Daljnji distribuirani algoritmi za međusobno isključivanje

# Uvod

U distribuiranim sustavima često više procesa moraju dijeliti neki zajednički resurs ili određeni skup podataka. Tada obično zahtijevamo da procesi koji pristupaju tim zajedničkim ili dijeljenim resursima to ne rade istovremeno nego sekvencijalno. Kada jedan proces pristupa resursu ili mijenja podatke (kažemo da je proces u kritičnoj sekciji), niti jedan drugi proces ne bi smio pristupati istom resursu, odnosno, mijenjati ili čitati podatke dokle god ih prvi proces ne prestane koristiti. Tek dok onaj prvi proces prestane koristiti resurse (kada izađe iz kritične sekcije), drugi proces im može pristupiti. Algoritmi za međusobno isključivanje namijenjeni su da osiguraju upravo to, a u ovom seminaru ćemo opisati neke od njih.

# Singhal-ov algoritam zasnovan na dinamičkoj strukturi podataka

Mnogo algoritama za međusobno isključivanje koriste statički pristup za postizanje isključivanja, to jest, procesi obavljaju isti slijed radnji svaki puta kada traže ulaz u kritičnu sekciju, bez obzira na stanje sustava. Singhal-ov algoritam, međutim, iskorištava promjenjive uvjete unutar sustava. Konkretno, Singhal-ov algoritam se okoriščuje činjenicom da neki procesi u sustavu traže dijeljeni resurs češće, dok neki drugi procesi traže taj isti resurs rjeđe.

Ideja je da proces koji često traži pristup resursu ne treba tražiti dozvolu od procesa koji tom resursu rijetko pristupa, već samo od procesa koji mu često pristupa.

## Opis Singhal-ovog algoritma

Za rad sa Singhal-ovim algoritmom, pretpostavljamo da imamo n nezavisnih procesa (S1, S2, … , Sn) koji komuniciraju slanjem poruka kroz mrežu. Poruke stižu na odredište u konačnom, ali nepredvidivom vremenu, te za par procesa vrijedi da poruke stižu onim redoslijedom kojim su poslane.

Svaki proces Si čuva dva skupa, skup Ri (Request set) i skup Ii (Inform set). Skup Ri pamti procese od kojih Si mora tražiti dopuštenje za ulaz u kritičnu sekciju, a skup Ii pamti procese kojima Si treba poslati dopuštenje za ulaz u kritičnu sekciju nakon što sam izađe iz kritične sekcije. Svaki proces Si će još imati i logički sat Ci koji će se ažurirati prema Lamportovim pravilima i svaki zahtjev za ulaz u kritičnu sekciju će imati vremenski žig prema kojem će se određivati prioritet zahtjeva (manji žig – veći prioritet). Nadalje, svaki proces S će pamtiti tri logičke varijable „Requesting“, „Executing“ i „My\_priority“. Requesting i Executing su postavljene na TRUE akko je proces traži dopuštenje odnosno izvršava kritičnu sekciju. My\_priority je TRUE ako je zahtjev koji je Si poslao, a još nije dobio odgovor, ima veći prioritet od zahtjeva koji Si trenutno prima.

Prije početka rada, svaki proces Si postavlja svoje skupove i varijable na:

Ri := { S1, S2, … , Si-1, Si }

Ii := Si

Ci := 0

Requesting := FALSE, Executing := FALSE

Drugim riječima, u početku, svaki proces mora slati zahtjeve onim procesima koji imaju manji redni broj od njega samoga. Poredamo li sve procese u sustavu prema rednom broju padajuće s lijeva na desno (Sn je sasvim lijevo, S1 je sasvim desno), početno stanje sustava imat će slijedeća dva svojstva:

1. Svaki proces tražit će dozvolu od svih procesa desno od sebe i od nijednog procesa lijevo od sebe, te obrnuto, proces može dobivati zahtjeve od svih procese lijevo od sebe i nijednog procesa desno od sebe. Na ovaj način, svaki proces Si dijeli ostale procese na dva disjunktna skupa, na one koje traže dozvolu od Si i one od kojih Si traži dozvolu.
2. Kardinalitet skupova Ri se smanjuje „stepenasto“ s lijeva na desno. (štoviše vrijedi |Ri|= i )

Proces koji želi ući u kritičnu sekciju čini slijedeće:

Postavi svoju *Requesting* varijablu na TRUE  
Poveća svoj logički sat, Ci := Ci + 1;  
Šalje poruku REQUEST(Ci, i ) svakom procesu iz Ri  
Čeka da se Ri isprazni (Ri će se prazniti kako stižu odgovori od procesa)  
Postavi svoju *Requesting* varijablu na FALSE

Postavi svoju *Executing* varijablu na TRUE

Proces sad može ući u kritičnu sekciju, a čim završi mora napraviti slijedeće:

Postavi svoju *Executing* varijablu na FALSE

Za svaki Sk iz Ii čini:

Izbaci Sk iz Ii  
Pošalji poruku REPLY(Ci, i ) procesu Sk  
Dodaj Sk u Ri

## Objašnjenje algoritma

Proces Si koji želi ući u kritičnu sekciju traži dopuštenje (šalje poruke) svim skupovima u *Request set*-u, a oni mu šalju odgovore (ukoliko oni sami ne žele ući u kritičnu sekciju). Kako stižu odgovori, proces koji želi ući u CS iz svog *Request* skupa uklanja one procese od kojih je dobio odgovor. Kada se *Request set* isprazni, znači da je proces dobio odgovore od svih procesa od kojih je tražio dozvolu, pa slobodno ulazi u kritičnu sekciju.

Ako nijedan drugi proces ne zatraži ulaz u CS, nakon što Si izađe iz CS, njegov *Request set* će ostati prazan, pa slijedeći put kada Si zatreba CS ući će u njega bez da šalje ikakve poruke. Međutim, to vjerojatno neće biti slučaj. Dok čeka odgovore ili dok izvršava CS, proces Si će vjerojatno primiti zahtjeve za kritičnom sekcijom od drugih procesa. Te zahtjeve će Si pamtiti u svom *Inform set*-u, a kad jednom izađe iz CS-a, na svaki taj zahtjev će odgovoriti i preseliti ih iz skupa Inform u skup Request.

Ako proces Si dok čeka na ulaz u CS primi zahtjev od nekog drugog procesa, i ako taj zahtjev ima veći prioritet (odn. poslan je ranije) tada Si odmah odgovara i, ako taj drugi proces nije već u Si-ovom *Request set*-u, Si ga dodaje u *Request set* i šalje mu zahtjev, a ako jest u skupu, to znači da mu je zahtjev već poslan. U svakom slučaju, Si će primiti odgovor od tog procesa kada ovaj napusti svoju kritičnu sekciju.

# Algoritmi zasnovani na kvorumu

Algoritmi međusobnog isključivanja zasnovani na kvorumu predstavljaju odstupanje od

trenda na dva načina:

1. Proces ne traži dopuštenje od svih drugih procesa, već samo od

podskupa procesa. Ovaj pristup je drugačiji u odnosu na

Lamporta i Ricarta – Agrawala algoritama, gdje svi sudjeluju

u rješavanju sukoba svih drugih procesa. U međusobnom isključivanju na temelju kvoruma

algoritmu, skup zahtjeva procesa je odabran tako da vrijedi ∀i ∀j: 1 ≤ i j ≤ N :: Ri ∩ Rj ≠ ∅. Prema tome, svaki par procesa ima proces koje posreduje u sukobima između tog para.

2. U algoritmu uzajamnog isključivanja koji se temelji na kvorumu, proces može poslati samo jednu poruku REPLY u bilo kojem trenutku. Proces može poslati poruku REPLY tek nakon što je

primio poruku RELEASE za prethodnu poruku REPLY.

Stoga proces Si zaključava sve procese u Ri prije izvršavanja KO -a.

Algoritmi međusobnog isključivanja zasnovani na kvorumu značajno smanjuju složenost razmjenjivanja poruka tako što procesi traže dopuštenje od samo podskupa procesa.

Budući da se ti algoritmi temelje na pojmovima "koterije" i "kvorumi", prvo opisujemo ideju koterija i kvoruma. Koterij C je definirano kao skup skupova, gdje se svaki skup g ∈ C naziva kvorumom. Sljedeća svojstva vrijede za kvorume u koteriju:

• **Svojstvo presjeka** Za svaki kvorum g, h ∈ C, g ∩ h ≠ ∅.

Na primjer, skupovi {1,2,3}, {2,5,7} i {5,7,9} ne mogu biti kvorumi

U koteriju jer prvi i treći set nemaju zajednički element.

• **Svojstvo minimalnosti** Ne postoje kvorumi g, h u koteriju C

takav da g ⊇ h. Na primjer, skupovi {1,2,3} i {1,3} ne mogu biti kvorumi

koteriju jer je prvi skup nadskup drugog.

Koteriji i kvorumi mogu se koristiti za razvoj algoritama kako bi se osiguralo međusobno

isključenje u distribuiranom okruženju. Jednostavan protokol funkcionira na sljedeći način:

neka "a" bude proces u kvorumu "A." Ako "a" želi pravo na korištenje kritičnog odsječka,

traži dopuštenje od svih procesa u svom kvorumu "A." Svaki proces čini isto kada traži dopuštenje za ulaz u kritični odsječak. Zbog svojstva presjeka, kvorum "A" sadrži najmanje jedan proces koje je zajednički svakom drugom kvorumu. Procesi u bilo kojem trenutku šalju dopuštenje samo jednom procesu. Tako je zajamčeno međusobno isključivanje.

Primjetimo da svojstvo minimalnosti osigurava učinkovitost, a ne ispravnost.

U najjednostavnijem obliku, kvorumi se formiraju kao skupovi koji sadrže većinu

procesa. Postoje različiti kvorumi i različiti načini za izgradnju

kvoruma. Na primjer, Maekawa je koristio teoriju projektivnih ravnina

za razvoj kvoruma veličine √N.

# Maekawa-in algoritam zasnovan na kvorumu

Maekawin algoritam bio je prvi algoritam međusobnog isključivanja zasnovan na kvorumu. Skup zahtjeva za procese (tj. Kvorumi) u Maekawinom algoritmu su

konstruiran tako da zadovoljava sljedeće uvjete:

**M1** (∀i ∀j: i ≠ j, 1 ≤ i j ≤ N :: Ri ∩ Rj ≠ ∅).

**M2** (∀i: 1 ≤ i ≤ N :: Si ∈ Ri).

**M3** (∀i: 1 ≤ i ≤ N :: Ri = K).

**M4** Bilo koje mjesto Sj sadržano je u K broju Ri-ova, 1 ≤ i j ≤ N.

Maekawa se poslužio teorijom projektivnih ravnina i pokazao da vrijedi

N = K(K −1) +1. Ova relacija daje |Ri| =≈ √N.

Budući da postoji barem jedan zajednički proces između skupova zahtjeva bilo koja dva

procesa (uvjet M1), svaki par procesa ima zajednički proces koji posreduje u sukobu između ta dva procesa. Proces u svakom trenutku može imati najviše jednu nepodmirenu REPLY poruku; odnosno daje dopuštenje za dolazni zahtjev samo ako nije već dao dopuštenje za neki drugi proces. Stoga je međusobno isključivanje zagarantirano. Ovaj algoritam zahtijeva da isporuka poruka bude onom u redu kojim su se slali između svakog para procesa.

Uvjeti M1 i M2 neophodni su za ispravnost; dok uvjeti

M3 i M4 pružaju druge poželjne značajke algoritmu. Uvjet M3

navodi da veličina skupova zahtjeva svih procesa mora biti jednak, što

podrazumijeva da bi svi procesi trebale obaviti jednaku količinu posla kako bi omogućili

međusobno isključivanje. Uvjet M4 zahtijeva da jednak broj procesa treba zatražiti dopuštenje od bilo kojeg procesa, što podrazumijeva da svi procesi

imaju “jednaku odgovornost” u davanju dopuštenja drugim procesima.

U Maekawinom algoritmu, proces Si izvodi korake prikazane ispod za izvršavanje KO -a.

## Opis Algoritma

**Zahtjevanje kritičnog odjeljka:**

(a) Proces Si traži pristup KO -u slanjem poruka REQUEST(i)

svim procesima u svom skupu zahtjeva Ri.

(b) Kad proces Sj primi poruku REQUEST (i) od procesa Si,

vraća poruku REPLY procesu Si ukoliko nije poslalo poruku REPLY

procesu od trenutka kada je primilo posljednju poruku RELEASE.

**Izvršavanje kritičnog odjeljka:**

(c) Proces Si izvodi KO tek nakon što je primio poruku REPLY

od svakog procesa u Ri.

**Otpuštanje kritičnog odjeljka:**

(d) Nakon završetka izvršenja CS -a, proces Si šalje RELEASE (i)

poruku svakom procesu u Ri.

(e) Kad proces Sj primi poruku RELEASE (i) od procesa Si, on šalje

poruku REPLY sljedećem procesu koja čeka u redu i briše ga iz reda čekanja.

U slučaju da je red prazan, proces Sj ažurira svoj status kako bi pokazala

da nije poslala nikakvu poruku REPLY od primitka zadnje poruke RELEASE.

## Ispravnost

**Teorem 9.3**

**Maekawin algoritam postiže međusobno isključivanje.**

Dokaz: Dokaz je kontradikcijom. Pretpostavimo da su dva procesa Si i Sj istodobno

Izvršavaju KO. To znači da je proces Si primio poruku REPLY od svih

procesa u Ri i istodobno proces Sj je morao primiti poruku REPLY

sa svih procesa u Rj. Ako je Ri ∩ Rj = {Sk}, tada je proces Sk moralo poslati REPLY

poruke Si i Sj istodobno, što je kontradikcija. Q.E.D.

## Izvođenje

Uočimo da je veličina skupa zahtjeva približno √N. Stoga izvršenje KO-a zahtijeva √N REQUEST-a, √N REPLY-a i √N RELEASE-a poruka, što rezultira s 3√N poruka po izvršenju KO -a. Odgoda sinkronizacije u ovom slučaju algoritma je 2T. To je zato jer nakon što proces Si izađe iz KO -a, prvo otpušta sve procese u Ri, a zatim jedno od tih procesa šalje poruku REPLY sljedećem procesu koje izvršava KO. Stoga su potrebna dva uzastopna prijenosa poruka između dva uzastopna izvršenja KO -a. Maekawin algoritam je sklon zastoju (deadlock). Za rješavanje zastoja trebaju nam dodatne poruke.