



REPORT PROGETTO "RAFT SHOP"

GRUPPO: I 3 JOLIE

MEMBRI: RIGHI MASSIMO

BOTA VINCENTIU

GAMBERINI JACOPO

IMPLEMENTAZIONE GENERALE DEL PROGETTO

Abbiamo deciso di suddividere l'implementazione del progetto in quattro fasi:

- 1 – Dialogo client-server
- 2 – Implementare un Modello Semplice, ovvero un sistema composto da 1 client, 1 server, 1 administrator e 1 network visualizer
- 3 – Implementare la comunicazione tra 5 server, ovvero il protocollo raft, separata dal modello semplice
- 4 – Assemblare le varie componenti per creare il Raft Shop

1) COMUNICAZIONE CLIENT-SERVER

Nell'implementazione della comunicazione tra server e client, facciamo riferimento principalmente a due variabili globali: la lista delle Merci disponibili nello Shop e la lista dei carrelli che vengono creati dai client. Successivamente abbiamo preso in considerazione la lista dei carrelli archiviati, ma le due variabili centrali all'interno del sistema sono le due sopra citate.

Una delle operazioni critiche è stata quella per l'aggiunta di una quantità di un prodotto all'interno della lista dei prodotti di un carrello specificamente nel caso in cui il prodotto era già presente nella lista dei prodotti del carrello.

Il problema sorgeva dal fatto che quando cercavamo di aggiornare la quantità del prodotto, la somma tra le due quantità veniva vista da Jolie come una concatenazione di stringhe. Es:

Nel carrello è già presente una quantità di banana -> 10

Utilizziamo l'operazione di aggiunta di banana -> 5

Risultato: banana -> 105

Dopo ripetute prove abbiamo trovato il modo di superare questo ostacolo: utilizzare la funzione primitiva di Jolie "**int()**" nel momento in cui vado a ricevere in input da tastiera la quantità di prodotto da aggiungere, convertendo così la stringa in ingresso nel suo corrispondente intero.

Quando si utilizzano le funzioni primitive di Jolie che permettono l'inserimento di un input da tastiera (es: **registerForInput@Console()**) e la seguente oneway **in()**), che l'input sia un numero o una parola, queste funzioni convertono automaticamente l'input in un dato di tipo stringa. Chiaramente questo modus operandi lo abbiamo esteso a tutte le altre operazioni che richiedevano l'inserimento di un intero da tastiera.

Inizialmente abbiamo fatto prove solamente di aggiunta di prodotti all'interno di un carrello, senza preoccuparci e tener conto della disponibilità di quel determinato prodotto all'interno dello Shop. Stessa cosa per l'eliminazione, poichè una volta eliminata la quantità di prodotto all'interno del carrello (avendo prima effettuato i dovuti controlli) non ci preoccupavamo di dover riaggiungere questa quantità all'interno dello Shop.

Dopo aver perfezionato e capito il meccanismo, abbiamo allora considerato anche la relativa aggiunta/eliminazione dei prodotti all'interno dello Shop.

La lista delle merci veniva caricata automaticamente dal Server al suo avvio, poichè come detto prima, non avevamo ancora considerato il dialogo con l'Administrator e la sua esistenza.

Per quanto riguarda l'architettura del sistema invece, per prima cosa abbiamo cercato di creare una struttura di dialogo composta da 1 solo Client e il Server. Quando siamo riusciti a far funzionare questo sistema, abbiamo esteso il tutto ad una comunicazione concorrente tra vari client che andavano ad agire su quello stesso server.

Per implementare ciò abbiamo inserito all'interno della struttura del client il comando chiave **"execution{concurrent}"** che ci ha permesso di creare più istanze del client.

2)IMPLEMENTAZIONE MODELLO SEMPLICE

Dopo aver testato con successo il funzionamento della parte precedente, siamo passati ad uno sviluppo più ampio in cui venivano inserite le componenti dell'Administrator e del Network Visualizer.

ADMINISTRATOR

Un'altra difficoltà nell'implementazione di questo progetto è stata quella di capire effettivamente che ruolo aveva l'administrator all'interno del sistema, ovvero non capivamo inizialmente se quando un client voleva aggiungere un prodotto nel carrello e dialogava col server, il server aggiornava la propria lista merci e inviava l'aggiornamento all'administrator oppure se il server girava il comando di acquisto all'administrator ed esso a sua volta eseguiva questa operazione di decremento della merce sulla lista delle merci contenuta nel server.

L'interpretazione logica ci ha portato a pensare l'Administrator come il proprietario di un negozio ortofrutticolo (la merce all'interno dello Shop è costituita da frutta e verdura) che in "quasi" totale autonomia dalle scelte effettuate dai clienti può decidere di esporre altra frutta/verdura fresca all'interno delle casse in vendita oppure togliere dall'esposizione della frutta/verdura deteriorata. Prima l'autonomia del negoziante è stata definita "quasi" totale poichè chiaramente uno dei vincoli è che il negoziante non può togliere una quantità di merci non presente o superiore a quelle esposte. All'avvio dell'Administrator, vengono caricati automaticamente all'interno del server 4 prodotti con le relative quantità.

```
Ho ricevuto da Administrator:4 prodotti
Banana-50
Cocomero-50
Carota-50
Insalata-50
```

(Terminale del server nel momento in cui riceve i prodotti dall'Administrator)

Nell'operazione **aggiungiProdotto**, abbiamo deciso che, nel caso in cui l'Administrator volesse aggiungere un prodotto non ancora presente nella lista dei prodotti, acconsentivamo l'aggiunta, aggiungendo il prodotto alla fine/in coda alla lista merci.

NETWORK VISUALIZER

Nelle specifiche del progetto, si faceva riferimento al fatto che il server leader inviava a cadenza costante una serie di informazioni sullo stato del sistema al Network Visualizer, che a sua volta aveva il compito di mostrarle a terminale.

Partendo da questa definizione, subito avevamo pensato di vedere il Network Visualizer come un piccolo server con un'unica operazione di request response, che veniva richiamata dal server inviandogli le variabili globali del sistema in esso contenute. Il Network, dopo aver ricevuto correttamente le variabili, le visualizzava su terminale e dava conferma della ricezione al server. Ma a un certo punto ha iniziato a sorgere in noi un dubbio: perché andare a creare un altro server? Il Network deve solamente visualizzare su terminale dei dati!!

A questo punto abbiamo ragionato al contrario, ovvero: dato che il Network Visualizer è solamente un tool Amministrativo di visualizzazione dello stato del sistema, ogni 4 secondi (sleep di 4000) gli abbiamo fatto inviare una richiesta ad una operazione, implementata questa volta specificamente sul server, in cui richiedeva tutte le variabili di sistema, le riceveva e le visualizzava con tre semplici for sul terminale. A questo punto il Network visualizzava a impulsi costanti:

- la lista dei carrelli disponibili con gli oggetti contenuti per ogni carrello
- la lista dei carrelli archiviati con gli oggetti contenuti per ogni carrello
- la lista delle merci presenti nel Raft Shop

Non si può negare che questa sia stata alla fine la parte più semplice dell'implementazione del progetto, poiché stavamo iniziando a prendere confidenza con il codice, le operazioni più complesse (ovvero quelle di aggiungi e cancella elementi all'interno del carrello) le avevamo già implementate nel client all'inizio e non dovevamo ancora gestire la comunicazione tra più server (Raft).

Alla fine quindi, tutto questo non era altro che una comunicazione tra 3 client (seppur richiedenti operazioni diverse) e un unico server.

3)IMPLEMENTAZIONE DEL PROTOCOLLO RAFT

Siamo arrivati alla parte più importante e complessa del progetto: il protocollo Raft.

Per prima cosa abbiamo deciso che la comunicazione tra i server e il loro comportamento tramite le singole operazioni andava diviso pesantemente all'inizio per evitare che si creasse confusione nell'implementazione dell'algoritmo. Per questo motivo abbiamo creato 5 nuovi file (uno per ogni server) totalmente separati dal modello semplice Administrator-Client-Server-Network Visualizer, e con una propria interfaccia di comunicazione (RaftInterface).

Siamo partiti dalla base, ovvero abbiamo pensato con quali porte di input/output doveva comunicare ogni server:

Input Port: 1 porta, quella che si riferisce al server stesso. E' l'unica porta tramite cui gli altri server possono comunicare con il server sotto esame.

Output Port: 5 porte, una per ogni server del Raft, tramite le quali il server può richiamare le loro operazioni. Perché 5 e non 4? Ce ne è una anche per lo stesso server? Perché deve/può comunicare anche con se stesso? Tra poco daremo questa spiegazione.

Dopo la creazione delle porte, ci siamo posti subito una seconda domanda: Come nasce il processo del server? Quanto dura?

Abbiamo deciso di gestire il processo con un for infinito, che mi andava a rappresentare la durata del processo del server (chiaramente infinita, ovvero fino a un eventuale crash del processo).

Mano a mano che proseguivamo con lo sviluppo del progetto, continuavamo a prendere sempre più confidenza con i costrutti e le strutture forniteci da Jolie, e come in questo caso abbiamo deciso di inserire il ciclo `for` all'interno di una struttura `define` chiamata **`infiniteLoop`**, che veniva poi richiamata direttamente all'interno dello scope `init` cosicché all'avvio del processo `server` sarebbe partito anche il ciclo infinito di `time alive`.

Implementando il codice in questo modo però ci siamo accorti successivamente che, richiamando direttamente l'`infiniteLoop` all'interno dell'`init`, esso veniva visto come un'istruzione che doveva essere terminata prima di poter uscire dallo scope. Perciù si rimaneva sempre in una situazione in cui eravamo infinitamente dentro all'`init` e non si poteva richiamare/effettuare nessun tipo di operazione all'interno del `server`.

Era chiaro che questa soluzione era infattibile, soprattutto data la necessità di poter richiamare operazioni all'interno del `server` da parte del client e dell'Administrator. Ma allo stesso tempo era anche obbligatorio far partire il ciclo infinito di `time alive` all'avvio del `server`.

Dopo varie prove e ragionamenti ci è sembrato sempre più chiaro che bisognava lavorare sul parallelismo tra lo scope dell'`init` e lo scope del `main`.

Per prima cosa però bisognava riuscire ad effettuare tutte le operazioni all'interno del `init` per poter uscire da esso e passare alle operazioni del `main`. Qua ci fu la svolta, poiché ricordammo che a lezione (e chiaramente se ne parlava anche sulla documentazione) eravamo rimasti a lungo sul meccanismo di funzionamento delle operazioni `one-way`, durante le quali ci era rimasto in presso una loro definizione, ovvero, un processo che utilizza un'operazione `one-way`, invia una richiesta e poi non attende la risposta alla sua richiesta prima di poter proseguire altre istruzioni come le operazioni `request-response`, ma va avanti e continua ad eseguire altre istruzioni, non preoccupandosi del momento in cui la sua richiesta verrà ascoltata.

Questo modo di agire si sposava perfettamente con il nostro caso.

Abbiamo creato un'operazione `one-way` chiamata **`startServer`**, definita, come tutte le altre, all'interno del `main`, ma con una sola e unica istruzione: il richiamo al `define` dell'`infiniteLoop`. Ma da chi sarebbe stata richiamata questa operazione? E proprio qua andiamo a rispondere alla domanda posta precedentemente sul perché abbiamo istanziato una porta di output anche per il `server` stesso. Adesso all'interno dell'`init` abbiamo inserito una chiamata all'operazione `startServer`, passandogli in input un simbolico tipo `void`, da parte proprio dello stesso `server`, ovvero un'auto-chiamata a se stesso.

In questo modo, il `server` fa una chiamata a se stesso per poi proseguire con l'operazione successiva, intanto alla ricezione della chiamata, fa partire il ciclo `for` infinito, ma essendo questo in parallelo con le operazioni del `main`, il tutto funziona.

Si riportano di seguito i due costrutti:

```
init{
    //Altre variabili

    println@Console( "Prima dell'invio" );
    startServer@ToServer1(t) //Richiamo l'operazione startServer che mi fa partire il processo del server (loop infinito).
}

main
{
    /*
    Questa operazione mi permette di avviare il processo del server.
    In essa è contenuto il ciclo infinito infiniteLoop che mi rappresenta la durata
    del processo del server.
    */
    [startServer( termineIdServer )] {
        //risposteServer
        println@Console( "Dentro startServer" );
        infiniteLoop
    }
}
```

Dopo aver avviato il server e constatato che l'implementazione del for era riuscita, ci siamo posti il problema di cosa avrebbe dovuto fare il server, ovvero quali operazioni richiamare, quali variabili aggiornare durante il ciclo di vita del suo processo. Ovvero abbiamo dibattuto su come e con cosa riempire il for.

Per prima cosa, analizzando sempre le specifiche, abbiamo visto che ogni server aveva come attributi personali il suo ID (da 1 a 5) e il suo termine, ovvero la suddivisione arbitraria del tempo (1,2,3,...). Infatti queste due variabili verranno scambiate/inviare molte volte a braccetto tra i vari server tramite il type **TermineIdServer**, molto importante nella realizzazione dell'algoritmo raft all'interno del nostro progetto.

Di seguito si visualizzano le variabili utilizzate per l'implementazione dell'algoritmo Raft:

```
init{
    //Inizializzo i semafori

    //Variabili per il raft

    //Variabile che mi rappresenta il tempo che scorre all'interno di ogni termine
    global.Timer = 0;
    //ID del server, ogni server ha un ID diverso
    global.id_Server = 1;
    //ID del server leader, all'inizio è settato a zero, dato che non c'è un leader
    global.id_ServerLeader = 0;
    //Tempo di Election Timeout, quando viene raggiunto dal Timer, si passa ad un altro termine
    global.TimeForNextTimeout = 0;
    //Setto/aggiorno un nuovo Election Timeout
    settaNuovoTimeout;
    global.termineIdServer.idS = global.id_Server; //1,2,3,...
    //Ogni termine inizia con un elezione nella quale i candidati cercano di diventare leader
    global.termineServer = 0;
    //Chiaramente al suo avvio il server è follower
    global.stato = 1;

    //Variabili per il dialogo con l'Administrator

    //Variabili per il dialogo con il Client
}
```

All'interno del ciclo for abbiamo effettuato due controlli in linea con la descrizione delle specifiche: uno per decidere quando inviare la richiesta di voto, e l'altro per decidere quando inviare l'heartbeat.

Anche in questi due casi abbiamo deciso di definire le procedure da effettuare all'interno dei due costrutti if tramite due strutture define implementate precedentemente.

- **sendRequestVote:** In questa procedura il server candidato invia agli altri 4 server una variabile di tipo TermineIdServer contenente, come detto prima, l'id del server e il termine corrente del server. Questa variabile composta verrà inviata in parallelo a tutti i 4 server richiamando l'operazione request-response **sendRequestVote** del server, ed ogni server, tralasciando i controlli che dovrà effettuare seguendo le specifiche, risponderà 1 se voterà per il server candidato, oppure 0 se non lo voterà. Per ogni voto favorevole, il server candidato incrementa la sua variabile **serverNumeroVoti**. Se all'interno di un elezione, la variabile serverNumeroVoti è ≥ 3 ovvero il server ha ottenuto la maggioranza, allora il server diventa leader. Dopo alcune prove abbiamo notato che se il server candidato cercava di comunicare con un server offline o caduto in crash venivano scaturite una serie di eccezioni, perciò abbiamo deciso di inserire il blocco di codice dentro il quale venivano effettuate le chiamate all'operazione sendRequestVote all'interno di uno scope controllato in cui venivano installate delle eccezioni che potevano essere scaturite.

- **sendHeartBeat:** Questa procedura è ancora più semplice di quella descritta precedentemente, poichè viene utilizzata dal server leader solamente per segnalare agli altri server che è ancora vivo. Quando un server riceve l'heartbeat del leader, diventa follower. Questa procedura, al momento dell'invio dell'heartbeat utilizza, come per la richiesta di voto, un'operazione di tipo request-response chiamata sendHeartBeat, che viene richiamata in parallelo per i 4 server ai quali viene inviata la solita variabile `termIdServer` che contiene gli attributi del leader. Anche in questo caso si è utilizzato uno scope in cui sono state installate eccezioni per gestire il problema della comunicazione con server offline.

Alla fine di ogni ciclo di `time alive` si incrementa la variabile `Timer`, che come detto prima, mi rappresenta lo scorrere del tempo all'interno del processo del server.

Per verificare la buona implementazione dell'algoritmo, abbiamo inserito alla fine di ogni ciclo una stampa di tutte le variabili interessate e scambiate durante l'algoritmo, ovvero: l'ID del server, il `Timer`, il prossimo `Election Timeout`, il termine corrente del server, lo stato del server (follower, candidate, leader), e per finire l'ID del server leader.

Nelle specifiche del progetto si consigliava: "Per evitare il possibile ripetersi infinito di termini vacanti, Raft usa degli `Election Timeout` casuali (ad ogni timeout, per ogni Server) entro un intervallo fisso, e.g., 150–300ms. Questo incrementa le probabilità che i timeout avvengano scaglionati, permettendo ad un solo Server di diventare Candidato, richiedere le elezioni e diventare Leader prima che un altro Server vada in timeout e richieda le elezioni a sua volta."

Ebbene siamo pienamente d'accordo con questa soluzione, ma nell'implementazione del progetto abbiamo deciso di definire degli `Election Timeout` per una durata da 1 a 13 in modo che si possa vedere e constatare con chiarezza il giusto funzionamento dell'algoritmo Raft, che sennò sarebbe stato di difficile lettura.

L'algoritmo Raft che abbiamo implementato è una versione più semplificata ma molto più efficace e snella di quella richiesta nelle specifiche.

I nostri dubbi sono derivati, come si può notare, dal concetto di replicazione del log. Poichè quando un client comunicava con il server leader, quest'ultimo doveva, tra le altre cose:

- salvarsi il comando,
- aggiungerlo al proprio log come una nuova voce (composta, come da specifiche, da 3 sezioni),
- inviare a tutti i Followers in parallelo il comando di `AppendEntries` per replicare la nuova voce,
- aspettare che il log venga replicato con successo sugli altri server, per poi eseguire il comando
- tenere traccia di due informazioni: 1) l'indice dell'ultima voce eseguita (i.e., giudicata correttamente replicata) e 2) il termine e l'indice della voce nel log immediatamente precedente alla nuova voce da aggiungere.

Questo modello sarebbe andato ad appesantire fortemente il nostro sistema.

Tutto ciò invece diventa molto più snello ed efficiente con un'unica operazione di sincronizzazione, ovvero il server leader riceve il comando dal client, lo esegue, e prima di inviare la risposta/conferma al client, aggiorna le variabili globali degli altri server follower, inviando loro in parallelo il comando di sincronizzazione. In pochi passaggi otteniamo il risultato che avrebbe dato un algoritmo Raft che facesse uso del log.

4)MERGE TOTALE



Terminata l'implementazione dell'algoritmo Raft, siamo passati alla fase finale, ovvero la fusione tra il modello semplice Administrator-Server-Client-Network Visualizer e il meccanismo di comunicazione tra i 5 server.

Per prima cosa abbiamo ricopiato tutte le operazioni che forniva il server del modello semplice in ognuno dei 5 server, dato che fino ad allora i server avevano solamente implementato le operazioni utilizzate nell'algoritmo Raft. Dopodichè ci siamo imbattuti nel primo dei due ostacoli che emergeranno in questa fase di progettazione: la comunicazione tra il Client e il server leader.

Poichè inizialmente il client che voleva effettuare un'operazione dialogava sempre con lo stesso client, che le gli permetteva sempre l'operazione. Ma ora che abbiamo 5 server, il client in un certo senso deve dialogare sempre con lo stesso server, ovvero sempre con il leader, ma il leader può cambiare ed essere uno dei 5 e quindi l'indirizzo può cambiare!

In linea con le specifiche bisognava modificare le operazioni dei server in modo che se un client avesse cercato di contattare un server non leader esso gli avrebbe risposto non eseguendo l'operazione, bensì restituendogli l'indirizzo (in questo caso l'ID) del server leader.

Abbiamo così pensato di inserire all'interno dei tipi ritornati dalle operazioni dei server, una nuova variabile, denominata **status**, che avrebbe contenuto l'ID del server leader qualora il server sotto esame non lo fosse stato, oppure un numero diverso da 1,2,3,4,5, (nel nostro caso 100) che mi avrebbe indicato che il server con cui il client stava dialogando era il leader, e che quindi l'operazione era stata effettuata.

Immediatamente dopo la prima prova, ci è apparsa la necessità di effettuare un'ulteriore modifica, questa volta però nel client, poichè il server che invocavamo (partivamo sempre dal server 1), se non era leader dava sì il messaggio di avviso e restituiva l'ID del server leader, ma non effettuava più ulteriori chiamate ai server. Questo problema ci ha portato a ripensare il meccanismo di richiesta di un'operazione da parte del client.

Abbiamo pensato ad una struttura in cui il client inviava richieste in continuazione, finchè non avesse trovato il server leader (ovvero finchè la risposta non fosse stata 100). Chiaramente una struttura while si sposò perfettamente con le nostre intenzioni.

Per l'implementazione abbiamo utilizzato un'altra variabile chiamata **currentSending**, che avrebbe contenuto il numero del server a cui il client avrebbe inviato la successiva chiamata.

All'inizio currentSending aveva valore 1, poichè partivamo ad effettuare le richieste dal server 1,

mentre la variabile status partiva da -1, dato che non sapevamo chi era il leader..

All'interno del ciclo while implementammo una serie di if a cascata che in base al valore del currentSending giravano la richiesta ad uno dei 5 server. Finchè il server richiamato non sapeva chi era il leader rispondeva con status = 0 e il currentSending si incrementava di 1 e si effettuava un altro controllo con gli if a cascata, mentre se il server richiamato era a conoscenza del leader, rispondeva con il suo ID, per cui si settava immediatamente il valore di currentSending = status per poi inviare la richiesta al server leader.

Chiaramente siamo arrivati a questa soluzione solamente dopo qualche aggiustamento, come per esempio l'inserimento di un controllo if all'interno dell'installazione dell'eccezione di IOException tale per cui se il valore di currentSending fosse arrivato a 6, e il client avesse cercato quindi di dialogare con un server inesistente, il valore di currentSending veniva resettato a 1, e si ripartiva da capo. Un'altra soluzione sempre riguardo al problema di comunicare con un server non attivo, fu quella di inserire sempre all'interno dell'installazione dell'eccezione un incremento della variabile currentSending, poichè avevamo notato che se il client cercava di dialogare con un server inattivo, l'eccezione veniva si controllata e quindi inviato il messaggio di avviso tramite terminale, ma tutto il processo veniva bloccato. Con questa soluzione, ogni volta che veniva scaturita un'eccezione, si passava alla chiamata successiva.

Constatando che alla fine degli ultimi ritocchi il costruito funzionava decisamente bene, ci sembrò ormai prossima la terminazione del progetto (ci mancava solamente da gestire il problema della concorrenza tra più processi). Ma questo nostro entusiasmo venne frenato subito già durante le prime prove di verifica, poichè notammo l'affioramento di un secondo problema: cioè che (giustamente) se l'Administrator aggiungeva una quantità di prodotto all'interno di un server che in quel momento era leader, il prodotto veniva si aggiunto in quel server, ma non negli altri, e la stessa cosa succedeva per le altre operazioni.

Era evidente che le variabili globali all'interno di ogni server, come la lista carrelli o la lista delle merci, erano globali solamente in relazione a quello specifico server. Bisognava aggiornare (o meglio "replicare" per stare in tema con le specifiche) le liste aggiornate in tutti gli altri server, per farli stare in linea di aggiornamento con il server leader.

Sinceramente questo problema lo abbiamo risolto abbastanza agevolmente poichè decidemmo subito di unanime accordo di utilizzare un operazione denominata **sincronizzaServers**, la quale, dopo ogni operazione terminata dal server in cui fosse stata modificata almeno una delle variabili globali, veniva inviata in parallelo a tutti gli altri server, che istantaneamente la ricevevano e aggiornavano i propri dati.

L'operazione sincronizzaServers utilizza, per il trasporto dei dati aggiornati, il tipo di macro-variabile **syncro**, che contiene tutte le variabili globali che servono per un aggiornamento, o che vengono modificate durante le operazioni.

SEMAFORI

Per gestire la concorrenza tra i vari processi (1 Administrator ed n client) abbiamo utilizzato le strutture dei semafori, oggetto di studio ed esercizio nel corso di Sistemi Operativi.

Per decidere dove inserire i semafori, siamo prima ricorsi al classico modello "carta e penna", ovvero abbiamo riportato su un foglio per esempio due processi concorrenti, ed abbiamo immaginato ogni possibile context switch dannoso che si potesse verificare tra questi due processi, dopodichè abbiamo inserito le soluzioni necessarie tramite l'utilizzo dei semafori, chiaramente sempre con l'intento di non creare un sistema granitico e poco flessibile, cercando così di non limitare troppo la concorrenza. Infine implementavamo questo schema logico su codice ed effettuavamo varie prove per vedere se il tutto funzionava e non si interrompeva.

UTILIZZO DEI SEMAFORI NELLE OPERAZIONI PER L'ADMINISTRATOR

Nell'operazione **caricaProdotti**, che mi carica automaticamente sul server leader un tot di prodotti predefiniti con le relative quantità, abbiamo inizialmente deciso di non utilizzare semafori.

L'operazione viene effettuata una sola volta, ed è stata studiata in modo che solamente dopo il caricamento effettivo delle merci nello Shop, viene settata la variabile globale `administratorHaCaricato = true`.

Con questo meccanismo, se avvenisse per un qualche motivo un context switch e si passasse ad un processo client che vuole andare ad aggiungere elementi all'interno del suo carrello, modificando quindi la lista merci, questo non potrà avvenire poichè la variabile `administratorHaCaricato` sarà ancora uguale a `false`.

Discorso diverso invece per le operazioni **aggiungiProdotto** ed **eliminaProdotto**, che possono essere eseguite in qualsiasi momento nella vita dell'Administrator.

Per l'operazione `eliminaProdotto` potremmo incorrere nella normalissima situazione in cui l'Administrator vuole eliminare una quantità dello stesso prodotto (per es. Prodotto: Banana Quantità: 40) dallo Shop.

L'Administrator invia una richiesta al server, il server verifica la disponibilità nello Shop del prodotto richiesto (per es. Prodotto: Banana Quantità: 50) ed accorda all'Administrator l'operazione. In questo istante avviene un context switch, si passa ad un processo client che richiede al server di aggiungere nel proprio carrello una quantità dello stesso prodotto che vuole eliminare l'Administrator (per es. Prodotto: Banana Quantità: 30).

Il server chiaramente acconsente dato che la quantità non è superiore a quella disponibile ed aggiorna la quantità anche all'interno dello Shop (Prodotto: Banana Quantità: $40-30 = 10$).

Altro context switch, torniamo all'operazione dell'Administrator, che era rimasto all'approvazione della propria operazione da parte del server. A questo punto il server, avendo già effettuato i controlli sulla disponibilità prima del context switch, elimina la quantità di prodotto richiesta dallo Shop (Prodotto: Banana Quantità: $10-40 = -30$), andando ad aggiungere la quantità con una negativa. Questa cosa potrebbe essere molto grave!!

Per risolvere questo problema di concorrenza sull'utilizzo di risorse condivise siamo ricorsi all'utilizzo di strutture già predefinite in Jolie: **i semafori**.

Importando la libreria **"semaphore_utils.iol"** possiamo utilizzare i semafori binari, ovvero variabili intere che possono avere solamente due valori: 0 quando la risorsa non è disponibile, 1 quando la risorsa è disponibile. Inizializzato un semaforo per ogni variabile condivisa, abbiamo deciso di crearne inizialmente 2, uno per la lista merci e una per la lista dei carrelli.

All' inizio di ognuna delle due operazioni allora abbiamo bloccato sia la lista delle merci che la lista dei carrelli. In questo modo quando l'Administrator esegue un'operazione di cancellazione di prodotti sullo Shop, anche se avvenisse un context switch e si ripresentasse la situazione di un client che vuole aggiungere prodotti al suo carrello, quest'ultimo non potrà farlo, poichè nel momento in cui cercherà di accedere alla lista delle merci verrà bloccato, dato che il semaforo ha valore 0, per cui significa che quella risorsa condivisa sta venendo utilizzata da un altro processo. Ma chi è quel processo? L'Administrator!

Quando l'Administrator avrà terminato la sua operazione allora rilascerà i due semafori rendendo disponibili le due variabili condivise.

Agendo in questo modo, con l'utilizzo dei semafori per le due variabili condivise, facciamo sì che ogni operazione del server venga vista come un'operazione atomica, indivisibile e protetta da un qualsiasi tipo di context switch. Questo permetterà al server di eseguire le operazioni una per volta senza avere il rischio di incorrere in context switch dannosi.

UTILIZZO DEI SEMAFORI NELLE OPERAZIONI PER IL CLIENT

L'utilizzo dei semafori lo abbiamo esteso successivamente alle operazioni del server che riguardavano il client.

Per le operazioni che riguardavano solamente la modifica della lista dei carrelli disponibili o dei

carrelli Archiviati, quali **crea/cancellaCarrello** e **acquistaCarrello** abbiamo utilizzato solamente un semaforo di blocco sui carrelli che si acquisiva all'inizio e si rilasciava alla fine.

In questo modo si evitano situazioni di errore, come per esempio se un client sta aggiungendo prodotti all'interno del suo carrello, e avviene un context switch e si passa ad un processo di un client che vuole cancellare quel carrello, lo cancella e poi con un altro context switch si ripassa al processo del primo client, che andrebbe ad aggiungere elementi in un carrello inesistente.

Abbiamo inserito i semafori anche nell'operazione **vediListaOggetti**, che permette al client di richiedere la lista delle merci vendute all'interno dello shop.

Nelle operazioni **aggiungi/rimuoviElementi** all'interno del carrello, che sono quelle più problematiche, dato che vanno ad utilizzare sia la lista dei carrelli che la lista delle merci, abbiamo deciso di utilizzare due semafori, uno (**semaforoCarrelli**) che si acquisiva quando si entrava nel carrello, e un altro che si acquisiva quando si andava a toccare la lista delle merci. Alla fine dei cicli sulle varie liste, i semafori venivano rilasciati. Con questa implementazione però, notammo che quando il client richiedeva di aggiungere elementi all'interno del carrello, il processo server andava in deadlock e si bloccava su quella operazione.

Come soluzione abbiamo deciso di rilasciare il semaforo dei carrelli prima di utilizzare quello per modificare la lista delle merci, e poi lo riacquisivamo al termine della modifica.

All'inizio eravamo abbastanza titubanti sull'utilizzo dei semafori anche nell'operazione di sincronizzazione, dato che alla fine consisteva solamente in un passaggio di variabili.

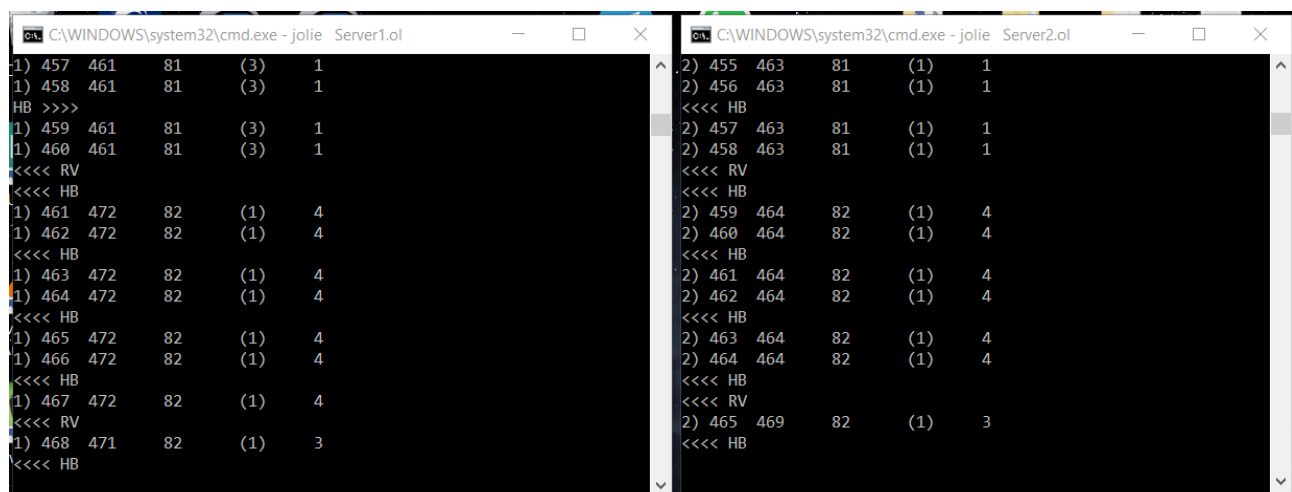
Ma ragionando meglio sulle istruzioni che vengono eseguite all'interno dell'operazione di sincronizzazione, abbiamo pensato che tutte quelle assegnazioni e aggiornamenti di variabili globali siano particolarmente vulnerabili da parte di un context switch. Quindi per sicurezza abbiamo deciso di gestire la concorrenza anche in questo caso.

Sottolineiamo infine che abbiamo deciso di utilizzare SODEP (Simple Operation Data Exchange Protocol), un protocollo binario creato e sviluppato appositamente per Jolie, al fine di fornire un protocollo semplice, sicuro ed efficiente per le comunicazioni di servizio.

DEMO DEL PROGETTO (2 Client, 5 Server, 1 Administrator, 1 Network Visualizer)

FASE 1: Avviare i 5 Server

(Si riportano di seguito gli screenshot dei 5 server già avviati)



```
C:\WINDOWS\system32\cmd.exe - jolie Server1.ol
1) 457 461 81 (3) 1
1) 458 461 81 (3) 1
HB >>>>
1) 459 461 81 (3) 1
1) 460 461 81 (3) 1
<<<< RV
<<<< HB
1) 461 472 82 (1) 4
1) 462 472 82 (1) 4
<<<< HB
1) 463 472 82 (1) 4
1) 464 472 82 (1) 4
<<<< HB
1) 465 472 82 (1) 4
1) 466 472 82 (1) 4
<<<< HB
1) 467 472 82 (1) 4
<<<< RV
1) 468 471 82 (1) 3
<<<< HB

C:\WINDOWS\system32\cmd.exe - jolie Server2.ol
2) 455 463 81 (1) 1
2) 456 463 81 (1) 1
<<<< HB
2) 457 463 81 (1) 1
2) 458 463 81 (1) 1
<<<< RV
<<<< HB
2) 459 464 82 (1) 4
2) 460 464 82 (1) 4
<<<< HB
2) 461 464 82 (1) 4
2) 462 464 82 (1) 4
<<<< HB
2) 463 464 82 (1) 4
2) 464 464 82 (1) 4
<<<< HB
<<<< RV
2) 465 469 82 (1) 3
<<<< HB
```

```
C:\WINDOWS\system32\cmd.exe - jolie Server3.ol

3) 454 460 81 (1) 1
3) 455 460 81 (1) 1
<<<< RV
<<<< HB
3) 456 461 82 (1) 4
3) 457 461 82 (1) 4
<<<< HB
3) 458 461 82 (1) 4
3) 459 461 82 (1) 4
<<<< HB
3) 460 461 82 (1) 4
3) 461 461 82 (1) 4
<<<< HB
SONO CANDIDATE
RV >>>>
Server 1 ha votato per me
Server 2 ha votato per me
Server 4 ha votato per me
Server 5 ha votato per me
3) 462 469 83 (3) 3
HB >>>>
3) 463 469 83 (3) 3
```

```
C:\WINDOWS\system32\cmd.exe - jolie Server4.ol
<<<< HB
4) 452 452 81 (1) 1
SONO CANDIDATE
RV >>>>
Server 1 ha votato per me
Server 2 ha votato per me
Server 3 ha votato per me
Server 5 ha votato per me
HB >>>>
4) 453 461 82 (3) 4
4) 454 461 82 (3) 4
HB >>>>
4) 455 461 82 (3) 4
4) 456 461 82 (3) 4
HB >>>>
4) 457 461 82 (3) 4
4) 458 461 82 (3) 4
HB >>>>
4) 459 461 82 (3) 4
<<<< RV
4) 460 468 82 (1) 3
<<<< HB

C:\WINDOWS\system32\cmd.exe - jolie Server5.ol
5) 445 452 81 (1) 1
5) 446 452 81 (1) 1
<<<< HB
5) 447 452 81 (1) 1
5) 448 452 81 (1) 1
<<<< HB
5) 449 452 81 (1) 1
<<<< RV
<<<< HB
5) 450 458 82 (1) 4
5) 451 458 82 (1) 4
<<<< HB
5) 452 458 82 (1) 4
5) 453 458 82 (1) 4
<<<< HB
5) 454 458 82 (1) 4
5) 455 458 82 (1) 4
<<<< HB
5) 456 458 82 (1) 4
<<<< RV
5) 457 460 82 (1) 3
<<<< HB
```

FASE 2: Avviare l'Administrator

```
C:\WINDOWS\system32\cmd.exe - jolie Administrator.ol

Benvenuto nell'ADMINISTRATOR
currentSending: 1 status: 0
Invio a SERVER 1
SERVER 1: Non sono io il leader.
currentSending: 5 status: 5
Invio a SERVER 5
SERVER 5: Lista Merci caricata con successo

Operazioni disponibili:
  1 - Aggiungere una quantita di un prodotto all'interno dello Shop
  2 - Eliminare una quantita di un prodotto all'interno dello ShopPort
  3 - Terminare il processo dell'Administrator
```

L'Administrator carica la lista delle merci predefinita in automatico nel server leader, che in questo caso è il 5.

```
C:\WINDOWS\system32\cmd.exe - jolie Server5.ol
<<<< HB
RV >>>>
Server 1 ha votato per me
Server 2 ha votato per me
Server 3 ha votato per me
Server 4 ha votato per me
5) 864 867 163 (3) 5
HB >>>>
5) 865 867 163 (3) 5
5) 866 867 163 (3) 5

Ho ricevuto da Administrator:4 prodotti
Banana-50
Cocomero-50
Carota-50
Insalata-50
SYNC >>>>
```

(Il server 5 riceve i prodotti dall'Administrator)

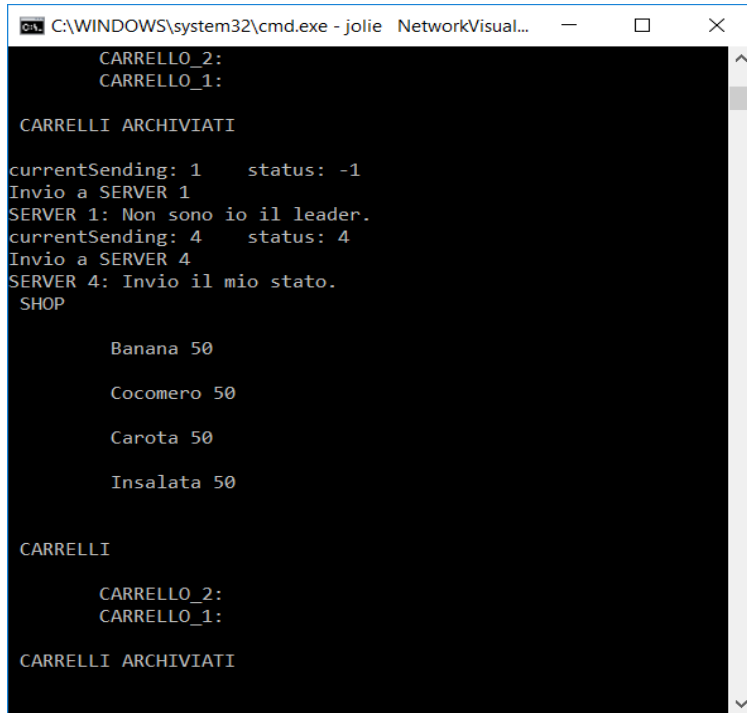
FASE 3: Avviare i 2 Client

```
C:\WINDOWS\system32\cmd.exe - jolie Client.ol
'BENVENUTO IN RAFT SHOP'
Digitare il numero dell'operazione da eseguire:
1 - Creare un carrello
2 - Visualizzare la lista degli oggetti presenti nello
3 - Cancellare un carrello
4 - Acquistare un carrello
5 - Aggiungere elementi nel carrello
6 - Rimuovi elementi dal carrello
7- Termina il client
1
Digitare il nome del carrello da creare (senza spazi):
CARRELLO_1
currentSending: 1 status: -1
Invio a SERVER 1
SERVER 1: Non sono io il leader.
currentSending: 5 status: 5
Invio a SERVER 5
SERVER 5: Carrello CARRELLO_1 creato con successo.
Si desidera effettuare un'altra operazione?(Digitare il

C:\WINDOWS\system32\cmd.exe - jolie Client.ol
'BENVENUTO IN RAFT SHOP'
Digitare il numero dell'operazione da eseguire:
1 - Creare un carrello
2 - Visualizzare la lista degli oggetti presenti nello Shop
3 - Cancellare un carrello
4 - Acquistare un carrello
5 - Aggiungere elementi nel carrello
6 - Rimuovi elementi dal carrello
7- Termina il client
1
Digitare il nome del carrello da creare (senza spazi):
CARRELLO_2
currentSending: 1 status: -1
Invio a SERVER 1
SERVER 1: Carrello CARRELLO_2 creato con successo.
Si desidera effettuare un'altra operazione?(Digitare il numero dell'operazione)
```

(Si crea prima il carrello CARRELLO_2 e poi il CARRELLO_1)

FASE 4: Avviare il Network Visualizer



```
C:\WINDOWS\system32\cmd.exe - jolie NetworkVisual...

CARRELLO_2:
CARRELLO_1:

CARRELLI ARCHIVIATI

currentSending: 1    status: -1
Invio a SERVER 1
SERVER 1: Non sono io il leader.
currentSending: 4    status: 4
Invio a SERVER 4
SERVER 4: Invio il mio stato.
SHOP

    Banana 50

    Cocomero 50

    Carota 50

    Insalata 50

CARRELLI

    CARRELLO_2:
    CARRELLO_1:

CARRELLI ARCHIVIATI
```

(Il Network riceve le liste aggiornate e le visualizza tramite terminale)