



Progetto di Sistemi Distribuiti

Implementazione dell'algoritmo di consenso Raft su middleware Jade

Eugenio Pierfederici

September 6, 2021

Contents

1	Il progetto	5
	Analisi	5
	Realizzazione	6
	Indice dei contenuti:	6
	Esempi di funzionamento e controllo di correttezza:	6
2	Progettazione (UML)	7
3	Il cluster	13
4	Il server agent	15
	HandleClientRequestBehaviour	15
	WaitForElectionTimeoutBehaviour	16
	SendRequestVoteRPCBehaviour	16
	ReceiveRequestVoteRPCBehaviour	16
	LeaderHeartbeatBehaviour	17
	SendAppendEntriesRPCBehaviour	17
	ReceiveAppendEntriesRPCBehaviour	17
5	Il client agent	19
	SendClientRequestBehaviour	19
6	Robustezza dell'algoritmo	21
	Punti di forza di Raft	21
	Debolezze di Raft	21
7	Test effettuati	23
	Test corretto funzionamento dell'elezione	23
	Test corretto funzionamento dell'heartbeat	25
8	Guida per principianti: creazione del primo cluster	29
	La LogAction	29
	Il server agent	30
	Il client agent	32
9	GUI (struttura per l'osservazione degli agenti)	33
	Il server	33
	Il client	33
	L'osservatore	34
10	Sviluppi futuri	35
	Stress test dell'algoritmo	35
	Cambio dei membri del cluster a runtime	35

Chapter 1

Il progetto

Il progetto sviluppato consiste in un'implementazione dell'algoritmo di consenso Raft in Java usando il framework Jade.

Analisi

Il problema

Il problema è quello di mantenere la coerenza in un insieme di server che non condividono la memoria e, probabilmente, non si trovano nemmeno nella stessa rete (perciò potrebbero avere problemi di comunicazione dovuti alla lentezza o ad un middleware di comunicazione malevolo). Perciò diventa complesso mantenere uno stato coerente su di essi.

Per risolvere questo problema sono stati ideati degli algoritmi, chiamati algoritmi di consenso; questi sono di vario tipo e risolvono problematiche differenti (protezione contro server malevoli, un middleware malevolo, o semplicemente dalla perdita di informazioni). In questo progetto si è scelto di usare l'algoritmo Raft, che è meglio spiegato nella sezione seguente.

Raft

Raft è un algoritmo di consenso che cerca di risolvere il problema dei generali Bizantini.

Il problema dei generali Bizantini è un problema di consenso secondo il quale un insieme di generali Bizantini devono accordarsi sul se attaccare o ritirarsi. Il caso peggiore lo si ha se solo una parte dell'esercito attacca e l'altra si ritira, quindi tutti i generali devono accordarsi sulla stessa azione.

Punto cruciale del problema è il fatto che i generali agiscono in autonomia e possono comunicare solo tramite scambio di messaggi (i messaggeri). I problemi che si possono verificare sono vari: perdita dei messaggi, falsificazione del contenuto, lentezza del messaggio, generale sleale, etc.

L'algoritmo di consenso Raft propone una possibile soluzione al problema, che però risolve solo alcuni di questi problemi (per esempio si fida della lealtà di ciascun generale e dell'affidabilità dei messaggeri). Nello specifico l'affidabilità dei messaggeri la garantisce Jade (meglio affrontato nella sezione seguente), mentre la lealtà dei generali è garantita solo finché non viene aggiunto al cluster un server malevolo (gli altri non hanno modo di proteggersi).

Maggiori dettagli sulle problematiche non risolte da questo algoritmo del consenso sono affrontate nella sezione relativa alla robustezza.

Jade

Dovendo risolvere un problema di consenso in un sistema distribuito, si è scelto di usare il framework Jade.

Questo framework supporta nativamente lo sviluppo di applicazioni distribuite e si basa sul paradigma di programmazione ad agenti (quindi adottato anche all'interno di questo progetto).

Oltre ad aver semplificato la programmazione del sistema distribuito tramite il paradigma ad agenti, Jade garantisce anche la consegna dei messaggi. Questo risolve un grosso problema negli algoritmi di consenso: l'affidabilità del messaggero. Infatti in questo modo ci è garantito dal middleware di comunicazione che qualunque messaggio inviato verrà eventualmente consegnato invariato al ricevente non appena possibile.

Realizzazione

Il progetto è stato realizzato come elaborato per l'esame di Sistemi Distribuiti. Nella realizzazione l'unico scopo non era quello di creare una versione funzionante dell'algoritmo in Java, ma anche quello di sperimentare con la programmazione ad agenti, quello di fare un buon uso di Jade e di affrontare alcune delle problematiche nello sviluppo di sistemi distribuiti.

Lo sviluppo è stato suddiviso in due parti:

- quella di sviluppo
- quella di testing e creazione di agenti per gestire l'interfacciamento dell'utente con il framework.

Nella seconda metà del progetto ho speso un buon numero di ore per migliorare la struttura del framework e creare agli agenti l'interfaccia per l'osservazione ed il controllo. In questa fase è stata anche creata la GUI [9] utilizzando i GuiAgent di Jade che implicitamente forniscono delle utility per interfacciarsi con Swing.

Indice dei contenuti:

- UML [2]
- Cluster [3]
- Server agent [4]
- Client agent [5]

Esempi di funzionamento e controllo di correttezza:

Per testare il corretto funzionamento e fare una serie di test, ho sviluppato un cluster di prova composto da 5 server. Nel particolare caso del mio cluster, questo gestisce il valore di una variabile e fornisce tre operazioni: Somma, sottrazione e visualizzazione del valore.

Naturalmente questo è solo un esempio di possibile tipo di dato gestito, ma in realtà può essere mantenuta la coerenza su una qualunque informazione (es. gestione files o database). La configurazione di server e client appositi per il particolare tipo di dato è estremamente semplice, ed in questa pagina [8] è mostrata una guida per ricreare in pochissime righe di codice il cluster che ho usato nei test.

Di seguito una lista dei test che ho effettuato:

- Test elezione
In questo test ho controllato il corretto funzionamento della procedura di elezione di un nuovo leader.
- Test heartbeat
In questo test ho controllato il corretto funzionamento dell'heartbeat, la propagazione della coerenza e l'effettiva applicazione della modifica su tutti i server.

Chapter 2

Progettazione (UML)

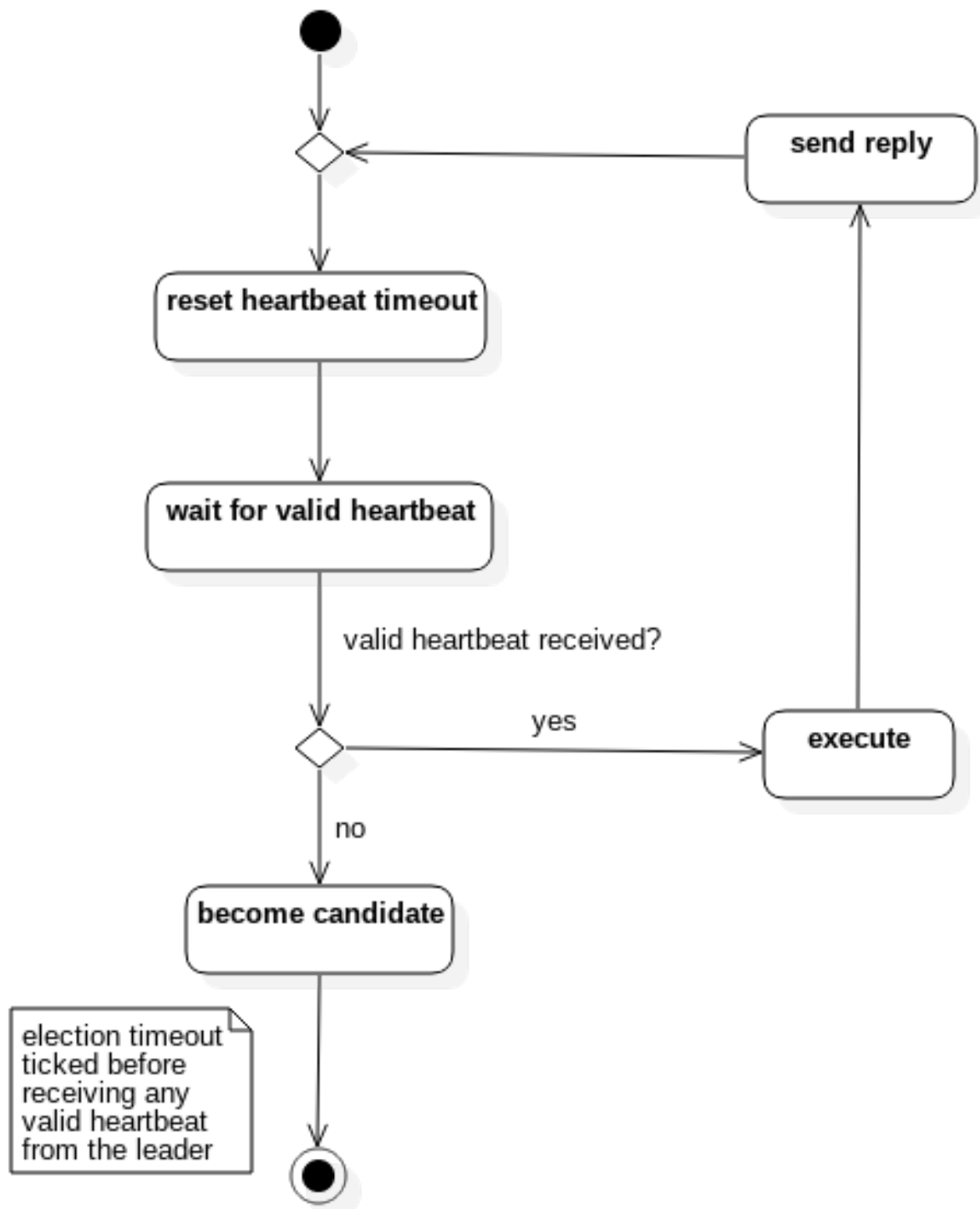


Figure 2.1: Diagramma delle attività del follower.

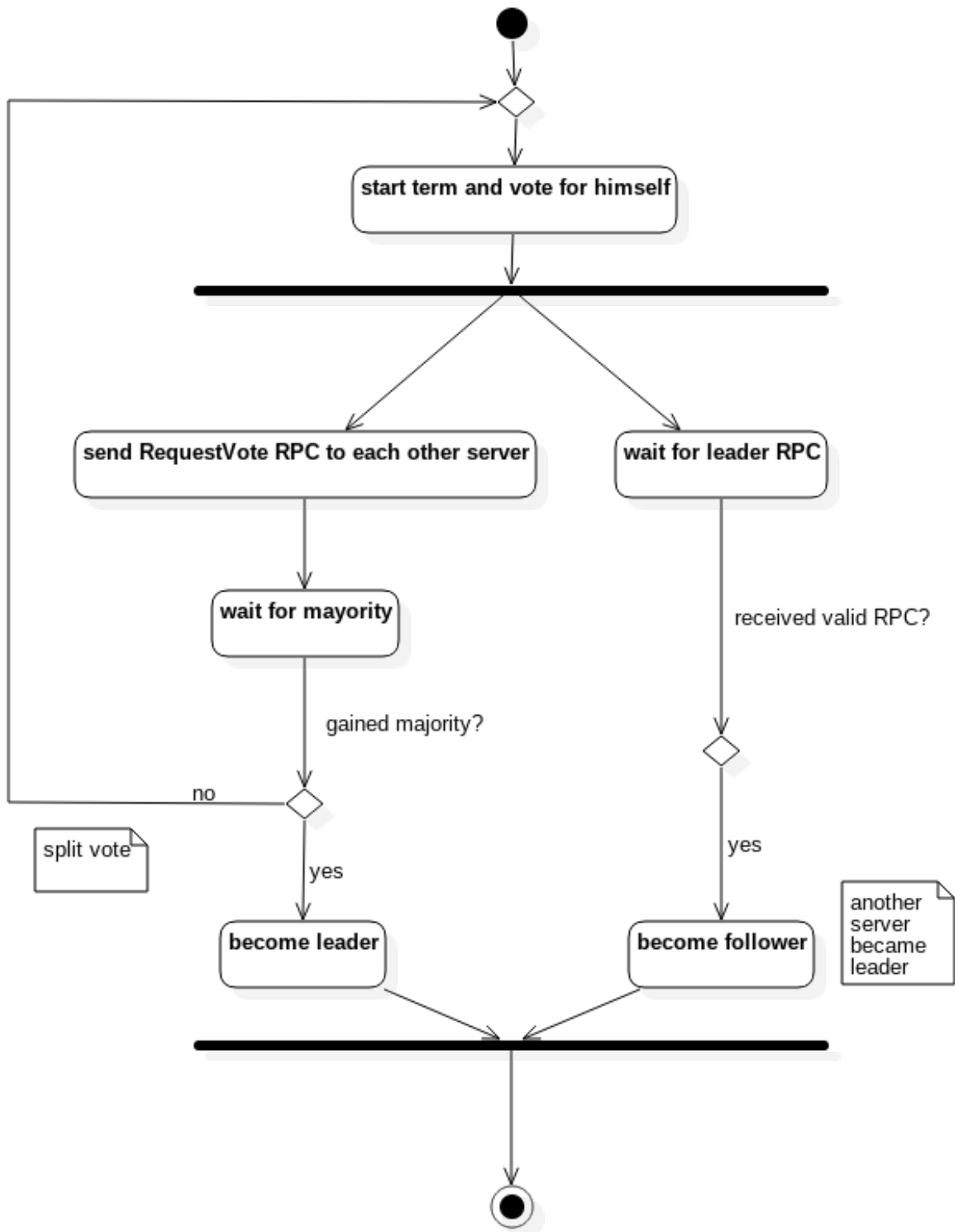


Figure 2.2: Diagramma delle attività del candidato.

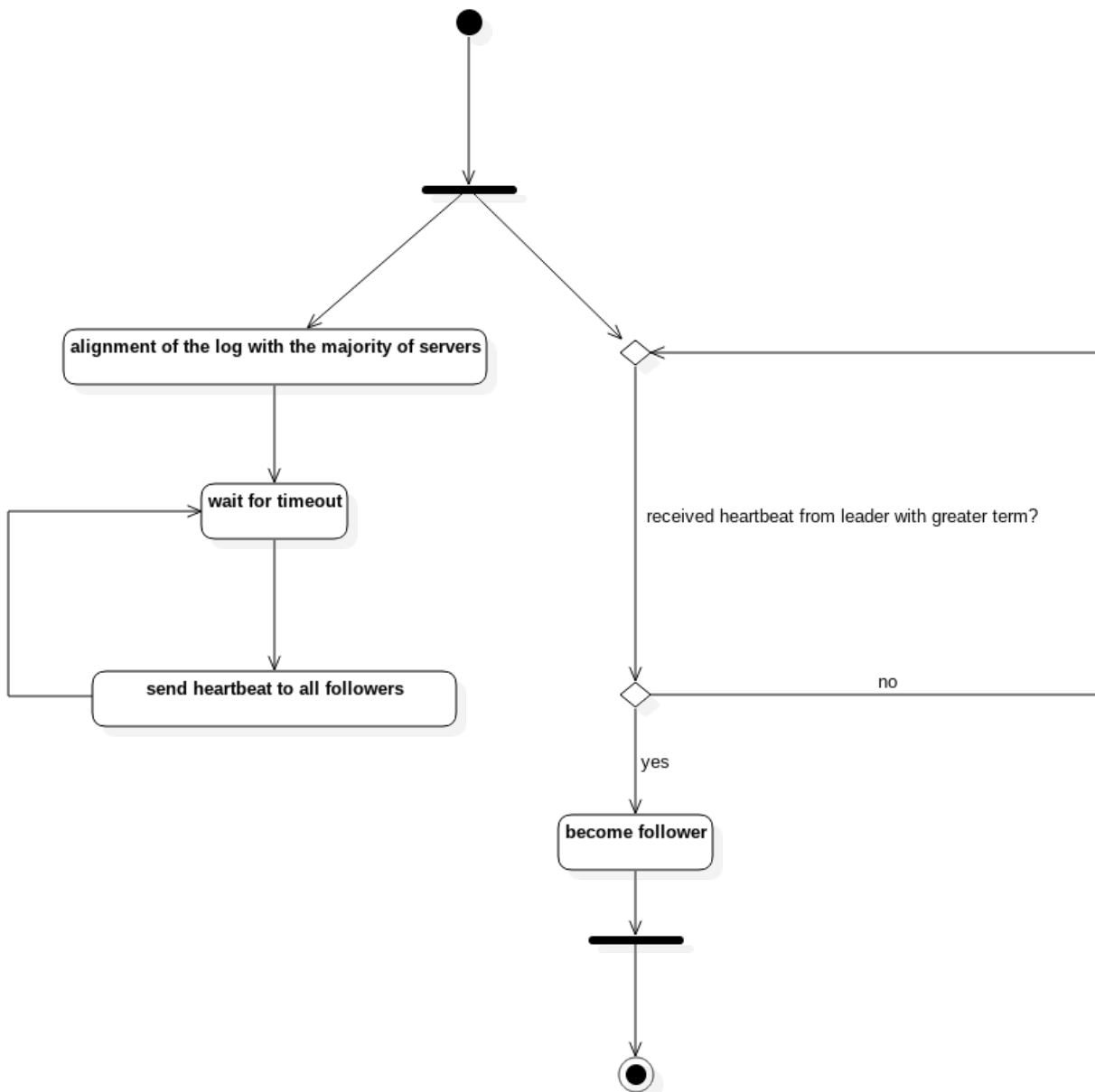


Figure 2.3: Diagramma delle attività del leader.

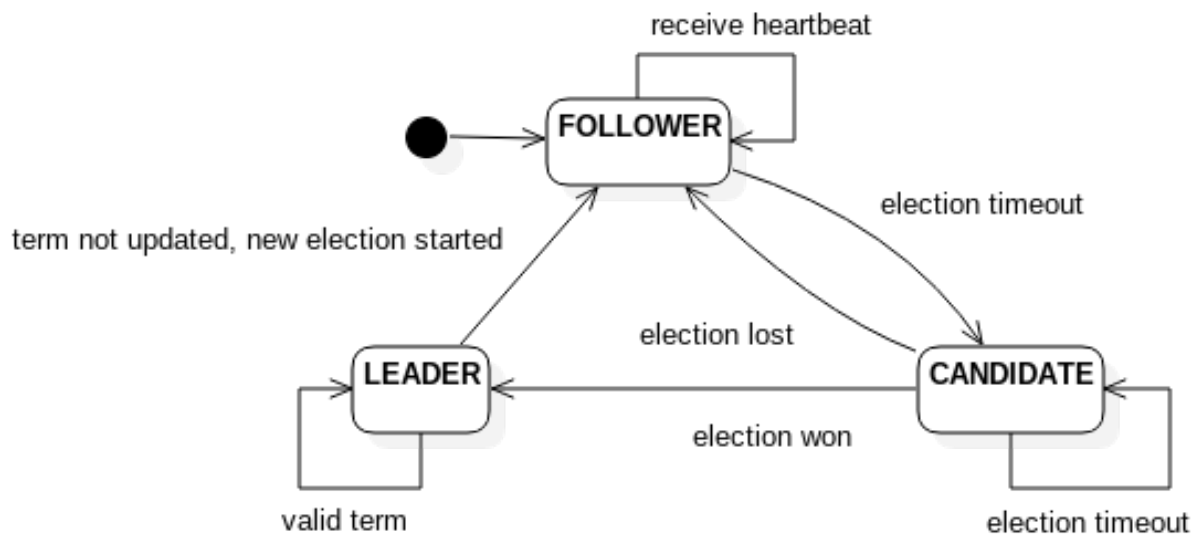


Figure 2.4: Diagramma a stati del cambiamento di stato tra follower, candidate e leader.

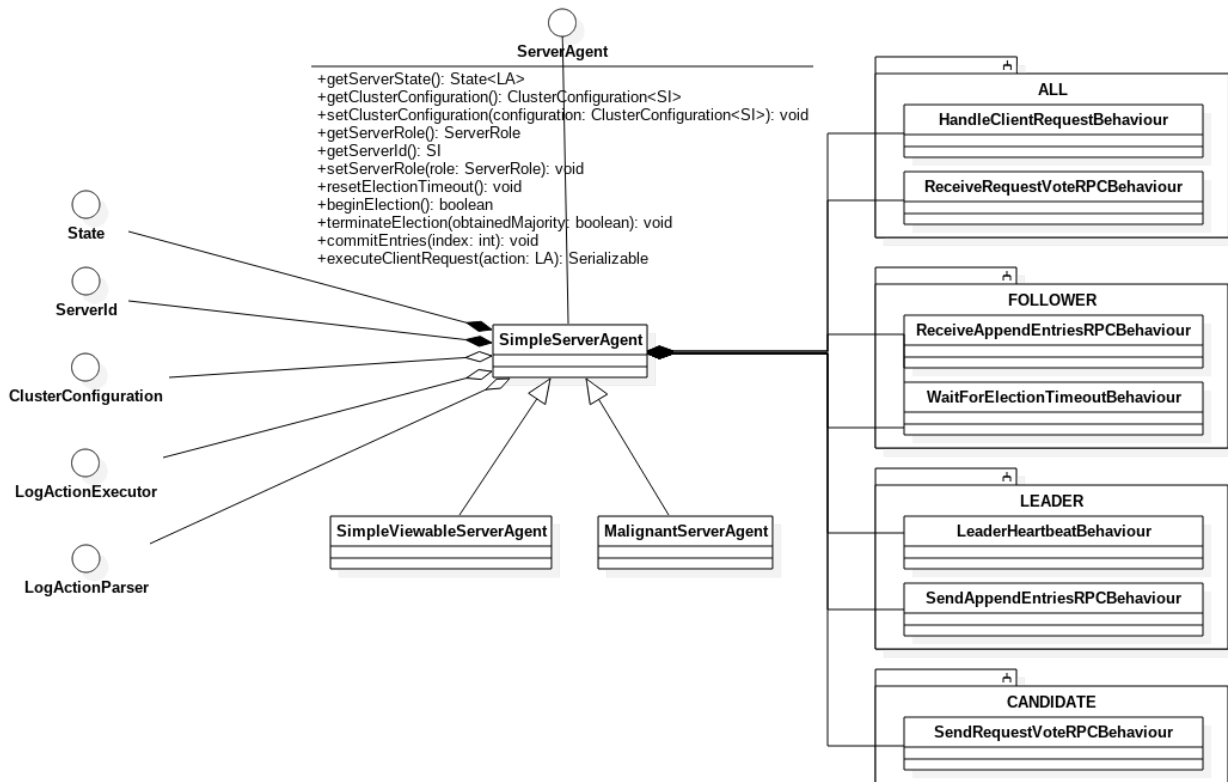


Figure 2.5: Diagramma delle classi della struttura dei server agent.

Chapter 3

Il cluster

Il cluster è l'insieme di server sui quali è necessario mantenere la coerenza; ciascuno di essi è rappresentato da un Jade Agent (il *ServerAgent*) e comunicano tra di loro a scambio di messaggi secondo il protocollo FIPA.

Mantenere coerenza all'interno del cluster vuol dire che non vi sono server che discordano di stato. Per esempio se lo stato del server è il valore del conto in banca di Mario Rossi, vogliamo che ogni server concordi sullo stesso saldo, altrimenti sarebbe estremamente difficile determinare il vero valore del saldo del signor Rossi.

Per mantener tale coerenza si è scelto di usare l'algoritmo di consenso RAFT. Questo algoritmo si basa sul concetto che vi sia un leader a coordinare tutti gli altri server. Tale leader viene eletto e ad ogni elezione di un nuovo leader inizia un nuovo mandato che termina con la candidatura di un altro server. (In realtà è molto approssimativo il funzionamento del mandato descritto in questa maniera, per maggiori dettagli rimando al paper Raft.

La coerenza dello stato si basa sul concetto che ogni azione venga eseguita nello stesso ordine su ogni server, perciò l'aggiornamento degli altri server (*follower*) da parte del leader avviene per passaggio del log. Il log è costituito da una lista di **LogAction** che rappresentano una operazione atomica da eseguire sul server per aggiornarlo allo stato successivo.

Perciò le richieste in lettura possono essere soddisfatte da un qualunque server, mentre quelle in scrittura possono essere soddisfatte solo dal leader (che provvede poi ad aggiornare i follower). È per questa ragione che le richieste in scrittura ad ogni server vengono reindirizzate al leader attivo in quel mandato.

Server	Role	Election timeout	Vote	Term value	Log	Commit index
server1@DSJadePlatform	FOLLOWER	87%	server3@DSJadePlatform	2	3	3
server2@DSJadePlatform	FOLLOWER	86%	server3@DSJadePlatform	2	3	3
server3@DSJadePlatform	LEADER	0%	server3@DSJadePlatform	2	3	3
server4@DSJadePlatform	FOLLOWER	81%	server3@DSJadePlatform	2	3	3
server5@DSJadePlatform	FOLLOWER	87%	server3@DSJadePlatform	2	3	3

Figure 3.1: Uno screenshot della situazione del cluster durante il normale funzionamento. Ogni riga rappresenta un server, le colonne invece sono le varie proprietà di ciascun server.

Chapter 4

Il server agent

Il componente principale del cluster è il server. Ciascun server è stato implementato come singolo agente, e tra di loro possono comunicare solamente tramite scambio di messaggi. Ricordo che per realizzare questa comunicazione ho rispettato lo standard FIPA.

I comportamenti che ciascun server ha in base al suo stato e ruolo (*follower*, *candidate* o *leader*) sono gestiti da specifici **behaviour** che nel seguito andrò a spiegare e che ora semplicemente elenco:

- HandleClientRequestBehaviour
- WaitForElectionTimeoutBehaviour
- SendRequestVoteRPCBehaviour
- ReceiveRequestVoteRPCBehaviour
- LeaderHeartbeatBehaviour
- SendAppendEntriesRPCBehaviour
- ReceiveAppendEntriesRPCBehaviour

HandleClientRequestBehaviour

Questo behaviour si occupa di restare in ascolto di messaggi dal client. Il suo scopo è quello di gestire la comunicazione con il client. In particolare resta in ascolto di messaggi con la seguente struttura:

Mittente: *any*

Performative: REQUEST

Payload: LogAction

QUANDO? Il behaviour viene aggiunto all'avvio dell'agente e deve restare sempre attivo fino alla sua morte.

Ricevuto un messaggio che rispetti questo template, l'agente può rispondere con una delle seguenti possibilità:

Refuse

Nel caso in cui il server non abbia il ruolo di leader. In questo caso nel contenuto della risposta troveremo: LEADER:%GUID_LEADER% (dove %GUID_LEADER% è l'AID globale dell'agente che in questo mandato ha il ruolo di leader).

NotUnderstood

Nel caso in cui il server non sia riuscito a decifrare il payload. Nel payload deve essere specificata la *LogAction* da eseguire. Nel contenuto della risposta viene scritto quello che è il messaggio di errore lanciato nella fase di lettura del payload.

Agree

Nel caso in cui il server abbia il ruolo di leader e sia riuscito a decifrare il payload. In questo caso il leader provvede ad aggiungere la *LogAction* al suo log e ad inviarla a tutti i follower al successivo heartbeat.

A questo punto le possibilità sono due:

- La LogAction viene committata, ed in tal caso viene inviato un ulteriore risposta al client con come performative: **INFORM**;

- Per una qualche ragione si fa rollback ad una LogEntry precedente, in questo caso al client viene inviata una risposta con performative: **FAILURE**.

WaitForElectionTimeoutBehaviour

Questo behaviour si occupa di richiedere l'avvio di una nuova elezione nel caso in cui non riceva alcun heartbeat per un determinato lasso di tempo.

QUANDO? Il behaviour deve essere presente in ogni serverAgent con il ruolo di *follower* o *candidate*, quindi lo si aggiunge (se non presente) quando si passa ad uno di questi ruoli e lo si rimuove quando si passa al ruolo di *leader*.

Naturalmente questo behaviour è proprio dei *follower* e *candidate*, e non del *leader*.

Nel caso in cui l'agente riceva un heartbeat dal leader, allora il timer viene resettato; se il timer scade, allora viene eseguita l'azione specificata, in particolare il server avvia una nuova elezione con la propria candidatura.

SendRequestVoteRPCBehaviour

Questo behaviour si occupa di inviare la candidatura del serverAgent a tutti gli altri server del cluster.

QUANDO? Il behaviour viene aggiunto ogni volta che è necessario effettuare una nuova candidatura del server; al termine del suo compito viene rimosso.

Questo behaviour invia la candidatura del server a tutti gli altri nel cluster. Dopo di che resta in attesa di una loro risposta per controllare se raggiunge la maggioranza o meno. In entrambi i casi, una volta ricevute tutte le risposte termina concludendo le elezioni con un successo (ha raggiunto la maggioranza) o un fallimento.

Il messaggio che invia agli altri server ha la seguente struttura:

Ricevente: %SERVER_AID%
Performative: QUERY_IF
Payload: RequestVoteRPC

ReceiveRequestVoteRPCBehaviour

Questo behaviour si occupa di restare in ascolto di eventuali candidature di uno qualunque degli altri server. I messaggi che può ricevere sono del seguente tipo:

Mittente: *any*
Performative: QUERY_IF
Payload: RequestVoteRPC

QUANDO? Il behaviour viene aggiunto all'avvio dell'agente e deve restare sempre attivo fino alla sua morte.

Ricevuto un messaggio che rispetti questo template, l'agente può rispondere con una delle seguenti possibilità:

Refuse

Se il voto non può essere dato al candidato per una qualsiasi ragione. Nel contenuto della risposta è specificata testualmente la ragione del rifiuto.

NotUnderstood

Nel caso in cui il server non sia riuscito a decifrare il payload. Nel payload deve essere specificato il *RequestVoteRPC*. Nel contenuto della risposta viene scritto quello che è il messaggio di errore lanciato nella fase di lettura del payload.

Inform

Se il *follower* ha dato il suo voto a quel *candidate*.

Il payload della risposta contiene il *RequestVoteReplyRPC*.

LeaderHeartbeatBehaviour

Questo behaviour si occupa di inviare periodicamente un heartbeat a tutti i follower in parallelo (*AppendEntriesRPC*).

QUANDO? Il behaviour deve essere presente in ogni *serverAgent* con il ruolo di *leader*, quindi lo si aggiunge quando si passa ad uno di questi ruoli e lo si rimuove quando si passa al ruolo di *follower* o *candidate*.

Il contenuto dell'*AppendEntriesRPC* è differente per ogni follower sulla base di quali entries deve ancora ricevere. Perciò il behaviour si occupa autonomamente di gestire i differenti invii per ogni follower.

Ad ogni tick costruisce gli *AppendEntriesRPC* per ogni follower e li invia in parallelo con degli *SendAppendEntriesRPCBehaviour*.

SendAppendEntriesRPCBehaviour

Questo behaviour si occupa di inviare un messaggio di append entries ad uno specifico follower.

QUANDO? questo behaviour viene aggiunto ogni volta che si vuole inviare l'heartbeat a quel follower e termina alla ricezione di una conferma o rifiuto da parte del follower.

Il compito di questo behaviour ha anche l'importante scopo di controllare il contenuto dell'eventuale rifiuto in maniera tale che il server possa cedere il ruolo di leader nel caso in cui il suo *termNumber* non sia aggiornato.

I messaggi che invia hanno la seguente struttura:

Ricevente: %SERVER_AID%
Performative: PROPAGATE
Payload: *AppendEntriesRPC*

ReceiveAppendEntriesRPCBehaviour

Questo behaviour resta in attesa di un messaggio di append entries da parte del leader.

Inoltre si occupa di tutta la procedura di aggiunta delle entries al log e loro gestione.

Mittente: *any*
Performative: PROPAGATE
Payload: *AppendEntriesRPC*

QUANDO? Il behaviour viene aggiunto all'avvio dell'agente e deve restare sempre attivo fino alla sua morte.

Ricevuto un messaggio che rispetti questo template, l'agente può rispondere con una delle seguenti possibilità:

Refuse

Se le entries non possono essere accettate per una qualunque ragione. Nel contenuto della risposta è specificata testualmente la ragione del rifiuto.

NotUnderstood

Nel caso in cui il server non sia riuscito a decifrare il payload. Nel payload deve essere specificato l'*AppendEntriesRPC*. Nel contenuto della risposta viene scritto quello che è il messaggio di errore lanciato nella fase di lettura del payload.

Inform

Nel caso in cui le entries siano state accettate e sia andato tutto a termine correttamente.

Il payload della risposta contiene un messaggio contenente l'*AppendEntriesReplyRPC*, che contiene una serie di informazioni richieste dal leader per gestire la coerenza.

Chapter 5

Il client agent

Ogni client che intende eseguire azioni sul cluster è un singolo agente e comunica con il server tramite scambio di messaggi.

Come visto nel server [4], questo ha un behaviour apposito per ascoltare le richieste provenienti dai client; il behaviour principale del client è proprio la sua controparte: il *SendClientRequestBehaviour*.

SendClientRequestBehaviour

Questo behaviour si occupa di inviare le richieste al server di riferimento per il client o al server cui viene reindirizzato. In particolare invia messaggi con la seguente struttura:

Ricevente: *SERVER_RIFERIMENTO*
Performative: REQUEST
Payload: LogAction

QUANDO? Viene aggiunta una nuova istanza del behaviour per ogni richiesta da inviare al server.

Il server può rispondere a questo messaggio in una delle seguenti possibilità:

Refuse

Nel caso in cui il server non abbia il ruolo di leader, allora nel contenuto della risposta troveremo: *LEADER:%GUID_LEADER%* (dove *%GUID_LEADER%* è l'AID globale dell'agente che in questo mandato ha il ruolo di leader) e il behaviour riprova ad reinviare la richiesta al server specificato; altrimenti è un rifiuto e il client rinuncia a soddisfare la richiesta mostrando un errore (quello contenuto nel messaggio).

NotUnderstood

Nel caso in cui il server non sia riuscito a decifrare il payload. Nel payload deve essere specificata la *LogAction* da eseguire. Nel contenuto della risposta è presente il messaggio di errore.

Agree

Quando la richiesta è stata accettata dal leader, che deve provvedere a farla applicare almeno alla maggioranza dei server nel cluster.

Inform

Se, dopo essere stata accettata, la richiesta è stata correttamente committata (applicata sulla maggior parte dei server nel cluster).

Failure

Se, dopo essere stata accettata, la richiesta non viene soddisfatta (per esempio per il fatto che vi è stato un cambio di termine e la *LogEntry* è andata persa).

Chapter 6

Robustezza dell'algoritmo

L'algoritmo di consenso Raft si “fida” molto degli altri server nel cluster e si preoccupa principalmente di risolvere i problemi di coerenza dovuti al fatto che manca la condivisione di memoria e che la rete potrebbe essere lenta e discontinua (ma comunque affidabile).

Punti di forza di Raft

Questo algoritmo di consenso garantisce che il cluster mantiene uno stato coerente: cioè una volta che la maggioranza concorda su una azione, quella viene eseguita prima o poi su tutti i server (commit). Una volta che queste azioni sono state committate, non possono più essere cancellate e il cluster avrà così aggiornato il suo stato.

Nel dettaglio le proprietà garantite sono le seguenti:

- **Election safety:** ad ogni termine può essere eletto al più un solo leader;
- **Leader Append-Only:** un leader non può mai sovrascrivere o cancellare le entries nel suo log, può solo aggiungere le nuove.
- **Log Matching:** se due log contengono un'entry con lo stesso indice e termine, allora i due log sono identici per tutte le entries fino a quella specificata dall'indice.
- **Lader Completeness:** una volta che una entry nel log è stata committata, questa sarà presente anche nel log dei leader per tutti i successivi termini.
- **State Machine Safety:** se un server ha applicato una entry con un determinato indice, allora nessun altro server potrà mai applicare una entry differente con lo stesso indice.

Debolezze di Raft

Di seguito sono descritte alcune delle problematiche che restano aperte avendo adottato questo algoritmo di consenso.

Server malevolo

Raft si basa sul concetto di leader: è il leader che coordina gli altri server (follower) e dice loro cosa fare; se però il leader per una qualche ragione muore o ha problemi, gli altri devono trovare una soluzione quanto prima per poter continuare un normale funzionamento: perciò eleggono un nuovo leader. Per fare ciò è sufficiente che un server si candidi inviando la propria richiesta di voto agli altri server nel cluster, e si presuppone che costoro lo votino, se non hanno già votato un altro leader.

Con questi presupposti un server potrebbe candidarsi appena acceso, in modo da diventare subito leader e prendere il controllo del server.

Oppure potrebbe succedere che il leader non ceda più il suo posto, per esempio continuando a avviare candidature alla fine dei suoi termini.

Rete inaffidabile

Un altro presupposto di Raft è che la rete sia affidabile: o invia i messaggi oppure li perde, ma non li modifica mai. Se il middleware di comunicazione fosse malevolo, potrebbe modificare i messaggi in volo e creare grossi

problemi di inconsistenza tra i vari server nel cluster.

Chapter 7

Test effettuati

Test corretto funzionamento dell'elezione

Elezione

All'avvio del sistema o nel momento in cui un leader non invia più l'heartbeat, è necessario trovare un nuovo leader il prima possibile per poter riprendere il corretto funzionamento.

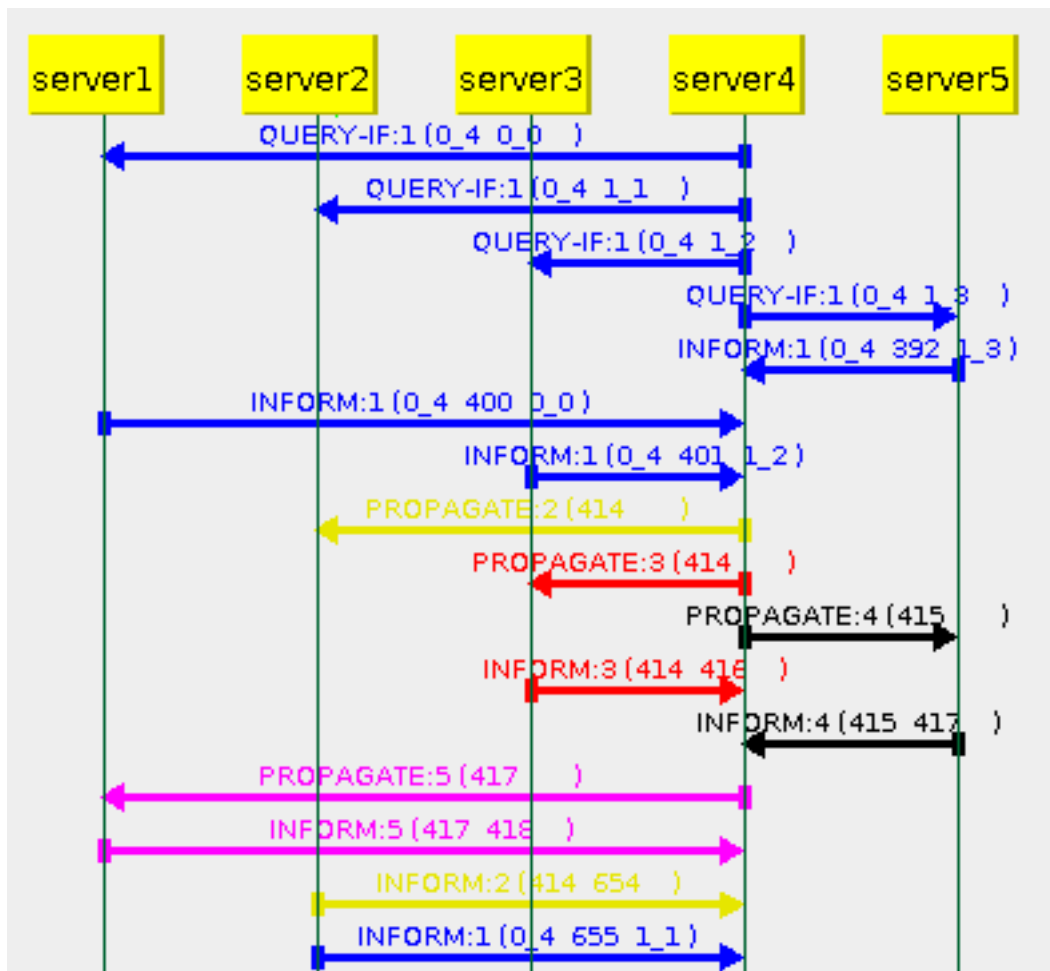
Ogni follower ha un timer configurato con un diverso timeout (tipicamente tra 150ms e 300ms) e al suo scadere si candida per essere eletto.

La candidatura consiste nella richiesta di votarlo agli altri server nel cluster, se questi gli danno il voto, questo server diventa il nuovo leader, se perde le elezioni torna al ruolo di follower, se si verifica uno split-vote (pareggio) si attende nuovamente lo scadere del timer e il primo a cui scade si ricandida per un nuovo termine.

Nella seguente figura è rappresentato lo scambio di messaggi necessario nella fase di elezione, subito dopo la candidatura del server4:

1. Il server4 invia la richiesta di voto con una `QUERY_IF` agli altri server nel cluster;
2. Questi rispondono tutti con una `INFORM` garantendo il loro voto;
3. Il server4 ottiene la maggioranza e diventa il nuovo leader, quindi inizia ad inviare regolarmente il nuovo heartbeat (`PROPAGATE`).

Nella seguente immagine vediamo una rappresentazione grafica dello scambio di messaggi all'inizio di un nuovo termine.



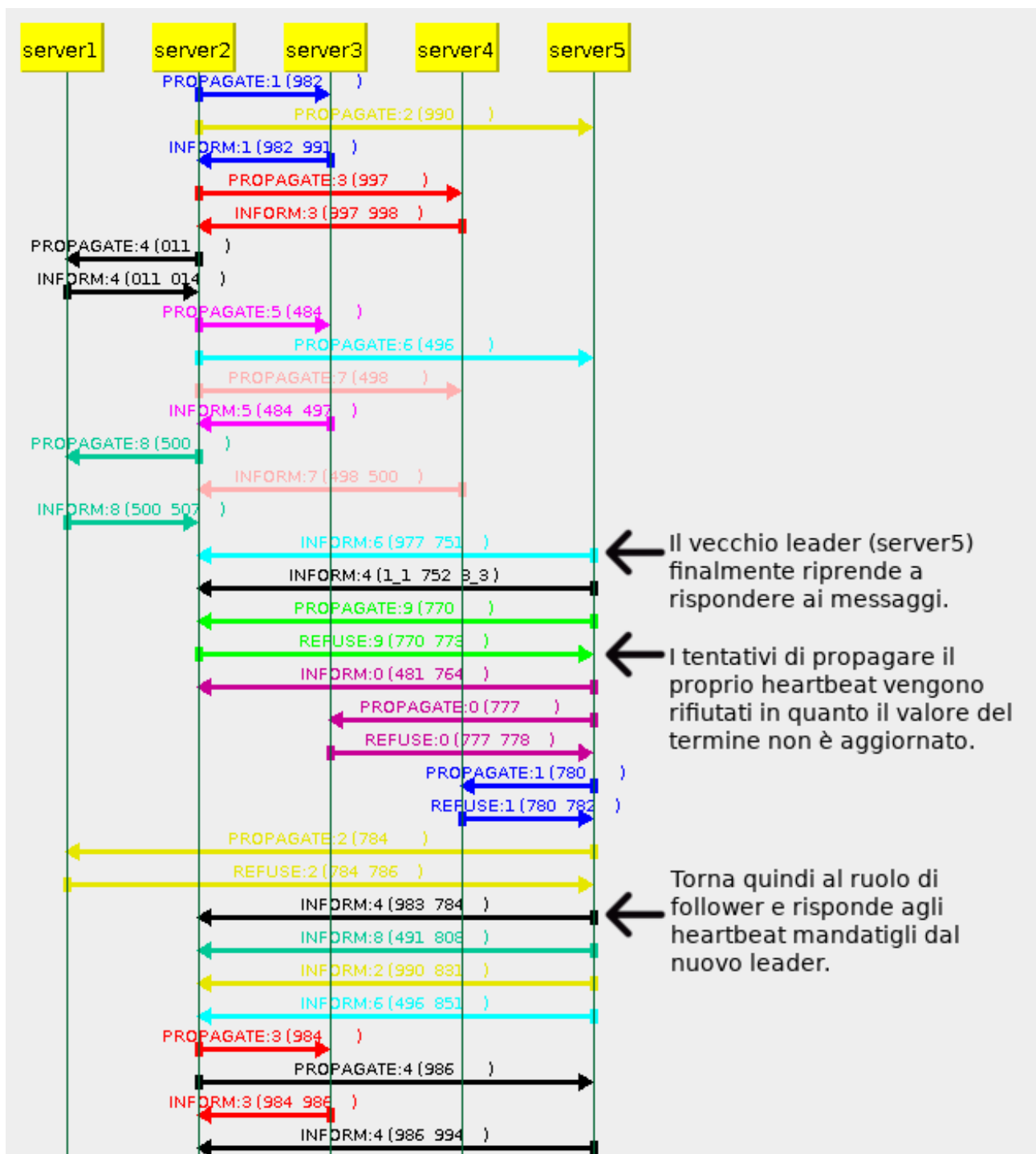
Vecchio leader torna online

Nel caso in cui un leader offline (perchè non riesce a rispondere ai messaggi o la rete momentaneamente non è funzionante) torna online, è necessario gestire correttamente i suoi messaggi in maniera che non vadano a modificare gli altri follower (più aggiornati di lui) e torni anche lui al ruolo di follower il prima possibile per essere aggiornato dal nuovo leader.

Questo consiste principalmente di due parti:

- Ognuno che riceve un suo heartbeat, controlla il suo termine e vede che è meno aggiornato del proprio, quindi non fa altro che rifiutare il messaggio con una **REFUSE**;
- Il vecchio leader, alla ricezione di un rifiuto o di un heartbeat dal nuovo leader che abbiano un *term value* più aggiornato del proprio, retrocede al ruolo di follower immediatamente.

Nella seguente immagine vediamo una rappresentazione grafica dello scambio di messaggi quando il vecchio leader torna online.



Test corretto funzionamento dell'heartbeat

Invio Heartbeat

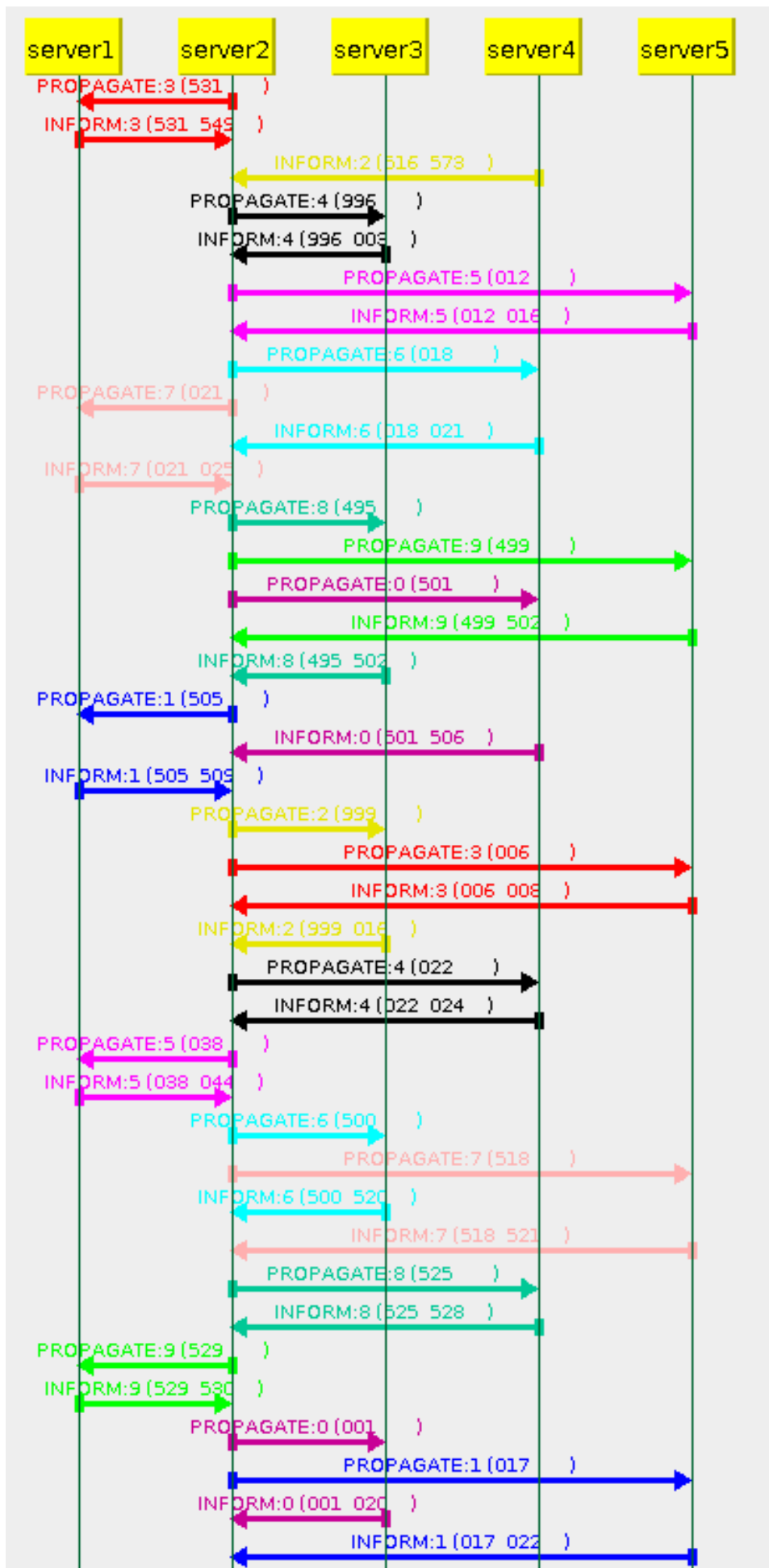
Dopo ogni elezione, il leader eletto inizia ad inviare il proprio heartbeat (un messaggio di AppendEntries) a tutti i propri follower ad un intervallo di tempo regolare. Questo serve per riuscire a mantenere il proprio ruolo di leader all'interno del cluster, altrimenti gli altri follower non saprebbero se il leader è ancora attivo e correttamente funzionante.

Ogni heartbeat viene inviato in parallelo a tutti i follower, ma potrebbe avere contenuti differenti. Infatti l'heartbeat è un AppendEntriesRPC e questo viene usato per propagare il log ai follower. A ciascun follower viene inviato un heartbeat contenente solo le entries del log che questo non ha ancora confermato di aver ricevuto, e queste potrebbero essere differenti da follower a follower.

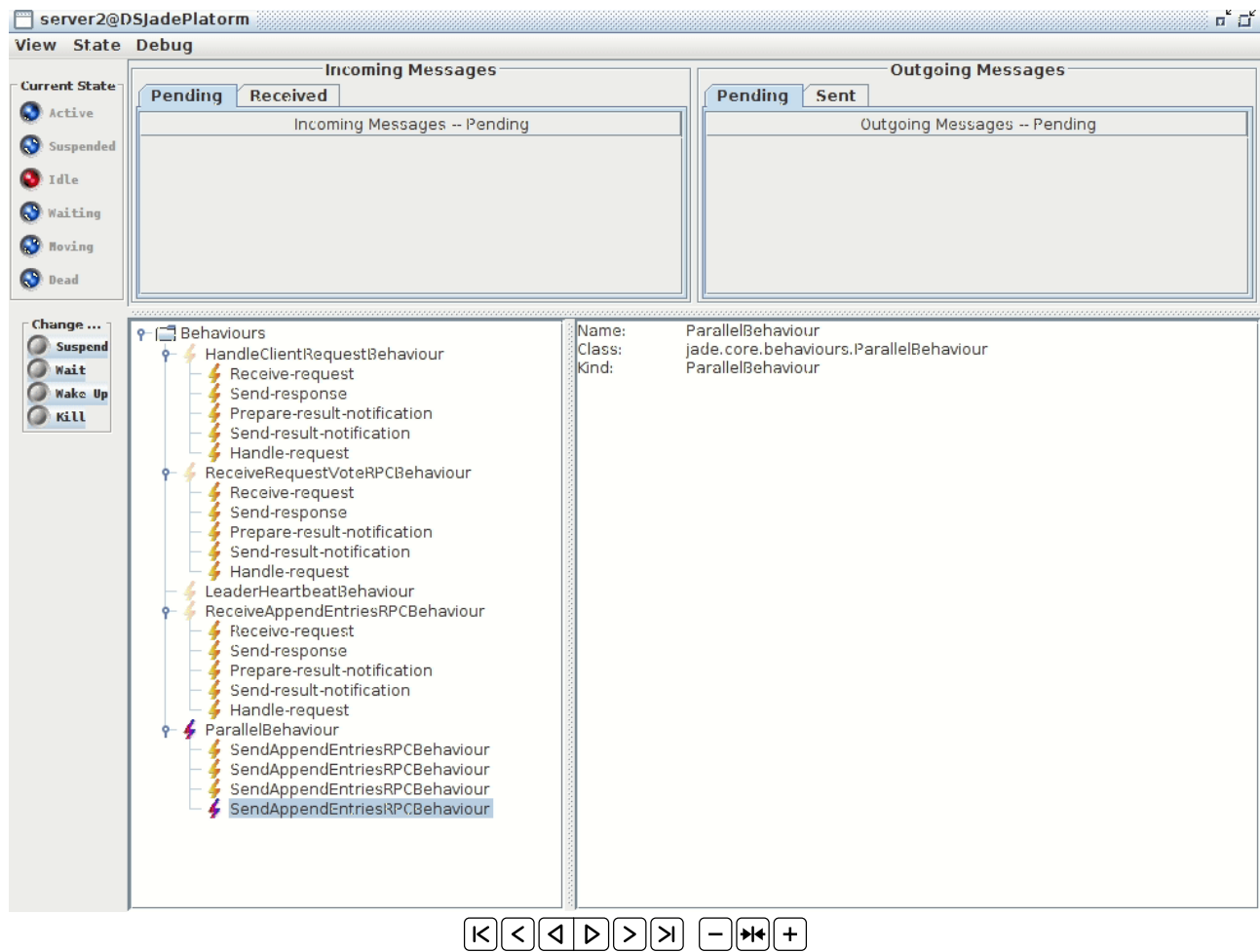
Nel caso in cui non vi siano nuove entries del log da propagare, l'heartbeat inviato è 'vuoto', e cioè non contiene alcuna entry ma solo dei metadati necessari per i controlli.

Durante la serie di test effettuati sono andato anche a controllare che questo comportamento sia rispettato, e così è stato. Ad ogni modo in questo test non scendiamo a questo livello di dettaglio ma mostro solo che la comunicazione tra gli agenti sia effettuata correttamente.

Nella seguente immagine si vede come il leader continui ad inviare heartbeat ai follower e come questi inviino al leader la propria conferma di ricezione (se qualche controllo dei metadati non è passato, allora al leader viene inviato un rifiuto)



Nella seguente gif ho ripreso il comportamento del leader, e cioè ad ogni tick del `LeaderHeartbeatBehaviour`, viene aggiunto il `ParallelBehaviour` che invia l'heartbeat ai vari follower che completa solo quando tutti gli heartbeat sono stati inviati.



Chapter 8

Guida per principianti: creazione del primo cluster

Per dimostrare la semplicità di utilizzo del framework da me creato, vado ora a mostrare in pochi brevi passi come sia possibile creare e configurare un semplice cluster per testare le potenzialità di un sistema distribuito che utilizzi un algoritmo di consenso come *Raft*.

NOTA: All'interno del package `it.unibo.ds.raftexamples.debug` sono presenti le classi che andremo ora a creare.

Al termine della guida otterremo un semplice cluster in grado di mantenere la coerenza sul valore di un numero intero. Le operazioni effettuabili sono:

- **Somma** (`/+%NUM%` dove `%NUM%` è il valore da sommare)
- **Sottrazione** (`/-%NUM%` dove `%NUM%` è il valore da sottrarre)
- **Risultato** (`/?`) che restituisce il valore corrente della variabile

La LogAction

Il cuore di ciascuna configurazione è il tipo di azioni che vengono eseguite su ciascun server. Ogni azione è definita estendendo `LogAction` e può essere personalizzata a piacimento.

Nel nostro caso andiamo a creare una **CalculatorLogAction** che è definita da un tipo dell'operazione e dal valore:

```
public class CalculatorLogAction implements LogAction {

    public enum Operator { SUM, SUB, RES }

    private final Operator operation;
    private final int value;

    CalculatorLogAction(Operator operation, int value) {
        this.operation = operation;
        this.value = value;
    }

    public Operator getOperation() { return this.operation; }

    public int getValue() { return this.value; }
}
```

Per semplificarne l'uso creiamo anche una classe factory che ne crea le varie istanze:

```
public class LogActionFactory {

    private LogActionFactory() {};
```

```

public static CalculatorLogAction createResultLogAction() {
    return new CalculatorLogAction(CalculatorLogAction.Operator.RES, 0);
}

public static CalculatorLogAction createSumLogAction(final int value) {
    return new CalculatorLogAction(CalculatorLogAction.Operator.SUM, value);
}

public static CalculatorLogAction createSubLogAction(final int value) {
    return new CalculatorLogAction(CalculatorLogAction.Operator.SUB, value);
}
}

```

Il server agent

Una volta definita il tipo di azioni che i server devono gestire, andiamo a definire quella che è la struttura di ciascun server all'interno del cluster.

L'executor della log action

Questo è il componente che si occupa della vera e propria esecuzione della log action.

Prende in input l'azione richiesta dal client e restituisce in output il risultato (serializzabile) dell'operazione.

Nel nostro caso andiamo a creare un **CalculatorLogActionExecutor**:

```

public class CalculatorLogActionExecutor extends LogActionExecutor<CalculatorLogAction>() {

    private int res = 0;

    @Override
    public Serializable execute(CalculatorLogAction logAction) {
        switch (logAction.getOperation()) {
            case RES:
                return res;
            case SUM:
                res += logAction.getValue();
                break;
            case SUB:
                res -= logAction.getValue();
                break;
        }
        return null;
    }
}

```

Il suo metodo principale è `execute(...)`, che viene usato da ogni server per eseguire le azioni presenti sul log.

Il parser della log action

Questa classe si occupa di analizzare l'azione per definire innanzi tutto se sia valida o meno e, nel caso sia valida, definisce se è un'azione in lettura o scrittura.

In questo modo la parte logica di classificazione delle azioni è demandata al server, e ciascun server è in grado di definire se riesce a soddisfare la richiesta o se deve reindirizzare il client al leader.

```

public class CalculatorLogActionParser LogActionParser<CalculatorLogAction>() {

    @Override
    public Type getType(CalculatorLogAction logAction) {
        switch (logAction.getOperation()) {
            case RES:
                return LogActionParser.Type.READ;
            case SUM:

```

```

        case SUB:
            return LogActionParser.Type.WRITE;
        default:
            return LogActionParser.Type.INVALID;
    }
}

```

Il metodo principale di questa classe, nonché l'unico che è necessario estendere, è il `getType(...)` che, ricevuta in input l'azione, restituisce in output il tipo.

L'agente

Perché l'agente sia un server, è sufficiente che implementi l'interfaccia *ServerAgent*, ma estendendo *SimpleServerAgent* abbiamo a disposizione un server già implementato e pronto all'uso.

Perciò il server di debug deve estendere *SimpleServerAgent* e passa al costruttore il parser e l'executor appena creati:

```
super(new CalculatorLogActionExecutor(), new CalculatorLogActionParser());
```

Inoltre è fondamentale che al server venga passata la configurazione di default del cluster. Per ora ciò viene fatto staticamente implementando l'apposito metodo come segue:

```

@Override
protected SimpleClusterConfiguration<AgentServerId> getDefaultClusterConfiguration() {
    List<AgentServerId> serverList = new ArrayList<>();
    serverList.add(new AgentServerId(new AID("server1", AID.ISLOCALNAME)));
    serverList.add(new AgentServerId(new AID("server2", AID.ISLOCALNAME)));
    serverList.add(new AgentServerId(new AID("server3", AID.ISLOCALNAME)));
    serverList.add(new AgentServerId(new AID("server4", AID.ISLOCALNAME)));
    serverList.add(new AgentServerId(new AID("server5", AID.ISLOCALNAME)));
    return new SimpleClusterConfiguration<>(serverList);
}

```

ATTENZIONE: Il nome dei server è assegnato staticamente e deve coincidere con i nomi che vengono dati a ciascun server agent quando viene creato.

Modifiche opzionali

Per rendere meglio visibile all'utente il comportamento del cluster, è possibile rallentarlo. Per fare ciò vanno reimplementati i metodi che definiscono i timeout, per esempio possono essere rallentati di **10** volte:

```

@Override
protected int getElectionTimeout() { return super.getElectionTimeout() * 10; }

@Override
protected int getHeartbeatTimeout() { return super.getHeartbeatTimeout() * 10; }

```

Esecuzione

A questo punto della guida, abbiamo già completato la configurazione del cluster ed è perfettamente funzionante. È possibile mandarlo in esecuzione avviando i 5 server agent all'interno della stessa piattaforma JADE.

```

java -cp `%JADE_INSTALLATION%`:bin/ jade.Boot -gui -agents
    server1:it.unibo.ds.raftexamples.debug.DebugServerAgent();\
    server2:it.unibo.ds.raftexamples.debug.DebugServerAgent();\
    server3:it.unibo.ds.raftexamples.debug.DebugServerAgent();\
    server4:it.unibo.ds.raftexamples.debug.DebugServerAgent();\
    server5:it.unibo.ds.raftexamples.debug.DebugServerAgent();

```

È rimasto ora solo da configurare il client per inviare le richieste e testare il corretto funzionamento.

Il client agent

Questo è l'agente che si occupa di inviare le richieste al cluster.

Per semplificare al massimo la sua implementazione, creeremo un semplice agente che all'avvio invia l'azione specificata come argomento al server di riferimento (sempre passato come argomento).

Per configurarlo correttamente i passaggi più importanti sono:

```
setCommunicationServer(new AgentServerId(new AID(%ARG_SERVERNAME%, AID.ISLOCALNAME)));
```

e

```
sendClientRequest(parseLogAction(`%ARG_ACTIONSTRING%`));
```

ho deciso che questo client prende in input i tre comandi nella seguente forma:

- `/+%VALUE%` l'operazione di somma che somma `%VALUE%`
- `/-%VALUE%` l'operazione di sottrazione che sottrae `%VALUE%`
- `/?` la richiesta di lettura del risultato

Perciò il client deve implementare una funzione di parsing analoga alla seguente:

```
private CalculatorLogAction parseLogAction(final String message) {
    try {
        if (message.startsWith("/+")) {
            return LogActionFactory.createSumLogAction(Integer.valueOf(message.split("\\+", 2)[1]));
        } else if (message.startsWith("/-")) {
            return LogActionFactory.createSubLogAction(Integer.valueOf(message.split("-", 2)[1]));
        } else if (message.equals("/?")) {
            return LogActionFactory.createResultLogAction();
        } else {
            return null;
        }
    } catch (NumberFormatException e) {
        return null;
    }
}
```

Eseguendo ora un client è possibile testare il cluster, ad esempio:

```
java -cp `%JADE_INSTALLATION%`:bin/ jade.Boot -gui -agents
    client1:it.unibo.ds.raftexamples.debug.DebugClientAgent(server1,/+2);\
    client2:it.unibo.ds.raftexamples.debug.DebugClientAgent(server2,/?);
```

Chapter 9

GUI (struttura per l'osservazione degli agenti)

Gli agenti server e client sono disponibili anche in versione *osservabile*, e cioè vi è una loro versione dotata di un behaviour che invia un messaggio ogni volta che cambia lo stato ed un behaviour che resta in ascolto di messaggi di richiesta invio dello stato attuale.

Queste versioni degli agenti sono utili per vari scopi, ad esempio per mantenere attivo un logger o per dotare questi agenti di una GUI.

In particolar modo nel corso del progetto si è mostrata necessaria questa funzionalità per lo sviluppo di una semplice GUI per mostrare una demo del framework.

Il server

Il server invia un messaggio all'agente osservante ogni volta che:

- È stato aggiornato il temine
- È stato garantito il voto ad un candidato
- È stata aggiunta una entry nel log
- È stato resettato il log ad un determinato indice
- È stata committata una entry
- È stato aggiornato il ruolo
- È stato resettato l'*election timeout*

Questo comportamento viene gestito direttamente dall'agente tramite il pattern strategy. Infatti i messaggi vengono inviati in maniera asincrona e non è necessario mantenere alcun behaviour in attesa della ricezione di alcuna risposta.

Il client

Il client invia un messaggio all'agente osservante ogni volta che:

- La richiesta non è stata compresa
- La richiesta è stata rifiutata
- La richiesta è stata accettata
- L'esecuzione della richiesta è stata completata
- L'esecuzione della richiesta è fallita

Per il client è stato implementato anche il behaviour che si occupa di accettare richieste da parte dell'osservatore: **ClientGuiToAgentReceiverBehaviour**.

Se il messaggio contiene una LogAction, allora questa viene inviata come richiesta al server di riferimento; altrimenti se il messaggio ha come contenuto KILL, l'agente termina immediatamente; altrimenti si risponde al messaggio esprimendo il fatto che il messaggio non conteneva un contenuto valido.

L'osservatore

L'implementazione dell'agente osservatore è a discrezione di chiunque vada ad usare questa libreria, nell'esempio è costituito da un GuiAgent. La libreria fornisce anche due behaviour che l'osservatore può aggiungere per ottenere la ricezione dei messaggi da parte dell'osservato già implementata.

Chapter 10

Sviluppi futuri

Stress test dell'algoritmo

Sarebbe molto interessante in futuro sviluppare dei server malevoli di vario genere per dimostrare i pregi e limiti dell'algoritmo adottato. Si potrebbero sicuramente testare le proprietà discusse nella sezione sulla robustezza, si potrebbe così analizzare quella che è la complessità insita in questi problemi e meglio comprendere le ragioni per le quali Raft ha deciso di ignorarli.

Cambio dei membri del cluster a runtime

Inoltre andrebbe completato l'implementazione dell'algoritmo. Nella **Sezione 6** del paper si discute come sia possibile cambiare i membri che appartengono al cluster durante l'esecuzione dello stesso senza dover interrompere alcun servizio.

Nello sviluppo del progetto, questa parte non è stata né progettata né implementata, e sarebbe un notevole upgrade del framework sviluppato.

Il cambio di configurazione si suddivide in 3 momenti:

- Nel cluster vi sono solo i vecchi server
- Nel cluster vi sono sia i vecchi che i nuovi server
- Nel cluster vi sono solo i nuovi server

Il passaggio nelle tre configurazioni è determinato dal cambio dei server appartenenti al cluster. Perciò ho dotato ciascun server agent di un metodo per cambiare il set di server appartenenti al cluster: `setClusterConfiguration(%CONFIGURATION%)`.

Il resto dei comportamenti potrebbero essere gestiti da appositi behaviour e la nuova configurazione potrebbe essere propagata tramite apposite LogAction.