AKKA RAFT

Author 1
sara.kiade@studio.unibo.it

Author 2
gyordan.caminati@studio.unibo.it

Author 3
luca.giulianini2@studio.unibo.it

January 2019

Indice

Sommario

Il problema del consenso rappresenta un elemento cruciale nei sistemi distribuiti, dove spesso diverse entità si trovano a dover concordare su un valore da assumere, al fine di convergere al medesimo stato. Nello specifico, il problema si presenta nel momento in cui si ha a che fare con **macchine a stati replicate**, dove la replicazione del dato deve mantenersi **consistente**. Per portare a termine questo compito, sono attualmente disponibili diversi algoritmi di consenso.
Il presente progetto consiste nello **studio approfondito e documentato** dell'algoritmo di consenso **RAFT** e delle sue applicazioni, arricchito da una implementazione che permette di evidenziare le sue peculiarità e di testarne il comportamento in diverse situazioni. Tale implementazione, infatti, fornisce un interfaccia utente tramite la quale è possibile agire su parametri significativi, al fine di poter eseguire agevolmente dei **test** e avere un riscontro degli effetti di questi ultimi sul **comportamento** del sistema.

Capitolo 1

Introduzione

Lo sviluppo in ambito ICT avvenuto negli ultimi anni unito alla informatizzazione massiva dei processi ha portato molte aziende a dover ripensare gran parte dei propri sistemi. Questi sistemi, infatti, erano progettati specificatamente per **environment statici** e a fronte di richieste di grande **scalabilità** ed elevata **dinamicità** mostrarono subito le loro debolezze.

1.1 Sistemi Distribuiti

Gran parte dei sistemi più complessi al giorno d'oggi sono **sistemi distribuiti**, ossia sistemi in cui la computazione non avviene in modo *situato* nella medesima macchina nel medesimo processo ma in modo distribuito fra differenti macchine. Il concetto di distribuzione porta ad una computazione che esula dal **tempo** e dallo **spazio** nella loro rappresentazione **assoluta**.

- **Tempo:** Esiste solo una concezione di **tempo logico** ossia un tempo relativo al singolo sistema distribuito. Per fare ciò vengono usate delle astrazioni *logiche* di tempo chiamate: **vector clocks** [7, Lamport:1978]. Vedremo che con l'algoritmo RAFT verrà utilizzata una astrazione simile per il concetto di `term`.
- **Spazio:** Un sistema distribuito nella sua forma più semplice consiste di un insieme di macchine interconnesse fra loro che eseguono una *computazione* in modo *coordinato* e *decentralizzato*. Il sistema idealmente dovrebbe *astrarre* dalla distribuzione spaziale delle singole unità che lo compongono.

1.2 Il Teorema CAP

Ogni sistema distribuito è caratterizzato da 3 proprietà fondamentali:

- **Consistency:** Vogliamo che un sistema si comporti correttamente, ossia che possa fornire output corretti a fronte di qualsiasi input.

- **Availability:** Vogliamo che il sistema sia sempre disponibile.
- **Partition Tolerance:** Vogliamo che il sistema sia robusto contro contro partizioni dello stesso e quindi crash di qualsiasi componente.

Il teorema **CAP** afferma che in un qualsiasi sistema distribuito si possono garantire al massimo due proprietà [3, Brewer:2000] Esiste dunque un tradeoff fra **safety** e **liveness**.

1.3 Terminologia

Safety Il concetto di consistenza richiama fondamentalmente il concetto di safety secondo il quale:
"*Something bad (ϕ) always not happends*"

$$\Box\neg\phi \tag{1.1}$$

Liveness Il concetto di availability riprende invece quello di liveness secondo cui:
"*Something good (ψ) eventually happends*"

$$\Box\Diamond\psi \tag{1.2}$$

Partition Tolerance: Quando un sistema è *partitioned* allora tutti i messaggi scambiati da un nodo in un componente della partizione a un altro nodo della partizione sono persi.

Componente In un sistema distribuito è formato da vari componenti; ogni componente ha queste caratteristiche:

- **Indipendenza:** Ogni componente è una entità autonoma e indipendente e dunque generalmente disaccoppiata.
- **Eterogeneità:** Non sono date *assunzioni* sulla **natura** delle singole entità sulla loro *struttura* e **comportamento**

Coerenza Un sistema distribuito ha come scopo principe quello di rappresentare e gestire in modo coerente i vari componenti con lo scopo di risultare dall'esterno come un sistema omogeneo *Working as One, Looking as One.*
La coerenza viene ottenuta anche attraverso l'utilizzo di un **Middleware** il quale permette a tutti i componenti di interfacciarsi su un *layer condiviso* che espone la *stessa interfaccia* su architetture e linguaggi diversi.

Interazione Secondo una definizione più specifica fornita da [4, Colouris:2012] un sistema distribuito è un sistema in cui componenti collegati in rete comunicano e si coordinano attraverso **scambio di messaggi**.
L'interazione in questo caso diventa fondamentale per garantire la **coerenza** e l'**uniformità** del sistema.

1.4 Replicazione e Fault Tolerance

Secondo molti un sistema distribuito porta in generale più complessità e addirittura meno vantaggi rispetto ad un sistema situato. Esistono però alcuni proprietà fondamentali che i sistemi situati, che operano nel concentrato, non riescono a garantire.

Replicazione e Consistenza

La replicazione porta parecchi benefici fra cui:

- **Reliability** Il sistema è molto più reliable dato che esistono repliche che possono essere operative anche a fronte di guasti di altre unità.
- **Fault Tolerance** Un componente può fallire mentre il resto del sistema continua a funzionare. Abbiamo dunque la resistenza contro *partial failure* nel sistema.
- **Accessibility** Avendo più repliche a disposizione, si rende più agevole l'accessibilità e il carico può essere bilanciato in modo più semplice.
- **Performance** Non avendo un solo punto di accesso, anche le performance traggono giovamento.
- **Scalability** IL sistema è indipendente dal numero di componenti presenti. Questo permette grande *elasticità* nella gestione dei componenti.

ma anche alcuni problemi che bisognerà arginare:

- **Costo** Il costo della replicazione è direttamente correlato al costo delle macchine utilizzate per replicare i dati. Più hardware viene utilizzato più il costo collegato è elevato.
- **Consistenza** Il problema della consistenza è il problema fondamentale per i sistemi distribuiti. Dato che un sistema multi-replica non garantisce nulla su come e quando verranno replicati i dati, bisogna trovare un modo per garantire che lo stato dei singoli componenti sia consistente con quello del sistema. Questo deve essere fatto anche a discapito di altre proprietà come la **availability** come già visto nel teorema CAP.

Fault Tolerance

Lo scopo fondamentale della fault tolerance è quello di permette al sistema di **sopravvivere** a e in alcuni casi **risolvere** autonomamente le **failures**. Generalmente questo comportamento viene ottenuto limitando l'impatto della failure sull'intero sistema cercando di sfruttare i componenti integri al fine di risolvere gli stessi fallimenti.

Capitolo 2

Consenso e State Machine Replication

Il principale vantaggio fornito dai sistemi distribuiti rispetto ai sistemi centralizzati sta nel fatto di poter garantire un servizio continuo malgrado *failures* e *malfunzionamenti* [5, Friedman:1996]. L'idea infatti è quella di poter realizzare sistemi che possano far fonte a componenti inaffidabili che non possono garantire una disponibilità costante; questi sistemi devono essere in grado di *reagire* e *adattarsi* automaticamente ai cambiamenti.
Fra le sfide più importanti che i sistemi distribuiti devono affrontare al giorno d'oggi troviamo: la gestione delle failures, la coordinazione, il discovery di servizi e la gestione della configurazione; tutti problemi difficili da risolvere in ambienti dinamici. Fortunatamente il consenso distribuito permette di risolvere gran parte di questi problemi.

2.1 Il problema del Consenso

Gli algoritmi di consenso rappresentano gli algoritmi fondamentali e più importanti nell'ambito dei sistemi distribuiti. Gli algoritmi di consenso permettono a una collezione di **macchine eterogenee** di lavorare in qualche modo come se fossero un **una sola macchina**.
Si parte dunque con un insieme di **macchine** singolarmente **inaffidabili** e si fa in modo che esse si comportino come un'**unica macchina super-affidabile** che garantisce di operare in modo reliable anche nel caso in cui alcune macchine falliscano.
All'interno di un gruppo di consenso (**consensus group**) i fallimenti sono gestiti in un modo studiato e comprovato, questo permette di avere un *modello di funzionamento* robusto e affidabile.
Data l'estremam **availability** e **raliability** dei consensus group, altri sistemi possono decidere di usare un consensus group come fondamenta per garanti-

re fault tolerance; questo porta gli algoritmi di consenso a giocare un ruolo fondamentale nella costruzione di sistemi reliable di grandi dimensioni.

2.1.1 Proprietà degli algoritmi di consenso

Gli algoritmi di consenso applicati a sistemi come le **Replicated State Machine** garantiscono queste proprietà al sistema:

- **Safety:** Non ritornano mai un risultato incorretto sotto **condizioni non bizantine**. Un algoritmo di consenso generalmente è safe rispetto a gran parte dei problemi che affliggono le **reti**:
  - *network delay*
  - *network partition*
  - *packet loss*
  - *duplication*
  - *reordering*
- **Liveness:** Gli algoritmi di consenso funzionano bene quando la maggioranza dei server sono operativi e di conseguenza sono in grado di comunicare fra loro o con clients. Empiricamente è stato dimostrato che un cluster di cinque server, mantenuti attraverso consenso, può tollerare il fallimento di 2 server.
  E' importante sottolineare che il stato di **fail** per un sistema non è definitivo ma, come abbiamo già detto, è possibile che alcuni componenti possano recuperare dallo stato di failure e tornare all'interno del cluster attraverso un processo di **joining**.
- **Indipendenza temporale:** La **consistenza** del sistema (dei log nel caso di una macchina a stati replicata) fornita da un algoritmo di consenso è **indipendente** dal concetto di **tempo** in senso assoluto. Ciò significa che è possibile che siano presenti ritardi e perdite di pacchetti ma questi problemi causeranno solamente problemi di **availability** del sistema, in accordo a quello che dice il teorema **CAP** [3, Brewer:2000].

2.2 Replicated State Machine

Gli algoritmi di consenso sono generalmente utilizzati nel contesto di quelle che chiamiamo replicated state machines. Prima di parlare di macchine a stati replicate bisogna definire cos'è una macchina a stati.
Una macchina a stati è generalmente un programma che:

- *accetta e risponde* a vari tipi di interazioni o **stimoli esterni**.
- Gli stimoli possono essere nella forma di *comandi* che provengono da clients e hanno lo scopo di gestire lo **stato interno** della macchina.

Al giorno d'oggi le state machine rappresentano un modello generale di computazione estremamente diffuso. Gran parte dei sistemi a larga scala (As-A-Service) che utilizziamo solitamente sono nella forma di macchine a stati.

2.2.1 Struttura di un server con SMR

Come si può vedere nella figura 2.1, all'interno di un sistema sono presenti solitamente diversi server di tipo SMR. Il numero di unità presenti è direttamente correlato alla robustezza della State Machine replicata.

Consistenza nella computazione del log

Log → Le macchine a stati replicate sono implementate solitamente utilizzando un **log replicato** (modulo `log` fig: 2.1). Ogni server implementa un log il quale contiene all'interno una **serie di comandi** che sono mantenuti sempre aggiornati e in **ordine** in modo *consistente* con gli altri componenti presenti nel sistema.

State Machine → I comandi presenti all'interno del log vengono **eseguiti in ordine**, sotto particolari condizioni, dalla state machine (modulo `state machine` (fig: 2.1). Le state machine sono macchine **deterministiche** questo sta a significare che producono lo stesso output e si portano sullo stesso stato a fonte dello stesso input. Questo fatto è fondamentale per il sistema!

Osservazione 1: *Se il log è mantenuto sempre in ordine e la sua computazione da parte della state machine è deterministica allora siamo certi che tutte le state machine agiranno nello stesso modo a fronte dello stessa serie di comandi presente nel log.*

Consistenza nello stato del log

Quello di mantenere consistente lo stato del log è un lavoro dell'algoritmo di consenso il quale è contenuto in un modulo a parte (modulo `consenso` (fig: 2.1).

Modulo Consenso → Il modulo del consenso ha uno scopo ben preciso che può essere riassunto come segue:

1. Una volta che il server riceve un comando da parte di un client, il **comando viene appeso** in fondo al log.
2. Il modulo di consenso si assicura che la **maggior parte** dei server presentino l'ultima **entry** del log (in questo caso una richiesta da parte del client) nello **stesso ordine** e nella **stessa posizione** del log.
3. Se una percentuale inaccettabile di server è inconsistente allora l'algoritmo può decidere di continuare ad **attendere** fino a quando il numero di server consistenti aumenta.

4. Una volta che il numero è stato raggiunto allora la entry viene definita **committed**.

5. Ogni state machine, ricevuto il commitment dell'entry, può finalmente **eseguire** il comando.

6. L'output è infine ritornato al client.

Osservazione 2: *Se l'algoritmo di consenso da il via libera per il commit di un comando allora è lecito pensare che il sistema sia safe e possa essere consistente da quel momento in poi in tutto il sistema.*

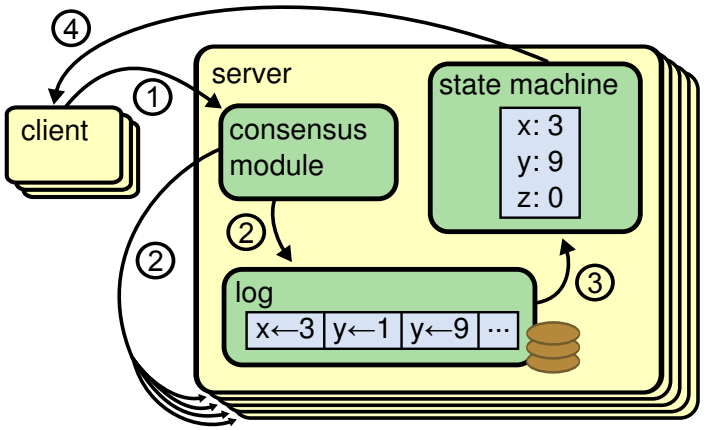

Figura 2.1: Un modello di state machine semplificato

2.2.2 Usi Pratici delle SMR

Le macchine a stati replicate vengono generalmente considerate come blocchi general pourpose su cui basare sistemi più complessi al fine di renderli fault-tolerant. Esistono vari paradigmi di utilizzo delle state machine: solitamente il pattern più comune consiste nell'utilizzo di una state machine molto **semplice**, formata da tre o cinque server, sulla quale si appoggiano altri server specifici con lo scopo di **coordinare** le proprie **attività**.
Molti sistemi di storaging utilizzano questo principio, fra i più importanti possiamo trovare: GFS, HDFS, RAMCloud.

Capitolo 3

RAFT

Come abbiamo visto nel capitolo precedente la struttura di una macchina a stati replicata è generalmente molto semplice. Questo è dato dal fatto che ogni singola state machine può essere decomposta in singoli **moduli** dedicati a uno **specifico scopo**.

- *Log Module*
- *State Machine Module*
- *Consensus Module*

I moduli sono **semi-dipendenti** ossia comunicano tra di loro attraverso chiamate a procedure standard. Per esempio il modulo di log potrebbe esporre funzionalità dedicate all'appending o commiting di una serie di comandi, o ancora la state machine potrebbe mettere a disposizione un metodo per l'esecuzione di un determinato comando.
In questo capitolo andremo a soffermarci maggiormente sul core-module di ogni SMR ossia il **modulo di consenso**; capiremo quali sono le sfide che un algoritmo di consenso deve affrontare e vedremo in dettaglio un'implementazione di un famoso algoritmo di consenso, **RAFT**.

3.1 Approcci al Consenso

Esistono generalmente due approcci al problema del consenso:

- **Symmetric | Leader-less:** in questa tipologia i server hanno tutti lo **stesso ruolo** e tutti lo stesso potere decisionale in qualsiasi momento.
  I **client** possono quindi **contattare** e fare richiesta verso **qualunque server** presente all'interno del cluster.
- **Asymmetric | Leader-base:** in qualsiasi momento i server non hanno **mai** lo **stesso ruolo** e lo stesso potere decisionale. Esiste infatti sempre

un server che si trova **in carica**. Questo particolare server viene denominato **leader**. Il leader gestisce solitamente tutte le operazioni del cluster, mentre gli altri server sono denominati **servant** e portano a compimento le richieste del leader.
In questo tipo di sistema i **client** possono **comunicare solamente con il leader**; nel caso in cui un client contatti un server diverso dal leader viene immediatamente **rediretto** al leader corrente.

Vedremo che RAFT implementa la seconda versione di consenso; questo garantisce all'algoritmo di essere estremamente più semplice dei suoi predecessori leader-less garantendo comunque lo stesso grado di safety.

3.2 L'algoritmo PAXOS

Paxos è un algoritmo di consenso, non resistente a Byzantine Fault, sviluppato verso la fine degli anni ottanta e pubblicato nel 1998. Esso ha ricoperto il ruolo di algoritmo standard per decenni, venendo utilizzato in diversi ambienti, compreso quello didattico, poichè rappresenta un'ottima alternativa al commit a due fasi e a quello a tre fasi che si affidano entrambi ad un unico coordinatore che rappresenta un single point of failure. Esistono molte varianti di Paxos, ma la più usata si chiama *Single Decree Paxos*.

In Paxos, i nodi possono assumere contemporaneamente uno o più dei seguenti ruoli:

- **Proposer:** propone un valore su cui accordarsi a tutti gli acceptor.
- **Acceptor:** riceve le proposte dei proposer, decide se accettarle o meno e inoltra la propria risposta ai learner.
- **Learner:** sulla base delle risposte ricevute, determina come valore vincitore quello scelto dalla maggioranza degli acceptor.

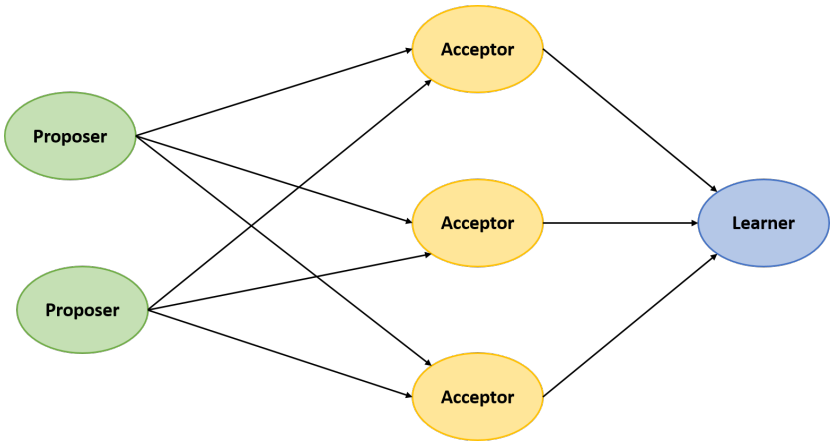

Figura 3.1: Esempio di configurazione dei ruoli con due **proposer**, tre **acceptor** e un **learner** che sono rappresentati come nodi distinti per semplicità: nella realtà un nodo può ricoprire più di un ruolo alla volta.

Qualsiasi criterio si scelga per far decidere agli acceptor quale valore accettare, si incorre in casi di **incorrettezza** a meno di non utilizzare un approccio in due passi. Il raggiungimento del consenso si sviluppa quindi in due fasi.

1. **Proposal:** in questa fase i proposer inviano una richiesta di ***prepare(n)*** agli acceptor, dove n è un *proposal number* scelto in maniera tale che sia maggiore di ogni altro numero incontrato fino a quel momento.
   Gli acceptor ricevono le proposte e, per ognuna, controllano se il valore corrispondente è il maggiore in cui si siano imbattuti fino a quel momento: in caso affermativo, rispondono alla proposta impegnandosi a non accettare proposte precedenti, altrimenti la ignorano.
   Se un proposer riceve una risposta alla propria proposta dalla maggiorparte degli acceptor, passa alla seconda fase.

2. **Accept:** in questa fase, il proposer invia il **proposal number** e il *value* tramite una richiesta ***accept(n,v)*** rivolta a tutti gli acceptor.
   Se il proposer si vede ritornare come risposta il proprio *proposal number* valore, allora la maggioranza degli acceptor ha concordato sul valore proposto, che quindi sarà quello scelto.
   Nel caso in cui ciò non accada, il proposer si vedrà arrivare un *proposal number* maggiore del poprio, di conseguenza ricomincerà l'iter, tornando all'inizio della prima fase.

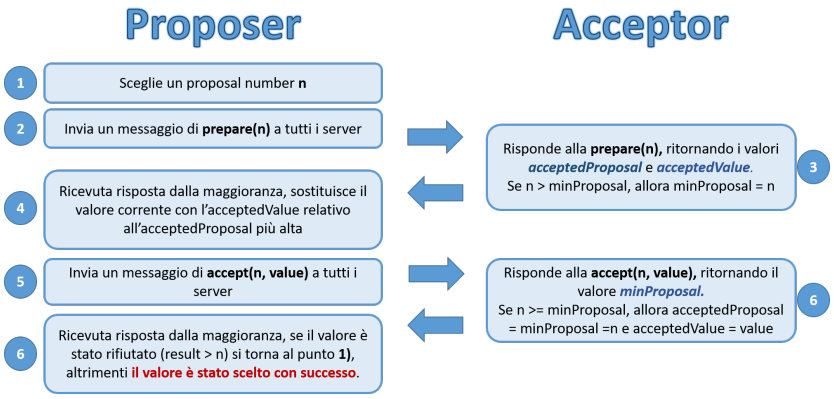

Figura 3.2: Schema riassuntivo delle interazioni tra proposer e acceptor, che non tiene conto, per semplicità, dei learner

L'algoritmo Paxos non è così semplice come potrebbe sembrare a una prima lettura, ma nasconde una serie di dinamiche complesse che non verranno analizzate in questo elaborato poichè è sufficiente un'infarinatura sul funzionamento generale di Paxos per capire in cosa differisce RAFT.

3.2.1 Problemi di Paxos

Nonostante sia ancora largamente usato, l'algoritmo Paxos presenta dei difetti che RAFT ha provato a colmare:

- **Difficoltà di comprensione:** la parte teorica e le dinamiche dell'algoritmo sono di difficile comprensione e spiegazione, come evidenziato anche dagli esperimenti condotti dagli autori di RAFT.
- **Difficoltà di implementazione:** anche una volta assimilata la parte teorica, è difficile applicarla. Quando si scende nei dettagli implementativi si va in contro a una confusione generale data dalla presenza di diverse interpretazioni, spesso errate.
- **Inefficienza:** prima che ci si possa accordare su un valore sono necessari almeno due round.
- **Valore unico:** il consenso viene raggiunto su un unico valore, per tutta la durata vitale del sistema.
- **Convergenza:** non è garantito che l'algoritmo converga, arrivando a una soluzione. L'unica garanzia è che se converge, il valore scelto sarà uno e uno solo.

3.3 RAFT Core

Quando si progetta un nuovo algoritmo è importante concentrasi sin da subito sui **metodi di valutazione** del risultato atteso. Generalmente i criteri utilizzati per la valutazione di algoritmi sono:

- **Correttezza:** se l'algoritmo fornisce il risultato atteso in tutte le condizioni.
- **Efficienza:** il numero di cicli di cpu o la quantità di memoria utilizzata dall'algoritmo. Solitamente questi due indicatori sono negativamente correlati, questo significa che cercando di ottimizzare un algoritmo sotto il profilo dei cicli di cpu si tenderà ad utilizzare più memoria.
- **Concisione:** un algoritmo è conciso quando le sue specifiche sono definite in modo preciso e chiaro.

Nel caso di RAFT, gli ideatori **Diego Ongaro e John Ousterhout** [9, raftPaper] [8, ongaro:2014] hanno deciso di usare un approccio inusuale, il loro scopo era prima di tutto quello della **understandability**[1].
Nella progettazione di RAFT sono state applicate varie tecniche volte a garantire undestandability:

- **Decomposizione del problema:** tecnica che opera in modo simile al divide et impera degli algoritmi. Nella decomposizione il problema viene suddiviso parallelamente in problemi più piccoli e più semplici da risolvere.
- **Minimizzazione dello spazio degli stati:** la minimizzazione dello spazio degli stati può essere vista in modo molto naive come la riduzione del numero di istruzioni condizionali presenti. Questa procedura garantisce come piacevole conseguenza altre proprietà estremamente importanti:
  - **Gestione di problemi multipli con lo stesso meccanismo;**
  - **Eliminazione degli stati speciali;**
  - **Massimizzazione della coerenza e la consistenza;**
  - **Minimizzazione del non-determinismo.**

RAFT in breve RAFT implementa il **consenso** eleggendo prima di tutto un `leader` e fornendo allo stesso la totale responsabilità della gestione dei log replicati. Il leader una volta eletto **accetta** delle richieste dai `client`, le **replica** sugli altri server e **comunica** ai server quando è safe eseguire le stesse richieste nelle proprie state machines.
L'esistenza del leader semplifica di molto l'algoritmo poiché garantisce un minor numero di stati del sistema. Nel caso in cui un leader fallisca o venga disconnesso verrà eletto un nuovo leader garantendo il progresso dell'algoritmo.

[1] Un algoritmo pensato per essere comprensibile implicitamente obbliga l'autore a dover pensare in modo semplice e come sappiamo per il rasoio di Occham, la soluzione più veloce e comprensibile è generalmente quella migliore. Nel caso dei sistemi distribuiti questo è ancora più vero dato che in questo caso è importante sapere non solo che un algoritmo funzioni ma sopratutto perché funzioni.

RAFT e decomposizione RAFT decompone il problema del consenso in tre problemi relativamente indipendenti:

- **Leader Election:** consiste nella **detection** di una **failure** da parte di un leader e nella definizione di un **protocollo di rielezione** che possa essere consistente.
- **Log Replication:** il leader deve essere l'unico server che può accettare comandi da parte di un client. Inoltre il leader deve assicurarsi di poter **replicare in modo safe** una entry presente nel log.
- **Safety:** la proprietà di safety, chiave di RAFT, è data dalla State **Machine Safety Property**.

3.3.1 Basics

Un cluster RAFT può contenere vari server, solitamente cinque è il numero tipico, questo permette al sistema di tollerare due fallimenti.

Stati di un server

Abbiamo tre stati in cui si può trovare generalmente un server:

- **Leader:** in operazioni normali esiste esattamente **un solo leader** mentre i rimanenti sono follower. Quando il leader viene eletto allora esso ha **pieno potere** e tutti i rimanenti follower devono seguire i suoi ordini.
  Il leader solitamente ha due compiti fondamentali:
  - *Gestione delle richieste da parte dei client*
  - *Replicazione del log*
- **Follower:** I follower sono server totalmente **passivi**, essi intraprendono tre azioni fondamentali:
  - *Risposta alle richieste di leader e candidate*
  - *Redirection al leader delle richieste da parte dei client*
  - *Se non hanno più notizie dal leader da un po' allora diventano "ansiosi" e si convertono allo stato di candidate*
- **Candidate:** Lo stato di candidate è usato per **eleggere un nuovo leader**.

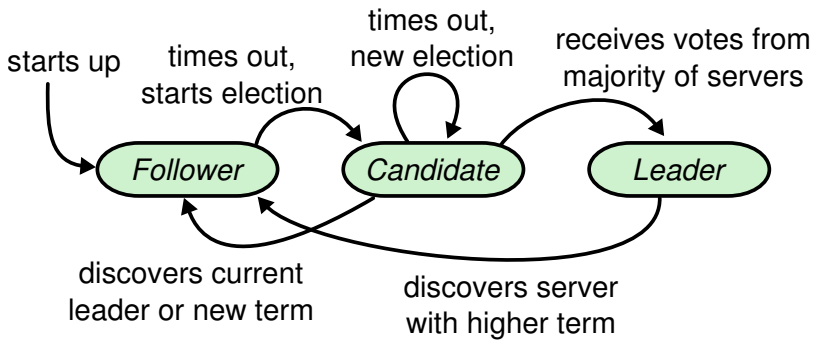

Figura 3.3: Diagramma a stati che descrive il passaggio di stato dei singoli server

Terms

Il tempo in RAFT è diviso in periodi di **lunghezza arbitraria** chiamati **terms**. I term godono di queste proprietà:

- **Identificativo:** a ciascun term è associato un numero in modo **incrementale** e **assoluto** nel tempo.
- **Leader-centric:** ogni term parte con l'elezione di **uno e un solo leader**: se l'elezione ha successo allora il candidate eletto avrà potere per tutta la durata del term. Può capitare che più candidate provino a diventare leader nello stesso instante: in questo caso il risultato della votazione premierà un solo candidate. RAFT garantisce che non ci sia *più di un leader per term*.
- **Leader-less:** in alcuni casi può capitare che nessun leader emerga, questo avviene perché nella votazione non è stata ottenuta una maggioranza assoluta. In questo caso si dice che è avvenuto uno **split-vote** e, al fine di garantire safety, *nessun leader può essere eletto*.
  In questo caso l'algoritmo considererà l'elezione come **failed** e verrà aperta immediatamente una **nuova elezione**.
- **Term come logical clock:** i term in RAFT hanno uno scopo molto più generale oltre al contesto dell'elezione: essi agiscono come **logical clock** [7, Lamport:1978] garantendo un concetto logico di tempo e permettendo ai server di accorgesi quando sono in possesso di **informazioni obsolete**.

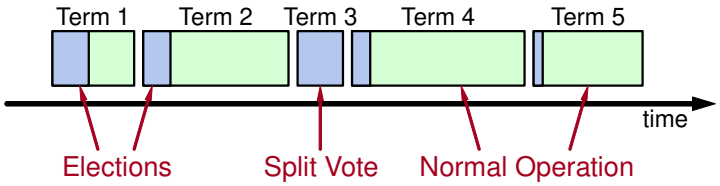

Figura 3.4: Un term è un periodo di tempo che parte con l'elezione di un nuovo leader e termina con la fine del suo "regno".

Identificazione di informazioni obsolete Non esiste in RAFT un concetto di stato/term globale ma ogni **server vede**, in un **dato istante**, una **parte del sistema** e non è detto che sia lo stesso per tutti.
L'identificazione di informazioni obsolete avviene in modo molto semplice:

1. Ogni server contiene al suo interno le informazioni sul **term attuale** che ovviamente può incrementare nel tempo in modo monotono.
2. Nel momento in cui avviene una comunicazione ogni server **include** nel proprio **messaggio** anche le informazioni sul **proprio term**.
3. Nel caso in cui un server si accorga di avere un term troppo obsoleto **aggiorna** immediatamente il proprio **term** con quello più **recente**.
    - Se il server si trova nello stato di leader allora esso **regredisce** immediatamente **a follower**
4. Se un server riceve messaggi con **term obsoleti** questi vengono immediatamente **rifiutati**.

Tipologie di Messaggio | Remote Procedure Calls

Al fine di massimizzare la comprensibilità RAFT definisce solamente due tipologie di messaggio:

- **Request Vote:** i messaggi di questo tipo sono spediti dai **candidates** durante il processo di *leader election.*
- **Append Entries:** questi messaggi sono usati dai **leader** con tre scopi fondamentali:
    - *Log replication di una o più entries*
    - *Heartbeat con lo scopo di evitare che i follower possano iniziare una nuova elezione*
    - *Trasferimento di Snapshot tra i vari server*

Gestione del tempo

L'unica nozione di tempo logico è quella di term, non ci sono *clock condivisi* fra le macchine. Esiste una nozione di tempo implicita che garantisce il progresso: essa risiede nel concetto di **timeouts**.

Scalabilità

Per il fatto che un algoritmo di consenso necessita della maggioranza dei server per effettuare qualsiasi operazione, il concetto di **scalabilità viene meno**. L'unico modo di garantire scalabilità consiste nel **partizionare** i server in **più cluster** e usare un algoritmo di consenso fra cluster: questo è il caso dei sistemi di storaging di elevate dimensioni come Megastore, Spanner e Scatter.

3.3.2 Leader Election

In un dato term, tutte le *operazioni di replicazione* del log sono affidate ad un server che ricopre il ruolo di *leader*. Al fine di accordarsi in maniera safe su un unico leader è necessario applicare un protocollo di elezione robusto.

Candidatura

Ogni elezione ha inizio nel momento in cui uno o più server si candidano come aspiranti leader.

1. **Stato iniziale:** ogni server inizialmente si trova nello stato di follower e vi resta fintantochè riceve nuovi messaggi dall'attuale server leader.
2. **Heartbeat:** il leader, periodicamente, invia un messaggio detto di *heartbeat*[2] per comunicare ai suoi follower che è ancora in funzione.
3. **Timeout:** i server **follower** mantengono internamente un **timer randomico** che si resetta ogni volta che viene ricevuto un messaggio dal leader. La durata di questo timer viene chiamata *election timeout*.
4. **Inizio candidatura:** allo scadere del timer, il follower considera il leader assente e si propone come leader effettuando un **cambiamento di stato** da follower a candidate e dando inizio a una **nuova leader election**.

Quando una *Leader Election* ha inizio il follower esegue i seguenti passaggi:

1. *Incrementa di uno il proprio term*
2. *Cambia il proprio stato in candidate*
3. *Vota per sè stesso*

[2] come heartbeat vengono utilizzati i messaggi di tipo *AppendEntry* che, nel caso in cui non ci sia effettivamente una entry da appendere, vengono lasciati vuoti.

4. *Invia in broadcast un messaggio di tipo "Request Vote" affinchè gli altri membri del cluster possano votarlo*

A questo punto possono verificarsi i seguenti tre scenari:

1. **Il candidate vince l'elezione:** questo accade quando il numero di voti ricevuti dagli altri membri del cluster (compreso il proprio) raggiunge la maggioranza.
   A questo punto *cambia il suo stato in* ***leader*** e invia un **heartbeat**, con lo scopo di controllare il cluster evitando che si inneschino nuove candidature.

2. **Un altro candidate vince l'elezione:** questo può accadere se ad esempio i due si candidano a poca distanza l'uno dall'altro. A questo punto il candidate che ha perso, riceverà un messaggio dal nuovo leader e tornerà follower.

3. **Nessuno vince le elezioni:** la contemporanea presenza di più **Candidate** può portare a uno **Split-Vote**, ovvero una situazione in cui nessuno dei candidate ottiene la maggioranza dei voti. In questo caso il *term termina senza aver avuto un leader* e viene dato inizio al term successivo.

Elezione: Safety e Liveness

Il protocollo di elezione, al fine di essere robusto, deve soddisfare requisiti di safety e di liveness.

- **No leader multipli** (Election Safety): non vogliamo che **più di una macchina** possa essere eletta come leader nel medesimo term.
  - *Proof:* ogni server rilascia **un solo voto** per term e lo salva su disco. Se malauguratamente avviene un crash allora, nel caso in cui il server torni operativo, esso verificherà la possibilità di voto recuperando l'informazione dal disco. Quindi possiamo dire che due candidates non possono accumulare due diverse maggioranze sullo stesso term.

- **No Elezioni bloccate** (liveness): vogliamo evitare che nessun server diventi leader all'infinito.
  - *Proof:* ogni candidate mantiene internamente un **timer randomico** che viene inizializzato ad un valore che oscilla tra un massimo e un minimo, come accade per i follower. Questo timer, quando innescato, viene impostato ad un nuovo valore e fa si che il candidate incrementi il suo term e invii nuovamente delle *Request-Vote* a tutto il cluster, come se si fosse ricandidato. Facendo uso di questo espediente si minimizza la possibilità che si verifichino consecutivamente più *Split-Vote*. Questo garantisce che le elezioni possano, prima o poi, progredire portando all'elezione di un leader in un futuro term. Il meccanismo funziona se il valore di timeout è settato a un valore

maggiore del **RTT** di una comunicazione broadcast. Solitamente il valore scelto si aggira intorno ai 150/300 ms.

$$T > broadcastRTT$$

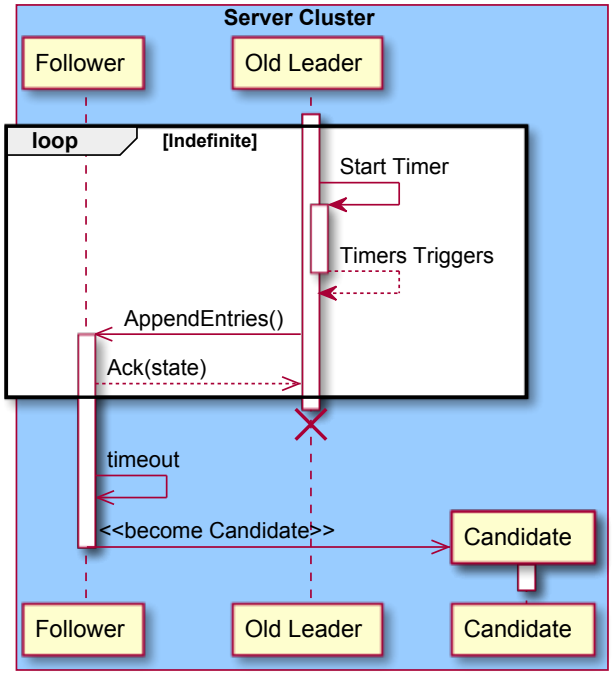

Figura 3.5: Uno scenario semplificato in cui viene mostrata una *Leader Election*

1. Il vecchio leader inizialmente svolge correttamente le sue mansioni, divulgando periodicamente i propri *heartbeats*
2. Successivamente il leader subisce un **fault** e di conseguenza non riesce più a svolgere le proprie attività
3. Prima o poi, non ricevendo più notizie dal leader, uno dei follower vedrà scadere il proprio timer.

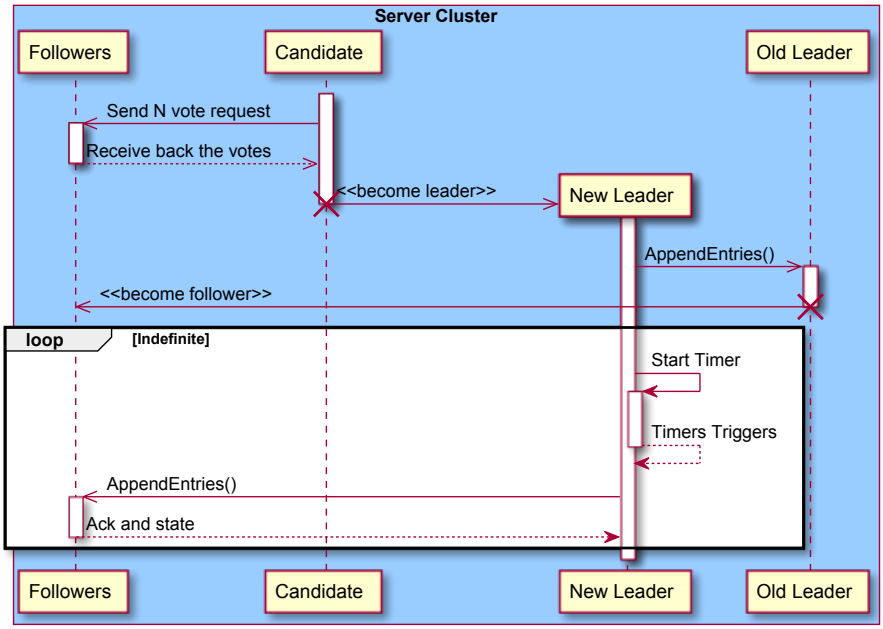

Figura 3.6: Uno scenario semplificato in cui viene mostrata una *Leader Election*

1. Diventato candidate, esso procederà a inviare le *Vote request* a tutti i follower del cluster.
2. Ricevuta la maggioranza dei voti verrà eletto leader e ripristinerà il normale funzionamento del cluster.
3. Il vecchio leader quando tornerà on-line riceverà un *AppendEntry* e tornerà allo stato di follower.

3.3.3 Normal Operation | Log Replication

Prima di proseguire con la descrizione della seconda fase dell'algoritmo (*log replication*) è bene fornire una definizione completa della struttura del log.

Struttura del Log

In RAFT ogni server, compreso il leader, mantiene una copia personale del log. Ogni singolo log è strutturato in questo modo:

- ***Entry:*** *ogni log è suddiviso in celle chiamate* ***entry*** *identificate da un* ***indice*** *che ne indica la* ***posizione*** *nel log*
- *ogni singola entry si compone di due campi:*
  - ***Comando:*** *un comando o una procedura sulla quale tutte le macchine presenti nel cluster devono trovare un accordo.*

- ***Term:*** *rappresenta il term relativo a quando la entry è stata aggiunta al log. Ciò sta a significare che la entry è stata aggiunta dal leader al suddetto term.*

- ***Memorizzazione:*** *ogni log è memorizzato in una memoria locale in modo da permettere resistenza ai crash del server.*

- ***Commited entry:*** *nel momento in cui una entry è replicata sulla maggioranza dei server allora si considera* ***committed*** *e può essere esguita dalle state machine.*

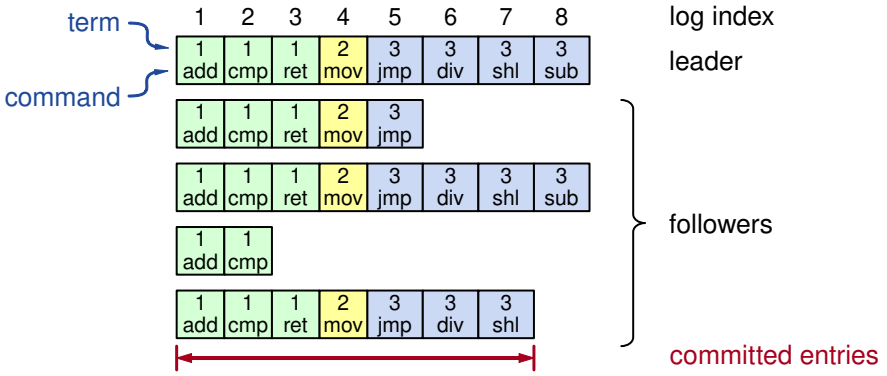

Figura 3.7: Nell'immagine possiamo vedere che la entry n.7 è committed poiché è replicata sulla maggioranza dei server. Entry n.8 non può essere committed poiché è replicata solo su 2/5 dei server.

Log Replication

Una volta eletto correttamente, il leader inizia a servire tutte le richieste che provengono dai client. La log replication in situazioni normali procede come segue:

1. *Il leader* ***appende*** *il comando al proprio log.*

2. *Il leader invia dei messaggi di tipo* ***AppendEntries*** *ai followers.*

3. *Quando la* ***maggioranza dei follower*** *comunica al leader di aver replicato il log allora la entry può essere* ***committed****. Nel caso di mancate risposte il leader attende riprovando a inviare la entry.*

4. *Il leader* ***passa il comando*** *alla propria state machine che lo* ***esegue****, il* ***risultato*** *viene poi ritornato al client.*

5. *Il leader inoltre comunica ai followers il committment e finalmente li abilita all' esecuzione del comando.*

Il leader mantiene traccia dell'ultima entry che è stata validata e include il suo indice e term in tutte le *AppendEntries* (inclusi gli *heartbeats*). Così facendo i followers potranno rimanere allineati riuscendo quindi ad eseguire in modo tempestivo il comando non appena riconoscono che esso è stato committato.

Log Matching Property

La proprietà di log matching garantisce che:

Se una entry è committed
allora anche tutte le entry precedenti lo saranno.

Questa proprietà deriva da queste sotto-proprietà:

- *Se due log hanno un entry con lo stesso index e stesso term, allora quell'entry contiene lo stesso comando(sono identiche).*
- *Se due entry in log diversi hanno lo stesso index e lo stesso term, allora anche tutte le loro precedenti entry sono identiche tra i due log.*

La prima proprietà segue dal fatto che un leader può creare **solamente una entry in un dato index per term** e **le entry non cambiano mai posizione**. La seconda proprietà è garantita da un semplice **consistency check** eseguito dai messaggi **AppendEntries**.

Consistency Check e Riparazione del Log

1. **AppendEntries con consistency check:** Nel momento in cui vengono inviate le **AppendEntries**, il leader include come argomento dei messaggi anche **l'indice e il term della entry immediatamente precedente la nuova entry da replicare**.

$$AppendEntries(term, index)$$

2. **Ack negativo:** Se il follower non trova una entry nel **fronte del log** con lo **stesso indice e term** allora rifiuta la nuova entry e manda un **ack(false)**.

$$Ack(false)$$

3. **AppendEntries induttiva:** IL leader allora procede in modo **induttivo** provando a inviare **AppendEntries** che fanno riferimento agli indici precedenti. Il leader invia l'*entry* precedente all'ultima inviata; se anch'essa non corrisponde, invia quella ancora prima e così via, fino a trovare una corrispondenza, come in figura 3.9

$$AppendEntries(term, index - 1)$$

4. **Ack positivo:** L'operazione di append ha successo solo nel caso in cui ci sia corrispondenza tra l'elemento precedente del log del leader e l'elemento nella stessa posizione del log del follower. In questo caso viene inviato un messaggio di ack positivo. **Tutte le entry successive all'indice di match vengono cancellate**

$$Ack(true)$$

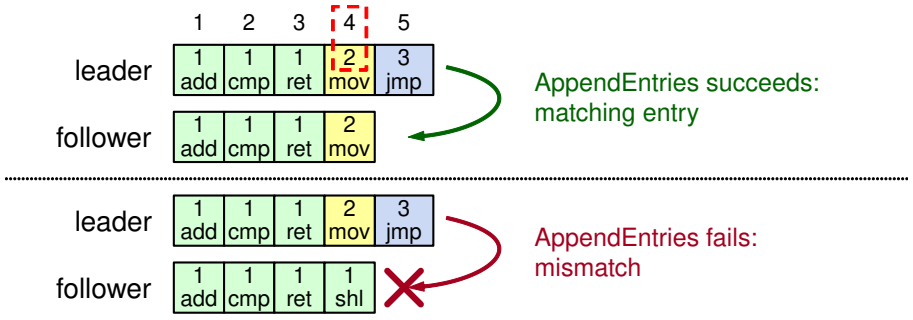

Figura 3.8: Il leader replica il proprio log inviando un elemento per volta al log dei vari follower, i quali accettano solo se c'è corrispondenza tra l'entry precedente a quella inviata dal leader e la loro ultima entry.

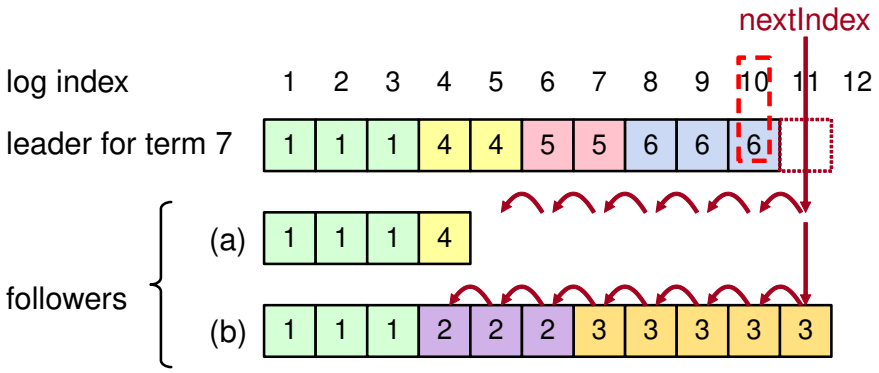

Figura 3.9: Il leader procede ad inviare entry con **indice sempre minore** fino a trovare una corrispondenza, per poi risalire fino alla cima del log, se necessario sovrascrive le entry inconsistenti del follower.
In figura, il leader deve inviare la entry in posizione 10, con term 6.

- **Caso a:** L'indice 10 è vuoto, di conseguenza il leader procede a ritroso finché non trova **corrispondenza** all'indice 4.
- **Caso b:** All'indice 10 c'è una entry diversa da quella del leader, di conseguenza esso procede a ritroso fino all'indice 3, dal quale inizierà a **ricostruire il log sovrascrivendo le entry inconsistenti**.

Il leader continua ad inviare *AppendEntries* ai followers che non rispondono, fino a quando non ottiene successo. Una eventuale presenza minoritaria di follower lenti o bloccati non intacca le performance del cluster e non rallenta la validazione di nuove entry da parte del leader.

Crash del Leader e inconsistenze del Log

In situazioni normali l'algoritmo garantisce che leader e followers si mantengano consistenti e dunque il **consistency check** delle **AppendEntries** non fallisce mai.
Può capitare però che il **leader crashi prima di riuscire a completare la replica sulla maggioranza dei follower**. In questo caso i log dei follower possono mostrare diversi tipi di inconsistenza (vedi figura 3.10):

- Entry mancanti
- Entry sovrabbondanti
- Entrambe le situazioni

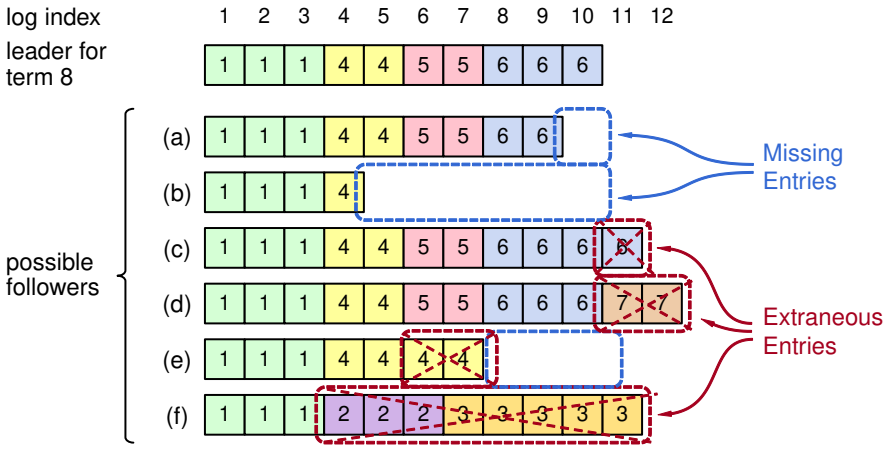

Figura 3.10: La figura mostra un esempio in cui vengono mostrati tutti i tipi di inconsistenza possibili in RAFT.

In RAFT le inconsistenze sono risolte senza introdurre dei comportamenti speciali da parte del leader. Questa semplicità risiede in una importante assunzione che viene fatta dall'algoritmo: **il leader ha pieno potere e possiede la verità assoluta.**
Questa assunzione permette al leader di portare tutti i server a convergere alla propria visione del log. Questo sta a significare che i follower potrebbero addirittura **cancellare o riscrivere entry non committate** presenti nei loro log se queste non soddisfano il leader. La riparazione del log viene fatta in modo automatico tramite **AppendEntries** con **consistency check induttivo**.

3.3.4 Safety e Consistency

Nelle sezioni precedenti è stato descritto il funzionamento generale di RAFT includendo alcune proprietà e vincoli essenziali per mantenere consistenza tra i log durante le principali operazioni.
Qui di seguito ricordiamo le proprietà nominate nelle sezioni precedenti:

- **Election Safety:** Per un dato term è possibile al più eleggere **un solo leader**. La dimostrazione di questa proprietà è già stata discussa precedentemente nella sezione 3.3.2.
- **Leader Append-Only:** Il leader non cancella **mai**, né sovrascrive le entry del proprio log. La proprietà è descritta in dettaglio nella sezione 3.3.3.
- **Log Matching:** La proprietà di log matching garantisce che:
  - *Se due log hanno un entry con lo stesso index e stesso term, allora quell'entry contiene lo stesso comando(sono identiche).*

– *Se due entry in log diversi hanno lo stesso index e lo stesso term, allora anche tutte le loro precedenti entry sono identiche tra i due log.*

La proprietà è descritta in dettaglio nella sezione 3.3.3.

L'elenco sovrastante non è però completo: attualmente ci sono casi che non sono gestiti e che possono causare notevole **inconsistenza** tra i log.
Ad esempio se il leader *valida* alcune *entry* mentre un dato follower è offline, nel caso in cui il follower venisse eletto esso potrebbe sovrascrivere le entry già eseguite con quelle presenti nel proprio log.
E' necessario dunque completare l'algoritmo aggiungendo la proprietà chiave di tutte le state machine: la **State Machine Safety** property.

State Machine Safety *Se una entry è stata committata ed eseguita in una state machine, allora nessun altra state machine può committare una valore differente per la stessa entry.*
la state machine safety porta all'introduzione di due **restrizioni** molto importanti all'algoritmo:

- **Restrizione durante la leader election**
- **Restrizione nei commitment**

Restrizioni nelle Elezioni

Durante le elezioni ci sono casi in cui non è possibile determinare se un *entry* è committed oppure no. Come si può vedere in figura 3.11 non è possibile determinare se l'ultima *entry* è stata committed: per questo si aggiunge un **ulteriore vincolo all'elezione di un candidate**.

Scegli il candidato con il log più completo.

Più precisamente il candidate invia solo le informazioni sull'**ultima entry** dato che queste informazioni definiscono interamente il log.

$$AppendEntries(term, index)$$

Il criterio con cui viene fatta questa valutazione è il seguente:
Sia $lastTerm_v$ il term presente a indice $lastIndex_V$ della entry del log del **server votante** e $lastTerm_c$ il term presente a indice $lastIndex_C$ della entry del log del **candidate**, Allora:

Se :

$$\begin{gathered}(lastTerm_v > lastTerm_c) \parallel \\ (lastTerm_v == lastTerm_c) \;\&\&\; (lastIndex_v > lastIndex_c) \implies Reject\end{gathered} \tag{3.1}$$

In altre parole:

- *Se l'ultimo term del log del server votante è maggiore di quello del server candidato, allora la richiesta viene rifiutata.*
- *Se invece i due ultimi term si equivalgono, la valutazione viene fatta tenendo conto della lunghezza del log:*
  - *Se l'ultimo indice del log di C è maggiore di quello di V la richiesta viene accettata, altrimenti viene rifiutata.*

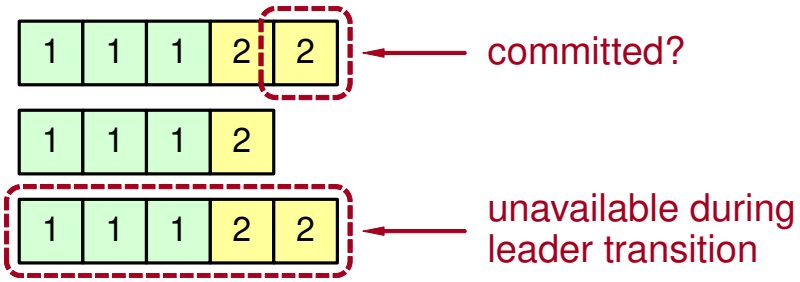

Figura 3.11: In questo caso non è possibile determinare se l'ultima *entry* è validata, inoltre solo il primo server può essere eletto dato che il terzo non è disponibile.

Commiting di Entries del term corrente Vediamo ora un esempio di quello che è stato detto fino ad ora. Nella figura 3.12 possiamo vedere che all'indice 4 il leader $S1$ al term 2 è riuscito a **replicare con successo** la propria entry sulla maggioranza dei server; la entry è dunque **committed**. A questo punto nel caso in cui il leader $S1$ crashasse, per l'**election restriction** introdotta sopra, i server $S4$ e $S5$ non potrebbero essere eletti leader.

- $S5$ non può essere eletto perché non possiede un term più piccolo di tutti gli altri → **condizione n:1**
- $S4$ non può essere eletto poiché anche se possiede un term aggiornato non ha un log completo (più corto) → **condizione n:2**

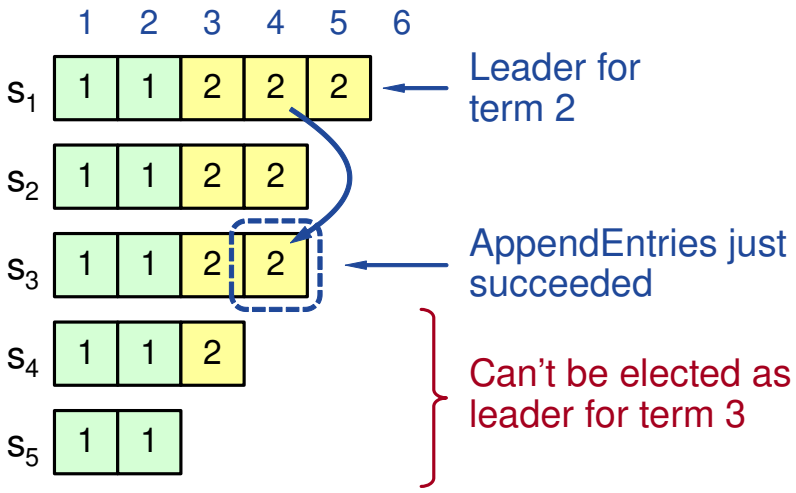

Figura 3.12: $S4$ e $S5$ si trovano impossibilitati ad essere eletti come leader e quindi non sono in grado di modificare il log di latri server.

Committing di Entries di term precedenti

Questo caso, descritto perfettamente nell'immagine 3.13 rappresenta la situazione peggiore e più inusuale per RAFT; essa si può riassumere come segue:

1. Il server $S1$, leader al **term 2**, **replica la entry** solamente su due server e poi crasha.
2. Un altro server, $S5$ viene eletto leader per il **term 3** da $S3$, $S4$ e $S5$, esso appende una serie di entry e poi crasha prima di poter comunicare.
3. Il server $S1$ si risveglia e viene eletto leader da $S1$, $S2$, $S3$, $S4$ e prima di tutto cerca di **riparare** il log del sever $S3$ inviando la entry mancante.
4. Il server $S5$ quando riceve gli **ack** da parte dei followers, però, **NON COMMITTA**
5. Infatti, se $S1$ **crashasse dopo aver committato la entry 3** allora $S5$ potrebbe essere eletto poichè soddisfa entrambi i casi di 3.1
   - **Caso A**

     $$lastTerm_{S5} > lastTerm_{Si} \qquad \forall i \in [2,3,4]$$

   - **Caso B**

     $$lastIndex_{S5} > lastIndex_{Si} \qquad \forall i \in [2,3,4]$$

6. Se $S5$ venisse eletto al **term 5** cercherebbe di propagare il proprio log e questo **obbligherebbe** $S1$**,** $S2$**,** $S3$**, e** $S4$ **a dover riscrivere i propri log**.
NB: Qua abbiamo una totale mancanza di **safety**, infatti $S1$ *non saprebbe cosa fare dato che le entry, una volta commitate, non possono essere cancellate dal log*.

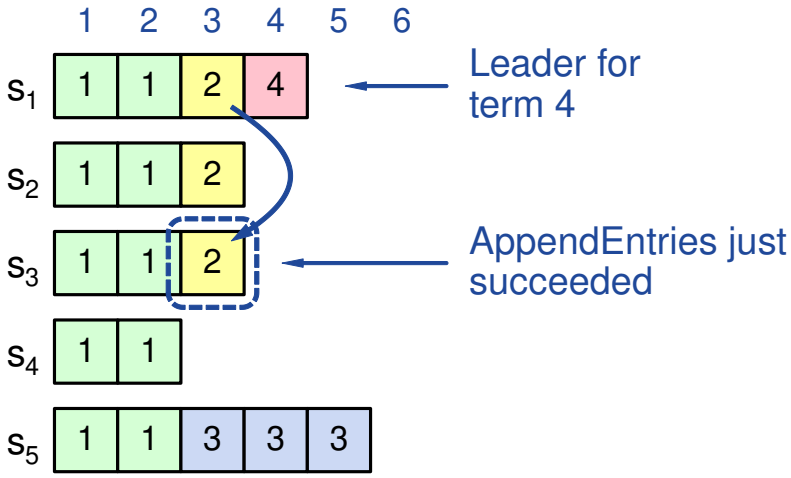

Figura 3.13

Soluzione: Le regole di committing vengono estese al fine di garantire safety. Ogni leader decide se una entry è da committare in base a queste regole:

- ***La entry deve essere presente nella maggioranza dei server***
- ***Almeno una nuova entry proveniente dal leader dell'attuale term deve essere presente nella maggioranza dei server***

Come vediamo in figura 3.14, nel momento in cui $S1$ viene eletto leader per il **term 4** e riesce a replicare la entry sulla maggioranza dei server, esso può finalmente **propagare le informazioni di committment** riferite a term passati. Questo garantisce che il server $S5$ non possa essere eletto leader al **term 5** per la regola 1 in 3.1.

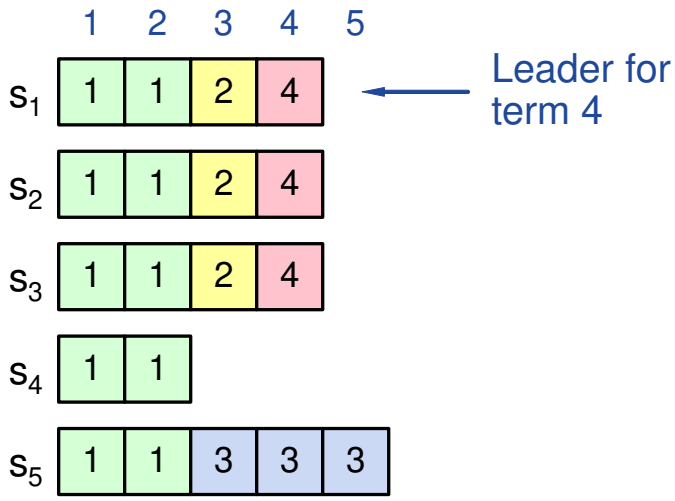

Figura 3.14: Quando $S4$ diviene leader non committa immediatamente le entry passate ma attende l'arrivo di nuove entry. Questo evita che $S5$ possa diventare leader sovrascrivendo i logs e portando inconsistenze.

Neutralizzazione di vecchi leader

Supponiamo che un leader venga momentaneamente disconnesso dalla rete, creando una cosiddetta **network partition**. RAFT, essendo un algoritmo che soddisfa il requisito di **partition tolerance**, garantisce che il cluster sopravviva alla failure di un membro: il cluster semplicemente eleggerà un **nuovo leader**. *Cosa succede però se il vecchio leader torna in campo e decide di continuare a governare?*
Per evitare inconsistenze RAFT mette in pratica un comportamento molto semplice, chiamato **neutralization of stale leader**, esso funziona in modo molto semplice:

- Ogni messaggio scambiato contiene al suo interno il **term del mittente**.
- **Caso A**

  $$senderTerm > receiverTerm$$

  - **Il messaggio viene rifiutato.**
  - **Il mittente si converte a follower.**
- **Caso B**

  $$receiverTerm > senderTerm$$

  - **Il ricevente si converte a follower.**
  - **Il ricevente aggiorna il suo term all'ultimo ricevuto.**
  - **Il ricevente processa il messaggio normalmente.**

Seguendo questi semplici casi, una volta che un' **elezione** per un nuovo leader è **terminata** non è possibile che un leader deposto possa inviare messaggi alla maggioranza del cluster poiché tale maggioranza possiederà un **term aggiornato dall'ultima elezione**.

3.4 RAFT Extensions

3.4.1 Client Protocol

Al fine di comprendere il funzionamento dell'algoritmo è necessario capire quali sono le modalità con cui i server interagiscono con i client. RAFT è un algoritmo leader-based, di conseguenza è il leader che svolge i compiti principali, ed è anche l'entità che si occupa della comunicazione con i client.

Caso base

L'interazione di base tra il client e il leader del server, che va dalla sottomissione della richiesta alla ricezione del risultato di quest'utlima attraversa le seguenti fasi:

1. **Sottomissione:** il **client** sottopone la richiesta al leader.
2. **Presa in carico:** il **leader** prende in carico la richiesta, accodandola al proprio log e replicandola sugli altri server.
3. **Commit:** il **leader**, prima di poter procedere col soddisfare la richiesta, deve **assicurarsi** che venga fatto il **commit** della stessa, per avere la garanzia che sia stata replicata nella maggioranza dei log.
4. **Esecuzione:** assicuratosi che il commit sia avvenuto, e che quindi sia *safe* computare il risultato, il **leader** fa eseguire l'istruzione alla propria **state machine**
5. **Risultato:** ottenuto il risultato, il leader lo comunica al client.

Ricerca del leader

Per dare inizio alla scambio, il client deve **contattare il leader** del cluster. Nel caso in cui il client **non sappia** quale tra i server del cluster sia il leader è sufficiente che contatti **un membro qualsiasi** del cluster. Se un follower riceve una richiesta da un client, risponde con le **informazioni sul leader**, neccessarie affinchè il client possa **contattarlo direttamente**.

Questo meccanismo entra in funzione anche nel momento in cui, a seguito di un crash del leader, il client ha bisogno di scoprire l'identità di quello nuovo al fine di risottomettere una richiesta.

Crash del leader

Abbiamo detto che nel caso in cui il client non riceva più notizie dal leader, ad esempio a causa di un crash di quest'ultimo, l'interazione si svolge nel seguente modo:

1. *Il **client** ritrasmette il comando ad **una qualsiasi** delle altre macchine.*
2. *Se la macchina contattata **non è il nuovo leader**, fornisce indicazioni su **a chi rivolgersi**.*
3. *Il **client** si rivolge al **nuovo leader** e, ad esso, **risottomette** la richiesta*

Questo iter **garantisce** che ogni richiesta verrà gestita, **prima o poi**, ma ancora **non è sufficiente** a garantire che venga eseguita **una e una sola volta**. Può accadere che il crash del leader avvenga **dopo** l'esecuzione del comando, ma **prima** che sia comunicata la risposta al Client. In questo caso il client non saprebbe che il comando è stato eseguito e ritrasmetterebbe la richiesta al nuovo leader il quale, nel momento in cui farà commit del comando, lo rieseguirà. In questo modo si avrebbe che il comando è stato eseguito **due volte**, il che non è accettabile.

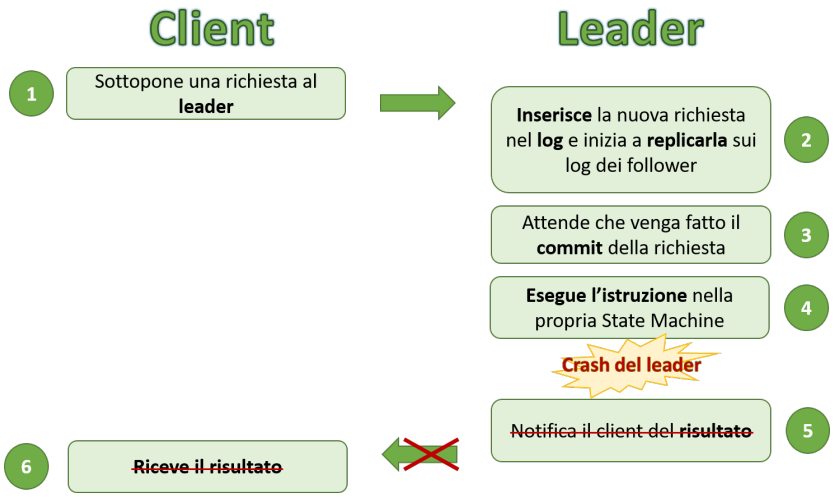

Figura 3.15: Lo schema mostra l'interazione di base tra il client e il leader. Nel caso in cui si verifichi un **crash del leader** nel punto evidenziato in figura, si corre il rischio che il client risottometta la richiesta al nuovo leader e che anche quest'ultimo la esegua, **violando** il vincolo secondo cui una richiesta debba essere eseguita esattamente **una e una sola** volta.

Introduzione di una exactly-once semantics

Per garantire che uno stesso comando richiesto da un client non possa essere eseguito più di una volta, si introduce un **identificatore univoco** per ogni richiesta.

- Il client genera di volta in volta l'identificativo e lo inserisce nella richiesta.
- L'identificativo è contenuto nell'entry el log relativa alla richiesta
- Se il leader riceve una richiesta con un identificativo **non presente** in una delle entry del proprio log si comporta come di consueto:
  1. *la accoda;*
  2. *inizia a replicarla;*
  3. *raggiunta la maggioranza dei consensi fa commit della stessa;*
  4. *la esegue*
  5. *comunica il risultato al client.*
- Se il leader riceve una richiesta con un identificativo già **presente** in una delle entry del proprio log, si comporta in questo modo:
  1. ***non** la accoda;*
  2. *attende che ne venga fatto il commit (se non è già successo);*
  3. *la esegue (se non l'ha già fatto;)*
  4. *comunica il risultato al client.*

In questo modo si garantisce che un comando venga eseguito una e una sola volta: **exactly-once semantics**.

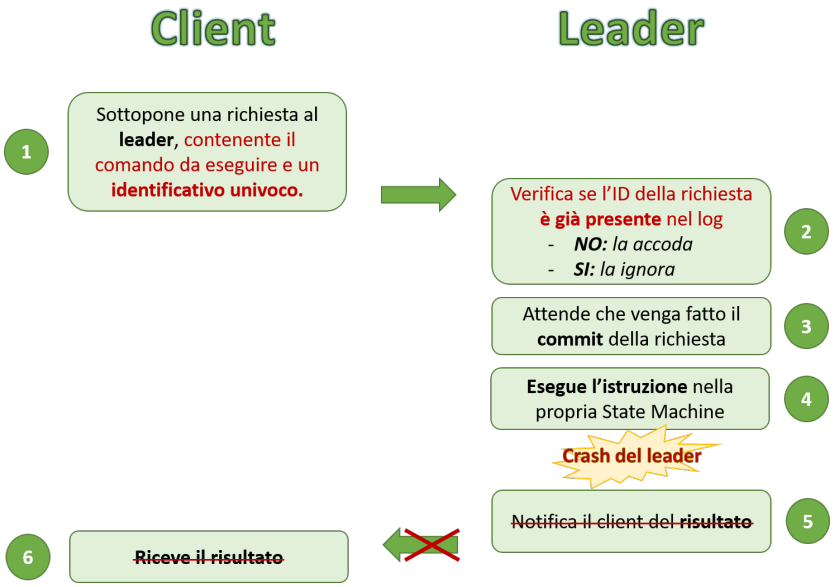

Figura 3.16: **Interazione** tra client e leader che tiene conto dell'aggiunta di un **identificativo** per le richieste, al fine di **evitare** la **molteplice esecuzione** di una richiesta. Diversamente dal caso precedente, un crash del leader nel punto evidenziato **non porta alla violazione di alcuna proprietà**.

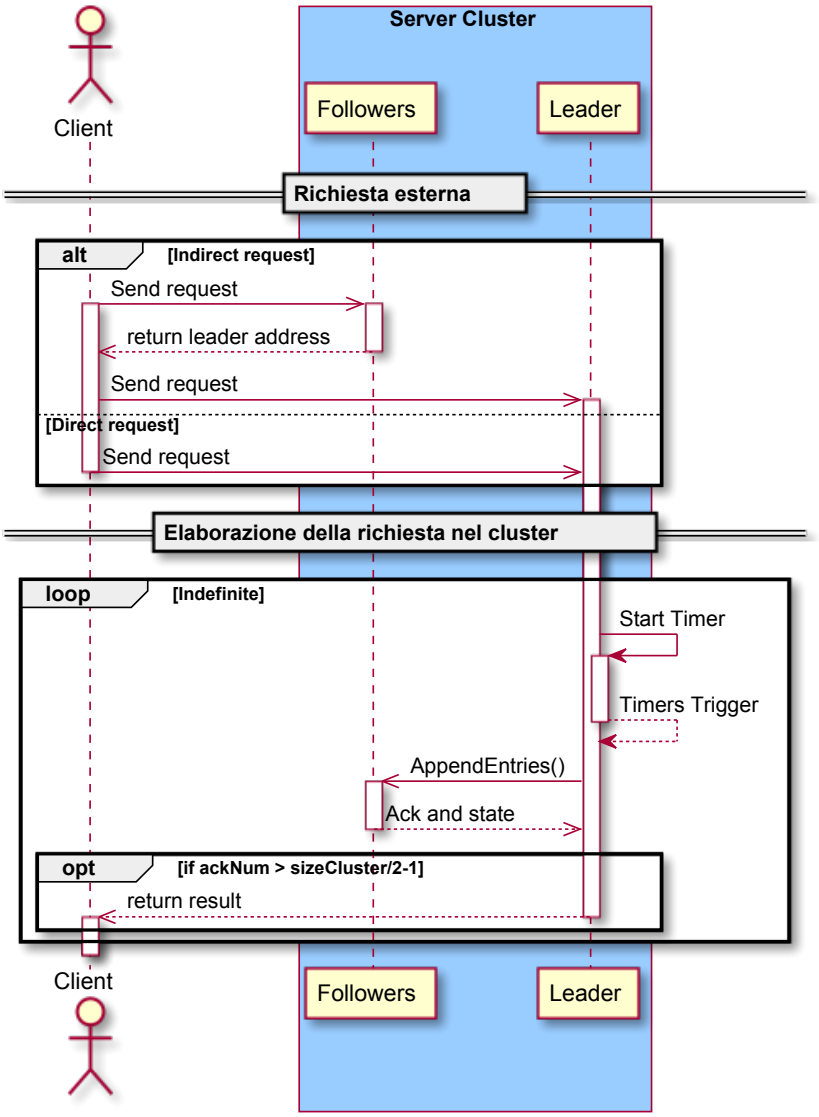

Figura 3.17: Rappresentazione completa dell'interazione tra il client e i server appartenenti al cluster.

3.4.2 Configuration Changes

La configurazione del sistema può cambiare nel tempo, ad esempio quando:

- *una macchina viene eliminata*
- *una macchina viene sostituita*

- *una nuova macchina viene aggiunta al cluster*

Le informazioni sulla configurazione del sistema sono fondamentali perchè sapere **quanti** e **quali** server fanno parte del cluster è necessario per operazioni che necessitano di decidere **basandosi sulla maggioranza**, come l'elezione del leader o il commit di una entry.

Contemporanea presenza di due configurazioni

Il problema che si incontra in tutti i sistemi distribuiti è che *la* ***transizione*** *tra una configurazione e l'altra* ***non avviene nello stesso istante*** *per tutte le macchine*. Esisterà dunque un tempo t per cui **alcune** macchine saranno aggiornate sulla **nuova configurazione** mentre **altre** saranno ancora nella **vecchia**. Ciò porta a una situazione in cui è possibile che ci siano due gruppi di server, che si trovano in **due diverse configurazioni** e che rappresentano la **maggioranza** per quella configurazione.

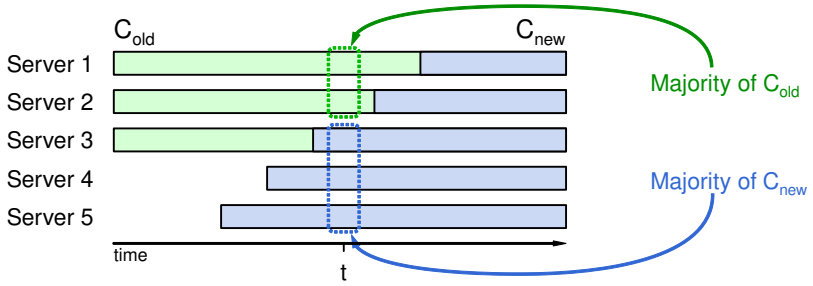

Figura 3.18: Esempio di **maggioranze diverse in due diverse configurazioni**: i server $S4$ e $S5$ entrano a far parte del cluster, ma al tempo t, si ha che $S1$ e $S2$ sono ancora alla **vecchia configurazione**, mentre $S3$, $S4$ e $S5$ sono **passati alla nuova**. $S1$ e $S2$ costituiscono una maggioranza, nella configurazione vecchia (sono **2 su 3**), ma anche $S3$, $S4$ e $S5$ possono formare una maggioranza nella loro configurazione di riferimento (**3 su 5**).

Problema

Sulla base di queste **maggioranze** possono essere portate a termine delle operazioni, come ad esempio, il commit di una entry. Questo potrebbe generare delle **inconsistenze**, come nel caso seguente:

1. *I server $S1$ e $S2$ formano una* ***maggioranza*** *nella* ***vecchia*** *configurazione*
2. *I server $S3$, $S4$ e $S5$ formano una* ***maggioranza*** *nella* ***nuova*** *configurazione*
3. *la* ***maggioranza*** *data da $S1$ e $S2$ decide di fare* ***commit*** *dell'entry $e1$*

4. *la **maggioranza** data da $S3$, $S4$ e $S5$ decide di fare **commit** dell'entry $e2 \neq e1$*

5. *in corrispondenza dello **stesso indice**, si avranno dei **log diversi** che porteranno all'esecuzione di istruzioni diverse, generando **inconsistenza***

Per risolvere questo problema, il passaggio tra una configurazione e l'altra non è immediato, ma viene fatto in **due fasi**.

Soluzione

Per cambiare la configurazione del sistema, si ricorre a una **fase intermedia** posta tra la vecchia configurazione e quella nuova, in maniera tale da **evitare** di incorrere in **inconsistenze**. Durante questa fase intermedia, chiamata **joint consensus**, per prendere le decisioni si tengono in considerazione entrambe le maggioranze: sia quella della nuova configurazione (C_{new}) che quella della vecchia (C_{old}) in maniera tale che nè C_{old} nè C_{new} possano prendere **decisioni unilaterali**.

Approccio bifase: joint consensus

Le caratteristiche di questo approccio sono le seguenti:

- **Cambiamento della configurazione:** il passaggio a una nuova configurazione viene innescato da una richiesta. Quando il leader riceve questa richiesta la aggiunge al proprio log e la propaga, come farebbe normalmente, ma l'azione ha **effetto immediato** poichè il leader applica il cambiamento di configurazione il prima possibile, **senza attendere il commit**.

- **Configurazione intermedia:** successivamente, il leader, passa alla configurazione intermedia ($C_{old+new}$) e prende tutte le successive decisioni basandosi su di essa.
  Ad esempio, per capire se è possibile procedere con il commit di una entry, verifica che essa abbia la ***maggioranza* sia nella vecchia** configurazione **che in quella nuova**.

- **Decisioni sulla base di C_{old} o $C_{old+new}$:** esiste un periodo di tempo in cui la **entry** della nuova configurazione è **presente** nel log del leader ma **non** è ancora stata **committata**. In questo lasso di tempo è possibile che le **decisioni** possano essere prese sotto **entrambe le configurazioni**.
  Ad esempio, se il **leader** decade prima di aver **replicato** l'entry della **nuova configurazione** negli altri log, è possibile che venga eletto come **nuovo leader** un server che ha ancora la **vecchia configurazione**.
  Tuttavia, prima o poi arriverà un leader che non fallirà anzitempo e la nuova configurazione verrà committata.

- **Joint consensus:** una volta fatto il **commit** diventa **impossibile** che vengano prese decisioni solo sulla base di C_{old}, perchè ora il sistema è

interamente sotto la configurazione $C_{old+new}$, in uno stato di *joint consensus*. Durante il joint consensus, la **nuova configurazione** può essere **aggiunta** al log e **propagata**.

- **Decisioni sulla base di $C_{old+new}$ o C_{new}:** esiste un periodo di tempo tra l'**aggiunta** nel log del leader della **nuova configurazione** e il **commit** della stessa, in cui le **decisioni** possono essere prese sulla base della configurazione $C_{old+new}$ o sulla base della **nuova configurazione**.
  Ciò accade perchè nel caso in cui il **leader** sia soggetto a **crash**, **prima** che abbia proceduto al **commit** dell'entry relativa alla **nuova** configurazione, può essere eletto un **nuovo leader** che ha ancora la **configurazione intermedia**.
  Tuttavia, come nel caso precedente, si ha che **prima o poi** un leader riuscirà a fare **commit della nuova configurazione** e, da quel momento in poi, le decisioni verranno prese **solo** sulla base di quest'ultima.

- **Elezione di leader in $C_{new+old}$ durante C_{new}:** non c'è **nessun momento** in cui sia C_{old} che C_{new} possono prendere **decisioni unilaterali**, generando **conflitti**. Tuttavia è possibile che anche **dopo** che C_{new} è stata **committata**, venga eletto un **leader** che **non è ancora** in tale configurazione e che quindi **non può prendere decisioni**. In questo caso esso dovrà **"dimettersi"** dando vita ad una **nuova elezione** nel momento in cui il timeout di uno degli altri follower scadrà.

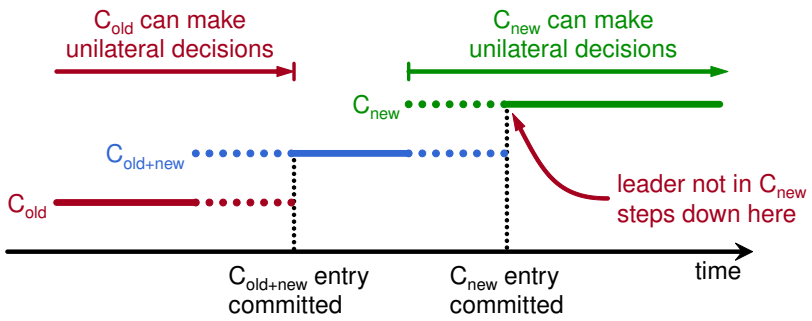

Figura 3.19: Linea temporale che mostra le fasi del passaggio del sistema da una configurazione all'altra.

Capitolo 4

RAFT: Implementazione

Dopo aver appreso in maniera approfondita le dinamiche dell'algoritmo RAFT, si è passati alle fasi di progettazione, sviluppo e testing di un applicativo che ne implementi il funzionamento.

4.1 Obiettivi e Requisiti

L'implementazione che si intende realizzare mira allo sviluppo di un sistema che simuli un ambiente distribuito in cui n server utilizzano l'algoritmo RAFT per raggiungere il consenso. In particolare si è deciso di utilizzare, come scenario, quello di un client che contatta i server per effettuare delle **operazioni sul database dei conti correnti bancari**.

- **Richieste del client:** il client deve poter sottomettere ai server delle operazioni da eseguire sul database dei conti correnti. Le operazioni consentite saranno:
  - *Deposito*
  - *Prelievo*
  - *Saldo*
- **Gestione lato server:** i server devono gestire le richieste provenienti dal client e rispondere con un risultato contenente l'**esito** dell'operazione e il **saldo** dopo che questa è stata effettuata.
- **Replicazione database:** il database dei conti correnti deve essere replicato su tutti i server in maniera tale che questi ultimi eseguano **le stesse identiche istruzioni nello stesso identico ordine** ottenendo il **medesimo risultato**.

Il sistema dovrà inoltre rispettare requisiti che riguardano la **visualizzazione** delle informazioni rilevanti e l'**interazione** con l'utente.

4.1.1 Requisiti di visualizzazione

All'utente dovrà essere dato un feedback dell'avanzamento delle operazioni, per questi motivi dovranno essere visualizzate le seguenti informazioni:

- **Ruolo:** per ciascun server bisognerà mostrare il ruolo che esso ricopre in un dato momento. I ruoli possibili sono:
  - *Leader (0-1)*
  - *Candidate (0-n)*
  - *Follower(0-n)*
- **Ultimo commit:** l'indice dell'ultima **entry** che è stata **committata** da ciascun server.
- **Term corrente:** il term in cui si trova correntemente ognuno dei server.
- **Log:** dovrà essere visibile lo stato dei log di ciascun server affinchè si possano seguire le fasi che portano al raggiungimento del consenso e valutare se l'implementazione ricalca correttamente l'algoritmo. Ogni log è costituito da **entry**; per ogni entry si vogliono visualizzare le seguenti informazioni:
  - **Term:** si tratta del **term** in cui si trovava il **leader** che ha creato l'entry, nel momento in cui questa è stata generata.
  - **Comando:** l'**operazione da eseguire**, contenuta nella richiesta del client
  - **Indice:** la **posizione** di quella determinata **entry** all'interno del log del relativo server.
  - **Stato:** se l'**entry** è **committed** o meno. Se una entry è committed, non dovrà in alcun modo essere possibile che ne compaia una non committata con indice precedente ad essa.
- **Stato di esecuzione:** dovrà esserci la possibilità di visualizzare le richieste sottomesse dal client e, per ognuna di esse, capire se essa è **già stata eseguita.**
- **Risultati:** nel caso in cui una richiesta del client sia stata eseguita, sarà necessario mostrare all'utente il risultato relativo, ricevuto dal client. Esso **non dovrà differire** dai risultati ottenuti su ciascuno dei server.

4.1.2 Requisiti di interazione

L'applicativo dovrà avere un'interfaccia utente che permetta di interagire con il sistema agendo su determinati parametri. I modi in cui il sistema potrà essere manovrato dall'utente sono i seguenti:

- **Operazione:** dovrà essere reso disponibile un modo per **effettuare una richiesta** (da parte del client) a uno dei server.

- **Stop/Resume:** il comando *stop* servirà per simulare la temporanea assenza di un server, mentre il comando *resume* ne farà ripartire l'attività.
- **Perdita dei messaggi:** l'utente potrà impostare una percentuale di perdita di messaggi per ogni server, in maniera indipendente. Questa funzionalità permetterà di testare la resistenza a questo tipo di fault.
- **Timer:** sarà possibile provocare lo scadere del timer. Tale azione assumerà significati diversi a seconda del ruolo che il server ricoprirà in quel momento:
  - **Leader:** lo scadere del timer porterà all'invio degli heartbeat.
  - **Candidate:** la scadenza del timer lo porterà ad aumentare il proprio term e ricandidarsi.
  - **Follower:** allo scattare del timer, il follower, non avendo ricevuto notizie dal leader, deciderà di aumentare il proprio term e cambiare il proprio stato in *candidate*.

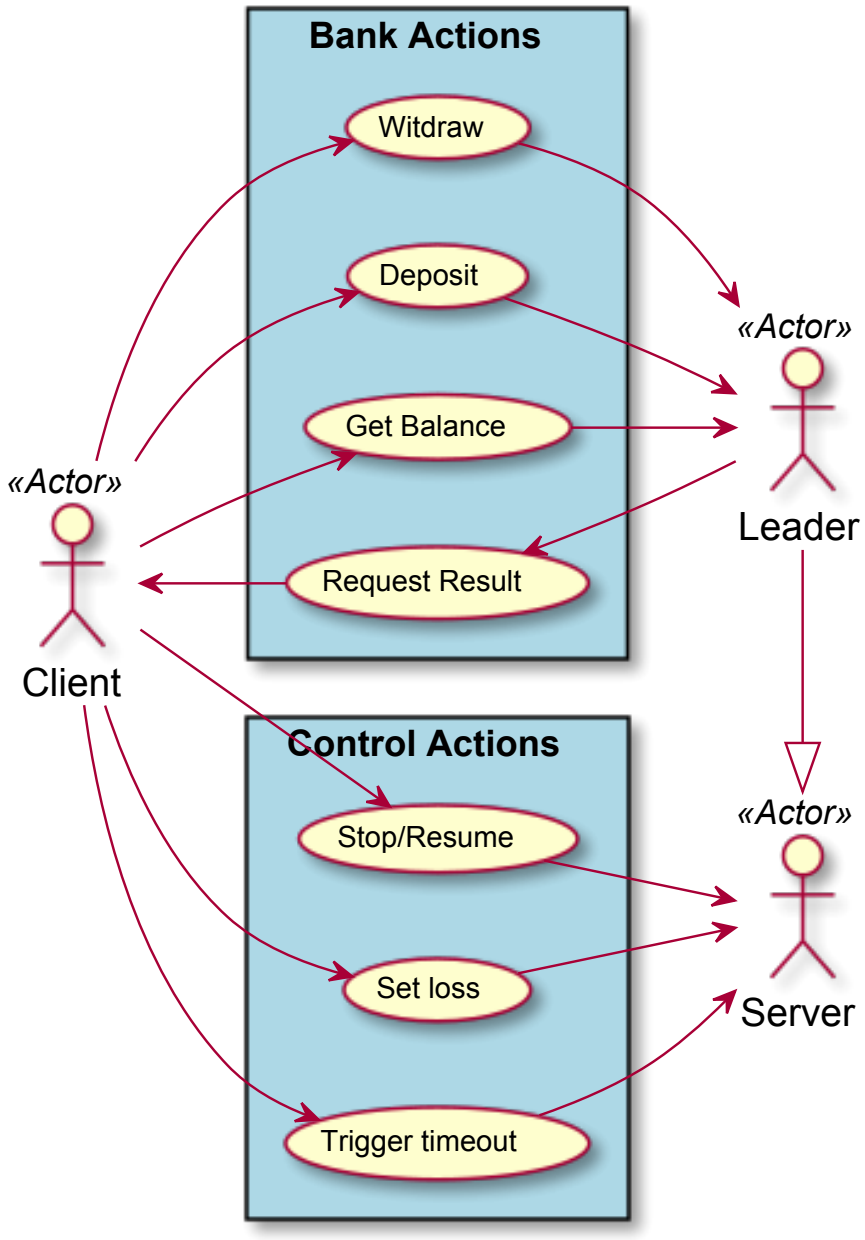

Figura 4.1: Diagramma dei casi d'uso che descrive l'interazione tra gli attori.

4.2 Analisi dei Requisiti

L'applicativo è una rappresentazione operativa di RAFT; per questo motivo, i dettagli dei requisiti che deve avere sono insiti nelle caratteristiche dell'algoritmo discusse in precedenza.

Il sistema dovrà consistere in un insieme di nodi che rappresentano due entità principali:

- **Un client:** si occupa di inoltrare le richieste ai server e ricevere gli esiti.
- n **server:** accolgono le richieste del client, le eseguono e comunicano il risultato. Ognuna di queste entità, in un dato istante, ricopre uno e un solo ruolo tra *Leader, Candidate* e *Follower*

Dal momento che queste entità devono essere **indipendenti** ed interagire tra loro, si è scelto di utilizzare il **paradigma ad attori**, che rende possibile agire ad un livello di astrazione molto elevato permettendo di facilitare la gestione delle **dinamiche di interazione**.

4.2.1 Il paradigma ad attori

Si tratta di un paradigma nato dagli stessi principi che hanno ispirato la programmazione ad **oggetti**. Nello specifico una classe è caratterizzata da due concetti fondamentali:

- **Stato:** rappresenta l'insieme dei valori delle variabili di una classe in un dato istante.
- **Comportamento:** espresso sotto forma di istruzioni contenute nei metodi.

Il paradigma ad attori riprende questi aspetti, arricchendoli con una caratteristica fondamentale: il **flusso di controllo**. Per **attore** si intende quindi un' entità **indipendente**, **univocamente identificata**, caratterizzata dai seguenti elementi:

- *Una **mailbox**, alla quale arrivano messaggi provenienti da altri attori.*
- *Uno **stato**, non accessibile dall'esterno.*
- *Un **comportamento** (behaviour), che definisce il modo in cui l'attore reagisce ai messaggi ricevuti.*
- *Un **flusso di controllo** indipendente.*

Per rendere possibile la realizzazione di un sistema ad attori, sono necessarie tre primitive:

- **Send:** per inviare un messaggio ad un altro attore, compreso il mittente.
- **Create:** per creare un attore figlio.

- **Become:** per cambiare il comportamento dell'attore. Ogni comportamento è caratterizzato da una diversa reazione ai messaggi ricevuti.

Il paradigma ad attori garantisce inoltre le seguenti proprietà fondamentali:

- **Reactiveness:** l'attore è reattivo poichè reagisce in maniera **individuale** ai messaggi ricevuti, dal momento che ha un proprio, unico, **event loop**. Bisogna però seguire una **golden rule:** *Never block the event loop.*
- **State encapsulation:** lo stato dell'attore, descritto dai parametri che esso assume in un dato momento, è incapsulato all'interno dello stesso, **impedendo accessi dall'esterno**.
- **Macrostep semantics:** si inizia ad eseguire una nuova operazione solo dopo aver concluso la precedente. Questo rende la gestione di ciascun messaggio **atomica**.
- **Fairness:** i messaggi accodati, **prima o poi** verranno gestiti.
- **Location transparency:** gli attori possono comunicare tra di loro **senza** essere a conoscenza delle rispettive posizioni.

Esistono diverse implementazioni del paradigma ad attori, che differiscono per il tipo e la gestione della **receive**. I sistemi più moderni preferiscono un approccio *a receive implicita*, dove la ricezione dei messaggi è gestita in maniera trasparente. Nelle sezioni successive si vedrà che, per quanto riguarda la realizzazione di questo progetto, la scelta è ricaduta proprio su questo tipo di implementazione.

4.3 Design

Come precedentemente specificato, si è deciso di procedere allo sviiluppo del sistema utilizzando il paradigma ad attori. Durante la fase di progettazione sono state designate come attori le seguenti entità:

- **Client Actor:** è l'attore che incapsula il comportamento del client.
- **Server Actor:** è l'attore che rappresenta il server e, in quanto tale, presenta tre diversi behaviour:
  - *Leader*
  - *Follower*
  - *Candidate*
- **State Machine:** è un attore figlio del Server Actor e incapsula la gestione delle richieste da eseguire e la restituzione del risultato.

4.3.1 Struttura

Il programma è composto da due sottosistemi indipendenti.

- Il sottosistema **Client** è strutturato in questo modo:
  - **Attore Client:** che si occupa dell'interazione con i server.
  - **Modulo View:** per la gestione dell'interfaccia grafica.

  L'interfaccia grafica, essendo un modulo univoco, è stata wrappata, per necessità, all'interno del Client. Questo, in combinazione con il *pattern observer* ha permesso di **disaccoppiare** le due componenti.

- Il sottosistema **Server** è invece organizzato come segue:
  - **Command log:** mantiene in memoria il log e fornisce operazioni per agire su di esso.
  - **BankStateMachine:** attore che modella la macchina a stati, la quale si occupa di gestire l'esecuzione dei comandi ad essa inoltrati e la notifica dei risultati.
  - **Modulo di consenso:** è la parte principale in cui si sviluppa l'algoritmo e gestisce il comportamento del server a seconda del ruolo che ricopre in un dato momento.

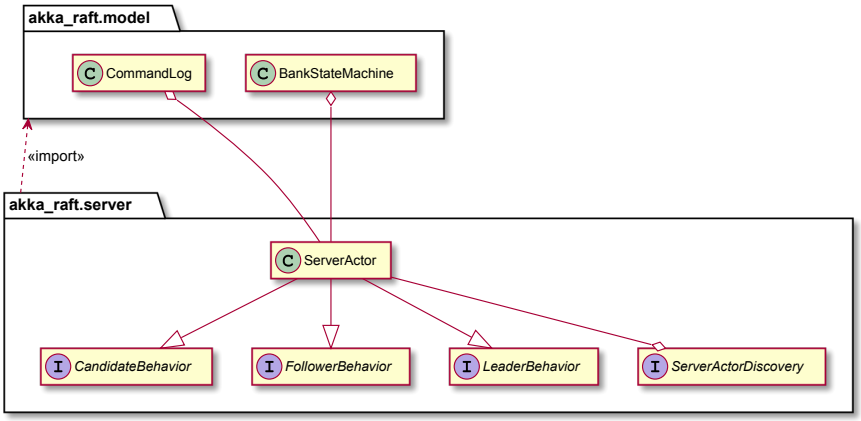

Figura 4.2: Raffigurazione del diagramma delle classi lato server

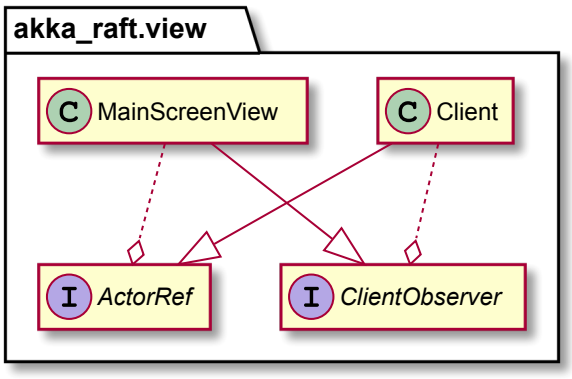

Figura 4.3: Raffigurazione del diagramma delle classi lato client

Qui di seguito viene mostrato un diagramma dei componenti raffigurante i due nodi principali del cluster. Come vediamo, i nodi server sono rappresentati con una nuvola; questo sta a significare che ci troviamo in un ambiente distribuito.

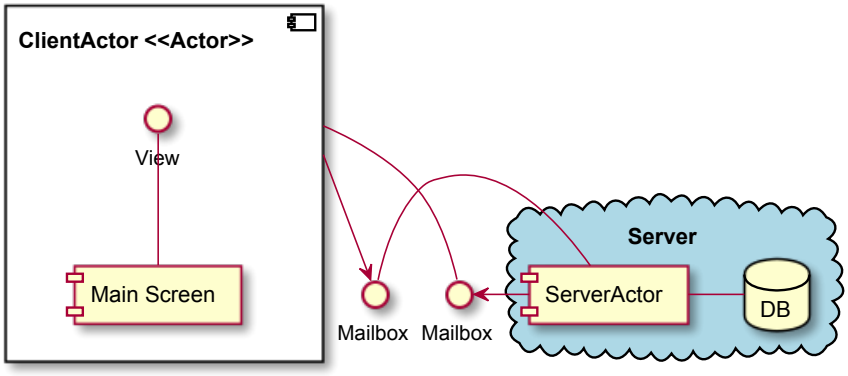

Figura 4.4: Diagramma dei componenti.

4.3.2 Comportamento

Tra gli attori identificati nel sistema, solo uno assume, a seconda dei casi, più di un comportamento: il **Server Actor.**

Tali behaviour sono gestiti nella seguente maniera:

- quando un server è **leader**, svolge le seguenti mansioni:

  - ***accogliere*** *le richieste del client, che arriveranno tramite un messaggio opportuno.*
  - ***appenderle*** *al proprio log, al fine di accodarle per una futura esecuzione.*
  - ***propagare*** *le proprie entry verso gli altri server, allo scopo di replicare il proprio log su tutti i server.*
  - ***committare*** *le entry quando diventa* ***safe*** *farlo.*
  - ***recedere*** *dal proprio ruolo di comando nel momento in cui si trova dinanzi a un poteziale nuovo leader, più qualificato.*

- un server **follower** porta a termine i seguenti compiti:
  - ***appendere*** *le entry del leader al proprio log, quando quest'ultimo ordina di farlo.*
  - ***modificare*** *il proprio log, eliminando le entry non compatibili con quelle del leader, al fine di convergere al suo stesso stato.*
  - ***committare*** *le entry quando il leader indica di farlo.*
  - ***ridirigere*** *il client, comunicando l'identità del leader, in caso si riceva una richiesta.*
  - ***votare*** *per i candidati che richiedono un voto, esprimendo un giudizio positivo o negativo, a seconda del soddisfacimento delle condizioni necessarie.*
  - ***candidarsi*** *come potenziale leader nel momento in cui si perdono i contatti con quello corrente e si assume, di conseguenza, che sia attualmente irraggiungibile.*

- un server in modalità **candidato** esegue le seguenti operazioni:
  - ***inviare*** *richieste di voto agli altri server.*
  - ***contare*** *le risposte positive e cambiare il proprio stato in* ***leader*** *in caso di maggioranza dei voti.*
  - ***recedere*** *a* ***follower*** *nel caso in cui le elezioni siano vinte da qualcun altro.*
  - ***ricandidarsi*** *una seconda volta nel caso in cui, allo scadere di un timer, non si sia raggiunta la maggioranza dei voti nè si sia venuti a conoscenza dell'avvenuta elezione di un nuovo leader.*

4.3.3 Interazione

Qui di seguito verranno elencati i principali messaggi che vengono scambiati tra i vari attori:

RAFT Protocol

- `AppendEntries(leaderTerm, previousEntry,entry, leaderLastCommit)`
- `AppendEntriesResult(success, matchIndex, followerTerm)`
- `RequestVote(candidateTerm, candidateId, lastLogTerm, lastLogIndex)`
- `RequestVoteResult(voteGranted, followerTerm)`
- `ClientRequest(requestId, command)`
- `RequestResult(requestiId, success, result)`
- `Redirect(requestId, leaderRef)`

Control Protocol

- `GuiSendMsg(serverId, commandType, iban, amount)`
- `GuiStopServer(serverId)`
- `GuiResumeServer(serverId)`
- `GuiTimeoutServer(serverId)`
- `GuiMsgLossServer(serverId, loss)`

4.4 Tecnologie

Come abbiamo visto in 4.2.1 il paradigma utilizzato per implementare l'algoritmo RAFT è quello ad **Attori**. Generalmente le tecnologie che implementano il paradigma ad attori assumono due forme fondamentali:

- **Linguaggi ad attori:** sono linguaggi in cui il concetto di attore è parte del linguaggio stesso. Non si può non citare **Erlang**, esso rappresenta uno dei primi linguaggi basati unicamente sul concetto di attore, (**Actor-Based language**). Successivamente altri linguaggi hanno incluso il concetto di attore ma in modo più sfumato; solitamente vengono usati DSL o librerie proprietarie al fine di garantire un'integrazione semi-nativa. Questo è il caso di **Scala** in cui esiste un'estensione ad attori.
- **Framework ad attori:** in questo caso il paradigma ad attori non è integrato nel linguaggio ma viene esteso da un **framework** esterno. La maggior parte delle implementazioni del paradigma ad attori al giorno d'oggi viene fornita attraverso librerie esterne. Fra le librerie più conosciute troviamo **Akka** e **Vertx**.

4.4.1 Akka

Akka è una libreria opensource gratuita che ha lo scopo di semplificare la costruzione e la progettazione di **applicazioni concorrenti e distribuite** in ambiente JVM [1]. La libreria è scritta in Scala e supporta sia Scala che Java; inoltre la suddivisione in moduli permette di poter estendere il numero di funzionalità in modo molto semplice. Nel nostro specifico caso sono stati utilizzati i seguenti moduli:

- **Akka Actor**
- **Akka Cluster**
- **Akka Testkit**

Akka Actor

Il modulo **Actor** rappresenta il modulo fondamentale di Akka: esso fornisce l'astrazione di **Attore** e **Actor System**. Grazie a questo modulo è possibile creare un sistema ad attori estremamente complesso, l'unica limitazione sta nel fatto che il sistema è deployabile solamente su una sola macchina locale. La creazione di un attore viene delegata a un unico Actor System che ha lo scopo di gestirne lo **stato** e l'**esecuzione**. L'Actor System infine gestisce il multithreading garantendo così che in qualsiasi momento un attore possa essere eseguito in maniera safe da un thread per volta.

Akka Cluster

Dato che il paradigma ad attori fornisce la sua massima espressione nel **distribuito**, si è deciso di includere anche l'estensione **Akka Cluster**. Akka Cluster permette di trasformare la propria architettura locale in una architettura **distribuita** o a **microservizi**. [1]
Akka Cluster sposta il concetto di attore nel distribuito, in questo senso un attore non è più locale alla macchina ma può trovarsi in qualsiasi **Nodo** del **Cluster**. Un cluster quindi, si comporrà di vari nodi (macchine) i quali al loro interno avranno in esecuzione un **Actor System**. L' Actor System acquista un ulteriore ruolo, esso infatti non dovrà limitarsi a creare singoli attori ma dovrà anche gestirne la **mobilità** e l'**interazione** nel cluster.

Akka Testkit

Modulo dedicato al **testing** di attori; verrà approfondito nella sezione 4.6 relativa al testing.

[1] Data la *natura indipendente* dei microservizi, essi permettono di *disaccoppiare* in modo molto più netto il flusso di lavoro, garantendo così cicli di sviluppo molto brevi e delivery estremamente veloci. Akka fornisce inoltre un modulo che permette di lavorare con il paradigma **Reactive Streams** insieme al paradigma ad attori. Questo nuovo paradigma chiamato **Reactive Microsystems** permette di gestire in modo estremamente agevole i flussi di dati.

4.4.2 Scala

Come abbiamo visto nella sottosezione precedente, Akka è un framework scritto fondamentalmente in Scala e proprio in Scala definisce un **DSL** (Domain Specific Language) particolare che permette di esprimere concetti complessi in modo semplice ed immediato. Per esempio, in Scala, l'invio di un messaggio viene fatto usando il metodo `tell` o alternativamente usando `!`; questo aspetto simbolico rende Akka molto simile ad una libreria nativa per Scala. [2]
La scelta di Scala è stata guidata anche dalla curiosità verso un nuovo linguaggio più **potente**, **conciso** ed **espressivo** di Java. Scala inoltre introduce un nuovo paradigma di programmazione, quello **funzionale** che permette di lavorare in maniera molto più **snella** ed **elegante** rispetto ad un **OOP castrato** come in Java.

4.5 Dettagli Implementativi

Qui di seguito verranno descritte in dettaglio alcune scelte implementative degne di nota che hanno permesso di migliorare la qualità del codice e l'integrazione fra i vari paradigmi di programmazione.

4.5.1 Incapsulamento di oggetti in attori

Utilizzando un approccio ad attori in un linguaggio OO e funzionale come Scala, è necessario introdurre nuovi pattern di programmazione al fine di garantire una buona integrazione. Nel caso di **classi modulari** come un interfaccia grafica, queste non possono essere trasformate semplicemente in un attore (estendendo la classe attore di Akka) ma devono essere **incapsulate in un attore esterno**. L'interazione fra modulo e attore avverrà tramite il **pattern observer** permettendo così di mantenere un'ottima separazione di ruoli.

4.5.2 Scala Self-Type per Disaccoppiamento Behaviour

Una annotazione self type di un trait definisce il tipo della classe che può estendere il trait. Una classe concreta che effettua il mixin di un trait deve assicurarsi quindi che il suo self-type (`this`) sia conforme a quello del trait con cui è mixata. Ciò significa che l'utilizzo di un self type permette il **mixing** dello stesso **trait all'interno della classe** anche se non lo estende direttamente; questo fa si che i **membri del trait** mixato possano essere **disponibili alla classe** senza dover utilizzare import.
Nel nostro caso i self type sono stati utilizzati per disaccoppiare i vari behaviour `leader`, `candidate` e `follower` e renderli disponibili al `server` in modo separato. Un ulteriore self type è stato utilizzato per gestire la `Discovery` dei nodi del cluster; dato che la discovery consisteva in un'operazione esterna a RAFT si è deciso di "nasconderla" agli attori.

[2] Fun Fact: Scala ha da poco deprecato la propria liberia ad attori in favore di Akka perché più affidabile e potente.

4.5.3 ScalaDoc

Ulteriori dettagli implementativi sono descritti e documentati in maniera approfondita sotto forma di **ScalaDoc**. La ScalaDoc viene generata automaticamente all'avvio della **build** di Gradle oppure usando il task scaladoc.

```
./gradlew scaladoc
```

E' possibile visitare la Scaladoc al link che segue: Akka-Raft ScalaDoc.

4.6 Validazione e Testing

La metodologia di sviluppo utilizzata per questo progetto è stata di tipo **Agile** con approccio **Test-Driven** e **Model-First**.

4.6.1 Approccio Model-First

Lo sviluppo si è concentrato inizialmente sulle classi di modello, nel nostro caso le uniche classi di modello presenti sono:

- `Bank`
- `CommandLog`

Un caso speciale è quello della `BankStateMachine` la quale possiede una natura particolare; essa infatti è un attore ma rappresenta un concetto che può riferirsi anche al modello.

4.6.2 Approccio Test-Driven

Ogni classe di modello è stata sviluppata con approccio Test-Driven, ossia seguendo lo schema ***Red, Green, Refactor***. L'approccio consiste in:

1. ***Red:*** *scrivere test che falliscano*
2. ***Green:*** *scrivere codice che faccia passare i test*
3. ***Refactor:*** *rifattorizzare il codice al meglio*

4.6.3 Test Automatici

I test automatici hanno riguardato solamente le classi di modello. Ogni test è stato eseguito automaticamente tramite *gradle* e ad ogni build ci si è premurati che tutti i test dessero esito positivo. Di seguito vengono riportate le **statistiche** relative ai **test** e la **coverage**:

Risultati Test

Numero totale di test	*25*
Suites completate	*6*
Test completati	*23*
Test ignorati	*2*

Coverage

Class	***Class %***	***Methods %***	***Line %***
Bank.scala	*75*	*88*	*85*
CommandLog.scala	*100*	*61*	*52*
Entry.scala	*50*	*85*	*87*
BankStateMaachine.scala	*69*	*82*	*89*

4.6.4 Integration Test

Sono stati svolti infine due tipologie di integration test sull'intero sistema:

- ***Byzantine test:*** test volto a verificare la vulnerabilità dell'algoritmo contro nodi di tipo **Bizantino**.
- ***Normal operation test:*** test dedicato a verificare la correttezza dell'algoritmo sull'intero cluster.

4.7 Istruzioni per il deployment

Il progetto è stato scritto in `Scala` per la parte core, mentre la parte grafica è stata scritta con il supporto della libreria `JavaFX`. Per la gestione delle dipendenze, per la compilazione e il testing abbiamo optato per il tool `Gradle`.

4.7.1 Installazione delle dipendenze software

Per compilare e eseguire l'applicativo è necessario munirsi di:

- **Java Development Kit:** nonostante sià possibile utilizzare un qualunque `JDK` superiore alla versione 8, consigliamo di utilizzare l'`Oracle JDK-8`. Un'altra valida alternativa è la distribuzione `Amazon Corretto-8` [2]. Queste distribuzioni incorporano nativamente `JavaFX`, permettendo di risparmiare così ulteriori installazioni.
- **Scala:** per l'esecuzione del programma è necessario munirsi di `Scala 2.12.10`, non garantiamo che un'installazione più recente sia retro-compatibile.

4.7.2 Build e run

L'adozione di `Gradle` ha reso molto semplice la compilazione del progetto. Per la generazione di un singolo `jar` eseguibile completo, è stato scelto l'utilizzo del plugin `Gradle Shadow` [6] che permette la generazione di un jar a sé stante che soddisfi tutte le dipendenze internamente.
Per compilare il progetto è sufficiente recarsi nella cartella principale e lanciare il seguente comando:

- Per i sistemi **Unix**:

```
./gradlew build
```

- Per i sistemi **Windows**:

```
gradlew.bat build
```

Il comando procederà alla compilazione dei sorgenti scala, fornendo in uscita un `jar` eseguibile che sarà collocato nella cartella */build/libs* sotto il nome di `akka-raft-1.0.0-all.jar`
Il `jar` esegue internamente il main presente nell'`object JarLauncher` del package akka_raft. Per lanciare il jar è necessario eseguire il comando:

```
scala akka-raft.jar tipologia id porta
```

- **Tipologia:** le **tipologie** disponibili sono client e server. Ogni tipologia esegue una sola istanza del nodo corrispondente. Nell'attuale configurazione, per l'esecuzione completa del cluster è necessario eseguire un solo client e esattamente cinque server, l'esecuzione di un numero diverso non permetterà al cluster di proseguire correttamente.
- **ID:** l'**id** è una stringa identificativa dell'attore che viene eseguito, è possibile scegliere la stringa che più ci piace, a patto che sia univoca.
- **Porta:** la **porta** è il numero di porta assegnato da akka remoting ad ogni attore, inserire un valore pari a 0 equivale a richiedere una porta random libera.

Per l'esecuzione del cluster Akka è necessario lanciare due **seed node**, i quali devono possedere una porta specifica(rispettivamente 5000 e 5001), definite nel file `cluster.conf`.
Si consiglia di eseguire 6 **istanze** del jar seguendo il seguente ordine:

```
scala akka-raft-1.0.0-all.jar client C0 5000
```

```
scala akka-raft-1.0.0-all.jar client S0 5001
```

```
scala akka-raft-1.0.0-all.jar client Sn 0
```

Sarà necessario eseguire il terzo comando con porta 0 altre quattro volte, per raggiungere il numero esatto necessario per la partenza del cluster.

4.7.3 Test

Sono stati prodotti vari test, automatizzati e di integrazione. Per il lancio dei test è stato creato un task chiamato `spec`, utilizzato per eseguire i test scala con `org.scalatest.tools.Runner`.
Per l'esecuzione dei test è sufficiente eseguire gradle dalla cartella principale del progetto, tramite il lancio del comando:

- Per i sistemi **Unix**:

```
./gradlew test
```

- Per i sistemi **Windows**:

```
gradlew.bat test
```

4.8 Esempi D'Uso

Una volta avviati tutti i componenti del cluster (Client e Servers): sarà necessario attendere che la schermata si aggiorni con le informazioni iniziali dei vari server.

La schermata principale con i suoi relativi comandi è la seguente:

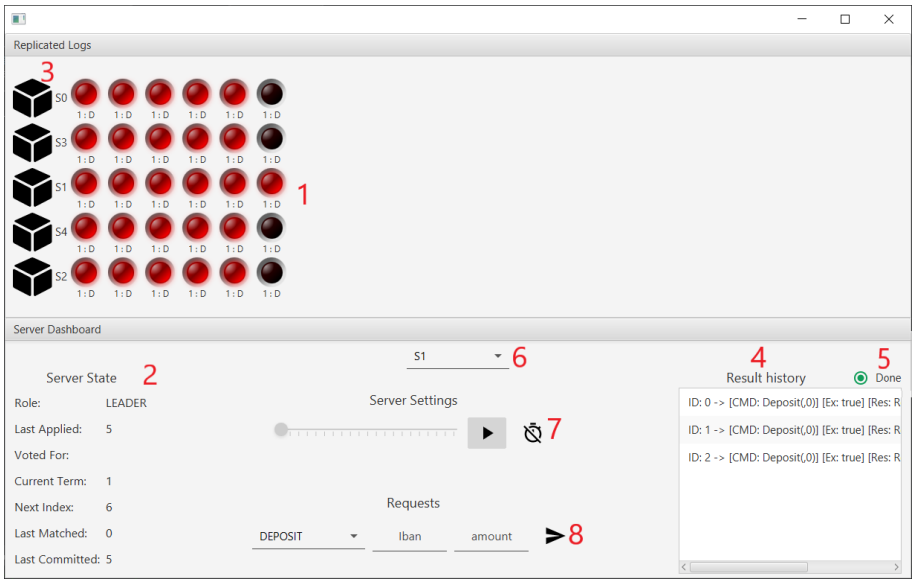

Figura 4.5: Screenshot della schermata principale durante il normale funzionamento. In rosso vediamo elencati tutti gli elementi fondamentali della Gui.

Procediamo nel descrivere tutti i principali elementi presenti nella schermata:

1. `Log di tutti i server`: ogni server ha un **log** rappresentato da un insieme di **entry**, ogni entry è costituita da un term (un numero) e un comando (`Deposit`, `Withdraw` e `Get Balance`) rappresentato dalle lettere: `D`, `W` e `G`. Inoltre ogni entry del log ha un colore: nero o rosso. Rosso significa che l'entry ha ancora ricevuto il commit, nero che non lo ha ricevuto.

2. `Stato del server selezionato:` in questo riquadro sono riportate le informazioni più importanti del server attualmente selezionato.

3. `Colonna dei server:` nella colonna sottostante per ogni server sono riportati il nome e lo stato del log allineato in riga.

4. `Storico risultati/richieste:` in questa textarea sono riportate tutte le richieste e i relativi risultati, se presenti. Nella stessa area, inoltre, sono riportate anche le richieste pendenti.

5. `Tasto per cambio storico:` tramite questo tasto è possibile passare dalla visualizzazione dello storico risultati, alla visualizzazione delle richieste pendenti.

6. `Selezione server:` tramite la combobox è possibile selezionare un server del cluster.

7. `Comandi server selezionato:` i tre comandi disponibili sono tutti indipendenti tra loro, da sinistra verso detra i significati sono:

   - `Slider:` Lo slider parte da zero e identifica la percentuale di perdita di pacchetti che il server selezionato ha la probabilità di subire. Il valore viene impostato una volta che lo slider viene rilasciato, e non durante lo spostamento; inoltre i valori vengono approssimati di cinque in cinque.
   - `Start/Stop:` il tasto start/stop permette di bloccare e riprendere l'esecuzione del server selezionato.
   - `Timeout:` tramite questo tasto è possibile inviare un messaggio di timeout al server selezionato. Questo messaggio farà scattare il **Timer** del server; che quindi si comporterà diversamente in base al suo stato corrente (Follower, Leader o Candidate).

   Ogni comando viene inviato al server selezionato in via prioritaria, e quindi viene recapitato il prima possibile; indipendentemente dallo stato del server selezionato.

8. `Interfaccia di invio richieste:` in questo punto è possibile inviare nuove richieste da parte del `Client` verso il `Server` attualmente selezionato. Una volta scelto il comando tra: Deposit, withdraw e get balance; sarà necessario inserire il codice `iban` del conto e l'importo per la specifica operazione. Infine è possibile inviare la richiesta. In caso non si selezioni nulla vengono scelti iban nullo e valore zero.

Non è possibile procedere con il lancio di alcun comando fino a quando sono presenti le scritte **undefined** nello stato del server.

In figura: 4.5 è riportato un esempio in cui il server $S1$, ovvero l'attuale, ha replicato cinque entry e stava procedendo alla diffusione dei messaggi di commit con il successivo `Heartbeat`. Ma è stato bloccato dal `Client`. Possiamo notare come tutte le entry hanno lo stesso term, quindi sono state replicate durante il regno del primo leader. Le entry da lui replicate non sono ancora committate, e data la sua assenza possiamo certamente aspettarci l'elezione di un nuovo **Leader**, che si metterà in attesa di nuove entry da replicare.

Capitolo 5

Conclusioni

Giunti alle conclusioni di questo progetto possiamo affermare di ritenerci estremamente soddisfatti del lavoro svolto. Il risultato ottenuto ha superato di gran lunga le nostre aspettative ma la strada che ha portato alla sua realizzazione è stata spesso impervia.
La realizzazione di un algoritmo di consenso richiede un enorme mole di studio e lavoro. Nel nostro caso il lavoro svolto si è suddiviso in questo modo:

- *Studio e documentazione:* abbiamo impiegato all'incirca una settimana di lavoro al fine di comprendere e documentare al meglio l'algoritmo RAFT. Una volta che l'algoritmo è stato compreso si è potuto lavorare sull'effettiva implementazione concentrandosi inizialmente sulle varie tecnologie implementative. Dopo una breve indagine è stato scelto il framework Akka.

- *Studio di Scala:* una volta che è stato individuato il framework di sviluppo, la scelta del linguaggio è stata quasi obbligata. Akka infatti, come abbiamo già visto, in Scala, permette di lavorare in modo estremamente conciso ed efficiente. L'apprendimento di Scala non è stato immediato e per comprenderlo al meglio abbiamo impiegato una settimana di studio full-time.

- *Studio di Akka:* appreso il linguaggio Scala ci siamo concentrati sullo studio del framework Akka, in particolare ci siamo concentrati sulla libreria di Akka dedicata agli attori e al clustering.

- *Implementazione vera e propria:* finalmente appresi i linguaggi e le tecnologie necessarie ci siamo concentrati sull'implementazione dell'algoritmo RAFT. Siamo partiti dal modello e abbiamo proseguito a sviluppare in modo incrementale i vari attori.

- *Debugging:* completata l'implementazione l'algoritmo ha presentato un numero considerevole di bug. La fase di debugging è stata la fase più

frustrante del progetto ma dopo grandi difficoltà siamo riusciti ad ottenere un'implementazione funzionante e pulita dell'algoritmo.

5.1 Lavori Futuri

Il lavoro svolto come abbiamo visto è limitato a una semplice implementazione dell'algoritmo RAFT. Dato che l'algoritmo è stato sviluppato relativamente a un **caso d'uso specifico**, *conti correnti bancari*, esso manca di estendibilità e genericità. Uno sviluppo futuro potrebbe risiedere nella possibilità di fornire una **versione generica** dell'algoritmo che sarà poi estendibile concretamente in base al caso d'uso.
Un'altra possibile estensione futura dell'implementazione consiste nell'inserimento delle *features extra* fornite dall'algoritmo RAFT: **Log Compaction** e **Configuration Changes**.

5.2 Cosa Abbiamo Imparato

Quando si sceglie un progetto sostanzioso spesso si rischia di rimanere sommersi dal carico di conoscenze necessarie per portare a termine il lavoro. Nel nostro caso ci sono stati momenti di sconforto che, con supporto reciproco, siamo riusciti a superare. Gli argomenti che inizialmente si sono mostrati più ostici alla fine sono entrati a far parte del nostro bagaglio conoscitivo.

- *Scala:* inizialmente abbiamo appreso solo le basi di questo linguaggio; il perfezionamento delle competenze è poi avvenuto in maniera passiva man mano che iniziavamo a far fronte ai diversi problemi che si presentavano. Questa esperienza ha avuto un impatto positivo: siamo entrati in contatto con un nuovo paradigma di programmazione, quello **funzionale**, e abbiamo imparato a lavorare in un ambiente OOP puro.

- *Akka:* il framework Akka è sterminato e uno studio di un mese non basterebbe a coprire tutte le estensioni presenti. Nel nostro caso ci siamo concentrati solo sui moduli dedicati agli attori e al clustering e possiamo affermare che la conoscenza dei moduli è completa.

- *Algoritmi di Consenso:* studiando RAFT abbiamo compreso il **funzionamento** e la **complessità** di un **algoritmo di consenso**, inoltre abbiamo toccato con mano la **complessità insita nelle interazioni fra entità nel distribuito**. Infine lo studio di RAFT ci ha permesso di entrare a conoscenza del fantastico mondo delle **macchine a stati replicate**.

Note Stilistiche

- Le entità di progetto sono espresse in formato `Teletypefont`.
- Per gli indici e le variabili numeriche è stata usata la modalità *math*. Es $n \in C$
- Gli spezzoni di condice sono espressi in formato *listing*

```
echo ``Hello World''
```

Bibliografia

[1] *Akka's site*. https://akka.io/docs/.

[2] *Amazon corretto's site*. https://aws.amazon.com/it/corretto/.

[3] E. A. Brewer. «Towards robust distributed systems». In: *In 19th Annual ACM Symposium on Principles of Distributed Computing* (2000).

[4] G. E. A. Coulouris. *Distributed Systems. Concepts and Design*. Pearson, 2012.

[5] R. Friedman e K. Birman. *Trading consistency for availability in distributed systems*. Technical report. Cornell University, 1996.

[6] *Gradle shadow's site*. https://imperceptiblethoughts.com/shadow/.

[7] Leslie Lamport. «Time, clocks, and the ordering of events in a distributed system». In: *Communications of the ACM* 21.7 (lug. 1978), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: http://doi.acm.org/10.1145/359545.359563.

[8] Diego Ongaro. *Consensus: Bridging Theory and Practice*. Stanford University, 2014. URL: http://purl.stanford.edu/qr033xr6097.

[9] Diego Ongaro e John Ousterhout. «In Search of an Understandable Consensus Algorithm». In: *Proc. ATC'14, USENIX Annual Technical Conference*. USENIX, 2014.