HQ Replication: Analisi e implementazione di un protocollo ibrido per accordo e BFT in un sistema distribuito

Leo Pasquale
pasquale.leo@studio.unibo.it

Cassano Francesco
francesco.cassano2@studio.unibo.it

Maggio 2021

Esistono fondamentalmente due approcci per fornire una macchina a replicazione di stato con tolleranza ai fallimenti bizantini: un approccio replica-based in cui le repliche si mettono d'accordo autonomamente parlando tra di loro e fornendo infine un risultato al client, e un approccio con quorum-based in cui i client contattano direttamente tutte le repliche e in caso di un quorum di consensi permette l'esecuzione delle operazioni. Entrambi hanno grosse debolezze, infatti se un approccio replica-based mantiene basso il numero di repliche, d'altra parte soffre l'enorme numero di comunicazioni che servono per scambiare le informazioni risultando quindi, molto lento, mentre per quanto riguarda i metodi quorum-based sono molto più veloci a rispondere ma ha bisogno di un numero molto più alto di repliche per resistere ai fallimenti che si verificano più di frequente rispetto ad un replica-based. Con un protocollo ibrido l'idea è quella di sfruttare i punti di forza di entrambi i sistemi e superare le loro debolezze. Il protocollo sviluppato unisce i protocolli Q/U (quorum-based) e BFT (replica-based), e premette di minimizzare il numero di repliche necessarie a 3f+1 dove f sono le repliche fallite (numero di repliche minimo per un sistema che deve resistere ai fallimenti).

Contents

1 Introduzione ai sistemi distribuiti

Un sistema distribuito è un particolare sistema in cui le componenti sono distribuite nello spazio e connessi da una rete di comunicazione asincrona, pertanto essi sono caratterizzati da assenza di clock condiviso, assenza di memoria condivisa e assenza di un accurato failure detector, ciò è dato dall'impossibilità di sapere in che ordine i messaggi verranno ricevuti in quanto soggetti a ritardi arbitrari e finiti. Appare evidente che in un sistema con queste caratteristiche è altamente vulnerabile a difetti e malfunzionamenti di ogni genere, infatti, anche un piccolo fallimento in un qualsiasi punto del sistema, può comprometterne l'intero funzionamento. Nonostante ciò, esistono diverse tecniche in grado di offrire consistenza e disponibilità ad un sistema distribuito. Uno dei meccanismi di tolleranza dei fallimenti è la state machine replication (SMR), che viene sviluppato replicando i server e coordinando le operazioni di interazione con i client. I benefici introdotti con un sistema SMR in un ambiente distribuito sono notevoli, tuttavia, se da un lato si migliora la disponibilità del servizio e la robustezza ai guasti, dall'altro si introducono nuove problematiche quali: inconsistenza dei dei dati e cooperazione dei processi. La SMR deve prevedere anche una struttura per la comprensione e la progettazione di protocolli di gestione della replica. Un sistema a replicazione di stato deve portare avanti i dati sulle diverse repliche in modo consistente e al quanto sincronizzato, ma dato che la sincronia del sistema e l'ordinamento dei messaggi sono dei casi particolari questo tipo di sistema ha un'alta probabilità di fallimento. Per fa si che le diverse repliche si evolvano in maniera uniforme, il sistema necessita di un "accordo". Questo problema è conosciuto come problema del consenso e viene risolto grazie a diversi algoritmi.

1.1 Fallimento di un sistema distribuito

La natura di un sistema distribuito, pone il terreno fertile all'insorgere di fallimenti che se non gestiti adeguatamente possono mettere "KO" il sistema nella sua interezza. In un contesto come questo, richiedere contemporaneamente certe proprietà come la consistenza, la tolleranza agli errori e l'alta disponibilità è difficili o quasi impossibili da garantire, come dimostra il teorema di CAP che afferma, che solo due di queste proprietà possono coesistere nello stesso istante. La conseguenza delle precedenti osservazioni è che: "non si può avere tutto" ma, per risolvere certe problematiche sono necessari dei compromessi.

Ad esempio, nel problema del consenso è possibile notare che la presenza di anche solo una singola failure non prevista in una rete distribuita asincrona rende il problema del consenso impossibile da risolvere, pertanto, risulta fondamentale inserire elementi sincroni nella rete che controllino il mantenimento delle condizioni che permettono il funzionamento del sistema. I tipi di fallimento rilevabili in sistemi distribuiti sono:

- **crash**: il fallimento corrisponde ad uno stop imprevisto di un processo
- **crash + link**: il fallimento può dipendere sia dallo stop di un processo che dal down di un collegamento
- **omissione**: un processo viene considerato fallito se invia o riceve meno messaggi del dovuto
- **fallimento bizantino**: un processo fallisce esibendo un comportamento arbitrario. Questo fallimento in particolare è più difficile da individuare.

Se per i primi 3 tipi di fallimento l'individuazione del nodo fallito è relativamente semplice il fallimento bizantino ha richiesto uno studio e il cambiamento di alcuni protocolli.

1.2 Il nodo bizantino

I difetti bizantini sono considerati la classe di guasti più generale e più complessa da risolvere. Per nodi bizantini si intendono quei server che mostrano diversi comportamenti con diversi osservatori, pertanto è difficile, per gli altri componenti, dichiararli falliti ed escluderli dalla rete, infatti, hanno bisogno di un consenso. Questo problema è conosciuto come il problema dei generali bizantini, in quanto Lamport ed al., i primi a descriverlo, lo hanno ipotizzato come il problema di diversi generali bizantini che devono attaccare una città sotto assedio, ma per farlo devono sferrare un attacco coordinato ed evitare falle nella comunicazione. Sempre secondo Lamport et al.[5], un sistema informatico affidabile deve essere in grado di far fronte al guasto di uno dei suoi componenti, pertanto propongono una soluzione in cui affermano che, è necessario un protocollo nel quale i generali fedeli (nodi non falliti) possano coordinare le operazioni a prescindere dalla presenza dei nodi faulty, e che dati f nodi faulty, un sistema può gestire i fallimenti se ha un minimo di 3f + 1 repliche.

2 Protocollo Hybrid-Quorum

Il protocollo Hybrid-quorum è un ibrido tra i protocolli quorum-based e quello replica-based. Fare ciò permette di unire i vantaggi di entrambi e sopperire alle loro mancanze, infatti, gli algoritmi basati su quorum richiedono un alto numero di repliche ma garantiscono una maggiore velocità di raggiungimento di consenso e quindi risposta, invece quelli basati su repliche consentono di mantenere basso il numero di repliche anche se potrebbe richiede un alto numero di messaggi per raggiungere il consenso. Nella fattispecie, i protocolli scelti nell'implementazione proposta sono il pBFT (Pratical Byzantine Fault Tollerance) come protocollo replica-based e il protocollo Query/Update per quanto riguarda i quorum-based. Il protocollo HQ quindi, nato per superare i limiti dei precedenti protocolli, si comporta sostanzialmente come un Query/Update mantenendo basso il numero di operazioni e controlli dei dati, ma richiede solo 3f + 1 repliche contro le 5f + 1 repliche. Questo perché l'alto numero di repliche serve a risolvere i conflitti, che nel caso di HQ, sono risolti in modo più agile grazie all'utilizzo del protocollo pBFT. In assenza di conflitto le richieste di lettura/scrittura utilizzano un protocollo basato sul quorum di 2f + 1 repliche, necessario per validare le operazioni di lettura e scrittura, che vengono eseguite rispettivamente in un round-trip e due round-trip di comunicazione fra client e i server, infatti se in lettura abbiamo solo richiesta e risposta, per quanto riguarda la fase di scrittura dopo la richiesta il client aspetta che tutte le repliche gli concedano il permesso, il cui quorum produrrà un certificato di scrittura necessario alla replica per completare l'operazione[2]. Se si verifica un conflitto, ovvero 2 o più client si contendono una risorsa, richiedendo una scrittura nello stesso istante, si utilizza l'algoritmo pBFT per trovare un accordo sull'ordine con cui le operazioni devono essere eseguite, e solo dopo vengono eseguite secondo l'ordine concordato.

2.1 Protocollo Query/Update

Query/Update (Q/U) è il principale protcollo su cui HQ si appoggia. È un protocollo per *fault-scalable system* che tende a distinguere nettamente operazioni di lettura, per l'appunto query, e quelle di scrittura, update, permettendo un veloce procedimento di accordo fondato sul quorum che non subisce logoramento delle performance all'aumentare della tolleranza ai fallimenti. Questo è il motivo per cui è stato scelto come core di HQ, infatti, seppur gli algoritmi BFT, o comunque con accordo siano più affidabili, sono poco portati alla scalabilità, in quanto subiscono un brusco calo di prestazioni all'aumentare dei fallimenti tollerati. Abd-El-Scalable et al. [3] stimano che le performance di Q/U diminuiscano solo del 36% portandone la tolleranza da 1 a 5. Seppur un servizio realizzato con Q/U sia molto efficiente, è anche vero che è particolarmente soggetto a conflitti che non permettono al richiedente di ottenere un quorum valido per completare l'operazione, in quanto entra in concorrenza con altri client per la risorsa. Per quanto riguarda le caratteristiche numeriche che consentono il funzionamento di una SMR e regolano la gestione dei fallimenti, bisogna fare delle assunzioni matematiche sul funzionamento del protocollo che Abd-El-Scalable et al. [3] hanno ottenuto estendendo quelle individuate da Malkhi e Reiter [6]. Definito U l'insieme di tutti i nodi n è identificato come la sua cardinalità, ovvero il numero di nodi. È possibile dividere questo insieme nei sottoinsiemi Q, che indica il set di repliche compone il quorum, T, che indica quello delle repliche fallite, B, che indica il set delle repliche bizantine, e infine R, che indica il set delle repliche riparabili, rispettivamente di cardinalità q, t, b ed r. Queste assunzioni sono necessarie per definire le proprietà fondamentali del sistema, ovvero:

$$\begin{aligned} n = &\quad 3t + 2b + 1 \\ q = &\quad 2t + 2b + 1 \\ r = &\quad t + b + 1 \end{aligned}$$

Essendo i fallimenti generici e fallimenti bizantini considerati come se facessero parte dello stesso gruppo di fallimenti si può semplificare il tutto ponendo $t = b$ e di conseguenza ottenere $n = 5b+1, q = 4b+1$ e $r = 2b+1$. Per rendere efficiente il meccanismo di Q/U è fatto largo uso dell'astrazione delle operazioni sugli oggetti, definite sul server ma richieste dai client tramite messaggi. L'astrazione raccoglie diversi tipi di informazioni riguardanti l'operazione, ovvero, la classe (query, update), il tipo e il set degli argomenti necessari per il suo svolgimento. Ne consegue che, lato client, l'operazione è così composta

$$Operation = \langle Method, Class, Arguments \rangle$$

È possibile concludere che Q/U sia un valido protocollo ma l'elevato rischio di corse alla sezione critica rende indispensabile rispettare i vincoli numerici posti affinché il servizio sia in grado di resistere ai fallimenti, e seppur le prestazioni restano piuttosto costanti all'aumentare della tolleranza, le repliche richieste aumentano in modo considerevole, infatti, nell'esempio posto da Abd-El-Scalable et al. al variare della tolleranza da 1 a 5 se le prestazioni calano del 36%, le repliche passano da 6 a 26, ovvero sono più che quadruplicate.

2.2 Practical Byzantine Fault Tolerance (pBft)

Il protocollo HQ si appoggia su un secondo protocollo per la risoluzione delle contese fra le repliche, noto come: *Practical Byzantine Fault Tolerance (pBft).*

Il protocollo in esame è una delle tante soluzioni presenti in letteratura, per risolvere il problema dei generali bizantini, dove, in un sistema distribuito con N repliche e f possibili fallimenti lo scopo è:

1. garantire che tutte le repliche non fallite si accordano sullo stato successivo;
2. fornire consistenza al sistema anche quando alcune repliche potrebbero essere inconsistenti.

Dunque, le proprietà che il protocollo deve fornire al sistema sono di:

- **Safety**: tutte le repliche non fallite devono essere d'accordo sullo stesso stato valido;
- **Liveness**: se uno stato è stato scelto, tutte le repliche eventualmente arrivano su quello stato.

2.2.1 PBft: Modello

Al fine da comprendere a pieno il corretto funzionamento del protocollo è bene fare delle brevi puntualizzazioni sul modello operativo, fatto come segue:

- un insieme di repliche pari a $|R| = 3f + 1$ (f è il numero massimo di fallimenti tollerati);
- ogni replica è identificata da un numero intero compreso $\{0, ..., 3f\}$;
- ogni replica è deterministica e inizia dallo stesso stato iniziale;
- la view è una configurazione di repliche, dove, $p = v \times mod|R|$ è il primario della view v, mentre tutte le altre costituiscono le repliche di backups.
- in caso il primario fallisce è invocato il cambio di vista (*View change*) che incrementa la vista e determina un nuovo primario.

2.2.2 PBft: Condizioni normali (primario non è fallito)

A questo punto abbiamo tutti gli elementi per addentrarci nel protocollo cosicché da comprendere al meglio le scelte di design nel capitolo preposto. La fasi principali del protocollo si articolano nei seguenti step:

1. un cliente invia la richiesta al primario;
2. il primario valida la richiesta e inizia il protocollo a tre fasi (pre-prepare ⇒ prepare ⇒ commit) utile a garantire il consenso fra tutte le repliche non fallite.
3. la replica esegue la richiesta e invia il risultato direttamente al cliente;
4. il cliente accetta il risultato dopo aver ricevuto $f + 1$ risultati identici.

Il cuore fondamentale del protocollo è appunto il protocollo a tre fasi (vedi fig. 1) il quale si propone di soddisfare le proprietà precedentemente citate. Vediamo brevemente in cosa consistono le singole fasi:

1. **Pre-prepare**: quando il primario riceve una richiesta attraverso il messaggio m, propone una sequenza numerica n. La sequenza numeri è utile per rafforzare l'ordine totale delle richieste, poiché essendo la rete asincrona le repliche potrebbero ricevere le richieste in ordine diverso. Dopo questo, il primario trasmette alle repliche il messaggio di *Pre-Prepare(m, n, v, d)*. Le repliche accettano il messaggio se:
   - v (view), n (sequenza numerica), d (digest del messaggio) validi;
   - $n \in \{h, h+L\}$, dove h è la sequenza numerica dell'ultimo checkpoint stabile, mentre L è la lunghezza del log;
2. **Prepare** quando una replica i accetta il messaggio *Pre-Prepare(m, n, v, d)* passa nella fase successiva inviando alle altre repliche il messaggio *Prepare(n, v, d, i)*. La replica aggiunge al log il messaggio di Pre-Prepare e Prepare. Il predicato di *Prepared(n, m, i)* diventa vero quando una replica i nel proprio log ha i seguenti elementi:
   - un messaggio di *Pre-Prepare* di una specifica richiesta m, relativo alla view v e sequenza numerica n;
   - almeno $2f$ messaggi di tipo *Prepare* provenienti dalle altre repliche di backup.
3. **Commit** una volta che il predicato *Prepared* diventa vero, la replica i invia alle restanti repliche il messaggio *Commit(n, m, i)*. Il predicato *Commited(n, m, i)* diventa vero quando:
   - una replica i custodisce nel proprio log $2f + 1$ messaggi di *Commit(n, m, i)* provenienti da differenti repliche;
   - la replica ha eseguito tutte le richieste aventi numero di sequenza minore di quello attuale.

   Quando il predicato diventa vero, la replica esegue l'operazione contenuta nel messaggio e può finalmente ritornare il risultato.

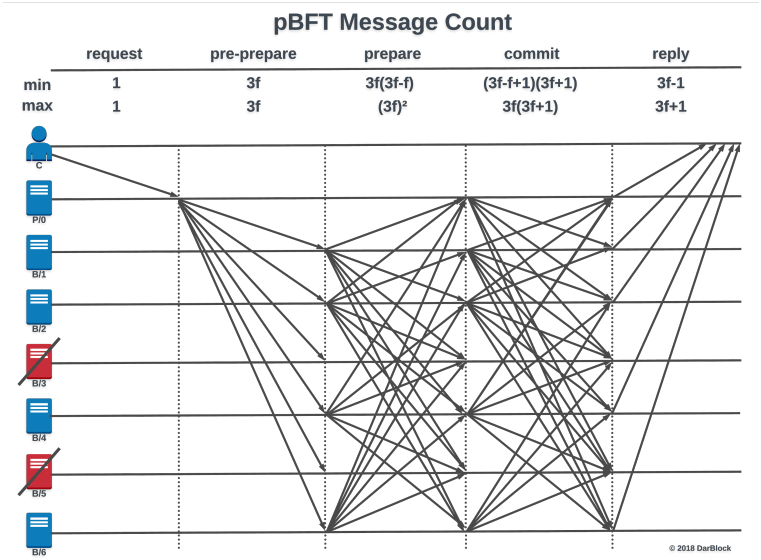

Figure 1: Protocollo a tre fasi utilizzato nel pBft

2.2.3 PBft: View Changes (primario fallito)

Una fondamentale procedura utilizzata dal protocollo per conferire al sistema la proprietà *liveness* al sistema è quella del *view change*, letteralmente cambio di vista e permette il progresso quando il primario fallisce. Il protocollo deve anche preservare la proprietà di *safety*, ovvero garantire che le repliche non fallite si accordano sulla sequenza numerica della richiesta relativa a quella view. Quando le repliche si accorgono che il primario è fallito invia al primario della vista successiva la richiesta di cambio di vista. Quando il nuovo primario riceve *2f* messaggi di ⟨ViewChange⟩ darà inizio ad una nuova view inviando a tutte le repliche il messaggio di $\langle NewViewChange \rangle$. Le repliche quando riceveranno tale messaggio incrementano la view e ritornano nello stato iniziale, pronte a iniziare un nuovo ciclo del protocollo.

2.2.4 PBft: Checkpoint

Il protocollo prevede una fase in cui tutte le repliche si accordano sullo stato raggiunto, noto come stato di *Chechpoint*. Tale stato funge come condizione di ripristino in caso di fallimenti da parte di qualche nodo della rete, ma soprattutto è utile per impedire la saturazione della memoria usata per il log dei messaggi.

Infatti, in modo ciclico il sistema sospende il suo normale funzionamento, dando modo alle repliche di accordarsi sull'ultima richiesta eseguita, attraverso lo scambio del messaggio $\langle CHECKPOINT, n, d, i \rangle$, allo scopo di eliminare tutti i messaggi dal log con sequenza numerica minore di n.

3 Obiettivi e requisiti

3.1 Obiettivi

Analizzati i problemi derivanti dalla costruzione di un sistema distribuito, gli obiettivi del progetto saranno:

- Analisi del protocollo Hybrid-quorum
- Implementazione del protocollo
- Test delle componenti dell'implementazione
- Adattamento per una semplice SMR

3.2 Requisiti

I risultati saranno considerati soddisfacenti se il sistema rispetterà questi canoni:

- il codice prodotto può essere adattato anche in altri contesti che non siano quelli della demo per la presentazione
- il sistema deve lavorare in maniera autonoma, ogni replica deve svolgere il suo lavoro e può essere controllato, gestito o può coordinarsi con altre repliche solo tramite messaggi
- il sistema non opera su uno stato globale, ogni replica mantiene la sua memoria e le altre non possono avervi accesso
- il sistema è concorrente possono essere eseguite operazioni diverse su macchine diverse
- nel tempo il sistema deve rispettare il più possibile le 3 proprietà dei sistemi distribuiti: coerenza, disponibilità, e tolleranza di partizione
- il sistema è in grado di raggiungere il consenso in presenza di fallimenti o nodi bizantini;
- il sistema è in grado di definire un ordine delle richieste utente, risolvendo le situazioni di conflitto in occasione di più richieste da parte di clienti diversi sulla stessa risorsa;

3.3 Scenario

L'utilizzo del sistema è perlopiù dimostrativo, per questo ci siamo proposti come obiettivo anche la composizione di una API con la quale è possibile implementare servizi che agiscono tramite il protocollo HQ. Pertanto ciò che intendiamo presentare è il funzionamento del protocollo e la sua resistenza ai fallimenti a prescindere dal motivo per cui si verificano, ovvero:

- normale funzionamento del protocollo in maniera distribuita: operazioni di lettura e scrittura senza conflitti (esempio di scrittura in figura 2);
- robustezza del protocollo al presentarsi del conflitto: le repliche sono in grado di trovare accordo, risolvere il conflitto per poi riprendere il normale funzionamento (esempio di scrittura in figura 3);
- robustezza del protocollo se una replica resta indietro: mentre le altre repliche garantiscono il funzionamento del sistema la replica che resta indietro recupera i dati e ritorna al normale funzionamento;

- robustezza del protocollo in concomitanza di conflitto e update: le repliche fallite o rimaste indietro (se sono meno di f) non influiscono con il corretto funzionamento del sistema;

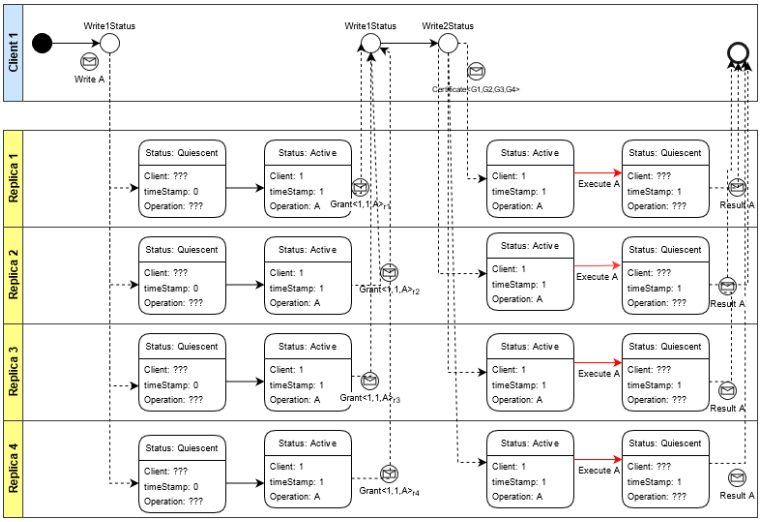

Figure 2: Operazione di scrittura in assenza di conflitto

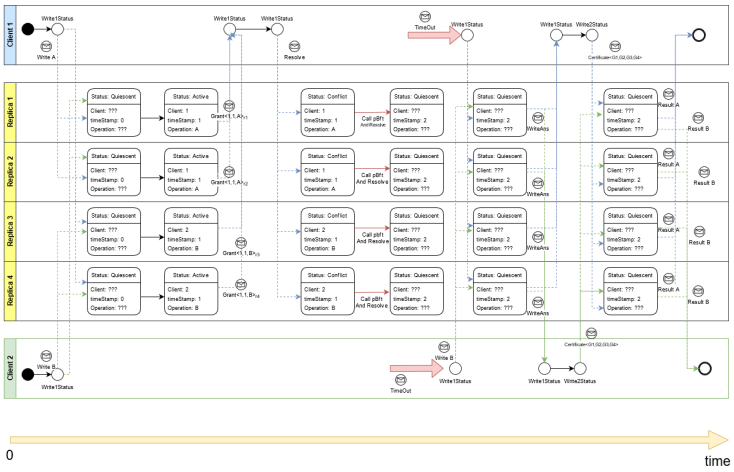

Figure 3: Operazione di scrittura in presenza di conflitto

3.4 Requisiti di implementazione

Il sistema presenta assenza di memoria condivisa, scambio di informazioni tramite messaggi asincroni, e una lieve sincronia delle operazioni raggiunta tramite il protocollo hq, di conseguenza abbiamo scelto l'astrazione dell'attore che permette di valorizzare tutti questi aspetti. L'attore è un componente reattivo, ovvero che reagisce a degli stimoli, i messaggi. Tutti i messaggi sono gestiti da un handler che è possibile definire tramite l'implementazione di un behaviour. I behaviour definiscono anche lo stato dell'attore in quanto al cambiare di questo cambieranno i messaggi da gestire e le procedure da eseguire. Per la realizzazione del sistema ad attori è stata usata la libreria akka. Il linguaggio usato per implementare il sistema è scala, in quanto akka esprime tutte le sue potenzialità in un linguaggio funzionale come scala. Per far si che scala funzioni su qualunque macchina abbiamo utilizzato sbt per fare la build del progetto.

3.5 Metodo di autovalutazione

I vari attori sono stati scritti in modo modulare e lo sviluppo è stato incrementale, di conseguenza abbiamo potuto valutare il sistema in termini di qualità ed efficacia utilizzando vari test, e verificando ogni passaggio realizzato assicurandoci del suo funzionamento normale e in casi limite. Essendo anche l'attore soggetto a cambiamento di comportamenti ci siamo assicurati del corretto funzionamento dei vari passaggi di stato. I test saranno discussi nel modulo Test.

3.6 Assunzioni

La realizzazione del protocollo si basa sulle seguenti assunzioni:

- saranno considerati falliti (f) tutti i nodi a prescindere dal tipo di fallimento;
- fallimenti nella consegna dei messaggi;
- duplicazione dei messaggio;
- corruzione dei messaggi;
- nessuna garanzia sull'ordine di arrivo dei messaggi, perciò non si possono fare supposizioni su un possibile ordine totale o parziale nella consegna dei messaggi;
- non ci sono limiti sui ritardi di consegna;
- non ci sono limiti sul tempo di esecuzione delle applicazioni;
- i nodi nella comunicazione utilizzano delle firme digitali che non possono essere falsificate;
- si etichettano i messaggi con un nonce (numero pseudocasuale) per impedire attacchi di tipo replay;
- ogni richiesta prima o poi verrà esaudita;
- per giungere a destinazione un messaggio ci impiegherà al più un tempo T, quindi, se non si arriva ad una risposta entro 2T il messaggio si considera perso;
- Non più di $f = \frac{n-1}{3}$ nodi possono fallire

4 Design

Il sistema realizzato fornisce di base delle API con le quali è possibile implementare ogni sorta di servizi distribuiti che intenda utilizzare il protocollo HQ. Per fare un parallelismo il protocollo si comporta come un database online, duplicato su più repliche, al quale vengono fatte richieste e, passando per passaggi intermedi quando richiesto, vengono eseguite con un certo parallelismo da tutte le repliche. Gli oggetti possono essere rapportati alle tuple di un database ma possono anche essere usati come tabelle, migliorandone l'utilità e usabilità. Ciò è possibile specificando le informazioni di cui si ha bisogno nelle operazioni, le quali devono essere specificate per consentire la comunicazione. Il sistema permette sostanzialmente due tipi di richieste: richieste di lettura, ovvero tutte le operazioni che ritornano un risultato al client senza modificare lo stato dell'oggetto, e richieste di scrittura nel caso contrario. Le richieste di lettura e scrittura vengono risolte in rispettivamente due e tre scambi di messaggio. Il protocollo realizzato non permette di realizzare nuovi oggetti ma se ciò fosse permesso bisognerebbe applicare una procedura simile a quella di scrittura e infine, informare tutti i client che hanno accesso a quell'oggetto e renderli noto id, tipo e tipologia di operazioni. Il tipo dell'oggetto deve essere previsto dal servizio implementato perché altrimenti non sarà possibile svolgerci operazioni. Il protocollo è un client-server con server replicato, e le comunicazioni sono mediate da degli attori proxy che si occupano di mandare e ricevere messaggi, in quanto sono gli unici a conoscere i riferimenti degli atri attori in gioco. Nella scrittura delle API sono state lasciate delle classi astratte, ciò permette ai meccanismi che si occupano della comunicazione di rimanere invariati lasciando la possibilità ad un programmatore terzo di implementare il suo servizio con il protocollo HQ. Le parti da implementare sono diverse: l'interfaccia client, i servizi del server, gli oggetti e le loro implementazioni delle operazioni. L'interfaccia del client fornisce, a chi implementa il servizio, la possibilità di dialogare in esclusiva con l'attore del client, il quale, svolgendo il compito di proxy, si occupa di sottoporre le richieste del ricevute direttamente a tutte le repliche, riducendo la richiesta del servizio ad una semplice invocazione di funzione. Il metodo dell'interfaccia è stato reso sincrono, il che vuol dire che si può rivelare bloccante qualora la risposta non arrivasse in tempi brevi, ma è fondamentale a garantire che ogni client faccia una sola richiesta alla volta, caratteristica fondamentale per il funzionamento del protocollo. Il server di per sé non richiede nessuna implementazione se non di esplicitare le operazioni che deve eseguire quando gli viene richiesto di leggere e scrivere su un determinato tipo di oggetto.

4.1 Struttura

L'architettura di sistema di HQ è illustrata in figura 4. È caratterizza da un insieme di client $C = c_1,..,c_n$ e un insieme di repliche $R = r_1,..,r_{3f+1}$ dove f rappresenta il numero di fallimenti massimi nella rete, a prescindere dal tipo di fallimento. Come è possibile notare in figura 4, il client chiede al suo proxy di inoltrare le richieste alle repliche e si occupa inoltre di raccogliere il quorum delle risposte, mentre nelle repliche è l'attore del proxy server, che reagendo alla richiesta ricevuta per messaggio, richiama il codice del server per richiedere che l'operazione venga eseguita e poter inoltrare la risposta al client. Il proxy server è anche in grado di comunicare con gli attori BFT, indispensabili per eseguire il processo del consenso che elimina il conflitto nelle operazioni di scrittura, infatti, come mostrato in fig. 5, il BFT è un servizio interno del proxy server che viene attivato automaticamente da quest'ultimo nel momento in cui se ne rileva la necessità.

In figura 5 è rappresentato il diagramma delle classi. Tutte implementano i meccanismi del protocollo mantenendo la logica nascosta, ma è possibile notare la presenza di classi astratte da implementare, ovvero la classe che si occupa di creare l'interfaccia per il client, *ClientService*,

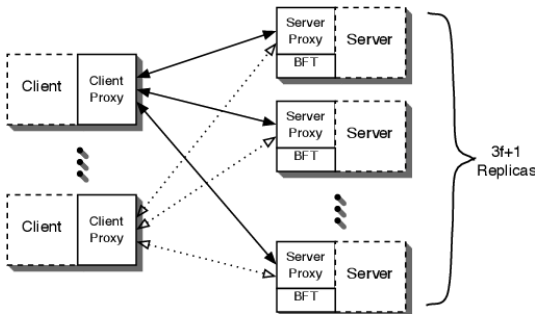

Figure 4: Modello del sistema che utilizza il protocollo HQ.

e quella si occupa della gestione delle operazioni di scrittura e lettura in *ServerService*.

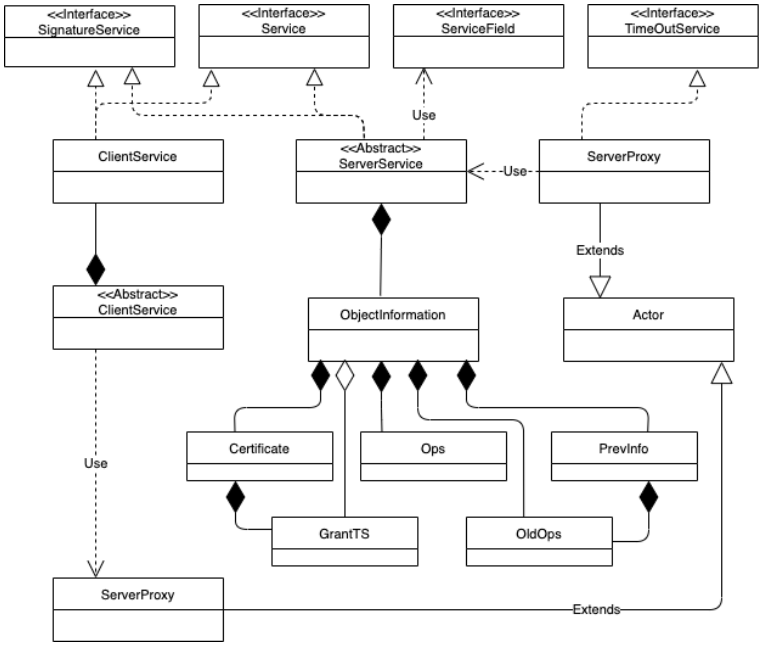

Figure 5: Rappresentazione grafica delle classi e le loro dipendenze.

Come è possibile notare in figura 5 la classe *ObjectInformation* è composta da molte altre classi. Tutte quante costituiscono il pacchetto di informazioni che viene preservato per ogni oggetto nel server, e comprendono:

- currentC: ultimo certificato di scrittura
- grantTS: informazioni sulla concessione attuale del time stamp
- ops: tutte le richieste che sono state ricevute al time stamp attuale e le relative risposte
- oldOps: è una tabella contenente per ogni client il op# più recente il risultato e il certificato di scrittura mandato dal client per autorizzare l'operazione salvata
- vs: è il view stamp
- conflictC: certificato di conflitto (se presente)
- backupC: penultimo certificato di scrittura dell'ultimo client ad averne fatta una

- prev: contiene le informazioni prima contenute in oldOps dell'ultimo client ad averne eseguita una

Altre classi fondamentali sono:

- GrantTS: contiene informazioni sul timestamp cui fa riferimento, ovvero il client che ha ricevuto la concessione, il suo numero operazione, l'oggetto a cui fa riferimento e il ts e il vs per l'istante a cui fa riferimento.
- ViewStamp: una coppia vista, numero di conflitti. La vista identifica il primario nel BFT durante un conflitto, si ottiene da una $f(x)$ con $x =$ il numero di conflitti.
- Certificate: è una semplice lista di grantTS, tanti quanto il quorum ne impone.

4.2 Messaggi

Il protocollo si compone di messaggi e procedure di richiesta predefinite. I messaggi si dividono a seconda di chi può inviarli e a chi sono rivolti, inoltre le comunicazioni tra client e server sono tutti inserite in un contesto di firma del messaggio. Il messaggio che l'interfaccia invia al proxy client è: ⟨REQUIRE, op#, oRef, op⟩ I messaggi che il proxy client può inviare al server sono:

- ⟨READ, cid, oid, op, nonce⟩
- ⟨WRITEBACKREAD, writeC, cid, oid, op, nonce⟩
- ⟨WRITE-1, cid, oid, op#, op⟩
- ⟨WRITE-2, writeC⟩
- ⟨WRITEBACKWRITE, writeC, w1⟩
- ⟨RESOLVE, clientConflictC, w1⟩

Il messaggio che il client invia al proxy al termine della richiesta è: ⟨REPLY, result, op#, statusCode⟩ Mentre i messaggi che il server invia sono:

- ⟨READ-ANS, result, nonce, currentC, rid⟩
- ⟨WRITE-1-OK, grantTS, currentC⟩
- ⟨WRITE-1-REFUSED, grantTS, cid, oid, op#, currentC⟩
- ⟨WRITE-2-ANS, result, currentC, rid⟩
- ⟨START, conflictC, ops, currentC, grantTS⟩
- ⟨UPDATE-REQ, rid, oId, currentC, primaryId⟩
- ⟨UPDATE-REPLICA, hash⟩
- ⟨UPDATE-PRIMARY, oid, objInfo, currentVal, primary⟩
- ⟨GRANT, grantTS⟩

I messaggi che gli attori BFT si scambiano per trovare l'accordo sono:

- ⟨PRE-PREPARE, vs, digest, request⟩
- ⟨PREPARE, vs, digest, replicaId⟩
- ⟨COMMIT, vs, message, replicaId⟩
- ⟨REPLY, vs, ops⟩

Il loro utilizzo verrà spiegato successivamente con l'esposizione delle procedure di richiesta, mentre in 1 sono mostrati i significati delle sigle nei messaggi.

simbolo	significato
cid	client id
op#	numero operazione
op	operazione
rid	id replica
oid	id oggetto
oRef	riferimento a oggetto
objInfo	informazione oggetto
currentVal	valore corrente
hash	hash delle informazioni delle repliche
w1	richiesta write-1
grantTS	concessione del TS
currentC	ultimo certificato di scrittura

Table 1: Legenda simboli dei messaggi.

4.3 Richieste

Tutte le operazioni che verranno implementate in HQ saranno divise in 2 macro categorie, ovvero lettura e scrittura. Ciò restringe di molto le procedure da dover implementare per portare a compimento un'operazione e permette di tenere sotto controllo casi particolari che potrebbero portare ad errori.

4.3.1 Operazioni di lettura

Il client attiva il metodo dell'interfaccia che chiede al proxy di iniziare l'operazione di lettura fornendogli tutte le informazioni di cui necessita. Per farlo utilizza in messaggio ⟨REQUIRE, op#, oRef, op⟩. La richiesta attende la risposta e quando scade rimanda la richiesta finché non viene fornito un risultato. Il proxy client invia a tutte le repliche un messaggio ⟨READ, cid, oid, op, nonce⟩ per segnalare la sua richiesta. Il nonce è usato unicamente per rendere unica la richiesta e renderla più resistente ad attacchi esterni. Il proxy della replica, ricevuta la richiesta, chiede al server di svolgere l'operazione e invia un messaggio ⟨READ-ANS, result, nonce, currentC, rid⟩ con il risultato al proxy client. Quest'ultimo attenderà di comporre un quorum di risposte prima di mandare un messaggio ⟨REPLY, result, op#, statusCode⟩ all'interfaccia che a sua volta lo manipolerà e, a seconda dello status code, restituirà al client un risultato o un errore. Se il proxy client riceve messaggi i cui certificati currentC mostrano time stamp e/o view stamp diversi viene mandata una richiesta ⟨WRITEBACKREAD, writeC, cid, oid, op, nonce⟩ alle repliche lente, le quali con questo messaggio potranno finire le operazioni in sospeso e successivamente rispondere alla richiesta di lettura che nel frattempo sarà tenuta da parte. É possibile osservare graficamente come avviene l'operazione di lettura in figura 6.

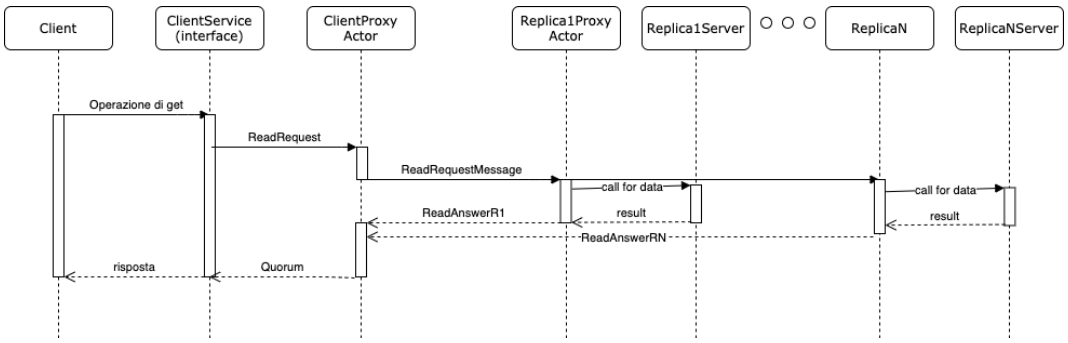

Figure 6: Diagramma che mostra come avviene un'operazione in lettura.

4.3.2 Operazioni di scrittura

In questo paragrafo tratteremo la dinamica di un'operazione di scrittura senza conflitti. Quando un client attiva il metodo dell'interfaccia per la scrittura, l'interfaccia si occupa di mandare la richiesta al proxy client tramite ⟨REQUIRE, op#, oRef, op⟩, il quale conterrà tutte le informazioni necessarie a costruire il messaggio per la richiesta di scrittura. Infine l'interfaccia sarà in attesa di ricevere una risposta e non sarà disponibile ad altre richieste finché l'operazione non è conclusa. Una volta che il proxy client ha ricevuto la richiesta di scrittura, invia alle varie repliche la richiesta dopo aver composto il messaggio ⟨WRITE-1, cid, oid, op#, op⟩ dove sono specificati l'oggetto su cui eseguirla, l'operazione, il numero interno dell'operazione e l'id del client. Il proxy server una volta ricevuto il messaggio valuta vari aspetti. Per prima cosa controlla che l'oggetto esista, se non esiste ritorna una risposta vuota, altrimenti valuta che il client abbia i permessi di scrittura, se non li ha torna una risposta vuota, altrimenti valuta il numero di operazione interno del client, se l'operazione è l'ultima operazione svolta dallo stesso client ritorna un messaggio ⟨WRITE-2-ANS, result, currentC, rid⟩ con la risposta salvata, se è più vecchia la richiesta viene ignorata invece se la richiesta è nuova viene presa in considerazione. Quando una richiesta WRITE-1 è valida, il server controlla che il timestamp non sia stato già concesso per assegnarlo al client della richiesta e viene inviato un messaggio ⟨WRITE-1-OK, grantTS, currentC⟩, altrimenti la richiesta viene respinta con un messaggio ⟨WRITE-1-REFUSED, grantTS, cid, oid, op#, currentC⟩. A questo punto il client attende che si verifichi un quorum di risposte, positive o negative, o arrivino le risposte di tutte le repliche, e ciò porta a 5 diverse risoluzioni:

1. Quorum di WRITE-1-OK: il client inizia la seconda fase.

2. Quorum di WRITE-1-REFUSED con timestamp e viewstamp uguali: vuol dire che il timestamp è già stato concesso ad un altro client, quindi il proxy chiede di "prenotare" il timestamp successivo con una richiesta ⟨WRITEBACKWRITE, writeC, w1⟩. Sia le repliche che il client trattano la richiesta WRITEBACKWRITE come se fosse una normale WRITE-1.

3. Le concessioni hanno un timestamp diverso tra loro: manda una richiesta WRITEBACKWRITE alle repliche che sono rimaste indietro.

4. Quorum di ⟨WRITE-2-ANS, result, currentC, rid⟩: l'operazione è già stata eseguita e il client può rispondere alla richiesta.

5. Quorum di messaggi con lo stesso timestamp e viewstamp ma diversi in tutto il resto: significa che il client sta contendendo le concessioni del timestamp con altri client, di con-

seguenza richiede alle repliche di risolvere il conflitto tramite il messaggio ⟨RESOLVE, clientConflictC, w1⟩. Questo caso verrà spiegato successivamente.

Una volta ottenuto un quorum di WRITE-1-OK si può passare alla fase 2, quindi, il proxy client raccoglie le concessioni e le usa per creare un certificato di scrittura che manda alle repliche tramite messaggio ⟨WRITE-2, writeC⟩ al quale la replica risponderà con un messaggio ⟨WRITE-2-ANS, result, currentC, rid⟩ contenente il risultato dell'operazione svolta. In figura 7 è possibile notare il normale funzionamento di un'operazione di scrittura.

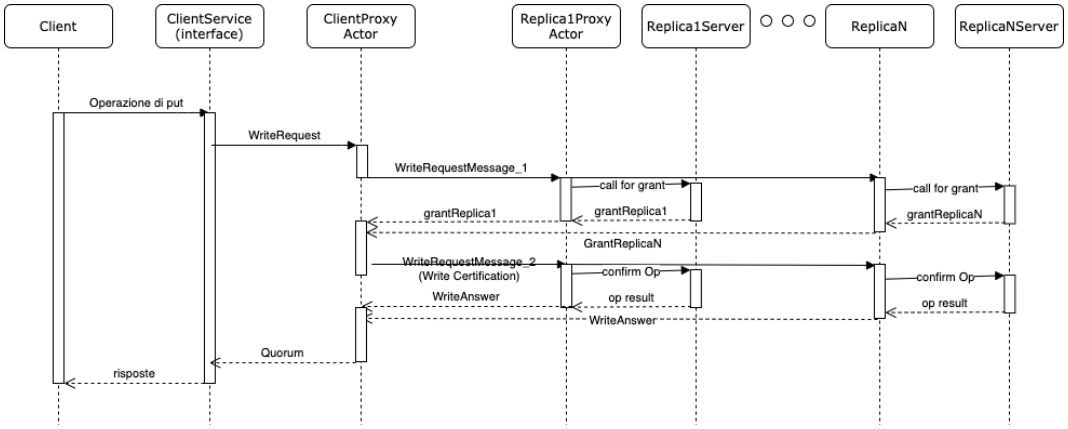

Figure 7: Diagramma che mostra come avviene un'operazione in scrittura.

4.3.3 Operazioni di scrittura con conflitto

Dato che non è possibile fare assunzioni sul tempo e sull'ordinamento dei messaggi, può succedere che 2 o più client si contendano una risorsa. In questo caso nessuno dei client riusce a fare un quorum valido e pertanto inviano un messaggio di ⟨RESOLVE, clientConflictC, w1⟩ alle repliche in modo che il conflitto venga risolto. Ricevuto il messaggio di RESOLVE la replica passa in uno stato di "freeze" in cui non accetta più richieste da parte dei client e richiama il servizio di BFT per risolvere il conflitto inviando un messaggio di ⟨START, conflictC, ops, currentC, grantTS⟩ a tutti gli attori BFT che restituiranno la lista di operazioni che la replica deve eseguire in modo ordinato, pertanto per ogni operazione costruisce la concessione che assegna un determinato timestamp ad una specifica operazione e la manda a tutte le altre repliche tramite messaggio ⟨GRANT, grantTS⟩ e a sua volta attende i grant delle altre repliche per poter costituire un certificato. Quando tutte le operazioni dispongono del certificato possono essere eseguite e si può porre fine al conflitto. Questa fase in termini di tempo è molto dispendiosa, di conseguenza i client non ricevendo risposta continueranno a mandare richieste finché le repliche, avendo risolto il conflitto risponderanno alla richiesta.

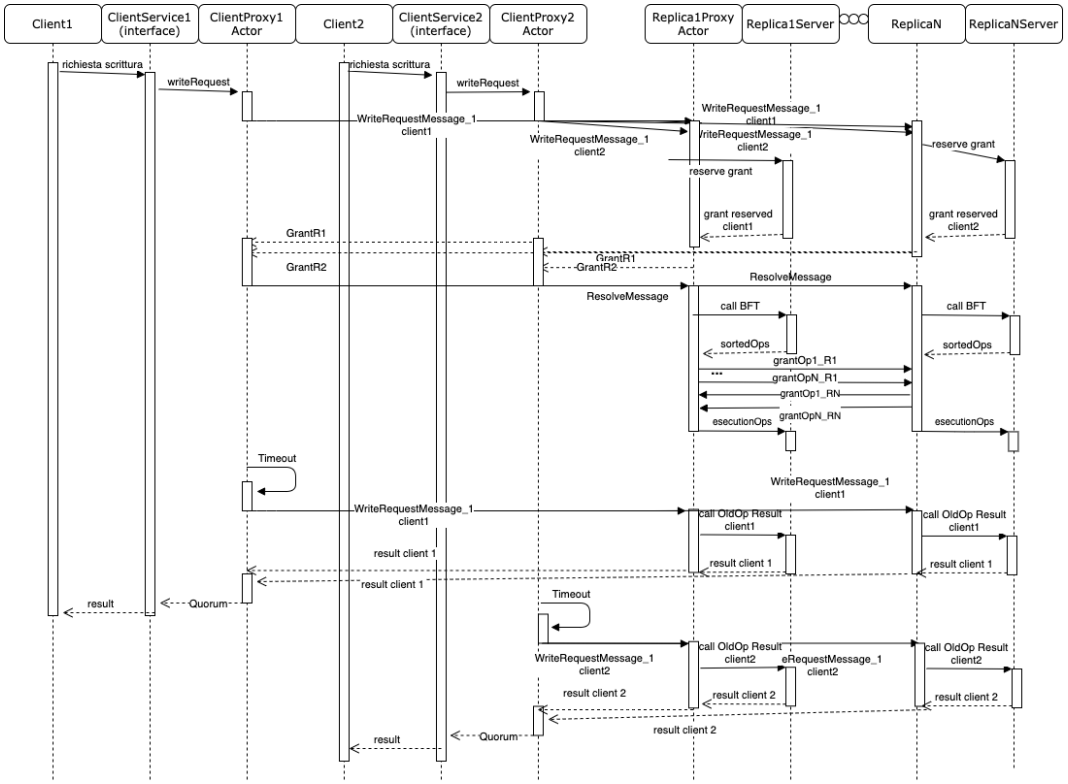

Figure 8: Diagramma che mostra come avviene un'operazione in scrittura con conflitto.

4.3.4 Operazioni di risoluzione conflitto

Questa fase di interazione avviene tra gli attori BFT che dialogano tra loro per trovare accordo sulle operazioni da eseguire, le quali verranno inserite in una lista e ordinate secondo lo stesso criterio da tutte le repliche. L'accordo dei BFT si compone sostanzialmente in fasi scandite dai cambiamenti di stato e quindi di behaviour. Le fasi sono: Request, Pre-prepare, Prepare, Commit, Reply. Nella fase di Request tutte le repliche sono viste come un unico client in un normale servizio con protocollo BFT, di conseguenza tutti gli attori BFT riceveranno tutti i messaggi di START dalle repliche, ma solo il nodo primary raccoglierà le richieste per comporre un quorum. Raggiunto il quorum il primary passa in fase di Pre-prepare e invia il messaggio di ⟨PRE-PREPARE, vs, digest, request⟩ agli altri nodi e passa in fase di Prepare. I nodi di backup passano in fase di Prepare appena ricevuto il messaggio, compongono il messaggio di ⟨PREPARE, vs, digest, replicaId⟩ e lo inviano a tutti i nodi del BFT. Tutti i nodi aspettano di comporre un quorum e una volta ottenuto passano alla fase di Commit. Tutti i nodi compongono il messaggio di ⟨COMMIT, vs, message, replicaId⟩ e lo inviano a tutte le repliche, attendo di comporre un quorum di Commit e passano alla fase finale, Reply, in cui mandano un messaggio di risposta ⟨REPLY, vs, ops⟩ a tutte le repliche. Le repliche una volta ricevuti i messaggi ordineranno ops secondo la stessa logica e si prepareranno a risolvere le richieste. In figura 9 è possibile osservare un normale scambio di messaggi tra attori BFT.

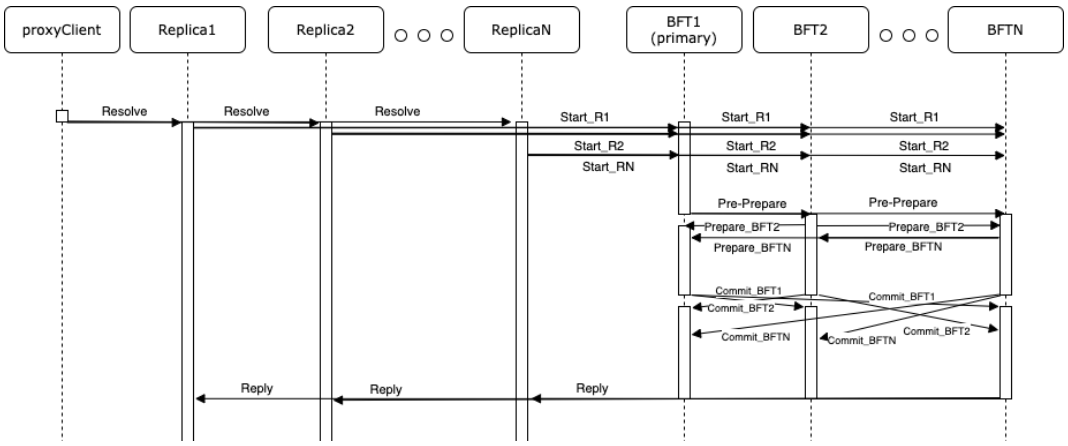

Figure 9: Diagramma che mostra come avviene un'operazione in scrittura con conflitto.

4.3.5 Operazioni di update

La replica si accorge che deve essere aggiornata solo dopo aver ricevuto un messaggio di WRITE-2 con certificato valido con un timestamp superiore al suo timestamp attuale più 1. In questo caso la replica chiede alle altre di mandargli tutti i dati mancanti dal suo ultimo timestamp valido componendo e inviando un messaggio ⟨UPDATE-REQ, rid, oId, currentC, primaryId⟩, nel quale specifica il suo id per essere ritrovata, l'oggetto sul quale le sue informazioni sono indietro, il timestamp al quale si è fermato e infine specifica l'id della replica che dovrà coordinare le operazioni di update. Infatti, nella procedura di update, si deve sì raggiungere il quorum per rendere l'update valido ma ad inviare i dati è solo la replica primaria, rispondendo con un messaggio ⟨UPDATE-PRIMARY, oid, objInfo, currentVal, primary⟩, tutte le altre risponderanno inviando solo l'hash degli stessi dati tramite ⟨UPDATE-REPLICA, hash⟩. La replica in ritardo comporrà l'hash dei dati ricevuti dal primary e lo confronterà con quello ricevuto dalle altre repliche cercando di comporre il quorum per validarli. Il protocollo, come possibile notare dall'immagine 10, è molto veloce e serve per non lasciare indietro la replica nelle operazioni successive.

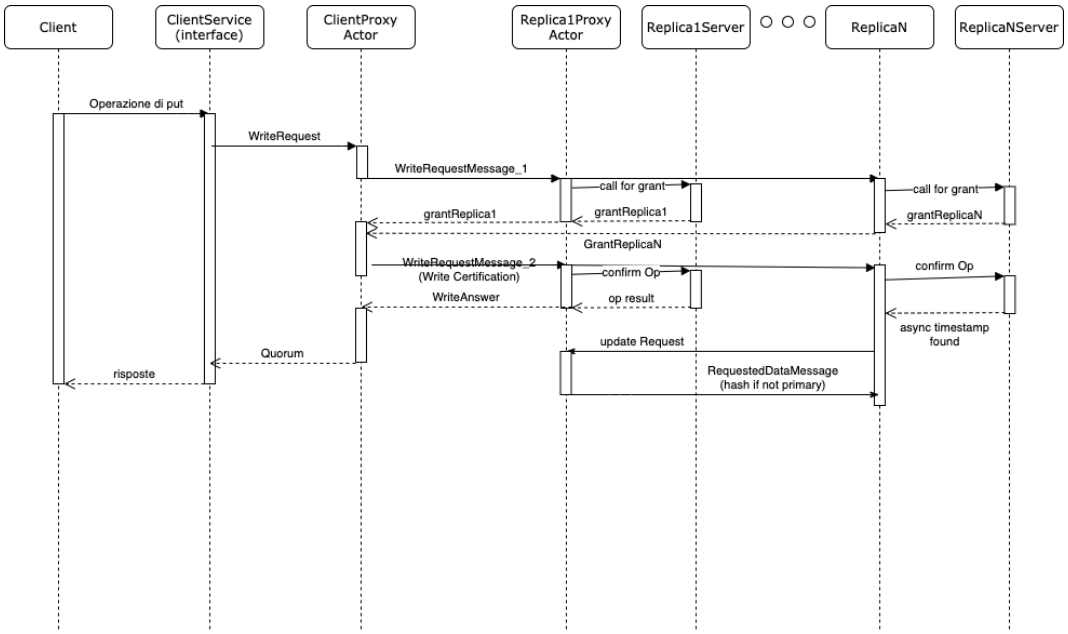

Figure 10: Diagramma che mostra come avviene un'operazione in scrittura con update.

4.4 Behaviour

Vista la natura concorrente e distribuita dell'applicazione è stato consono utilizzare il modello ad attori per organizzare l'intero codice. Una buona prassi di design quando si ha a che fare con gli attori è quella di definire l'insieme di stati e di eventi che ne provocano il passaggio da uno stato all'altro.

Nella nostra applicazione abbiamo utilizzato tre differenti attori:

- Client Actor;
- Proxy Server;
- Pbft Actor;

di seguito verranno descritti in modo dettagliato.

4.4.1 Client Actor

Il Client Actor, come preannuncia lo stesso nome, è l'attore che si occupa di effettuare le chiamate asincrone alle repliche e ad interagire con esse. Lo scopo del client actor non è solo quello di eseguire le chiamate asincrone ma, ha un ruolo attivo nel funzionamento del protocollo, infatti, qualora dovesse accorgersi di eventuali incongruenze di risposte da parte delle repliche richiede una procedura di risoluzione di conflitto. Quindi appare evidente, che tale attore è in grado di capire lo stato delle repliche sulla base delle risposte ricevute e ha una visione complessiva sull'intero sistema.

Come già accennato in precedenza, le tipologie di richieste si suddividono in due categorie quelle di scrittura e quelle di lettura, perciò adesso analizzeremo come vengono gestite tali richieste lato cliente. Nella progettazione dell'attore cliente sono stati previsti tre differenti stati (vedi fig. 11). La scelta di tre differenti stati non è stata casuale, ma dettata dalle modalità di funzionamento del protocollo, infatti, uno stato è impiegato per la gestione della richiesta di lettura, mentre gli atri due per gestire la richiesta di scrittura che da protocollo richiede due round.

Quando l'attore riceve il messaggio di $\langle RequireWriteMsg \rangle$ inizia la fase di lettura, commuta il proprio stato in "ReadStatus", invia la richiesta alle repliche e attende un quorum di messaggi validi. Dopo aver ricevuto un quorum (2f + 1) di risposte valide ritorna al cliente il risultato della richiesta. In caso non dovesse ricevere una risposta in un tempo utile, il client actor ripeterà nuovamente la richiesta. Si nota che, non accadrà mai la situazione di "buisy-wait" da parte delle repliche, poiché uno dei requisiti del sistema è quello che le richieste prima o poi saranno eseguite e riceveranno una risposta.

La richiesta di scrittura, invece, inizia in occasione della ricezione del messaggio di $\langle RequireReadMsg \rangle$. La fase di scrittura è più articolata rispetto a quella di lettura poiché si basa su due round. La gestione delle due fasi è stata modellata attraverso due stati distinti, ciò agevola la capacità di reazione ai distinti messaggi. Detto ciò, subito dopo aver ricevuto la richiesta di scrittura e dopo averla inoltrata alle repliche, l'attore commuta il proprio stato in "Write1Status", in cui attende la risposte da parte delle repliche. Il meccanismo che permette al client di capire lo stato delle repliche è quello di osservare il modo con cui esse rispondono e i possibili messaggi ricevuti sono:

- $\langle write - 1 - ok, grantTS, current \rangle$: se la replica concede il permesso a questo cliente al timestamp successivo;
- $\langle write - 1 - refused, grantTS, cid, oid, op\#, currentC \rangle$: se la replica ha dato la concessione al timestamp successivo ad un altro cliente;

- $\langle write - 2 - ans, result, currentC, rid \rangle$: se la richiesta del cliente è già stata eseguita;

In condizioni normali, cioè in assenza di conflitto, l'attore in "Write-1-Status" colleziona un quorum di messaggi di tipo $\langle Write - 1 - ok \rangle$ validi e utilizza il grant in essi contenuti per formare un certificato di scrittura. Un quorum di messaggi di tipo $\langle Write - 1 - ok \rangle$ è valido se il client riceve la stessa concessione da un quorum di repliche, in altre parole, un numero pari a 2f + 1 repliche inviano una concessione a scrivere su uno specifico oggetto, ad uno specifico timestamp da parte di uno specifico cliente avente uno certa sequenza numerica. In questo modo, si ha la garanzia che la maggioranza delle repliche sono consistenti. Una volta ottenuto il quorum di certificati validi, l'attore cliente può formare un certificato di scrittura, richiedere la scrittura per quel certificato formato e infine commutare nello stato di "Write-2-Status". La procedura di scrittura termina quando il client riceve un quorum di messaggi di tipo $\langle Write - 2 - Ans, result, currentC, replicaID \rangle$ e soddisfatta tale condizione, restituisce il risultato al richiedente.

A questo punto analizziamo come l'attore processa i messaggi ricevuti in "Write-1-Status":

1. se riceve un quorum di messaggi di tipo $\langle Write - 1 - ok \rangle$ con lo stesso viewstamp e timestamp forma un certificato di scrittura;

2. se riceve un quorum di $\langle Write - 1 - refused \rangle$ con lo stesso viewstamp e lo stesso timestamp significa che un altro cliente ha ricevuto la concessione . Per facilitare la progressione in caso di clienti lenti o falliti viene trasmesso il messaggio di $\langle WriteBackWrite, writeC, w1 \rangle$ alle repliche.

3. se riceve delle risposte di Write-1-ok aventi grantTS.ts o grantTS.vs differente significa che delle repliche sono indietro con alcune operazioni di scrittura. Il client manderà un messaggio di risoluzione $\langle writeBackWrite, writeC, w1 \rangle$ solo alle repliche lente, dove writeC sarà pari al certificato più recente fra quelli ricevuti;

4. se riceve un quorum di messaggi contenenti delle concessioni con stesso viewstamp e timestamp ma per il resto differenti, lui invia il messaggio di $\langle Resolve \rangle$ alle repliche. Il conflitto emerge perché repliche diverse hanno concesso allo stesso timestamp e viewstamp operazioni diverse a clienti diversi sullo stesso oggetto.

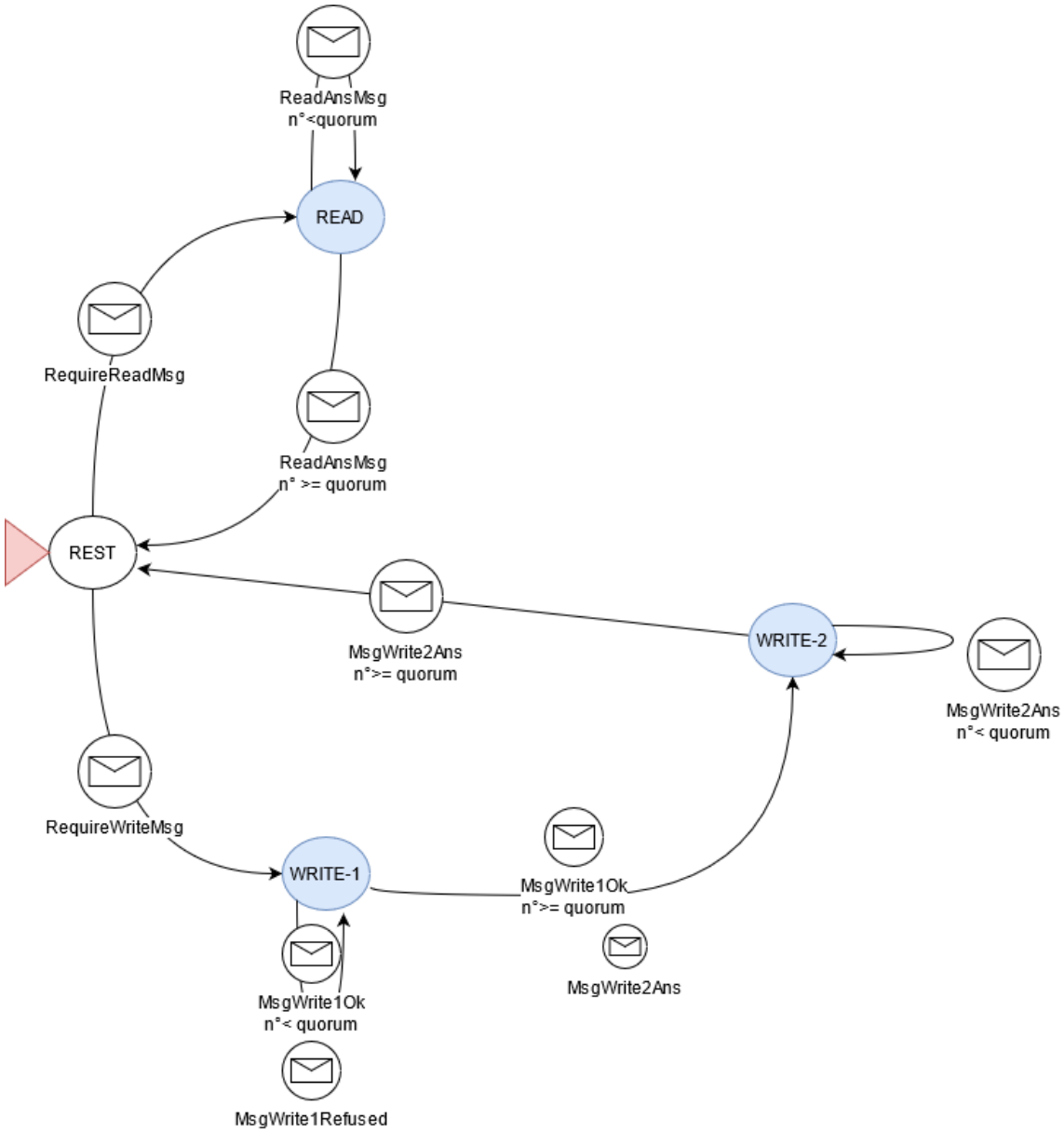

Figure 11: Diagramma degli stati dell'attore cliente.

4.4.2 Proxy Server

L'attore Proxy Server è l'attore preposto a elaborare le richieste asincrone da parte dei clienti, nella fattispecie, si occupa di eseguire le operazioni di lettura e scrittura. Il numero di attori di tipo proxy server è pari al numero di repliche presenti nel sistema, poiché esso si occupa di gestire le richieste lato server. Essendo anche il proxy server modellato come un attore, gli stati previsti per definire il suo comportamento, in accordo con il protocollo HQ, sono mostrati nella figura 13.

Come si osserva dal diagramma degli stati, le richieste di lettura e scrittura non provocano un cambio di stato poiché l'attore deve essere sempre reattivo alle richieste, mentre, gli unici eventi che ne determinano il passaggio di stato sono:

- la risoluzione del conflitto;
- l'update.

Analizziamo in modo più dettagliato quali sono gli eventi che ne determinano il passaggio di stato e in cosa consistono i singoli stati.

Un situazione alquanto ricorrente in un sistema distribuito è quella di conflitto e nella sua accezione più generale si intende, quella circostanza in cui n contendenti sono in contesa sulla stessa risorsa. Nel caso dell'*Hybrid Quorum* si fa riferimento allo scenario in cui un quorum di repliche hanno concesso il permesso di scrittura a diversi clienti nello stesso timestamp. La procedura di risoluzione di conflitto nasce lato cliente, qualora dovesse accorgersi dell'incongruenza dei messaggi ricevuti. A tal proposito, il client si prodiga a richiedere la risoluzione del conflitto trasmettendo il messaggio di $\langle Resolve \rangle$ alle repliche. L'attore proxy, alla ricezione del messaggio, attiva la procedura di risoluzione di conflitto invocando il servizio *Practical Byzantine Fault Tolerance* e attende un sua risposta nello stato di *"Waiting For Resolution"*. Quando riceve dal servizio *pBFT* messaggi di tipo $\langle ReplyMsg \rangle$ in un numero pari a $n \geq \frac{N-1}{3} + 1$ (dove N è il numero di repliche) elabora le informazioni ritornate dal servizio al fine da risolvere il conflitto e modificare lo stato delle variabili interne. Affinché la chiamata al servizio abbia esito positivo è necessario che un quorum di repliche ne richiedano la risoluzione, dove esse propagano la conoscenza locale circa le meta informazioni su uno specifico oggetto. Il servizio risponde a tutte le repliche con il messaggio di tipo $\langle ReplyMsg \rangle$ e esse utilizzano questo messaggio allo scopo di:

1. risolvere il conflitto, ovvero tutte le repliche concederanno allo stesso timestamp la stessa operazione, sullo stesso oggetto da parte di uno stesso cliente;
2. creare uno stato consistente con le altre repliche.

Analizziamo in modo più dettagliato quest'ultimo punto in quanto è un passaggio piuttosto delicato del protocollo. Creare uno stato consistente si intende creare una certa uniformità fra i diversi stati delle repliche, tuttavia, essendo il sistema asincrono non si hanno garanzie che nel tempo le repliche evolvono in modo simultaneo, perciò questa procedura mira a garantire consistenza tra esse. Tecnicamente, la procedura si basa sulla seguente osservazione: se i singoli proxy server "girano" su macchine deterministiche allora a partire da uno stesso input, che nel nostro caso è il messaggio di tipo $\langle ReplyMsg \rangle$, e date un insieme finito di istruzioni l'output prodotto sarà uguale su tutte le macchine. Quindi, le repliche dopo aver eseguito quelle istruzioni su quello specifico input convergeranno verso uno stato consistente. Si nota che nel messaggio inviato al servizio pBft, le repliche propagano la propria conoscenza locale circa l'oggetto che ha prodotto il conflitto e richiedono ad esso di risolverlo. Si evince che quando ciascuna replica riceve la risposta da parte del pBft ha informazioni anche sullo stato delle altre repliche, quindi può stabilire se il proprio stato è in:

1. anticipo, se il certificato locale è maggiore a quello emerso dal messaggio $\langle ReplyMsg \rangle$;
2. ritardo, se il certificato locale è minore a quello emerso dal messaggio $\langle ReplyMsg \rangle$;
3. accordo, se il certificato locale è uguale a quello emerso dal messaggio $\langle ReplyMsg \rangle$.

In base alle precedenti condizioni, la replica decide cosa fare per poter uniformare il proprio stato alle altre. Dopo aver aggiornato la conoscenza locale, prima di terminare la risoluzione del conflitto deve eseguire le restanti richieste ai timestamp successivi che emergono dal messaggio $\langle ReplyMsg \rangle$. L'ordine con la quale le richieste vengono eseguite e in ordine crescente di client Id. Una volta redatta la lista di richieste usando il criterio precedente, la replica si pone nello stato di "Waiting for Certificate" in attesa di formare i certificati per ogni richiesta. Quando riceve le concessioni da un quorum di repliche per ogni richiesta, può finalmente formare i certificati ed eseguire le richieste associate, rispettando l'ordine citato. Una volta conclusa questa fare, la replica rimuove lo stato di conflitto e ritorna nello stato iniziale.

Lo stato di updete viene invocato qualora un replica è in ritardo rispetto alle altre, e tale fase può avvenire sia se la replica sta risolvendo un conflitto e sia in condizioni normali. L'idea di base dell'update risiede, nella richiesta da parte della replica "ritardataria" alle restanti delle informazioni mancanti ai timestamp successivi. Una replica si accorge di essere in ritardo, qualora dovesse ricevere un certificato avente un timestamp successivo rispetto a quello custodito localmente. In tale circostanza, la replica fa una richiesta di update inviando il messaggio $\langle UpdateReqMsg \rangle$ alle restanti repliche e attende nello stato di "Update Status" una risposta. Al fine di evitare di saturare la banda e minimizzare la quantità di informazioni trasmesse sulla rete, è stato fatto in modo che solo il primario trasmette le informazioni inviando il messaggio di $\langle UpdatePRMsg \rangle$ mentre le restanti mandano un hash delle informazioni richieste inviando $\langle UpdateRMsg \rangle$. Il richiedente si preoccuperà una volta ricevuti i messaggi di ricalcolare l'hash del messaggio $\langle UpdatePRMsg \rangle$ e confrontarlo con quello $\langle UpdateRMsg \rangle$. Nel caso gli hash combaciano, utilizza le informazioni ricevute dal primario per aggiornare il proprio stato e poi ritornare nello stato iniziale, viceversa, se gli hash non combaciano verrà eseguita nuovamente la richiesta di update cambiando il primario. Si sottolinea che, la fase di update può essere eseguita anche in occasione di conflitto, infatti, il protocollo prevede che una replica prima esegue l'update e poi continua normalmente nella risoluzione del conflitto.

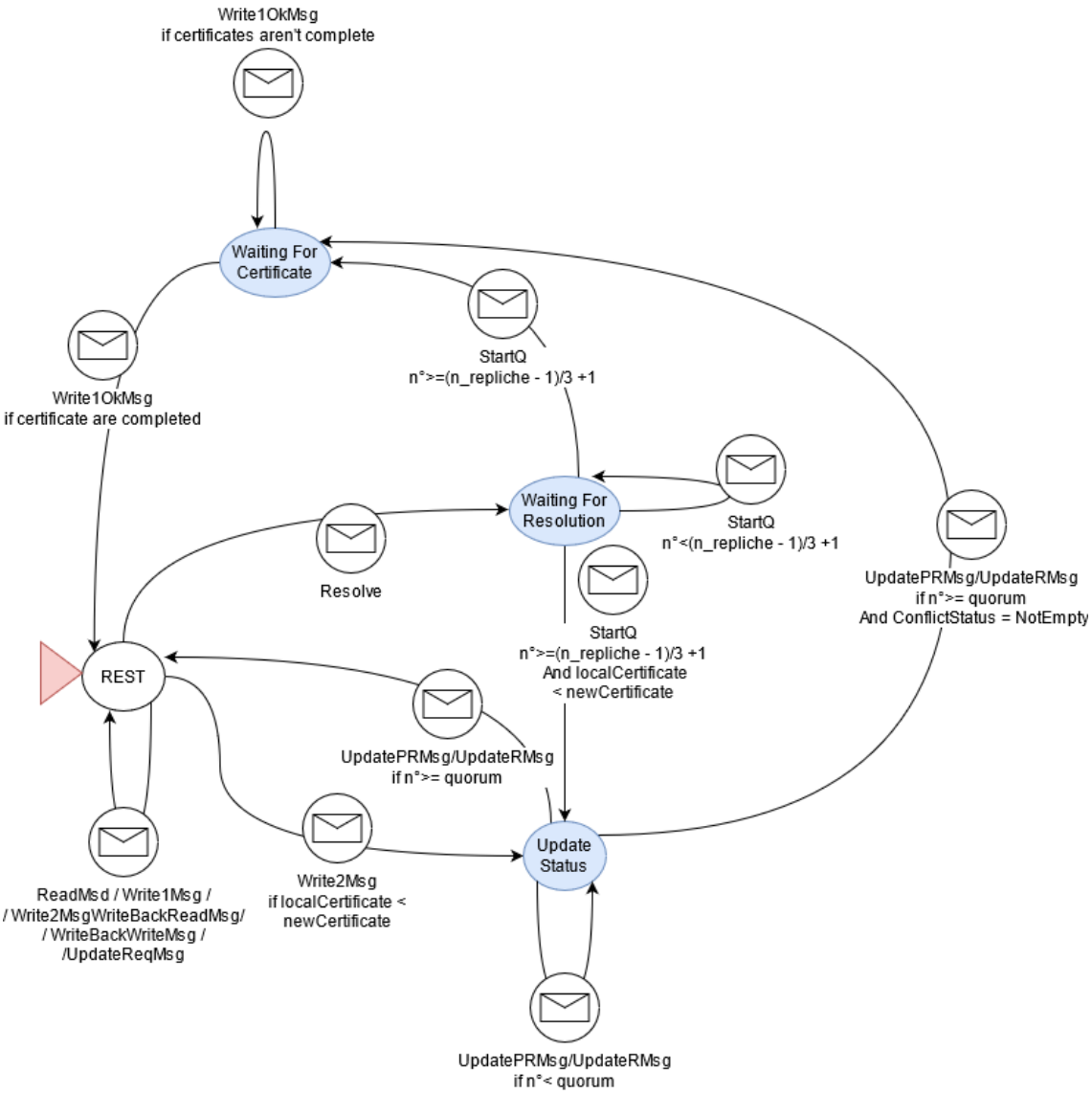

Figure 12: Diagramma degli stati dell'attore Proxy Server.

4.4.3 Pbft Actor

L'attore Pbft è utilizzato per modellare il protocollo del Practical Byzantine Fault Tolerance. Come già preannunciato, il protocollo si basa su tre fasi principali che ne permettono il corretto funzionamento, la figura 13 mostra l'insieme degli stati dell'attore durante l'evoluzione del protocollo. Vediamo il funzionamento concreto del protocollo e quali sono gli adattamenti fatti al sistema in esame.

Quando un primario riceve una quorum di messaggi di tipo $\langle StartMsg \rangle$, lui crea un'operazione BFT a risolvere il conflitto e l'argomento di questa operazione è l'insieme di tali messaggi: chiamati *startQ*. Il primario avvia l'operazione atta a risolvere il conflitto, in altre parole mira a uniformare le concessioni presenti nello startQ. L'operazione che viene richiesta al pBft è detta "OperationOrderingRequest" la quale richiede di stabilire quale richiesta deve essere eseguita al timestamp successivo. Si puntualizza che, tale protocollo è stato pensato in modo da generalizzare le richieste del pBft, infatti, è stata definitia una classe astratta denominata "Request" che modella il concetto di richiesta, cosicché, una classe che la estende deve semplicemente definire l'operazione su uno specifico tipo di dato in input e con uno specifico tipo di ritorno.

Una volta accettata la richiesta dal cliente, il primario propone una sequenza numerica per quella richiesta, può iniziare il protocollo a tre fasi descritto nel capitolo 2.2.2.

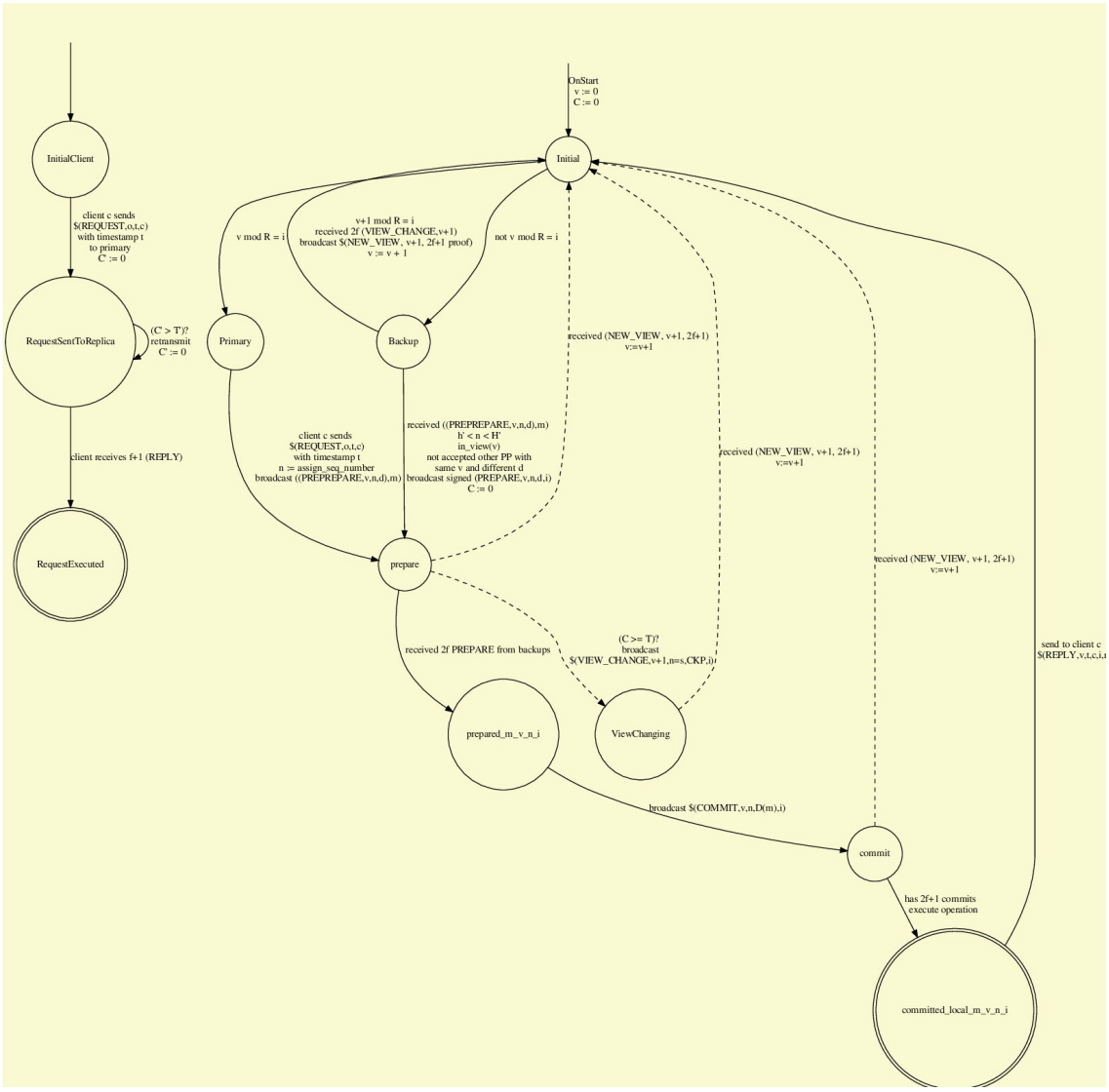

Figure 13: Diagramma degli stati dell'attore utilizzato per la realizzazione del servizio pBft.

5 Test

Data la modularità del sistema, normale conseguenza dell'uso di tre attori principali per eseguire le diverse fasi richieste dal protocollo, la fase di test è volta a testare la correttezza dei *behaviour*. Essendo "Akka" la libreria utilizzata per implementare attori, essa mette anche a disposizione **"Akka TestKit"** libreria indispensabile per il testing degli stessi. Essendo gli attori dei soggetti reattivi hanno bisogno di stimoli per svolgere le loro attività, quindi, ciò che viene fatto nei test è porre gli attori in situazioni controllate, sia normali che estreme, nella quale l'invio dei messaggi è manipolato ad-hoc, e controllare che i comportamenti e le risposte degli attori siano quelli attesi. Ogni parte del sistema è stata testata con cura ed è stato redatto un report dei test che specifica gli aspetti testati e i cambiamenti di stato che ci si aspettavano. I cambiamenti di stato non possono essere controllati in modo esplicito, in quanto interni agli attori, ma i messaggi che l'attore ha accettato, rifiutato e inviato, essendo univoci di determinati stati (come già discusso nella sezione Behaviour) forniscono indicazioni a riguardo. Per svolgere i test è stato implementato un servizio base con una operazione di get, una di set e una di take. E' stata testata anche l'interfaccia che permette il dialogo tra il codice del client e l'attore del proxy client. I test di interfaccia sono riportati nella tabella che segue (2).

Descrizione	Stato	Risultato atteso
richiesta di get ok	/	ricezione risultato
richiesta di put ok	/	ricezione risultato
richiesta di take ok	/	ricezione valore di default
risposta lenta	/	ricezione risultato
risposta errore interno	/	InternalErrorException
risposta con number operation non sincronizzato	/	ricezione risultato

Table 2: Tabella dei test di interfaccia.

I test del client testano sia la capacità di rispondere all'interfaccia che la capacità di inviare richieste ai server e gestirne in modo efficace le risposte. I cambiamenti di stato sono dimostrati dalla capacità dell'attore di reagire nei modi previsti ai messaggi inviati ad-hoc per testarne i comportamenti. I test prevedono anche il controllo delle funzionalità di time out, che il protocollo usa come una forma di sicurezza nei confronti di un *crash di rete*, prevedendo, come d'assunzione, che prima o poi il client raggiunga la replica che non ha risposto o riesca a comporre il quorum.

Descrizione	Stato	Risultato atteso
richiesta di lettura	iniziale ⇒ read	Ricezione 4x ReadMsg
lettura dati	iniziale ⇒ read ⇒ iniziale	Ricezione Reply
richiesta di scrittura	iniziale ⇒ write-1	Ricezione 4x Write1Msg
passaggio dell'attore in fase 2 di scrittura	iniziale ⇒ write-1 ⇒ write-2	Ricezione 4x Write2Msg
passaggio dell'attore in fase 2 con un writebackwrite	iniziale ⇒ write-1 rightarroe write-2	Ricezione 4x Write2Msg
quorum di messaggi rifiutati	iniziale ⇒ write-1 ⇒ write-1 ⇒ write-2 ⇒ iniziale	Ricezione 4x Write2Msg
identificazione del conflitto	iniziale ⇒ write-1 ⇒ write-2 ⇒ iniziale	Ricezione 4x ResolveMsg
client lento attende permessi per fase2 ma arriva la risposta write2ans	iniziale ⇒ write-1 ⇒ write-2 ⇒ iniziale	Ricezione 4x Write2Msg
Scrittura completa	iniziale ⇒ write-1 ⇒ write-2 ⇒ iniziale	Ricezione Reply
scadenza timeout in read	read ⇒ read	Ricezione dei messaggi dalle repliche fallite
scadenza timeout in write fase 1	w1 ⇒w1	Ricezione dei messaggi dalle repliche fallite
scadenza timeout in write fase 2	w2 ⇒ w2	Ricezione dei messaggi dalle repliche fallite

Table 3: Tabella dei test del proxy client.

Essendo il server dotato di vari *behavoiur* che implementano meccanismi per risolverne i fallimenti, ci sono molti test che vanno a ricreare situazioni normali e particolari nei tre *behavoiur*. Nella tabella 4 sono elencati i test fatti sul proxy server in caso di normale funzionamento delle operazioni.

Descrizione	Stato	Risultato atteso
Richiesta di lettura	Normal	ReadAnsMsg
2 letture diverse su stesso oggetto	Normal	ReadAnsMsg
2 letture diverse su oggetti diversi	Normal	ReadAnsMsg
lettura e scrittura su oggetti diversi	Normal	ReadAnsMsg + Write2AnsMsg
lettura e scrittura su stesso oggetto	Normal	ReadAnsMsg + Write2AnsMsg
scrittura e lettura su stesso oggetto -WRITEBACKREAD-	Normal	Write2AnsMsg + ReadAnsMsg
scrittura su oggetto	Normal	Write2AnsMsg
2 scritture su oggetti diversi	Normal	2x Write2AnsMsg
2 scritture su stesso oggetto -WRITEBACKWRITE-	Normal	Write2AnsMsg
scrittura su oggetto insesitente	Normal	Attualmente la richiesta è scartata
2 scritture in competizione sulla replica (un client richiede di scrivere ma la concessione da il permesso all'altro client)	Normal	RefusedMgs

Table 4: Tabella dei test del proxy server in condizioni di comportamento normale.

Nella tabella 5 sono elencati i test fatti sul proxy server in caso di la replica si accorga di essere indietro con le operazioni e debba recuperarle. Vengono anche verificati gli aggiornamenti in oldOps, ovvero le operazioni già eseguite.

Descrizione	Stato	Risultato atteso
Durante fase di update vengono ricevuti prima il primary e dopo gli hash	Normal ⇒ Update ⇒ Normal	Completamento di update di tutti i dati
Durante fase di update vengono ricevuti prima gli hash e dopo il primary	Normal ⇒ Update ⇒ Normal	Completamento di update di tutti i dati
Durante fase di update vengono ricevuti messaggi hash prima e dopo aver ricevuto il primary	Normal ⇒ Update ⇒ Normal	Completamento di update di tutti i dati
Il primary viene cambiato se quello eletto è la replica stessa	Normal ⇒ Update ⇒ Normal	Completamento di update di tutti i dati
Il primary viene cambiato se update fallisce (timeout)	Normal ⇒ Update ⇒ Normal	Completamento di update di tutti i dati
Il primary viene cambiato se impiega troppo tempo a completare la procedura di update	Normal ⇒ Update ⇒ Normal	Completamento di update di tutti i dati
La replica riceve una richiesta di update ed è il primary scelto	Normal ⇒ Update ⇒ Normal	Invio messaggio di tipo UPDATE-Primary
La replica riceve una richiesta di update e non è il primary scelto	Normal ⇒ Update ⇒ Normal	Invio messaggio di tipo UPDATE-Replica

Table 5: Tabella dei test del proxy server in condizioni di comportamento di update dei dati.

Nella tabella 6 sono elencati i test fatti per verificare il comportamento sia della sola replica, che del gruppo di repliche e il servizio di BFT per la risoluzione dei conflitti.

Descrizione	Stato	Risultato atteso
La replica chiama il servizio BFT	Normal ⇒ ContentResolution	4x StartMsg
La replica completa il processo di risoluzione del conflitto	Normal ⇒ ContentResolution ⇒ Normal	Operazioni in conflitto completate
La replica completa il processo di risoluzione del conflitto e conserva anche le oldops recuperate	Normal ⇒ ContentResolution ⇒ Normal	Operazioni in conflitto completate e oldOps recuperate
Le repliche e il servizio BFT completano la risoluzione del conflitto	Normal ⇒ ContentResolution ⇒ Normal	Operazioni in conflitto completate
Le repliche e il servizio BFT completano la risoluzione del conflitto con una replica in ritardo di una operazione	Normal ⇒ ContentResolution ⇒ Normal	Operazioni in conflitto completate e problemi risolti
Le repliche e il servizio BFT completano la risoluzione del conflitto con una replica in ritardo di più operazioni	Normal ⇒ ContentResolution ⇒ Update ⇒ Normal	Operazioni in conflitto completate e problemi risolti
Le repliche e il servizio BFT completano la risoluzione del conflitto con una replica in avanti di una operazione	Normal ⇒ ContentResolution ⇒ Update ⇒ Normal	Operazioni in conflitto completate

Table 6: Tabella dei test del proxy server in condizioni di comportamento di contention resolution dei dati.

Infine, la tabella 7 mostra i test fatti per verificare il funzionamento dell'attore pBFT e del procedimento con cui riescono a raggiungere l'accordo. Ciò assicura la robustezza del protocollo pBFT in caso venga chiamato per la risoluzione di un conflitto.

Descrizione	Stato	Risultato atteso
Completamento fase Pre-Prepare	Init ⇒ Pre-Prepare	Messaggio Pre-Prepare
Inizia fase di Prepare	Init ⇒ Prepare	Messaggio Prepare
Completa Prepare e inizia Commit	Init ⇒ Prepare ⇒ Commit	Messaggio Commit
Completa Commit e risponde Reply	Init ⇒ Prepare ⇒ Commit ⇒ Init	Messaggio Reply
BFT funzionamento completo se chiamato da un proxy server	Init ⇒ Prepare ⇒ Commit ⇒ Init	Quorum con lista di operazioni da eseguire (accordo trovato)
Servizio BFT chiamato più volte	Init ⇒ Prepare ⇒ Commit ⇒ Init	Risoluzione di tutti i conflitti
Cambio di primary	Init ⇒ Prepare ⇒ Commit ⇒ Init	Risoluzione conflitto con cambio di primary

Table 7: Tabella dei test del servizio pBFT che il server utilizza per risolvere i conflitti.

I test presenti nelle tabelle sono tutti risultati positivi, ed è possibile verificarlo. Infatti dopo aver eseguito le operazioni descritte per il deployment è possibile eseguire, sempre su sbt, il comando:

```
sbt test
```

6 Istruzioni per il Deployment

La corretta compilazione del progetto richiede l'installazione di scala 2.12.1 o versioni superiori.

Per permettere un facile e agile *deployment* si è fatto uso di **sbt**. Per lanciare l'applicazione seguire i seguenti passi:

1. clonare il *repository* da GitLab (link qui);
2. aprire il terminale e posizionarsi nella *root* principale del progetto;
3. fare la *build* del progetto lanciando il seguente comando:

   ```
   sbt compile
   ```

4. lanciare il programma usando il seguente comando:

   ```
   sbt run
   ```

Dopo l'esecuzione del comando run apparirà la view con la quale si potrà interagire con il programma.

7 Demo

L'implementazione della versione demo è una semplice dimostrazione del protocollo. Sono stati aggiunti metodi ad-hoc per creare situazioni che altrimenti non sarebbe possibile, o al quanto poco probabile, vedere, come per esempio un update o una fase di conflitto. Ciò succede perché anche se viene simulato un ambiente distribuito e non v'è certezza sull'ordine dei messaggi, essendo tutto in locale le comunicazioni sono troppo veloci per poter provocare un conflitto o una perdita di messaggio. Il servizio, come prevede l'utilizzo delle nostre API di HQ, si occupa di implementare l'interfaccia *StockOrderClient* per permettere al client di fare richieste, lo *StockServiceServer* che implementa l'esecuzione delle operazioni sugli oggetti nella replica e, infine, il *ProductField* che definisce l'oggetto e le ripercussioni che ogni azione ha su di esso.

7.1 GUI: Distributed Warehouse Manager

Per mostrare il funzionamento della demo e quindi del protocollo è stata realizzata una semplice interfaccia che permette di interagire e mostrare eventi e risultati che coinvolgono il funzionamento del sistema. Nella demo proposta, la classe "DashBoardApp" funge da main che ha il compito di mostrare la gui e avviare un thread a parte nel quale è istanziato "l'ActorSystem" su cui avviare il sistema di repliche e clients.

Come mostrato nella figura 14, la demo si pone nelle condizioni minime richieste dal protocollo, ovvero, un sistema fatto da quattro repliche e due clienti. Un sistema così fatto è in grado di tollerare $f = 1$ fallimenti e generare contese fra due clienti, la ragione di questa scelta è legata ad una questione di natura dimostrativa e didattica. Inoltre nella view, sono stati introdotti due campi opzionali "Send Only To" e "Send Late To" per simulare la perdita di messaggi e l'assenza di garanzie sull'ordine dei messaggi di un vero contesto distribuito, ciò è utile per forzare il sistema verso gli scenari di conflitto e update oggetto di studio del protocollo.

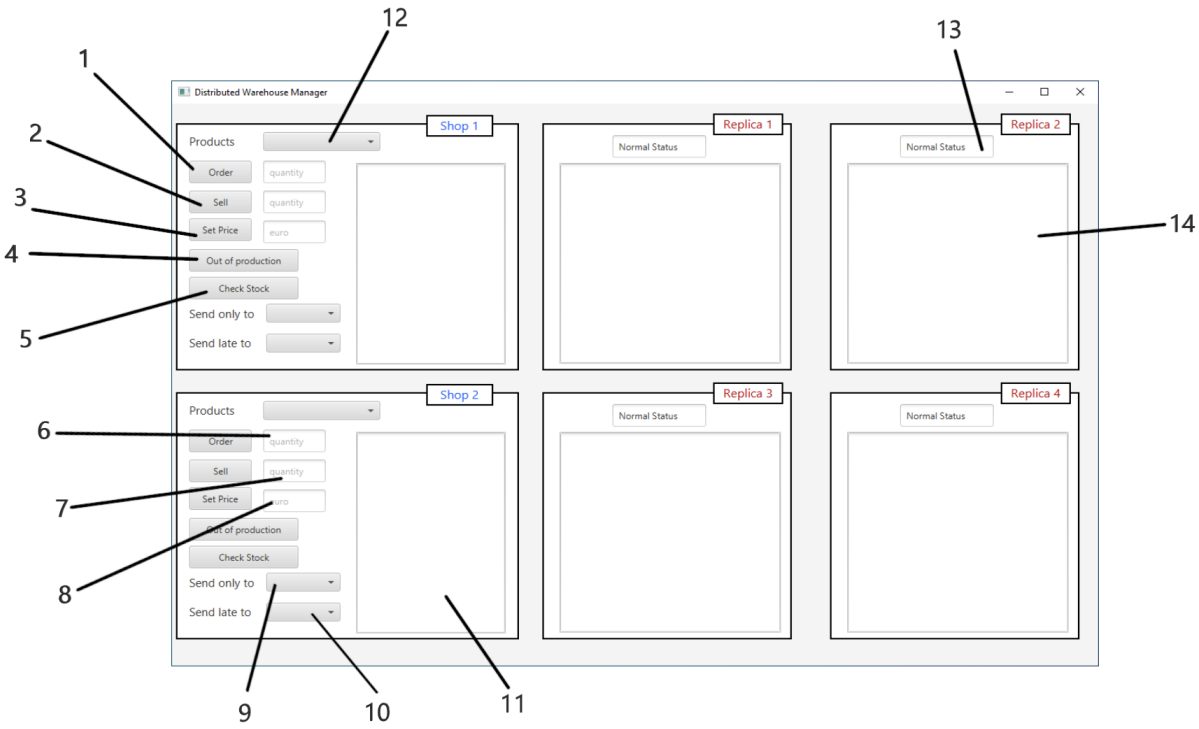

Figure 14: GUI della demo proposta.

Di seguito sarà descritto il funzionamento dei singoli componenti utilizzati nell'interfaccia grafica:

1. **Order Button**: richiede l'ordine di un certo prodotto, causando l'aumento delle scorte in magazzino (per il protocollo è un'operazione di scrittura);

2. **Sell Button**: richiede la vendita di una certa quantità di prodotto selezionato, causando la diminuzione delle scorte di magazzino (per il protocollo è un'operazione di scrittura);

3. **Set Price Button**: cambia il prezzo di uno specifico prodotto in catalogo (per il protocollo è un'operazione di scrittura);

4. **Out Of Production Button**: setta un prodotto fuori produzione. Tale operazione inabilita la richiesta di ordine (per il protocollo è un'operazione di scrittura);

5. **Check Stock Button**: richiede le informazioni su uno specifico prodotto (per il protocollo è un'operazione di lettura);

6. **Order Text Field**: campo di inserimento delle quantità da ordinare. Il valore inserito nel campo deve essere un intero positivo.

7. **Sell Text Field**: campo di inserimento delle quantità venduta. Il valore inserito nel campo deve essere un intero positivo.

8. **Set Price Text Field**: campo di inserimento del prezzo del prodotto. Il valore inserito nel campo deve essere un double positivo.

9. **Send Only Combo Box**: questo campo è di tipo opzionale, è stato introdotto per simulare la ricezione di messaggi solo ad un sottoinsieme di repliche;

10. **Send Late Combo Box**: questo campo è di tipo opzionale, è stato introdotto per inviare in modo ritardato il messaggio ad un sottoinsieme di repliche;

11. **Shop Text Panel**: mostra i risultati della richiesta lato cliente;

12. **Products Combo Box**: permette la scelta dei prodotti presenti in catalogo;

13. **Status Field**: mostra lo stato della replica durante l'evoluzione del sistema (Normal-Status, Update-Status, Conflict-Status, Write-1-Status, Wite-2-Status, Read-Status);

14. **Replica Text Panel**: mostra i messaggi ricevuti e trasmessi della replica.

8 Conclusioni

In questo report è stato riportato la realizzazione di un sistema sistema distribuito che implementa un algoritmo del consenso con annesse le dovute assunzioni. Si è poi redatto il modo dettagliato il protocollo "Hybrid Quorum" evidenziandone i punti di forza rispetto ad altre soluzioni presenti il letteratura. Infine, si è fatta una realizzazione concreta del protocollo utilizzando il *framework* Akka e fornendo una interfaccia grafica atta a mostrare, anche ai meno esperti, i possibili scenari di inconsistenza che un sistema "SMR" può incorrere e come il protocollo agisce per garantire nel tempo la consistenza e tolleranza ai guasti. Il protocollo non è particolarmente difficile da implementare e il fatto che le operazioni siano definibili per il tipo di oggetto con cui il server lavora lo rende molto versatile e paragonabile ad un vero e proprio server con database. Per letteratura è noto che le prestazioni del protocollo siano elevate e all'aumentare della tolleranza il numero totale delle repliche è tenuto sotto controllo, grazie soprattutto ai miglioramenti apportati per coprire le criticità di Q/U. I dati raccolti da Cowling et al.[2] mostrati nei grafici in figura 15. I grafici a sinistra denotano che il carico di lavoro sui server (in scrittura) è notevolmente alleggerito in HQ e Q/U a discapito del client che tutto sommato non riceve grossi rallentamenti, mentre nei grafici a destra la congestione di rete mostra la complessa gestione delle operazioni da parte di BFT e Q/U rispetto alla più leggera di HQ.

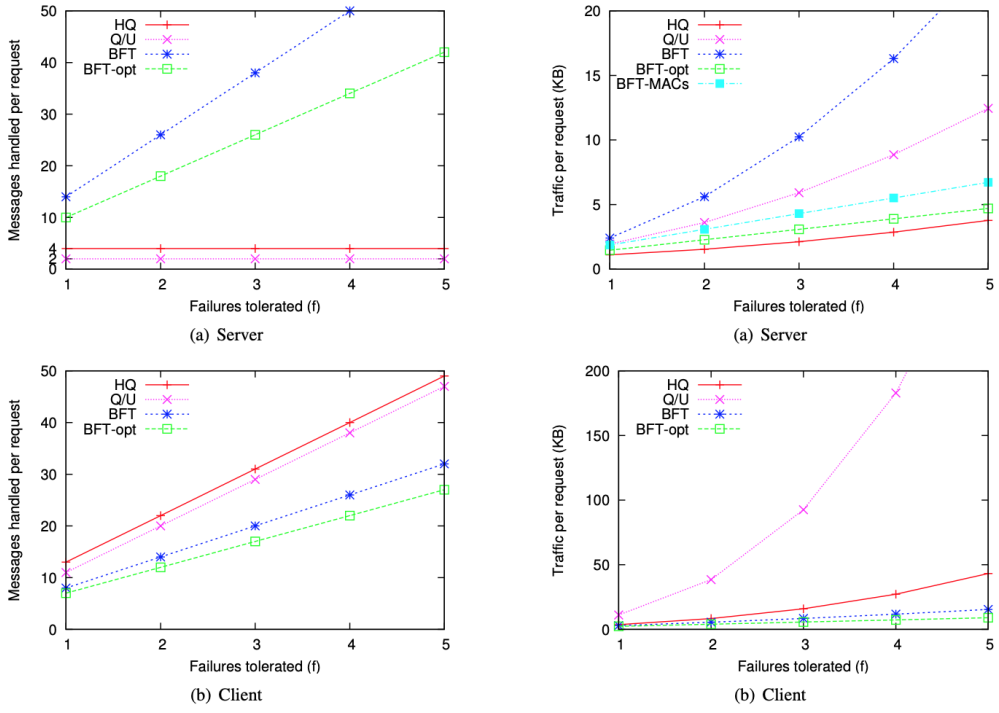

Figure 15: Grafici confronto performance protocolli.

8.1 Lavori Futuri

Il sistema che abbiamo realizzato presenta alcuni limiti, che possono essere migliorati in futuro. Infatti, un aspetto su cui non ci si è soffermati molto è la crittografia, in quanto è stato utilizzato un semplice sistema di firme la cui chiave è uguale per tutti i componenti del sistema. Dunque, avremmo potuto utilizzare, un servizio di distribuzione delle chiavi che fornisce a ciascun nodo la chiave pubblica di qualsiasi altro nodo nel sistema, consentendo cosi la creazione di chiavi di sessione simmetriche da utilizzare nei MAC. Un importante sviluppo potrebbe essere l'aggiunta di un protocollo che gestisce l'inserimento di nuove repliche a run time. Il protocollo deve prevedere per la replica l'assegnamento delle chiavi per la comunicazione con i client e le chiavi di firma, dovrebbe aggiornare lo stato e recuperare le informazioni di cui non dispone. Infine, tutti i client attivi dovrebbero essere avvisati dell'esistenza della nuova replica affinché essa non venga trascurata nelle future comunicazioni, senza contare che l'aggiunta di repliche cambierà il numero del quorum. Infine, si potrebbero implementare operazioni che coinvolgono più oggetti. Dato che, come detto in precedenza, HQ è similare ad un database, l'implementazione di operazioni come ad esempio una inner join di sql può diventare necessaria, dato che potrebbero crearsi relazioni tra gli oggetti. Ciò rende indispensabile lo sviluppo di un protocollo di *happen before* in cui le azioni si susseguono con una stretta correlazione. Rimozione dei metodi messi ad-hoc per far si che la demo potesse creare tutti i casi possibili.

8.2 Cosa abbiamo imparato

- utilizzo di strumenti per il *Version control* come **GitLab** per agevolare il lavoro in team;
- utilizzo di scala come linguaggio di programmazione, il quale, essendo un linguaggio funzionale si è rilevato un potente strumento per scrivere applicazioni distribuite;
- abbiamo toccato con mano quali sono tipiche problematiche insite in un sistema distribuito e quali sono gli approcci per la risoluzione;
- utilizzo del modello ad attori per la gestione degli aspetti asincroni di un sistema distribuito;
- algoritmo del consenso: *Hybrid Quorum Protocol for Byzantine Fault Tolerance*

References

[1] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20:398–461, 2002.

[2] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *In Proc. OSDI*, pages 177–190, 2006.

[3] Michael Abd el malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *In Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74. ACM Press, 2005.

[4] Idit Keidar and Danny Dolev. Increasing the resilience of distributed and replicated database systems, 1995.

[5] L Lamport. The byzantine generals problem,". *ACM Transactions on Programming Languages and Systems,*, pages 382–401, 1982.

[6] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11:203–213.