RaftShop

Esame di Laboratorio Sistemi Operativi A.A. 2016-2017

01011001

Componenti:
- Berti, Matteo
- Cesco, Arnaldo
- Fiorilla, Salvatore

Introduzione

RaftShop implementa un sistema distribuito per il commercio elettronico. Grazie all'implementazione dell'algoritmo di consenso "*Raft*" è fault-tolerant, ovvero una parte dei server possono riscontrare problemi lasciando complessivamente il sistema intatto. L'architettura di RaftShop è composta da 3 soggetti principali: i Client, i Server(5) e L'Admin. A questi si aggiunge il Network Visualizer, un tool amministrativo per monitorare lo stato dei Server.

- Report

Questo report è suddiviso in tre macro-sezioni: la prima, "Introduzione", in cui si spiega a grandi linee il progetto nel suo complesso. Una sezione "Esecuzione" in cui si spiega tutto il necessario per il corretto utilizzo del sistema RaftShop e un'ultima "Implementazione" in cui si vanno ad approfondire gli aspetti tecnici dell'implementazione del sistema. Talvolta tali sezioni possono essere ulteriormente suddivise in sotto-paragrafi per distinguere meglio gli argomenti.

- Linguaggi

Per la costruzione di tale sistema sono stati utilizzati vari linguaggi:
- Jolie (linguaggio di programmazione basato su microservizi): sono implementati i Server, il Network Visualizer e le interfacce da terminale di Client e Admin.
- HTML (linguaggio di markup): è stata implementata un interfaccia Client e Admin a cui è possibile accedervi tramite browser.
- JavaScript (linguaggio di scripting): utilizzato sia per la comunicazione tra i Server in Jolie e l'interfaccia HTML da browser che per dinamizzare i contenuti dell'interfaccia web. E' inoltre stata utilizzata la libreria JavaScript esterna jQuery, e il Framework per applicazioni web Bootstrap.

- Mission

Nella costruzione di questo sistema si è voluto ottimizzare sempre (dove possibile) la concorrenza, tenendo a mente durante lo sviluppo anche scenari, a volte molto rari nella realtà, ma che a livello teorico avrebbero potuto creare problemi al sistema.

Esecuzione

Prima di tutto è necessario far partire almeno 3 Server (in quanto perché venga eletto un leader è necessario che la maggioranza voti per lui), quest'operazione può essere fatta solo da terminale. Si consiglia tuttavia di eseguire tutti e 5 i Server in modo che si possano fare delle prove in cui uno, o più, di essi cadano.
In secondo luogo è consigliabile eseguire un Admin, tale operazione può essere fatta da terminale oppure tramite browser; sarà quindi possibile aggiungere (o cancellare) elementi disponibili al sistema.
Infine si possono avviare i Client e il Network Visualizer, anche qui il Client è avviabile sia da terminale che tramite browser, a differenza del Network Visualizer avviabile solo tramite terminale. Il client potrà quindi creare carrelli e aggiungervi elementi, mentre con il Network Visualizer è possibile monitorare lo stato del sistema.

- Prerequisiti

Per l'esecuzione dei file Jolie è necessaria l'installazione del Jolie Installer (http://www.jolie-lang.org/downloads.html) il quale necessita che sia preinstallata una versione uguale o superiore di Java 8 (https://www.java.com/it/download/).
Per l'esecuzione dell'interfaccia HTML del Client e dell'Admin è necessario aver installato nella propria macchina un Web Server Locale, ad esempio XAMPP (https://sourceforge.net/projects/xampp/).

- Demo

Per l'avvio dei Server, in 5 terminali diversi digitare:
> jolie ServerN.ol
[Dove N è un numero compreso tra 1 e 5, estremi inclusi]
Per l'avvio dell'Admin da terminale:
> jolie Admin.ol
> jolie Admin.ol *id/item1Name/item1Qnt id/item2Name/item2Qnt ...*
[E' possibile inserire o rimuovere quanti elementi si vuole direttamente all'avvio tramite la forma *id/itemName/itemQnt* digitando quindi un id (2 per aggiungere e 3 per rimuovere), il nome dell'elemento e la quantità che si vuole modificare. Tali voci vanno separate da uno slash "/" e ogni operazione separata da uno spazio.]
Per l'avvio del Client da terminale:
> jolie Client.ol

Per l'avvio del Network Visualizer da terminale:
> jolie NetworkVisualizer.ol

Per l'utilizzo dell'interfaccia HTML dell'Admin avviare il proprio web server locale, aprire un browser e digitare : "**localhost:10001/admin.html**".
[Il numero di porta può variare nell'intervallo 10001-10005 in base a chi è il server leader, in caso quest'ultimo cambi si verrà reindirizzati alla porta corretta]
Per l'utilizzo dell'interfaccia HTML del Client avviare il proprio web server locale, aprire un browser e digitare : "**localhost:9001/client.html**".
[Il numero di porta può variare nell'intervallo 9001-9005 in base a chi è il server leader, in caso quest'ultimo cambi si verrà reindirizzati alla porta corretta]
Tutte le librerie e framework esterni utilizzati sono stati scaricati in locale in modo che possa essere utilizzato anche senza una connessione internet.

Implementazione

I file Client.ol, Admin.ol, Server1.ol, Server2.ol Server3.ol Server4.ol, Server5.ol, NetworkVisualizer.ol implementano i componenti di RaftShop. Il file maininterface.ol, contiene tutte le interfacce necessarie alla comunicazione con gli elementi esterni (ClientInterface, AdminInterface, VisualizerInterface) ed interni al sistema (ServerInterface, TimerInterface, ElectionInterface) più le interfacce per i micro servizi (ClientActionInterface, AdminActionInterface, VisualizerActionInterface).
Il file DataManager.ol, presente come micro-servizio embeddato nei server, offre due servizi: esecuzione sulla state-machine dei comandi arrivati al server dal Client o Admin e accesso allo stato di RaftShop (elementi disponibili, carrelli, …).
La directory Timers contiene 6 file, micro-servizi embdeddati all'interno dei server, molto simili tra loro ma con compiti diversi e non riassumibili in un unico file. La directory HTTP contiene tutti i file necessari ad implementare l'interfaccia utente in HTML e JavaScript.

- Client

I Client hanno a disposizione 8 comandi: (1.) richiedere lista e quantità delle merci in vendita; (2.) registrare un nuovo carrello, il cui nome sarà unico; (3.) acquistare un carrello; (4.) cancellare un carrello; (5.) richiedere gli elementi presenti in un carrello; (6.) aggiungere un elemento in un carrello; (7.) rimuovere un elemento da un carrello e infine (0.) terminare l'esecuzione e uscire. Il tutto è eseguito secondo le specifiche date (il nome di un carrello comprato rimane riservato, il nome di uno cancellato ritorna disponibile, …). Il Client comunica solo con un server alla volta tramite la porta di output (serverPort), implementata con la tecnica del dynamic binding per via della possibilità di reindirizzamento. L'esecuzione single è utilizzata per creare un flusso di

richiesta da terminale con conseguente esecuzione, l'utilizzo di RequestResponse vuole impedire che si richieda una nuova azione al sistema prima di aver ricevuto risposta dalla precedente. La comunicazione Client-Sistema avviene inizialmente generando un numero casuale da 0 a 4 che sarà la prima posizione del server (global.serverPos) a cui il Client tenterà di connettersi. Il risultato potrà essere un server down o up, nel primo caso tramite un install(IOException) si salva che nella posizione [0-4] relativa al soggetto contattato è presente un server down. Nel secondo caso il server potrà essere o follower o leader; in caso di follower si salva nuovamente, nello stesso array usato per memorizzare i down, che il soggetto è un follower (tramite .isDown e .isFollower), in questo caso si provvederà a contattare il leader che viene indicato. In caso di leader inizia la comunicazione vera e propria. In presenza di server down si provvederà a cercare tra tutti il primo server che non è stato registrato come down o follower, in tal modo l'utente dovrà digitare una sola volta il comando da eseguire, poi sarà questo meccanismo a trovare il server leader, connettersi e passargli la richiesta, restituendo un risultato corretto. La connessione con il leader verrà mantenuta finché questo non indica di essere follower o down. Nell'eventualità in cui nessuno dei 5 server risulta leader (ad esempio un follower indica come leader un server down o un altro follower) l'esecuzione termina indicando che il sistema è momentaneamente non disponibile.
Quando il Client contatta un server manda una serie di dati che a seconda del comando hanno caratteristiche diverse ma a con elementi comuni: la variabile che viene passata ha come figli .code, che contiene il codice della chiamata [1-7 Client o 1-3 Admin] e .data, che contiene i dati relativi al comando (la cui struttura varia in base al tipo di richiesta). Quest'ultimo figlio accetta oggetti EditItemInCartType, EditItemInListType, EditCartType, void o string dove string è per l'integrazione html, che non può trasferire void. Non sono stati effettuati controlli scrupolosi sul formato dei valori ricevuti dal client (accenti nei nomi per oggetti, carrelli, quantità) in quanto ci siamo concentrati più sulla gestione della concorrenza, confidando nella correttezza dell'utente. La risposta ricevuta può avere 3 tipi: LeaderAddress, nel caso in cui il server contattato sia un follower (nel figlio .address è presente l'indirizzo del leader), BasicResponse, che ha due figli: .result di tipo boolean (true se l'operazione è stata eseguita con successo, false altrimenti) e .msg (stringa che contiene un eventuale messaggio da parte del server). E undefined, nel caso dei comandi 1 e 5: questo perché la struttura dati ricevuta non può avere un tipo fisso in quanto è nella forma global.items.nomeItem. Questo formato è stato preferito a un array in quanto in caso di ricerca di un elemento è possibile trovarla immediatamente ad esempio undef(global.items.(request.data.name)), invece che scorrere tutto l'array.

- Admin

L'Administrator ha a disposizione 4 comandi: (1.) richiedere lista e quantità delle merci in vendita; (2.) aggiungere un elemento al sistema, con una certa quantità disponibile; (3.) cancellare una certa quantità disponibile di un elemento e infine (0.) terminare l'esecuzione e uscire. L'admin è implementato in modo del tutto simile al client per quanto riguarda la ricerca e connessione al server leader, di conseguenza per le note implementative non presenti si rimanda a quanto scritto sopra.
Una feature presente nell'Admin che il Client non ha è la possibilità di inserire (o cancellare) una lista di elementi direttamente all'avvio, da riga di comando.
Tramite la creazione di un buffer contenente queste operazioni è possibile scorrere ciclicamente il blocco di codice che gestisce l'esecuzione delle richieste, come se fossero tutte operazioni provenienti da un utente fisico; solo al termine di tale buffer è richiesto l'input da terminale (nella versione HTML dell'Admin questa feature non è presente).

- Interfaccia Utente HTML

È presente un interfaccia per il Client e una per l'Admin, collegandosi alle porte riservate al Client (9001-9005) non è possibile accedere all'admin.html, e viceversa dalle porte (10001-10005) non si può accedere al file client.html. In tali interfacce è presente una tabella in cui vengono mostrati gli elementi disponibili, o quelli presenti in un carrello specifico filtrato tramite il bottone "Filter", e vari campi e bottoni il cui funzionamento è autoesplicativo. Una feature che è stata aggiunta è la possibilità di auto reindirizzamento in caso di collegamento ad una porta non leader, ciò è possibile grazie al fatto che un server quando riceve un contatto dal Client/Admin se non è leader invia l'indirizzo corretto, completo di porta.
Passando all'implementazione è stata utilizzata la tecnica AJAX fornita dalla libreria jQuery per comunicare tramite protocollo HTTP e oggetti JSON con i server in Jolie. A lato un esempio di codice che invia al server tramite la RequestResponse "AdminRequest" un oggetto JSON contenuto in "request" e riceve risposta tramite "response" restituendo un messaggio di errore o di successo, per un analisi più approfondita si rimanda al file /HTTP/js/code.js

```
function addItemToList(request) {
  $.ajax({
    url: '/AdminRequest',
    contentType: "application/json",
    data: JSON.stringify(request),
    type: 'POST',
    success: function(response) {
      if(response.address) // Se ho cont
        redirectToLeader(response);
      else {
        var result = response.result;
        if(result)
          displaySuccess(response.msg);
        else
          displayError(response.msg);
      }
    }
  })
}
```

- Network Visualizer

E' uno strumento di monitoraggio dello stato di RaftShop, tramite una OneWay GlobalStatus riceve periodicamente (ogni secondo) lo stato del sistema. È presente un' unica input port ListenerPort alla porta 12000 per ricevere una struttura di tipo VisualizerType formata da: .Leader che contiene a sua volta due figli: .id e .port; il

primo è l'identificativo numerico intero del leader e il secondo una stringa di caratteri contenete il numero di porta del leader. .servers* che contiene un array di boolean indicante lo stato dei server (la cui cardinalità può essere potenzialmente infinita in caso di estensioni future, grazie all'operatore * di Jolie). .items, di tipo undefined perchè può contenere diversi elementi ognuno identificato dal proprio nome. .carts che è definito anche esso come undefined per lo stesso motivo è l'identificativo dei carrelli. L'esecuzione degli scope è sequenziale in quanto se messi in parallelo non mostrerebbero sempre lo stesso ordine nella grafica e quindi renderebbe il tutto poco chiaro.

- Server

Ogni server ha due compiti: eseguire le operazioni richieste da Client o Admin e gestire la distribuzione del consenso attraverso l'algoritmo Raft. Il primo compito è soddisfatto utilizzando un servizio embedded in locale DataManager, il cui codice è contenuto nel file "DataManager.ol", che si occupa anche di gestire la concorrenza tra le richieste e contiene una lista di oggetti e carrelli presenti all'interno del negozio.
Lo stato del server è inizializzato nello scope init secondo quanto richiesto da Raft. In particolare, status è una variabile global, perché deve essere comune a tutte le istanze. Inoltre, per l'implementazione del log è usato un array contenente un termine ed una entry, che a sua volta contiene il proprio termine di entrata nel log, un booleano adminAction ed una richiesta che può essere inviata al DataManager. In aggiunta a Raft, è presente un intero myID che serve a identificare il server quando invia richieste particolari, per esempio una setVisualizerTimeout. Infine, sono inizializzati i dati necessari alla comunicazione con il Network Visualizer e settato il timer per l'elezione.
Nel server, la gestione delle richieste di Admin/Client è separata a seconda delle tipologie: le richieste di lettura ottengono una risposta diretta, poiché non modificano la struttura dati contenente gli elementi del negozio, mentre le richieste di scrittura devono passare attraverso Raft.
Per quanto riguarda le operazioni di scrittura nel log il flusso è il seguente: aggiunta al log della nuova voce, acquisizione di un semaforo specifico per la richiesta Admin/Client, acquisizione di un semaforo generale, attesa del rilascio di quest'ultimo (che avverrà solo dopo che l'entry è stata replicata dalla maggioranza), esecuzione della richiesta, rilascio del semaforo specifico per la richiesta successiva a quella appena conclusa. Per un'analisi più approfondita e dettagliata di tale meccanismo si rimanda alla sezione "Replicazione del log" infondo al report.

- DataManager

I carrelli e gli oggetti presenti nel negozio sono memorizzati in due array dinamici, carts ed items, di tipo global poiché devono contenere gli stessi valori per ogni istanza del servizio. Essendoci la possibilità di operazioni concorrenti di lettura e scrittura sui

due array si è presentato un problema riconducibile al modello Readers-Writers, risolto modificando lievemente la soluzione classica nota in letteratura. Per aumentare la concorrenza, si è deciso di creare due semafori, uno per gestire la mutua esclusione in fase di scrittura dei carrelli (wrtCarts) ed uno per la scrittura degli oggetti (wrtItems). Analogamente, sono presenti due variabili globali, readCountItems e readcountCarts, che indicano il numero di lettori presenti al momento sulla lista di oggetti e di carrelli; queste due variabili non hanno un semaforo associato ma due token di synchronized (rispettivamente mutexItems e mutexCarts) perché, a parità di grado di concorrenza consentito rispetto ai semafori, il loro utilizzo è risultato in un codice più chiaro. Per contro, la sostituzione dei semafori citati in precedenza con dei blocchi synchronized avrebbe reso meno concorrente la gestione dei dati (vedi il caso di eliminazione di un carrello da parte di un client). Sia wrtItems che wrtCarts sono inizializzati a 1 nel blocco init del codice, mentre readcountCarts e readcountItems sono inizializzati, com'è ovvio, a 0.
Tutte le operations sono delle request-response, perché il compito del DataManager è eseguire sulla state-machine quanto richiesto, pertanto deve avere comportamento bloccante. Sia Client che Admin hanno un'unica operation, chiamata ClientAction o AdminAction, che accetta come richiesta un RequestType formato da un .code intero e un .data di vario tipo (a seconda dell'operazione da eseguire sulla state machine) e fornisce una risposta di tipo ResponseType che può essere un indirizzo (in caso di necessità di reindirizzamento), un BasicResponse (successo o meno dell'operazione e un messaggio correlato) o undefined, il cui motivo è stato discusso precedentemente in Client; sono presenti due interfacce ClientInterface e AdminInterface.
Inoltre il DataManager offre anche un'operation request-response di visualizzazione dello stato del negozio, concorrente con le altre, che non richiede parametri in entrata e restituisce sia la lista di tutti i carrelli che quella degli oggetti, GetShopStatus.

- Raft

L'algoritmo Raft è stato scomposto in due fasi principali: elezione di un leader e replicazione del log. Lo svolgersi delle fasi è scandito da dei timer che gestiscono l'invio di richieste tra server. Per l'elezione è usata la RequestResponse ElectionRequest perché, anche se un candidato si bloccasse in attesa di risposta Raft garantisce che ad un altro potrà scadere l'electionTimer, candidarsi e ottenere la maggioranza. Per la replicazione del log invece sono usate due operation OneWay (AppendEntries ed Ack) così che il leader non si blocchi in attesa di risposta da parte di un follower, caso che rallenterebbe di molto il sistema distribuito portando potenzialmente a una infinita serie di elezioni senza alcuna replicazione del log riuscita. Ogni appendEntries è inviata dentro ad un proprio scope per aumentare la concorrenza e poter catturare un eventuale IOException in caso di server down senza bloccare l'invio delle altre.

- Timers

Ogni timer è implementato in un microservizio embedded: ElectionTimer, VisualizerTimer e HeartBeatTimerX, $A \leq X \leq D$. Ogni timer, che esegue in maniera concorrente, può essere invocato attraverso la OneWay SetHeartbeatTimer che ha come parametri la porta di input del produttore della richiesta ed un intero indicante il numero di microsecondi da aspettare. All'arrivo della richiesta è invocata l'operation setNextTimeout@Time(..)(..) e allo scadere inviato un ElectionTimeout o HeartbeatTimeout al server produttore della richiesta.
La presenza di più file aventi concettualmente lo stesso ruolo e contenenti codice molto simile è dovuta alla natura dell'operation setNextTimeout@Time: ogni volta che viene invocata, se quel timer era già attivo esso viene azzerato. Pertanto, per aumentare la concorrenza, specialmente nel caso dell'invio di heartbeat, si è deciso di avere un timer per ogni follower: così si hanno momenti di invio dell'heartbeat diversi, a seconda di quando il leader riceve l'ultimo ack, mantenendo comunque costante il tempo di invio.

- Elezione

La fase di elezione avviene quando l'ElectionTimer invia una richiesta di ElectionTimeout. Il server chiama la procedura election e parallelamente fa scattare un altro ElectionTimer. Election implementa le specifiche di Raft. Nello scope initElectionRequest è inizializzata la richiesta da inviare, mentre sendElectionRequest invia parallelamente a tutti i server una richiesta di voto, gestendo il caso in cui un server sia down aggiornando il relativo stato. I risultati della votazione sono inseriti nell'array result e successivamente analizzati in analyzeResults: il candidato vota per sé stesso; se esiste un server con termine maggiore rispetto a quello del candidato, il candidato stesso passa a follower ed i suoi nextIndices e matchIndices resettati; se invece ottiene la maggioranza parallelamente il suo stato diventa leader, gli indici si aggiornano, si setta il timer per inviare dati al Network Visualizer e viene inviato il primo heartbeat agli altri server. In caso di maggioranza non ottenuta, il candidato ritorna follower. Le modifiche su NextIndices e MatchIndices sono racchiuse in blocchi synchronized per evitare che possano essere cambiati in maniera diversa da quanto richiesto nelle specifiche a causa di un'esecuzione concorrente, per esempio l'arrivo di una AppendEntries valida.
Ricevuta una richiesta di voto il server si comporta come da specifiche di Raft. La variabile global votedFor viene ripulita ogni volta che si trova un candidato con termine superiore al proprio, perché qualora fosse occupata ci sarebbe un'incorrettezza nei dati. Per una maggior concorrenza non a discapito della correttezza solamente l'operazione di lettura dell'ultimo termine del log è stata inclusa nel blocco synchroninzed operationOnLog, perché in caso di arrivo concorrente di una AppendEntries valida è necessario evitare ci siano modifiche sul lastIndexOfLog.

- Replicazione del log

Le strutture utilizzate per la replicazione sono: un semaforo *global.replicationSemaphore* (che ha la funzione di bloccare il flusso di esecuzione in attesa che la voce che si sta per eseguire sia stata replicata dalla maggioranza dei server), un array di semafori *global.executionSemaphores[…]* (che permettono di creare un ordine nell'esecuzione degli elementi del log, seguendo il FCFS), un array parallelo al log *global.replicationInServers[…]* (che memorizza per ogni suo indice il numero di server che hanno replicato la voce del log in quella posizione; ha anche un figlio .valid di tipo booleano). Per prima cosa nell'init è creato *global.executionSemaphores[0]* e rilasciato una volta (per renderlo disponibile alla prima richiesta Admin/Client). A questo punto la prima richiesta di scrittura arrivata entra in un blocco synchronized nel quale si effettuano principalmente 3 azioni: prima si crea un semaforo in coda alla lista di executionSemaphores e si assegna il semaforo in posizione *#global.executionSemaphores* – 1 come semaforo da acquisire, e quello in posizione *#global.executionSemaphores* come semaforo da rilasciare. In questo modo la richiesta successiva avrà come semaforo da acquisire quello che la precedente aveva da rilasciare. Come seconda cosa si incrementa l'ultimo indice del log e in quella posizione, il global.replicationInServers[..], si setta a 0, in quanto nessuno ha ancora replicato, e il .valid a true. Infine si aggiunge al log l'entry che è stata ricevuta.

Successivamente, fuori dal precedente synchronized, si acquisisce il semaforo relativo a questa richiesta (se è la prima esecuzione è già stato rilasciato, se no verrà rilasciato dall'esecuzione precedente), si invia un appendEntries a tutti e si acquisisce il semaforo *global.replicationSemaphore* . A questo punto l'unico modo per sbloccare tale semaforo è il rilascio effettuato in seguito alla replicazione da parte della maggioranza dei server, dell'entry aggiunta. Infatti ogni AppendEntries restituisce un Ack contenente un *replicatedIndex* e *replicatedTerm* che indicano l'indice e il termine dell'entry appena replicata (se è stata replicata un entry, altrimenti undefined in caso di heartbeat). Quando arriva un Ack al leader che contiene questi valori, dopo vari controlli (tra cui l'esito positivo dell'Ack, che il termine in posizione *replicatedIndex* del log sia uguale a *replicatedTerm*, la validità dell'elemento *global.replicationInServers[replicatedIndex].valid*, ...) va ad aumentare la posizione *replicatedIndex*-esima del *global.replicationInServers* fino a che non diventa == 2 (ciò significa che due server hanno replicato tale entry). In caso si arrivi a 2 replicazioni, si imposta il .valid a false (per evitare che futuri Ack di tale voce sblocchino inutilmente il semaforo), si aggiorna il commitIndex al lastIndex arrivato con l'Ack e si rilascia il *lobal.replicationSemaphore*. A questo punto il flusso precedente continua e l'admin esegue tramite il DataManager sulla propria state-machine, aumentando anche il lastApplied. Infine viene rilasciato il semaforo per la richiesta successiva, in modo che possa continuare il flusso.