Simulazione-Transazioni

Progetto del corso di Sistemi Operativi 2021-2022
A cura di: Eduard Antonovic Occhipinti, Francesco Mauro, Riccardo Oro

---

Master

- Inizializza gli oggetti IPC sfruttando il parser[1] per leggere il file di configurazione e salvarne i valori in una struct di parametri
- Inizializza il signal handler al fine di poter usare il `SIGINT` per interrompere la simulazione in maniera pulita e a proprio piacimento usando la shortcut `CTRL-C`
- Crea `SO_NUM_NODES` *nodes* usando una fork; il *children* generato assegna a ciascun nodo una coda di messaggi personale che utilizza il PID stesso del nodo come key ed effettua una *execve* a *./nodes* passandogli come *arguments* gli ID degll oggetti IPC generati in precedenza
- Crea una lista amici per ciascun nodo, la quale verrà inviata un amico alla volta sulla coda di messaggi corrispondente
- Crea `SO_NUM_USER` *users* usando una fork come per i nodi e facendo una *execve* a *./users*, passandogli come *arguments* gli ID degll oggetti IPC
- Imposta un *alarm()* a `SO_SIM_SEC`, il `SIGALRM` verrà gestito allo stesso modo del `SIGINT`
- Effettua una *fork()*
  - il master inizia a mostrare a schermo ogni secondo il numero di processi attivi nonché quelli con bilancio più significativo
  - il figlio generato invece si metterà in ascolto per transazioni inviate sulla queue privata del *master* nella quale sono inviate le transazioni che non sono riuscite a trovare un *node* che le processasse entro `SO_HOPS`, quando ne riceve una crea un nuovo nodo (fino ad un massimo di `SO_NODES_NUM` nodi extra), nel caso ci siano giá troppi nodi extra generati la transazione viene semplicemente scartata[2]

Nodes

- Attacca la propria coda usando come key il proprio PID
- Attacca gli oggetti IPC a partire dagli ID che vengono passati come arguments
- Inizializza un `SIGINT` handler che rimuoverà la coda associa al *node* alla terminazione
- Si mette in ascolto per riempire la propria lista amici, il processo non può partire finché il master non gli invia tutti i *friends*
- Inizializza la *transaction pool* come *linked list*
- Entra in un *loop* nel quale inizia il *fetch* dei messaggi inviati sulla propria *queue* aspettando di ricevere una transazione da aggiungere alla *transaction pool*[3], dopodiché controlla -con flag `IPC_NOWAIT`- se sono presenti nuovi *friends* da aggiungere alla lista di amici, in caso affermativo realloca l'array di friends in maniera tale da poter far posto a un nuovo *node* amico.
  - Periodicamente[4], e se la *transaction pool* è piena, la transazione che è presente nella *pool* da più tempo viene inviata a un amico scelto casualmente, se invece la transazione ha già esaurito gli *hop* dispnibili, questa viene inviata al *master*, il quale si occuperà di creare un nuovo nodo.
- Dopodiché il nodo controlla se ha abbastanza transazioni per creare un nuovo blocco, in caso affermativo effettua una *fork()*
  - il figlio crea e si occupa di generare e confermare le transazioni del nuovo blocco, dopodiché effettua una *sleep()* che simula l'elaborazione dei blocchi
  - il padre ricomincia il loop in maniera tale da poter continuare a fare il *fetch* dei messaggi in parallelo con l'elaborazione del blocco

---

Users

- Attacca gli oggetti oggetti ipc a partire dagli ID passati come arguments
- Inizializza il `SIGINT` handler che aggiornerà il bilancio del nodo in maniera tale da avere una rappresentazione più accurata del valore a terminazione[5]
- Inizializza il `SIGUSR1` handler che permette di inviare transazioni a comando
- Entra in un loop nel quale calcola il proprio bilancio a partire dalle transazioni confermate sul ledger sottraendo quelle ancora in uscita, se questo é ≥ 2:
  - imposta il proprio stato ad *alive*, estrae il PID di un *user* casuale e il PID di un *node* casuale e genera una transazione di *amount* compreso tra i valori 2 e *current balance* e la invia allo *user* estratto sfruttando il *node* estratto, se non riesce ad inviare la transazione generata diminuisce il valore di retry, nel caso in cui questo arrivi a 0 cambia status in *dead* e termina la propria esecuzione. A prescindere dalla riuscita dell'invio di una transazione lo *user* a questo punto effettua una *sleep()*[6]
  - se il bilancio é minore lo stato viene impostato a *broke*

IPC Objects

- Tante code di messaggi quanti sono i nodi più una, che è quella del *master* per ricevere le transazioni scartate per insufficienti *hops*
- In Shared memory un array contenente i parametri di ciascun *node* e uno contenente i parametri di ciascun *user*, il ledger[7] e una struct contente tutti i parametri della simulazione, letti dal parser
- 2 semafori
  - Uno per gestire la scrittura su *nodesPID* e *userPID*[8]
  - Uno per gestire la scrittura sul ledger

Scelte Implementative

> Il codice preso da lezioni o fonti esterne è stato opportunamente segnalato

common.h

É un *header* dove sono definite la maggior parte delle macro e delle *struct* usate all'interno del progetto

Funzioni di stampa

Tendenzialmente sono raggruppate in un unico file "print.c" con corrispettivo *header* per diminuire il *clutter*.
Le funzioni di stampa si occupano di mostrare a schermo attraverso le apposite funzioni informazioni di output, lo stato delle transazioni, i *timestamp*,numero di processi nodo e user attivi in quell'istante, ecc...
L'utilizzo di stampe a lunghezza fissa (i.e %10lu) è stato necessario per avere tutto allineato correttamente e l'utilizzo delle ANSI Escape Sequences ci ha permesso di avere una print più pulita e facile da leggere.
Il ledger viene stampato a fine simulazione sul file "ledger.txt"

Makefile

Il progetto può essere compilato con "make debug" per abilitare le print aggiuntive stampate tramite la macro TRACE su standard error (altamente consigliato redirezionare lo standard error con "./master 2> log.txt" perché è molto verboso), *master*, *users* e *nodes*, possono essere compilati separatemente

Directory "Utils"

Contiene codice "utlity" che viene compilato in file oggetto, successivamente linkati a *nodes users* e *master* dal makefile

Problemi noti

Il bilancio totale a fine simulazione è leggermente diverso da quello iniziale, il problema potrebbe essere dovuto alle approssimazioni che avvengono calcolando tutto come int oppure nel modo in cui calcoliamo il bilancio dei *nodes*.
Da una veloce analisi ci risulta che sia tendenzialmente $< 0.1\%$

---

1. implementazione molto semplice che "tokenizza" il testo in [string][unsigned long] e confronta poi le stringhe per assegnare il valore corrispondente al paramtero letto come stringa ↩

2. Utilizzando un sistema più complesso rispetto alla shared memory di linux si potrebbe allocare in maniera dinamica lo spazio necessario per memorizzare le informazioni relative ai nodi extra in maniera tale da rimuovere il limite massimo di *nodes* extra. Il valore scelto, pari a quello di `SO_NODES_NUM` è puramente arbitrario↩
3. Questa implementazione causa deadlock nel caso in cui due amici vengano inviati uno dietro l'altro sulla coda di messaggi, una maniera per risolvere è di settare la flag `IPC_NOWAIT` anche alla *msgrcv* delle transazioni ma così facendo le performance peggiorano considerevolmente. Il rischio che accada è estremamente basso, l'unico caso realistico in cui potrebbe accadere è quello nel quale tutti gli *user* sono *broke* e il master crea ed invia allo stesso *node* due amici ma è un *edge case* abbastanza estremo (mai capitato nella pratica) e dovuto al fatto che periodicamente bisogna inviare una transazione ad un amico a prescindere da quante se ne hanno nella *transaction pool*↩
4. ogni 20 check ma il numero è completamente arbitrario↩
5. a questo punto le transazioni nella *linked list* "outGoingTransactions" possono essere considerate come scartate quindi facciamo che ri-sommarle al bilancio totale, gli utenti potrebbero risultare come *broke* nella print finale anche se il calcolo finale li porta ad avere un numero di $UC \geq 2$, questo comportamento è voluto al fine di rappresentare lo stato in cui gli utenti dovrebbero trovarsi alla fine della simulazione se questa venisse "congelata", non sarebbe rappresentativo avere una print piena di *users* "*alive*" quando invece il programma poco prima era fermo con *users* senza budget che aspettavano che un nodo processasse le transazioni che avevano inviato↩
6. utilizziamo una *clock_nanosleep()* anziché la *nanosleep()* per via di un bug segnalato su quest'ultima nella *man page*, il tempo rimasto ancora della sleep viene salvato in una struct separata in maniera tale che questo venga completato anche se lo *user* viene forzato ad inviare istantaneamente una transazione con `SIGUSR1`↩
7. AKA "Libro Mastro", un array di "blocchi" (struct definita in common.h) che contiene tutte le transazioni confermate. Avremmo potuto definire una struct apposta per il libro mastro in maniera tale da tracciare più velocemente la dimensione ma ci è sembrato superfluo, se fosse fattibile gestirlo come lista linkata definire una struct separata avrebbe molto più senso↩
8. array di struct "node" e "user", rispettivamente, definite in common.h e che raccolgono informazioni relative a ciascun processo↩