Mobilni objekti koji putuju mrežom

Matea Stanišić
stmatea@student.math.hr

Margarita Tolja
tomarga@student.math.hr

Sadržaj

1 Sažetak

Mobilni objekt je objekt koji, izazvan korisničkim zahtjevima, putuje od procesa do procesa. U ovom radu opisujemo dva algoritma koji omogućuju *mobilnom objektu* (kratko, *objekt*), da putuje mrežom. Oba algoritma zadovoljavaju svojstvo *konzistentnosti* (*objekt* nikada nije simultano dio više procesa) i *odsustvo izgladnjivanja*, odnosno da svaki proces koji zatraži *objekt* ga eventualno i dobije.

2 Definicija problema

Mobilni objekt je objekt (datoteka, struktura podataka) kojemu mogu sekvencijalno pristupiti različiti procesi. Odnosno, radi se o objektu koji se kreće od procesa do procesa u mreži svih procesa tako da u jednom trenutku može biti dio samo jednog procesa. Također, pretpostavljamo da niti jedna obrada objekta od strane nekog procesa neće trajati beskonačno te da će ga on, eventualno, biti spreman otpustiti, tako da objekt može nastaviti putovati do drugog procesa koji ga je zatražio.

Potrebno je definirati navigacijski sustav (*navigation service*) koji osigurava postojanje procesa sa definiranim operacijama `acquire_object()` i `release_object()` tako da bilo koje korištenje objekta od strane procesa p_i je ograničeno s jednom od sljedeće tri operacije:

- `acquire_object()`,
- korištenje objekta od strane procesa p_i,
- `release_object()`.

Pri tome, proces p_i poziva operaciju `release_object()` ukoliko je prethodno zaprimio i obradio objekt, a ukoliko ponovno postane zainteresiran za njega, mora iznova pozvati `acquire_object()`.

Slučaj kada posjedovanje objekta daje vlasniku pristup je poseban slučaj problema međusobnog isključivanja. Odnosno, mobilni objekt mogli bismo promatrati kao žeton. To slijedi, zato što za proces, u problemu međusobnog isključivanja, koji posjeduje žeton u nekom trenutku, vrijedi da on može pristupiti njegovim podacima, dok ostali ne mogu. Također, svaki takav proces može zatražiti posjedovanje žetona kako bi u nekom konačnom vremenu pristupio njegovim podacima.

U nastavnim materijalima kolegija *Distribuirani procesi* [2] u Poglavlju 4 upoznali smo se sa dva algoritma međusobnog isključivanja zasnovana na žetonima. Kako je opisani problem samo instanca problema međusobnog isključivanja, mogli bismo kao jedno rješenje problema predstaviti neki od tih poznatih algoritama. U ovom radu obradit ćemo dva nova algoritma, *Navigacijski algoritam za potpune mreže* (3.1) te *Navigacijski algoritam utemeljen na razapinjućim stablima* (3.2).

3 Predložena rješenja

Algoritmi koje ćemo opisati imaju sljedeće važno svojstvo:

Ako, kada proces p_i završi s obradom objekta te niti jedan drugi proces nije zatražio objekt, on ostaje u procesu p_i.

Posjedujući ovo svojstvo, predloženi algoritmi postaju efikasniji u slučaju kada, u takvom stanju, p_i nanovo zatraži objekt jer tada nema potrebe za slanje poruka svakom drugom procesu.

3.1 Navigacijski algoritam za potpune mreže

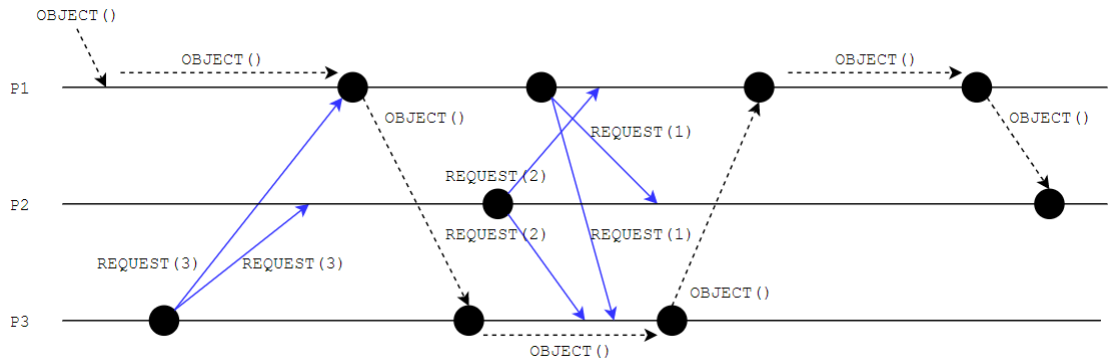
Navigacijski algoritam za potpune mreže je vrlo jednostavno rješenje problema navigacije mobilnih objekata kroz mrežu jer se kao preduvjet zahtijeva *potpunost* mreže. Odnosno, svaka dva procesa u mreži moraju biti povezana *asinkronim* kanalom. Također, kanali ne moraju osiguravati $FIFO$ načelo. Ovaj algoritam predložili su *G.Ricart* i *A.K.Agrawala* (1983.) kojeg su kasnije modificirali *I.Suzuki* i *T.Kasami* (1985.).

Svojstva algoritma:

- Sustav se sastoji od N procesa $p_1, p_2, \ldots, p_N$ te su svim procesima poznata imena svih ostalih procesa, kao i ime vlastitog procesa, odnosno oznaka.
- Ne postoji *centralni proces* kao u centralnom algoritmu kojemu se objekt *vraća* nakon što proces završi s radom objekta. Stoga slijedi glavno pitanje - *gdje će se poslati objekt nakon što proces završi rad s njim.* No, odgovor na to već smo spomenuli - *objekt ostaje kod posljednjeg procesa koji ga je obradio.*
- Da bi postavio zahtjev za objektom, proces p_j šalje poruku `REQUEST(j)` svim procesima u mreži.
- Kada proces p_i primi poruku `REQUEST(j)` on to zabilježi u lokalnoj strukturi.
- Ukoliko je proces p_i zaprimio poruku `REQUEST(j)` te p_i ima objekt koji **ne koristi** i **niti jedan drugi proces nije zainteresiran**, p_i šalje poruku `OBJECT()` procesu p_j što reprezentira kretanje objekta iz jednog procesa u drugi.
- Ukoliko je proces p_i zaprimio poruku `REQUEST(j)` te p_i ima objekt koji **koristi** ili ako ga ne koristi, ali p_j nije jedini zainteresirani proces, poruka se zabilježi u lokalnoj strukturi. Nakon što proces p_i završi s radom nad objektom, ako se radi o tom slučaju, on odabire po prioritetu sljedeći proces koji će primiti objekt te mu šalje poruku `OBJECT()`.
- Zainteresirani procesi, iz skupa svih zainteresiranih procesa nekog procesa p_i, poredani su u prioritetnom redu na sljedeći način:

$$i+1,\ i+2,\ \ldots, n,\ 1,\ 2, \ldots, i-1.$$

- Proces dobiva objekt ukoliko je primio poruku `OBJECT()`.



Slika 1: Primjer kretanja mobilnog objekta koristeći navigacijski lagoritam za potpune mreže.

Na Slici 1 možemo vidjeti primjer proces-vrijeme dijagrama s primjerom rada ovog algoritma. U primjeru imamo 3 procesa - *P1, P2* i *P3*. Na početku se objekt nalazi u procesu *P1*, koji ga ne koristi ali se zadržava kod njega. U nekom trenutku, proces *P3* šalje zahtjev za objektom svim ostalim procesima. Kako proces *P1* ne koristi objekt, on ga odmah šalje procesu *P3* koji dalje radi s objektom. U međuvremenu, zahtjeve za objektom šalju, prvo proces *P2*, a nedugo nakon njega proces *P3*. Zahtjevi dolaze

do procesa *P3*, koji ima objekt, dok ga on još koristi. Kada proces *P3* napokon završi rad s objektom, prosljeđuje ga dalje procesu *P1* zbog definiranog prioriteta u algoritmu. Nakon što proces *P1* završi s radom objekta, prosljeđuje ga do procesa *P2* kako u međuvremenu nije bilo novih zahtjeva za objektom. No, i da ih je bilo, na primjer, od strane procesa *P3*, na red bi svejedno prvo došao proces *P2*, opet zbog definiranosti prioriteta u algoritmu.

Na Slici 3.1 možemo vidjeti implementaciju predloženog algoritma.

U nastavku slijedi detaljno objašnjenje klase `MOSuzukiKasami`:

- Svaki proces u lokalnoj varijabli `interested` pamti je li on zainteresiran za objekt ili ne, postavljajući je na `true` ukoliko je zainteresiran, a inače na `false`. Varijabla se inicijalizira na `false` kako niti jedan proces na početku nije zainteresiran za obekt.
- Kako bi pamtio nalazi li se objekt trenutno kod njega, ili ne, proces u lokalnu varijablu `object_present` sprema `true` ukoliko on sadrži objekt, odnosno `false` u suprotnom. Varijabla se inicijalizira na `false` za svaki proces osim za proces koji na početku ima objekt.
- Svaki proces pamti, u lokalnoj listi `request_by[1...n]`, broj poruka poslanih od strane svakog od procesa, koliko je poznato tom procesu. Lista se inicijalizira sa nul-listom kako na početku niti jedan proces nije zainteresiran za objekt.
- Kontrolni podaci unutar mobilnog objekta spremaju se u listu `obtained[1...n]` koja služi za označavanje koliko je puta svaki od procesa posjedovao objekt. Vrijednost `obtained[i]` mijenja se samo ukoliko proces p_i posjeduje objekt. Ova varijabla je globalna, odnosno nju bi kao poruku trebao prenijeti mobilni objekt s porukom `OBJECT()`. Kako se radi o listi čije se vrijednosti mijenjaju samo od strane procesa koji ima objekt, odnosno kako nije moguće da dva različita procesa izmjenjuju vrijednosti te liste u isto vrijeme, definirali smo tu varijablu kao statičku kako bi promjena u jednoj instanci klase uzrokovala promjene u svim instancama klase `MOSuzukiKasami`, odnosno, u svim procesima.
- Svaki proces zna koji su procesi zainteresirani za objekt promatrajući sljedeći skup:

  $$S = \{\, p_j \mid \texttt{request\_by[j]} > \texttt{obtained[j]} \,\}.$$

  Ukoliko je skup S neprazan proces zna da postoje neki drugi procesi koji su zainteresirani za objekt.
- U metodi `acquire_object()` proces postavlja lokalanu vrijednost `intersted` na `true` kako bi zapamtio da je zainteresiran za objekt. Ukoliko proces ne posjeduje objekt, on šalje svim drugim procesima žigosanu poruku `REQUEST(myID)`, poveća lokalnu vrijednost od `request_by[myId]` za 1 te uđe u stanje za čekanje.
- U metodi `release_object()` proces postavlja lokalanu vrijednost `intersted` na `false` kako bi zapamtio da više nije zainteresiran za objekt. U metodi se također postavlja `obtained[myId]` na `request_by[myId]` kako bi proces obavijestio ostale procese da on više nije zainteresiran za objekt. Nadalje se pretražuje skup svih zainteresiranih procesa na prethodno definirani način, te ukoliko se nađe neki zainteresirani proces, postavlja se `object_present` na `false` te se prvom takvom procesu šalje poruka `OBJECT()`. Nakon toga, izlazi se iz metode. U slučaju da nema takvih procesa, objekt ostaje kod procesa (`object_present = true`), ali je zapamćeno da proces više nije zainteresiran za njega (`interested = false`).
- U metodi `handleMsg`, kod primitka poruke `OBJECT` postavlja se varijabla `object_present` na `true` i o tome se obavještavaju svi procesi. Kod primitka poruke `REQUEST`, ažurira se vrijednost `request_by[srcId]` uvećavanjem za 1. Ukoliko se radi o procesu koji ima objekt (`object_present = true`), ali za njega nije zainteresiran (`interested = false`), šaljemo objekt procesu koji ga je zatražio slanjem poruke `OBJECT` te prethodnim postavljanjem `object_present` na `false`.

```java
public class MOSuzukiKasami extends Process implements Lock {
    boolean interested;
    boolean object_present;
    int[] request_by;
    static Integer[] obtained;

    public MOSuzukiKasami(Linker initComm, int starter) {
        super(initComm);
        interested = false;
        object_present = (myId == starter);
        request_by = new int[N];
        for (int i = 0; i < N; i++) request_by[i] = 0;

        obtained = new Integer[N];
        for (int j = 0; j < N; j++) {
            obtained[j] = 0;
        }
    }
    // acquire_object()
    public synchronized void requestCS() {
        interested = true;
        if( !object_present ){
            request_by[myId] = request_by[myId] + 1;
            broadcastMsg("REQUEST", myId);
            while ( !object_present ){
                myWait();
            }
        }
    }
    // release_object()
    public synchronized void releaseCS() {
        interested = false;
        obtained[myId] = request_by[myId];

        for(int k = myId + 1; k < N; k++){
            if( request_by[k] > obtained[k] ){
                object_present = false;
                sendMsg(k, "OBJECT");
                return;
            }
        }
        for(int k = 0; k < myId; k++){
            if( request_by[k] > obtained[k]){
                object_present = false;
                sendMsg(k, "OBJECT");
                return;
            }
        }
    }
    public synchronized void handleMsg(Msg m, int src, String tag) {
        if (tag.equals("OBJECT")) {
            object_present = true;
            notifyAll();
        }
        else if (tag.equals("REQUEST")){
```

```
56              int k = m.getSrcId();
57              request_by[k] = request_by[k] + 1;
58              if( object_present && !interested){
59                  object_present = false;
60                  sendMsg(k, "OBJECT");
61              }
62          }
63      }
64  }
```

Broj poruka potrebnih za jedno korištenje objekta je 0 ukoliko je objekt već prisutan u zainteresiranom procesu, odnosno N poruka ($N-1$ `REQUEST` i jedna `OBJECT` poruka) inače. Stoga je složenost ovoga algoritma $O(log_2 N)$.

Algoritam možemo istestirati jednostavnim dodavanjem linija

```
1       if (args[3].equals("MobileObject"))
2           lock = new MOSuzukiKasami(comm,0);
```

u `LockTester.java` datoteku na linije 18 i 19. Nakon toga potrebno je samo pokrenuti

```
1       java LockTester <bazno ime> <i> <N> MobileObjects
```

gdje je prvi argument naredbenog retka nakon `LockTester` bazno ime pod kojim `NameServer` evidentira izvođenje programa. Slijede argumenti koji označavaju redni broj dotičnog procesa i ukupan broj procesa.

Svojstva algoritma:

- Ispunjeno je svojstvo *sigurnosti* jer postoji samo jedan objekt pa samo onaj proces koji ga posjeduje može ući u *kritičnu sekciju*, koristeći terminologiju problema međusobnog isključivanja.
- Vrijedi *odsustvo izgladnjivanja*. Definiranjem prioriteta zainteresiranih procesa garantira se eventualno posjedovanje objekta svakom procesu jer na opisani način nismo favorizirali niti jedan proces.
- Ne vrijedi svojstvo *pravednosti*. Protuprimjer možemo vidjeti na Slici 1.
- Sve četiri metode klase `ModelObjects` izvode se atomarno (međusobno su isključiva), osim `myWait()`. No, to ne znači da se proces koji trenutno koristi objekt ne može prekinuti `REQUEST` porukom nekog drugog procesa.

Dakle, mogli bismo reći da su prednosti ovog algoritma ispunjenost *sigurnosti*, *odsustvo izgladnjivanja* te da nije nužno da kanali u mreži osiguravaju *FIFO* načelo. S druge strane, nedostaci algoritma su pretpostavka o potpunosti mreže te neispunjenost *pravednosti*.

3.2 Navigacijski algoritam utemeljen na razapinjućem stablu

Prethodni navigacijski algoritam se temelji na raspršivanju zahtjeva po cijelog mreži procesa koja može biti proizvoljne veličine. Suprotno tome, *navigacijski algoritam utemeljen na razapinjućem stablu* podrazumijeva **razmjenu poruka samo između susjednih procesa u statičkom razapinjućem stablu mreže**, čime se postiže lokalnost algoritma. Kao i u prošlom algoritmu, kanali ne moraju osiguravati *FIFO* načelo.

Algoritam je predložio *K. Raymond* (1989.).

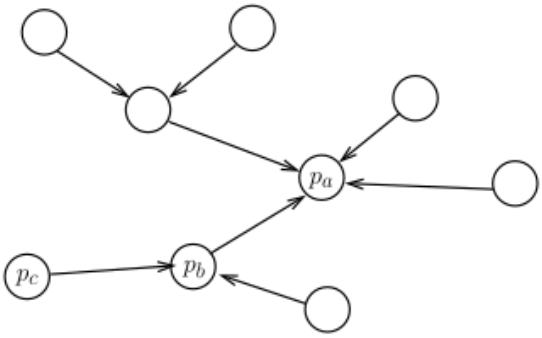
3.2.1 Principi algoritma

Invarijantnost strukture stabla (*tree invariant*)

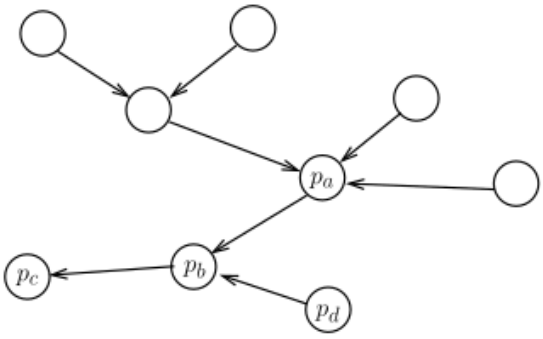
Ponovimo, ovaj algoritam koristi isključivo kanale statičkog razapinjućeg stabla mreže. Pritom, u svakom trenutku vrijede sljedeća svojstva:

- Proces koji trenutno posjeduje *objekt* se nalazi u korijenu stabla.
- Svaki proces ima *pokazivač* na susjedni proces u podstablu koji sadrži *objekt*.

Dolje su prikazani primjeri strukture stabla neke mreže procesa.



Slika 2: Inicijalno stanje



Slika 3: Stanje nakon pomaka *objekta*

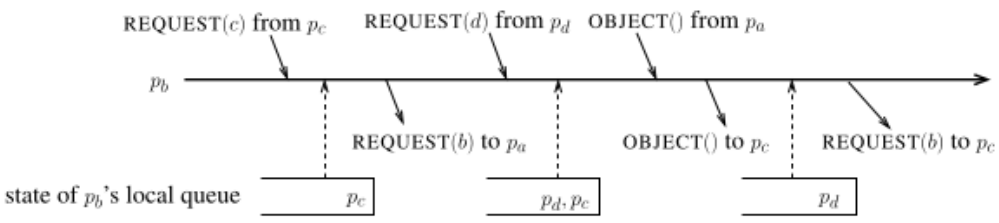
Lijeva slika prikazuje inicijalno stanje mreže. Korijen stabla, tj. proces koji posjeduje *objekt*, je proces p_a, budući da ne postoji *pokazivač* s tim procesom kao ishodištem, te su svi *pokazivači* orijentirani prema podstablu čiji je on dio.

Sada promotrimo slučaj kada p_c želi zatražiti *mobilni objekt*. Prateći strelice na slici 2 vidimo kako se zahtjev prvo predaje njegovom *roditelju*, procesu p_b, koji ga onda prosljeđuje procesu p_a. Nakon što p_a primi zahtjev, *mobilni objekt* putuje do naručitelja p_c **istim putem kojim je putovao zahtjev** u suprotnom smjeru. Prelazeći iz jednog procesa u drugi, *objekt* mijenja orijentaciju pripadnih kanala te tako osigurava invarijantnost opisanih svojstava. Ova tehnika se naziva *edge reversal*. Stanje nakon pomaka objekta je prikazano na desnoj slici te je očito kako je sada p_c korijen stabla.

Svojstvo zastupnika (*proxy behavior*)

U prošlom slučaju smo naveli kako proces p_b *prosljeđuje* p_c-ov zahtjev do korijena p_a. Međutim, ipak nije moguće doslovno proslijediti zahtjev od p_c do p_a. Budući da p_a nema informaciju o procesima koji mu nisu susjedni (pa tako ni o p_c), takvim prosljeđivanjem p_a ne bi znao odakle je zahtjev zapravo došao. Ideja *prosljeđivanja* se zato ostvaruje tako da posredni proces p_b, nakon što primi zahtjev od p_c, šalje zahtjev za objekt prema p_a **u svoje ime** te *pamti* da je naručitelj zapravo proces p_c. U ovom slučaju onda možemo reći da p_b igra ulogu *zastupnika* od p_c. Kada p_b povratno zaprimi objekt, on ga prosljeđuje stvarnom naručitelju p_c.

No što ako p_b, nakon što primi zahtjev od p_c, također primi zahtjev od p_d? Kako bi se osiguralo pravedno upravljanje ovim zahtjevima, p_b vodi lokalnu FIFO strukturu u kojoj pohranjuje pristigle zahtjeve. Upravljanje takvom strukturom je demonstrirano u sljedećem primjeru.



Slika 4: Lokalna FIFO struktura procesa p_b

Na početku je lokalni red procesa p_b prazan. Nakon primitka prvog zahtjeva, ime naručitelja p_c se sprema u red te p_b šalje zahtjev u svoje ime k p_a. Sljedeći zahtjev, od strane procesa p_d, se također sprema u red, ali budući da je p_b već poslao zahtjev za objektom k p_a, taj zahtjev se ne ponavlja. Tek nakon što p_b zaprimi objekt te ga proslijedi prvom procesu u redu, dakle procesu p_c, p_b šalje novi zahtjev za objektom, ponovno u svoje ime, kojeg će kasnije proslijediti do p_d.

3.2.2 Algoritam i implementacija

Struktura algoritma je jednaka kao u navigacijskom algoritmu za potpune mreže.

Lokalne varijable:

- Cjelobrojna varijabla `parent` koja sadrži identifikator procesa koji mu je *roditelj*, tj. susjednog procesa u stablu na putu do korijena.
- Logička varijabla `interested` ima ista svojstva kao u prošlom algoritmu.
- Logička varijabla `present` više nije potrebna, budući da je implicitno poznata preko `parent` varijable:

  $$\texttt{present == true} \iff \texttt{parent}_i \texttt{ == i}.$$

  Radi jednostavnosti, u našoj implementaciji ipak koristimo ovakvu varijablu.
- Lokalna *FIFO* struktura queue koja sadrži identifikatore susjednih procesa od čije strane je stigao zahtjev za *objektom*. Ova struktura je inicijalno prazna.

Implementacija

Klasa koja predstavlja proces koji radi s *mobilnim objektom* nasljeđuje klasu `SpanTree`, obrađenu u [2], koja je prigodna za simulaciju mreže procesa temeljene na strukturi razapinjućeg stabla.

```
1  public class MOSpanTree extends SpanTree implements Lock{
2
3          private boolean interested;
4          private boolean present;
5          private IntLinkedList queue;
6
7          public MOSpanTree( Linker initComm, boolean isRoot ) {
8                  super( initComm, isRoot );
9
10                 interested = false;
11                 present = isRoot;
12                 queue = new IntLinkedList();
13         }
14         // acquire_object
15         @Override
16         public synchronized void requestCS() {
17                 interested = true;
18                 if ( !present ) {
19                         queue.add( myId );
20                         if ( queue.size() == 1 ) {
21                                 sendMsg( parent, "REQUEST" );
22                         }
23                         while( !present ) {
24                                 myWait();
25                         }
26                 }
27         }
```

```
28          // release_object
29          @Override
30          public synchronized void releaseCS() {
31                  interested = false;
32                  if ( !queue.isEmpty() ) {
33                          int head = queue.removeHead();
34                          sendMsg( head, "OBJECT" );
35                          parent = head;
36                          present = false;
37                          if ( !queue.isEmpty() ) {
38                                  sendMsg( parent, "REQUEST" );
39                          }
40                  }
41          }
42          @Override
43          public synchronized void handleMsg( Msg m, int src, String
               tag ) {
44                  super.handleMsg( m, src, tag );
45                  if ( tag.equals( "REQUEST" ) ) {
46                          if ( present ) {
47                                  if ( interested ) {
48                                          queue.add( src );
49                                  } else {
50                                          sendMsg( src, "OBJECT" );
51                                          parent = src;
52                                          present = false;
53                                  }
54                          } else {
55                                  queue.add( src );
56                                  if ( queue.size() == 1 ) {
57                                          sendMsg( parent, "REQUEST" );
58                                  }
59                          }
60                  }
61                  if ( tag.contentEquals( "OBJECT" ) ) {
62                          int head = queue.removeHead();
63                          if ( myId == head ) {
64                                  parent = myId;
65                                  present = true;
66                                  notify();
67                          } else {
68                                  sendMsg( head, "OBJECT" );
69                                  parent = head;
70                                  if ( !queue.isEmpty() ) {
71                                          sendMsg( parent, "REQUEST" );
72                                  }
73                          }
74                  }
75          }
76  }
```

Slijedi detaljniji opis metoda `MOSpanTree` klase koja simulira rad navigacijskog algoritma. Sve metode navedene klase nose oznaku `synchronized` čime se onemogućuje da više dretvi istovremeno izvršavaju istoimenu metodu.

- `requestCS (acquire_object())`. Proces poziva ovu metodu ukoliko želi predati zahtjev za *mobilnim objektom*. Metoda zato prvo postavlja pripadnu `interested` zastavicu na `true`. Dalje, vrši se provjera o prisutnosti *objekta* u tom procesu – ako je *objekt* već u nadležnosti procesa, tu metoda završava. U suprotnom, identifikator procesa se dodaje u lokalni red sa zahtjevima – `queue` (linija 19). U slučaju da taj red sadrži samo upravo dodani identifikator, *procesu roditelju* se šalje `REQUEST(myId)` poruka. Ako je red duljine veće od jedan, poruka se ne šalje, budući da se podrazumijeva kako je takva poruka već poslana za proces na početku reda. Proces se na kraju zadržava u ovoj metodi sve dok ne zaprimi *mobilni objekt* (linije 23, 24).

- `releaseCS (release_object())`. Ova metoda se poziva nakon što proces završi rad s *mobilnim objektom* te više nema potrebe za njim. Metoda zato prvo resetira `interested` zastavicu na `false`. Sljedeće se provjerava je li `queue` prazan, te ako je, metoda se završava, tj. *objekt* ostaje kod trenutnog procesa jer nema drugih zainteresiranih. Ako je red zahtjeva neprazan, *objekt* se predaje susjedu s početka reda. Ovo se realizira na način da se proces s početka ukloni iz reda te se istom pošalje `OBJECT()` poruka (linije 33, 34). Zatim se lokalna `parent` varijabla ažurira tako da pokazuje na susjedni proces kojemu je netom predan *objekt*. Na taj način se održava ranije spomenuta invarijantnost strukture stabla – *roditelj* procesa je uvijek onaj susjedni proces *na putu prema procesu koji posjeduje objekt*. Na samom kraju metode se još provjerava je li `queue` ostao prazan nakon uklanjanja procesa s početka reda. Ako to nije slučaj, znači da postoji još procesa koji su ranije poslali zahtjev za objekt te se zato šalje novi `REQUEST(myId)` prema novom *roditelju*. Budući da postoji slučaj kada se `OBJECT()` i `REQUEST(myId)` poruke šalju jedna za drugom, i to prema istom procesu, u praksi je moguće te poruke poslati kao jednu i tako smanjiti mrežni promet.

- `handleMsg`. Ova metoda upravlja načinom na koji se zaprimljene poruke obrađuju.
  - Zaprimljena poruka `REQUEST(k)`. Ovakva poruka znači da je susjedni proces p_k poslao zahtjev za *objektom*. Proces koji je primio poruku, istu obrađuje ovisno o tome posjeduje li trenutno *objekt* ili ne. Ako proces posjeduje *objekt*, potrebna je dodatna provjera je li proces spreman prepustiti *objekt* ili ne. To se određuje preko `interested` zastavice. Ako proces i dalje koristi *objekt*, p_k (tj. identifikator `k`) će se dodati u `queue` te metoda završava s radom. Ako procesu više nije potreban *objekt*, prosljeđuje ga procesu p_k, te ažurira `parent` varijablu da pokazuje na p_k. Ukoliko proces koji je primio zahtjev ne posjeduje *objekt*, `k` se dodaje u lokalni `queue` te ukoliko je to jedini zahtjev u redu, proces kao *zastupnik* p_k-a šalje novi `REQUEST(myId)` prema *roditelju*.
  - Zaprimljena poruka `OBJECT()`. Ovakva poruka simulira dobivanje *mobilnog objekta* na korištenje. Proces koji je primio *objekt* prvo provjerava je li on dobio pravo na korištenje ili je samo u ulozi *zastupnika*. To se obavlja uklanjanjem i provjerom procesa s početka lokalnog reda zahtjeva `queue`. Ukoliko je identifikator na početku reda jednak identifikatoru trenutnog procesa, trenutni proces postaje vlasnik *objekta*, tj. korijen razapinjućeg stabla. Zbog toga je potrebno ažurirati `parent` varijablu na `myId`. Također, o promjeni stanja `present` zastavice je nužno obavijestiti dretvu koja provjerava njeno stanje u `while` petlji metode `requestCS()`. To se postiže upotrebom `notify` metode u liniji 66. Ako je proces koji je zaprimio *objekt* zahtjev inicijalno poslao za susjeda, *objekt* će biti proslijeđen točno tom susjedu, koji zbog toga postaje njegov *proces roditelj*. Preostale linije 70 i 71 su jednake kao zadnje linije metode `releaseCS()` te se, kao i tamo, odnose na slanje novog `REQUEST(myId)` zahtjeva ukoliko je red sa zahtjevima neprazan.

Algoritam ponovno testiramo po uzoru na `LockTester` klasu iz [2]. Budući da ovaj algoritam ne zahtijeva potpunu mrežu, povezanost procesa u mreži možemo zadati proizvoljno. U tu svrhu, za svaki proces u mreži se napravi jednostavna ASCII datoteka naziva *topologyIdProcesa* u kojoj su navedeni identifikatori procesa s kojima je taj proces povezan. Na temelju tih veza, tester će prvo konstruirati razapinjuće stablo (tj. postaviti odgovarajuće `parent` varijable). Donje linije su ubačene u originalni tester prije `while` petlje u kojoj se izmjenjuju *release* i *request* metode.

```
1    lock = new MOSpanTree(comm,myId == 0);
2    for (int i = 0; i < numProc; i++)
3       if (i != myId)
4          (new ListenerThread(i, (MsgHandler)lock)).start();
5    lock.waitForDone();
```

U liniji 1 je, radi jednostavnosti, postavljeno da *objekt* inicijalno bude prisutan u procesu p_0. Linija 5 se dodaje, kao i kod `SpanTreeTester` klase ([2]), kako bi se pravo testiranje počelo odvijati tek nakon što se osigura struktura razapinjućeg stabla.

Složenost algoritma. Kao i u prošlom algoritmu, broj razmijenjenih poruka u najboljem slučaju je trivijalno jednak 0 (slučaj kada *objekt* zahtijeva proces koji ga već posjeduje). Najgori slučaj ovisi o veličini, tj. *promjeru D* razapinjućeg stabla te se ostvaruje kada od dva *najudaljenija* procesa jedan posjeduje, a drugi zahtijeva *objekt*. U tom slučaju, potrebno je D poruka i vremenskih jedinica da zahtjev stigne do procesa s *objektom* te još toliko poruka i jedinica kako bi *objekt* stigao do procesa naručitelja. Dakle, složenost algoritma, bilo po broju poruka ili vremenskih jedinica, je $O(D)$.

Svojstva algoritma

- O identifikatorima. Budući da svaki proces p_i razmjenjuje poruke samo sa svojim susjedima u razapinjućem stablu, jedini identifikatori koji se mogu pojaviti u pripadnom redu queue_i su identifikatori susjeda te i. Zbog toga je potrebno da identifikatori susjednih procesa budu različiti. Međutim, kada je riječ o procesima s udaljenošću 2 ili većom, moguće je da oni imaju jednake identifikatore, bez da se korektnost algoritma naruši. Ovo svojstvo je posebice važno istaknuti u smislu skalabilnosti i lokalnosti ovog navigacijskog algoritma.

- O prioritetima. Lokalne queue strukture u kojima se spremaju zahtjevi originalno rade prema *FIFO* načelu. Međutim, moguće je ovim strukturama upravljati na temelju nekog drugog prioritetnog koncepta, koji eventualno favorizira neke procese, sve dok takav koncept ne vodi do izgladnjivanja. Ovakva varijabilnost algoritma se može pokazati korisnom u praksi.

- *Pravednost.* Ovo svojstvo algoritam ne može osigurati jer se, zbog specifičnosti mreže, ne može garantirati da će razmjena poruka uvijek teći pravilnim redoslijedom. Jednostavan protuprimjer je slučaj u kojem dva procesa, p_i i p_j šalju zahtjev istom procesu p_k. Pretpostavimo da iako je proces p_i poruku poslao prije procesa p_j, proces p_k je ranije zaprimio zahtjev od p_j. Stoga, identifikator j će biti ispred identifikatora i u queue_k, te će p_j dobiti *objekt* na korištenje prije p_i.

- *Sigurnost i odsustvo izgladnjivanja.* Ova svojstva su ispunjena po dokazu u 3.2.3.

Dakle, mogli bismo reći da ovaj algoritam ima jednake prednosti kao i prošli. Dodatno, kao prednost ovog algoritma nad prošlim, treba istaknuti to što ne zahtjeva potpunu mrežu procesa, lokalan je, te vrijede svojstva navedena u prve dvije točke. Mali nedostatak algoritma je što se eventualno treba utrošiti vrijeme na definiranje ili pronalaženje razapinjućeg stabla koje će se koristiti kao topologija.

3.2.3 Dokaz korektnosti

Teorem 1. *Algoritam opisan u 3.2.2 garantira sljedeća svojstva.*

(i) Sigurnost – mobilni objekt se niti u jednom trenutku ne nalazi kod dva različita procesa istovremeno.

(ii) Odsustvo izgladnjivanja – svaki proces koji preda zahtjev za objektom ga eventualno dobije na korištenje.

Dokaz. (i) Dokaz za svojstvo *sigurnosti* se provodi indukcijom po broju prenošenja *objekta*.
Baza indukcije je zadovoljena budući da se incijalno *objekt* nalazi kod samo jednog procesa.

Pretpostavimo sada da je svojstvo *sigurnosti* ispunjeno za prvih m prenošenja *mobilnog objekta*. Neka je p_i proces koji dobije *objekt* na korištenje u m-tom pomaku. Ako p_i zadrži *objekt* zauvijek, *sigurnost* je trivijalno ispunjena. Zato pretpostavljamo da p_i eventualno otpušta *objekt*. Razmotrimo sve moguće scenarije kako p_i može poslati *objekt* dalje.

- p_i poziva `release_object()` metodu. U ovom slučaju vrijedi $i \notin \text{queue}_i$. Ovo je zato jer nakon zadnjeg poziva `acquire_object()` metode, kada je p_i zaprimio *objekt*, identifikator i se nalazio na početku queue_i te je iz istog uklonjen (linija 62).

  Sada imamo dva slučaja. Ako $\text{queue}_i = \emptyset$, p_i će zadržati *objekt* te je time sigurnost zadovoljena. Ako $\text{queue}_i \neq \emptyset$, p_i šalje *objekt* susjedu p_k čiji je identifikator bio na početku queue_i reda. Također, $\texttt{parent}_i$ se postavlja na k. Kao svojstvo ispravno implementiranog algoritma smo ranije spomenuli da identifikatori susjednih procesa nužno moraju biti različiti. Sada imamo $\texttt{parent}_i = k \neq i$, što znači da p_i više ne posjeduje *objekt* te je *sigurnost* zadovoljena.

- p_i predaje *objekt* po primitku `REQUEST(k)` poruke (linija 50). Budući da samo proces p_i može poslati `REQUEST(i)` poruku, a proces ne šalje poruke sam sebi, slijedi $i \neq k$. Dakle, nakon što se $\texttt{parent}_i$ postavi na k (linija 51), vrijedit će $\texttt{parent}_i \neq i$, tj. p_i više neće posjedovati $objekt$ pa je *sigurnost* zadovoljena.

- p_i prosljeđuje *objekt* odmah nakon što ga dobije (linija 68). *Objekt* se prosljeđuje procesu p_k s početka queue_i. Ovaj odlomak se izvršava uz uvjet $i \neq k$ pa ponovno imamo $\texttt{parent}_i \neq i$ čime je *sigurnost* zadovoljena.

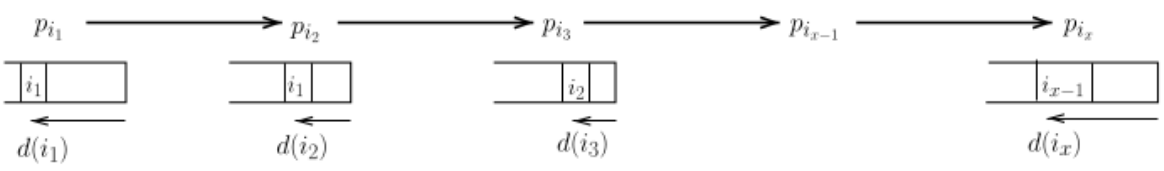
(ii) Dokaz za drugo svojstvo se provodi kroz dva dijela.

Deadlock freedom. Prvo je potrebno dokazati da ako postoje procesi koji zahtijevaju *objekt*, jedan od tih procesa će *objekt* i dobiti na korištenje. Pretpostavimo da proces p_i posjeduje *objekt*, te bar jedan proces p_j zahtijeva *objekt*. Dakle, p_j šalje `REQUEST(j)` poruku svom roditelju, nakon čega i njegov roditelj dalje šalje $\texttt{REQUEST(parent}_j\texttt{)}$ poruku svom roditelju itd. Zbog orijentacije bridova stabla prema korijenu, te odsustva ciklusa u stablu, p_i će naposljetku zaprimiti `REQUEST(k)` poruku (iako možda ta poruka ne dolazi nužno zbog p_j-ovog zahtjeva). Sada, ako je $\texttt{interested}_i$ istinit, zahtjev od p_k se dodaje u queue_i. Proces p_i će onda otpustiti *objekt* tek nakon što pozove `release_object()` metodu. U suprotnom, ako je $\texttt{interested}_i$ neistinit, p_i će poslati objekt procesu p_k. Dakle, za bilo koju vrijednost $\texttt{interested}_i$ zastavice, p_i će eventualno poslati *objekt* prema p_k. Ako je k na početku queue_k reda, p_k je dobio traženi *objekt* na korištenje. Inače, p_k prosljeđuje *objekt* procesu čiji je identifikator prvi u queue_k. Iterirajući ovaj postupak, te uzimajući u obzir svojstva definiranog apstraktnog razapinjućeg stabla, slijedi da *mobilni objekt* dobija neki proces koji ga je i zatražio. Dakle, algoritam je *deadlock free*.

Starvation freedom. Još je ostalo dokazati da će svaki proces koji zahtijeva *mobilni objekt* istog eventualno dobiti na korištenje. Neka je D *promjer* razapinjućeg stabla, $1 \leq D \leq n-1$. Neka je proces p_i poslao zahtjev `REQUEST(i)`, dok proces p_j posjeduje *objekt*. Definirajmo još i sljedeće pomoćne oznake:

- $p_i = p_{i_1}, p_{i_2}, \ldots, p_{i_{x-1}}, p_{i_x} = p_j$ — procesi u stablu *na putu* od p_i do p_j
- $N(i_y)$ — broj susjednih procesa od p_{i_y} u stablu
- $d(i_y)$ — indeks identifikatora i_{y-1} u redu queue_y $(1 \leq d(i_y) \leq N(i_y))$

Koristeći ovakve oznake *put* koji p_i-jev zahtjev prolazi se može vizualizirati na sljedeći način.



Slika 5: Kretanje zahtjeva od p_i do p_j

Sada za navedeni zahtjev procesa p_i definiramo niz R td.

$$R = [d(i_1),\ d(i_2),\ldots,\ d(i_{x-1}),\ d(i_x),\ 0,\ldots,\ 0]\,.$$

Istaknimo da, budući da je R dimenzije D, te svaka od dimenzija ima ograničen broj mogućih vrijednosti, slijedi da postoji konačan skup mogućih vrijednosti niza R, te je isti leksikografski totalno uređen. Također, znamo da je svaka lokalna struktura zahtjeva uređena po FIFO principu te da procesi ne mogu izdavati nove zahtjeve dok im prethodni nije ispunjen. Zbog toga, kada proces p_x šalje *objekt* (a ovo se nužno događa po prvom dijelu dokaza), niz R se može izmijeniti na sljedeća dva načina:

- $R' = [d(i_1),\ d(i_2),\ldots,\ d(i_{x-1}) - 1,\ 0,\ldots,\ 0]$, ako $d(i_x) = 1$
- $R'' = [d(i_1),\ d(i_2),\ldots,\ d(i_{x-1}),\ d(i_x) - 1,\ *,\ldots,\ *]$, ako $d(i_x) > 1$.

Prvi slučaj se ostvaruje kada p_{i_x} šalje *objekt* procesu $p_{i_{x-1}}$ s početka reda zahtjeva, a drugi kada je na tom mjestu neki drugi proces. Zvjezdice u nizu R'' označavaju prikladne vrijednosti s obzirom na trenutna stanja ostalih $(D - x)$ lokalnih redova zahtjeva. Važno je primijetiti kako prelaskom s niza R na niz R' ili R'', prelazimo na manji niz s obzirom na totalni uređaj skupa mogućih R nizova. Analogno dalje, s niza R' ili R'' prelazimo na niz R''' koji će opet biti uređajno manji od prethodnih. Ovaj postupak se ponavlja sve dok ne dobijemo niz oblika $[1,\ 0,\ldots,\ 0]$. U tom trenutku je jasno da je proces p_i zaprimio *mobilni objekt*, čime je zadovoljeno svojstvo odsustva izgladnjivanja.

□

4 Zaključak

Mobilni objekt je struktura podataka koja putuje mrežom u skladu sa zahtjevima procesa. Problem njegovog putovanja mrežom zapravo je poopćenje problema međusobnog isključivanja. U ovom radu smo, po uzoru na [3], ponudili dva navigacijska algoritma koji predstavljaju rješenja problema. Navedena je implementacija pripadnih klasa pomoću kojih se može simulirati rad algoritama. Analizom svojstava algoritama, ustanovljeno je kako algoritam, koji kao podlogu koristi razapinjuće stablo mreže, ima kvalitetnija svojstva, dok algoritam za potpune mreže predstavlja jednostavno, ali i dalje dobro rješenje. Još jedno zajedničko svojstvo obrađenih algoritama je njihova *neprilagodljivost* (*inadaptivity*). Naime, ukoliko postoje procesi koji nakon nekog vremena nikada više neće biti zainteresirani za korištenje *mobilnog objekta*, ovi algoritmi ne nude način da isti spontano prestanu sudjelovati u algoritmu. Neke od poboljšanih verzija navigacijskih algoritama koje nude ovo svojstvo se također mogu naći u [3].

Literatura

[1] Vijay K. Garg. URL: http://users.ece.utexas.edu/~garg/ (pogledano 13. 7. 2020).

[2] Robert Manger. *Distribuirani procesi*. 2017.

[3] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer Publishing Company, Incorporated, 2013. ISBN: 3642381227.