Distribuirano izvršavanje transakcija

“Two-phase commit” protokol

Ivona Čižić

Mateo Martinjak

1. Projektni zadatak

Transakcija je niz operacija nad podacima koje se ponašaju kao jedna nedjeljiva cjelina. Istovremeno izvođenje više transakcija mora biti ekvivalentno nekom njihovom sekvencijalnom izvođenju. Ako transakcija zbog greške ne dođe do kraja, tada se njezin dotadašnji učinak mora neutralizirati (rollback). Ako je transakcija došla do kraja, tada se njezin ukupni učinak mora trajno pohraniti (commit). U projektu treba obraditi problem kontrole konkurentnog izvođenja transakcija, kao i tehnike za neutralizaciju transakcije u slučaju kad je došlo do greške prije kraja njezinog izvođenja. No najveću pažnju u projektu treba posvetiti problematici distribuiranog izvršavanja transakcije. Dakle ako se transakcija sastoji od dijelova koji su raspoređeni na više procesa (računala), kako osigurati da svi ti dijelovi složno naprave ili commit ili rollback? Riječ je o specifičnom problemu usuglašavanja. U projektu treba objasniti, analizirati, implementirati i testirati poznati algoritam dvofaznog pohranjivanja (two-phase commit).

2. Uvodno o transakcijama

Transakcija je veoma koristan koncept koji objedinjuje niz operacija nad podacima koje se zajedno ponašaju kao jedna nedjeljiva cjelina. S mjesta promatrača, stječemo dojam da su sve operacije u nizu izvršene ili da se niti jedna operacija u nizu nije izvršila. Transakcija čuva ovo svojstvo nevidljivosti neovisno o neuspjesima u izvođenju ili istovremenosti izvođenja nekoliko transakcija u isto vrijeme. Transakcije pri istovremenom izvršavanju moraju imati isti učinak kao da su se sekvencijalno izvršavale. S druge strane, ako se transakcija koja se ne izvrši do kraja i došlo do pogreške, svi podaci se moraju vratiti na početno stanje prije početka transakcije, drugim riječima, dosad obavljene operacije se moraju poništiti. Ako je transakcija uspješno završena, njen učinak nad podacima mora postati trajan.

Kao primjer transakcije navodimo transfer novca sa računa označenog slovom A, na račun označen slovom B. Transakciju T_1 možemo zapisati na sljedeći način:

```
započni_transakciju
     umanji iznos na računu A za x (povuci);
     povećaj iznos na računu B za x(položi);
završi_transakciju
```

Nakon što grupiramo niz operacija koje čine transakciju, prividno dobijemo nedjeljivu cjelinu. Kako bi čim zornije objasnili, pretpostavimo da istovremeno s transakcijom T_1 započinje transakcija T_2. Transakcija T_2 jednostavno povećava iznos na oba računa za određenu svotu. Kako su transakcije jedna drugoj nedjeljive cjeline, znamo da se povećanje iznosa na računu A kao sastavni dio transakcije T_2 neće dogoditi nakon povlačenja iznosa x s računa A u sklopu transakcije T_1 i prije povećanja iznosa za x na računu B u sklopu transakcije T_1. Ukoliko je došlo do greške prilikom izvršavanja operacija transakcije, zbog nedjeljivosti transakcije promatrač će steći dojam da niti jedna u nizu operacija koje čine transakciju nije izvršena, iako je do greške moglo doći na bilo kojoj od prve do zadnje operacije u nizu onih koje čine navedenu transakciju. Ako iz nekog razloga druga operacija iz transakcije T_1 nije mogla biti izvršena(jedan takav razlog bio bi nepostojanje računa B), tada nije vidljiv ni učinak prve operacije, što znači da iznos na računu A ostaje nepromijenjen.

Transakciju možemo svesti na implementaciju sljedećih operacija:

1. **započni_transakciju**: označava početak transakcije
2. **završi_transakciju**: označava kraj transakcije, svi učinci između početka i kraja transakcije čine samu transakciju, a izvršenjem ove operacije učinci transakcije postaju trajni
3. **prekini_transakciju**: korisnik može pozvati ovu funkciju usred transakcije i posljedično će sve promjene unutar transakcije biti poništene, a vrijednosti vraćene u stanje prije početka transakcije
4. **čitaj**: unutar transakcije, program može čitati
5. **piši**: unutar transakcije, program također može pisati

a. ACID svojstva

Svojstva koja zahtijevamo od transakcije skraćeno možemo nazvati ACID svojstva. Kratica dolazi od riječi "atomicity"(nedjeljivost), "consistency"(konzinstentnost), "isolation"(izoliranost) i "durability"(trajnost). Pobliže ćemo objasniti navedene termine:

- **Nedjeljivost**: Ovo svojstvo se odnosi na ranije objašnjenu nedjeljivost transakcije
- **Konzinstentnost**: Transakcija mora ne smije kršiti ograničenja integriteta sustava. Tipični primjer je transakacija novaca u financijskom sustavu gdje pri prebacivanju novaca s jedan na drugi račun, ukupan zbroj mora ostati isti. Programer snosi odgovornost da poštuje ograničenja i očuva integritet sustava.
- **Izoliranost**: Transakcije su izolirane od učinaka istovremenih transakcija. Zahvaljujući tome, dobivamo dojam da su sve transakcije izvršene slijedno u nekom redoslijedu.
- **Trajnost**: Svojstvo koje se odnosi na pohranjivanje transakcije. Jednom kada je transakcija uspješno izvršena i pohranjena, njeni učinci moraju postati trajni, čak i ako je došlo do pogreške.

b. Kontrola konkurentnosti

Svojstvo izoliranosti transakcije još nazivamo serijazibilnim stanjem. Konkurentna povijest H transakcija T_1, T_2, ... , T_n je serijzibilna ako je ekvivalentna serijskoj povijesti. Kao primjer, uzmimo dvije transakcije: T_1 i T_2. T_1 je transakcija prebacivanja 100$ s računa A na račun B, a T_2 je transakcija prebacivanja 200$ s računa B na račun C. Ako pretpostavimo da je svaki račun prije početka ovih transakcija imao 1000$, konačna stanja na računima A, B i C će slijedno biti jednaka 900$, 900$ i 1200$. Pseudokod za T_1 će glasiti:

```
započni_transakciju;
```

```
x = pročitaj(A);
     x = x - 100;
     zapiši x u A;
     x = pročitaj(B);
     x = x + 100;
     zapiši x u B;
završi_transakciju;
```

Pseudokod za T_2 će glasiti:

```
započni_transakciju;
y = pročitaj(B);
     y = y - 200;
     zapiši y u B;
     y = pročitaj(C);
     y = y + 200;
     zapiši y u C;
završi_transakciju;
```

Jasno je da sve konkurentne povijesti nisu serijazibilne. U gornjem primjer, pretpostavimo da se dogodio sljedeći slijed događaja:

```
     x = pročitaj(A);
     x = x - 100;
     zapiši x u A;
     x = pročitaj(B);
     y = pročitaj(B);
     y = y - 200;
     zapiši y u B;
```

```
y = pročitaj(C);

y = y + 200;

zapiši y u C;

x = x + 100;

zapiši x u B;
```

U ovom slučaju konačne vrijednosti bile bi 900, 1100 i 1200, što su očito pogrešni rezultati. Jedan od načina da osiguramo serijazibilnost bio bi zaključavanje cijele baze podataka koju koristimo tijekom transakcije. Ipak, to bi omogućilo samo slijedne povijesti.

Postoji praktičnija tehnika koja se često koristi i naziva se “two-phase locking”(dvofazno zaključavanje). Ova tehnika transakciju shvaća kao 2 faze: faza zaključavanja i faza otključavanja. U fazi zaključavanja, za koju se također koristi i naziv “faza rasta”, transakcija može samo zaključati objekte, a u fazi otključavanja, koja se ponekad naziva i “faza sažimanja”, transakcija može otključati objekte. Korištenjem ove tehnike transakcija T_1 bi bila implementirana kao:

```
započni_transakciju;

zaključaj(A);

x = pročitaj(A);

     x = x - 100;

     zapiši x u A;

     zaključaj(B);

     x = pročitaj(B);

     x = x + 100;

     zapiši x u B;

     otključaj(A);

     otključaj(B);

završi_transakciju;
```

c. Rješavanje neuspjeha

Postoje dvije glavne tehnike za rješavanje neuspjeha, “private workspace”(privatni radni prostor) i “logging”(evidentiranje).

U prvom pristupu koji koristi privatni radni prostor pri čemu transakcija ne mijenja originalnu primarnu kopiju. Svi objekti na koje transakcija ima učinak su pokranjeni u posebnoj kopiji koju nazivamo kopija sjena. Ako prekinemo transakciju, tada se privatna ili sjena kopija odbacuju i ništa nismo izgubili. Ako dođe do commita i transkaciju treba trajno pohraniti, tada sve kopije sjene postaju primarne kopije. Ova tehnika je drugačija od tehnike “nonblocking synchronization”. Ako koristimo privatni radni prostor, ne kopiramo cijelu bazu prije početka transakcije. Umjesto toga, kopiramo samo one objekte ili stranice koje je transakcija promijenila. Ovu tehniku možemo implementirati na sljedeći način. Pretpostavimo da objektima možemo pristupiti samo preko njihovih pointera u indeksnoj tablici. Neka je S primarni indeks tablice objekata. Na početku transakcije, stvorena je kopija ove tablice i označena s S'. Sve operacije čitanja i pisanja se odvijaju preko tablice S'. Pošto čitanje ne mijenja objekte, obje tablice tada pokazuju na isti objekt pa sve operacije čitanja mogu ići preko primarne kopije. Kod operacije pisanja, stvorena je nova kopija objekta i pointer u tablici S' tada pokazuje na novu kopiju. Ako dođe do prekida transakcije, odbacujemo tablicu S'. U suprotnom, tablica S' postaje primarna tablica. Uočite da ova shema zahtjeva zaključavanje objekata kako bi transakcija bila serijazibilna.

U metodi evidentiranja, sve promjene se odbijaju na jednoj kopiji. Čuva se popis svih zapisa kako bi se u slučaju neuspjeha možemo se vratiti u evidenciju i poništiti sve dosad obavljene operacije. Uzmimo kao primjer operaciju koja mijenja vrijednost objekta x sa 5 na 3. Tada je u evidenciji zapisano da je x promijenjen sa 5 na 3. Ako dođe do prekida transakcije, tada je lako poništiti ovu operaciju.

3. "Two-phase commit" protokol

d. Distribuirani commit

Kada se transakcija odvija na više lokacija, zahtijevamo da ili svi prihvate transakciju i pohrane ju ili ju svi odbace. Ovaj problem nazivamo "distributed commit" problem. Problem je jednostavan kada nema neuspjeha. Sada ćemo se posvetiti rješavanju problema u slučaju neuspjeha. Pretpostavimo da su sve veze pouzdane.

Imamo sljedeće zahtjeve:

- **Usuglašenost**: Nijedna dva procesa(uspješna ili neuspješna) neće imati različit rezultat transakcije.
- **Valjanost**: Ako bilo koji proces počne sa "abort", tada je to jedini moguć konačan rezultat. Ako svi procesi započnu sa "commit" i nema neuspjeha, tada je to jedini moguć rezultat.
- **Slab završetak**: Ako ne dođe do neuspjeha, svi procesi će u konačnom vremenu donijeti odluku.
- **Non-blocking**: Svi procesi koji nisu neispravni će u konačnom vremenu donijeti odluku.

e. Algoritam

Sada definiramo "Two-phase commit" protokol koji zadovoljava prva tri uvjeta. Koraci protokola su:

- Koordinator šalje *request* poruku svim sudionicima.
- Pri primitku request poruke, svaki sudionik odgovara sa "yes" ili "no" porukom. "Yes" poruka označava da sudionik može izvršiti commit svih operacija koje su se odvile na njegovoj lokaciji. Ovime završava prva faza algoritma.
- Koordinator čeka da primi poruke od svih sudionika. Ako mu svi odgovore potvrdno sa "yes", koordinator tada šalje *finalCommit* poruku. U suprotnom, šalje *finalAbort* poruku.
- Sudionici provode operacije ovisno o poruci koju su primili od koordinatora.

f. Opis

Na temelju toga vidimo da postoje 2 faze: faza glasanja i faza odluke. U fazi glasanja koordinator prikuplja sve glasove, dok u fazi odluke koordinator obznanjuje konačnu odluku svim sudionicima.

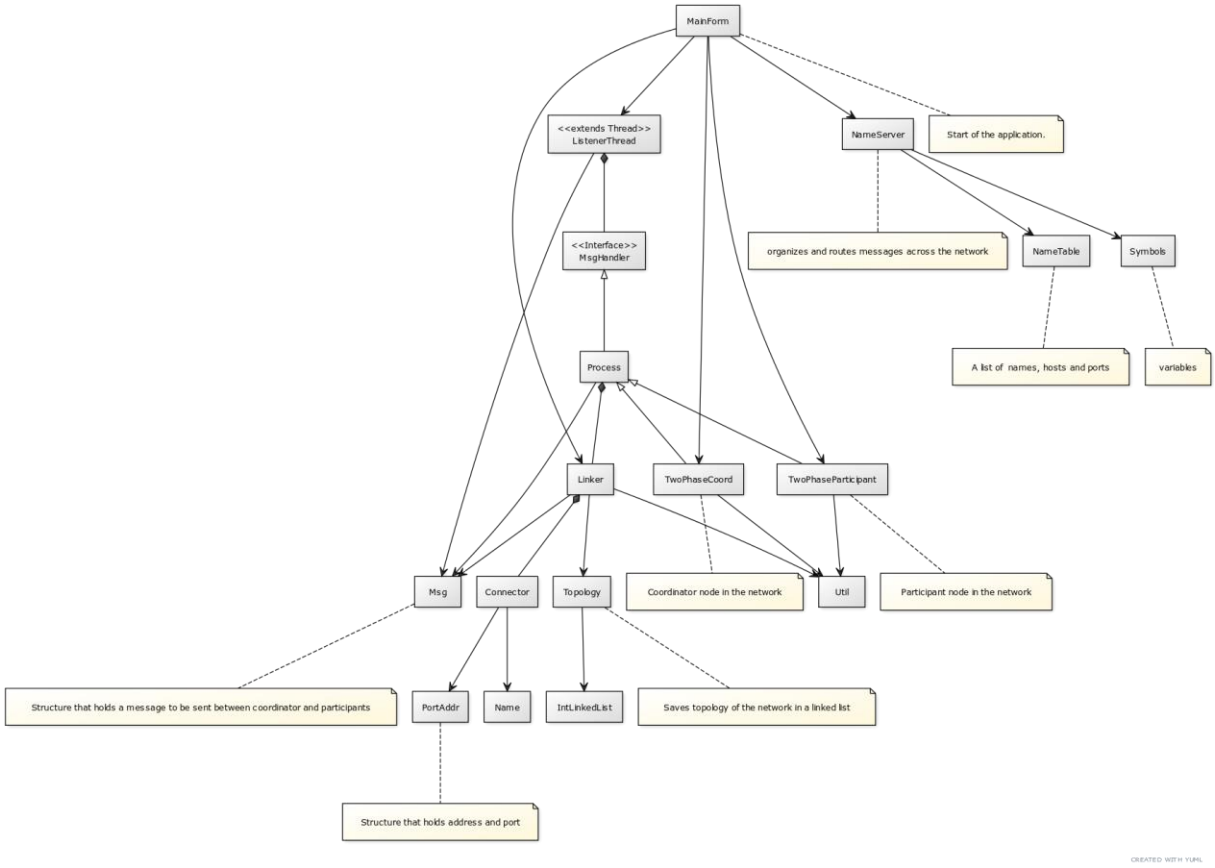
g. Svojstva algoritma

Analizirajmo sada protokol iz perspektive koordinatora. Ako ne dobije odgovor ni od kojeg sudionika u prvoj fazi, tada koordinator može prekinuti i napustiti cijelu transakciju. Posljedično, ako sudionik doživi grešku prije nego je poslao svoj glas u fazi glasanja, lako razriješimo neuspjeh do kojeg je došlo. Što ako do greške kod sudionika dođe nakon što je on poslao svoj glas? Budući da se transakcija možda već izvršila kao *commit*, kada se proces oporavi on mora saznati u kojem stanju je transakcija i primijeniti sve promjene koje su se desile. Zahvaljujući tome, uviđamo da je sudionik mogao poslati potvrdnu poruku "yes" samo ako može izvršiti sve potrebne promjene i uz pojavu greške. Drugim riječima, sudionici moraju voditi evidenciju u svojoj stabilnoj pohrani sa potrebnim informacijama za *commit* transakcije.

Analizirajmo sada protokol iz perspektive sudionika. Inicijalno očekuje *request* poruku koja možda ne dođe unutar intervala unaprijed predviđenog za timeout. U ovom slučaju, sudionik jednostavno može poslati negativan odgovor "ne" koordinatoru i pretpostaviti da je transakcija globalno prekinuta i napuštena. Koordinator također može imati grešku i u drugoj fazi. Što ako je sudionik odgovorio potvrdno sa "yes" porukom u prvoj fazi i nije dobio odgovor od koordinatora u drugoj fazi? U ovom slučaju, sudionik ne zna je li koordinator izvršio commit transakcije globalno. Tada bi sudionik trebao pitati druge sudionike o konačnoj odluci. Ako je koordinator doživio grešku nakon slanja poruke *finalCommit* ili *finalAbort* bilo kojem sudioniku koji nije doživio grešku, tada će svi sudionici saznati konačnu odluku. Ipak, to ne rješava slučaj u kojem je koordinator doživio grešku prije nego je informirao bilo kojeg sudionika(ili ako je sudionik kojeg je obavijestio također doživio grešku). U ovom slučaju, svi sudionici nemaju izbora nego čekati da se koordinator oporavi. To je razlog iz kojeg se *two pahse commit* protokol naziva i *blocking*.

4. Implementacija

a. UML



[Potpuni UML dijagram se može naći ovdje.]

1. Klasa "TwoPhaseCoord"

Algoritam za koordinatora je predstavljen klasom *TwoPahseCoord*. Koordinator poziva metodu *doCoordinator()* kako bi izvršio protokol . U prvoj fazi, koordinator čeka sve dok zastava *donePhase1* ne postane istinita. Ova zastava postaje istinita nako što su svi sudionici odgovorili sa potvrdnom porukom "yes" ili nako što je jedan od sudionika odgovorio negativno sa porukom "no". Ove poruke se obrađuju u metodi *handleMsg* gdje se onda prikladno poziva funkcija *notify()*.

2. Klasa "TwoPhaseParticipant"

Algoritam za sudionika je predstavljen klasnom *TwoPhaseParticipant*. Sudionik implementira inteface za usuglašavanje metodama *propose* i *decide*. Kada sudionik pozove metodu *decide*, sudionik je blokiran dok ne zaprimi poruku sadržaja *finalCommit* ili *finalAbort* od

koordinatora. Nismo još prikazali što koordinator i sudionici moraju poduzeti kada dođe do *timeouta* pri čekanju poruka. Također nismo još pokazali koje korake proces mora poduzeti pri oporavku od neuspjeha. [nadopuni, problem 16.6]

3. Klasa "Process"

Riječ je o klasi koja nastaje dogradnjom klase Linker. Implementira sučelje MsgHandler, no ne i sučelje Lock. Uvodi se nekoliko metoda za slanje poruka. Primanje poruka je slično kao kod Linker. Metoda handleMsg() za obradu poruke je prazna. Bilo koja klasa koja proširuje Process mora redefinirati tu metodu u skladu sa svojim potrebama. Metoda myWait()će se koristiti u svim implementacijama lokota za čekanje dok se lokot ne oslobodi. Metode myWait() i handleMsg()su sinkronizirane, tj. ne mogu se izvoditi unutar istog objekta u isto vrijeme. Očekuje se da će handleMsg() prekinuti čekanje dretve koja je pozvala myWait().

4. Klasa "Topology"

Prvi korak u povezivanju je učitavanje „topologije" (strukture povezanosti) mreže. To radi metoda readNeighbors( ) iz klase Topology. Pretpostavlja se da je lista susjeda od Pj, dakle procesa koji su s Pj povezani izravnom komunikacijskom vezom, zapisana na lokalnom disku u datoteci s imenom oblika topologyj. Ako takve datoteke nema, pretpostavlja se da su svi drugi procesi susjedi od Pj. Klasa Topology koristi pomoćnu klasu Util .

5. Klasa "Connector"

Služi za uspostavljanje utičnica između procesa. S obzirom da procesi mogu krenuti u različita vremena i na različitim lokacijama, koristimo se uslugama poslužitelja imena NameServer. Na početku rada svaki proces ubacuje svoje podatke u NameServer, a kasnije čita podatke koje su ubacili drugi procesi. Na taj način, procesi uz pomoć NameServera lociraju jedan drugoga. Podaci koje proces Pj ubacuje u NameServer oblikuju se na sljedeći način. Ime procesa Pj gradi se kao bazno ime (ime aplikacije) prošireno s j. Bazno ime u pravilu je zadano kao argument naredbenog retka kojim je Pj bio pokrenut. Ime računala na kojem Pj radi doznaje se pozivom statične metode InetAddress.getLocalHost( ) te zatim pretvorbom dobivene IP adrese u ime računala. Redni broj porta za Pj računa se po formuli: (port od NameServera) + 10 + j. Formula osigurava da svi procesi, uključujući i NameServer, „slušaju" na različitim portovima. To je potrebno onda kad se procesi izvode na istom računalu. Postupak uspostavljanja veza između procesa reguliran je metodom connect( ). Proces Pj najprije stvara poslužiteljsku utičnicu koja mu treba da bi slušao zahtjeve za spajanjem od procesa s manjim rednim brojevima. Zatim Pj kontaktira NameServer i ubacuje svoje podatke u njegovu tablicu. Svi procesi s manjim rednim brojevima prvo čekaju da se podaci za Pj pojave u NameServeru, a nakon toga kao klijenti šalju prema Pj zahtjev za spajanjem. Nakon što je Pj (u ulozi poslužitelja) uspostavio TCP veze sa svim manje numeriranim procesima, on se (sada kao klijent) pokušava spojiti s više numeriranim procesima.

5. Primjer pokretanja

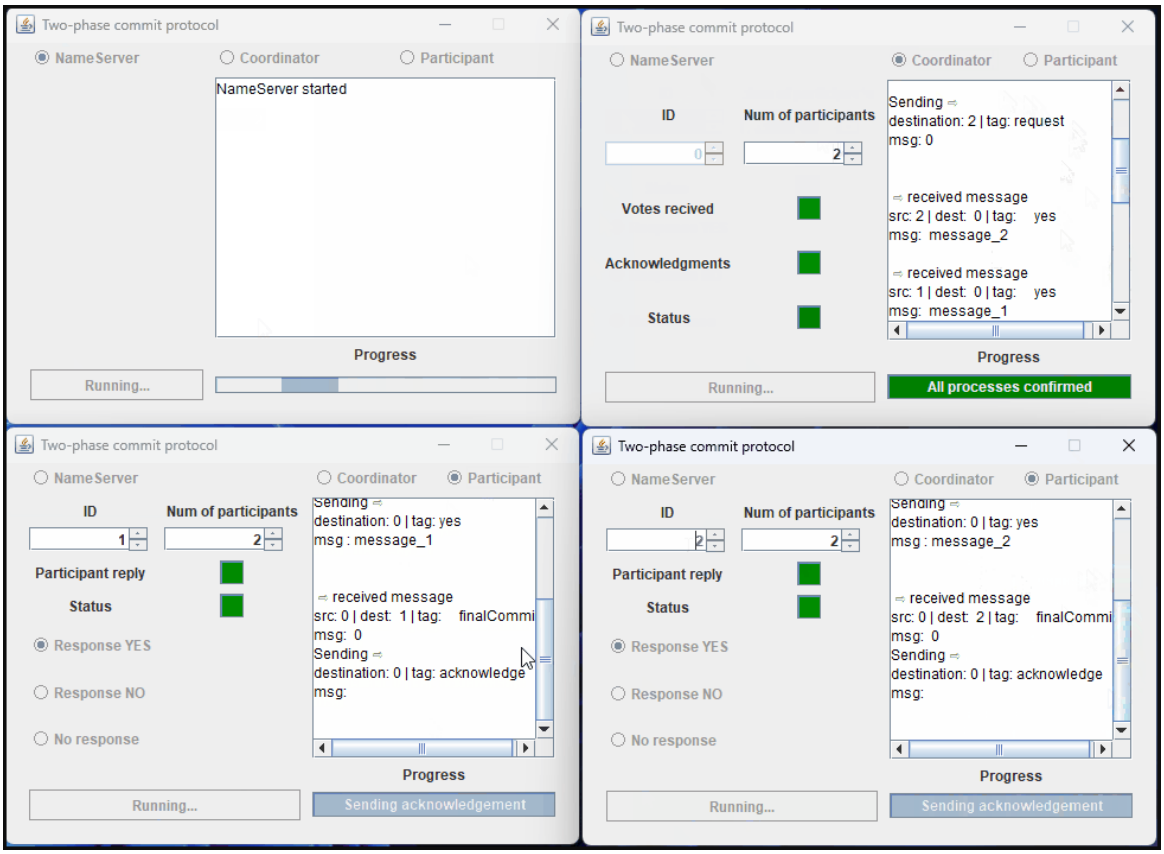
Program se pokreće s:

> java -jar ./transactions_java.jar <numOfParticipants>

Gornja naredba otvara **<numOfParticipants>** + 2 prozora, jedan za NameServer i još jedan za koordinatora.

Ako se nedaje nikakav argument, tada se otvori samo jedan generičan prozor koji se može koristiti kao name server, koordinator ili kao participant.

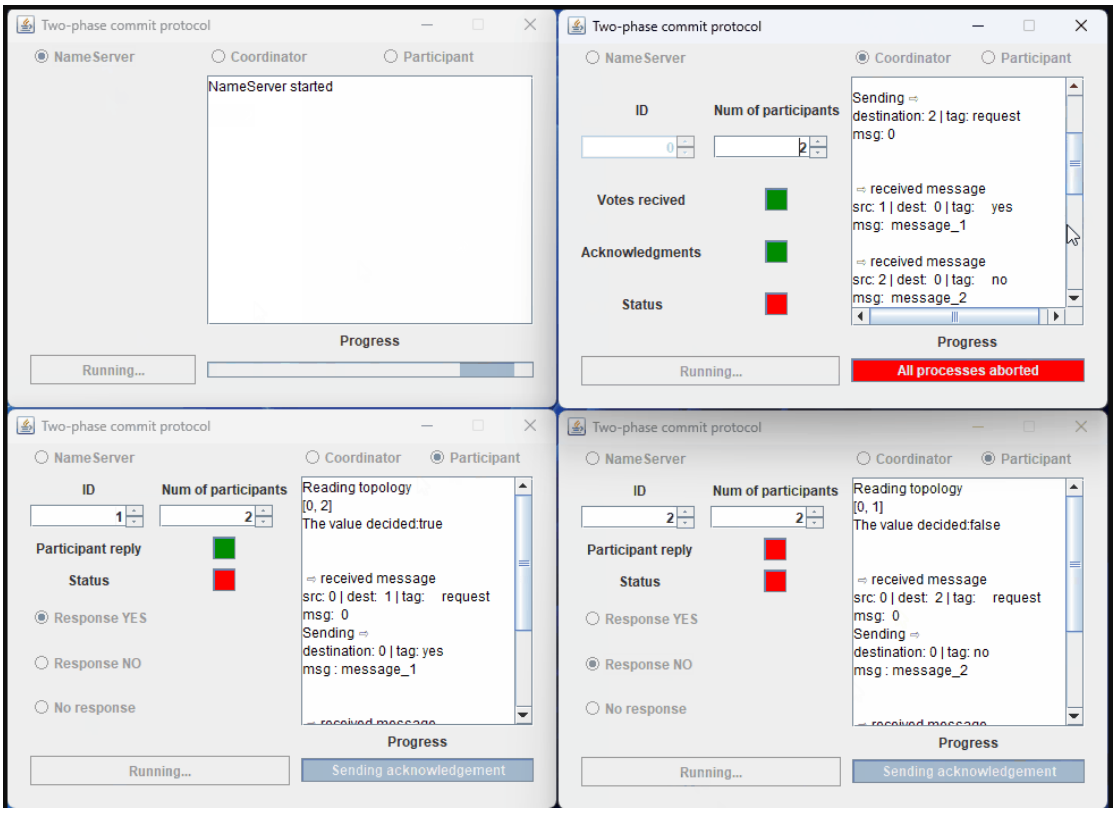
a. Primjer uspješnog "commit"-a



Prvo se pokrene Name Server, tada Coordinator te 2 Participanta. Oba participanta vraćaju poruku "YES" coordinatoru, te se u primjeru vidi da je commit bio uspješan.

Animacija uspješnog commita se može tu naći.

b. Primjer neuspješnog "commit"-a – rollback



Prvo se pokrene Name Server, tada Coordinator te 2 Participanta. Prvi vraća poruku "YES" coordinatoru,ali drugi vrati "NO", te se u primjeru vidi da je commit bio neuspješan,pa su **oba** participanta dobila naredbu da učine "rollback".

Animacija neuspješnog commita se može tu naći.

Literatura

- Garg V.K. Concurrent and Distributed Computing in Java Wiley – IEEE Press CHAPTER 16
- Robert Manger Distribuirani procesi