UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Miha Baloh

Realizacija porazdeljenega algoritma za izračun drevesa najkrajših poti

DIPLOMSKO DELO
NA VISOKOŠOLSKEM STROKOVNEM ŠTUDIJU

Mentor: doc. dr. Boštjan Slivnik

Ljubljana, 2010

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00529/2010
Datum: 01.09.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MIHA BALOH**

Naslov: **REALIZACIJA PORAZDELJENEGA ALGORITMA ZA IZRAČUN DREVESA NAJKRAJŠIH POTI**

THE IMPLEMENTATION OF THE DISTRIBUTED SHORTEST PATHS ALGORITHM

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

V programskem jeziku Java realizirajte porazdeljeni algoritem za izračun drevesa najkrajših poti med vozlišči v porazdeljenem sistemu. Komunikacija med vozlišči naj temelji na Javini tehnologiji oddaljenega klicanja metod (RMI). Natančno opišite izbrani porazdeljeni algoritem in njegovo realizacijo ter opravljene preizkuse delovanja.

Mentor:
B. Slivnik
doc. dr. Boštjan Slivnik

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00529/2010
Datum: 01.09.2010

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **MIHA BALOH**

Naslov: **REALIZACIJA PORAZDELJENEGA ALGORITMA ZA IZRAČUN DREVESA NAJKRAJŠIH POTI**

THE IMPLEMENTATION OF THE DISTRIBUTED SHORTEST PATHS ALGORITHM

Vrsta naloge: Diplomsko delo visokošolskega strokovnega študija

Tematika naloge:

V programskem jeziku Java realizirajte porazdeljeni algoritem za izračun drevesa najkrajših poti med vozlišči v porazdeljenem sistemu. Komunikacija med vozlišči naj temelji na Javini tehnologiji oddaljenega klicanja metod (RMI). Natančno opišite izbrani porazdeljeni algoritem in njegovo realizacijo ter opravljene preizkuse delovanja.

Mentor:
B. Slivnik
doc. dr. Boštjan Slivnik

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisani/-a Miha Baloh,

z vpisno številko 63060015,

sem avtor/-ica diplomskega dela z naslovom:

Realizacija porazdeljenega algoritma za izračun drevesa najkrajših poti

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  doc. dr. Boštjan Slivnik

  in somentorstvom (naziv, ime in priimek)

- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela

- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne _____ Podpis avtorja/-ice: _____

Zahvala

Doc. dr. Boštjanu Slivniku za pomoč, pregledovanje in nasvete pri izdelavi diplomske naloge.

Sodelavcem v službi za razumevanje pri izdelavi diplomske naloge.

Družini in punci, ki so mi vedno stali ob strani in so mi vedno v opomin, da v življenju obstajajo tudi pomembnejše stvari od računalnikov.

Hvala vsem.

Kazalo

Povzetek 1
Abstract 3
1. Uvod 5
1.1. Moorov zakon 5
1.2. Večjedrni procesorji 7
1.3. Vzporedni računalniki 7
1.3.1. Sistem s porazdeljenim pomnilnikom 8
1.3.2. Sistem s skupnim pomnilnikom 9
1.4. Porazdeljeni sistemi 9
1.5. Porazdeljeno procesiranje in programiranje 10
2. Porazdeljen algoritem za izračun drevesa najkrajših poti 13
2.1. Pregled 13
2.2. Definicija problema drevesa najkrajših poti 13
2.3. Metode reševanja problema na klasičen način 15
2.3.1. Dijkstrin algoritem 15
2.3.2. Bellman–Fordov algoritem 15
2.4. Definicija reševanja na porazdeljen način 16
2.5. Algoritem poplavljanja 16
2.5.1. Slabe lastnosti 16
2.5.2. Dobre lastnosti 16
2.5.3. Prikaz delovanja algoritma 16
2.6. Metoda reševanja na porazdeljen način z metodo poplavljanja 17
3. Java RMI programska tehnologija 19
3.1. Osnovni primer delovanja tehnologije 19
3.1.1. Definicija oddaljenega vmesnika 19
3.1.2. Implementacija strežnika 20
3.1.3. Implementacija odjemalca 21
3.1.4. Potek povezovanja odjemalca s strežnikom 23
3.1.5. Prevajanje izvornih datotek 23
3.1.6. Zagon registra Java RMI, strežnika in odjemalca 24
4. Java program Vozlišče 27
4.1. Predstavitev programa Vozlišče 27
4.2. Programska arhitektura programa Vozlišče 27
4.2.1. Glavni strežnik (Server) 27

4.2.2. Glavni odjemalec (Client) 29
4.2.3. Seznam klientov sosedov (Neigbours) 30
4.2.4. Seznam klientov vozlišč (Nodes) 30
4.3. Implementacija 30
4.3.1. Nastavitve programa Vozlišče 30
4.3.2. Korensko vozlišče 32
4.3.3. Navadno vozlišče 34
4.3.4. Algoritem za izračun usmerjevalnih tabel 34
4.3.5. Algoritem za izris drevesa najkrajših poti 35
5. Zaključek 37
Slovar tujk 39
Dodatek 41
A: Izvorna koda 41
Kazalo slik 43
Kazalo tabel 45
Viri in literatura 47

Povzetek

Danes uporabljamo večjedrne procesorje in širokopasovni internet, ki nam je »skoraj« vedno na voljo. Da bi v današnjih časih čim bolj izkoristili vire, ki so nam na voljo, bi potrebovali porazdeljene algoritme. Porazdeljeni algoritmi se uporabljajo v sistemih kjer imamo na voljo več računalniških enot ali več procesorskih jeder in za izračun rešitve problema uporabimo vso razpoložljivo strojno opremo. Za dober izkoristek virov potrebujemo porazdeljeno programsko opremo, ki bi tekla na vseh procesnih enotah hkrati.

V diplomski nalogi sem rešil problem iskanja drevesa najkrajših poti v grafu s porazdeljenim algoritmom. Porazdeljeni algoritem je zasnovan tako, da vsakemu vozlišču grafa pripada njemu lastna procesna enota in da ima vsako vozlišče zgolj informacijo o svojih sosedih, ne pa tudi informacije o celotnem grafu. Tako sem procesne enote razbil na vozlišča. Iz pogleda na izvirni problem vidimo, da je graf sestavljen iz vozlišč in povezav. Napisal sem program Vozlišče, ki teče na eni procesni enoti in se poveže z drugimi Vozlišči, skupaj pa tvorijo mrežo Vozlišč, ki predstavlja graf v katerem iščemo drevo najkrajših poti. Za sporočanje med Vozlišči sem uporabil Javino RMI tehnologijo.

Ključne besede: porazdeljeni algoritem, porazdeljeni sistemi, drevo najkrajših poti, tehnologija Java RMI.

Abstract

Nowadays we use multi-core processors and broadband internet, which is usually available. If we want efficient use of available resources, we should use distributed algorithms. Distributed algorithms are used in systems with more computer units or more processor cores and for computing a result we use all available hardware. For efficient use of resources we need distributed software, which runs on all computers at the same time.

In thesis I had written and solved the problem of shortest paths tree with distributed algorithm. The distributed algorithm is designed in a way, that every graph node belong its own process unit. And every node has information only about its neighbours and not information about complete graph. In my program I have divided process units into nodes regarding to the main problem, that the graph includes nodes and edges. I have created program called Node. The program Node runs in his own process unit and connects to other Nodes. Nodes are connected into network. The network represents the graph in which we search the shortest paths tree. The technology that I used for messaging between the Nodes is the Java RMI.

Key words: distributed algorithm, distributed systems, shortest paths tree, technology Java RMI.

1. Uvod

Do sedaj je veljal Moorov zakon. Število tranzistorjev v eni centralno procesni enoti se je z leti podvajalo. Centralno procesne enote so postajale zmogljivejše na potrebo vse bolj zahtevnih opravil. Toda danes Moorov zakon ne velja več. Danes imamo CPE z več jedri. Za reševanje bolj zahtevnih opravil pa tudi več povezanih računalnikov. Zaradi tega imamo na voljo vedno več procesorske moči. Da bi to procesorsko moč sto odstotno izkoristiti, pa moramo narediti porazdeljeno programsko opremo. Taka programska oprema bo izkoristila vsa jedra procesorja ali več povezanih računalnikov za rešitev enega računskega problema. Porazdeljena programska oprema uporablja porazdeljene algoritme, ki bodo sočasno tekli na večih jedrih ali na večih računalnikih povezanih v omrežje.

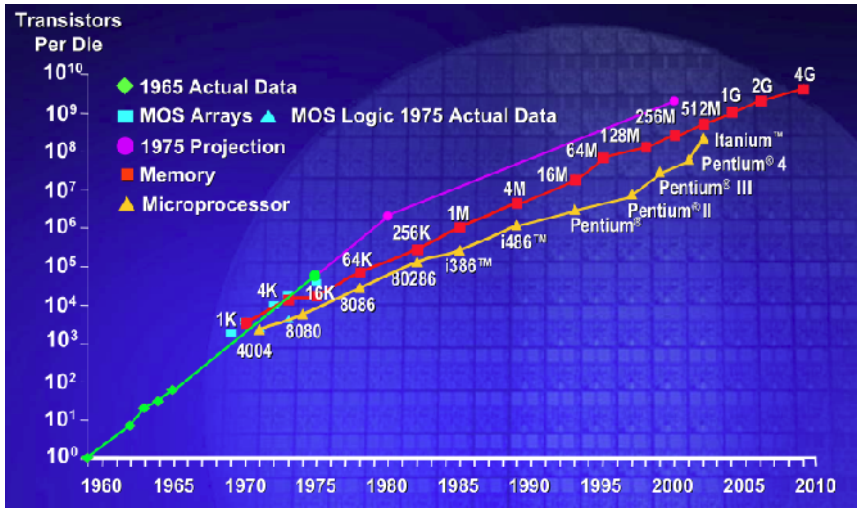
1.1. Moorov zakon

Dr. Gordon Earle Moore, raziskovalec in soustanovitelj podjetja Intel, je v reviji Electronics leta 1965 objavil članek Trpanje več komponent na integrirana vezja [1]. V članku je napovedal integrirana vezja visoke gostote, do 65000 tranzistorjev na eni sami silicijevi rezini. S tem je želel napovedati prihodnje trende integracije. Leta 1975 je v reviji Technical Digest objavil članek Napredek v digitalni elektroniki [2], v katerem je potrdil, da se razvoj integriranih vezji dogaja močno pospešeno glede na prejšnja leta.

Z leti razvoja integriranih vezji in računalniške strojne opreme se je Moorova napoved samo še potrjevala in izkazovala za resnično, zato so strokovnjaki na področju integriranih vezji označili to ugotovitev za Moorov zakon.

Moorov zakon pravi, da se približno vsake dve leti podvoji število integriranih elementov na enem integiranem vezju in s tem tudi podvoji zmogljivost visoko integriranih vezji, tudi centralno procesnih enot (CPE).

Dr. Moore je svoje ugotovitve še večkrat z navdušenjem povzel v kasnejših letih, leta 1995 ob izdaji centralno procesne enote Pentium I in leta 2004 ob začetku razvoja večjedrnih centralno procesnih enot. Do sedaj Moorov zakon velja že skoraj pol stoletja. Moorova krivulja iz leta 1975 prikazuje eksponentno rast števila itegriranih komponent na eni selicijevi rezini. Na sliki 1 so prikazani tudi podatki integriranosti tranzistorjev v centralno procesnih enotah in pomnilniških enotah vse od leta 1970 do leta 2010 [3]. Na sliki 1 so prikazani tudi podatki iz leta 1962, ko so bila logična vezja narejena iz MOS tranzistorjev. Tudi v tistih letih stopnja integracije komponent sovpada z Moorovo krivuljo.



Slika 1: Moorova krivulja.

V tabeli 1 vidimo število tranzistorjev posamezne centralno procesne enote od leta 1970 do leta 2010 [6]. Tabela je sortirana po letih in številu tranzistorjev v smeri naraščanja. V tabeli so navedene najbolj poznane CPE v svojem času. Poleg omenjenih podatkov o CPE je navedena tudi velikost najmanše integrirane komponente v proizvodnjem procesu CPE. Iz tabele je razvidno podvajanje števila tranzistojev približno vsake dve leti, predvsem v začetnih letih razvoja CPE. V nekaterih letih se je podvajanje dogajalo še celo hitreje kot v časovni dobi dveh let. V zadnjih letih pa vidimo, da se je podvajanje števila tranzistorjev ustavilo. Velika razlika med starejšimi in današnjimi CPE je, da so danes CPE večjederne in posledično tudi zmoglivejše. Kar pomeni, da pri istem številu tranzisotjev v CPE v starejši in današnji dobi imajo nove CPE več jeder in s tem tudi večjo procesorsko moč. CPE z več jedri omogočajo vzporedno procesiranje več opravil.

Tabela 1: Število tranzistorjev.

Ime centralno procesne enote	Število tranzistorjev	Leto predstavitve	Izdelovalec	Proizvodnji process
Intel 4004	2,300	1971	Intel	10 µm
Intel 8008	3,500	1972	Intel	10 µm
Intel 8080	4,500	1974	Intel	6 µm
MOS Technology 6502	3,510	1975	MOS Technology	ni podatka
Intel 8088	29,000	1979	Intel	3 µm
Intel 80286	134,000	1982	Intel	1.5 µm
Intel 80386	275,000	1985	Intel	1.5 µm
Intel 80486	1,180,000	1989	Intel	1 µm
Pentium	3,100,000	1993	Intel	0.8 µm
AMD K5	4,300,000	1996	AMD	0.5 µm
Pentium II	7,500,000	1997	Intel	0.35 µm
AMD K6	8,800,000	1997	AMD	0.35 µm
Pentium III	9,500,000	1999	Intel	0.25 µm
AMD K6-III	21,300,000	1999	AMD	0.25 µm
AMD K7	22,000,000	1999	AMD	0.25 µm
Pentium 4	42,000,000	2000	Intel	180 nm
Barton	54,300,000	2003	AMD	130 nm
AMD K8	105,900,000	2003	AMD	130 nm
Itanium 2	220,000,000	2003	Intel	130 nm
Itanium 2 with 9MB cache	592,000,000	2004	Intel	130 nm
Cell	241,000,000	2006	Sony/IBM/Toshiba	90 nm
Core 2 Duo	291,000,000	2006	Intel	65 nm
Dual-Core Itanium 2	1,700,000,000	2006	Intel	90 nm
AMD K10	463,000,000	2007	AMD	65 nm
POWER6	789,000,000	2007	IBM	65 nm
Atom	47,000,000	2008	Intel	45 nm
Core i7 (Quad)	731,000,000	2008	Intel	45 nm
AMD K10	758,000,000	2008	AMD	45 nm
Six-Core Xeon 7400	1,900,000,000	2008	Intel	45 nm
Six-Core Opteron 2400	904,000,000	2009	AMD	45 nm
Six-Core Core i7	1,170,000,000	2010	Intel	32 nm
POWER7	1,200,000,000	2010	IBM	45 nm
z196	1,400,000,000	2010	IBM	45 nm
Quad-Core Itanium Tukwila	2,000,000,000	2010	Intel	65 nm
8-Core Xeon Nehalem-EX	2,300,000,000	2010	Intel	45 nm

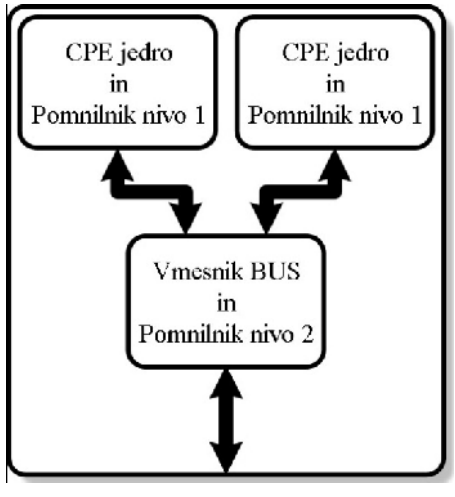
1.2. Večjedrni procesorji

Prvi večjedrni mikroprocesor je bil razvit v podjetju IBM leta 2001 [4]. Imel je osem neodvisnih izvajalnih enot: dve enoti za izvajanje operacij z plavajočo vejico, dve enoti za izvajanej operacij naloži-shrani, dve enoti za izvajanje operacij z fiksno vejico, eno enoto za vejitev in eno enoto za pogoj. Z tehnologijo vzporednih cevi je ta mikroprocesor lahko izvajal več operacij vzporedno v različnih operacijskih enotah.

Leta 2005 je prišla na trg prva večjedrna CPE AMD Athlon 64 X2 namenjena za namizne računalnike [5]. Primer take CPE je na sliki 2. Do takrat so CPE imele eno samo jedro. CPE so izvajale inštrukcije dolžine 32 bitov ali do 64 bitov. CPE je zaporedno prebrala in izvedla eno inštrukcijo iz pomnilne enote. To pomeni, da se je v eni časovni enoti izvedla ena sama inštrukcija. Inštrukcije so se dovolj hitro izvajale, da smo izkusili tekoče izvajanje ukazov v programu.

V sedanjem času je na enem samem vezju integriranih več jeder. Večjedrne CPE so razvite tako, da je možno izvajati več inštrukcij vzporedno in s tem tudi sočasno izvajati več niti v programu.

Vzporedno procesiranje idejno povzroči hitrejše izvajanje programa, ker ga poganja več »motorjev«, centralno procesnih enot ali jeder [8]. V splošni rabi je težko razdeliti program na tak način, da ga lahko izvaja več jeder brez oviranja eden drugega. Da bi dosegli pravo moč procesiranja, bi potrebovali porazdeljene algoritme.



Slika 2: Dvojedrna CPE.

1.3. Vzporedni računalniki

Z enojedrnimi računalniki je prav tako možno vzporedno procesiranje. Računalnike lahko povežemo med sabo v mrežo. Tak način procesiranja seveda zahteva zelo specifično programsko opremo imenovano porazdeljena programska oprema.

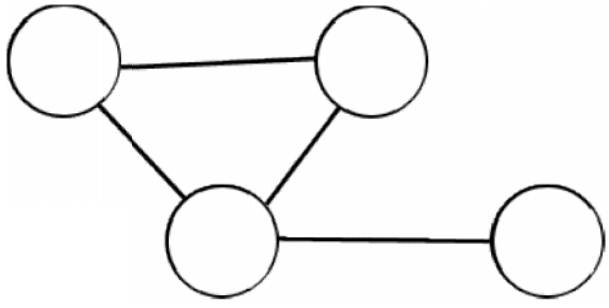
Izraza kot sta **vzporedno računanje** in **porazdeljeno računaje** imata veliko skupnih prekrivanj. Isti sistem ima lahko značilnosti kot so vzporeden in porazdeljen. Procesorske enote v tipičnem porazdeljenem sistemu tečejo vzporedno. Vendar je mogoče opredeliti trenuten sistem kot vzporeden ali porazdeljen glede na sledeča kriterija:

- V vzporednem računanju imamo sistem, ki je homogen, natančno določen in vedno v celoti dostopen.

- V porazdeljenem računanju imamo sistem, ki je heterogen, geografsko razpršen in nezanesljiv.

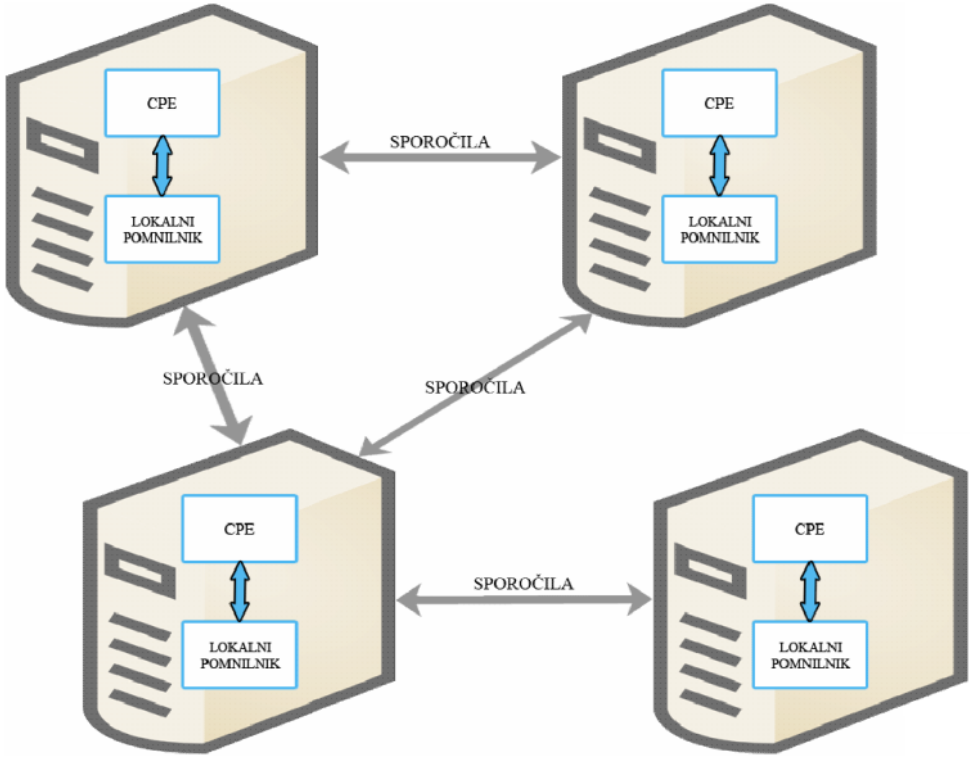
1.3.1. Sistem s porazdeljenim pomnilnikom

Slika 3 prikazuje porazdeljen sistem podan v glavnih potezah. Sistem je predstavljen kot graf, v katerem je vsako vozlišče računalnik in vsaka povezava ali črta med dvema vozliščema komunikacijska linija, ki lahko predstavlja tudi mrežni kabel.



Slika 3: Graf porazdeljenega sistema.

Slika 4 prikazuje povsem enak porazdeljen sistem z več podrobnosti. Vsak računalnik ima svoj lokalni pomnilnik in informacije se izmenjujejo izključno samo preko pošiljanja sporočil od enega vozlišča do drugega vozlišča skozi razpoložljive komunikacijske povezave.



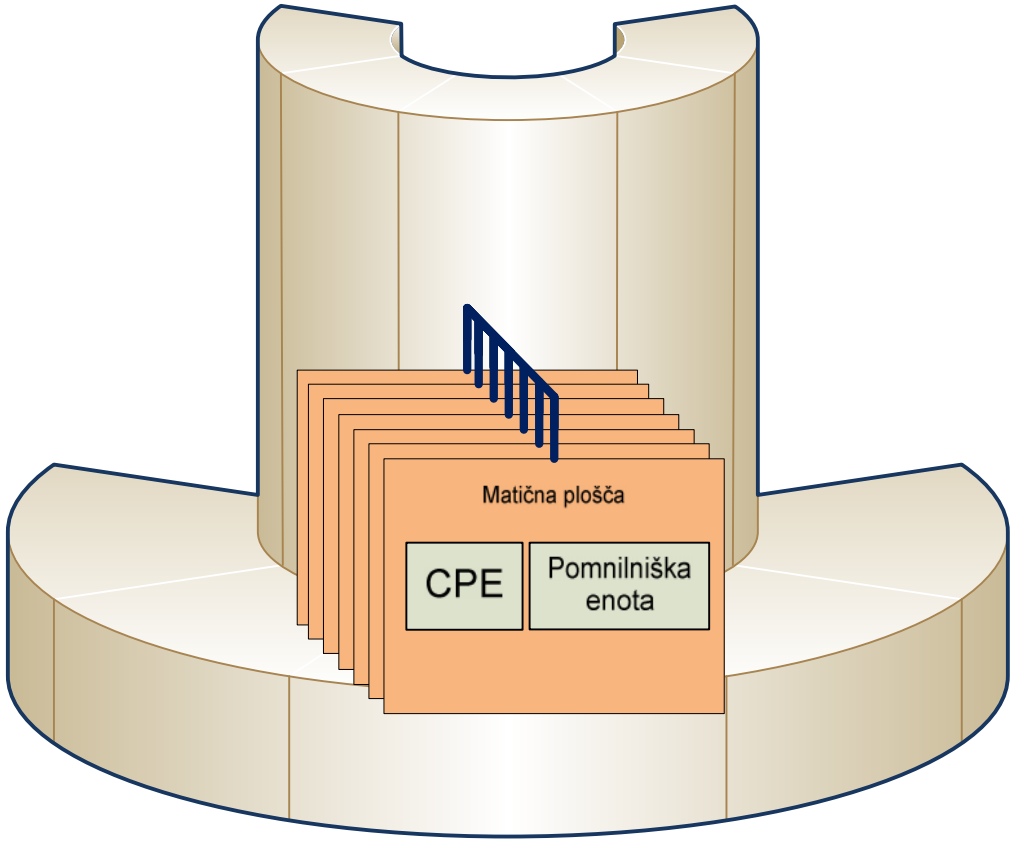
Slika 4: Podrobnosti v porazdeljenem sistemu.

Poznamo tudi vzporedne računalnike s porazdeljenim pomnilnikom. Danes so v takih sistemih tudi večjedrne centralno procesne enote. Matične plošče so med seboj povezane s

posebnimi hitrimi vodili. Super računalniki uporabljajo posebne tehnologije, ki več takih računalniških vezji povežejo v posebno omrežje, ki skupaj predstavlja super računalnik. Super računalniki so danes narejeni po meri, glede na zahteve uporabe takega sistema. Super računalnike v svetu uporabljajo za zahtevne izračune na naslednjih področjih: kvantna fizika, napovedovanje vremena, dinamika tekočin, aerodinamika, molekulsko modeliranje, fizikalne simulacije, itd.

1.3.2. Sistem s skupnim pomnilnikom

Slika 5 prikazuje vzporeden sistem s skupnim pomnilnikom v katerem ima vsak procesor neposreden dostop. Informacije med procesi se prenašajo izključno preko skupnega pomnilnika po posebnih vodilih. Centralno procesne enote so na svoji matični plošči, kjer imajo tudi svoj pomnilnik. Na eni matični plošči je ena ali več centralno procesnih enot. Prvi tak super računalnik je bil CRAY-1 leta 1975.



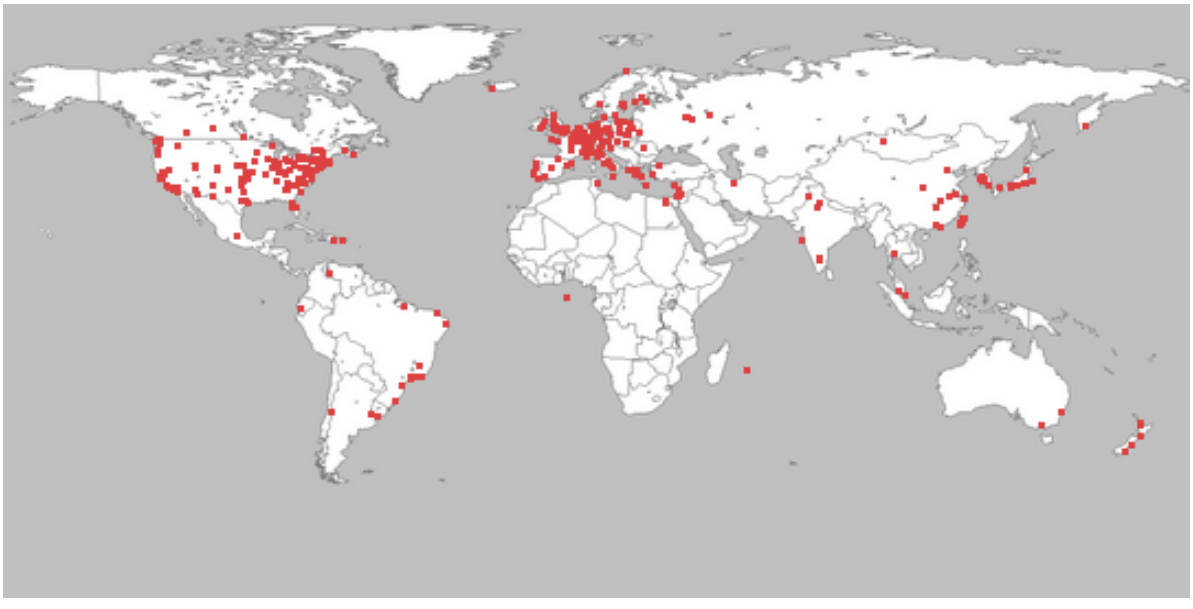
Slika 5: Vzporeden sistem.

1.4. Porazdeljeni sistemi

Porazdeljeni sistemi so sestavljeni iz večjega števila vozlišč povezanih v omrežje. Uporabljajo se za raziskovalna dela na področju računalniških omrežji in porazdeljenega računanja. Eden izmed njih je sistem PlanetLab. V sistemu PlanetLab je trenutno povezanih 1138 vozlišč iz 520 inštitucij [7]. Število vozlišč in inštitucij še vedno narašča. Vozlišča v takih sistemih predstavljajo strežniki. V sistemu PlanetLab sodelujejo strežniki iz

raziskovalnih centrov in univerz po celem svetu (Slika 6). PlanetLab je globalna raziskovalna mreža, ki podpira razvoj novih omrežnih servisov. V tej raziskovalni skupnosti sodeluje več kot 1000 raziskovalcev iz akademskih institucij in industrijskih raziskovalnih laboratorijev. Raziskovalci uporabljajo PlanetLab za razvoj novih tehnologij za porazdeljeno shranjevanje, omrežno mapiranje, peer-to-peer sisteme, porazdeljene hash tabele in procesiranje na zahtevo.

Porazdeljeni sistemi so v večini primerov sestavljeni iz računalnikov in strežnikov. Ker so cene takih komponent nekajkrat manjše od specializiranih komponent super računalnikov, je razmerje med ceno in zmogljivostjo pri porazdeljenih sistemih precej ugodnejše.



Slika 6: Porazdeljenost sistema PlanetLab.

Pri porazdeljenih sistemih se pojavijo tudi tako imenovani računalniški skupki (ang. clusters). Pri sami sestavi vozlišč so zelo podobni porazdeljenim sistemom, čeprav v večini primerov vozlišča predstavljajo računalniki in ne strežniki. Računalniški skupki so tudi manjši in ponavadi predstavljajo laboratorij ali učilnico z nekaj deset računalniki. V primeru računalniških skupkov so vozlišča zbrana na enem mestu. V primeru porazdeljenih sistemov pa se vozlišča nahajajo po celem svetu, kot je vidno na sliki 6.

1.5. Porazdeljeno procesiranje in programiranje

Porazdeljeno procesiranje je področje računalniške znanosti, ki preučuje porazdeljene sisteme. Porazdeljeni sistem vsebuje večje število avtonomnih računalnikov, ki so povezani v računalniško omrežje. Računalniki si sporočajo med seboj, da bi dosegli skupen cilj. Računalniški program, ki teče v porazdeljenem sistemu, se imenuje porazdeljeni program. Porazdeljeno programiranje pa je proces pisanja in razvijanja take vrste programa.

Porazdeljeno procesiranje se uporablja v porazdeljenih sistemih za reševanje računskih problemov. V porazdeljenem procesiranju je problem razdeljen na več opravil. Vsako opravilo reši ena enota. Ne obstaja ena sama definicija porazdeljenega sistema, se pa pogosto pojavljajo lastnosti:

- Več avtonomnih računskih enot, kjer ima vsaka enota svoj lokalni spomin.
- Enote med seboj komunicirajo s pošiljanjem sporočil.

V nadaljevanju bomo posamezno računsko enoto poimenovali kar vozlišče.

Še nekaj zelo pogostih lastnosti porazdeljenega sistema:

- Sistem mora znati tolerirati odpoved posameznega vozlišča.
- Struktura sistema ni poznana v naprej. Sistem je sestavljen iz različnih vozlišč, omrežnih povezav.
- Sistem se lahko spremeni med izvajanjem porazdeljenega računanja.
- Vsako vozlišče ima omejen in nepopoln pogled na sistem. Vsako vozlišče pozna samo nek del vhodnih podatkov.
- Vsako vozlišče ne pozna vseh vozlišč v sistemu ampak pozna samo svoje sosede.

Problemi in težave, ki se pojavljajo pri porazdeljenem programiranju, so:

- sinhronizacija med vozlišči,
- način in metode sporočanja med vozlišči,
- shranjevanje rezultatov v vozliščih in izpis končnega rezultata,
- obravnavanje odpovedi vozlišča,
- prilagoditev izvajanja porazdeljenega algoritma pri odpovedi vozlišča,
- zagon porazdeljenega programa v vozliščih,
- zaprtje porazdeljenega programa v vozliščih,
- zaznava končnega izračuna v vseh vozliščih.

Porazdeljen program lahko sprogramiramo v programskem jeziku C. Največkrat uporabljena in podprta tehnologija v tem programskem jeziku je vtič (ang. socket). Porazdeljen program lahko napišemo tudi v drugih programski jezikih. Nekateri programski jeziki omogočajo celo lažje porazdeljeno programiranje. Eden od takih je Java z tehnologijo RMI. Ta tehnologija poenostavi problem sporočanja med vozlišči. Omogoča nam oddaljen klic metode. Na prvi pogled zgleda kot spletni servis, v resnici pa je veliko bolj enostaven. Programski jezik Java nam omogoča tudi sinhronizacijo v programski kodi s sinhroniziranimi metodami in bloki kode.

Sporočanje med vozlišči je možno narediti tudi s spletnimi servisi. Spletni servis lahko razvijemo v programskem jeziku C ali Java. Razvoj spletnega servisa je zahtevno in časovno bolj potratno, kot uporaba tehnologije Java RMI. Pri spletnem servisu je potrebno vzdrževati tudi opisno datoteko WSDL. Ta datoteka opisuje spletni servis. Namenjena je odjemalcu in opisuje kako dostopamo do spletnega servisa, kako kličemo in kakšne odgovore nam vračajo metode. Vprašanja (ang. requests) in odgovori (ang. responses) med strežnikom in odjemalcem potekajo v XML obliki. Pri velikih odgovorih so potrebni zmogljivejši strežniki, ki dovolj hitro procesirajo velike XML odgovore. Navadni računalniki ne zadostijo tem zahtevam. Spletni servisi tečejo znotraj programske in strojne opreme, ki se imenuje aplikacijski strežnik. To pa je še dodatna zahteva, ki jo mora vsako vozlišče imeti. Poleg tega je večina dobrih aplikacijskih strežnikov plačljivih (IBM WebSphere).

Ker je že sam razvoj porazdeljene programske opreme dovolj zahteven in kompleksen, iščemo čim bolj preprosto tehnologijo sporočanja med vozlišči. Ker nam spletni servisi to omogočajo, še ne pomeni, da so najboljša rešitev. Java RMI tehnologija je preprosta in brezplačna rešitev. Poleg tega je z Java RMI tehnologijo lažje razvijati in testirati spletno storitev, kot pri spletnih servisih. Pri tehnologiji Java RMI je opisna datoteka spletnega servisa napisana v Java programskem jeziku. Opisno datoteko predstavlja Java vmesnik (ang. interface) po katerem implementiramo funkcionalnosti strežnika.

2. Porazdeljen algoritem za izračun drevesa najkrajših poti

2.1. Pregled

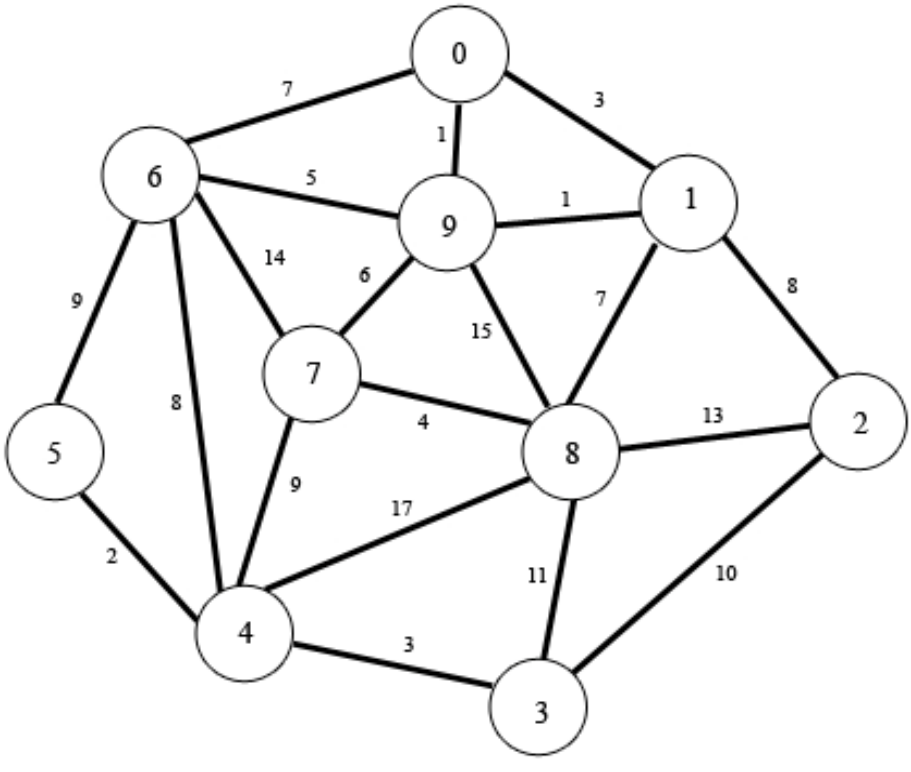
Problem, s katerim se bomo soočili, je preiskovanje grafa vozlišč. Vozlišča so povezana s povezavami. Povezave so različnih dolžin. Radi bi izračunali drevo najkrašiih poti v grafu.

V tem poglavju bomo pregledali klasične metode reševanja problema, ki so narejene za računalnike z enim jedrom. To so Dijkstrin, Bellman-Fordov, posplošeni Bellman-Fordov in Floyd-Warshallov algoritem.

Radi bi definirali problem in tudi rešitev na porazdeljen način. Torej pri reševanju problema bi uporabili več računalnikov in s tem tudi več CPE. V primeru, da uporabimo samo en računalnik, bi radi izkoristili vsa jedra procesorja. Vsako vozlišče bi predstavljalo en računalnik ali eno jedro računalnika.

2.2. Definicija problema drevesa najkrajših poti

Imamo poljubno število vozlišč. Vozliča so med seboj povezana in tvorijo povezan graf. Imamo poljubno število povezav med vozlišči. Povezave med vozlišči so dvosmerne ali enosmerne. Na ta način tvorimo graf sestavljen iz n vozlišč in m povezav G(n,m). Vozlišča so med seboj različno oddaljena. Vsaka povezava ima svojo dolžino poti. Eno vozlišče označimo za koren. Sedaj bi radi izračunali drevo najkrajših poti iz korena do vseh vozlišč v mreži. Primer takega grafa vozlišč je prikazan na spodnji sliki.



Slika 7: Primer grafa vozlišč.

Na opisanem grafu lahko definiramo množico vozlišč N in množico povezav M. V množici N je vozlišče nič označeno z zvezdico, ker predstavlja koren drevesa najkrajših poti. Za podani primer grafa je množica vozlišč:

$$N = \{0*,1,2,3,4,5,6,7,8,9\}.$$

Množice povezav je:

$$\begin{aligned} M = \{&< 0,1 >, < 0,6 >, < 0,9 >, < 1,2 >, \\ &< 1,8 >, < 1,9 >, < 2,3 >, < 2,8 >, \\ &< 3,4 >, < 3,8 >, < 4,5 >, < 4,6 >, \\ &< 4,7 >, < 4,8 >, < 5,6 >, < 6,7 >, \\ &< 6,9 >, < 7,8 >, < 7,9 >, < 8,9 >\}. \end{aligned}$$

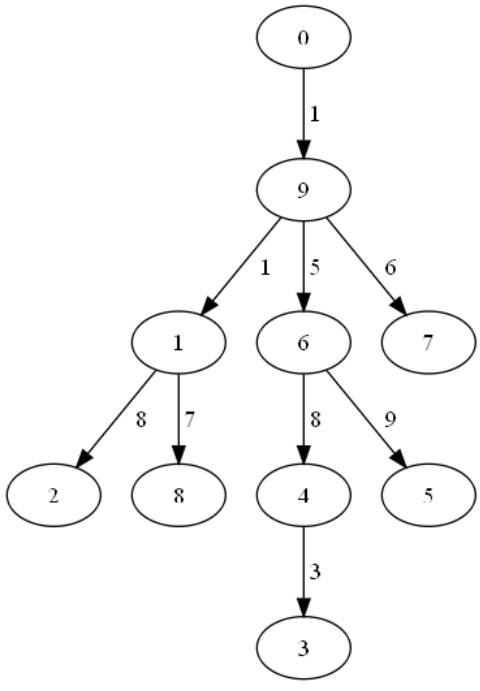
Ocene povezav v navedenem primeru so:

$$\begin{array}{llll} d(< 0,1 >) = 03, & d(< 0,6 >) = 07, & d(< 0,9 >) = 01, & d(< 1,2 >) = 08, \\ d(< 1,8 >) = 07, & d(< 1,9 >) = 01, & d(< 2,3 >) = 10, & d(< 2,8 >) = 13, \\ d(< 3,4 >) = 03, & d(< 3,8 >) = 11, & d(< 4,5 >) = 02, & d(< 4,6 >) = 08, \\ d(< 4,7 >) = 09, & d(< 4,8 >) = 17, & d(< 5,6 >) = 09, & d(< 6,7 >) = 14, \\ d(< 6,9 >) = 05, & d(< 7,8 >) = 04, & d(< 7,9 >) = 06, & d(< 8,9 >) = 15. \end{array}$$

Še boljši opis zgornjega grafa bi bil v simetrični matriki, katero je lažje sprogramirati in uporabiti v programu. Na podlagi zgornjega primera bi matrika v programskem jeziku Java izgledala takole:

```
public static final Integer matrika[][]  = {
                /*00  01  02  03  04  05  06  07  08  09*/
        /* 00 */{ 0,  3,  0,  0,  0,  0,  7,  0,  0,  1},
        /* 01 */{ 3,  0,  8,  0,  0,  0,  0,  0,  7,  1},
        /* 02 */{ 0,  8,  0, 10,  0,  0,  0,  0, 13,  0},
        /* 03 */{ 0,  0, 10,  0,  3,  0,  0,  0, 11,  0},
        /* 04 */{ 0,  0,  0,  3,  0,  2,  8,  9, 17,  0},
        /* 05 */{ 0,  0,  0,  0,  2,  0,  9,  0,  0,  0},
        /* 06 */{ 7,  0,  0,  0,  8,  9,  0, 14,  0,  5},
        /* 07 */{ 0,  0,  0,  0,  9,  0, 14,  0,  4,  6},
        /* 08 */{ 0,  7, 13, 11, 17,  0,  0,  4,  0, 15},
        /* 09 */{ 1,  1,  0,  0,  0,  0,  5,  6, 15,  0}
        };
```

Iz primera grafa vozlišč bi radi poiskali drevo najkraših poti v mreži. Pri tem je na korenu drevesa izbrano vozlišče iz množice vozlišč N, ki je označeno z zvezdico. V našem primeru je to vozlišče številka nič. Drevo najkrajših poti v grafu je dejansko vpet pod graf ali vpeto drevo z najkrajšimi povezavami. Drevo najkrajših poti iz primera grafa vozlišč je predstavljeno na sliki 8.



Slika 8: Primer drevesa najkrajših poti.

2.3. Metode reševanja problema na klasičen način

2.3.1. Dijkstrin algoritem

Primer klasičnega algoritma za reševanje problema najkrajših poti v grafu je **Dijkstrin algoritem**. Zasnoval ga je računalniški znanstvenik Edsger Dijkstra leta 1959. To je algoritem za preiskovanje grafa, ki reši problem iz enega vozlišča v vsa ostala vozlišča, v kolikor imajo vse povezave v grafu pozitivne ocene. Ta algoritem se pogosto uporablja v omrežnih usmerjevalnikih. Ekvivalenten algoritem je razvil Edward Forrest Moore leta 1957.

2.3.2. Bellman–Fordov algoritem

Bellman-Fordov algoritem reši problem najkrajše poti z enega vozlišča v vsa ostala vozlišča v usmerjenem grafu z predznačenimi ocenami povezav. Za reševanje problema v grafu brez negativnih ocen povezav je hitrejši Dijkstrin algoritem. Bellman-Fordov algoritem se primarno uporablja za grafe z negativnimi utežmi povezav. Algoritem se imenuje po razvijalcu Richardhu Bellmanu in razvijalcu Lesterju Fordu mlajšem.

Razširjena različica algoritma Bellman-Ford se uporablja za usmerjevalne protokole, na primer usmerjevlni informacijski algoritem RIP (Routing Information Protocol). Algoritem je porazdeljen, saj vključuje večje število vozlišč usmerjevalnikov v avtonomnem sistemu. To predstavlja zbirka omrežij IP navadno v lasti ISP (Internet Service Provider). Algoritem deluje po naslednjih korakih:

1. Vsako vozlišče izračuna razdaljo med njim in vsemi drugimi vozlišči in shrani informacije v obliki tabele.
2. Vsako vozlišče pošlje svojo tabelo vsem sosednjim vozliščem.
3. Ko vozlišče prejme tabelo razdalj od svojega soseda, izračuna najkrajše poti do vseh drugih vozlišč in posodobi svojo tabelo, da odraža kakršne koli spremembe.

Glavne slabosti algoritma Bellman-Ford v tem okviru so:

- Ne ocenjuje najboljše. Časovna kompleksnost je večja kot pri Dijkstrinem algoritmu.
- Spremembe v topologiji omrežja se ne odražajo hitro, saj so posodobitve širjenja od vozlišča do vozlišča.
- Štetje do neskončnosti. Če vozlišče postane nedosegljivo, nekatera vozlišča, ki imajo poti do njega preko drugih vozlišč, postopoma povečujejo svojo oceno razdalje. V tem času pride do usmerjenih zank.

2.4. Definicija reševanja na porazdeljen način

Uporabili bomo več posameznih enot. Vsako vozlišče bo svoja enota. Vsaka enota bo imela svoj lasten pomnilnik, torej bomo uporabili porazdeljen pomnilnik. Enote bodo med seboj komunicirale z sporočili. Ena procesna enota pozna samo svoje sosede – sosednja vozlišča. Uporabili bomo porazdeljen način računanja.

2.5. Algoritem poplavljanja

Poplavni algoritem je algoritem za porazdelitev informacije v vsako vozlišče povezano v grafu. Ime izvira iz koncepta širjenja - poplavljanja. Poplavni algoritem se uporablja v sistemih za delitev svojih datotek, kot so izvirni sistem Usenet, sistem Gnutella in drugi sistemi peer-to-peer. V svetu obstaja več izvedb tega algoritma. Algoritem v grobem deluje na način:

1. Vsako vozlišče deluje kot pošiljatelj in kot sprejemnik.
2. Vsako vozlišče posreduje vsako sporočilo vsem svojim sosedom razen izvornemu vozlišču.

Realni poplavni algoritmi morajo biti bolj kompleksni, kot je opisano zgoraj. Upoštevati morajo navodila, da se izognemo dostavljanju nepotrebnih podvojenih sporočil in izogibanju neskončnih zank. Pomembno je, da se sporočilo navsezadnje dokončno dostavi znotraj sistema in se konča njegova življenjska doba. Zelo pomembno je, da se poplavni algoritem uporablja tudi za reševanje matematičnih problemov vključno z veliko problemov v teoriji grafov.

2.5.1. Slabe lastnosti

Obstaja nekaj slabih lastnosti pri tem pristopu usmerjanja. Poplavni algoritem zelo obremeni pasovno širino omrežja. Ko moramo poslati sporočilo do končnega cilja, mora to sporočilo potovati po celotni poti do cilja. To poveča maksimalno breme nad omrežjem. Sporočilo lahko postane v omrežju tudi podvojeno. To še dodatno poveča breme pasovne širine omrežja in tudi poveča procesiranje v posameznih vozliščih. Vrsta poplavljanja imenovana selektivno poplavljanje naslovi te probleme z pošiljanjem sporočil usmerjevalnikom v smeri cilja.

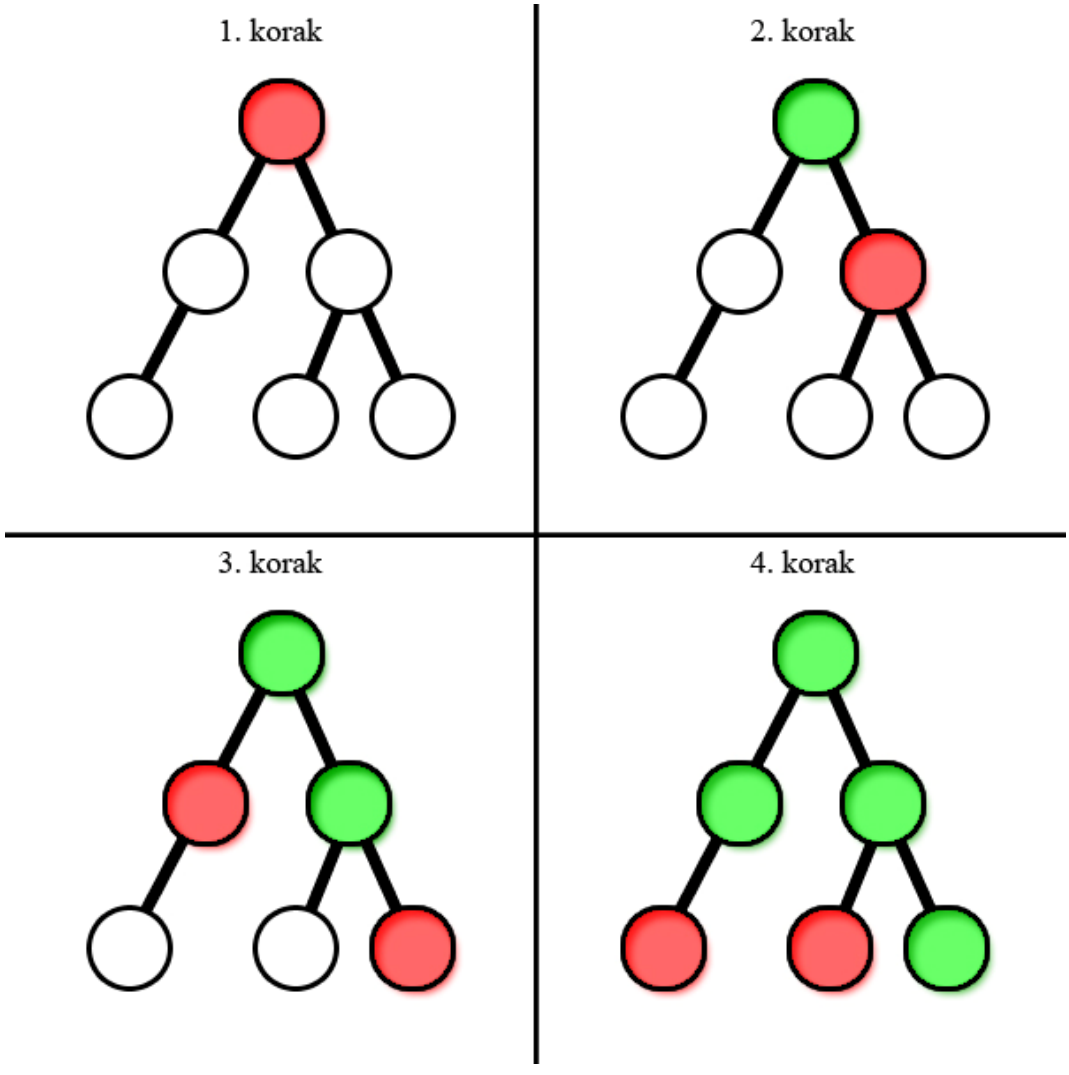
2.5.2. Dobre lastnosti

Glavna prednost algoritma poplavljanja je povečanje zanesljivosti usmerjanja. Ker se sporočilo prenese vsaj enkrat do vsakega vozlišča, je skoraj zagotovljeno, da bo doseglo svoj cilj. Poleg tega bo sporočilo doseglo svoj cilj po najkrajši možni poti.

2.5.3. Prikaz delovanja algoritma

Slika 9 prikazuje delovanje algoritma poplavljanja, ki je prikazan na drevesu vozlišč. V danem primeru algoritem poplavi vsa vozlišča v drevesu. Algoritem se začne v korenskem

vozlišču in se nadaljuje po vozliščih navzdol. Rdeča vozlišča so označena kot trenutno poplavljena. Zelena vozlišča so označena kot že poplavljena. Bela vozlišča so označena kot še ne poplavljena.



Slika 9: Delovanje algoritma poplavljanja.

2.6. Metoda reševanja na porazdeljen način z metodo poplavljanja

Predpogoj reševanja problema je, da vozlišča tvorijo povezan graf. Povezava med dvema povezanima vozliščema je dvosmerna. Po povezavi med dvema vozliščema je možno v obe smeri pošiljati sporočila. Vozlišča med seboj komunicirajo izključno z sporočili. Za rešitev problema uporabimo metodo poplavljanja. V grafu vozlišč je natanko eno vozlišče označeno kot koren drevesa najkrajših poti v grafu. V tem vozlišču se zažene poplavni algoritem. Algoritem iskanja drevesa najkrajših poti v grafu deluje na opisan način. V izbranem vozlišču imenovanem koren se izvrši klic algoritma za izračun drevesa. Trenutno vozlišče se vedno smatra kot otrok v relaciji oče – otrok.

Informacija o starših je zapisana kot množica vozlišč, razdalja pa kot število milisekund od neposrednega starša do njega. Če gre za vozlišče koren prejme v množici staršev samega sebe in kot zamudo programsko nastavljeno nič.

Algoritem se izvaja po naslednjih korakih:

1. Trenutno vozlišče preko parametrov prejme informacijo, kdo so njegovi straši in kolikšna je razdalja od njegovega neposrednega starša.
2. Trenutno vozlišče iterira preko vseh svojih sosedov. V vsaki iteraciji izvede naslednje korake:
   2.1. Preveri, če je sosed v množici staršev.
      2.1.1. Če je, se vrne v korak 2.
   2.2. Ustvari novo prazno množico vozlišč A.
   2.3. Doda soseda v množico A.
   2.4. Doda svoje starše v množico A.
   2.5. Poplavi soseda z informacijo o starših in z razdaljo od sebe do soseda.
   2.6. Sosed vrne tabelo izračunanih zamud od svojih poplavljenih vozlišč.
   2.7. Algoritem iterira skozi tabelo izračunanih zamud soseda:
      2.7.1. Če zamuda še ni v tabeli algoritma trenutnega vozlišča ali je zamuda strogo manjša:
         2.7.1.1. Zamudo doda v tabelo algoritma trenutnega vozlišča.
3. Če nismo na koncu seznama sosedov, se vrnemo v korak 2.
4. Zgrajeno tabelo razdalj shranimo v usmerjevalno tabelo trenutnega vozlišča.
   4.1. Vozlišče iterira čez zgrajeno tabelo razdalji:
      4.1.1. Če trenutno vozlišče ni enako vozlišču iz tabele razdalji (pogoj A) in če razdalja še ni v usmerjevalni tabeli (pogoj B) ali je razdalja vozlišča strogo manjša od razdalje v usmerjevalni tabeli (pogoj C), je pogoj izpolnjen (A in (B ali C)):
         4.1.1.1. Razdaljo vozlišča doda v usmerjevalno tabelo.
   4.2. Če ni na koncu seznama razdalji, se vrne v korak 4.1.
5. Vrne tabelo razdalj.

3. Java RMI programska tehnologija

Java programska tehnologija, ki jo programski jezik Java nudi programerjem za realizacijo oddaljenega klica metod, omogoča programerjem da ustvarijo porazdeljeno programsko opremo. Tehnologija Java RMI omogoča, da se metode oddaljenega objekta pokličejo (ang. invoke) iz drugega Java navideznega stroja (Java VM). Oddaljen klic metode je mogoče izvesti tudi iz drugega gostovanja (ang. host). Java RMI uporablja metodo serializacije za ureditev prenosa parametrov in podatkov metod pri tem pa ne okrne podatkovne tipe. Dejansko podpira resnično objektno orientirano porazdeljeno programiranje.

V nadaljevanju je opisan preprost program, s katerim je prikazan način programiranja z uporabo Javine tehnologije RMI.

3.1. Osnovni primer delovanja tehnologije

Opisani primer prikazuje osnovne korake, da ustvarimo porazdeljeno različico programa Pozdravljen svet z uporabo Java RMI tehnologije. Porazdeljen program Pozdravljen svet uporablja preprost odjemalec, ki naredi oddaljen klic metode na strežnik. Strežnik lahko teče na oddaljenem gostovanju.

V sledečih podpoglavjih je v korakih predstavljena in z primeri prikazana uporaba in izvedba preprostega porazdeljenega programa Pozdravljen svet. Opisano je kako se:

- napiše Server Interface,
- implementira Server Interface,
- ustvari Server,
- ustvari Client in
- uporabi register RMI (rmiregistry).

3.1.1. Definicija oddaljenega vmesnika

Oddaljen objekt je instanca razreda, ki implementira oddaljen vmesnik. Oddaljen vmesnik razširi (ang. extends) vmensik `java.rmi.Remote` in napove množico oddaljenih metod. Vsaka oddaljena metoda mora napovedati tudi `java.rmi.RemoteException`. Spodaj je napisana definicija vmesnika uporabljenega v programu Pozdravljen svet. Vmesnik napove samo eno metodo `sayHello`, ki njegovemu klicatelju vrača niz tipa `String`.

```
1 package example.hello;
2
3 import java.rmi.Remote;
4 import java.rmi.RemoteException;
5
6 public interface Hello extends Remote {
7     String sayHello() throws RemoteException;
8 }
9
```

Slika 10: Vmesnik Hello.

Oddaljen klic metode lahko odpove na veliko različnih načinov v primerjavi z lokalnim klicem metode. Take težave so lahko povezane z težavami omrežja in z težavami strežnika. Oddaljena metoda bo na take težave odgovorila z metanjem (ang. throws) izjeme `java.rmi.RemoteException`. To izjemo lahko ujamemo in jo na različne načine obravnavamo, glede na zahteve aplikacije.

3.1.2. Implementacija strežnika

Razred strežnik v tem kontekstu je razred, ki vsebuje tudi glavno metodo `main`. Glavna metoda ustvari instanco oddaljenega objekta, izvozi oddaljen objekt in potem veže (ang. bind) instanco objekta v Java RMI register (rmiregistry). Razred, ki vsebuje glavno metodo `main` je lahko tudi implementacijski razred sam ali pa povsem svoj razred. V tem primeru je glavna metoda `main` definirana v razredu `Server`, ki poleg tega tudi implementira oddaljen vmesnik `Hello`. Glavna metoda razreda `Server` naredi naslednje: ustvari in izvozi oddaljen objekt in registrira oddaljen objekt v registru Java RMI. Spodaj je primer razreda `Server`. Opis posameznega pomembnega dela kode je zapisan kot komentar.

```
1  package example.hello;
2
3  import java.rmi.registry.Registry;
4  import java.rmi.registry.LocateRegistry;
5  import java.rmi.server.UnicastRemoteObject;
6
7  public class Server implements Hello {
8
9      public Server() {}
10
11     public String sayHello() {
12         return "Hello, world!";
13     }
14
15     public static void main(String args[]) {
16         try {
17             Server obj = new Server();
18             //izvozi oddaljen objekt
19             Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
20             //veže oddaljen objekt v rmi register
21             Registry registry = LocateRegistry.getRegistry();
22             registry.bind("Hello", stub);
23             System.err.println("Server ready");
24         } catch (Exception e) {
25             System.err.println("Server exception: " + e.toString());
26             e.printStackTrace();
27         }
28     }
29 }
30
```

Slika 11: Razred Server.

Implementacijski razred `Server` implementira oddaljen vmesnik `Hello` s čimer proizvede implementacijo oddaljene metode `sayHello`. Pri metodi `sayHello` ni potrebno navesti da vrže izjemo, zato ker implementacija sama ne vrže nobene take izjeme.
Razred lahko definira tudi metodo, ki ni specificirana v oddaljenem vmesniku `Hello`. Take metode lahko kličemo samo znotraj Java virtualnega stroja v katerem teče servis in ne morejo biti klicane oddaljeno.

Ustvarjanje in izvoz oddaljenega objekta

Glavna metoda strežnika mora ustvariti oddaljen objekt, ki proizvede servis. Poleg tega, mora izvoziti oddaljen objekt, da med delovanjem lahko sprejema oddaljene klice. To se naredi tako:

```
17             Server obj = new Server();
18             //izvozi oddaljen objekt
19             Hello stub = (Hello) UnicastRemoteObject.exportObject(obj, 0);
```

Slika 12: Izvoz oddaljenega objekta.

Statična metoda `UnicastRemoteObject.exportObject` izvozi podan oddaljen objekt, da sprejema vhodne klice metod na anonimnem TCP/IP omrežnem naslovu in vratih in vrača ročico oddaljenega objekta, ki ga poda odjemalcem. Kot rezultat klica metode `exportObject`, med delovanjem začne poslušati na novem vtiču ali pa na deljenem vtiču strežnika, ki sprejema vhodne klice metod. Ročica, ki jo vrne, implementira isto množico metod od oddaljenega vmesnika, kot metode oddaljenega objekta in vsebuje tudi naslov in vrata gostovanja preko katerega se oddaljen objekt lahko poveže.

Registracija oddaljenega objekta v Java RMI register

Java RMI register za odjemalca (ang. client, peer, applet) omogoči klic metode oddaljenega objekta. Odjemalec mora najprej dobiti ročico od oddaljenega objekta. Za proces bootstrapping Java RMI uporablja register API, da se lahko aplikacije povežejo na oddaljen objekt glede na njihovo ime.

Register Java RMI je poenostavljeno ime za spletni servis, ki omogoča da odjemalec najde referenco (ročico) do oddaljenega objekta. Na splošno se register uporablja, da lociramo prvi oddaljen objekt, ki ga odjemalce potrebuje. Potem tipično ta objekt predstavlja aplikacijo, ki omogoča to vrsto podpore, da najdemo neke druge objekte. Primer tega je, da hranimo referenco kot parameter za ali vrednost, ki jo vrne od neke druge oddaljene metode. To se imenuje tudi tovarniški vzorec Java RMI (ang. factory pattern).

Ko je oddaljen objekt registriran na strežniku, ga klicatelji lahko najdejo po imenu objekta in tako pridobijo referenco objekta preko katere lahko kličejo oddaljene metode objekta.

Sledeča koda v navedenem primeru na strežniku ustvari ročico v registru na lokalnem naslovu in prevzetih vratih in nato uporablja ročico registra, da zveže ime »Hello« z ročico oddaljenega objekta v tem registru.

```
21            Registry registry = LocateRegistry.getRegistry();
22            registry.bind("Hello", stub);
```

Slika 13: Registracija objekta v registru RMI.

Statična metoda `LocateRegistry.getRegistry` se kliče brez parametrov in vrne ročico. Ročica implementira oddaljen vmesnik `java.rmi.registry.Registry` in pošlje vabila v register lokalnega strežnika preko prevzetih vrat registra 1099. Vezane metode so klicane preko ročice registra, če so v registru vezane na ročico oddaljenega objekta z imenom »Hello«.

Pomembno tukaj je, ko se kliče metoda `LocateRegistry.getRegistry`, da preprosto vrne primerno ročico za register. Klic ne preveri, če register resnično teče. Če noben register ne teče na TCP/IP vratih številka 1099 in naslovu »localhost«, ko se pokliče metoda `bind`, bo strežniku spodletelo in bo vrnil `RemoteException`.

3.1.3. Implementacija odjemalca

Odjemalec obdrži ročico za register na gostovanje strežnika, pogleda ročico oddaljenega objekta glede na ime in potem pokliče metodo `sayHello` oddaljenega objekta z uporabo ročice. Tukaj je glavni del izvorne kode odjemalca.

```
14          Registry registry = LocateRegistry.getRegistry(host);
15          Hello stub = (Hello) registry.lookup("Hello");
16          String response = stub.sayHello();
17          System.out.println("response: " + response);
```

Slika 14: Povezovanje odjemalca na strežnik.

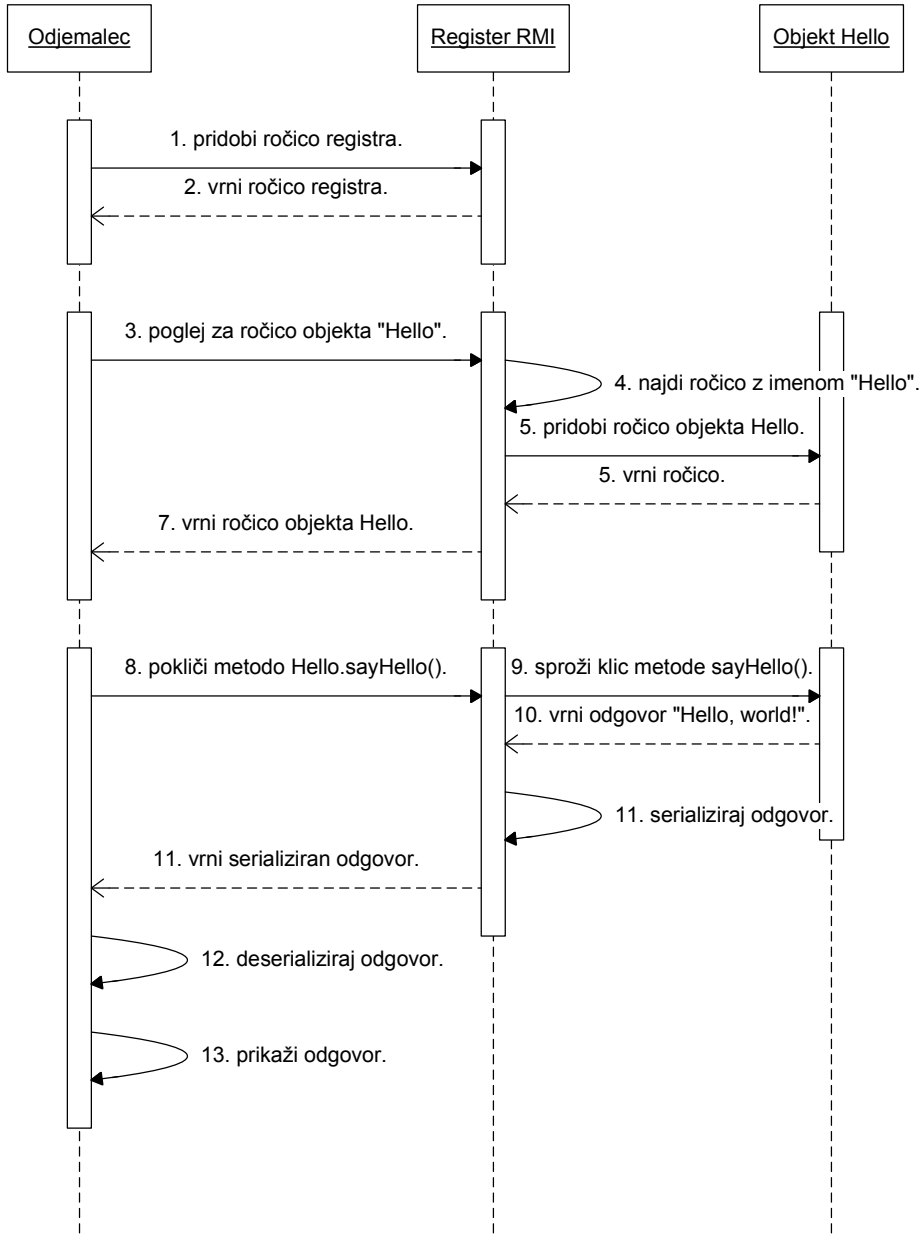
Odjemalec najprej pridobi ročico za register z klicem statične metode `LocateRegistry.getRegistry`. Metodo kliče z imenom gostovanja določenem v ukazni vrstici štirinajst. Če noben parameter ni določen, potem se uporabi null vrednost in na ta način zazna, da se uporabi prevzeto ime gostovanja, lokalni naslov »localhost«.

Nato odjemalec kliče oddaljeno metodo `lookup` z ročico registra, da pridobi ročico za oddaljeni objekt iz registra strežnika.

Končno odjemalec pokliče metodo `sayHello` z ročico oddaljenega objekta, ki povzroči da se zgodijo sledeče akcije:

1. Na strani odjemalca se med delovanjem odpre povezava na strežnik z uporabo naslova in vrat gostovanja. Odjemalec uporabi oddaljeno ročico, da pošlje serializirane podatke.
2. Na strani strežnika med delovanjem sprejema vhodne klice, deserializira klice in jih posreduje oddaljenemu objektu. Oddaljen objekt serializira odgovor. V tem primeru gre za niz znakov »Hello, world!« in jih pošlje odjemalcu.
3. Na strani odjemalca med delovanjem sprejme odgovor, ga deserializira in vrne rezultat klicatelju.

3.1.4. Potek povezovanja odjemalca s strežnikom



Slika 15: Potek klica oddaljene metode.

Celotni potek povezovanja odjemalca s strežnikom in klicanjem oddaljene metode je predstavljen v zgornjem sekvenčnem diagramu.

3.1.5. Prevajanje izvornih datotek

Izvorno kodo za zgornji primer se prevede z ukazom

```
javac –d <ciljni direktorij> Hello.java Server.java Client.java
```

kjer je **ciljni direktorij,** pot kamor se bodo shranile class datoteke. Poleg tega pa mora biti ta razred datoteka na voljo, da jo odjemalec lahko naloži k sebi. V našem primeru gre za datoteko `Server.class`.

3.1.6. Zagon registra Java RMI, strežnika in odjemalca

Da zaženemo opisani primer, moramo narediti naslednje:

- zagnati register Java RMI.
- zagnati strežik Server.
- zagnati odjemalca Client.

Zagon registra Java RMI

Za zagon registra poženemo metodo `rmiregistry` na strani strežnika. Ta ukaz ne proizvede nobenega izhoda, če je uspešno izveden. Tipično teče v ozadju in se ga požene v ukazni vrstici operacijskega sistema. Kako se to naredi je odvisno od operacijskega sistema (Solaris ali Windows).

Primer za Solaris operacijski sistem:

```
rmiregistry &
```

Primer za Windows operacijski sistem:

```
start rmiregistry
```

Prevzeto program rmiregistry teče na TCP/IP vratih številka 1099. Za zagon programa na različnih vratih, moramo podati številko vrat v ukazni vrstici. Za primer vzemimo register, ki bo tekel na vratih številka 2010 v Windows OS.

```
start rmiregistry 2010
```

Če bo register tekel na različni številki vrat kot so 1099, bomo morali specificirati številko vrat pri klicu metode `LocateRegistry.getRegistry` v razredu strežnika in odjemalca. Za primer, če register teče na vratih 2010, mora biti klic metode `getRegistry` takšen:

```
Registry registry = LocateRegistry.getRegistry(2010);
```

Zagon strežnika

Da zaženemo strežnik, moramo pognati razred `Server` z uporabo ukaza `java`.

Na operacijskem sistemu Solaris:

```
java -classpath classDir -Djava.rmi.server.codebase=file:classDir/
example.hello.Server &
```

Na operacijskem sistemu Windows:

```
start java -classpath classDir -Djava.rmi.server.codebase=file:classDir/
example.hello.Server
```

Tukaj nastavimo tudi `java.rmi.server.codebase` sistem lastnosti, ki zagotavlja da register lahko naloži definicijo oddaljenega vmesnika. Kjer je **classDir** korenski direktorij razrednega datotečnega drevesa.

Izhod iz razreda `Server` bi moral izgledati takole:

```
Server ready
```

Strežnik teče dokler proces ni ustavljen od uporabnika, tipično z ubijanjem procesa.

Zagon odjemalca

Ko je strežnik pripravljen, lahko poženemo odjemalca:

```
java –classpath classDir example.hello.Client
```

Kjer je **classDir** koren drevesa datotek razredov. Izhod odjemalca je sporočilo »Hello, world!«.

4. Java program Vozlišče

V tem poglavju je opisan porazdeljen program Vozlišče. Opisano je, kako deluje in kako se ga uporabi. Opisani so tudi glavni deli kode in glavni algoritem za izračun drevesa najkrajših poti v mreži vozlišč. Opisano je tudi kako se vzpostavi mrežo vozlišč. Poleg programa Vozlišče sem opisal tudi kako zaženem celotno mrežo vozlišč. Opisano je tudi kako izrišem sliko drevesa, kot rezultat algoritma najkrajših poti v mreži.

4.1. Predstavitev programa Vozlišče

Program Vozlišče je porazdeljena programska oprema. Porazdeljen program Vozlišče se povezuje z drugimi programi Vozlišče. Z povezanimi programi Vozlišče tvorimo mrežo, ki v teoriji grafov predstavlja utežen povezan graf. Prvo vozlišče v mreži vozlišč je korensko vozlišče. Vsa vozlišča, ki se vpnejo v graf, se najprej registrirajo v korenskem vozlišču, od koder pridobijo informacijo, kam se morajo povezati. Na ta način zna aplikacija Vozlišče generično ali konfigurabilno graditi graf vozlišč, ki ga nato poplavi z porazdeljenim algoritmom za izračun drevesa najkrajših poti. Aplikacija Vozlišče zna v tem grafu izračunati usmerjevalne tabele, ki predstavljajo informacijo drevesa najkrajših poti v grafu. Algoritem za izračun usmerjevalnih tabel uporablja algoritem selektivnega poplavljanja. S selektivnim algoritmom poplavljanja se po grafu širi informacija o razdaljah za izračun usmerjevalnih tabel najkrajših poti. Ko se algoritem konča, program Vozlišče tudi izriše drevo najkrajših poti v grafu.

4.2. Programska arhitektura programa Vozlišče

Aplikacija Vozlišče predstavlja program, ki teče v Javinem navideznem stroju (Java VM). Za delovanje uporablja tehnologijo RMI. Vsaka aplikacija Vozlišče uporablja svoj register RMI. Aplikacija vozlišče je sestavljena iz več instanc:

- glavnega programa `main`,
- glavnega strežnika, razred `Server`,
- glavnega odjemalca, razred `Client`,
- odjemalcev sosedov, seznam `Neigbours` in
- v korenskem Vozlišču tudi odjemalcev vozlišč, seznam `Nodes`.

Aplikacija vozlišče deluje v dveh načinih:

- korensko vozlišče ali
- navadno vozlišče.

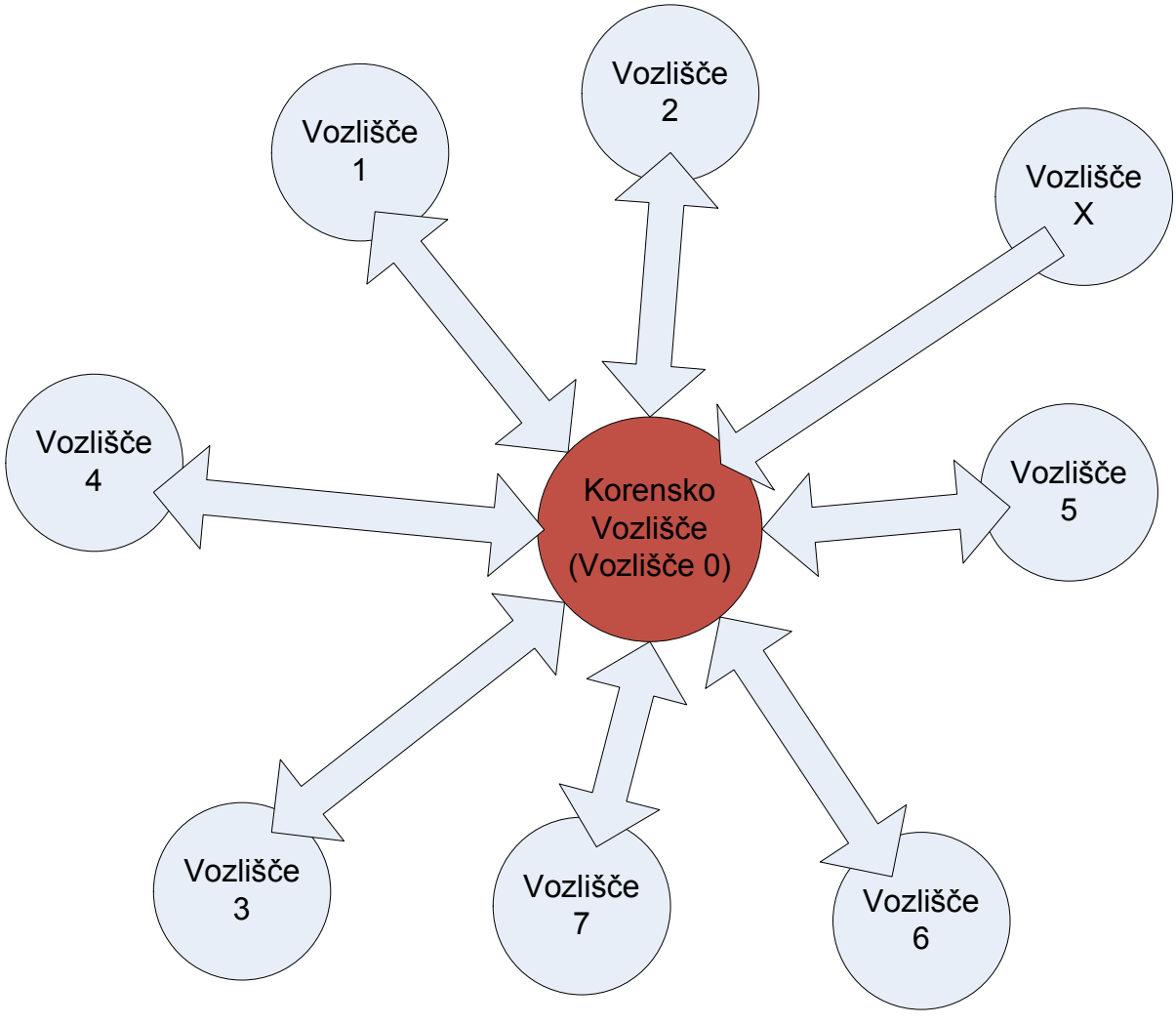
V vsaki mreži vozlišč je natanko eno korensko vozlišče in n navadnih vozlišč, pri čemer je n večje ali enako ena.

4.2.1. Glavni strežnik (Server)

Program Vozlišče ima svoj strežnik `Server`, na katerega se povežejo druge aplikacije Vozlišča. V načinu korensko vozlišče igra strežnik `Server` dve vlogi. Ena vloga je, da registrira nova vozlišča. Druga vloga je, da deluje kot navadno vozlišče in da se vpne v graf vozlišč.

Razred `Server` implementira `ServerInterface` in s tem nudi nabor metod, ki so vidne drugim programom Vozlišča. Drugi programi se povežejo na korenski strežnik preko TCP/IP omrežnega naslova in vrat. Ta podatek je shranjen v nastavitveni datoteki posameznega vozlišča. Vsako posamezno Vozlišče ima podatek, kje se nahaja strežnik korenskega vozlišča.

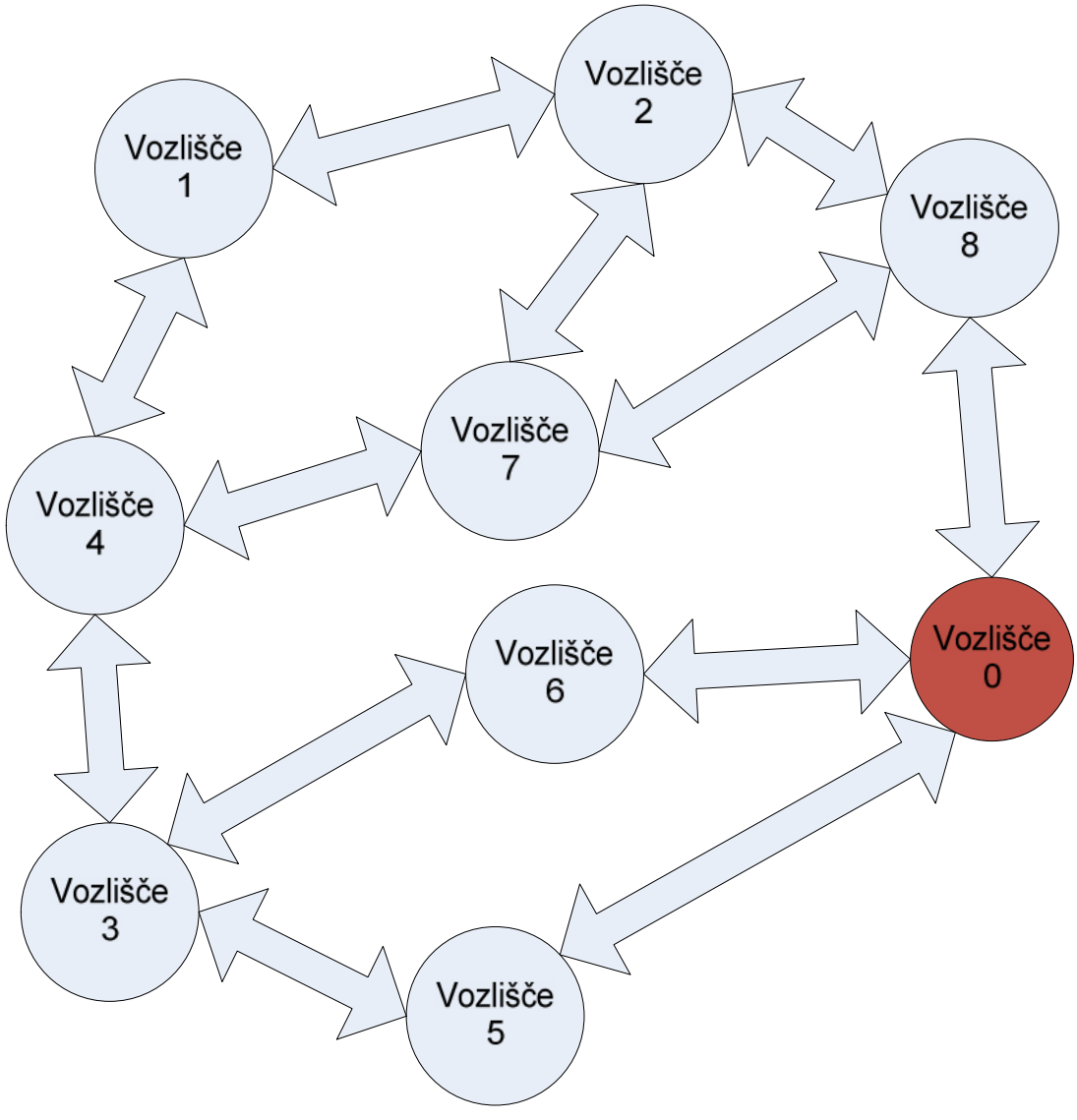
Glavni strežnik `Server` korenskega vozlišča sprejema nova vozlišča. Korensko vozlišče registrira nova vozlišča in jim dodeljuje indekse. Vsako novo vozlišče tudi shrani v listo vozlišč `Nodes` z dodeljenim indeksom. Korensko vozlišče za posamezno novo vozlišče ustvari in shrani odjemalca. Ko doseže nastavljeno maksimalno število vozlišč v grafu, preneha registrirati nova vozlišča in preneha sprejemati nova vozlišča. V tem trenutku uporabi generator naključnih grafov ali pa nastavljive grafe iz konfiguracije. Kaj bo korensko vozlišče naredilo, je odvisno od tega, kaj je nastavljeno v konfiguraciji. Ne glede na to kakšno mrežo vozlišč bo sestavilo, naredi naslednje. Z uporabo odjemalcev, shranjenih v listi vozlišč `Nodes`, posreduje vsakemu vozlišču informacijo o njegovih sosedih. Na ta način vsako vozlišče prejme informacijo o svojih sosedih. Nato se avtomatsko ustvari mreža vozlišč, ki predstavlja naš preiskovalni graf. Pomembno tukaj je še omeniti, da se Vozlišča med seboj predstavijo, identificirajo in sporazumejo s svojo indeksno številko.



Slika 16: Registracija novih vozlišč v korenskem vozlišču.

Vidimo da je začetna vloga strežnika, da registrira nova vozlišča. Prav tako se korensko vozlišče vpne v graf vozlišč, kot je tudi razvidno v spodnjem primeru.

V običajnem načinu tudi korenski strežnik deluje kot Vozlišče na katerega se povežejo Vozlišča sosedi. Tudi korensko vozlišče je v mreži vozlišč predstavljeno kot sosed.



Slika 17: Vozlišča povezana v mrežo vozlišč.

4.2.2. Glavni odjemalec (Client)

Vsako Vozlišče ima glavnega odjemalca. Glavni odjemalec se ob zagonu poveže na korenski strežnik. Njegova naloga je, da se predstavi in registrira v korenskemu strežniku. Ob registraciji, strežniku posreduje registracijske podatke. Registracijski podatki so TCP/IP omrežni naslov in vrata svojega strežnika. Nato korensko vozlišče vzpostavi odjemalca na naslov glavnega strežnika. Ob uspešni vzpostavitvi odjemalca mu posreduje registracijsko indeksno številko.

4.2.3. Seznam klientov sosedov (Neigbours)

Vsako Vozlišče, ki je vpeto v mrežo vozlišč, ima seznam svojih sosedov. To je seznam `Neigbours`. V vsakem objekt hrani podatke o sosedu. V objektu je shranjena tudi instanca klienta za soseda. Tako je vozlišče povezano z sosedi.

4.2.4. Seznam klientov vozlišč (Nodes)

Korensko vozlišče poleg ostalih instanc Vozlišča uporablja tudi instanco `Nodes`. Pri sebi hrani registracijske podatke klientov vseh vozlišč v mreži. Ti podatki so shranjeni v seznamu objektov `Node`. Ta seznam se potrebuje za gradnjo mreže vozlišč.

4.3. Implementacija

4.3.1. Nastavitve programa Vozlišče

Kot vsak drug program tudi program Vozlišče uporablja nastavitve. V programu Vozlišče imamo več sklopov nastavitev. Prvi sklop nastavitev je namenjen korenskemu strežniku, ki teče v registru RMI. Pomembni nastavitvi v tem delu sta TCP/IP omrežni naslov in vrata gostovanja korenskega strežnika. Poleg tega je nastavljen tudi indeks korenskega vozlišča in nekaj dodatnih nastavitev za izpis v dnevnik.

Za nastavitve se v večini primerov uporablja dva načina. Prvi najbolj pogosto uporabljen način je uporaba statičnih razredov. Posamezna nastavitev je konstanta, ki je vidna na ven in se jo ne da spreminjati. Drugi najbolj pogosto uporabljeni način je uporaba datotek property, katero nato beremo preko Java Property API vmesnika. Jaz sem se odločil za uporabo prvega načina, ker predstavlja preprosto in hitro uporabo nastavitev, ki se ne spreminjajo pogosto.

```
 8    /**
 9     * Root server RMI access information
10     */
11    public static final String ROOT_SERVER_HOST = "127.0.0.1";
12    public static final int ROOT_SERVER_PORT = 2010;
13    public static final int ROOT_SERVER_INDEX = 0;
14    public static final String NODE_SNAME = "NodeX";
15    public static final String NODE_CNAME = "ClientX";
16
```

Slika 18: Nastavitve korenskega strežnika.

Za tem sledijo sistemske nastavitve Javinega navideznega stroja. Nastavi se lastnost `java.rmi.server.codebase`, ki lokalnemu registru RMI pove, kje so dostopne Javine class datoteke.

```
17    /**
18     * System VM setting
19     */
20    public static final String NODE_SYSTEM_PROP_KEY = "java.rmi.server.codebase";
21    public static final String NODE_SYSTEM_PROP_VAL = "file:///"+"d:\\Miha\\diplomska\\java\\Diplomska\\bin\\";
22
```

Slika 19: Nastavitve Java navideznega stroja.

Tretji sklop nastavitev se navezuje na revizijsko sled. Nastavitve omogočajo vklop ali izklop zapisovanja revizijske sledi. Zapisovanje se nastavi na različnih nivojih, odvisno od potreb razvoja, testiranja in uporabe programa. Nivoji se navezujejo na izpisovanje:

- glavnih akcij v programu,
- manjših akcij v programu,
- akcije povezovanja programov Vozlišč v mrežo in
- izpisovanja izjem.

```
23    /**
24     * Log message settings
25     */
26    public static final boolean LOG_TITLES = true;
27    public static final boolean LOG_MESSAGES = true;
28    public static final boolean LOG_SAY_HELLO = true;
29    /**
30     * Print exceptions
31     */
32    public static final boolean EXCEPTIONS = true;
33
```

Slika 20: Nastavitve revizijske sledi.

Četrti sklop nastavitev se navezuje na nastavitev korenskega vozlišča in navadnega vozlišča. Korensko vozlišče ima nastavitve:

- testni primer,
- število vozlišč ki jih korensko vozlišče pričakuje v mreži in
- nastavitev mreže.

Navadno vozlišče ima nastavitve kot so:

- širina zaslona in
- število iteracij ping metode.

Širina zaslona je mišljene predvsem za lepši izpis sporočil na zaslon. Število iteracij ping metode se uporabi za izračun povprečne zakasnitve med dvema vozliščema.

```
34    /**
35     * Root node settings
36     */
37    public static final char TEST_CASE = 'b';
38    public static int HOW_MANY = Configuration.configureHOWMANY();
39    public static final Integer NETWORK[][] = Configuration.configureNETWORK();
40    /**
41     * Node settings
42     */
43    public static final int SCREEN_WIDTH = 120;
44    public static final int PING_N_ITERATIONS = 100;
```

Slika 21: Nastavitve korenskega vozlišča in navadnega vozlišča.

Zadnji sklop nastavitev je namenjen izrisovanju mreže vozlišč in drevesa najkrajših poti, ki jih izriše korensko vozlišče. To se naredi z uporabo zunanje programske opreme GraphViz. Program GraphViz uporablja datoteke dot za zapis grafa v tekstovni obliki, ki ga nato izriše v sliko PNG. Sama izvedba izrisa in prikazovanja grafa pa je v aplikaciji vozlišče narejeno z pisanjem v dot datoteko in izvajanjem bat datoteke.

```
46    /**
47     * Dot graph file path
48     */
49    public static final String DOT_MREZA = "c:\\work\\dot\\mreza.dot";
50    public static final String BAT_MREZA = "c:\\work\\dot_mreza.bat";
51    public static final String DOT_DREVO = "c:\\work\\dot\\drevo.dot";
52    public static final String BAT_DREVO = "c:\\work\\dot_drevo.bat";
```

Slika 22: Nastavitve izrisa drevesa in mreže vozlišč.

4.3.2. Korensko vozlišče

Postavitev korenskega strežnika v register RMI

Prvi korak pri postavitvi korenskega vozlišča je, da veže oddaljeni objekt `Server` z registrom RMI. Tako drugim aplikacijam Vozlišče omogoči dostop do korenskega strežnika preko TCP/IP omrežnega naslova in vrat. Korensko vozlišče ima glavni strežnik, ki je dostopen v registru RMI.

Sprejemanje in registracija klientov v korenskem strežniku

V naslednjem koraku korensko Vozlišče čaka in sprejema nove povezave v korenskem strežniku. Navadna Vozlišča se povežejo na korenski strežnik in se v njem registrirajo. Glavni strežnik nudi nabor metod za registracijo in vozlišča dodaja v listo vozlišč `Nodes`. Nato ciklično preverja, če so se nanj povezala še kakšna nova vozlišča. Ko to ugotovi gre skozi listo novih vozlišč. Ob vsakem dodajanju vzpostavi klienta na strežnik novega vozlišča. Pred dodajanjem novega vozlišča vedno preveri, če se je nanj že povezalo pričakovano število vozlišč v mreži. V tem primeru ne sprejema več novih vozlišč.

Povezovanje vozlišč v mrežo

Ko korensko vozlišče ugotovi, da se je nanj že povezalo pričakovano število vozlišč in je vzpostavil kliente naredi naslednje. Prekine zanko čakanja in nato vsakemu klientu pošlje serializirano listo njegovih sosedov. Kot podatek kdo so njegovi sosedi posreduje TCP/IP omrežni naslov in vrata gostovanja strežnika ter indeks vozlišča s katerim se mora povezati. Te podatke pridobi iz liste vozlišč. Sosede pa določi na podlagi matrike iz konfiguracije, ki opisuje mrežo vozlišč.

Preverjanje povezav v mreži

Ko korensko vozlišče posreduje vse podatke vsem vozliščem, nekaj časa še počaka. Čas čakanja se izračuna glede na število vozlišč v mreži. Ko ta čas poteče pri sebi sproži obojestransko preverjanje povezanosti. V drugih vozliščih prav tako po določenem času poteka preverjanje povezanosti z sosedi.

Vzpostavitev in preverjanje povezav do vseh svojih sosedov

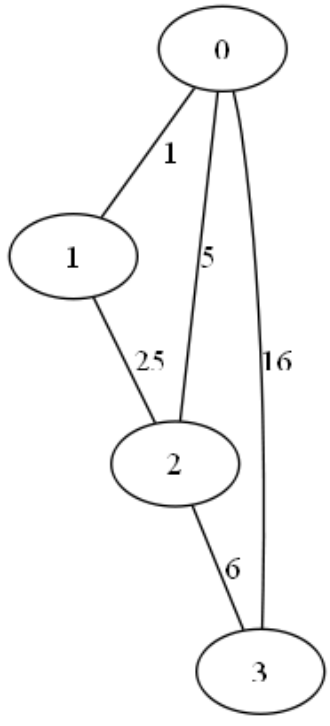
Ko ostala vozlišča sprejmejo podatke o svojih sosedih vzpostavijo svojega klienta na vsak sosedov strežnik. Ko je vzpostavljanje klientov končano nekaj časa še počakajo. Glede na število sosedov s katerimi so povezani. Po preteklem času pa vsako vozlišče preveri obojestransko povezanost s svojimi sosedi.

Proženje poplavnega algoritma

Ko korensko vozlišče ugotovi, da je uspešno povezano z svojimi sosedi, počaka da se omrežje »ustali« in se ostala vozlišča med seboj uspešno povežejo. Nato vpraša vsa vozlišča v mreži, če so uspešno povezana. V primeru pozitivnega odgovora na svojem strežniku sproži poplavni algoritem za izračun usmerjevalnih tabel. Strežnik iterira čez svoje sosede kliente in proži poplavni algoritem na strežnikih sosedov. Ti sosedi potem spet naprej prožijo algoritem, tako da se na koncu selektivno poplavijo vse veje drevesa vseh možnih poti v mreži. Pri tem pa se izračunavajo in shranjujejo najboljše ocene poti v posameznem vozlišču. Ko se poplavljanje konča, v usmerjevalnih tabelah ostanejo samo še najboljše ocene.

Izris mreže vozlišč

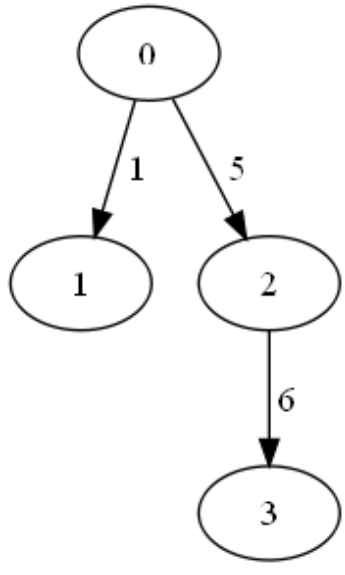
Ko vozlišče čaka na proženje poplavnega algoritma izriše mrežo vozlišč iz opisane matrike. To naredi s pomočjo bat skript in uporabo zunanjega programa GraphViz. Izris mreže zgleda kot na sliki 23.



Slika 23: Primer mreže vozlišč.

Izris drevesa najkrajših poti v grafu

Ko se ocenjevanje konča in korensko vozlišče prejme oceno najboljših poti v mreži, izriše drevo najkrajših poti. Za izris mora sprožiti algoritem, ki selektivno potuje po najkrajših poteh do vseh vozlišč in zgradi strukturo drevesa najkrajših poti. Pri selektivnem potovanju uporablja usmerjevalne tabele z najboljšimi ocenami. To strukturo nato zapiše v tekstovni obliki v datoteko dot. Iz datoteke dot zunanji program GraphViz izriše drevo najkrajših poti v mreži iz korenskega vozlišča. Iz primera mreže vozlišč (slika 23) se izriše drevo najkrajših poti v mreži vozlišč (slika 24).



Slika 24: Primer drevesa najkrajših poti v mreži vozlišč.

4.3.3. Navadno vozlišče

Povezovanje na korenski strežnik

Navadno vozlišče se najprej poveže na korenski strežnik. To naredi tako, da ustvari klienta in se poveže na TCP/IP omrežni naslov in vrata korenskega strežnika. Ko vzpostavi povezavo, se predstavi korenskemu vozlišču. To naredi tako da mu posreduje podatke svojega strežnika TCP/IP omrežni naslov in vrata. Korenski strežnik pa mu vrne indeks.

Postavitev strežnika v register RMI

Naslednji korak v izvajanju navadne vozlišča je postavitev svojega strežnika. Ob zagonu vozlišča aplikacija preko parametrov programa prejme podatke na katerih TCP/IP omrežni naslovu in vratih ima pripravljen register RMI. Te podatke v zgornjem koraku tudi posreduje korenskemu vozlišču. V tem koraku pa navadno vozlišče veže svoj strežnik v ta register RMI, da bo lahko dostopen drugim vozliščem.

Navadno vozlišče nato pozdravi korenski strežnik z funkcijo `childSayHello()` in mu tako sporoči, da strežnik navadnega vozlišča že teče v navedenem registru RMI.

Verifikacija povezav v mreži

Vozlišče čaka da bo povezano z vsemi sosedi. Vsako sekundo preveri, če je že povezan z vsemi vozlišči sosedi in če so že vsa vozlišča sosedje povezani z njo. Če je odgovor negativen, ponovna čaka eno sekundo. Ko je povezan z vsemi vozlišči je njegovo glavno delo končano. Glavno vlogo igra samo še strežnik vozlišča, ki čaka na proženje metod.

4.3.4. Algoritem za izračun usmerjevalnih tabel

Poplavni algoritem se prične izvajati, ko ima korensko vozlišče zadostno število vozlišč katera so med seboj povezana v mrežo. Če je pogoj zadoščen, sproži metodo `server.flood` , ki poplavi vozlišča. Pri poplavljanju vedno naprej poplavi svoj strežnik, kateremu posreduje podatke, da je on sam njegov starš in da je zakasnitev do njega nična. Strežnik nato pogleda kdo so njegovi sosedi.

Strežnik gre iterativno čez svoje sosede. Za vsakega soseda naredi naslednje. Pogleda če je sosed v množici staršev in če je ga ne poplavi. V nasprotnem primeru doda soseda v množico starše. Z klientom na strežnik soseda kliče metodo `server.flood` in tako poplavi tega soseda. Pri klicu posreduje tudi zakasnitev do njega. Kot vidimo se klicanje nadaljuje rekurzivno dokler se ne poplavijo vse možne veje v mreži vozlišč. Klic nam vrne tabelo ocen poti do vozlišč.

Poti vozlišč, ki nam jih vrne sosed preverimo v naši tabeli ocen. Če te ocene poti še nimamo v naši tabeli jo dodamo. Če to pot že imamo dodano in je ocena soseda boljša od naše trenutne ocene, v tem primeru to oceno izboljšamo. Ko so poplavljeni vsi sosedi imamo v usmerjevalni tabeli najboljše ocene poti.

Najboljše ocene poti se seveda shranijo v strežniku. Shranjene so v usmerjevalni tabeli.

```
171     public HashMap<Integer, RoutingObject> flood(Integer[] parents, int delay)
172             throws RemoteException {
173         // dodam svoj delay
174         HashMap<Integer, RoutingObject> delays = new HashMap<Integer, RoutingObject>();
175         delays.put(this.index, new RoutingObject(this.index, delay));
176         // sortiramo starše za binarno iskanje
177         Arrays.sort(parents);
178         // poplavim vse sosede
179         for (int i = 0; i < neighbours.length; i++) {
180             // preverim če sosed ni med starši
181             if (Arrays.binarySearch(parents, neighbours[i].getIndex()) < 0) {
182                 // dodamo soseda za starša
183                 Integer[] childParents = Arrays.copyOf(parents,parents.length + 1);
184                 childParents[childParents.length - 1] = neighbours[i].getIndex();
185                 // poplavim soseda
186                 HashMap<Integer, RoutingObject> nDelays = neighbours[i].flood(childParents);
187                 // preverim poti soseda
188                 for (Iterator<Integer> it = nDelays.keySet().iterator(); it .hasNext();) {
189                     Integer nKey = it.next();
190                     RoutingObject nVal = nDelays.get(nKey);
191                     RoutingObject dVal = delays.get(nKey);
192                     // editiram delays
193                     if (dVal == null || dVal.delay > nVal.delay + delay) {
194                         delays.put(nKey,
195                                 new RoutingObject(neighbours[i].getIndex(),
196                                 nVal.delay + delay));
197                     }
198                 }
199             }
200         }
201         // shranim ali prepišem najboljše poti v routing tabeli in
202         this.saveRouting(delays);
203         // vrnem najboljše poti
204         return delays;
205     }
206
207
```

Slika 25: Poplavni algoritem za izračun usmerjevalnih tabel.

4.3.5. Algoritem za izris drevesa najkrajših poti

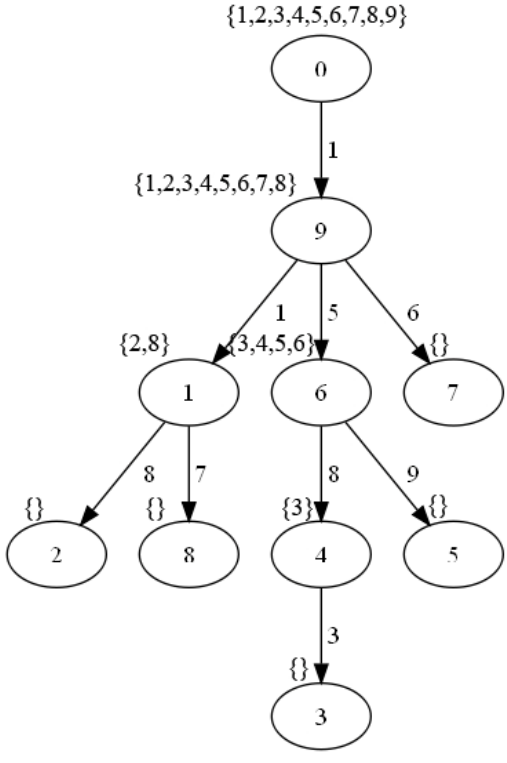
Algoritem se začne izvajati, ko se konča izvajanje poplavnega algoritma. Algoritem kot parameter dobi katera vozlišča mora obiskati. Podobno kot prejšnji algoritem, je tudi ta algoritem poplavni. Z razliko od prejšnjega ta potuje po mreži vozlišč na selektivni način. V vsakem vozlišču pogleda katere poti mora še obiskati in izbere najboljše poti iz usmerjevalne tabele.

Pokliče se metoda `server.floogGraph`. Vozlišče pripravi strukturo NodeObject. Ta struktura predstavlja drevesno strukturo z n vejami. Maksimalno število vej, ki jih ima vozlišče je omejeno z številom sosedov, ki jih ima vozlišče. Vozlišče doda sebe v množico staršev. Pogleda katera vozlišča mora še vse obiskati. Če je seznam teh vozlišč enak nič se algoritem konča in vrne strukturo v kateri zapiše samo sebe, brez svojih otrok. V nasprotnem primeru poišče vse svoje otroke. Iterativno gre čez seznam teh vozlišč in poplavi vozlišča z seznama vozlišč, ki jih v tej veji moramo še obiskati. Seznam, ki se izračuna je na spodnjem drevesu napisan v zavitih oklepajih.

```
226     public NodeObject floodGraph(int level, int parent, int delay,
227             HashSet<Integer> parents, HashSet<Integer> paths)
228             throws RemoteException {
229         NodeObject[] childs = new NodeObject[neighbours.length];
230         parents.add(this.index);
231         int i = 0;
232         HashMap<Integer, HashSet<Integer>> toGo = pathsToGo(paths);
233         if (paths.toArray().length > 0) {
234             for (Iterator<Integer> it = toGo.keySet().iterator(); it.hasNext();) {
235                 Integer key = it.next();
236                 childs[i++] = getNeighbour(key).floodGraph(level + 1,
237                         this.index, getNeighbour(key).delay, parents,
238                         toGo.get(key));
239             }
240         }
241         NodeObject obj = new NodeObject(name, primary, host, port, this.index,
242                 childs, level, parent, delay);
243         return obj;
244     }
245
246
```

Slika 26: Poplavni algoritem za izris drevesa.

V spodnji sliki je izrisana struktura drevesa najkrajših poti v grafu. V zavitih oklepajih je izračunan seznam, katerega algoritem ugotovi iz usmerjevalnih tabel najboljših poti. Iz teh poti in glede na seznam staršev posameznega vozlišča, ki ga prejme preko parametra `parents`, se izračuna seznam predstavljen v zavitih oklepajih. V seznamu so zapisana vozlišča, ki se v tej drevesni veji morajo še obiskati. Ko pridemo do listov drevesa, se izračuna prazen seznam in to predstavlja tako imenovani robni pogoj tega algoritma.



Slika 27: Primer strukture drevesa najkrajših poti.

5. Zaključek

Realizacija porazdeljenega algoritma za izračun drevesa najkrajših poti v grafu ni bila težka. Problem najkrajših poti je bil v zgodovini računalništva že do dobra defeniran in rešen z različnimi algoritmi. Porazdeljeni algoritem je bil zanimiva naloga za programirati. Program Vozlišče, katerega sem sprogramiral, uspešno reši ta problem. Poleg tega je napisan kot porazdeljen program, tako da so doseženi vsi cilji, ki sem si jih zadal pri pisanju diplomske naloge.

Danes se porazdeljeni algoritmi uporabljajo večinoma za raziksovalne namene. Ampak menim, da se bodo v prihodnosti vedno bolj uporabljali tudi za preprosto namizno programsko opremo, ki jo uporabljajo običajni uporabniki. Programska oprema namiznih računalnikov bo vedno bolj izkoriščala tehnologije za razvoj porazdeljene programske opreme. Porazdeljena programska oprema, s katero se že srečujemo danes, so odjemalci datotek v P2P omrežjih. V omrežjih P2P je svetla prihodnost in prav tako v porazdeljeni programski opremi za namizne računalnike, ki bo znala dobro izkoriščati razpoložljive vire.

Tehnologija Java RMI, ki sem jo uporabil v mojem porazdeljenem programu Vozlišče, je preprosta za uporabo in omogoča širok spekter možnosti za nadalnji razvoj porazdeljene programske opreme. Ker kot glavni programski jezik na trgu programske opreme prevladuje prav Java, menim da se bo s časom razvilo vedno več programov, ki bodo izkoristili možnost porazdeljenih algoritmov.

Slovar tujk

Tabela 2: Tuji izrazi.

ISP	Ponudnik interneta (ang. Internet Service Provider).
Flood algorithm	Algoritem za poplavljanje omrežji.
Usenet	Prvo omrežje, ki je uporabljalo algoritem poplavljanja.
Peer-to-peer	Omrežje v katerem se vozlišča povežejo med seboj.
Stub	Ročica.
Host	Gostovanje (tudi naslov gostovanja).
Port	Vrata gostovanja.
Peer	Računalnik udeleženec v določenem omrežju.
Applet	Manjša aplikacija, ki naredi eno samo opravilo.
Localhost	Lokalni TCP/IP omrežni naslov računalnika.
WSDL	Opisni jezik spletnih servisov.

Dodatek

A: Izvorna koda

Tabela 3: Izvorne datoteke.

A.1	**Razred Node**	Node.java
A.2	**Razred Client**	Client.java
A.3	**Razred Server**	Server.java
A.4	**Razred ServerInterface**	ServerInterface.java
A.5	**Razred Configuration**	Configuration.java
A.6	**Razred Neighbour**	Neighbour.java
A.7	**Razred NodeObject**	NodeObject.java
A.8	**Razred NodeSerializable**	NodeSerializable.java
A.9	**Razred RoutingObject**	RoutingObject.java
A.10	**Razred NodeUtils**	NodeUtils.java
A.11	**Razred NodeList**	NodeList.java
A.12	**Razred HardcodedDelays**	HardcodedDelays.java

Datoteke so shranjene na priloženem CD-ju.

Kazalo slik

Slika 1: Moorova krivulja. 5
Slika 2: Dvojedrna CPE. 7
Slika 3: Graf porazdeljenega sistema. 8
Slika 4: Podrobnosti v porazdeljenem sistemu. 8
Slika 5: Vzporeden sistem. 9
Slika 6: Porazdeljenost sistema PlanetLab. 10
Slika 7: Primer grafa vozlišč. 13
Slika 8: Primer drevesa najkrajših poti. 15
Slika 9: Delovanje algoritma poplavljanja. 17
Slika 10: Vmesnik Hello. 19
Slika 11: Razred Server. 20
Slika 12: Izvoz oddaljenega objekta. 20
Slika 13: Registracija objekta v registru RMI. 21
Slika 14: Povezovanje odjemalca na strežnik. 22
Slika 15: Potek klica oddaljene metode. 23
Slika 16: Registracija novih vozlišč v korenskem vozlišču. 28
Slika 17: Vozlišča povezana v mrežo vozlišč. 29
Slika 18: Nastavitve korenskega strežnika. 30
Slika 19: Nastavitve Java navideznega stroja. 30
Slika 20: Nastavitve revizijske sledi. 31
Slika 21: Nastavitve korenskega vozlišča in navadnega vozlišča. 31
Slika 22: Nastavitve izrisa drevesa in mreže vozlišč. 31
Slika 23: Primer mreže vozlišč. 33
Slika 24: Primer drevesa najkrajših poti v mreži vozlišč. 33
Slika 25: Poplavni algoritem za izračun usmerjevalnih tabel. 35
Slika 26: Poplavni algoritem za izris drevesa. 36
Slika 27: Primer strukture drevesa najkrajših poti. 36

Kazalo tabel

Tabela 1: Število tranzistorjev. 6
Tabela 2: Tuji izrazi. 39
Tabela 3: Izvorne datoteke. 41

Viri in literatura

[1] Gordon E. Moore, »Cramming More Components onto Integrated Circuits«, »Electronics«, 114-117, 1965.

[2] Gordon E. Moore, »Progress in Digital Integrated Electronics«, »Technical Digest«, 11-13, 1975.

[3] (2010) Moorova krivulja. Dostopno na: http://www.cmg.org/measureit/issues/mit41/m_41_2.html.

[4] (2010) IBM POWER4. Dostopno na: http://en.wikipedia.org/wiki/POWER4.

[5] (2010) AMD Athlon 64 X2. Dostopno na: http://en.wikipedia.org/wiki/Athlon_64_X2.

[6] (2010) Število tranzistorjev. Dostopno na: http://en.wikipedia.org/wiki/Transistor_count.

[7] (2010) Planet Lab. Dostopno na: http://www.planet-lab.org/.

[8] (2010) Vzporedno procesiranje. Dostopno na: http://en.wikipedia.org/wiki/Parallel_processing.

[9] B. Vilfan, Osnovni algoritmi, Založba FE in FRI, 2002.

[10] (2010) Porazdeljena programska oprema. Dostopno na: http://en.wikipedia.org/wiki/Distributed_software.

[11] (2010) Java RMI. Dostopno na: http://download.oracle.com/javase/6/docs/technotes/guides/rmi/index.html.