri vodje zagotavlja, da je gospodar določene naprave največ ena instanca. Številka mandata gospodarja, ki se z vsako spremembo gospodarja monotono povečuje, se uporabi tudi v sporočilu za spremembo vloge *ROLE_REQUEST* kot parameter *generation_ID*, na podlagi katerega lahko omrežna naprava zavrne zastarelo zahtevo za spremembo vloge. Na ta način je zagotovljena pravilnost delovanja tudi v primeru, ko zahteva za spremembo vloge zaradi

nepredvidenih zakasnitev do omrežne naprave prispe pozneje kot neka druga novejša zahteva.

Poleg vlog gospodarjev so strogo konsistentno shranjeni tudi podatki o končnih točkah v omrežju, konfiguracija in druge zahteve aplikacij, kot so visokonivojska tokovna pravila, namere in zahteve za pošiljanje odhodnih paketov.

Eventuelno konsistentna shramba

Eventuelno konsistentna shramba krmilnika ONOS žrtvuje strogo konsistentnost v zameno za bistveno hitrejše branje in pisanje, saj temelji na optimističnem pristopu replikacije in ne zagotavlja trajnosti podatkov. Vsa branja so rezultat lokalnega stanja, pri pisanju pa instanca zahtevo sicer razpošlje preostalim instancam v gruči, vendar ne čaka na njihove potrditve. Zaradi uporabe optimističnega pristopa pri replikaciji se lahko določene zahteve izgubijo že med normalnim delovanjem, zato sinhronizacija instanc poteka tudi z uporabo protokola Gossip. Instanca si vsakih nekaj sekund izbere drugo naključno instanco, s katero uskladi stanje na podlagi časovnih žigov. Na ta način lahko tudi nove instance ali instance, ki nekaj časa niso mogle komunicirati s preostalimi člani gruče, hitro uskladijo svoje stanje. Shramba je različnim storitvam na voljo v obliki eventuelno konsistentnega slovarja, ki lahko za razvrščanje dogodkov uporablja časovne žige na osnovi realnega časa ali poljubne logične ure.

Eventuelno konsistentna shramba v porazdeljenem jedru hrani večino informacij, ki tvorijo globalni pogled na omrežje. To so predvsem podatki o omreženih napravah, stanju njihovih vmesnikov in povezavah med njimi. Kljub šibkim zagotovilom same shrambe je globalni pogled relativno konsistenten. Spremembe stanja namreč izhajajo iz samih omrežnih naprav, pri čemer krmilnik garantira, da dogodke določene naprave procesira in v shrambi ustrezno posodablja njihovo stanje samo ena instanca. To je instanca, ki ima za to napravo vlogo gospodarja. Gospodar skrbi tudi za časovno žigosanje dogodkov, ki je neodvisno od realnega časa, saj časovni

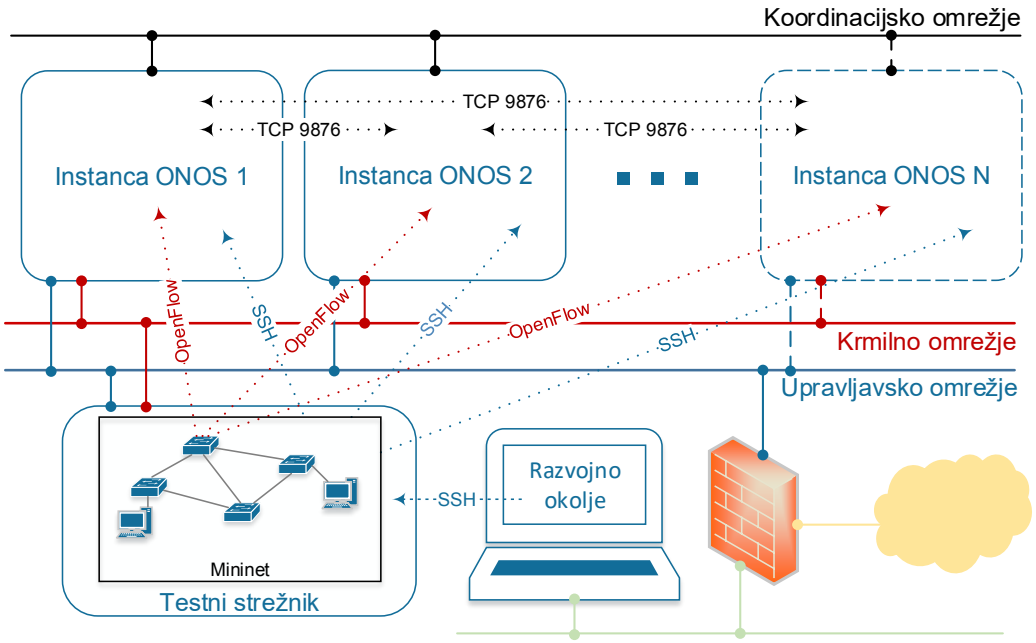
žig sestavljata številka mandata gospodarja in zaporedna številka dogodka znotraj mandata. To omogoča striktno razvrščanje dogodkov posamezne naprave in identifikacijo zastarelih dogodkov. Poleg tega gospodar periodično preverja dejansko stanje naprave in morebitne neskladnosti ustrezno posodablja v shrambi. Do neskladnosti lahko pride v primeru, da gospodar odpove, preden uspe določen dogodek, ki ga je sicer prejel od omrežne naprave, npr. spremembo stanja vmesnika, razposlati preostalim članom gruče.

Poglavje 5

Pilotno okolje

Za potrebe predvidene analize obnašanja programsko določenega omrežja v primeru različnih odpovedi smo najprej vzpostavili pilotno okolje z izbranim krmilnikom ONOS in simulatorjem programsko določenih omrežij Mininet. Ker smo želeli pilotno okolje kar se da približati resnični postavitvi, smo vozlišča oziroma instance krmilnika ONOS ločili od samega simulacijskega okolja, kot prikazuje slika 5.1. Gruča ONOS se tako izvaja na treh ali več namenskih strežnikih, medtem ko se simulacijsko okolje nahaja na ločenem strežniku in služi orkestraciji testov ter emulaciji programsko določenega omrežja z uporabo navideznih stikal Open vSwitch. Prepoznali smo potrebo po ponovljivosti celotnega procesa testiranja, vključno s samo vzpostavitvijo krmilnika, zato smo namestitev instanc krmilnika in njihovo povezovanje v gručo avtomatizirali s pomočjo skript, ki za oddaljeno izvedbo ukazov uporabljajo protokol SSH. Te smo v nadaljevanju integrirali v testno ogrodje, ki predstavlja razširitev simulatorja Mininet. Ogrodje omogoča interakcijo s krmilnikom ONOS in simulacijo različnih odpovedi instanc krmilnika ter povezav med njimi. V prvi fazi smo uporabili navidezne strežnike na hipervizorju VMware ESXi, ki smo jih pozneje z namenom večje konsistentnosti rezultatov nadomestili s fizičnimi strežniki. Na koncu smo rešitev preizkusili tudi v javnem oblaku. Za zajem in analizo tako krmilnega kot uporabniškega prometa smo uporabili orodji *tcpdump* in *Wireshark*, medtem ko smo si pri

odpravljanju težav pomagali tudi z orodjem *ovs-vsctl* za konfiguracijo ter orodjem *ovs-ofctl* za upravljanje in nadzor stikal Open vSwitch.



Slika 5.1: Predvidena arhitektura pilotnega okolja

5.1 Priprava

Za vzpostavitev pilotnega okolja smo izbrali stabilno izdajo krmilnika ONOS, in sicer različico 1.10.3, ki smo jo pozneje nadomestili z različico 1.10.4. Preučili smo dokumentacijo za namestitev in konfiguracijo stabilne izdaje, ki za razliko od razvojnega okolja običajno teče kot sistemska storitev z uporabo ločenega uporabniškega imena in pod nadzorom sistemskega upravljavca storitev, v našem primeru je to *systemd*. V skladu s predvideno arhitekturo pilotnega okolja smo za instance krmilnika pripravili več strežnikov z operacijskim sistemov Linux Ubuntu 16.04 LTS, saj je ta priporočen za namestitev krmilnika. Različica 16.04 LTS je bila ob tem tudi aktualna različica s podaljšano podporo. Poskrbeli smo za osnovno konfiguracijo omrežja in

strežnikov z omrežnimi nastavitvami, imeni in ključi SSH za oddaljen dostop, pri čemer smo onemogočili dostop z gesli. Z orodjem *iptables* smo definirali paketne filtre, ki preprečujejo ves dohodni promet z izjemo oddaljenega dostopa s protokolom SSH iz upravljavskega omrežja, komunikacije med strežniki na vratih TCP 9876 za potrebe koordinacije gruče, dostop iz simulacijskega okolja preko vrat TCP 6653 za povezavo stikal s krmilnikom in dostop do spletnega vmesnika na vratih TCP 8181. Trajnost definiranih pravil smo dosegli z uporabo paketa *iptables-persistent*. Za usklajevanje časa smo namestili programsko opremo *ntpd* in nastavili sinhronizacijo na Arnesova javna strežnika za točen čas.

Namestitev instanc krmilnika smo na pripravljene strežnike najprej opravili ročno, vendar smo v prvi fazi testiranja naleteli na različna napačna stanja krmilnika in težave pri spreminjanju števila strežnikov v gruči, medtem ko smo za izvedbo vseh testov želeli enotno izhodišče. Postopek namestitve smo zato avtomatizirali s pomočjo skripte, ki celoten postopek opravi z oddaljeno izvedbo ukazov preko protokola SSH. Na izbranem strežniku najprej doda sistemskega uporabnika, pod katerim se bo storitev izvajala. V kolikor ni nameščena ustrezna različica jave, doda manjkajoč repozitiorij in namesti javansko izvajalno okolje. Nato z orodjem *scp* na ciljni strežnik prenese namestitvene datoteke in po potrebi pripravi ključe za dostop do ukazne vrstice krmilnika, ki je z različico 1.10.4 privzeto zaščitena z geslom ali ključi SSH. V nadaljevanju sledi čiščenje morebitne obstoječe namestitve, ekstrakcija namestitvenih datotek, njihovo kopiranje v ustrezne imenike, določanje imeniških pravic, nameščanje storitvenih datotek in njihova registracija s sistemskim upravljavcem storitev *systemd* ter dodajanje določenih opcij v nastavitvene datoteke. Skripta med drugim nastavi parametre za maksimalno velikost kopice v javanskem izvajalnem okolju, ki je pomembna za pravilno delovanjem ob večjem številu povezanih omrežnih naprav. Če skripti podamo tudi pot do shrambe ključev, poskrbi za njen prenos na ciljni strežnik in definicijo ustreznih parametrov za uporabo varnega kanala med krmilnikom in omrežnimi napravami s protokolom TLS. Na koncu krmilniško

storitev tudi zažene. Uporabo skripte smo predvideli iz testnega strežnika v sklopu izvajanja testov.

Pripravo omenjenih ključev za vzpostavitev varnega kanala med omrežnimi napravami in krmilnikom smo ravno tako avtomatizirali s skripto, ki z uporabo orodij *ovs-pki*, *keytool* in *openssl* generira certifikatni agenciji, ključe in pripadajoča digitalna potrdila tako za krmilnik kot stikala ter jih pretvori v ustrezne formate. Privatni ključ in digitalno potrdilo stikala ter digitalno potrdilo certifikatne agencije krmilnika v formatu pem tudi namesti na lokalno stikalo Open vSwitch, medtem kot javansko shrambo ključev s privatnim ključem in digitalnim potrdilom krmilnika ter digitalnim potrdilom certifikatne agencije stikala prepusti skripti za namestitev krmilnika. Z avtomatizacijo celotnega procesa namestitve skupaj z nadaljnjimi koraki za vzpostavitev gruče, ki smo jih implementirali v okviru testnega ogrodja, smo omogočili tudi enostavnejšo prenosljivost pilotnega okolja.

Za simulacijsko okolje smo uporabili simulator Mininet [100] različice 2.2.2, ki smo ga namestili na testni strežnik z operacijskim sistemom Linux Ubuntu 14.04 LTS. Z novejšimi različicami operacijskega sistema se namreč pojavljajo težave pri uporabi simulatorja Mininet zaradi hrošča v jedru [101], ki povzroči obešenje sistema pri izvedbi določenih sistemskih klicev za delo z imenskimi prostori. Testni strežnik smo pripravili podobno kot preostale strežnike. Dovolili smo zgolj oddaljen dostop s protokolom SSH z uporabo ključev SSH. Poskrbeli smo tudi, da je s ključem testnega strežnika mogoč dostop do vseh instanc krmilnikov, kar predstavlja osnovni predpogoj tako za namestitev kot tudi samo interakcijo s krmilnikom v okviru naše rešitve.

5.2 Simulacijsko okolje

5.2.1 Mininet

Mininet je omrežni simulator, ki z uporabo lahke virtualizacije na nivoju operacijskega sistema Linux omogoča enostavno prototipiranje programsko določenih omrežij preko ukazne vrstice ali aplikacijskega programskega vme-

snika v jeziku Python [102, 100]. S programsko kodo lahko definiramo množico gostiteljev, stikal in krmilnikov ter topologijo, ki določa povezave med njimi. S posameznimi elementi lahko nato tudi upravljamo. Vsak gostitelj je realiziran s procesom ukazne lupine v lastnem omrežnem imenskem prostoru (angl. *network namespace*), ki omogoča ločevanje omrežnega konteksta na nivoju jedra operacijskega sistema. Omrežni imenski prostori imajo izolirane omrežne vmesnike, sklade protokola IP, usmerjevalne tabele in imenik */proc/net*. Standardni vhod in izhod procesa sta z uporabo cevi (angl. *pipe*) preusmerjena v starševski proces mininet, preko katerega lahko iz programske kode izvršujemo različne ukaze z metodo *cmd()* ali preko vmesnika *popen()*. Na ta način lahko v gostiteljih poganjamo poljubno programsko opremo za operacijski sistem Linux, npr. orodje *ping* ali strežnik HTTP. Različna navidezna stikala, ki nadomeščajo fizične omrežne naprave, običajno tečejo v privzetem imenskem prostoru in so z drugimi stikali ter gostitelji povezana z uporabo navideznega etherneta (angl. *virtual-ethernet, veth*).

5.2.2 Navidezna stikala

Za naše potrebe smo izbrali navidezno stikalo Open vSwitch. Kot smo omenili v razdelku 2.3.1, se stikalo Open vSwitch za razliko od nekaterih eksperimentalnih implementacij uporablja tudi v komercialnih rešitvah, zato smo ocenili, da bo najbolj primerno tudi za naše pilotno okolje. Preučili smo možnosti za povezovanje stikala na več instanc krmilnika iz simulacijskega okolja. Okolje Mininet namreč definira razred *OVSSwitch*, ki omogoča upravljanje stikala Open vSwitch. Ravno tako so na voljo razredi za interakcijo z različnimi enostavnimi krmilniki, kot je Pox, in razred *RemoteController*, ki omogoča povezovanje na poljuben zunanji krmilnik, vendar v osnovi ne ponuja možnosti za povezovanje s porazdeljenimi krmilniki. Ugotovili smo, da stikalo Open vSwitch to sicer omogoča in stikalo tudi uspešno povezali na vse instance krmilnika ONOS z orodjem *ovs-vsctl*. Rešitev smo najprej preizkusili z uporabo protokola OpenFlow 1.0, ki ga podpirata tako stikalo kot krmilnik in z lastniškimi razširitvami omogoča tudi povezovanje na več krmilnikov. Pri

tem smo naleteli na različne težave pri prevzemanju vloge gospodarja, če smo prekinili zgolj krmilni kanal med stikalom in eno instanco krmilnika. Želeli smo izključiti možnost težav zaradi različice protokola, saj je povezovanje z več krmilniki standardizirano šele z različico 1.2 (razdelek 4.1.2). Ocenili smo, da bi bilo najbolje uporabiti OpenFlow 1.3, ki je aktualna različica s podaljšano podporo. Tudi knjižnica logixen [103], ki je osnova za implementacijo protokola na krmilniški strani, za produkcijsko uporabo priporoča različico 1.3, medtem naj bi bila različica 1.2 namenjena zgolj eksperimentalni uporabi. Pri tem smo ugotovili, da stikalo Open vSwitch 2.0.2, ki je nameščeno skupaj s simulatorjem Mininet, nima popolne podpore za OpenFlow 1.2 in 1.3. Tudi praktičen preizkus je pri uporabi omenjenih različic pokazal določene napake v dnevniških zapisih krmilnika. Odločili smo se za nadgradnjo stikala Open vSwitch, vendar se je izkazalo, da namestitveni paketi za novejše izdaje stikala za operacijski sistem Ubuntu 14.04 LTS niso na voljo, medtem ko operacijskega sistema nismo mogli nadgraditi zaradi že omenjenega hrošča v novejšem jedru. Na koncu smo uspeli iz izvorne kode prevesti in namestiti Open vSwitch različice 2.7.2, ki je bila v času postavitve tudi aktualna stabilna različica in v celoti podpira protokol OpenFlow 1.3.

V sklopu razvoja testnega ogrodja smo v nadaljevanju implementirali ustrezne razširitve razredov krmilnika in stikala za delo s porazdeljenim krmilnikom ONOS.

5.3 Testno ogrodje

Za izvedbo predvidenih testov in analizo obnašanja smo razvili ogrodje, ki predstavlja razširitev simulacijskega okolja Mininet in omogoča interakcijo s porazdeljenim krmilnikom ONOS ter simulacijo različnih odpovedi. Pri tem smo uporabili razvojno okolje PyCharm, v katerem smo nastavili oddaljen interpreter Python, ki se z uporabo protokola SSH izvaja na testnem strežniku v pilotnem okolju. Za hitrejšo izvedbo smo poskrbeli tudi za avtomatsko nameščanje izvorne kode iz razvojnega okolja na testni strežnik s pomočjo

protokola SFTP.

5.3.1 Krmilnik

Za interakcijo in povezovanje s porazdeljenim krmilnikom ONOS smo implementirali razred *ONOSController*, ki razširja generičen razred *Controller* okolja Mininet in med drugim hrani seznam vseh vozlišč krmilnika, ki so predstavljena razredom *ONOSNode*. Uporaba vgrajenega razreda *RemoteController* poleg tega, da predstavlja eno samo instanco krmilnika, namreč služi zgolj kot referenca za povezavo stikal na krmilnik s protokolom OpenFlow, medtem ko ne omogoča interakcije s samim krmilnikom.

Konstruktor razreda *ONOSController* sprejme seznam domenskih imen oziroma naslovov IP strežnikov, na katerih je nameščen oziroma na katere želimo namestiti krmilnik ONOS, različico krmilnika in nekatere druge parametre, kot sta velikost kopice in zahteva po uporabi varnega kanala med omrežnimi napravami in krmilnikom. Od konstruktorja lahko z zastavico *deploy* zahtevamo svežo namestitev krmilnika. V tem primeru se ob inicializaciji s pomočjo pripravljene skripte izvede vzporedna namestitev krmilnika na vse podane strežnike preko protokola SSH. Po potrebi se pred namestitvijo iz uradnega repozitorija prenesejo tudi namestitvene datoteke krmilnika in pripravijo ključi za vzpostavitev varnega kanala med omrežnimi napravami ter krmilnikom. Okolje nato počaka, da so vse instance krmilnika pripravljene. To je takrat, ko se ukazne vrstice vseh instanc začnejo odzivati in v seznamu vozlišč vrnejo lastno stanje, ki je enako *READY*. Sledi povezovanje vseh instanc v gručo z ukazom *onos-form-cluster*. Vzpostavitev krmilnika se zaključi, ko je gruča delujoča, kar pomeni, da je na vseh instancah mogoča izvedba ukazov. Preverjanje poteka z uporabo ukaza *summary* preko ukazne vrstice krmilnika, ki vrne povzetek stanja gruče le v primeru, da lahko instanca komunicira z večino članov v gruči, saj zahteva branje iz strogo konsistentne porazdeljene shrambe.

Za nadaljnjo interakcijo s krmilnikom smo implementirali različne metode, ki na oddaljenih strežnikih omogočajo izvedbo sistemskih ukazov, iz-

vedbo ukazov preko ukazne vrstice krmilnika, zagon in zaustavitev storitve, preverjanje stanja posameznih instanc ali celotne gruče, kakor tudi enostavno izvedbo določenih pogostih operacij, kot je aktivacija aplikacij ali izpis vseh tokovnih pravil krmilnika. Izvedba vseh operacij temelji na oddaljeni izvedbi ukazov s protokolom SSH. Klic metode *activateForwarding()* na instanci razreda *ONOSNode* npr. na dotičnem vozlišču, ki ga instanca razreda predstavlja, izvede ukaz za aktivacijo aplikacije, ki omogoča posredovanje prometa na drugi plasti. Če istoimensko metodo pokličemo na instanci razreda *ONOSController*, ki predstavlja celotno gručo, se ukaz izvede na prvi delujoči instanci krmilnika oziroma vrne napako v primeru, da nobena instanca ukaza ne more izvršiti.

5.3.2 Stikalo

Z razredom *MultiOVSSwitch* smo ustrezno razširili tudi razred *OVSSwitch*, tako da za instanco razreda *ONOSController* stikalu posreduje naslove vseh vozlišč porazdeljenega krmilnika in privzeto uporablja protokol OpenFlow 1.3. Poleg tega v fazi inicializacije preveri, če je nameščena različica stikala Open vSwitch 2.7.2 ali novejša.

5.3.3 Topologije

Okolje Mininet že vključuje različne enostavne omrežne topologije. Za izvedbo testov smo z razredom *LeafAndSpineTopo* dodatno implementirali topologijo listov in hrbtenic, ki je sestavljena iz dveh nivojev omrežnih naprav, in sicer nivoja hrbtenic ter nivoja listov. Gostitelji so povezani na posamezne liste, vsak list pa je povezan na vse hrbtenice. Topologija listov in hrbtenic zaradi vedno večje količine prometa med strežniki (angl. *east-west traffic*) v podatkovnih centrih počasi nadomešča tradicionalno tronivojsko arhitekturo, saj zagotavlja enako pasovno širino in razdaljo med gostitelji na poljubnih dveh listih. Poleg tega omogoča enostavno razširljivost z dodajanjem listov in povečanje pasovne širine z dodajanjem hrbtenic. Topologijo smo parame-

trizirali s parametri s, l in h. Parameter s določa število hrbtenic, parameter l število listov in parameter h število gostiteljev na vsakem listu. Za lažje razhroščevanje smo zaporedne številke listov, hrbtenic in gostiteljev uporabili v njihovih identifikatorjih podatkovne poti oziroma naslovih MAC in IP.

5.3.4 Simulacija odpovedi vozlišč

Simulacijo odpovedi vozlišč smo implementirali z metodami, ki na ciljnem vozlišču izvedejo zaustavitev procesa, njegovo sesutje ali ponovni zagon strežnika. Metoda za simulacijo sesutja procesa z orodjem *systemctl* najprej pridobi identifikator procesa (angl. *process identifier, PID*) pod katerim teče storitev krmilnika in z orodjem *kill* procesu pošlje signal *SIGKILL*. Ker zna upravljavec storitev proces tudi ponovno zagnati, smo dodali možnost, ki pred sesutjem procesa prepreči njegov ponovni zagon z odstranitvijo storitvenih datotek in njihovo ponovno namestitvijo po uspešnem sesutju procesa. S pomožnimi metodami smo omogočili hkratno zaustavitev, sesutje ali ponovni zagon manjšine, večine ali celotne gruče.

5.3.5 Simulacija odpovedi komunikacijskih kanalov

Za simulacijo odpovedi komunikacijskih kanalov smo preizkusili več možnosti. Prvi prototip je temeljil na spreminjanju stanja omrežnih vmesnikov, vendar na ta način ni mogoče preprečiti komunikacije zgolj z določenimi vozlišči. Ravno tako nismo želeli uporabiti rešitve, ki bi bila odvisna od spodaj ležeče fizične ali navidezne infrastrukture. Poleg tega smo tekom razvoja ogrodje za večjo fleksibilnost zlasti pri prenosu rešitve na oblačno infrastrukturo in fizične strežnike prilagodili tako, da lahko pilotno okolje uporablja enotno omrežje tako za koordinacijo gruče, krmiljenje stikal kot tudi upravljanje strežnikov preko protokola SSH. Simulacijo razdelitev smo zato implementirali z uporabo paketnih filtrov, ki jih simulacijsko okolje namesti na posamezna vozlišča z orodjem *iptables*.

Rešitev smo zasnovali tako, da okolje ob zagonu krmilnika na vsa vozlišča

z orodjem *iptables* doda uporabniško določeni verigi *ONOS-IN* in ONOS-OUT, v kateri s pravilom z najvišjo prioriteto iz privzetih verig *INPUT* ter *OUTPUT* v tabeli *filter* preusmeri ves dohodni oziroma odhodni promet z izjemo prometa za vrata TCP 22, ki omogoča upravljanje s tem vozliščem preko protokola SSH. Obe verigi na koncu vsebujeta pravilo z akcijo *RETURN* za vrnitev v verigo, iz katere je bil skok izveden. Ves promet, ki ga za potrebe simulacije odpovedi komunikacijskih kanalov eksplicitno ne preprečimo v verigah *ONOS-IN* in ONOS-OUT, se tako filtrira v skladu s paketnimi filtri, ki so sicer definirani na tem strežniku. Metode, ki omogočajo simulacijo razdelitev omrežja med vozlišči krmilnika kot tudi med vozlišči in stikali oziroma simulacijskim okoljem, smo izvedli z dodajanjem pravil za ustrezne naslove IP z akcijo *DROP* v verigi *ONOS-IN* in *ONOS-OUT*, medtem ko z metodo *networkPartitionsClearAll()*, ki odpravi vse razdelitve, odstranimo vsa pravila v omenjenih verigah. Ob zaključku testov verigi enostavno izpraznimo in odstranimo.

5.3.6 Meritev zmogljivosti

V skladu z zadanimi cilji smo želeli primerjati tudi zmogljivost krmilne ravnine med izvedbo nadomestnega načina delovanja in normalnim delovanjem. Odločili smo se, da bomo merili količino sporočil *PACKET_IN*, ki jih lahko krmilna ravnina obdela na časovno enoto. Ta metrika je pogosto uporabljena za primerjavo zmogljivosti različnih krmilnikov [6]. Ugotovili smo, da z uporabo normalnega prometa, ki ga lahko ustvarimo v simulacijskem okolju, to ni mogoče. Tovrstno meritev nam med drugim omogoča orodje *cbench*, ki simulira množico stikal OpenFlow zgolj s pošiljanjem paketov *PACKET_IN* in štetjem odgovorov krmilnika, vendar ne omogoča povezovanja s porazdeljenimi krmilniki. Enako omejitev ima tudi večina drugih orodij vključno s komercialno rešitvijo PktBlaster. Odločili smo se za integracijo orodja *cbench* v naše ogrodje na način, da na vsako instanco krmilnika povežemo ločen oziroma več ločenih procesov *cbench*. To smo dosegli z implementacijo lastnega razreda stikala, ki v resnici ne predstavlja dejanskega navideznega stikala

temveč posamezen proces orodja *cbench*, razreda topologije, ki poskrbi za zagon in povezavo ustreznega števila procesov, ter logike za obdelavo rezultatov. Vsak proces vmesne rezultate na določen interval npr. vsakih 100 ms beleži v datoteko s preusmeritvijo standardnega izhoda in uporabo orodja *stdbuf* za prilagoditev medpomnenja. Datoteke z rezultati na koncu ustrezno obdelamo, da dobimo povprečne vrednosti za celotno gručo. Pri tem vmesne rezultate vsakega procesa najprej normaliziramo, jih na podlagi časovnih žigov razvrstimo v časovne koše in izračunamo njihove povprečne vrednosti. Vrednosti košev različnih procesov za isti časovni interval nato seštejemo.

5.3.7 Beleženje

V okviru naše rešitve smo omogočili beleženje celotne izvedbe testov za potrebe poznejše analize. V ta namen za vsak test ob inicializaciji okolja ustvarimo imenik, ki je označen s časovnim žigom in hrani vse rezultate ter pomožne datoteke posameznega testa. V omenjen imenik med izvedbo testa z orodjem *tcpdump* zajemamo tudi ves promet OpenFlow med krmilnikom in stikali v simulacijskem okolju. Ob končanju testa se v imenik prenesejo tudi dnevniški zapisi vseh instanc krmilnikov.

Poglavje 6

Analiza obnašanja

V pričujočem poglavju bomo analizirali obnašanje programsko programsko določenega omrežja v primeru različnih odpovedi in ovrednotili izbrano implementacijo krmilnika predvsem z vidika zagotavljanja visoke razpoložljivosti, ki je ključnega pomena za produkcijsko uporabo. Najprej bomo definirali testne scenarije, nato bomo opisali izvedbo testov in predstavili ugotovitve ter predloge za izboljšave na podlagi rezultatov teh testov.

6.1 Scenariji za izvedbo testov

Obnašanje programsko določenega omrežja je v primeru težav s krmilno ravnino precej odvisno od zasnove same aplikacije, kot bomo v nadaljevanju tudi podrobneje razdelali. Pri izvedbi testov smo se osredotočili na primer, pri katerem je razpoložljivost krmilne ravnine najbolj kritična. To je reaktivni način delovanja, kjer stikalo neznan promet posreduje krmilniku s sporočilom *PACKET_IN*. Krmilnik nato odgovori z namestitvijo ustreznega tokovnega pravila in paket skupaj z navodili za procesiranje vrne s sporočilom *PACKET_OUT*. V primeru nedosegljivosti krmilne ravnine posredovanje prometa, ki ne pripada nobenemu od obstoječih podatkovnih tokov, tako ni več mogoče, medtem ko je posredovanje prometa obstoječih podatkovnih tokov mogoče le do poteka časovne omejitve pripadajočih tokovnih pravil. V ta

namen smo uporabili aplikacijo za posredovanje prometa na drugi plasti, ki je del krmilnika ONOS.

Vsak scenarij za izvedbo testa je v grobem sestavljen iz

- vzpostavitve gruče krmilnika ONOS,
- aktivacije podsistema OpenFlow,
- aktivacije aplikacije za posredovanje prometa na drugi plasti,
- čakanja na uspešno povezavo z vsemi stikali,
- prerazporeditve vlog gospodarjev,
- preizkusa zmožnosti komunikacije in izpisa stanja krmilnika,
- simulacije odpovedi ter
- ponovnega preizkusa zmožnosti komunikacije in izpisa stanja krmilnika.

Pri tem smo želeli simulirali različne odpovedi vozlišč ter komunikacijskih kanalov, in sicer:

- sesutje procesa na manjšini oziroma večini vozlišč,
- sesutje procesa na manjšini oziroma večini vozlišč, pri čemer smo preprečili ponovni zagon procesa,
- zaustavitev procesa na manjšini oziroma večini vozlišč,
- razdelitev v omrežju, ki manjšini oziroma večini vozlišč v celoti onemogoča komunikacijo tako s stikali kot preostalimi vozišči,
- razdelitev v omrežju, ki manjšini oziroma večini vozlišč onemogoča komunikacijo s preostalimi vozlišči, in
- razdelitev v omrežju, ki enemu ali več vozliščem onemogoča komunikacijo s stikali.

Razlikovanje med nedelovanjem oziroma nezmožnostjo komunikacije večine in manjšine vozlišč v gruči je pomembno predvsem z vidika algoritma soglasja Raft, ki je ključen element za koordinacijo gruče in za izvedbo operacij zahteva večinsko sklepčnost.

Za omenjene scenarije nas je v prvi vrsti zanimala razpoložljivost programsko določenega omrežja, ki je odvisna tako od razpoložljivosti krmilne ravnine kot tudi njenega pravilnega delovanja. Zmožnost komunikacije smo v simulacijskem okolju preverjali z metodo *pingall*, ki preverja dosegljivost vseh kombinacij gostiteljev v omrežju z orodjem *ping*, stanje krmilnika pa s pomočjo ukazov za izpis tokovnih pravil, priključenih naprav, vlog gospodarjev in stanja vozlišč preko ukazne vrstice krmilnika. Alternativno smo želeli preveriti tudi dostopnost vmesnika REST in možnost izvedbe ukazov preko ukazne vrstice posameznih instanc krmilnika. V nadaljevanju nas je v primerih, kjer je bila izvedba nadomestnega načina delovanja uspešna, zanimal tudi čas, ki je potreben za njeno izvedb, ter vpliv odpovedi na zmogljivost celotne gruče. Pri analizi obnašanja smo si pomagali še z vsebino dnevniških datotek, ki se ob koncu vsakega testa prenesejo v imenik z rezultati testov, pri čemer smo z ukazom *log:set TRACE io.atomix* na krmilniku omogočili beleženje vseh dogodkov algoritma Raft v porazdeljenem jedru.

Predvideli smo izvedbo testov na topologiji listov in hrbtenic z različnim številom omrežnih naprav in gostiteljev ter tremi, petimi oziroma sedmimi instancami krmilnika v gruči.

6.2 Izvedba in rezultati

Izvedbo testov smo začeli s topologijo s tremi hrbtenicami, štirimi listi in po dvema gostiteljema na vsakem listu ter tremi instancami krmilnika v gruči. Kljub temu, da smo v okviru testnega ogrodja postopek vzpostavitve krmilnika v celoti avtomatizirali, smo med izvedbo testov večkrat naleteli na težave že pri samem zagonu gruče. V določenih primerih je po izvedbi ukaza za povezovanje instanc krmilnika v gručo ta ostala nedosegljiva. Iz dnevniških

datotek smo ugotovili, da je prišlo do različnih napačnih stanj. Ena izmed težav je bila uporaba napačnih naslovov IP, preko katerih so želele nekatere instance komunicirati s preostalimi člani gruče. Vsaka instanca krmilnika ima namreč tako vlogo strežnika kot odjemalca gruče Raft, pri čemer odjemalski del samodejno pridobi naslove strežnikov. V teh primerih je šlo za naslove napačnih mrežnih vmesnikov, ki niso bili namenjeni koordinaciji gruče. Težavi smo se izognili tako, da smo uporabili enotno omrežje za koordinacijo gruče, krmiljenje stikal kot tudi upravljanje strežnikov. To sicer nekoliko odstopa od predvidene arhitekture našega pilotnega okolja, vendar drugače nismo uspeli zagotoviti, da so vse instance vedno uporabile naslove pravih mrežnih vmesnikov. Še vedno so se takoj po vzpostavitvi gruče občasno pojavljale nekatere druge napake. Med drugim smo naleteli na izjeme zaradi manjkajočih posnetkov stanja, ki jih algoritem Raft periodično izvaja za potrebe zgoščevanja dnevniških zapisov, izjeme zaradi manjkajočih upravljavcev sporočil, kot tudi izjeme zaradi uporabe nedefiniranih objektov *null* v določenih komponentah. Nobene od omenjenih napak nismo uspeli deterministično reproducirati, vendar so se pojavile na določeno število poskusov, pri čemer je ponovni zagon celotne gruče težavo odpravil. Poleg tega smo ugotovili, da so bile napake manj pogoste, ko smo pilotno okolje preselili iz navideznih na fizične strežnike, kjer smo izvedli tudi večino testov z meritvami časa in zmogljivosti. Predvidevamo, da so napake posledica hroščev v implementaciji krmilnika in se pojavijo v primerih, ko se določene komponente krmilnika ne naložijo pravočasno, zato je tudi obnašanje na fizični infrastrukturi, kjer viri niso deljeni, nekoliko bolj predvidljivo. Za izvedbo testov smo prilagodili postopek vzpostavitve krmilnika tako, da simulacijsko okolje v fazi čakanja na pripravljenost gruče po določenem času, ki je bistveno daljši od predvidenega časa za vzpostavitev, poskrbi za njen ponovni zagon in se s tem izogne napačnemu stanju krmilnika, še vedno pa so omenjene težave problematične za uporabo v produkcijskem okolju.

Na določene anomalije, za katere menimo, da niso posledica zasnove krmilnika temveč hroščev pri implementaciji, smo naleteli tudi v fazi samega

delovanja krmilnika oziroma pri simulaciji odpovedi. Konkretno smo pri simulaciji razdelitve omrežja, ki zgolj eni izmed treh instanc krmilnikov preprečuje komunikacijo s preostalimi člani gruče, oziroma pri odpravi te razdelitve v nekaterih ponovitvah ugotovili, da komunikacija med gostitelji vseeno ni bila več mogoča ali pa je prišlo do podvojitve paketov. Glede na stanje gruče, je bila ob pojavu razdelitve izvedba nadomestnega načina delovanja uspešna, saj je vlogo gospodarja za naprave, katere je krmilila instanca v manjšinski skupini, uspešno prevzela druga instanca krmilnika. Od odpravi razdelitve, se je instanca ponovno pridružila gruči, vendar delovanje omrežja ni bilo več pravilno. Iz izpisa priključenih naprav, vmesnikov, poti med gostitelji in tokovnih pravil na obeh preostalih instancah krmilnika nismo opazili nekonsistentnosti. Ravno tako, je bilo stanje tokovnih pravil konsistentno z dejanskim stanjem nameščenih tokovnih pravil na sama stikala, kar smo preverili z orodjem *ovs-ofctl*. Podroben pregled tokovnih pravil je pokazal, da nekatera tokovna pravila vsebujejo napačen izhodni vmesnik, kljub temu, da je bil izpis poti, ki so osnova za izračun tokovnih pravil, pravilen. Stikala so tako promet gostiteljev posredovala na napačne vmesnike. V nekaterih primerih je to pomenilo izgubo prometa, v primeru, ko je paket prispel na drugo stikalo po nepredvideni poti, pa je krmilnik uporabil celo akcijo za poplavljanje paketa na vse vmesnike, kar je vodilo v podvajanje paketov. Napačnega stanja nismo uspeli odpraviti niti z ukazom *wipe-out*, ki na krmilniku pobriše vse podatke o stanju omrežja. Krmilna ravnina je delovala pravilno šele z deaktivacijo in ponovno aktivacijo podsistema OpenFlow ter aplikacije za posredovanje na drugi plasti in brisanjem stanja. Večje težave smo opazili v primerih, ko smo onemogočili zgolj povezavo med določeno instanco krmilnika in stikali, saj izvedba nadomestnega načina ni bila vedno uspešna, pri čemer je krmilnik ostal v napačnem stanju.

Zaradi omenjenih težav, ki so se pojavljale tudi po spremembi na vmesnem času izdano novejšo različico krmilnika 1.10.4, smo teste izvedli v omejenem obsegu. Pri tem smo po potrebi ročno preverjali vzroke napačnih stanj, saj je na pravilno delovanje vplivalo veliko različnih faktorjev, ki pa niso bili

vedno posledica zasnove krmilnika.

6.2.1 Razpoložljivost krmilne ravnine

Sesutje procesa brez ponovnega zagona

V primeru sesutja procesa na enem od treh vozlišč krmilnika, smo opazili, da preostali instanci po izteku časovnika za začetek volitev najprej izvolita novo vodjo za podatkovni razdelek oziroma instanco algoritma Raft, katere vodja je bil sesuti proces. Po določenem času poteče tudi seja odjemalca gruče Raft sesutega strežnika, zato ga vodje izločijo iz skupin vozlišč, ki predstavljajo kandidate za gospodarje posameznih naprav. Ob tem izberejo nove gospodarje za naprave, katerih gospodar je bilo sesuto vozlišče. Novi gospodarji nato s sporočili *ROLE_REQUEST* od naprav zahtevajo vlogo gospodarja in pričnejo s procesiranjem njihovih dogodkov. S prevzemom vlog gospodarjev vseh naprav je uspešno zaključena tudi izvedba nadomestnega načina delovanja, zato je ponovno mogoča komunikacija med vsemi gostitelji v omrežju. V vmesnem času je še vedno mogoča komunikacija preko naprav z delujočimi gospodarji, vendar zaradi naključne razporeditve vlog gospodarjev, večino poti med gostitelji vsebuje vsaj eno napravo z nerazpoložljivo krmilno ravnino.

V primeru sesutja dveh ali več procesov, postane gruča v skladu s pričakovanji nerazpoložljiva, saj ni več možno doseči soglasja. V primeru, da ena instanca krmilnika ostane živa, le-ta krajši čas še ohrani vlogo gospodarja za obstoječe naprave, vendar ob prvem branju iz strogo konsistentne shrambe povezavo z njimi prekine. Opazili smo, da naprave v tem primeru poskušajo povezavo ponovno vzpostaviti, vendar krmilnik v fazi rokovanja povezavo prekine, saj ne more ustvariti skupine za izbiro vodje te naprave. Tudi izvedba vseh operacij, ki zahtevajo branje ali pisanje v strogo konsistentno shrambo, preko ukazne vrstice ali vmesnika REST ni mogoča, medtem ko je mogoč izpis eventualno konsistentnih podatkov, npr. seznama povezav, vendar so rezultati večinoma prazni, saj so vezani na omrežne naprave.

Ko smo scenarij ponovili s petimi oziroma sedmimi vozlišči krmilnika, smo ugotovili, da je razpoložljivost še vedno zagotovljena ob sesutju največ enega procesa. Razlog je namreč v načinu deljenja podatkov med več instanc algoritma Raft (razdelek 4.3.4), pri čemer je privzeta velikost razdelkov enaka tri ne glede na število vozlišč, kar smo ugotovili šele v sami fazi testiranja. Odpoved več kot enega vozlišča tako povzroči nerazpoložljivost najmanj ene instance algoritma Raft oziroma najmanj enega razdelka podatkov, medtem ko morajo biti za delovanje strogo konsistentne shrambe razpoložljivi vsi razdelki. Z večanjem števila vozlišč na ta način celo zmanjšamo razpoložljivost krmilnika, saj je verjetnost za odpoved posameznega vozlišča v gruči z večjim številom vozlišč večja.

Sesutje procesa

V primeru sesutja procesa na enem ali več vozlišč krmilnika ne da bi pred tem preprečili njegov ponovni zagon, je sistemski upravljavec storitev *systemd* zaznal sesutje in proces ponovno zagnal. Obnašanje je bilo v osnovi enako kot v primeru sesutja procesa brez ponovnega zagona, saj je do volitev novega vodje in izteka seje prišlo, preden se je proces ponovno zagnal. Po ponovnem zagonu se je proces uspešno priključil gruči, vendar je za razdelek podatkov, v katerem je imel pred sesutjem vodilno vlogo, ostal sledilec, saj se volitve izvedejo zgolj ob nezmožnosti komunikacije z vodilnim strežnikom. Ravno tako ni prevzel vloge gospodarja za nobeno izmed omrežnih naprav. V primeru sesutja procesa na več kot enem vozlišču je s pridružitvijo ponovno zagnanih procesov gruča spet postala razpoložljiva.

Zaustavitev procesa

Zaustavitev procesa pravzaprav ne predstavlja dejanske odpovedi, vendar je pravilno delovanje z vidika zagotavljanja visoke razpoložljivosti pomembno tudi v tem primeru npr. za potrebe vzdrževanja strežnika, na katerem teče proces. Glede na to, da lahko gruča nemoteno deluje tudi ob odsotnosti posamezne instance krmilnika in da je celoten postopek zaustavitve nadzorovan,

večjih težav nismo pričakovali. Ugotovili smo, da za razliko od sesutja, proces v primeru zaustavitve prekine povezavo z napravami, pri čemer pobriše nameščena tokovna pravila. Z vidika razpoložljivost to predstavlja problem, saj naprave do zaključka izvedbe nadomestnega načina delovanja ne morejo posredovati prometa niti na podlagi morebitnih že nameščenih tokovnih pravil.

Polna razdelitev omrežja

Ob razdelitvi omrežja, ki določeni skupini vozlišč v celoti onemogoča komunikacijo tako s stikali kot preostalimi vozlišči, je bila izvedba nadomestnega načina delovanja ravno tako uspešna v primeru nezmožnosti komunikacije največ enega vozlišča. Za razliko od sesutja, je bilo vozlišče ob tem še vedno aktivno, kar pomeni, da sta bila vmesnik REST in ukazna vrstica še vedno dostopna, vendar ni bila mogoča izvedba operacij, ki bi povzročile nekonsistentno stanje gruče. Po odpravi razdelitve smo občasno naleteli na anomalijo, ki smo jo opisali na začetku tega razdelka.

Razdelitev koordinacijskega omrežja

V primeru razdelitve omrežja, ki določeni skupini vozlišč onemogoča zgolj komunikacijo s preostalimi vozlišči, je pravilnost prevzema vlog gospodarjev še bolj pomembna, saj lahko vozlišča oziroma vozlišče v manjšini še vedno komunicira s stikali. Zaradi uporabe algoritma soglasja pri izbiri gospodarja kljub temu ne pride do tveganih stanj, kar smo preverili tudi v praksi. Izvedba nadomestnega načina delovanja je bila z izjemo že omenjenih anomalij uspešna v primerih, ko so lahko med seboj komunicirale vse razen ene instance. Smo pa ugotovili, da lahko instanca vlogo krmilne ravnine opravlja že v primeru, da lahko komunicira vsaj z eno aktivno instanco iz vsakega podatkovnega razdelka tudi, če le-ta ni vodja. V primeru treh instanc to pomeni, da zadostuje komunikacija z eno izmed preostalih dveh instanc, ki morata za delovanje gruče med seboj seveda uspešno komunicirati. Konkretna implementacija algoritma Raft namreč odjemalca, ki se poveže na

poljuben strežnik v gruči, ne preusmeri na vodjo, temveč lahko vodji zahteve tudi posreduje. Instanca v tem primeru ne more sodelovati v replikaciji podatkov z algoritmom Raft, lahko pa v vlogi odjemalca še vedno uporablja strogo konsistentno shrambo. Dodatno smo preverili še situacijo, ko s tem onemogočimo ravno povezavo z vodjo, saj bi v skladu z osnovnim algoritmom soglasja Raft, kot smo ga opisali v razdelku 3.6, to lahko vodilo v izmenično prevzemanje vloge vodje teh dveh instanc. Ugotovili smo, da implementacija v ta namen vsebuje dodaten mehanizem predglasovanja, s katerim strežnik Raft pred dejanskim začetkom volitev preveri, ali sploh lahko dobi večino glasov. Ostali strežniki predglasovanje zavrnejo, če so v manj kot enem intervalu preverjanja živosti prejeli zahtevo vodje oziroma imajo popolnejši dnevnik od strežnika, ki zahteva predglasovanje. Tako ne pride do morebitnih težav zaradi izmeničnega prevzemanja vloge vodje.

Razdelitev krmilnega omrežja

Ugotovili smo, da je pri razdelitvi omrežja med vozlišči krmilnika in omrežnimi napravami, načeloma mogoča izvedba nadomestnega načina delovanja v vseh primerih, ko lahko vsako stikalo komunicira vsaj z enim vozliščem krmilnika, saj tovrstne razdelitve ne vplivajo na delovanje same gruče. Vsaka instanca, ki izgubi povezavo s stikalom oziroma več stikali, sama poskrbi za odstop od vloge gospodarja oziroma kandidata za gospodarja, pri čemer se med preostalimi kandidati, ki s temi napravami še vedno lahko komunicirajo, izberejo novi gospodarji. Ti nato s sporočili *ROLE_REQUEST* od naprav zahtevajo vloge gospodarjev in pričnejo s procesiranjem njihovih dogodkov. Kot smo omenili, je pri izvedbi omenjenega scenarija večkrat prišlo tudi do popolnoma napačnega stanja krmilnika, v katerem nobena instanca ni prevzela vloge gospodarja za naprave, ki niso mogle več komunicirati z obstoječim gospodarjem. Opazili smo, da je bila tudi topologija omrežja, ki jo lahko vidimo preko spletnega vmesnika, napačna. Določene povezave so bile označene kot enosmerne oziroma so se sproti pojavljale in izginjale. Tudi lokacije gostiteljev so bile napačno prikazane. Stanje krmilnika se po daljšem čakanju

še vedno ni normaliziralo, zato smo poskusili s ponovnim zagonom celotne gruče, ki napake ravno tako ni odpravil. Po deaktivaciji in ponovni aktivaciji podsistema OpenFlow ter aplikacije za posredovanje na drugi plasti in brisanjem stanja, so vse naprave dobile nove gospodarje. Tudi topologija v spletnem vmesniku je bila videti pravilna, vendar komunikacija med gostitelji še vedno ni bila mogoča. Ob naslednji ponovitvi testa s popolnoma enakimi koraki je bila običajno izvedba nadomestnega načina delovanja uspešna, zato predvidevamo, da gre za hrošča pri sami implementaciji krmilnika.

Povečanje razpoložljivosti

Z namenom povečanja razpoložljivosti pri uporabi večjega števila vozlišč smo poiskusili spremeniti privzeto konfiguracijo razdelkov tako, da so velikosti razdelkov enake številu vseh vozlišč. V tem primeru je bila izvedba nadomestnega načina delovanja uspešna tudi ob sesutju oziroma izolaciji dveh od petih oziroma treh od sedmih vozlišč. Po drugi strani smo s tem žrtvovali skalabilnost krmilnika, ki je sicer glavni razlog za delitev podatkov med več instanc algoritma Raft. Alternativno bi lahko razpoložljivost v določeni meri povečali s spremembo implementacije strogo konsistentne shrambe na način, da bi za vsako instanco algoritma Raft preostala vozlišča, ki ne sodelujejo v aktivni replikaciji, uporabili kot pasivne in rezervne replike. Krmilnik še vedno ne bi bil odporen na več kot eno hkratno odpoved vozlišča, vendar bi lahko lahko nedelujoče replike avtomatsko nadomestil s pasivnimi ter tako preprečil nerazpoložljivost v primeru morebitnih nadaljnjih odpovedi.

6.2.2 Čas za izvedbo nadomestnega načina delovanja

Za primere, v katerih je bila izvedba nadomestnega načina delovanja uspešna, nas je zanimal čas, ki je potreben za njeno izvedbo z vidika razpoložljivosti oziroma nerazpoložljivosti programsko določenega omrežja. Torej čas, v katerem komunikacija med gostitelji zaradi odpovedi vsaj delno ali v celoti ni mogoča. Kot smo že omenili, je razpoložljivost precej odvisna od zasnove same aplikacije, pri čemer je razpoložljivost krmilne ravnine za raz-

položljivost programsko določenega omrežja najbolj kritična pri uporabi reaktivnega načina krmiljenja. Ugotovili smo, da je tudi pri reaktivnem načinu, ki ga uporablja aplikacija za posredovanje prometa na drugi plasti krmilnika ONOS, zmožnost komunikacije med dvema gostiteljema v primeru odpovedi še vedno odvisna od tega, ali katero izmed naprav na poti krmili instanca, pri kateri je prišlo do napake, ter ali gre za promet, ki pripada obstoječemu podatkovnemu toku. Ker nas je v resnici zanimalo, po kolikšnem času je omrežje v celoti razpoložljivo, smo prilagodili parametre aplikacije tako, da se vsak paket posreduje krmilniku, ki določi ustrezno akcijo, kot da bi šlo za neznan promet. S tem smo se izognili potrebi po simulaciji prometa, ki ne pripada obstoječim pravilom v tokovnih tabelah. Poleg tega smo prilagodili topologijo tako, da je različno število naprav povezanih zaporedno, medtem ko sta gostitelja, med katerima smo preverjali zmožnost komunikacije, priključena na prvo in zadnjo napravo. Komunikacija med gostiteljema je bila tako mogoča šele takrat, ko so pravilno delovale vse naprave v omrežju.

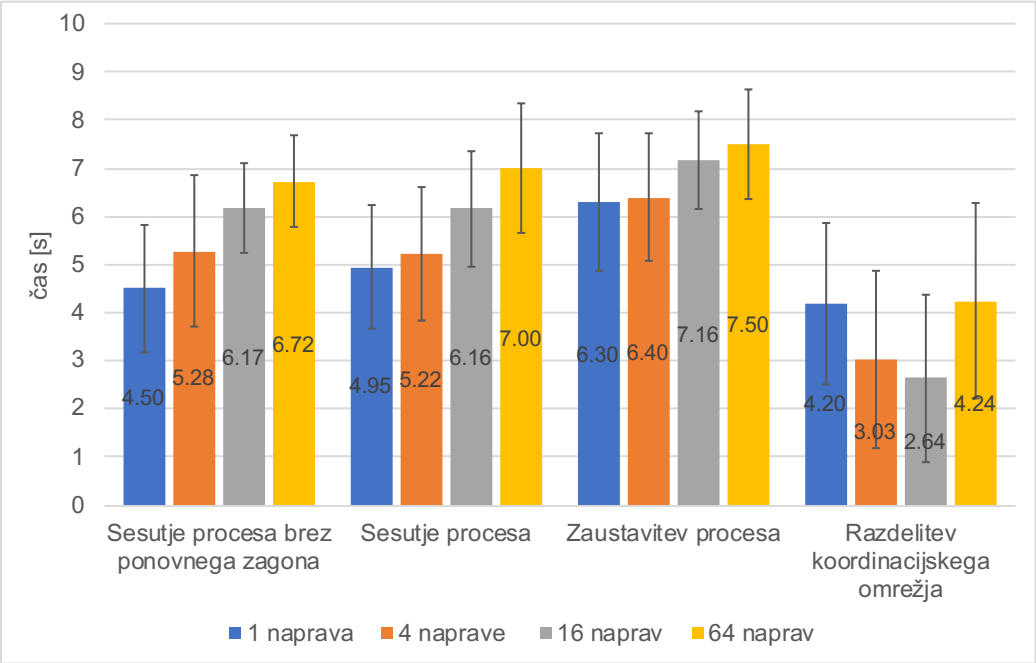
Proces vsake meritve je sestavljen iz inicializacije simulacijskega okolja, vzpostavitve gruče krmilnika ONOS, aktivacije podsistema OpenFlow, aktivacije aplikacije za posredovanje prometa na drugi plasti, nastavitve parametrov aplikacije, čakanja na uspešno povezavo krmilnika z vsemi stikali, prerazporeditve vlog gospodarjev, zagona orodja *tcpdump* za zajem prometa ICMP v imenskem prostoru prvega gostitelja, zagona orodja *ping* za preverjanje dosegljivosti drugega gostitelja v imenskem prostoru prvega gostitelja, čakanja na uspešno komunikacijo med gostiteljema, izpisa stanja krmilnika, simulacije odpovedi, čakanja na ponovno uspešnost komunikacije med gostiteljema, ponovnega izpisa stanja krmilnika, zaustavitve orodij *ping* ter *tcpdump*, izračuna časa za izvedbo nadomestnega načina delovanja ter zaustavitve simulacijskega okolja. Pri čakanju na uspešnost komunikacije po izvedbi odpovedi smo dodali pogoj za avtomatsko prekinitev testa, v kolikor komunikacija ni bila mogoča po več kot petih minutah. V tem primeru smo namesto časa shranili vrednost -1 kot indikator, da izvedba nadomestnega načina delovanja ni bila uspešna. To se je zaradi že omenjenih anomalij

večkrat zgodilo pri simulaciji razdelitve omrežja med krmilnikom in stikali. Pri zagonu orodja ping smo uporabili stikalo *-D* za beleženje časovnih žigov in stikalo *-i* z vrednostjo 0.01 za pošiljanje sporočil *ICMP ECHO_REQUEST* vsakih 10 ms. Standardni izhod smo pri tem preusmerili v datoteko, ki smo jo na koncu obdelali za izračun časa nedosegljivosti na podlagi intervala, ko gostitelj ni prejemal odzivov *ICMP ECHO_RESPONSE* od drugega gostitelja. Gre pravzaprav časovni interval izgube paketov, ki je posledica nedosegljivosti krmilne ravnine za eno ali več omrežnih naprav na poti. Vrednosti smo primerjali tudi z analizo zajetega prometa v orodju *Wireshark*.

Meritve smo izvedli s pomočjo gruče treh fizičnih strežnikov, kjer je bilo delovanje krmilnika najbolj stabilno. Pri tem smo simulirali različno število naprav. Ugotovili smo, da lahko zaradi omejitev simulacijskega okolja uporabimo največ 100 naprav oziroma navideznih stikal Open vSwitch. Pri stotih napravah so bile zakasnitve paketov že znatno večje, medtem ko so bile pri testu s 500 napravami zakasnitve tako velike, da krmilnik ni uspel odkriti vseh povezav med napravami, zato komunikacija niti ni bila mogoča. Vsako odpoved smo zato simulirali s topologijo, ki vsebuje 1, 4, 16 ali 64 naprav. Za vsako kombinacijo števila naprav in različnih odpovedi smo opravili 30 meritev. Pri izračunu povprečnih vrednosti in standardnih odklonov smo si pomagali s paketom *NumPy*.

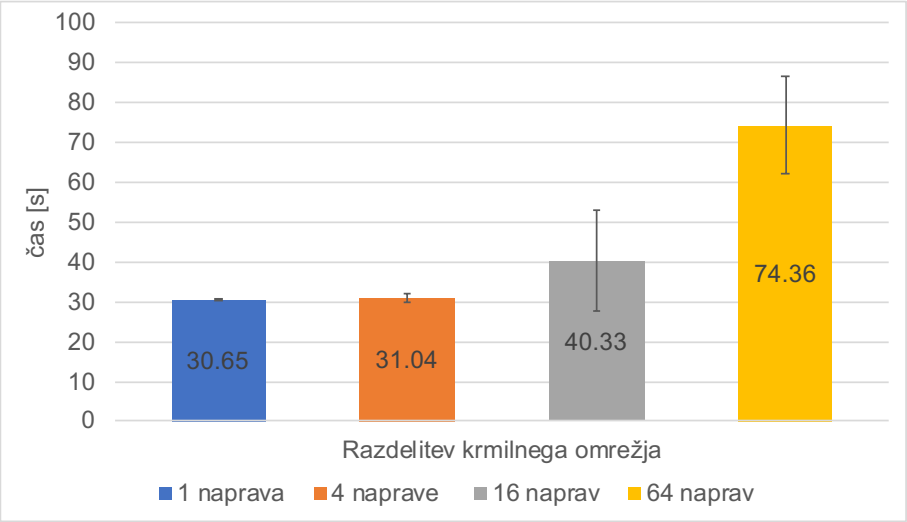
Rezultate meritev prikazujeta grafa na slikah 6.1 in 6.2, kjer so za vsak primer odpovedi zbrane povprečne vrednosti časa, ki je potreben za izvedbo nadomestnega načina delovanja pri različnem številu omrežnih naprav.

Časi za izvedbo nadomestnega načina delovanja v primeru sesutja procesa in sesutja s ponovnim zagonom so zelo podobni, saj pride do volitev novega vodje in izteka seje, s čimer se začne izbira novih gospodarjev, še preden se proces ponovno zažene. V fazi analize razpoložljivosti krmilne ravnine smo že ugotovili, da pri zaustavitvi procesa instanca predhodno ne prepusti vloge gospodarja za svoje naprave, zato se izbira novih gospodarjev začne šele po poteku njene seje, enako kot v primeru sesutja procesa. Vidimo lahko, da v praksi izvedba nadomestnega načina delovanja v primeru nadzorovane zau-



Slika 6.1: Čas za izvedbo nadomestnega načina delovanja v primeru odpovedi vozlišč ali komunikacijskih kanalov med njimi

stavitve traja še dlje kot v primeru sesutja. Ugotovili smo, da je to posledica dodatnih operacij, ki jih instanca izvede pred samo zaustavitvijo, pri čemer relativno hitro pobriše obstoječa tokovna pravila na napravah, ki jih nadzoruje, in prekine povezavo z njimi, s čimer onemogoči posredovanje prometa. Menimo, da bi bilo v postopek zaustavitve procesa pred samo prekinitvijo povezav smiselno vključiti spremembo gospodarja za vse naprave, ki jih ta instanca nadzoruje. Postopek smo preizkusili tako, da smo v simulacijskem okolju pred zaustavitvijo procesa z ukazom *device-role* za vse naprave ročno spremenili vlogo te instance na *NONE* ter počakali, da je instanca vlogo gospodarja vseh naprav tudi uspešno prepustila. V tem primeru je bil čas nedostopnosti omrežja zgolj nekaj 10 ms, kolikor je trajala sprememba vloge gospodarja s sporočili *ROLE_REQUEST*, saj ni potrebno čakanje na izbiro nove vodje algoritma Raft in poteka seje odjemalca, pa tudi v času izbire novega gospodarja za napravo, lahko z njo še vedno upravlja obstoječ gospo-



Slika 6.2: Čas za izvedbo nadomestnega načina delovanja v primeru razdelitev krmilnega omrežja

dar. Hkrati se s tem izognemo tudi problemu brisanja obstoječih tokovnih pravil ob zaustavitvi.

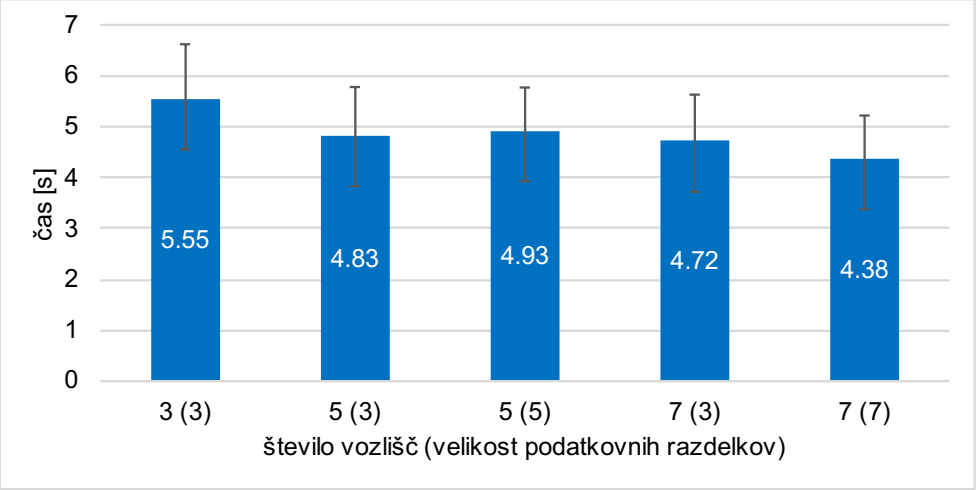
Za primer razdelitve koordinacijskega omrežja smo pričakovali podobne rezultate kot v prejšnjih primerih, saj se v osnovi z vidika gruče izvede enako zaporedje operacij. Kljub temu lahko vidimo, da je omrežje nerazpoložljivo manj časa. Ko smo primerjali časovne žige dogodkov v dnevniških datotekah krmilnika s časovnimi žigi zajetega prometa med gostiteljema ter časom simulacije dejanske odpovedi, smo ugotovili, da gostitelja določen čas po odpovedi še vedno komunicirata. Izkaže se, da instanca za nek krajši čas še vedno ohrani vlogo vodje za to napravo, ker pa za razliko od prejšnjih primerov s to napravo lahko tudi komunicira, je posredovanje prometa še vedno mogoče. Situacija bi bila nekoliko drugačna, v kolikor bi bilo procesiranje zahteve odvisno od branje podatka iz strogo konsistentne shrambe, ki v tem času ni več dosegljiva.

Ugotovili smo, da največ časa pri izvedbi nadomestnega načina delovanja v primeru odpovedi vozlišč ali komunikacijskih kanalov med njimi terja detekcija napake oziroma potek seje, ki je osnova za začetek postopka za izbiro

novih gospodarjev. Časovne omejitve za potek seje so relativno velike, saj ohranjanje seje zahteva replikacijo dnevnika Raft. Poleg tega ob odpovedi vodje, Raft ne garantira maksimalnega časa za izvedbo volitev. Novi vodja mora vse seje podaljšati, saj bi lahko v vmesnem času sicer potekle. V tem primeru mora poleg časa, ki je potreben za izvedbo volitev, pred dejanskim potekom seje preteči celotna časovna omejitev za potek seje. Ker ONOS deli podatke med več instanc algoritma Raft, v povprečju ob odpovedi poljubnega vozlišča odpove tudi en vodja. Predvidevamo, da bi lahko čas za izvedbo nadomestnega načina delovanja zmanjšali z uporabo aktivne detekcije živosti članov na vodilnem strežniku neodvisno od replikacije dnevnika.

V primeru razdelitev omrežja med krmilnikom in omrežnimi napravami smo zabeležili najslabši povprečen čas za izvedbo nadomestnega načina delovanja, kljub temu, da je čas odvisen predvsem od zaznave izgube krmilnega kanala med gospodarjem ter določeno omrežno napravo in ne od samega delovanja gruče. Pri tem je traba omeniti, da smo upoštevali samo uspešne meritve, saj približno polovica poskusov zaradi že omenjenih težav ni bila uspešna.

Dodatno smo po istem postopku izvedli še meritve časa za izvedbo nadomestnega načina delovanja pri različnem številu instanc krmilnika, in sicer s pomočjo navideznih strojev v oblačni infrastrukturi ponudnika Digital Ocean. Uporabili smo instance z 16GB delovnega pomnilnika, šestimi navideznimi centralnimi procesnimi enotami in pogonom SSD. Meritev smo izvedli za primer sesutja ene instance brez možnosti ponovnega zagona s topologijo, ki vsebuje 16 omrežnih naprav. Uporabili smo tri, pet oziroma sedem instanc krmilnika, pri čemer smo za primera petih in sedmih instanc poleg privzete velikosti podatkovnih rezdelkov poizkus ponovili tudi z uporabo razdelkov, ki se raztezajo čez celotno gručo. Rezultate prikazuje graf na sliki 6.3, kjer lahko za različno število instanc oziroma kombinacije števila instanc ter velikosti podatkovnih razdelkov, ki so navedene v oklepaju, razberemo povprečen čas stotih meritev. Vidimo lahko, da je bil čas za izvedbo nadomestnega načina delovanja pri večjem številu instanc manjši. To je predvsem posledica dej-

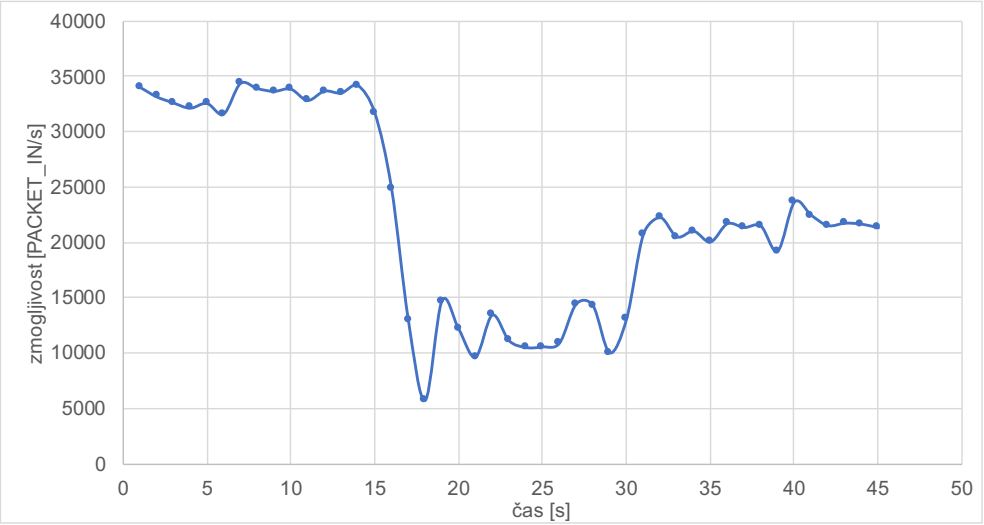


Slika 6.3: Čas za izvedbo nadomestnega načina delovanja pri različnem številu instanc krmilnika

stva, da je pri večjem številu instanc krmilnika na posamezno instanco priključeno manjše število naprav, za katere je potrebno ob izpadu izbrati novo vodjo. V primeru sedmih instanc krmilnika je bil povprečen čas z uporabo večjih podatkovnih razdelkov celo manjši od tistega v privzeti konfiguraciji, vendar predvidevamo, da bi bili rezultati drugačni ob večji obremenitvi krmilne ravnine, saj večji razdelki zahtevajo več režijskega dela pri replikaciji podatkov.

6.2.3 Vpliv na zmogljivost gruče

Vpliv izvedbe nadomestnega načina delovanja na zmogljivost celotne gruče smo merili z uporabo več instanc orodja *cbench*, kot smo opisali v razdelku 5.3.6. Meritev smo izvedli z uporabo treh fizičnih strežnikov za primer sesutja ene instance brez možnosti ponovnega zagona. Poleg orodja *cbench* smo na krmilnik povezali tudi 16 stikal Open vSwitch, da je prišlo tudi do menjave gospodarjev. Ugotovili smo, da je rezultate težko povprečiti, saj gre za časovni potek, ki se od meritve do meritve razlikuje.



Slika 6.4: Zmogljivost gruče v času izvedbe nadomestnega načina delovanja

Graf na sliki 6.4 prikazuje časovni potek povprečne zmogljivosti celotne gruče merjene v številu odgovorov na pakete *PACKET_IN*. Do odpovedi je prišlo v 15 sekundi meritve. Vidimo lahko, da se zmogljivost gruče najprej znatno zmanjša in je nato nekaj časa omejena. To je za čas izvedbe nadomestnega načina delovanja, ki pa je zaradi zelo velike obremenitve krmilne ravnine tudi precej daljši od meritev, ki smo jih izvedli pred tem. Po uspešni izvedbi nadomestnega načina delovanja se zmogljivost ustali okoli vrednosti, ki je sorazmerna s številom delujočih instanc v gruči.

6.3 Splošne ugotovitve

Med samo analizo obnašanja smo prišli še do nekaterih drugih ugotovitev, ki se nam zdijo ravno tako pomembne za uporabo v produkcijskem okolju.

6.3.1 Spletni grafični vmesnik in vmesnik REST

Krmilnik ONOS na severni strani med drugim omogoča tudi dostop do spletnega grafičnega vmesnika in vmesnika REST, preko katerega izpostavlja

določene operacije za lažjo integracijo z zunanjimi sistemi. Vmesnika sta dostopna na vseh instancah preko protokola HTTP oziroma HTTPS, vendar za visoko razpoložljiv dostop potrebujemo dodaten mehanizem izenačevanje bremena med trenutno delujoče instance krmilnika. Ugotovili smo, da izenačevanje bremena zgolj na podlagi zapisov DNS ne zadostuje, saj sta vmesnika dosegljiva tudi v primeru razdelitev, ko instanca ne more komunicirati s preostalo gručo. To pomeni da se odjemalci še vedno lahko povežejo nanju, čeprav vmesnik REST v tem primeru ne omogoča izvedbe operacij, ki zahtevajo uporabo strogo konsistentne shrambe, medtem ko se grafični vmesnik z izjemo prijavne forme niti ne prikaže. Ena izmed možnosti je uporaba izenačevalnika bremena, ki živost preverja na aplikacijski plasti in promet posreduje zgolj v celoti delujočim instancam. V ta namen smo preizkusili odprotokodno programsko opremo HAProxy in uspešno zagotovili visoko razpoložljiv dostop do omenjenih vmesnikov. Glavna težava je preverjanje živosti vmesnika. V dokumentaciji krmilnika smo zasledili, da lahko za preverjanje živosti vmesnika REST uporabimo prijavno formo spletnega vmesnika, vendar smo ugotovili, da je le-ta dosegljiva tudi v primeru, ko ni mogoča izvedba vseh operacij. Za preverjanje živosti smo uporabili akcijo GET na viru, ki vrne povzetek konfiguracije krmilnika. To sicer ni optimalno z vidika obremenitve strežnika, saj je branje konfiguracije v resnici nepotrebno. Krmilnik bi lahko ponujal dodaten vir vmesnika REST zgolj za preverjanje stanja instance. Vseeno smo na ta način zagotovili, da lahko HAProxy zazna nezmožnost izvedbe operacij posamezne instance v primeru razdelitev in zahteve preusmeri na preostale delujoče instanca. Pri tem smo ugotovili, da seja spletnega grafičnega vmesnika ni deljena, zato uporabnik pri prehodu med instancami krmilnika sejo izgubi. Izenačevalnik bremena smo nastavili tako, da vse zahteve določenega odjemalca posreduje na isto instanco, dokler je ta živa. Težava z izgubo seje se tako pojavi zgolj ob odpovedi instance. Pri uporabi vmesnika REST težav z izgubo sej ni, saj se uporablja osnovna avtentikacijska shema HTTP, vendar je to tudi edini možen način avtentikacije, ki pa ni najbolj varen.

6.3.2 Porazdelitev vlog gospodarjev

Videli smo, da vsaka instanca krmilnika nadzoruje določeno podmnožico omrežnih naprav, pri čemer je razporeditev vlog gospodarjev odvisna od tega, s katero instanco omrežna naprava prva vzpostavi krmilni kanal. To ni optimalno tako z vidika obremenitve same krmilne ravnine kot tudi z vidika razpoložljivosti. V primeru izpovedi nekaterih instanc je tako potrebna sprememba gospodarja za večje število naprav, kar podaljša čas za izvedbo nadomestnega načina delovanja. Možno je sicer ročno prerazporejanje naprav z ukazom *balance-masters*, ki naprave približno enakomerno razporedi med vse delujoče instance. Ukaz smo uporabili tudi v okviru naših testov, da smo se izognili neenakomerni razporeditvi naprav. Menimo, da bi bilo smiselno avtomatsko prerazporejanje naprav v primeru, ko se določena instanca krmilnika po odpovedi ponovno pridruži gruči. Rešitev bi lahko nadgradili še tako, da bi upoštevali topologijo omrežja. Trenutno so namreč vloge gospodarjev prerazporedijo naključno, kar pomeni, da v primeru odpovedi obstaja velika verjetnost, da je na določeni poti vsaj ena naprava, ki jo je ta instanca nadzoruje. Če bi upoštevali lokalnost, bi lahko omejili vpliv odpovedi na določen del omrežja. Za enakomernejšo obremenitev krmilne ravnine, bi bilo smiselno upoštevati tudi dejansko obremenitev posameznih instanc.

6.3.3 Delitev podatkov in vloge vodje

Krmilnik ONOS podatke strogo konsistentne shrambe z namenom večje učinkovitosti deli med več instanc algoritma Raft, kot smo videli v razdelku 4.3.4. Ker je število instanc algoritma enako številu instanc krmilnikov v gruči, ima lahko vsaka instanca vlogo vodje za določeno podmnožico podatkov oziroma instanco algoritma Raft. V praksi smo ugotovili, da razporeditev vlog ni vedno enakomerna, saj je izbira vodij odvisna od izteka časovnika za začetek volitev, ki je izbran naključno na vnaprej določenem intervalu. Tako se lahko zgodi, da ima posamezna instanca vlogo vodje tudi za 2 ali več instanci algoritma. Do tega pride tudi v primeru odpovedi, saj takrat vlogo vodje

prevzame ena izmed preostalih instanc in jo tudi obdrži dokler lahko komunicira s preostalimi vozlišči, četudi se instanca, ki je imela vlogo vodje pred tem, ponovno pridruži gruči.

Za enakomernejšo začetno razporeditev vlog bi lahko implementacijo prilagodili tako, da bi imela za vsako instanco algoritma po ena instanca krmilnika manjši parameter, ki določa interval za iztek časovnika, in s tem večjo možnost, da postane vodja. Pri tem bi morali paziti, da vrednost ne bi bila premajhna, saj bi s tem ogrozili živost algoritma. Druga možnost s katero bi lahko rešili tudi kasnejšo razporeditve vlog po odpovedi in ponovni pridružitvi instanc je dodaten mehanizem za prenos vloge vodje na drug strežnik v gruči, pri čemer bi lahko želene vodje izbrali deterministično z eno izmed instanc algoritma na enak način, kot poteka izbira vodij za druge skupine npr. za izbiro vloge gospodarja posamezne naprave.

6.3.4 Nadgradnje

Kot eno izmed potencialnih pomanjkljivosti za produkcijsko uporabo krmilnika ONOS bi izpostavili tudi nezmožnost nadgradnje krmilnika med delovanjem brez prekinitve. Trenutno je za prehod na novejšo različico krmilnika potrebna zaustavitev celotne gruče ter sveža namestitev in ponovna konfiguracija krmilnika, kar pomeni, da je za čas nadgradnje celotno programsko določeno omrežje nerazpoložljivo. Poleg tega tudi sam proces prenosa in združljivost konfiguracije različice nista posebej definirana.

Poglavje 7

Sklep

V magistrskem delu smo analizirali koncepte programsko določenih omrežij. Raziskali smo, kje ima njihova uporaba največji potencial ter identificirali različne izzive, ki otežujejo uvedbo v produkcijska okolja. Pri tem smo kot eno izmed ključnih ovir prepoznali problem zagotavljanja visoke razpoložljivosti in skalabilnosti omrežja zaradi centralizacije krmilne ravnine. Prišli smo do spoznanja, da je kljub logični centraliziranosti modela nujna fizična porazdeljenost krmilnika. V nadaljevanju smo preučili temeljne lastnosti porazdeljenih sistemov ter različne omejitve pri njihovi zasnovi in implementaciji. Osredotočili smo se na algoritme soglasja, ki rešujejo temeljni problem porazdeljenih sistemov, odpornih na napake, in tako predstavljajo ključno komponento za zagotavljanje visoke razpoložljivosti krmilnikov s tem pa tudi samih programsko določenih omrežij. Nato smo predstavili ključne strategije pri zasnovi porazdeljenih krmilnikov in raziskali, katere odprtokodne implementacije omogočajo njihovo visoko razpoložljivost. Ugotovili smo, da je večina implementacij z izjemo krmilnikov OpenDaylight, ONOS, OpenMUL in RuNOS še vedno fizično centraliziranih in ne zagotavlja visoke razpoložljivosti krmilne ravnine. Kot dominantni rešitvi tako po obsegu funkcionalnosti kot tudi z vidika porazdeljene zasnove smo označili krmilnika OpenDaylight in ONOS. Oba krmilnika problem koordinacije rešujeta z uporabo algoritma soglasja Raft. Za nadaljnjo analizo smo izbrali krmilnik

ONOS. Z vidika zagotavljanja visoke razpoložljivosti sicer nismo prepoznali večjih razlik v primerjavi s krmilnikom OpenDaylight, smo pa ocenili, da arhitektura krmilnika ONOS omogoča večjo skalabilnost. Arhitekturo krmilnika ONOS smo tudi podrobno analizirali.

Glavni prispevek magistrskega dela je analiza obnašanja programsko določenega omrežja v primeru različnih odpovedi, na podlagi katere smo ovrednotili izbrano implementacijo krmilnika z vidika zagotavljanja visoke razpoložljivosti in podali predloge za izboljšave. V ta namen smo najprej vzpostavili pilotno okolje z izbranim krmilnikom ONOS in simulatorjem programsko določenih omrežij Mininet. Za izvedbo testov smo razvili testno ogrodje, ki predstavlja razširitev simulacijskega okolja Mininet in omogoča interakcijo s porazdeljenim krmilnikom ONOS ter simulacijo različnih odpovedi. Pri tem smo avtomatizirali celoten proces vzpostavitve krmilnika. Z uporabo omenjene rešitve smo uspešno simulirali scenarije, ki vključujejo različne odpovedi vozlišč krmilnika in komunikacijskih kanalov, ter pri tem analizirali obnašanje sistema.

Videli smo, da krmilnik z uporabo gruče vsaj treh vozlišč zagotavlja razpoložljivost krmilne ravnine pri izpadu oziroma nezmožnosti komunikacije največ enega vozlišča. Pri tem nismo zasledili pojava tveganih stanj pri prevzemu nadrejene vloge krmilnika ali drugih nepravilnosti z izjemo anomalij pri delovanju, za katere menimo, da niso posledica same zasnove krmilnika. Pokazali smo, da je mogoče z uporabo večjega števila vozlišč in spremembo konfiguracije podatkovnih razdelkov zagotoviti razpoložljivost tudi ob izpadu več kot ene instance krmilnika.

V nadaljevanju smo z uporabo lastne metode na podlagi intervala izgube paketov izmerili tudi čas, ki je potreben za izvedbo nadomestnega načina delovanja. Ker smo ugotovili, da je razpoložljivost omrežja v primeru težav s krmilno ravnino odvisna od različnih faktorjev in da je rezultate na podlagi specifičnega primera uporabe zelo težko posplošiti, smo meritev opravili za najslabši možen primer, kjer je za komunikacijo potrebno pravilno delovanje vseh naprav v omrežju. Prišli smo do spoznanja, da pri odpovedi

vozlišč ali komunikacijskih kanalov med njimi največ časa terja sam potek seje, zato smo za hitrejšo izvedbo nadomestnega načina delovanja predlagali detekcijo odpovedi, ki ni odvisna od replikacije dnevnika. Pokazali smo tudi, kako se lahko v primeru nadzorovane zaustavitve procesa skoraj v celoti izognemo nerazpoložljivosti omrežja. Občutno slabše rezultate smo opazili pri razdelitvi krmilnega omrežja, čeprav smo v tem primeru pričakovali hitrejšo zaznavanje napake, saj le-ta v osnovi ni odvisna od replikacije dnevnika v gruči Raft. Predvidevamo, da bi lahko čas bistveno izboljšali že s pogostejšim pošiljanjem sporočil OpenFlow *ECHO_REQUEST* in krajšo časovno omejitvijo za zaznavanje odpovedi.

Tako v fazi vzpostavitve kot med samim delovanjem krmilnika oziroma simulacijo odpovedi smo naleteli na različna napačna stanja krmilnika, ki so povzročila nerazpoložljivost programsko določenega omrežja tudi v primerih, ko tega nismo pričakovali. Ugotovili smo, da ne gre za problem zasnove krmilnika temveč hroščev pri njegovi implementaciji. Vseeno se nam zdi to pomemben pokazatelj nezrelosti rešitve, kar je po našem mnenju trenutno tudi največja ovira pri uporabi krmilnika ONOS v produkcijskem okolju. Pričakujemo, da bodo te težave odpravljene s prihajajočimi različicami, saj se rešitev aktivno razvija.

Menimo, da bi lahko z implementacijo podanih predlogov izboljšali razpoložljivost krmilnika, hkrati pa je po našem mnenju smiselno predvsem rezultate meritev časa, ki je potreben za izvedbo nadomestnega načina delovanja, upoštevati tudi pri zasnovi aplikacij. Videli smo, da je čas nerazpoložljivosti krmilne ravnine nekaj sekund tudi v najboljšem primeru, kar lahko za določene primere že predstavlja problem. S proaktivnim nameščenjem pravil v tem času zgolj ni mogoča sprememba poti, medtem ko lahko z vnaprejšno namestitvijo alternativnih poti, ki jih omogočajo skupinske tabele OpenFlow, v določeni meri zagotovimo celo spremembo posredovanja brez posega krmilnika in s tem omejimo težave zaradi nerazpoložljivosti krmilne ravnine.

Za nadaljnje delo bi bila po našem mnenju smiselna implementacija pre-

dlaganih izboljšav. Njihovo učinkovitost bi lahko preizkusili z uporabo ogrodja in testov, ki smo jih razvili v okviru tega dela. Prav tako menimo, da bi bilo smiselno za primerjavo teste ponoviti še z drugimi implementacijami krmilnikov.

Po našem mnenju je prihodnost programsko določenih omrežij vsekakor obetavna in pričakujemo, da bo z dostopnostjo stabilnih implementacij krmilnikov, ki zagotavljajo ustrezno mero visoke razpoložljivosti, tudi njihova uporaba v praksi vse bolj pogosta.

Literatura

[1] Cisco, "The zettabyte era—trends and analysis." [ONLINE]. Dosegljivo: `http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html`, 2016.

[2] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller in N. Rao, "Are we ready for sdn? implementation challenges for software-defined networks," *IEEE Communications Magazine*, vol. 51, str. 36–43, July 2013.

[3] H. Kim in N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, str. 114–119, February 2013.

[4] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky in S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, str. 14–76, Jan 2015.

[5] P. Göransson, C. Black in T. Culver, *Software Defined Networks: A Comprehensive Approac*. Boston: Morgan Kaufmann, druga izd., 2017.

[6] O. Salman, I. H. Elhajj, A. Kayssi in A. Chehab, "Sdn controllers: A comparative study," v *2016 18th Mediterranean Electrotechnical Conference (MELECON)*, str. 1–6, April 2016.

[7] G. Coulouris, J. Dollimore, T. Kindberg in G. Blair, *Distributed Systems: Concepts and Design*. USA: Addison-Wesley Publishing Company, 5th izd., 2011.

[8] S. Gilbert in N. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, str. 30–36, Feb 2012.

[9] E. Brewer, "Cap twelve years later: How the "rules"have changed," *Computer*, vol. 45, str. 23–29, Feb 2012.

[10] D. Abadi, "Consistency tradeoffs in modern distributed database system design: Cap is only part of the story," *Computer*, vol. 45, str. 37–42, Feb 2012.

[11] O. Blial, M. Ben Mamoun in B. Redouane, "An overview on sdn architectures with multiple controllers," vol. 2016, str. 1–8, 01 2016.

[12] Y. E. Oktian, S. Lee, H. Lee in J. Lam, "Distributed sdn controller system: A survey on design choice," *Computer Networks*, vol. 121, no. Supplement C, str. 100 – 111, 2017.

[13] D. Levin, A. Wundsam, B. Heller, N. Handigol in A. Feldmann, "Logically centralized?: State distribution trade-offs in software defined networks," v *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, (New York, NY, USA), str. 1–6, ACM, 2012.

[14] F. A. Botelho, F. M. V. Ramos, D. Kreutz in A. N. Bessani, "On the feasibility of a consistent and fault-tolerant data store for sdns," v *2013 Second European Workshop on Software Defined Networks*, str. 38–43, Oct 2013.

[15] M. Casado, N. Foster in A. Guha, "Abstractions for software-defined networks," *Commun. ACM*, vol. 57, str. 86–95, sept. 2014.

[16] D. Ongaro in J. Ousterhout, "In search of an understandable consensus algorithm," v *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), str. 305–320, USENIX Association, 2014.

[17] N. Feamster, J. Rexford in E. Zegura, "The road to sdn," *Queue*, vol. 11, str. 20:20–20:40, dec. 2013.

[18] Open Networking Foundation, "Tr-502: Sdn achitecture." [ONLINE]. Dosegljivo: `https://www.opennetworking.org/wp-content/uploads/2013/02/TR_SDN_ARCH_1.0_06062014.pdf`, 2014.

[19] Open Networking Foundation, "Tr-521: Sdn achitecture." [ONLINE]. Dosegljivo: `https://www.opennetworking.org/wp-content/uploads/2014/10/TR-521_SDN_Architecture_issue_1.1.pdf`, 2016.

[20] S. Shenker, "The future of networking, and the past of protocols." [ONLINE]. Dosegljivo: `https://www.youtube.com/watch?v=YHeyuD89n1Y`, 2011.

[21] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka in T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Communications Surveys Tutorials*, vol. 16, str. 1617–1634, Third 2014.

[22] IETF, "Rfc1987: Ipsilon's general switch management protocol specification." [ONLINE]. Dosegljivo: `https://tools.ietf.org/html/rfc1987`, 1996.

[23] Ipsilon Networks, "The intelligence of routing, the performance of switching." [ONLINE]. Dosegljivo: `http://cs.hadassah.ac.il/staff/martin/Seminar/ip_switch/wp-ipswitch.html`, 1996.

[24] IETF, "Datatracker: Forwarding and control element separation (forces) working group." [ONLINE]. Dosegljivo: https://datatracker.ietf.org/wg/forces/about/, 2014.

[25] IETF, "RFC4655: A path computation element (pce)-based architecture." [ONLINE]. Dosegljivo: https://tools.ietf.org/html/rfc4655, 2006.

[26] IETF, "Rfc5440: Path computation element (pce) communication protocol (pcep)." [ONLINE]. Dosegljivo: https://tools.ietf.org/html/rfc5440, 2009.

[27] IETF, "Rfc7752: North-bound distribution of link-state and traffic engineering (te) information using bgp." [ONLINE]. Dosegljivo: https://tools.ietf.org/html/rfc7752, 2016.

[28] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan in H. Zhang, "A clean slate 4d approach to network control and management," *SIGCOMM Comput. Commun. Rev.*, vol. 35, str. 41–54, okt. 2005.

[29] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown in S. Shenker, "Ethane: Taking control of the enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, str. 1–12, avg. 2007.

[30] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker in J. Turner, "Openflow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, str. 69–74, mar. 2008.

[31] MIT Technology Review, "Tr10: Software-defined networking." [ONLINE]. Dosegljivo: http://www2.technologyreview.com/news/412194/tr10-software-defined-networking/, 2009.

[32] Open Networking Foundation. [ONLINE]. Dosegljivo: https://www.opennetworking.org, 2017.

[33] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart in A. Vahdat, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, vol. 43, str. 3–14, avg. 2013.

[34] Open Networking Foundation, "Software-Defined Networking: The New Norm for Networks." [ONLINE]. Dosegljivo: `https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf`, 2012.

[35] I. Pepelnjak, "MPLS and MPLS Transport Profile (MPLS-TP): The technology differences." [ONLINE]. Dosegljivo: `http://bit.ly/gYu6cb`, 2010.

[36] IETF, "Rfc7674: Dissemination of flow specification rules." [ONLINE]. Dosegljivo: `https://tools.ietf.org/html/rfc5575`, 2016.

[37] S. Li, D. Hu, W. Fang, S. Ma, C. Chen, H. Huang in Z. Zhu, "Protocol oblivious forwarding (pof): Software-defined networking with enhanced programmability," *IEEE Network*, vol. 31, str. 58–66, March 2017.

[38] Project Floodlight, "Indigo." [ONLINE]. Dosegljivo: `http://www.projectfloodlight.org/indigo/`, 2017.

[39] NEC, "ProgrammableFlow Virtual Switch PF1000." [ONLINE]. Dosegljivo: `http://www.nec.com/en/global/prod/pflow/pf1000.html`, 2017.

[40] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon in M. Casado, "The design and implementation of open vswitch," v *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, (Berkeley, CA, USA), str. 117–130, USENIX Association, 2015.

[41] Juniper, "Juniper QFX10002 Technical Overview." [ONLINE]. Dosegljivo: https://forums.juniper.net/t5/Data-Center-Technologists/Juniper-QFX10002-Technical-Overview/ba-p/270358, 2017.

[42] Cumulus Networks, "Cumulus Linux." [ONLINE]. Dosegljivo: https://cumulusnetworks.com/products/cumulus-linux/, 2017.

[43] PICA8, "PicOS." [ONLINE]. Dosegljivo: http://www.pica8.com/products/picos, 2017.

[44] NoviFlow, "NoviSwitch." [ONLINE]. Dosegljivo: https://noviflow.com/products/noviswitch/, 2017.

[45] F. Wang, H. Wang, B. Lei in W. Ma, "A research on high-performance sdn controller," v *2014 International Conference on Cloud Computing and Big Data*, str. 168–174, Nov 2014.

[46] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown in S. Shenker, "Nox: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, str. 105–110, julij 2008.

[47] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama in S. Shenker, "Onix: A distributed control platform for large-scale production networks," v *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, (Berkeley, CA, USA), str. 351–364, USENIX Association, 2010.

[48] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, P. Ingram, E. Jackson, A. Lambeth, R. Lenglet, S.-H. Li, A. Padmanabhan, J. Pettit, B. Pfaff, R. Ramanathan, S. Shenker, A. Shieh, J. Stribling, P. Thakkar, D. Wendlandt, A. Yip in R. Zhang, "Network virtualization in multi-tenant datacenters," v *11th*

USENIX Symposium on Networked Systems Design and Implementation (NSDI 14), (Seattle, WA), str. 203–216, USENIX Association, 2014.

[49] D. Erickson, "The beacon openflow controller," v *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, (New York, NY, USA), str. 13–18, ACM, 2013.

[50] BigSwitchNetworks, "Project Floodlight." [ONLINE]. Dosegljivo: `http://www.projectfloodlight.org/floodlight/`, 2017.

[51] Trema, "Full-Stack OpenFlow Framework in Ruby and C." [ONLINE]. Dosegljivo: `https://trema.github.io/trema/`, 2017.

[52] BigSwitchNetworks, "Project Floodlight." [ONLINE]. Dosegljivo: `http://www.projectfloodlight.org/floodlight/`, 2017.

[53] Juniper, "OpenContrail." [ONLINE]. Dosegljivo: `http://www.opencontrail.org/`, 2017.

[54] Linux Fundation, "OpenDaylight." [ONLINE]. Dosegljivo: `https://www.opendaylight.org/`, 2017.

[55] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow in G. Parulkar, "Onos: Towards an open, distributed sdn os," v *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, (New York, NY, USA), str. 1–6, ACM, 2014.

[56] openMUL, "High performance SDN." [ONLINE]. Dosegljivo: `http://www.openmul.org/`, 2017.

[57] A. Shalimov, S. Nizovtsev, D. Morkovnik in R. Smeliansky, "The runos openflow controller," v *2015 Fourth European Workshop on Software Defined Networks*, str. 103–104, Sept 2015.

[58] Open Networking Foundation, "Northbound Interfaces Wroking Group." [ONLINE]. Dosegljivo: https://www.opennetworking.org/images/stories/downloads/working-groups/charter-nbi.pdf, 2013.

[59] F. Khan, "Intent-based Networking – A Must for SDN." [ONLINE]. Dosegljivo: http://resources.solarwinds.com/intent-based-networking-not-an-option-but-a-must-for-sdn/, 2015.

[60] Open Networking Foundation, "OpenFlow Switch Specification 1.5.1." [ONLINE]. Dosegljivo: https://www.opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf, 2017.

[61] D. Samociuk, "Secure communication between openflow switches and controllers," v *The Seventh International Conference on Advances in Future Internet*, AFIN 2015, str. 32–37, IARIA, 2015.

[62] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill in M. Nanduri, "Achieving high utilization with software-driven wan." [ONLINE]. Dosegljivo: https://www.microsoft.com/en-us/research/publication/achieving-high-utilization-with-software-driven-wan/, August 2013.

[63] Nuage, "SD-WAN." [ONLINE]. Dosegljivo: http://www.nuagenetworks.net/sd-wan-branch-office/, 2017.

[64] VMware, "VMware NSX Network Virtualization and Security Platform." [ONLINE]. Dosegljivo: https://www.vmware.com/products/nsx.html, 2017.

[65] C. S. Li, B. L. Brech, S. Crowder, D. M. Dias, H. Franke, M. Hogstrom, D. Lindquist, G. Pacifici, S. Pappe, B. Rajaraman, J. Rao,

R. P. Ratnaparkhi, R. A. Smith in M. D. Williams, "Software defined environments: An introduction," *IBM Journal of Research and Development*, vol. 58, str. 1:1–1:11, March 2014.

[66] ETSI, "Network functions virtualisation." [ONLINE]. Dosegljivo: `http://www.etsi.org/technologies-clusters/technologies/nfv`, 2017.

[67] ONF, "CORD." [ONLINE]. Dosegljivo: `https://opencord.org/`, 2017.

[68] A. Gudipati, D. Perry, L. E. Li in S. Katti, "Softran: Software defined radio access network," v *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, (New York, NY, USA), str. 25–30, ACM, 2013.

[69] K. Yap, M. Motiwala, J. Rahe, S. Padgett, M. Holliman, G. Baldus, M. Hines, T. Kim, A. Narayanan, A. Jain, V. Lin, C. Rice, B. Rogan, A. Singh, B. Tanaka, M. Verma, P. Sood, M. Tariq, M. Tierney, D. Trumic, V. Valancius, C. Ying, M. Kallahalla, B. Koley in A. Vahdat, "Taking the edge off with espresso: Scale, reliability and programmability for global internet peering," 2017.

[70] V. Hadzilacos in S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," tech. rep., Cornell University, 1994.

[71] C. Dwork, N. Lynch in L. Stockmeyer, "Consensus in the presence of partial synchrony," *J. ACM*, vol. 35, str. 288–323, apr. 1988.

[72] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, str. 558–565, julij 1978.

[73] E. A. Akkoyunlu, K. Ekanadham in R. V. Huber, "Some constraints and tradeoffs in the design of network communications," v *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, SOSP '75, (New York, NY, USA), str. 67–74, ACM, 1975.

[74] L. Lamport, "Proving the correctness of multiprocess programs," *IEEE Transactions on Software Engineering*, vol. SE-3, str. 125–143, March 1977.

[75] M. J. Fischer, N. A. Lynch in M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, str. 374–382, apr. 1985.

[76] E. A. Brewer, "Towards robust distributed systems (abstract)," v *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, (New York, NY, USA), str. 7–, ACM, 2000.

[77] S. Gilbert in N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, vol. 33, str. 51–59, jun. 2002.

[78] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Comput. Surv.*, vol. 22, str. 299–319, dec. 1990.

[79] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, str. 133–169, maj 1998.

[80] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," v *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), str. 335–350, USENIX Association, 2006.

[81] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes in R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, str. 4:1–4:26, jun. 2008.

[82] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd in V. Yushprakh, "Megastore: Providing

scalable, highly available storage for interactive services," v *Proceedings of the Conference on Innovative Data system Research (CIDR)*, str. 223–234, 2011.

[83] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang in D. Woodford, "Spanner: Google’s globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, str. 8:1–8:22, avg. 2013.

[84] L. Lamport *et al.*, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, str. 18–25, 2001.

[85] B. M. Oki in B. H. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," v *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, (New York, NY, USA), str. 8–17, ACM, 1988.

[86] F. P. Junqueira, B. C. Reed in M. Serafini, "Zab: High-performance broadcast for primary-backup systems," v *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, (Washington, DC, USA), str. 245–256, IEEE Computer Society, 2011.

[87] Diego Ongaro, "Consensus: Bridging Theory and Practice." [ONLINE]. Dosegljivo: `https://github.com/ongardie/dissertation`, 2014.

[88] Facebook, "HydraBase – The evolution of HBase@Facebook." [ONLINE]. Dosegljivo: `https://code.facebook.com/posts/321111638043166/hydrabase-the-evolution-of-hbase-facebook/`, 2014.

[89] CoreOS, "etcd - A distributed, reliable key-value store for the most critical data of a distributed system." [ONLINE]. Dosegljivo: `https://coreos.com/etcd/`, 2017.

[90] Linux Foundation, "Kubernetes - Production-Grade Container Orchestration." [ONLINE]. Dosegljivo: `https://kubernetes.io`, 2017.

[91] HashiCorp, "Consul - Service Discovery and Configuration Made Easy." [ONLINE]. Dosegljivo: `https://www.consul.io`, 2017.

[92] Atomix, "An open source software stack for distributed systems infrastructure." [ONLINE]. Dosegljivo: `http://atomix.io`, 2017.

[93] Y. Zhang, E. Ramadan, H. Mekky in Z.-L. Zhang, "When raft meets sdn: How to elect a leader and reach consensus in an unruly network," v *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet'17, (New York, NY, USA), str. 1–7, ACM, 2017.

[94] P. Lin, J. Bi in Y. Wang, *East-West Bridge for SDN Network Peering*, str. 170–181. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013.

[95] A. Tootoonchian in Y. Ganjali, "Hyperflow: A distributed control plane for openflow," v *Proceedings of the 2010 Internet Network Management Conference on Research on Enterprise Networking*, INM/WREN'10, (Berkeley, CA, USA), str. 3–3, USENIX Association, 2010.

[96] K. Phemius, M. Bouet in J. Leguay, "Disco: Distributed multi-domain sdn controllers," v *2014 IEEE Network Operations and Management Symposium (NOMS)*, str. 1–4, May 2014.

[97] N. Katta, H. Zhang, M. Freedman in J. Rexford, "Ravana: Controller fault-tolerance in software-defined networking," v *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, (New York, NY, USA), str. 4:1–4:12, ACM, 2015.

[98] ONOSProject, "Open Network Operating System Wiki." [ONLINE]. Dosegljivo: `https://wiki.onosproject.org`, 2017.

[99] hazelcast, "In-Memory Data Grid." [ONLINE]. Dosegljivo: `https://hazelcast.com/`, 2017.

[100] GitHub, "Mininet: Rapid Prototyping for Software Defined Networks." [ONLINE]. Dosegljivo: `https://github.com/mininet/mininet`, 2017.

[101] Bugzilla, "Bug 81211 - Regression: unregister_netdevice: waiting for ¡virtual network interface¿ to become free." [ONLINE]. Dosegljivo: `https://bugzilla.kernel.org/show_bug.cgi?id=81211`, 2017.

[102] B. Lantz, B. Heller in N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," v *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, (New York, NY, USA), str. 19:1–19:6, ACM, 2010.

[103] GitHub, "Loxigen: OpenFlow protocol bindings for multiple languages." [ONLINE]. Dosegljivo: `https://github.com/floodlight/loxigen`, 2017.