PBFT

Practical Byzantine Fault Tolerance

Leonardo Perugini
leonardo.perugini2@studio.unibo.it
Mat. 0000954275

Il seguente documento ha lo scopo di studiare l'algoritmo di consenso PBFT (Pratical Byzantine Fault Tolerance), un algoritmo introdotto nel 1999 che garantisce la Byzantine Fault Tolerance, ovvero la capacità di un sistema distribuito di raggiungere il consenso nonostante alcuni suoi nodi siano irraggiungibili, impossibilitati a rispondere o tentino di propagare informazioni errate. I nodi del sistema sono organizzati in sequenza, senza l'ausilio di un cordinatore centrale, e comunicano tramite messaggi adottando un protocollo di comunicazione basato su 3 principali fasi: *PREPREPARE*, *PREPARE* e *COMMIT*. La correttezza di ogni fase del protocollo viene garantita mediante l'utilizzo di Quorum, rappresentati dai vari messaggi scambiati dalle repliche, grazie ai quali per ogni richiesta vengono rilasciati dei certificati che permettono l'avanzamento.

Contestualmente allo studio dell'algoritmo verrà analizzato il meccanismo di *Proactive Recovery* dello stesso PBFT, che consente alle repliche attive di innescare proattivamente e periodicamente dei processi di riavvio al fine di scongiurare o risolvere eventuali failure e recuperare lo stato aggiornato del servizio.

Verrà infine presentato un *proof of concept,* sviluppato in linguaggio Java ed eseguibile mediante interfaccia a riga di comando, in grado di replicare il funzionamendo dell'algoritmo nei suoi aspetti di primaria importanza.

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

Il costante aumento di sistemi online accessibili via Internet comporta naturalmente anche l'aumento delle vulnerabilità di tali sistemi a difetti e guasti. In una situazione ottimale i sistemi in questione dovrebbero fornire servizi in maniera corretta e senza mai interrompersi. A tale scopo esistono molte tecniche basate sulla State Machine Replication (SMR) che cercano di fare fronte alle cause che potrebbero portare all'interruzione del servizio o ad un suo funzionamento arbitrario: questi problemi derivano principalmente dalla presenza di bug nel codice, errori manuali o compromissione del sistema in seguito ad attacchi malevoli. Nell'articolo analizzato, viene presentato l'algoritmo di state machine replication PBFT che è in grado di garantire sia liveness che safety del sistema anche a fronte di molteplici Byzantine Failures tra i suoi nodi e che, a differenza delle soluzioni che lo hanno preceduto, porta molti miglioramenti soprattutto in quelli che sono i presupposti necessari al suo funzionamento.

1.1 Obiettivi e piano di lavoro

La struttura del progetto pone due obiettivi principali:

1. Basandosi sul paper fornito, sarà necessario comprendere il funzionamento dell'algoritmo PBFT nella sua implementazione di base e successivamente comprendendo anche l'integrazione del meccanismo di Proactive Recovery.

2. Basandosi sulle conoscenze apprese dallo studio dell'articolo si procederà all'implementazione di un sistema in grado di replicare il funzionamento dell'algoritmo.

Il piano di lavoro del progetto seguirà i due obiettivi prefissati:

- in una prima fase, di natura teorica, si procederà allo studio approfondito dell'articolo per conoscere l'algoritmo, i suoi aspetti chiave, i punti di forza e le debolezze.

- nella fase successiva, più a stampo realizzativo, si procederà all'implementazione affinando le conoscenze sul funzionamento dell'algoritmo.

1.2 Scenari realizzativi

Presa in considerazione la prevista durata complessiva di progetto, l'implementazione non potrà essere fedele in ogni aspetto descritto nell'articolo. Il funzionamento previsto per il sistema sarà quindi limitato a:

- replicare il funzionamento del sistema in una situazione di normalità, in cui nessuna replica risulta essere fallata.

- replicare una failure in una o più repliche, provocandone comportamento arbitrario nelle decisioni.

- replicare la fase di avanzamento del sistema a fronte di suddette failure.

- ripresa della normale attività.

Per la realizzazione si renderà inoltre necessario stabilire ulteriori presupposti, che verranno dettagliati successivamente nell'apposita sezione. Il meccanismo di Proactive Recovery non viene preso in considerazione tra le varie features dell'algoritmo che verranno replicate.

2 Consenso nei sistemi distribuiti

L'assenza di centralizzazione dello stato e delle decisioni nei sistemi distribuiti comporta che i processi coinvolti debbano giungere a decisioni comuni accordandosi di volta in volta sull'avanzamento, sul risultato di una computazione o sullo stato dell'intero sistema. In un sistema basato su consenso è necessario che i vari nodi che prendono parte alle decisioni rispettino delle precise caratteristiche. Per prima cosa è necessario che ogni nodo riesca prendere una decisione in un tempo limitato: ogni computazione avviata dovrà quindiavere un termine. Inoltre, osservando l'intero sistema, ogni nodo non fallato all'interno del sistema dovrà giungere in modo deterministico alla stessa decisione.

2.1 Byzantine Faults

Nell'ambito del consenso all'interno di un sistema distribuito è necessario prevedere e far fronte ai guasti o alle compromissioni dei singoli nodi, affrontandoli non come critiche eventualità, ma come fasi preventivate dell'esecuzione. Un nodo potrebbe essere in stato di failure per differenti motivi:

- *Denial of Service*, se il nodo è temporaneamente impossibilitato ad inviare o ricevere messaggi.
- *Operator mistakes*, se nel nodo si verifica un errore dovuto ad un'attività errata di un operatore.
- *Software errors*, se l'errore in questione deriva da un bug o da insesattezze nel codice sorgente.
- *Malicious attacks*, se il nodo o il suo comportamento sono manipolati malignamente da un intrusore.

In tutti questi casi è possibile che il nodo compromesso possa smettere di funzionare, omettere alcuni step del processo o fornire risultati che non rispecchiano il reale stato del sistema.

2.2 PBFT

PBFT rappresenta un algoritmo/protocollo di State Machine Replication utile a far fronte al problema dei Byzantine Faults e che garantisce *liveness* e *safety* del sistema, riuscendo a raggiungere il consenso tra gli n nodi attivi anche in situazioni in cui al più $f = \left\lfloor \frac{(n-1)}{3} \right\rfloor$ di questi nodi siano in stato di failure. Questo significa che i client riusciranno sempre ad ottenere una risposta e che tale risposta sarà corretta, mantenendo così la linearizability[1] del sistema. L'algoritmo fa affidamento su alcuni presupposti per il suo funzionamento, ovvero:

- *Strong Cryptography*, ovvero assume che la potenza computazionale di eventuali intrusori sia limitata, rendendo non revertibili le tecniche crittografiche utilizzate.
- *Weak Synchrony*, ovvero assume che i tempi di ricezione dei messaggi siano limitati.
- *Bounds on failures*, ovvero assume un limite massimo al numero di repliche fallate.

Nella versione PBFT-PR, ovvero con meccanismo di Proactive Recovery, le repliche sono in grado di riavviarsi e recuperare lo stato attuale del sistema senza intaccare la correttezza dell'esecuzione, ma piuttosto aumentandone il tempo di vita assoluto.

[1] Proprietà di correttezza del sistema in cui un'esecuzione simultanea risulta equivalente ad un'eventuale esecuzione sequenziale.

3 PBFT: System Model

In questo capitolo verrà descritto il modello alla base dell'algoritmo, fornendo alcuni concetti base ed andando nel dettaglio delle assunzioni accennate nel capitolo precedente.

3.1 Modello

Un servizio replicato è rappresentato da n repliche che eseguono le operazioni richieste dai client; repliche e client sono entità in esecuzione su differenti nodi all'interno del sistema, connessi tra loro mediante una rete di comunicazione.

PBFT implementa una forma di State Machine Replication che permette la replica dei servizi che eseguono operazioni arbitrarie a patto che ogni replica assicuri un esecuzione deterministica, ovvero è necessario che l'esecuzione della stessa sequenza di istruzioni in più repliche distinte produca uno stesso risultato.

3.2 Crittografia

Ogni messaggio che viene inviato per il funzionamento del protocollo viene inviato alle altre entità in chiaro, ma ognuna di esse fa uso di determinate tecniche crittografiche per effettuare due diversi controlli sui messaggi e sul loro contenuto:

- *Correttezza del contenuto*: per verificare che il contenuto del messaggio sia corretto, si fa uso dei codici di *digest*. Il contenuto di ogni messaggio inviato viene processato mediante una funzione di hash D generando un codice che viene accorpato al contenuto stesso. Il destinatario può verificare che il messaggio non sia stato manipolato processando a sua volta il contenuto del messaggio e confrontando il digest ottenuto con quello ricevuto.
- *Autenticità del mittente*: per verificare che il messaggio sia stato generato dal mittente riportato all'interno vengono utilizzati dei codici ***MAC*** *(Message Authentication Code)*. In modo del tutto simile alla generazione del digest il messaggio viene pocessato dal mittente, in questa fase utilizzando un'algoritmo di cifratura simmetrica, generando un codice che viene accorpato al messaggio stesso. Il destinatario potrà quindi verificare l'identità del mittente processando il messaggio ricevuto e confrontando il MAC generato con quello ricevuto.

3.2.1 Note sulla crittografia simmetrica

Ogni coppia (i, j) di entità (client-server e server-server) dovrà condividere una chiave in modo che $k_{i,j} = k_{j,i}$, la quale dovrà quindi essere utilizzata per la creazione dei MAC nella comunicazione bidirezionale tra i e j.

Queste chiavi possono essere inizializzate e aggiornate dinamicamente dalle entità comunicanti mediante l'algoritmo descritto nei capitoli successivi o qualsiasi altro protocollo di scambio chiavi.

3.3 Tipologie di comunicazione

La comunicazione tra le entità può avvenire in due modalità: mediante messaggi *point-to-point* o mediante messaggi *multicast*.

- I messaggi che vengono inviati da un mittente i ad un destinatario j (o viceversa) in modalità *point-to-point*, vengono denotati con $\langle m \rangle_{\mu_{i,j}}$ e contengono un singolo MAC al loro interno, calcolato utilizzando $k_{i,j}$.
- I messaggi che vengono inviati da un mittente i a tutti gli altri in modalità *multicast*, vengono denotati con $\langle m \rangle_{\alpha i}$ e contengono al loro interno un *authenticator*. Per authenticator si intende un array di MAC, uno per ogni replica j ($j \neq i$), dove il MAC alla posizione j è calcolato utilizzando $k_{i,j}$. Ogni destinatario del messaggio dovrà controllare la propria entry nell'authenticator per verificare l'autenticità del messaggio.

3.4 Presupposti fondamentali

PBFT si aspetta veramente poco dai nodi della rete: un nodo viene considerato corretto solo se il suo comportamento corrisponde a quello descritto dall'algoritmo PBFT. Il protocollo si basa sul *Byzantine Failure Model*, ovvero ci si aspetta che all'interno della rete possano essere presenti nodi con comportamento arbitrario. Inolte ci si aspetta che la rete di connessione potrebbe fallire nella consegna dei messaggi, ritardarli o consegnarli in ordine sparso.

Come già accennato, i presupposti principali su cui si basa l'algoritmo PBFT base sono 3. I primi due riguardano il comportamento dei nodi e assicurano entrambi *safety* e *liveness*, mentre il terzo riguarda il comportamento della rete ed è richiesto solo per mantenere la proprietà di *liveness*.

3.4.1 Bounds on failures

Il protocollo PBFT assume un limite di $f = \lfloor (n-1)/3 \rfloor$ al numero di repliche che possono essere fallate. Nella versione che include il meccanismo di Proactive Recovery, questo numero f può essere illimitato durante l'intero ciclo di vita del sistema, a patto che esistano al più f repliche fallate all'interno della stessa *finestra di vulnerabilità*.

3.4.2 Strong Cryptography

Il protocollo PBFT assume che la capacità di calcolo di un possibile intrusore sia limitata, tanto da permettere (con alte probabilità) la non reversibilità dei meccanismi di crittografia utlizzati.

Le funzioni hash utilizzate da PBFT vengono considerate *collision resistant*, impedendo all'avversario di trovare due messaggi m e m' per i quali $D(m) = D(m')$

Inoltre PBFT assume che l'intrusore non sia capace di imitare i MAC: ipotizzando che i e j siano due repliche corrette e che il messaggio $\langle m \rangle_{\mu_{i,j}}$ non sia stato ancora generato, l'intrusore non sarà capace di generarlo.

3.4.3 Weak Synchrony

Definendo $delay(t)$ il tempo trascorso tra il momento t in cui un messaggio viene inviato da i e il momento in cui questo raggiunge il destinatario j, con i e j entrambi corretti, PBFT considera che questo tempo sia limitato, nonostante possa essere soggetto a ritardi dovuti alla rete o alla raggiungibilità degli altri nodi.

3.5 Proprietà del servizio

Rispettando i presupposti descritti in precedenza, per garantire safety e liveness è necessario che almeno $3f + 1$ repliche siano corrette e al massimo f siano fallate. Per garantire la liveness dell'intero sistema, il servizio potrà inviare una risposta prima che la relativa richiesta sia ricevuta da più di $n - f$ repliche, visto che f repliche potrebbero essere fallate o non rispondere. È possibile che alcune delle f repliche che non hanno risposto siano comunque corrette, e quindi che alcune di quelle che hanno risposto siano fallate.

Supponiamo uno scenario in cui un servizio gestisca lo stato di una variabile mutabile sulla quale è possibile eseguire azioni di lettura e scrittura.

Ad una richiesta di scrittura, il sistema potrebbe quindi inviare una risposta dopo che il nuovo valore sia aggiornato solo per un sottoinsieme W composto da $n - f$ repliche, e ad una successiva richiesta di lettura il sistema potrebbe rispondere basandosi sullo stato di un sottoinsieme R composto anch'esso di $n - f$ repliche.

L'intersezione tra i sottoinsiemi W e R potrà contenere al massimo $n-2f$ repliche e al massimo $n - 3f$ repliche non fallate in comune. Sarà impossibile assicurare che la lettura restituisca un valore corretto a meno che i sottoinsiemi W e R abbiano almeno 1 replica non fallata in comune.

La proprietà di safety non è invece garantita mediante *sincronia*. Piuttosto, PBFT utilizza la sincronia per garantire liveness a patto che al massimo $f = \lfloor (n-1)/3 \rfloor$ repliche siano fallate

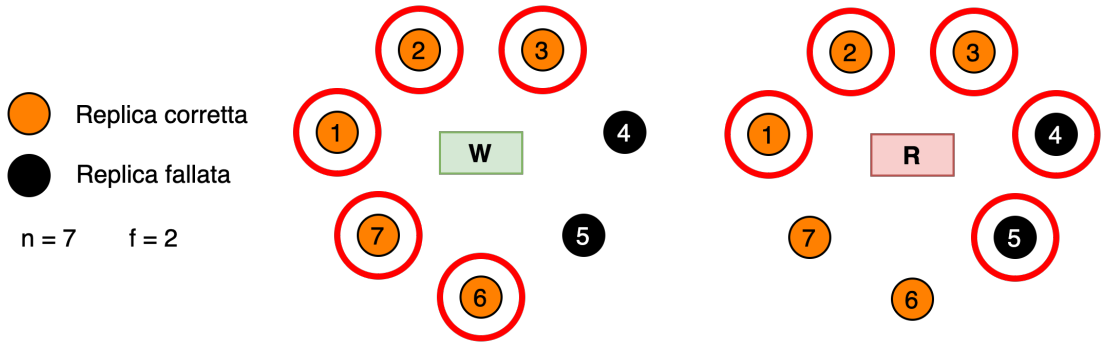

Figura 1: Rappresentazione dei sottoinsiemi W e R nelle operazioni di scrittura e lettura.

e che $delay(t)$ non cresca più velocemente di t. Questa ipotesi di sincronia è piuttosto debole, ma è presumibilmente vera in qualsiasi sistema a condizione che i guasti di rete o gli attacchi denial-of-service rientrino in un tempo limitato.

4 L'algoritmo PBFT

Un sistema che adotta PBFT è modellato come una Macchina a Stati replicata su molteplici nodi che fanno parte della rete. Ogni replica inizia l'esecuzione nello stesso stato delle altre, implementa le operazioni allo stesso modo ed ha un comportamento deterministico. Quindi, tutti i nodi non fallati produrranno la stessa risposta per ogni operazione. In più, vista la natura distribuita dell'algoritmo, ogni replica dovrà mantenere internamente l'intero stato del sistema.

4.1 Cenni sull'interazione Client-Replica

I client inviano delle richieste alle repliche volte ad eseguire delle operazioni di lettura o modifica dello stato del servizio. L'algoritmo garantisce che tutte le repliche non fallate eseguano le stesse operazioni nello stesso ordine. Al termine delle operazioni di consenso, le repliche restituiranno ognuna una risposta al client che ha effettuato la richiesta. Il client deve attendere la ricezione di $f + 1$ risposte con lo stesso risultato da parte delle repliche e, siccome è noto dalle proprietà del servizio che almeno una di queste repliche non è fallata, il risultato può essere considerato corretto.

4.2 Cenni sull'interazione Replica-Replica

Senza un entità di coordinamento centrale, per assicurare che ogni replica esegua le stesse operazioni nello stesso ordine, le repliche devono agire con una logica *Primary-Backup* e fare uso di tecniche di *Quorum*. Le richieste che giungono alle repliche devono essere numerate progressivamente in modo *denso*, ovvero senza lasciare numeri non assegnati.

4.2.1 Meccanismo Primary-Backup

Le repliche del sistema attraversano, durante l'esecuzione, una successione di configurazioni dette *viste*: in ciascuna di queste viste una replica svolge il ruolo di primary, mentre le altre agiranno da backup. Il nodo primary ha il compito di comunicare alle repliche di backup un numero detto *sequence number* che viene assegnato di volta in volta alle richieste dei client in modo crescente per garantirne l'ordinamento. Questa tecnica assegna al primary p un ruolo di centrale importanza, che potrebbe essere rischioso per il sistema nel caso in cui lo stesso primary sia fallato: se così fosse, la replica p potrebbe assegnare più volte lo stesso sequence number a richieste differenti, lasciare dei sequence number non assegnati o smettere di assegnarli. Per questo motivo le repliche di backup incorporano al loro interno anche dei meccanismi di controllo sui sequence number ricevuti, oltre a dei timeout per rilevare eventuali inattività del nodo primary. Quando il primary è considerato fallato dai backup, questi dovranno avviare una procedura di cambio della vista (*View Change*), un processo che consente alle repliche di concordare il passaggio ad una vista successiva eleggendo un nuovo nodo come primary. Durante le operazioni di View Change alcuni sequence number potrebbero risultare inutilizzati, violando il principio di densità. Per evitare che questo accada, i sequence number non utilizzati dovranno comunque essere assegnati a delle richieste di *no-op*, ovvero richieste nulle la cui esecuzione non innesca alcuna operazione sullo stato del servizio.

4.2.1.1 Scelta del primary

In un sistema formato da R repliche, ogni nodo ed ogni vista vengono rappresentati da un numero intero nel range $\{0, ..., |R| - 1\}$. Per semplicità viene definito $|R| = 3f + 1$, con f corrispondente al massimo numero di repliche fallate tollerabili. Il primary di una vista sarà la replica $p = v \mod |R|$, dove v è l'intero che rappresenta la vista.

4.2.2 Quorum System

Viene definito quorum un qualsiasi sottoinsieme di repliche che prende parte ad una certa decisione. I quorum condividono tra loro due proprietà fondamentali:

- *Intersection*: ogni coppia di quorum condivide almeno una replica corretta in comune.
- *Availability*: è sempre possibile ottenere un quorum in cui non ci sono repliche fallate.

Grazie a queste proprietà, l'utilizzo dei quorum costituisce un meccanismo di memoria per le informazioni scambiate tra i nodi del protocollo e permette l'avanzamento del protocollo. Memorizzando le informazioni ricevute dagli altri nodi sotto forma di messaggi, le repliche "ottengono" ad ogni step dei *certificati di quorum* che garantiscono l'avvenuta memorizzazione delle informazioni. Ogni certificato è in sostanza un set composto da $2f + 1$ messaggi, ognuno ricevuto da un diverso nodo del sistema, che assicura l'affidabilità di una decisione presa in quanto appunto sostenuta da un quorum di repliche.

L'algoritmo PBFT utilizza inoltre dei certificati *weak*, deboli, composti solo da *f+1* messaggi da repliche diverse, i quali garantiscono che almeno una replica corretta nel sistema abbia memorizzato le informazioni.

4.3 Funzionamento dei Client

Un generico client c richiede al sistema l'esecuzione di un'operazione o effettuando un invio multicast alle repliche di un messaggio del tipo

$$\langle \textit{REQUEST}, o, t, c \rangle_{\alpha_c}$$

in cui il timestamp t viene utilizzato per assicurare la semantica *exactly once* nell'esecuzione delle richieste. I timestamp di c sono ordinati in modo che una richiesta successiva abbia un timestamp maggiore rispetto alle precedenti. La richiesta dovrà essere autenticata dalle repliche che la ricevono per essere accettata, per cui è compito del client computare e accorpare alla richiesta il relativo MAC. Dopo le operazioni di consenso e l'esecuzione dell'operazione richiesta, le repliche invieranno singolarmente le risposte al client mediante un messaggio del tipo

$$\langle \textit{REPLY}, v, t, c, i, r \rangle_{\mu_{ic}}$$

in cui v è il numero della vista in cui l'operazione è stata eseguita, t è il timestamp della richiesta corrispondente, i è il numero identificativo della replica che ha inviato il messaggio e r è il risultato dell'esecuzione dell'operazione sullo stato. Prima di accettare il risultato r, il client deve ricevere un certificato weak, detto *reply certificate*, composto di $f + 1$ risposte ricevute da altrettante repliche differenti e che contengono MAC valido e gli stessi valori di t ed r.

Ogni client deve mantenere attivo un timer, avviato al momento dell'invio della richiesta, che limiti l'attesa dell'ottenimento del certificato di risposta. Nel caso in cui un client non riceva un certificato di risposta in tempo, ovvero entro la scadenza dell'attuale timer, effettuerà la ritrasmissione della stessa richiesta a tutte le repliche.

Si assume che ogni client attenda la ricezione della risposta ad una sua richiesta prima di inviare la successiva.

4.4 Funzionamento delle Repliche

Per giungere al consenso e all'esecuzione di una richiesta, le repliche adottano un protocollo suddiviso in 3 fasi principali: *pre-prepare*, *prepare* e *commit*. Eseguendo le fasi di pre-prepare e prepare, il sistema riesce a garantire il corretto ordinamento delle richieste anche a fronte di un primary fallato, mentre eseguendo le fasi di prepare e commit si garantisce l'ordinamento delle operazioni di commit, anche a fronte di cambiamenti di vista.

Ogni replica mantiene al suo interno una copia dello stato del servizio, rappresentato dall'insieme di più valori: il valore dello stato del servizio, un log contenente tutti i messaggi accettati e inviati dalla replica e un valore intero che denota il numero della vista corrente.

Quando una replica riceve una richiesta $m_{\alpha_c} = \langle REQUEST, o, t, c \rangle_{\alpha_c}$ da un client c, deve innanzitutto autenticarla verificando che il MAC contenuto nel messaggio corrisponda al MAC calcolato utilizzando la chiave condivisa con c e solo dopo può iniziare le operazioni di consenso.

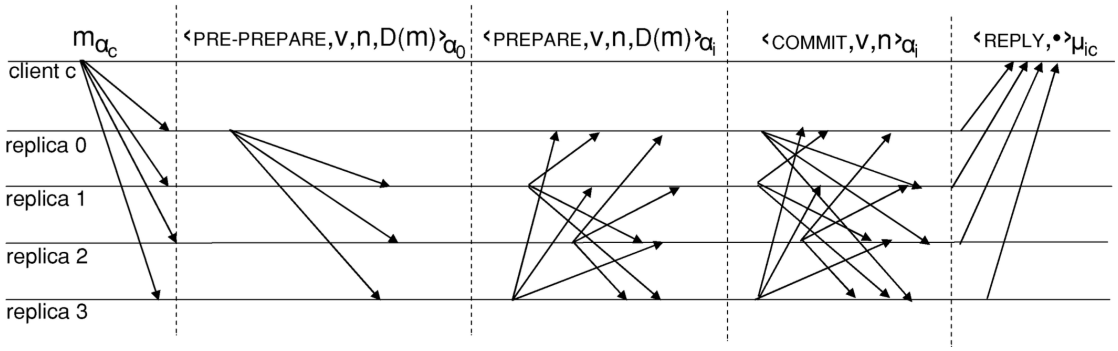

Figura 2: Rappresentazione di un normale scenario di esecuzione con 1 client e 4 repliche.

4.4.1 Messaggi di PRE-PREPARE

Il comportamento di una replica quando viene ricevuta una richiesta può variare a seconda che essa sia primary oppure no. Una replica di backup avrà solo il compito di autenticare e memorizzare la richiesta ricevuta nei propri log mentre per quanto riguarda il primary, dopo aver autenticato la richiesta, le assegna il sequence number n successivo e ne calcola il digest $D(m)$, dopodichè invia in modalità multicast alle repliche di backup un messaggio di pre-prepare del tipo

$$\langle \textit{PRE-PREPARE}, v, n, D(m) \rangle_{\alpha_p}$$

dove v rappresenta la vista in cui il primary ha ricevuto il messaggio. Tutti i messaggi di pre-prepare e i successivi messaggi di prepare e commit contengono al loro interno i valori v ed n: questi tipi di messaggi possono essere accettati dalle altre repliche solo se, oltre ad essere autenticati, contengono un valore di v che corrisponde alla vista attuale ed n è compreso tra due valori limite, detti *low water mark* h e *high water mark* H.

Una replica di backup che riceve un messaggio di pre-prepare deve controllare che tutte le condizioni descritte in precedenza siano soddisfatte, oltre a verificare di non aver già accettato una pre-prepare con sequence number n e vista v, ma differente $D(m)$.

4.4.2 Messaggi di PREPARE

Una replica che accetta una PRE-PREPARE e nei suoi log contiene la relativa richiesta, entra nella fase di *prepare* effettuando un'invio multicast di un messaggio del tipo

$$\langle \textit{PREPARE}, v, n, D(m), i \rangle_{\alpha_i}$$

e aggiungendo entrambi i messaggi di PRE-PREPARE e PREPARE ai propri log. Se, altrimenti, il messaggio non viene accettato, la replica non deve compiere nessuna azione.

Una richiesta si dice *pre-prepared* in una determinata replica se quella replica ha effettuato l'invio di almeno un messaggio di PRE-PREPARE o di PREPARE per quella richiesta.

Una replica i continuerà ad accettare messaggi finchè non avrà nei suoi log la PRE-PREPARE e almeno $2f$ PREPARE con uguali n, v e m: quando questa condizione si verifica, la replica ottiene un *prepared certificate*, il quale attesta che un quorum di repliche è d'accordo nell'assegnare il sequence number n al messaggio m nella vista v. In questo caso la richiesta si dice *prepared* dalla replica i: PBFT garantisce che non è possibile ottenere un prepared certificate per due richieste differenti con stesso sequence number nella stessa vista.

Per dimostrare la veridicità di questa affermazione, si provi ad assumere per assurdo che essa sia falsa. In questo caso esisterebbero due richieste distinte m ed m' con un prepared certificate con stesso sequence number n nella vista v. Per la proprietà di intersezione dei quorum è noto che qualsiasi coppia di quorum condivide almeno una replica non fallata, quindi i quorum di repliche che hanno consentito la generazione del certificato avranno sicuramente almeno una replica corretta in comune. Tale replica dovrebbe aver quindi inviato messaggi di PRE-PREPARE e PREPARE per assegnare lo stesso sequence number sia ad m che ad m' nella stessa vista, ma

sapendo che la replica è corretta questo non è possibile quindi dalla contraddizione ne deriva che m ed m' non sono distinte.

4.4.3 Messaggi di COMMIT

L'esecuzione delle prime due fasi consente di far accordare le repliche su un ordinamento delle richieste nella stessa vista, ma non è in grado di garantirlo a fronte di un processo di viewchange. Le repliche infatti potrebbero generare i prepared certificates in viste diverse per lo stesso sequence number e richieste differenti. La fase di commit riesce a risolvere tale problema.

Al momento dell'ottenimento di un prepared certificate, ogni replica lo comunica generando un messaggio di COMMIT

$$\langle \textit{COMMIT}, v, n, i \rangle_{\alpha_i}$$

che aggiunge ai propri log ed invia in modalità multicast alle altre repliche. Dopodichè la replica rimane in attesa finchè non avrà almeno $2f + 1$ messaggi di COMMIT per il sequence number n e vista v dalle altre repliche. Al verificarsi di questa condizione, la replica ottiene un committed certificate. Una richiesta si dice *committed* da una replica se questa possiede sia un prepared certificate che un committed certificate per quella richiesta. Quando una richiesta n è in stato committed, il client può esegure l'operazione richiesta a patto che abbia già eseguito tutte le operazioni relative alle richieste con sequence number minori di n. Infine, dopo l'esecuzione, la replica deve inviare un messaggio di REPLY direttamente al client comunicando il risultato dell'operazione. Per garantire la semantica *exactly once*, la replica dovrà scartare tutte le richieste da parte di quella replica che hanno un timestamp inferiore a quello appena evaso.

4.4.4 Ulteriori proprietà

Il protocollo non fa affidamento sulla consegna ordinata dei messaggi, quindi le richieste potrebbero giungere in stato di committed in modo disordinato. Questo non rappresenta un problema per lo svolgimento delle normali operazioni, visto che ogni replica mantiene tutte le informazioni relative ai messaggi di PRE-PREPARE, PREPARE e COMMIT fino a che la richiesta non viene definitivamente eseguita.

4.5 Checkpoint System

Con l'avanzamento dell'esecuzione può verificarsi il caso che il numero di log del sistema raggiunga un numero elevato, specialmente all'aumentare delle repliche attive nel sistema. Per far fronte a questo, il protocollo PBFT suggerisce un meccanismo per prevenire che i log memorizzati dalle repliche crescano senza un limite. Le informazioni sulle richieste già eseguite potrebbero essere rimosse dai log delle singole repliche, ma alcuni certificati potrebbero rendersi necessari anche in seguito per garantire la safety del sistema. Per questo le repliche necessitano di una prova, detta *checkpoint*, che garantisca la correttezza del proprio stato in un determinato punto dell'esecuzione. Grazie ai checkpoint, le repliche potranno quindi procedere con la rimozione dai propri log di tutte le richieste la cui esecuzione è già contemplata nello stato del servizio.

La generazione di un checkpoint corrispondente ad ogni richiesta potrebbe chiaramente essere troppo dispendiosa per cui le repliche devono procedere con la generazione di un checkpoint periodicamente, idealmente ogni qualvolta che verrà eseguita una richiesta il cui sequence number sia divisibile per un determinato intervallo K.

Una replica i che produce o recupera un checkpoint invia alle altre repliche con modalità multicast un apposito messaggio

$$\langle \textit{CHECKPOINT}, n, d, i \rangle_{\alpha_i}$$

dove n rappresenta il sequence number dell'ultima richiesta eseguita e d il digest dello stato. Quando una replica raccoglie $2f + 1$ messaggi di CHECKPOINT da differenti repliche che contengono stessi n e d, ottiene uno *stable certificate*. Questo assicura che le altre repliche potranno raccogliere uno weak certificate sullo stato del servizio se ne avranno bisogno.

Quando una replica ottiene lo stable certificate per un sequence number n potrà rimuovere tutte le informazioni nei propri log che fanno riferimento a richieste con sequence number inferiore ad n, oltre che a tutti i precedenti checkpoint.

In questo modo il protocollo riesce anche a far scorrere in avanti la finestra dei sequence number accettabili, rappresentata dai water mark h e H. Il limite maggiore sarà rappresentato come $H = h + L$, dove L rappresenta la dimensione dei log e può essere definita moltiplicando l'intero K per un certo fattore, anche piccolo.

4.6 Processo di View Change

Il processo di cambiamento di vista garantisce liveness al sistema consentendo alle operazioni di procedere anche a fronte di un fallimento della replica primary. Inoltre, serve anche al mantenimento della prorpietà di safety assicurando che le repliche si accordino sui sequence number assegnati a richieste in stato di committed nel passaggio dalla vista v alla vista $v + 1$.

L'idea di base è far raccogliere alla nuova replica primary le informazioni riguardanti gli stable certificate e i prepared certificate di un quorum di repliche e propagarle nella nuova vista. Per la proprietà di intersezione dei quorum, il nuovo primary è sicuro di ottenere informazioni che rappresentano tutte le richieste committed nelle viste precedenti e tutti gli stable checkpoint.

4.6.1 Strutture dati utilizzate

Le repliche fanno uso di due insiemi P e Q contenenti le informazioni relative ai sequence number compresi tra h ed H. Questi insiemi consentono al sistema di avanzare anche a fronte di Viewchange multipli e sono normalmente vuoti durante le normali operazioni. Inoltre, tutte le richieste corrispondenti alle entry di questi insiemi devono essere memorizzate dalle repliche.

- P_i conterrà tutte le informazioni riguardanti le richieste che sono state prepared dalla replica i nelle viste precedenti.
- Q_i conterrà tutte le informazioni riguardanti le richieste che sono state pre-prepared dalla replica i nelle viste precedenti.

Ogni entry di P e Q ha forma $\langle n, d, v\rangle$ e attesta che per una richiesta con digest d, sequence number n nella view v la replica ha, rispettivamente, un prepared certificate e inviato almeno un messaggio di PRE-PREPARE o PREPARE.

4.6.2 Messaggi di VIEW-CHANGE

Un nodo di backup i che sospetta della correttezza dell'attuale primary avvia il processo di viewchange cambiando la propria vista in $v+1$ ed inviando in modalità multicast alle altre repliche un messaggio

$$\langle \textit{VIEW-CHANGE}, v+1, h, C, P, Q, i\rangle_{\alpha_i}$$

in cui h rappresenta il sequence number dell'ultimo stable checkpoint noto ad i, C è un insieme formato dalle coppie $\langle n, d\rangle$ di sequence number e digest di ogni checkpoint noto ad i, mentre P e Q sono gli insiemi descritti in precedenza. Tali insiemi devono essere aggiornati appena prima dell'invio del messaggio di viewchange utilizzando le informazioni nei log della replica. Dopo aver inviato il messaggio la replica rimuove dai propri log tutti i messaggi di PRE-PREPARE, PREPARE e COMMIT.

4.6.2.1 Algoritmo di aggiornamento di P e Q

La procedura per l'aggiornamento degli insiemi P e Q segue una semplice ma fondamentale logica. Per ogni richiesta con sequence number compreso tra i water mark h e H l'algoritmo deve svolgere in sequenza due operazioni:

1. Controllare che la richiesta con sequence number n e digest d sia in stato prepared nella vista v: in caso affermativo l'algoritmo rimuove da P, se presente, qualsiasi altra richiesta in stato prepared avente sequence number n, digest d' e view v', dopodichè aggiunge la richiesta all'insieme P.

2. Controllare che la richiesta con sequence number n e digest d sia in stato pre-prepared nella vista v: in caso affermativo l'algoritmo rimuove da Q, se presente, qualsiasi altra richiesta in stato prepared avente sequence number n e view v', dopodichè aggiunge la richiesta all'insieme Q.

La procedura è ben descritta dal seguente pseudo-codice.

let v be the view before the view change, L be the size of the log, and h be the log's low water mark.

for all n such that $h < n \leq h + L$ do
  if request number n with digest d is prepared in view v then
    if $\exists \langle n, d', v' \rangle \in \mathcal{P}$
      remove $\langle n, d', v' \rangle$ from $\mathcal{P}$
    add $\langle n, d, v \rangle$ to $\mathcal{P}$
  if request number n with digest d is pre-prepared in view v then
    if $\exists \langle n, d, v' \rangle \in \mathcal{Q}$
      remove $\langle n, d, v' \rangle$ from $\mathcal{Q}$
    add $\langle n, d, v \rangle$ to $\mathcal{Q}$

Figura 3: Aggiornamento degli insiemi P e Q.

4.6.3 Messaggi di VIEW-CHANGE-ACK

Le repliche ricevono i messaggi di VIEW-CHANGE per la vista $v + 1$, accettandoli solo se contengono nei loro insiemi P e Q informazioni relative a viste minori o uguali alla propria vista v. Le repliche inviano quindi al primary della nuova vista dei messaggi di acknowledgment

$$\langle \textit{VIEW-CHANGE-ACK}, v + 1, i, j, d \rangle_{\mu_{ip}}$$

contenenti l'identificatore i del mittente, il digest d del messaggio di VIEW-CHANGE a cui l'ack fa riferimento e l'identificatore j della replica che ha inviato la VIEW-CHANGE. Grazie a questi messaggi, il nuovo primary riesce a stabilire l'autenticità dei messaggi di VIEW-CHANGE inviati da repliche fallate.

4.6.4 Messaggi di NEW-VIEW

La replica primary p raccoglie nei suoi log i messaggi di VIEW-CHANGE e di VIEW-CHANGE-ACK, inclusi quelli che si sarebbe inviato da solo. Ogni volta che p riceve $2f + 1$ messaggi di VIEW-CHANGE-ACK per il messaggio di VIEW-CHANGE inviato da una replica i, aggiunge tale VIEW-CHANGE in un set S in cui ogni entry fa riferimento ad una specifica replica.

L'aggiunta di un elemento ad S sta a significare che p ha ottenuto un *view-change certificate* per la VIEW-CHANGE inviata da i.

Ogni volta che verrà aggiunto un nuovo elemento nell'insieme S, il primary userà le informazioni contenute in S e la *decision procedure* descritta in seguito per determinare un checkpoint e una serie di richieste che verranno, al momento opportuno, inviate alle altre repliche per permettere la ripresa delle normali operazioni. Questo avviene mediante l'invio in modalità multicast alle altre repliche di un messaggio di NEW-VIEW del tipo

$$\langle \textit{NEW-VIEW}, v + 1, V, X \rangle_{\alpha_p}$$

in cui X contiene il checkpoint e l'insieme delle richieste selezionate mentre V è un insieme contenente, per ogni elemento in S, una coppia di valori formata dall'identificativo della replica e dal digest del suo messaggio di VIEW-CHANGE.

4.6.4.1 Decision Procedure

Il primary inizia questa procedura selezionando il checkpoint che verrà utilizzato come *punto di partenza* per lo stato della nuova vista. Per farlo seleziona il checkpoint con il più alto numero h che abbia sequence number maggiore del *low water mark* e che è noto essere corretto, ovvero per il quale è presente uno weak certificate, per almeno $f + 1$ repliche.

Dopodichè, per ogni sequence number compreso tra h e $h + L$, il primary seleziona una richiesta di cui inviare PRE-PREPARE. Ogni richiesta scelta deve rispettare la condizione **A** o la condizione **B**:

- **Condizione A.**
  - ***Condizione A1.*** La richiesta era in stato prepared in almeno una replica facente parte di un quorum nella precedente vista v.
  - ***Condizione A2.*** La richiesta era in stato pre-prepared in almeno una replica non fallata in una precedente vista e quindi può essere posta in stato prepared nella successiva vista $v + 1$.
- **Condizione B.** Esiste un quorum di repliche per le quali non è presente alcuna richiesta che soddisfa interamente la condizione **A**.

Nel caso in cui si verifichi la condizione **B**, verrà selezionata una speciale richiesta impotente di *no-op* che viene riconosciuta regolarmente dal protocollo ma che non provoca modifiche allo stato.

Il primary potrebbe impiegare più di $n - f$ messaggi prima di completare la procedura, ma nel caso in cui abbia ricevuto tutte le VIEW-CHANGE dalle repliche non fallate è sicuro di poterla portare a termine.

$$\text{let } \mathcal{C} = \{\langle n, d \rangle \mid \exists S, S' \subseteq \mathcal{S} : |S| > 2f \wedge |S'| > f \wedge \forall m \in S : m.h \leq n \wedge \forall m \in S' : \langle n, d \rangle \in m.\mathcal{C}\}$$

$$\text{if } \exists \langle h, d \rangle \in \mathcal{C} : \forall \langle n', d' \rangle \in \mathcal{C} : n' \leq h \text{ then}$$
select checkpoint with digest d and number h
else exit

for all n such that $h < n \leq h + L$ do
A. if $\exists m \in \mathcal{S}$ with $\langle n, d, v \rangle \in m.\mathcal{P}$ that verifies:
A1. $\exists\, 2f + 1$ messages $m' \in \mathcal{S}$:
$m'.h < n \wedge \forall \langle n, d', v' \rangle \in m'.\mathcal{P} : v' < v \vee (v' = v \wedge d' = d)$
A2. $\exists\, f + 1$ messages $m' \in \mathcal{S} : \exists \langle n, d', v' \rangle \in m'.\mathcal{Q} : v' \geq v \wedge d' = d$
then select the request with digest d for number n

B. else if $\exists\, 2f + 1$ messages $m \in \mathcal{S}$ such that $m.h < n \wedge m.\mathcal{P}$ has no entry for n
then select the null request for number n

Figura 4: Pesudo codice per l'algoritmo di *Decision Procedure*

4.6.4.2 Elaborazione dei messaggi di NEW-VIEW

Al momento dell'invio dei messaggi di NEW-VIEW, il primary deve aggiornare il proprio stato in modo che rispecchi le informazioni appena comunicate alle repliche. Per questo avrà bisogno di recuperare dalle altre repliche eventuali richieste dall'insieme X che non possiede. Questa procedura dovrà essere ripetuta anche se il primary non è in possesso del checkpoint selezionato. Se tutte le informazioni sono presenti o recuperate, il primary registrerà nei propri log che le richieste sono pre-prepared per la vista $v + 1$.

Una replica di backup riceve messaggi finchè non ottiene, per ogni coppia nell'insieme V, un messaggio di NEW-VIEW corretto e il corrispondente messaggio di VIEW-CHANGE. Nell'eventualità che una replica non abbia ricevuto il messaggio di VIEW-CHANGE da qualche replica non sarà in grado di verificare l'autenticità del messaggio ricevuto dal primary, ma potrà fare affidamento sui messaggi di VIEW-CHANGE-ACK. Siccome p includerà un messaggio di VIEW-CHANGE nell'insieme S solo dopo aver ottenuto un certificato valido, almeno $f+1$ repliche potranno garantire l'autenticità di ogni VIEW-CHANGE il cui digest è contenuto in V. Se una replica non dovesse cooperare all'avanzamento il primary ritrasmetterà il messaggio di VIEW-CHANGE, mentre le repliche corrette ritrasmetteranno i propri VIEW-CHANGE-ACK. Inolte, una replica di backup potrà accettare un messaggio di VIEW-CHANGE non autenticato a patto che riceva almeno f VIEW-CHANGE-ACK per i quali identificatore e digest sono contenuti in V.

Quando una replica di backup accetta un messaggio di NEW-VIEW e possiede i rispettivi messaggi di VIEW-CHANGE, effettua un controllo riguardante le decisioni comunicate dal primary eseguendo la *decision procedure*. Se le informazioni non coincidono la replica deve passare direttamente alla vista $v+2$, mentre in caso di coincidenza devono essere accettate le informazioni comunicate dal primary aggiungendole ai propri log ed inviando un messaggio di PREPARE valido per la vista $v+1$ per ogni messaggio impostato in stato di pre-prepared. In seguito a questo la replica può riprendere le sue azioni abituali.

4.6.5 Correttezza del sistema

È fondamentale che l'algoritmo continui a garantire le due proprietà fondamentali di *safety* e *liveness*, anche durante il processo di View Change.

4.6.5.1 Safety

Se una richiesta m raggiunge lo stato di commit con sequence number n in una replica corretta per la vista v, deve essere assicurato che nessun'altra richiesta possa diventare committed con uguali v ed n in una replica corretta e che la *Decision Procedure* non scelga una richiesta differente da m per il sequence number n e vista $v' > v$. Questo implica che dopo il commit della richiesta m con sequence number n nella vista v, nessuna richiesta distinta da m con lo stesso sequence number potrà essere pre-prepared da una replica corretta.

Di conseguenza, le repliche sono d'accordo su un ordine totale perchè non effettuano mai commit di richieste distinte con lo stesso sequence number. Questo può essere dimostrato per induzione sul numero di views tra v e v': al momento del commit di m la replica i riceve i messaggi di commit da un quorum Q di repliche in cui la richiesta è stata prepared con sequence number n e vista v.

- *Caso base* ($v' = v$): è vero, visto che la proprietà di intersezione dei quorum assicura che due richieste distinte non possono essere committed in una replica corretta con stesso sequence number n e nella stessa vista v.

- *Passo induttivo* ($v' > v$): assumendo per assurdo che venga scelta una richiesta $m' \neq m$ con sequence number n nella vista v, questo significa che le condizioni A1 o B sono verificate. La proprietà di intersezione dei quorum assicura che esista sempre una VIEW-CHANGE da una replica corretta $j \in Q$ per cui $h < n$ in ogni quorum certificate usato per soddisfare A1 o B.

  La condizione *B* non potrà essere vera in quanto il messaggio di VIEW-CHANGE inviato da j per v' conterrà all'interno della sua componente P l'elemento $\langle n, D(m), v_c \rangle$ con $v_c \geq v$.

  La condizione *A1* potrebbe essere vera se un messaggio di VIEW-CHANGE da una replica fallata contenesse nella sua componente P l'elemento $\langle n, D(m'), v_f \rangle$ con $v_f > v_c$, ma la condizione *A2* previene questa eventualità. Infatti essa sarà verificata solo in caso esista un messaggio di VIEW-CHANGE da una replica corretta contenente nella sua componente Q l'elemento $\langle n, D(m'), v'_c \rangle$ con $v'_c \geq v_f$.

  Siccome, con probabilità quasi massima, $D(m) \neq D(m')$, l'ipotesi induttiva implica che $v'_c \leq v$ e di conseguenza $v_f \leq v$: questo fa sì che *A1* e *A2* non potranno mai essere contemporaneamente vere.

4.6.5.2 Liveness

Per garantire liveness del sistema le repliche devono effettuare un processo di View Change nel caso non sia possibile eseguire una richiesta. Questo processo potrebbe essere innescato sia dalle repliche stesse quando queste sospettano che l'attuale primary possa essere fallato oppure da un timeout, per prevenire che una replica aspetti all'infinito l'esecuzione di una richiesta. Una replica di backup è in stato di attesa se ha ricevuto una richiesta valida ma non l'ha ancora eseguita. La replica avvia un timer al momento della ricezione della richiesta, sempre che non sia già avviato, e lo interrompe quando non è più in attesa di eseguirla (in questo caso il timer potrebbe essere riavviato per una richiesta successiva). Al fine che il processo sia considerato *live*, vivo, dovrà essere dimostrato che il nuovo primary p sia in grado di inviare i messaggi di NEW-VIEW (posto che abbia abbastanza tempo prima che le altre repliche passino alla vista successiva).

Si assuma per assurdo che un primary corretto, senza limiti di tempo, sia incapace di raggiungere una decisione utilizzando la *Decision Procedure*. Questo starebbe a significare che il primary non è in grado di (a) determinare un checkpoint che soddisfi le condizioni, oppure che (b) per qualche n compreso tra h e $h + L$ il primary non è in grado di stabilire una richiesta che soddisfi le condizioni.

Come prima cosa il primary sarà sempre in grado di soddisfare (a), ovvero riuscire a trovare almeno un checkpoint che soddisfa le condizioni della procedura, quindi riuscirà a proseguire scegliendo uno dei checkpoint trovati. Sia h_c il sequence number dell'ultimo checkpoint stabile in una delle repliche corrette. Siccome nel sistema sono presenti almeno $2f + 1$ repliche corrette e almeno $f + 1$ di queste possiedono il checkpoint h_c, il primary sarà in grado di scegliere il valore di h_c per h nella procedura. Qualsiasi sia il checkpoint scelto dal primary, una replica sarà sempre in grado di recuperarlo nel caso non ne sia in possesso, visto che almeno una replica corretta lo conosce.

Inolte, per ogni sequence number n compreso tra i water mark h e $h+L$ il primary sarà sempre in grado di soddisfare (b), selezionando una richiesta che soddisfa le condizioni *A1* e *A2* oppure la condizione *B*. In questa situazione le eventualità che possono verificarsi sono due: (1) la richiesta è in stato prepared in alcune repliche corrette, oppure (2) non esiste una replica corretta in cui la richiesta è prepared.

1. Se la richiesta è in stato prepared, la condizione *A1* sarà sempre verificata visto che nel sistema sono presenti sempre almeno $2f + 1$ repliche corrette, le quali non possono preparare due richieste distinte con stesso sequence number n e vista v. Allo stesso tempo sarà vera anche la condizione *A2* perchè se almeno una replica corretta ha preparato la richiesta, tale richiesta sarà stata pre-prepared in almeno $f + 1$ repliche corrette. In più, la condizione *A2* assicura anche l'autenticità della richiesta, visto che almeno una replica corretta deve averla memorizzata.

2. La veridicità della condizione *B* deriva dall'ipotesi iniziale per la quale esistono $2f + 1$ repliche corrette che non hanno alcuna richiesta prepared con sequence number n.

Perciò, siccome entrambe le condizioni (a) e (b) saranno sempre verificate, l'ipotesi che la procedura non riesca a terminare non trova conferma.

Un altro fattore importante per la liveness del sistema è la massimizzazione del periodo in cui almeno $2f + 1$ repliche corrette si trovano nella stessa view e una di loro è il primary. In aggiunta a questo, i timer che regolano l'avanzamento possono essere modificati in automatico a run-time in modo che questo periodo aumenti esponenzialmente fino a che alcune operazioni non vengano eseguite. A meno che il ritardo di consegna dei messaggi all'interno del sistema cresca maggiormente della durata dei timer, la liveness del sistema viene garantita con 3 tecniche.

1. Per evitare che il processo di View Change venga eseguito troppo presto, una replica che invia in modalità multicast una VIEW-CHANGE per $v+1$ attenderà $2f+1$ messaggi di VIEW-CHANGE per $v + 1$ prima di avviare il proprio timer di durata T. Se il timer della replica dovesse scadere prima che questa abbia ricevuto un messaggio di NEW-VIEW oppure prima che sia stata eseguita nella nuova vista una richiesta che non era stata mai eseguita, allora la replica avvierà un nuovo processo di VIEW-CHANGE per $v+2$ ma questa volta aumenterà la durata del suo timer a $2T$.

2. Se una replica dovesse ricevere messaggi di VIEW-CHANGE per differenti viste maggiori della sua vista attuale, invierà una sola VIEW-CHANGE per la vista il cui numero è minore tra quelle ricevute. Questo è necessario per evitare di innescare una ulteriore View Change troppo presto.

3. Le repliche fallate non saranno in grado di forzare un processo di View Change in quanto sono necessari almeno $f + 1$ messaggi di VIEW-CHANGE inviati da repliche diverse a fronte di un massimo di f repliche fallate. Unica eccezione è data dall'eventualità in cui la replica fallata che innesca la VIEW-CHANGE sia il primary, inviando messaggi errati o interrompendo i propri invii. Anche nel caso peggiore, ovvero in cui le f repliche fallate diventino primary una dopo l'altra, lo svolgimento delle normali operazioni potrà essere ripristinato in un tempo finito, al più dopo f processi di View Change.

4.6.5.3 Fairness

PBFT è un algoritmo che assicura *fairness* in quanto consente ai client di ottenere risposte anche quando sono presenti altri client che utilizzano il servizio. Infatti, un primary non fallato assegna ad ogni richiesta un sequence number n con modalità *FIFO*. I backup mantengono tali richieste al loro interno in una coda FIFO e, durante un processo di View Change, fermeranno il proprio timer solo in seguito all'esecuzione di una delle richieste in questa coda, impedendo al primary di favorire l'esecuzione da parte di certi client a discapito degli altri.

5 PBFT con Proactive Recovery

I sistemi con una durata prolungata nel tempo vedono incrementare di molto la possibilità di superare il numero massimo f di repliche fallate. Il meccanismo di Proactive Recovery affiancato all'algoritmo PBFT permette alle repliiche difettose di tornare ad avere un comportamento corretto durante l'esecuzione, consentendo quindi di poter avere un numero di repliche fallate superiore ad f durante l'intero ciclo di vita del sistema, a patto che f sia il numero massimo di repliche fallate all'interno di una *finestra di vulnerabilità*. Siccome le repliche *Bizantine* possono continuare a comportarsi correttamente, quindi di fatto potrebbero essere fallate senza essere scoperte, è necessario che il meccanismo di PR si azioni in modo proattivo anche quando non ci sarebbero i presupposti per sospettare di una replica.

5.1 Ulteriori assunzioni

Per rendere possibile il processo di recovery è necessario che le repliche fallate che si riavviano siano in grado di autenticarsi mutualmente con le altre repliche attive, oltre ad avere un meccanismo affidabile che inneschi questi riavvii periodici. Alcune delle seguenti considerazioni presuppongono che un eventuale intrusore del sistema non abbia accesso fisico alle repliche, in caso contrario la recovery diventerà un onere dell'amministratore del sistema. Si assume anche che i client possiedano un co-processore crittografico che faciliterà il processo di scambio delle chiavi, al contrario il processo può essere modificato aggiungendo un ulteriore round di messaggi senza perdere efficacia.

5.1.1 Secure Cryptography

Ogni replica deve possedere un co-processore ausiliario per la crittografia, mediante il quale cifrare e decifrare i messaggi senza esporre le chiavi private memorizzate al suo interno. Il coprocessore deve inoltre mantenere al suo interno un contatore che viene incrementato progressivamente e il cui valore viene di volta in volta accorpato al messaggio che deve essere inviato.

5.1.2 Read-Only memory

Le chiavi pubbliche delle altre repliche devono essere memorizzate in una zona di memoria in sola lettura e che rimarrà inalterabile, anche in caso di failure. Un esempio potrebbe essere una porzione del bios, che in alcune macchine richiede accesso fisico per essere modificato.

5.1.3 Watchdog timer

Ogni replica ha un timer *di guardia* che, periodicamente, interrompe l'esecuzione passando il controllo ad un monitor per la recovery, memorizzato in una porzione read-only di memoria. Al contrario di altre implementazioni della PR, che richiedono di mantenere nella memoria read-only tutto il programma eseguito dalla replica oltre ad utilizzare dei canali autenticati per la comunicazione, PBFT fa utilizzo solo del suddetto monitor e demanda al co-processore le operazioni di autenticazione dei messaggi. PBFT-PR necessita di un'ulteriore assunzione sulla sincronia per garantire liveness. Dovrà esistere un certo punto dell'esecuzione dopo il quale (a) tutti i messaggi vengono consegnati entro un tempo costante Δ (dipendente dai valori dei timer scelti) oppure (b) tutte le repliche corrette hanno ottenuto risposta alle proprie richieste.

5.2 L'algoritmo PBT-PR

L'algoritmo utilizza i quorum come memoria affidabile oer memorizzare le informazioni sull'ordine delle richieste, quindi è richiesto che tale meccanismo di memoria continui a funzionare anche durante e dopo il processo di recovery. Ogni certificato di quorum ricevuto da una replica non fallata deve essere supportato da un quorum, ovvero nello stato delle repliche che compongono quel quorum sarà registrato l'invio di tale messaggio oppure dovrà essere presente un checkpoint stabile.

Il meccanismo di recovery deve assicurare che lo stato attuale del servizio sia consistente con lo stato del protocollo. In ogni replica corretta, lo stato corrente in corrispondenza del sequence number n deve essere uguale allo stato ottenuto eseguendo tutte le richieste in stato committed comprese tra il sequence number $h+1$ ed n in ordine crescente, partendo dal checkpoint stabile h.

Devono essere affrontati diversi problemi per garantire la totale invarianza dopo il recupero di una replica.

1. Deve essere impossibile per un intrusore impersonare una replica dopo che il suo processo di recovery è terminato. Per questo la chiave utilizzata per l'autenticazione deve essere modificata dopo ogni recovery e le altre repliche devono scartare tutti i messaggi autenticati con la vecchia chiave.

2. Se una replica attende dei messaggi per un certificato per un tempo sufficiente, potrebbe ricevere più di f messaggi inviati da delle repliche quando queste erano in uno stato non corretto. Per questo sia le repliche che i client devono scartare tutti i messaggi che non fanno parte di un certificato completo al momento della propria recovery. Per assicurare liveness, in questa situazione, tutti i messaggi che verranno ritrasmessi dovranno essere cifrati con la nuova chiave.

3. Vista la natura proattiva del processo di recovery potrebbe capitare che esso venga innescato in una delle repliche corrette, la quale non dovrà trovarsi in uno stato fallato dopo il termine della recovery. Nel caso di replica corretta deve essere quindi mantenuta la correttezza e garantita la partecipazione della replica allo scambio di messaggi, utile al completamento della recovery stessa. Nel caso di replica fallata invece, il processo di recovery dovrà ripristinare lo stato in modo che torni a soddisfare tutte le invarianti, oltre ad evitare che la replica diffonda informazioni errate. La difficoltà è data dal fatto che non si può sapere se la replica che sta eseguendo la recovery sia corretta oppure no.

5.2.1 Scambio delle chiavi

Le repliche e i client aggiornano periodicamente le chiavi utilizzate per l'autenticazione dei messaggi ricevuti scambiandosi dei messaggi di NEW-KEY della forma

$$\left\langle \textit{NEW-KEY}, i, ..., \{k_{j,i}\}_{\in j}, ..., t \right\rangle_{\sigma i}$$

in cui t, detto timestamp, rappresenta il valore del contatore della replica, mentre $\{k_{j,i}\}$ è la chiave che la replica j utilizzerà in fututo per autenticare i messaggi che invierà ad i. Tutti questi messaggi sono firmati dal coprocessore utilizzando la chiave privata della replica. Le repliche fanno uso di t per bloccare i falsi messaggi di NEW-KEY, controllando che questo valore sia più grande del valore accettato per i precedenti messaggi di NEW-KEY. Ogni replica condivide con il client anche una chiave segreta utilizzata per le comunicazioni bidirezionali, la quale viene aggiornata forzatamente dal client ad intervalli regolari comunicando quella nuova alle repliche.

Siano t_1 e t_2 ($t_1 < t_2$) due istanti temporali in cui due differenti messaggi di NEW-KEY sono stati inviati da una replica i, definiamo l'intervallo $[t_1, t_2]$ come *epoca di aggiornamento* e la sua durata $t_1 - t_2$ come *periodo di aggiornamento*.

Al momento dell'invio di un nuovo messaggio di NEW-KEY, il client o la replica dovranno rimuovere dai propri log tutti i messaggi che non hanno un certificato completo, eccetto i messaggi di PREPARE e PRE-PREPARE. Da questo momento tutti i messaggi autenticati con le vecchie chiavi saranno scartati, quindi viene garantito che ogni nodo corretto accetterà solo i messaggi autenticati in quell'epoca.

5.2.2 Processo di Recovery

La recovery permette di ripristinare il comportamento di una replica non corretta, consentendo al sistema di supportare più di f repliche fallate durante il suo ciclo di vita. Quindi, al termine della recovery, è necessario che la replica (i) esegua codice corretto, (ii) non sia impersonata da un intrusore e (iii) il suo stato rispetti le invarianti.

5.2.2.1 Reboot

Se una replica che deve iniziare la propria recovery crede di essere il primary per la vista v, deve effettuare un invio multicast di un messaggio di VIEW-CHANGE per $v + 1$. In questo modo l'*availability* del sistema aumenta, visto che viene forzata una View Change prima che i timer dei singoli backup scadano. L'esecuzione trarrà giovamento soprattutto nel caso il primary sia fallato, visto che è sempre cosa buona rimpiazzare un primary non corretto.

Il recovery monitor salva su disco lo stato della replica (formato da tutti i log, dallo stato del servizio e da tutti i checkpoint salvati), dopodichè riavvia il sistema ricaricando da disco il codice corretto e ripristinando lo stato salvato. La correttezza del codice può essere verificata salvando il digest del codice sorgente in una parte read-only della memoria e confrontandolo ad ogni recovery con quello salvato su disco. Nel caso in cui il controllo abbia esito negativo, la replica può richiedere il codice corretto alle altre repliche o recuperarlo mediante altre tecniche.

La replica dovrà continuare ad elaborare richieste anche durante la recovery per mantenere sia la safety che la liveness del sistema, in caso contrario potrebbe portare ad un numero di $f + 1$ repliche fallate nel sistema nella stessa finestra di vulnerabilità. Se la replica fosse fallata, il suo stato potrebbe essere corrotto e potrebbero essere inviati messaggi conoscendo le chiavi utilizzate per la generazione dei MAC sia in ricezione che in invio. È importante quindi che siano scartate tutte le chiavi condivise con i client e che venga eseguita una nuova procedura di NEW-KEY per cambiarle, altrimenti un intrusore potrebbe ostacolare il successo della recovery impersonando qualsiasi client o replica.

5.2.2.2 Estimation Protocol

Dopo aver completato la fase di reboot, la replica dovrà eseguire una procedura detta *Estimation Protocol* che permette di effettuare una stima circa il valore massimo H_m del suo watermark che poteva avere in precedenza nel caso non fosse stata fallata ed elimina tutti i log e i checkpoint con sequence number maggiore di quello stabilito. In questo modo viene posto un limite massimo al sequence number dei messaggi non corretti inviati dalla replica, pur garantendo che nessuno stato venga scartato se la replica era corretta.

La replica i inizia la procedura inviando in modalità multicast un messaggio

$$\langle \textit{QUERY-STABLE}, i \rangle_{\alpha i}$$

alle altre repliche. Una replica j che riceve questo messaggio deve rispondere alla replica i con un messaggio

$$\langle \textit{REPLY-STABLE}, c, p, i \rangle_{\mu_{ji}}$$

in cui c e p sono rispettivamente i sequence number dell'ultimo checkpoint stabile e dell'ultima richiesta in stato prepared nella replica j. La replica i processa tutti i messaggi mantenendo il minimo valore di c e il massimo valore di p ricevuti da ogni replica, oltre ad un proprio valore sia per c che per p.

Da questo momento, fino al termine dell'esecuzione della procedura, la replica non accetterà altri messaggi che non siano NEW-KEY e REPLY-STABLE.

La replica i utilizza le risposte ricevute per selezionare il valore di $H_M = L + c_M$, dove L è la dimensione dei log e c_M è un valore ricevuto da una qualche replica j che soddisfa due condizioni:

- $2f$ repliche diverse da j devono aver comunicato un valore c minore o uguale a c_M.
- f repliche devono aver comunicato un valore di p maggiore o uguale a c_M.

Il valore di c_M deve inoltre essere maggiore del sequence number di ogni checkpoint stabile noto alla replica i nel caso in cui non sia fallata, così da non rimuovere voci di log. Questo è assicurato dal fatto che se un checkpoint è stabile, sarà sicuramente presente in $f + 1$ repliche non corrotte e avrà un sequence number minore o uguale al valore di c proposto. Il controllo su p garantisce che c_M sia vicino ad un checkpoint in una delle repliche corrette, dal momento che almeno una replica corretta ha comunicato un p maggiore o uguale a c_M.

È garantita la *liveness* del sistema anche durante l'esecuzione della procedura, visto che ci saranno sempre $2f + 1$ repliche corrette nel sistema e ognuna propone un valore di c solo se

la corrispondente richiesta è in stato committed. Questo implica che la richiesta con sequence number c sarà in stato prepared in almeno $f + 1$ repliche corrette, quindi la replica i potrà fare affidamento sicuramente su un insieme di messaggi inviati da repliche corrette.

Alla fine della procedura i riprende la normale interazione ma non potrà inviare messaggi con sequence number maggiore di H_M finchè non avrà un checkpoint stabile per sequence number maggiore o uguale a H_M, in modo da limitare il numero di messaggi malevoli che la replica potrebbe inviare.

5.2.2.3 Richieste di Recovery

Terminato il protocollo di stima, i invia con modalità multicast alle altre repliche un messaggio di richiesta di recovery

$$\langle \mathit{REQUEST}, \langle \mathit{RECOVERY}, H_M \rangle, t, i \rangle_{\sigma i}$$

prodotto dal co-processore crittografico, in cui t è il counter interno del coprocessore in modo da avere un valore ogni volta differente. Le altre repliche controllano il valore di t e scartano il messaggio se si tratta di un valore già processato oppure se hanno già processato recentemente una richiesta di recovery dalla replica i, dove per recentemente si intende un periodo di tempo pari alla metà della durata del proprio watchdog timer. Questo controllo è importante per sconfiggere tentativi di Denial of Service innescando continue richieste di recovery nelle repliche. Le richieste di recovery vengono trattate come le altre tipologie di richieste, quindi assumeranno un proprio sequence number n_R e attraverseranno le tre fasi PRE-PREPARE, PREPARE e COMMIT con un'unica differenza al momento dell'esecuzione, in cui ogni replica invierà il proprio messaggio di NEW-KEY.

Le repliche devono inviare un NEW-KEY anche quando, durante la procedura *fetching* dello stato, si accorgono che è dovuto all'esecuzione di una recovery request. Questo perchè se una replica fosse fallata le chiavi potrebbero essere note all'intrusore, ma cambiandole si pone un limite $H_R = \lfloor n_R/K \rfloor \times K \times L$ al sequence number di messaggi corrotti che le altre repliche potranno accettare.

Le repliche useranno la stessa tecnica utilizzata dai client per la raccolta delle risposte, ma aspetteranno $2f + 1$ risposte. Quando una replica possiede un numero sufficiente di risposte procederà a determinare il proprio recovery point $H = max(H_M, H_R)$ e determinerà una nuova view mantenendo la sua vista corrente v_R se ha almeno $f + 1$ risposte con view maggiore o uguale a v_R oppure se fosse entrata in v_R appena prima della recovery, mentre in tutti gli altri casi passerà ad una vista intermedia tra quelle comunicate. Se si rendesse necessario passare ad una vista v_M, la replica invierà una VIEW-CHANGE per v_M ma dovrà aspettare il messaggio di NEW-VIEW per diventare operativa in tale vista. Questo controllo sulla view è fondamentale per impedire alla replica di cambiare in una vista minore di quella in cui era prima di avviare la recovery.

5.2.2.4 Controllo e recupero dello stato

Durante la sua ripresa, una replica i utilizza un meccanismo di trasferimento dello stato per capire di quali parti necessiti dalle altre repliche. Una replica è correttamente recuperata quando possiede un checkpoint con sequence number maggiore o uguale ad H. Questa informazione può quindi essere sfruttata dalle altre repliche, che ricevendo un checkpoint H da i capiranno che è recuperata in modo completo e corretto.

In aggiunta le altre repliche potranno effettuare delle valutazioni sul tempo che sta impiegando i a compiere la propria recovery, cercando di rilevare possibili Denial of Service che ne rallentano la terminazione.

5.2.3 Ulteriori proprietà del servizio

PBFT-PR assicura sia *liveness* che *safety* del sistema, posto che ci sia un numero massimo di f repliche che falliscano contemporaneamente all'interno di intervalli di durata $T_v = 2T_k + T_r$, dove T_v rappresenta la finestra di vulnerabilità, T_k è il maggiore dei periodi di aggiornamento delle chiavi in repliche non fallate e T_r è il periodo di tempo maggiore che impiega una replica fallata

a recuperarsi completamente. Per evitare che i nodi corretti raccolgano certificati con più di f messaggi corrotti è opportuno impostare la durata della finestra di vulnerabilità dell'algoritmo ad un valore maggiore o uguale di T_v, anche se i valori di T_k e T_r sono specifici di ogni esecuzione e non sono noti all'algoritmo. Il meccanismo di aggiornamento delle chiavi assicura che i nodi non fallati accettino solo certificati composti da messaggi generati all'intenro di un intervallo di tempo di $2T_k$. In più limitando il numero di repliche che possono fallire all'interno di un intervallo $T + T_r$ viene assicurato che non ci saranno mai più di f repliche fallate in un qualsiasi intervallo T. Perciò ogni certificato conterrà al massimo f messaggi inviati da delle repliche mentre queste erano in uno stato fallato.

Il meccanismo di recovery preserva le invarianti visto che le repliche non perdono il loro stato durante la procedura. Tali invarianti sono rispettate anche quando la replica è fallata, visto che le altre repliche non accetteranno messaggi corrotti inviati dalla replica in fase di recovery contenenti sequence number maggiori del recovery point e soprattutto tale replica possiederà entro la fine del suo recupero dei log corretti e un checkpoint con sequence number corrispondente al recovery point. Questo assicura che la replica avrà un checkpoint stabile con sequence number maggiore di ogni messaggio inviato prima e inoltre potrà essere accettata come parte di un certificato da repliche e client.

L'algoritmo ha poco controllo sul valore di T_v, in quanto il suo valore è dipendente da T_r e potrebbe aumentare in caso di D.o.S. L'algoritmo riesce invece a controllare maggiormente il valore di T_k e T_w, il valore massimo tra i watchdog timer delle repliche, visto che l'affidabilità dei timer è considerata piuttosto alta. Per determinare il valore del timeout è necessario considerare un trade-off tra sicurezza e performance visto che si otterrà un sistema più sicuro con valori piccoli, ma allo stesso tempo causeranno un numero maggiore di recovery e cambi di chiave degradando le performance. È comunque necessario impostare T_k abbastanza grande da comprendere tre ritardi di messaggi in normali condizioni di esecuzione al fine di essere sicuri di garantire liveness.

Il valore di T_w deve basarsi sul valore di R_n, ovvero il tempo necessario ad una replica non fallata di svolgere la sua recovery, evitando di avviare recovery prima che una precedente non sia terminata o di avviare la recovery in più di f repliche contemporaneamente. Il valore può quindi essere impostato come $T_w = 4 \times s \times R_n$, dove s è un fattore di sicurezza e il fattore 4 aiuta a scaglionare il recupero di $3f + 1$ repliche. R_n è formato principalmente dal tempo di reboot e dal tempo di controllo dello stato di una replica. In condizioni normali una replica non fallata non carica eccessivamente la rete per controllare il suo stato, quindi ci si aspetta che il recupero contemporaneo di f repliche abbia una durata di circa R_n e per questo s può assumere un valore vicino all'1.

Il valore di T_v non può essere considerato limitato se è in corso un D.o.S. ma le repliche possono misurare il tempo di recovery ed avvisare l'amministratore di sistema se questo tempo dovesse superare un tempo costante definito da R_n, concedendo all'amministratore scelto la facoltà di scegliere se portare a termine la recovery della replica. Per rafforzare ulteriormente il sistema è anche consigliabile che le repliche loggino tutte le informazioni riguardo alle recovery, anche e soprattutto quelle relative alle tempistiche e ai faults che si verificano.

6 Tecniche implementative

In questa sezione vengono riassunte alcune ottimizzazioni che possono essere implementate nell'algoritmo e viene fornito un dettaglio ulteriore sulla gestione dei checkpoint.

6.1 Ottimizzazioni

Le prestazioni del sistema possono essere aumentate implementando alcune ottimizzazioni nell'algoritmo senza tuttavia intaccare le proprietà di safety e liveness. Una prima ottimizzazione è stata già descritta in precedenza ed è rappresentata dall'utilizzo della crittografia simmetrica per una computazione decisamente più rapida dei MAC.

6.1.0.1 Digest replies

Per la riduzione del consumo di banda e degli overhead di CPU in corrispondenza di grandi risultati, è possibile agire sulle informazioni che vengono scambiate tra client e repliche. Nell'inviare una richiesta, un client sceglie anche una specifica replica dalla quale farsi restituire il risultato utilizzando un criterio random o una tecnica definita. Solo la replica indicata dal client provvederà ad inviare al client il risultato dell'esecuzione per intero, tutte le altre repliche ne calcoleranno il digest e lo invieranno al client che, dopo aver ricevuto $f + 1$ risposte comprensive del risultato per intero, provvederà a verificarne la correttezza. Nel caso in cui il controllo dia esito negativo il client inoltrerà la richiesta alle repliche, stavolta specificando che tutte dovranno rispondere con il risultato per intero.

6.1.0.2 Tentative execution

La terza ottimizzazione consente di ridurre il numero totale dei ritardi dei messaggi per ciascuna invocazione di esecuzione. Ogni replica esegue provvisoriamente le richieste non appena (i) viene ottenuto un prepared certificate, (ii) lo stato non riflette l'esecuzione di tutte le richieste con sequence number inferiore e (iii) tutte le richieste con sequence number superiore sono in stato di committed. Dopo l'esecuzione della richiesta la replica effettua dei tentativi di risposta al client che deve attendere un quorum certificate con risposte contenenti tutte lo stesso risultato, in modo da essere certo che la richiesta si trova in stato prepared in almeno $f + 1$ repliche e che entrerà sicuramente in stato committed in futuro nelle repliche corrette. Nel caso il timer scada prima dell'ottenimento del certificato, il client ritrasmette la richiesta attendendo non più tentativi ma risposte effettive.

Se la replica dovesse iniziare un processo di View Change mentre sta effettuando esecuzione provvisoria di alcune richieste, è necessario che ripristini il proprio stato a quello del checkpoint contenuto nell'ultima NEW-VIEW o in un successivo checkpoint, avendo cura di scegliere quello con sequence number maggiore.

Questa ottimizzazione potrebbe anche permettere di eliminare lo step dei messaggi di COMMIT, accorpandoli ai successivi messaggi di PRE-PREPARE o PREPARE che la replica invierà senza inficiare sulla latenza dei messaggi o sovraccaricare la CPU.

6.1.0.3 Operazioni Read-Only

Un'altra ottimizzazione riguarda la gestione delle operazioni di sola lettura che non modificano in alcun modo lo stato del servizio, e per le quali è possibile ridurre la latenza ad un solo scambio di messaggi client e repliche. Alla ricezione di una richiesta di sola lettura, le repliche provvederanno immediamente alla sua esecuzione previa autenticazione del client ed invieranno indietro la risposta solo dopo che tutte le richieste precedenti saranno in stato committed. Se il client riesce ad ottenere un certificato con le risposte ricevute accetterà il valore comunicato, altrimenti inoltrerà la richiesta alle repliche sotto forma di richiesta read-write in modo che possa essere eseguita seguendo il corretto ordinamento.

6.1.0.4 Batching di richieste

Questa tecnica permette di ridurre l'overhead del protocollo assegnando un singolo sequence number ad un intero gruppo di richieste e, di conseguenza, avviare una sola esecuzione per tutto il gruppo. Questo è possibile mediante un meccanismo a *finestra scorrevole* che specifica quante richieste è possibile eseguire in parallelo.

Siano P l'attuale primary, e il sequence number dell'ultimo batch eseguito da P, p il sequence number dell'ultima richiesta in stato prepared in P e W la dimensione della finestra. Quando P riceve una richiesta, se $p \geq e + W$ verrà messa in coda, altrimenti viene avviata una nuova istanza del protocollo. Dopo l'esecuzione della richiesta la finestra scorrerà in avanti permettendo alle richieste in coda di essere eseguite e P selezionerà tante richieste fino ad arrivare ad un limite fisso di dimensione W assegnando un unico sequence number all'intero gruppo ed inviandole tutte nello stesso messaggio di PRE-PREPARE. Le repliche che riceveranno il batch, provvederanno all'esecuzione di una richiesta per volta seguendo l'ordine comunicato da P ed invieranno risposte singole per ciacuna richiesta.

6.2 Checkpoint management

Questa sezione entra nel dettaglio del funzionamento del meccanismo di *Garbage Collection* dell'algoritmo PBFT, descrivendo come vengono creati i checkpoint, calcolati i digest e quali sono le strutture dati necessarie. Dopodichè verrà descritta una tecnica per il trasferimento e il controllo dello stato nelle repliche che effettuano una recovery. Negli esempi riportati lo stato del servizio viene rappresentato come una gerarchia di partizioni, in modo da ridurre sia il costo computazione nel calcolo dei digest che la quantità di informazioni trasferite per la recovery di una replica. Per ogni replica si pone che sia in esecuzione un'istanza di PBFT-PR che implementa sia *tentative execution* che *batching di richieste*.

6.2.0.1 Strutture dati

L'organizzazione in partizioni gerarchiche ha origine da un nodo radice contenente l'intero stato del servizio, il quale è suddiviso in s partizioni figlie di uguale dimensione e contigue tra loro. Ogni nodo non foglia dell'albero viene chiamato *metadata* ed ha a sua volta s partizioni figlie, mentre i nodi foglia sono detti *pagine*.

Ogni replica mantiene una copia dell'intero albero per ogni checkpoint che viene memorizzato, creandola quando genera il checkpoint e rimuovendola quando questo viene scartato. Queste copie sono logiche in quanto, utilizzando la tecnica del *copy-on-write*, vengono effettivamente memorizzate solo le copie delle tuple che sono state modificate da quando il checkpoint è stato generato. I checkpoint vengono creati dopo l'esecuzione provvisoria di un batch di richieste con sequence number n divisible per l'intervallo di checkpoint K, anche se i relativi messaggi di checkpoint vengono inviati solo dopo che il batch di richieste assume stato committed. Nell'albero verranno memorizzate delle tuple con forma $\langle l_m, d \rangle$ per ogni metadata ed una tupla con forma $\langle l_m, d, p \rangle$ per ogni pagina, in cui l_m rappresenta il sequence number del checkpoint alla fine dell'ultima epoca in cui la partizione è stata modificata, d è il digest della partizione e p è il valore della pagina. I digest delle partizioni assumono un valore di primaria importanza sia per permettere alle repliche di accordarsi su un valore iniziale per le nuove view, sia per ridurre la mole di dati che transita durante un trasferimento dello stato. Tuttavia, in base alla tipologia della partizione, i digest sono calcolati in modo differente:

- Il digest di una **pagina** viene calcolato concatenando assieme l'indice della pagina, lo stato, il valore di l_m e p.
- Il digest di un **metadata** viene invece calcolato concatenando l'indice della pagina, il suo livello, il valore di l_m e un altro valore ottenuto dividendo per un intero abbastanza grande la somma dei singoli digest delle sotto-partizioni.

6.2.0.2 State Transfer

Una replica avvia una procedura di recupero dello stato quando ottiene un checkpoint per un sequence number maggiore del valore dell'high water mark nei suoi log, ricevendolo mediante

appositi messaggi di checkpoint o durante una View Change. Questa procedura deve essere efficiente riducendo la quantità di informazioni inviate alle altre repliche, in quanto potrebbe essere eseguita frequemente in occasione delle recovery di una replica e deve anche assicurare la correttezza dello stato a fronte di repliche fallate o di modifiche concorrenti allo stato.

L'idea alla base consiste nella visita ricorsiva dell'albero al fine di determinare quali partizioni non sono corrette facendo affidamento di volta in volta sul digest della partizione padre, che *riassume* i valori delle sue sotto-partizioni per un certo sequence number. Una replica da inizio alla procedura ottenendo uno weak certificate con il digest della partizione radice per un certo checkpoint c e userà questo valore per determinare la correttezza delle sotto-partizioni che verranno recuperate, per le quali non saranno necessari certificati a meno che non siano state scartate.

Una replica i richiede una partizione inviando alle altre repliche in modalità multicast un messaggio

$$\langle \textit{FETCH}, l, x, lc, c, k, i \rangle_{\sigma i}$$

in cui x rappresenta il valore della partizione, l è il suo livello all'interno dell'albero, lc è il sequence number dell'ultimo checkpoint noto per la partizione, mentre il valore c potrebbe essere nil oppure specificare che la replica sta cercando il valore della partizione con sequence number c dalla replica k. Generalmente il messaggio di FETCH servirà per richiedere informazioni sulla root partition, specificando in lc il valore dell'ultimo checkpoint. Il valore di c non sarà nil se la replica già conosce il digest corretto della partizione in corrispondenza del checkpoint c, infatti dopo una View Change la replica potrebbe conoscere il digest anche se non possiede il checkpoint. La replica crea una nuova copia logica dell'albero per memorizzare lo stato che recupera e parallelamente inizializza una tabella $\mathcal{LC}$ nella quale memorizza il numero dell'ultimo checkpoint riflesso sullo stato di ogni partizione, impostando come valore iniziale lc in tutte le entry. Se il messaggio di FETCH arriva alla replica designata k e tale replica possiede un checkpoint per il sequence number c, risponderà alla replica i con un messaggio

$$\langle \textit{META-DATA}, c, l, x, P, k \rangle$$

in cui P è un insieme di tuple $\langle x', lm, d \rangle$, ognuna per ogni sotto-partizione di (l, x) con indice x', digest d e $lm > lc$. Siccome la replica conosce il digest del valore della partizione per il checkpoint c, ne può verificare la correttezza senza bisogno di certificati, riducendo il lavoro richiesto alle altre repliche e mantenendo la liveness durante i processi di View Change nei casi in cui lo stato iniziale sia noto solo da una replica corretta. Le repliche diverse da k risponderanno al messaggio di FETCH solo nel caso in cui possiedano un checkpoint stabile maggiore di lc e c, inviando un messaggio di META-DATA autenticato con MAC ed in cui il valore c corrisponde al valore del checkpoint in questione.

La replica i continua a trasmettere il messaggio di FETCH cambiando ogni volta il destinatario K finchè qualche replica K non invia la propria risposta, oppure finchè non ottiene uno weak certificate per il nuovo sequence number c' formato da risposte con stesso valore per la sottopartizione. A questo punto i confronta i propri digest corrispondenti ad ogni sotto-partizione di (l, x) con i digest appena ricevuti procedendo come segue:

- per una partizione che viene aggiornata, la replica modifica la entry relativa alla partizione nella tabella $\mathcal{LC}$ impostando c come valore.
- per una partizione il cui digest non combacia, verrà inviato il relativo messaggio di FETCH

La replica effettua la visita ricorsiva di un ramo dell'albero finchè non completa l'invio di tutti i messaggi di FETCH per le sue foglie, ai quali otterrà in risposta solo il digest e il sequence number dell'ultima modifica alla pagina anzichè alle sottopartizioni. Solo la replica designata K invierà un messaggio

$$\langle \textit{DATA}, x, p \rangle$$

diverso dalle altre in cui specifica l'indice della pagina x e il suo valore p. Quando i riceve un nuovo valore per una pagina procederà a modificare l'albero aggiornando lo stato, il digest e il sequence number dell'ultima modifica di quella pagina, oltre ad aggiornare il valore corrispondente nella tabella $\mathcal{LC}$. Terminata la visita di un ramo dell'albero, l'algoritmo passa ad un ramo *gemello* per verificarne la consistenza: una partizione viene considerata consistente ad un sequence number c

se c è il valore minore tra tutti quelli contenuti nella tabella $\mathcal{LC}$ e se c è maggiore o uguale al massimo dei sequence number corrispondenti alle ultime modifiche nelle sotto-partizioni. Seguendo un comportamento ricorsivo, l'algoritmo procederà alla visita delle sotto-partizioni solo in caso di inconsistenza, altrimenti passa al ramo successivo o sale di livello.

Quando l'algoritmo completa la visita della partizione root e ne verifica la consistenza al sequence number c, la procedura termina e la replica i riprende le normali operazioni processando richieste con sequence number maggiore di c.

Questa procedura viene eseguita da i mentre le altre repliche svolgono le loro operazioni di routine e in conseguenza delle quali sono libere di scartare i propri checkpoint, quindi la replica potrebbe impiegare del tempo per terminarla. Le informazioni ricevute per ogni partizione sono considerabili già *vecchie* e potrebbero essere state di nuovo modificate al momento in cui i le riceve, di conseguenza la procedura potrebbe non terminare mai se la frequenza di aggiornamento delle partizioni è maggiore della frequenza di aggiornamento di i. Nonostante ci sia l'eventualità che questo accada, il tutto è molto improbabile e potrebbe essere anche migliorato effettuando traferimento in parallelo delle pagine. In più, se la replica che sta recuperando è necessaria all'avanzamento a causa del fallimento delle altre, il protocollo farà sì che le altre repliche rimangano in attesa del termine del recupero della replica per procedere.

6.2.0.3 State Checking

Il recupero dello stato consente alla replica che sta effettuando la recovery di aggiornare all'ultima versione tutte le singole pagine che possiede, in modo che al termine della procedura la replica possieda una copia dello stato del servizio corretta ed aggiornata.

Si rende anche necessario però controllare la correttezza delle partizioni che compongono lo stato. Per fare ciò la replica procede con il calcolo dei digest di tutte le partizioni di tipo *metadata* all'interno del suo albero, per poi iniziare una procedura di Fetch con apposito messaggio al cui interno il valore specificato di lc sarà -1 così da richiedere alle altre repliche tutte le partizioni. Ogni risposta che viene ricevuta dalla replica viene processata seguendo la procedura già descritta, con la differenza che le partizioni aggiornate non vengono ignorate ma ne viene controllato il digest per confrontarlo con quello memorizzato in precedenza nell'albero. Nel caso il controllo abbia esito negativo la partizione viene accodata per una nuova fetch, altrimenti viene accodata per il controllo. Il controllo di una partizione avviene controllando i digest di tutte le sue pagine con i corrispondenti all'interno dell'albero, accodando per il recupero tutte quelle pagine il cui valore non combacia.

7 Libreria PBFT

L'algoritmo può essere implementato come una libreria che utilizza un modello di comunicazione *connectionless* tra i nodi usando *UDP over IP*. Repliche e client aderiscono ad una semplice interfaccia, di seguito riportata in linguaggio C++, mediante un approccio single-threaded event-driven in cui ogni messaggio ricevuto ed ogni timer del sistema vengono gestiti da uno specifico handler.

```
Client:
int Byz_init_client(char *conf);
int Byz_invoke(Byz_req *req, Byz_rep *rep, bool ro);

Server:
int Byz_init_replica(char *conf, char *mem, int size, proc exec, proc nondet);
int Byz_modify(char *mod, int size);

Server upcalls:
int execute(Byz_req *req, Byz_rep *rep, Byz_buffer *ndet, int cid, bool ro);
int nondet(Seqno seqno, Byz_req *req, Byz_buffer *ndet);
```

7.1 Client

Lato client la libreria offre una procedura di inizializzazione mediante file di testo in cui sono contenute le chiavi pubbliche iniziali e gli indirizzi IP delle repliche. La procedura *invoke* invece viene richiamata ogni volta che un'operazione deve essere eseguita a seguito della risposta da parte di sufficienti repliche ad una richiesta.

7.2 Server

L'inizializzazione del server è sempre basata sulla lettura di un file di configurazione, ma sono richiesti inoltre anche (i) la regione di memoria in cui verrà memorizzato lo stato, (ii) il riferimento ad una procedura per l'esecuzione delle richieste e (iii) il riferimento ad una procedura per l'esecuzione di scelte non deterministiche. La procedura *execute* viene richiamata ogni volta che una replica ha necessità di eseguire un'operazione, specificando (a) un buffer contenente l'operazione richiesta, (b) i relativi argomenti, (c) un buffer in cui verrà memorizzato il risultato dell'operazione e (d) un flag per l'esecuzione mediante ottimizzazione read-only. Questo flag può essere usato per effettuare un primo filtro sulle richieste, scartando quelle che si presentano come read-only ma che in verità modificano lo stato del servizio.

Ogni volta che una replica deve modificare lo stato del servizio, una chiamata alla procedura *modify* consente di informare la libreria riguardo all'area di memoria che viene modificata.

8 Proof of Concept del Sistema

In questa sezione verranno descritti gli approcci strutturali e implementativi utilizzati nella realizzazione del proof of concept del sistema. Considerato il limitato ammontare di ore assegnate per la realizzazione del progetto, unita alla massiccia quantità di features da replicare si è optato per la realizzazione di un modello abbastanza basilare, replicando l'interazione distribuita di client e repliche per gestire lo stato del servizio rappresentato da un valore intero.

Il sistema verrà realizzato facendo uso del linguaggio **Java** e reso eseguibile mediante un'interfaccia a *riga di comando*. Per la build automation del sistema e per la gestione delle dipendenze di progetto verrà utilizzato **Gradle**, mentre per la gestione delle versioni si è fatto uso di **Git**.

Sempre a causa del limitato tempo a disposizione si è optato per garantire più funzionalità possibili al sistema a discapito, per esempio, del deployment dell'applicazione che non beneficia allo stato attuale di facilitazioni ma deve avvenire interamente in modo manuale.

8.1 Features del servizio

Viste le limitazioni temporali dell'intero progetto, le numerose funzionalità presentate dall'algoritmo PBFT sono state solo in parte riportate in questo *proof of concept*. Nello specifico tutto il *core* dell'algoritmo è stato replicato, mentre alcune funzioni aggiuntive o marginali sono state adattate o escluse.

Fanno parte dell'attuale versione del sistema:

- Protocollo di scambio in 3 fasi PRE-PREPARE, PREPARE, COMMIT.
- Gestione dei punti di controllo mediante CHECKPOINT.
- Simulazione di Byzantine Fault nelle repliche.
- Cambiamento della vista a fronte di un primary corrotto, mediante procedura di View Change e creazione della nuova vista mediante messaggi di NEWVIEW.

Non sono incluse nell'attuale versione del sistema:

- Procedura di PROACTIVE RECOVERY.
- Procedura di STATE TRANSFER e procedura di STATE CHECKING.
- Procedura di KEY EXCHANGE.

8.2 Rilassamenti

Le limitazioni temporali del progetto hanno avuto un influenza anche sull'approccio verso alcuni vincoli e proprietà del sistema.

- **High and Low Watermarks:** i parametri H e h che delineano una finestra di accettazione per i sequence number, vengono determinati in maniera semplificata definendo *h = LAST-CHECKPOINT* e *H = NEXT-CHECKPOINT*, riducendo quindi la larghezza della finestra da L, pari alla dimensione dei log, a K, dimensione dell'intervallo tra i checkpoint.
- **Corruzione di repliche:** vista la natura esemplificativa di questo proof-of-concept si decide di optare per una gestione semplificata del tema corruzione delle repliche. Sarà possibile, come descritto in seguito, corrompere dall'esterno una replica, simulando un intrusore esterno che possa aver compromesso i messaggi. L'effetto di una corruzione avrà però un effetto rilevante solo sulla replica con il ruolo di primary nell'attuale vista provocando l'assegnazione di sequence number errati per le nuove richieste. Questo comportamento non canonico consente alle repliche di backup di accorgersi della corruzione del primary e di innescare un processo di View Change.
- **Gestione della View Change:** in questa versione esemplificativa del sistema si opta per una gestione dedicata dei messaggi per la procedura di View Change. Nello specifico, le repliche che avviano la procedura entrano in una fase detta VIEWCHANGE_PHASE durante la quale potranno accettare solo messaggi di tipologia VIEWCHANGE, VIEWCHANGE_ACK E NEWVIEW, mettendo in coda tutti gli altri.

8.3 Design del sistema

Ogni entità che prende parte all'interazione nel sistema incorpora una logica applicativa propria, dunque si è resa necessaria la suddivisione in più componenti:

- **Common:** componente che fornisce utilità comuni a tutte le altre componenti in gioco.
- **PBFT:** componente che incapsula al suo interno le funzionalità del protocollo PBFT utilizzate sia dai Client che dai Server.
- **Client:** entità dedicata all'invio di richieste verso i server del sistema al fine di ottenere un risultato.
- **Server:** entità con il compito di processare le richieste inviate dai client, adottando politiche di consenso.
- **SoB:** entità utile ad innescare eventi esterni al protocollo, ad esempio la *corruzione di una replica*.

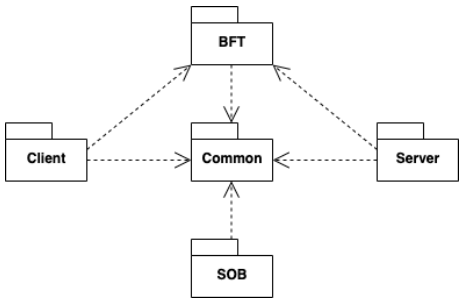

Figura 5: Rappresentazione delle relazioni fra i vari package

8.3.1 Componente Common

Questo componente rappresenta un'entità centrale a cui si appoggiano tutte le altre componenti e che fornisce classi di utility comuni indispensabili per la configurazione e l'avanzamento delle altre entità, oltre a mettere a disposizione un sistema di log a video comune.

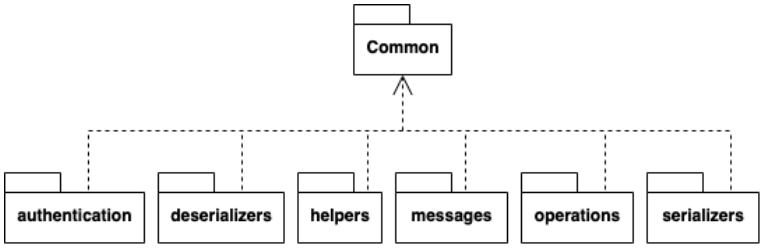

Figura 6: Struttura del package Common.

8.3.1.1 Authentication

Il sotto-package authentication contiene l'interfaccia **IAuthenticator** e la classe **Authenticator** che permette a client e server di autenticare i messaggi computandone codici MAC, codici digest e verificandone la validità.

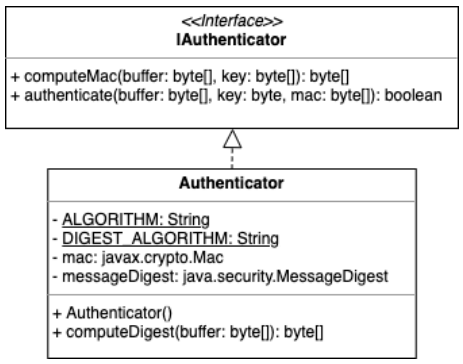

Figura 7: Struttura dei sotto-package authenticator.

8.3.1.2 Helpers

Questo sotto-package contiene una serie di classi che vengono utilizzate dalle varie componenti in ogni fase dell'esecuzione e che forniscono funzioni di utility, di configurazione o strutture necessarie alla comunicazione e all'avanzamento:

- **Pair** rappresenta una classe generica per la gestione di una coppia di valori con logica *key-value*.
- **Tuple** rappresenta una classe generica per la gestione di tre valori.

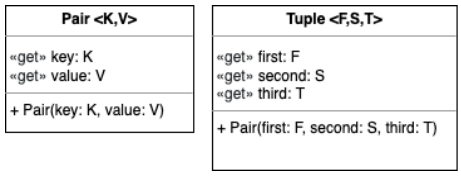

Figura 8: Pair e Tuple.

- **CommonHelper** rappresenta l'utility principale e fornisce funzionalità per la configurazione, per la comunicazione, mediante le funzioni di *send* e *receive*, e per la serializzazione di alcune componenti, oltre ad offrire una funzionalità di log a video.

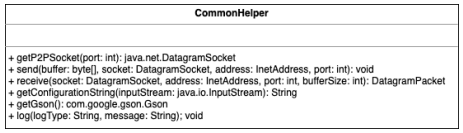

Figura 9: Classe CommonHelper.

- **Configuration** è la classe generica di configurazione che viene utilizzata da ogni entità per memorizzare le informazioni lette dal file di configurazione. Al suo interno sono memorizzati gli indirizzi, le porte e le chiavi per la comunicazione tra client e server, informazioni generiche come la *dimensione del buffer* e altre informazioni specifiche per i server, ad esempio il *checkpointInterval* che regola la frequenza di generazione dei punti di controllo.
- **ClientConfiguration** è la classe di configurazione che viene utilizzata dal client a seguito della lettura delle configurazioni da file e della memorizzazione di quest'ultime nell'oggetto **Configuration**, al fine di mantenere solo le informazioni ad esso strettamente necessarie. Mediante questa classe viene anche gestita la durata del timer per la ritrasmissione dei messaggi, con durata iniziale pari al valore *DEFAULT_TIMEOUT_MILLIS* che nel corso dell'esecuzione può aumentare ed essere ripristinato.

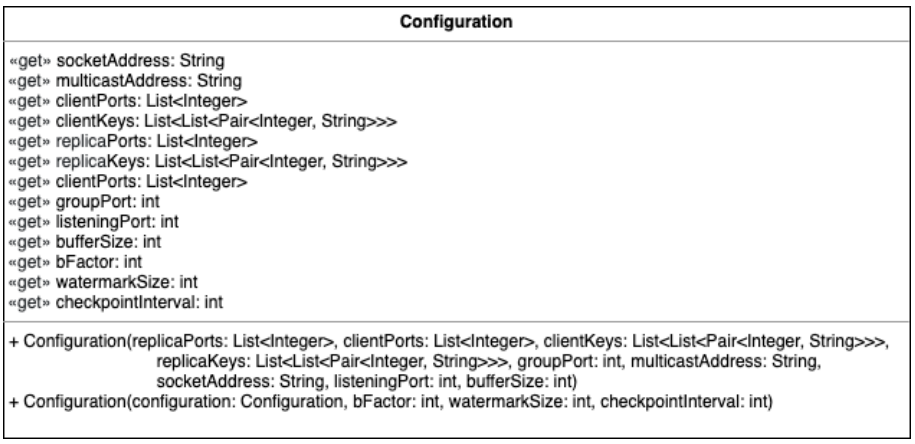

Figura 10: Classe Configuration.

- **ServerConfiguration** è la classe che rappresenta il duale lato server di **ClientConfiguration**. Nel caso del server le informazioni necessarie sono maggiori rispetto al client, visto che serviranno anche tutte le chiavi relative ai client attivi ed informazioni più specifiche come il *checkpointInterval*.

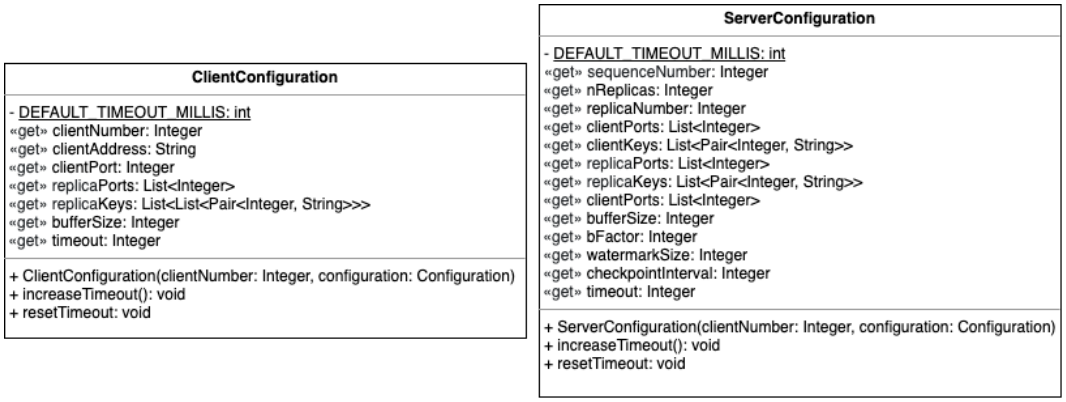

Figura 11: Classi ClientConfiguration e ServerConfiguration.

- **SobMessage** rappresenta un messaggio inviato da una entità SOB, e che deve essere interpretabile sia dalle entità client che dalle entità server. Ogni messaggio di questo tipo contiene un **SobType** che ne denota la tipologia e un *entityNumber* che specifica il numero dell'entità a cui è indirizzato il messaggio.
- **SobType** rappresenta la tipologia di un messaggio inviato dal SOB. Ogni tipologia, se interpretabile dall'entità destinataria, innescherà una specifica operazione.

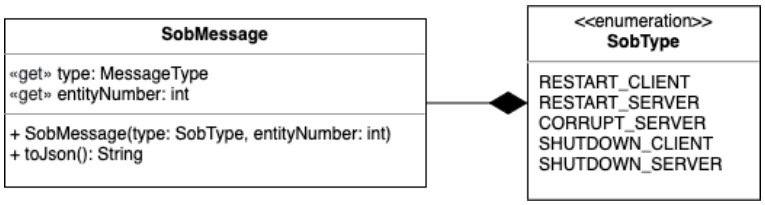

Figura 12: SobMessage e SobType.

8.3.1.3 Messages

Il sotto-package **messages** contiene le classi che modellano tutte le tipologie di messaggi che vengono scambiati tra client e server durante l'esecuzione. Ogni messaggio è composto da un **Mac**, da un **MessageType** che ne denota la tipologia e un **MessageContent** che ne rappresenta il contenuto, definito nello specifico dalle singole classi.

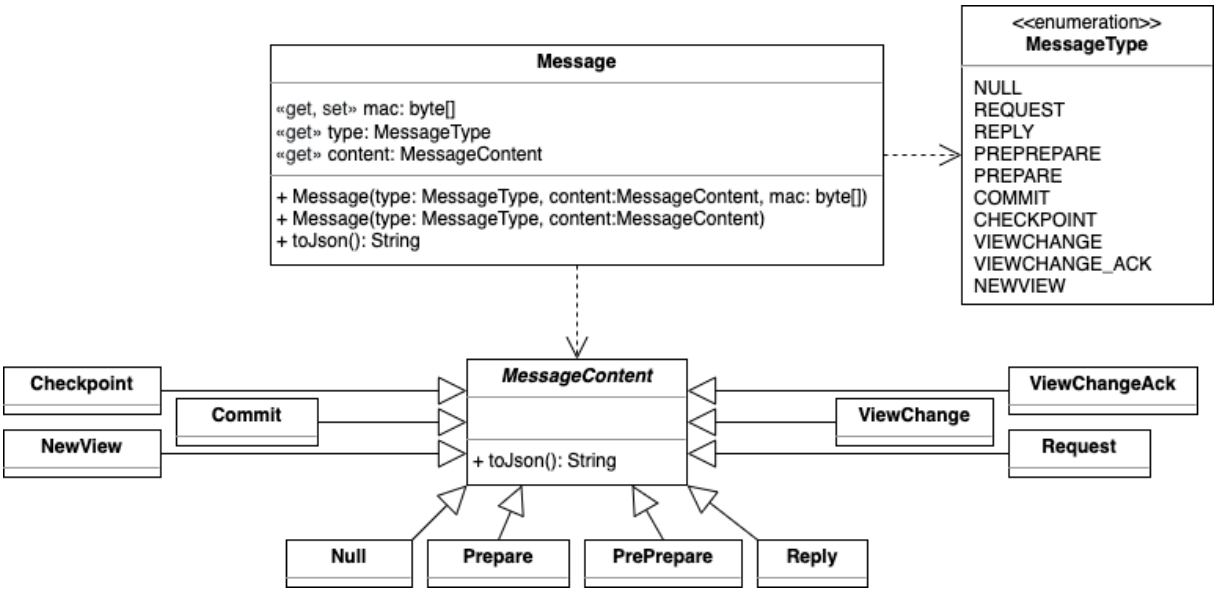

Figura 13: Struttura del sotto-package messages.

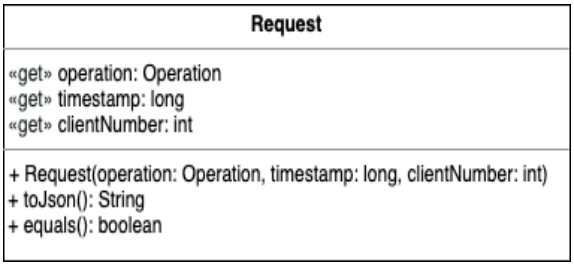

Figura 14: Request

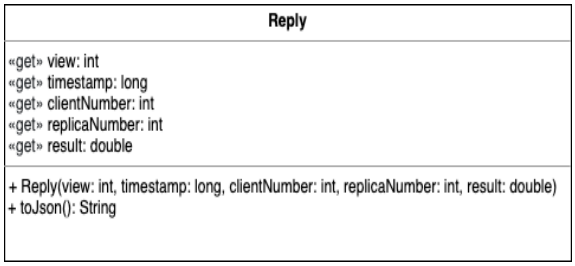

Figura 15: Reply

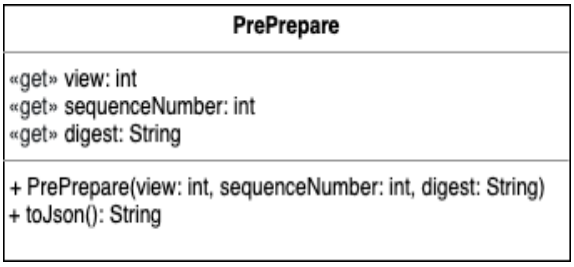

Figura 16: PrePrepare

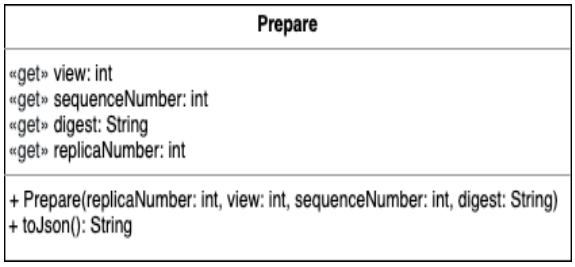

Figura 17: Prepare

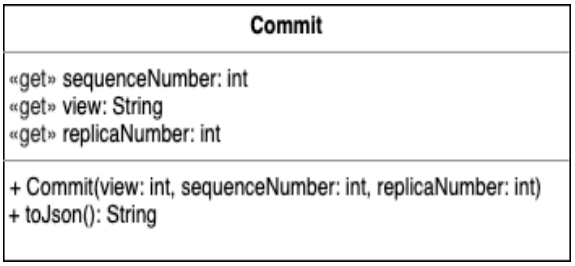

Figura 18: Commit

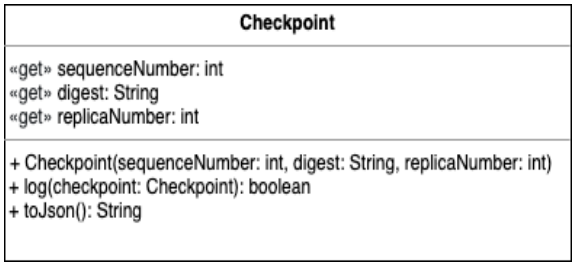

Figura 19: Checkpoint

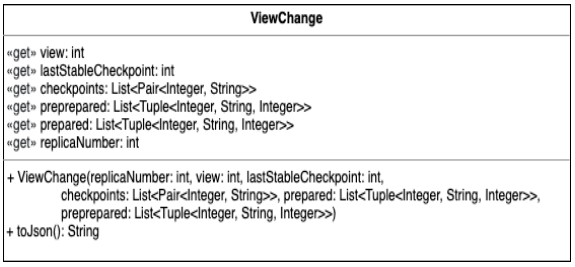

Figura 20: ViewChange

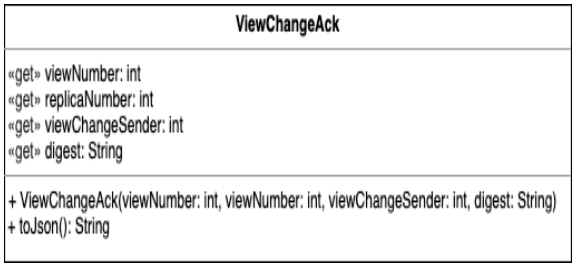

Figura 21: ViewChangeAck

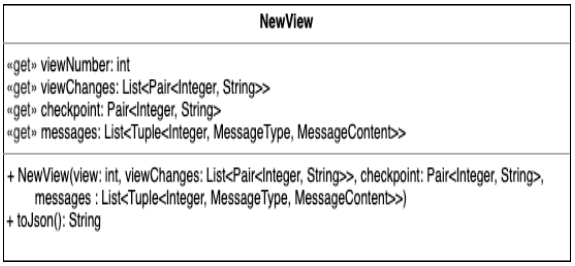

Figura 22: NewView

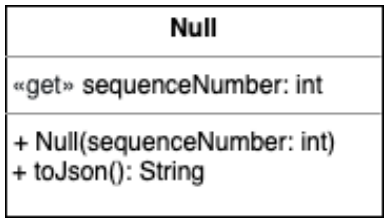

Figura 23: Null

8.3.1.4 Operations

Il sotto-package **operations** contiene la classe che definisce la struttura di un'operazione che il client può chiedere di eseguire alle repliche. Ogni operazione contiene quindi una **OperationType** che ne denota la tipologia e un **operand** che rappresenta il secondo operando dell'operazione, assumendo che il primo operando sia lo *stato* della replica.

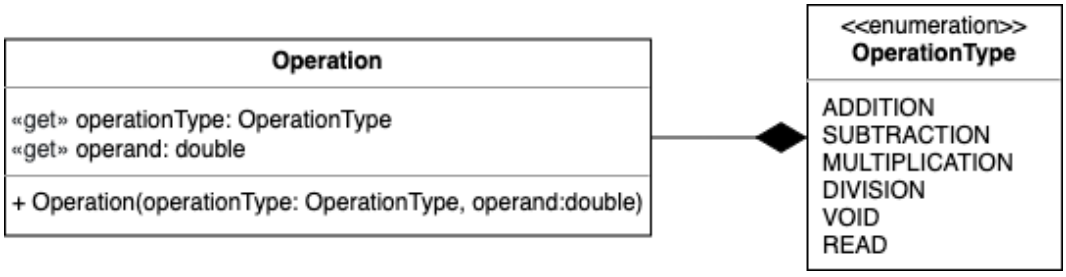

Figura 24: Struttura del sotto-package operations.

8.3.1.5 Serializers e Deserializers

Nei sotto-package **serializers** e **deserializers** sono presenti le classi che offrono le funzioni necessarie alla serializzazione e deserializzazione dei messaggi prima dell'invio e dopo la ricezione.

Il formato adottato per la comunicazione è il **Json**, perciò è importante che sia in fase di invio che in fase di ricezione i dati inviati aderiscano allo standard.

Le classi di serializzazione permettono conversione personalizzata degli oggetti a runtime in oggetti Json, che vengono poi convertiti in stringhe al momento dell'invio. Al contrario, le classi di deserializzazione permettono l'interpretazione di una stringa ricevuta in oggetti Json, con cui vengono costruiti gli oggetti che rappresentano i messaggi.

Questo approccio permette conversioni standard, utili soprattutto per oggetti che riportano al loro interno strutture dati non banali come liste o altri oggetti composti.

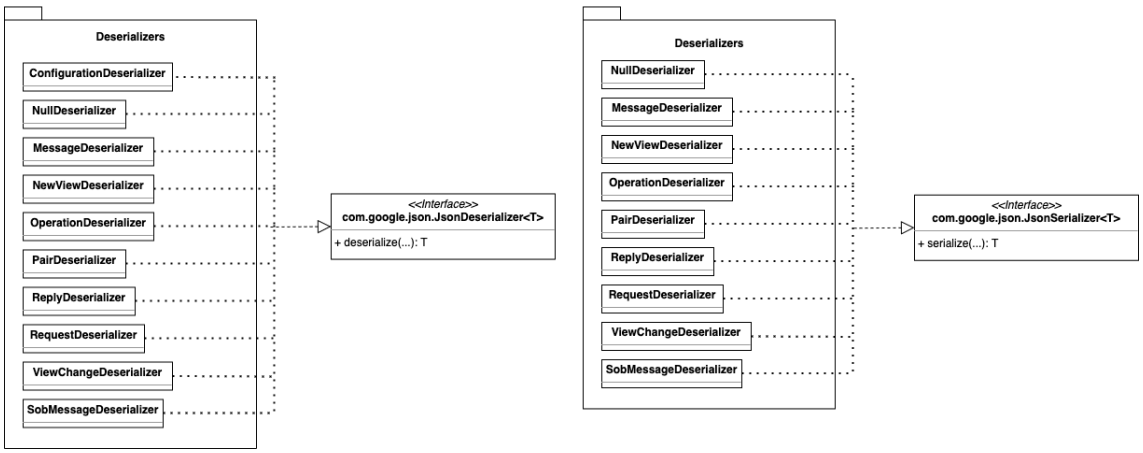

Figura 25: Struttura dei sotto-package serializer e deserializer.

8.3.2 Componente PBFT

Questo componente rappresenta il core del sistema in quanto contiene al suo interno la logica dell'algoritmo e per questo viene utilizzato sia dai Client che dai Server. L'interfaccia **PBFT** rappresenta una riproduzione dell'interfaccia presentata dal protocollo nella sezione precedente. Da essa derivano le due classi effettive **BFTClient** e **BFTServer**, mediante le quali rappresenta un'adattamento della libreria PBFT. La procedura di inizializzazione **ByzInit** è pressochè simile sia per Client che per Server e necessita, oltre del progressivo che identifica la specifica istanza client o server in esecuzione, anche del riferimento al nome del file di configurazione da cui leggere tutti i parametri, come numeri di porta e chiavi, necessari per l'esecuzione e la comunicazione con le altre entità..

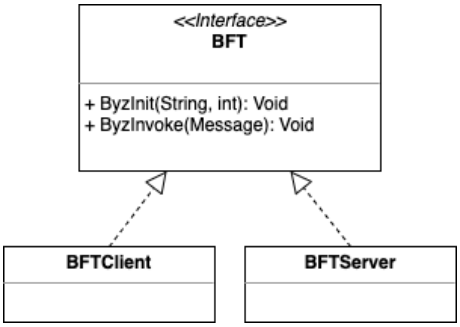

Figura 26: Interfaccia PBFT.

8.3.2.1 BFT-Client

La procedura **ByzInvoke** del client prende come input un messaggio ricevuto mediante socket e, dopo averne controllato la giusta tipologia e la correttezza del MAC, procede con la gestione.

Figura 27: Classe BFTClient

Figura 28: Classe BFTServer

Come previsto dall'algoritmo, la procedura logga il messaggio e, nel caso sia sufficiente per l'erogazione di un certificato, accetta tale valore e modifica il suo stato di conseguenza. All'interno di questa classe saranno presenti anche un **Logger** mediante il quale vengono memorizzate le risposte ottenute dalle repliche per il controllo dei certificati, lo **Status** del client che viene aggiornato in base alle risposte delle repliche, un'**Authenticator** per la creazione e la verifica dei codici MAC e tutte le componenti necessarie, come **ClientConfiguration** utile per la comunicazione mediante socket.

8.3.2.2 BFT-Server

La procedura **ByzInvoke** viene richiamata all'arrivo di un messaggio via socket, il quale può essere messo in coda o gestito mediante apposite funzioni di handling in base allo stato attuale della replica e alla tipologia del messaggio stesso. La classe fa uso anche di un'**Authenticator** per la computazione e il controllo di MAC e digest, un **Logger** per la memorizzazione dei messaggi e la gestione dei certificati, un **CheckpointLogger** per la memorizzazione dei checkpoint prodotti ed un **ViewChangeLogger** utile durante la procedura di ViewChange.

8.3.2.3 ClientStatus

Lo stato di una entità client è rappresentato semplicemente da una copia del valore che rappresenta lo stato del servizio, memorizzato all'interno di **value**. Lo stato di un client può essere aggiornato mediante richieste di operazioni di tipo READ, oppure può modificare lo stato del servizio, richiedendo l'esecuzione di operazioni come ADDITION.

8.3.2.4 ServerStatus

Lo stato di un'entità server viene rappresentato da una coppia di valori.

Il primo, **value**, rappresenta lo stato attuale del servizio all'interno della replica.

Il secondo, **phase**, rappresenta la fase di esecuzione in cui si trova il sistema. L'importanza di questo valore è data dal fatto che in determinati momenti la replica avrà necessità di sospendere la regolare operatività al fine di portare a termine delle procedure.

Come ben comprensibile, la fase *REQUEST_PHASE* modella la fase di ordinaria operatività, in cui la replica può accettare liberamente tutte le richieste che riceve. Una replica passa alla fase *VIEWCHANGE_PHASE* quando sospetta che l'attuale replica primary sia fallata avviando una procedura di View Change. Quando invece la replica necessiterà di recuperare lo stato attuale del servizio passerà nella *FETCH_PHASE*, prevista per sviluppi futuri ma non gestita attualmente.

Al termine delle procedure di View Change o Fetch la replica tornerà nella fase *REQUEST_PHASE* riprendendo la normale attività.

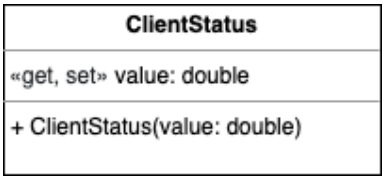

Figura 29: Classe ClientStatus.

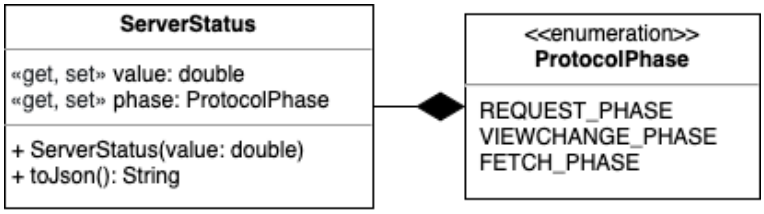

Figura 30: Classe ServerStatus.

8.3.2.5 View

La classe View modella la serie di configurazioni che attraversa ogni singola replica durante l'esecuzione. Questa classe è utile per il processing della gran parte dei messaggi ricevuti, per i quali il controllo sulla vista ne discrimina l'accettazione, oltre che per la procedura di View Change, durante la quale ogni replica deve conoscere chi sarà il prossimo primary.

Il numero della vista in cui si trova la replica in un determinato momento è memorizzato in **currentView**. In ogni momento è possibile conoscere l'attuale primary mediante **getPrimary()**, il primary di una certa view mediante **getPrimaryForView(viewNumber:int)**, oltre che effettuare un passaggio alla configurazione successiva utilizzando **nextView**.

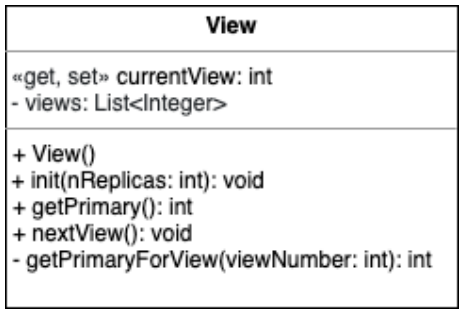

Figura 31: Classe View.

8.3.2.6 Logger

La classe Logger rappresenta la struttura mediante la quale ogni replica memorizza le informazioni sulle richieste ricevute, sui messaggi di consenso scambiati e sulle risposte inviate ai client.

La classe di base è **Log**, in cui per ogni richiesta vengono memorizzati il *sequence number* assegnato, la *view* a cui fa riferimento, il *digest* e tutti i messaggi PrePrepare, Prepare, Commit e Reply inviati o ricevuti e riferiti alla richiesta. Viene inoltre tracciata l'informazione riguardo allo stato *prepared*, *committed* e *executed* della richiesta.

Un generico Logger possiede quindi un elenco di Log relativi alle richieste ricevute e al loro avanzamento, permettendo quindi ad una replica di memorizzare i messaggi tramite le rispettive

funzioni **log()** e verificare le informazioni necessarie come, ad esempio, l'emissione dei certificati di Prepare e di Commit.

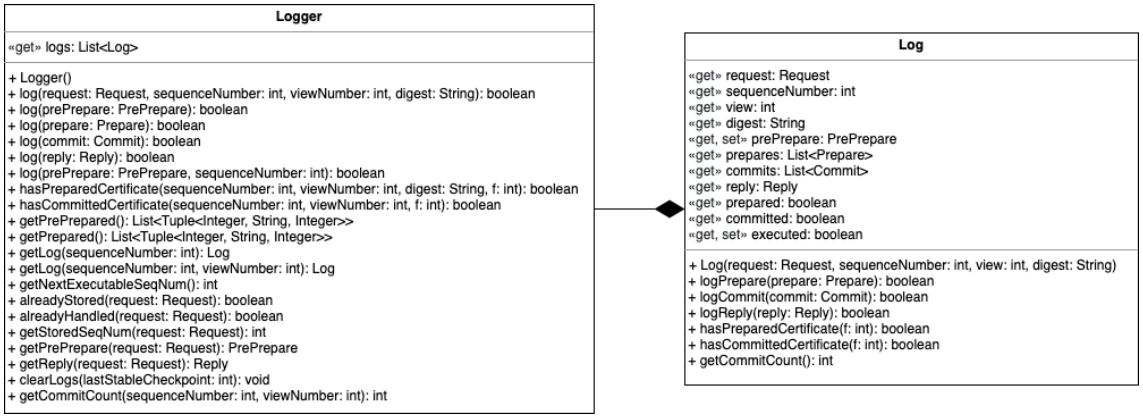

Figura 32: Classe Logger.

8.3.2.7 CheckpointLogger

Questa classe svolge un compito simile al generico Logger, ma ha un uso più specifico e limitato alla memorizzazione dei messaggi di Checkpoint che vengono scambiati dalle repliche e al loro stato.

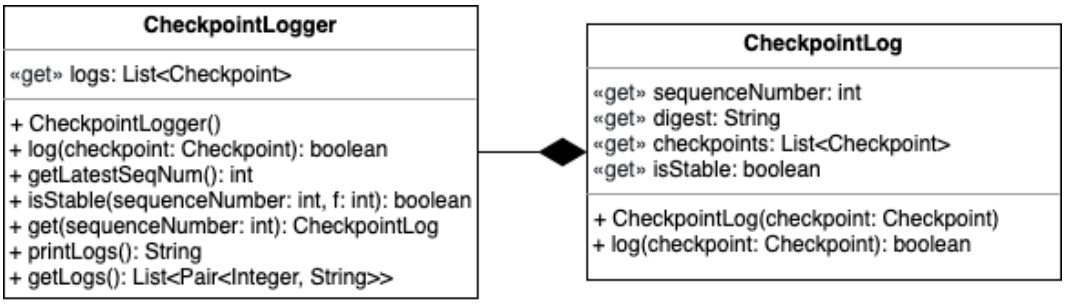

Figura 33: Classe CheckpointLogger.

La classe di base è il **CheckpointLog**, rappresentato dal *sequence number* a cui fa riferimento, dal proprio *digest*, dai relativi messaggi di Checkpoint e dal flag *isStable* che ne indica la stabilità.

Ogni CheckpointLogger contiene un elenco di CheckpointLog, in cui ogni elemento corrisponde ad un preciso Checkpoint del sistema, e permette di memorizzare i messaggi ricevuti mediante la funzione **log**, verificarne la stabilità e visualizzarne il contenuto.

8.3.2.8 ViewChangeLogger

Questa classe svolge un compito simile al generico Logger, ma ha un uso più specifico e limitato alla memorizzazione dei messaggi che vengono scambiati dalle repliche durante il processo di View Change. Le classi su cui si basa sono i messaggi **ViewChange**, **ViewChangeAck** e **NewView** già descritti.

Un ViewChangeLogger fa riferimento ad una singola procedura di ViewChange, memorizzando al proprio interno i messaggi di ViewChange, il set contenente i messaggi *confermati*, e tutti i messaggi di *ViewChangeAck* e *NewView* ricevuti. Il ViewChangeLogger permette di memorizzare tutti i messaggi sopra descritti mediante le funzioni di **log**, oltre a consentire di verificare l'avanzamento della procedura di ViewChange mediante funzioni specifiche come, ad esempio, **hasCertificate** che permette di verificare l'emissione di un certificato.

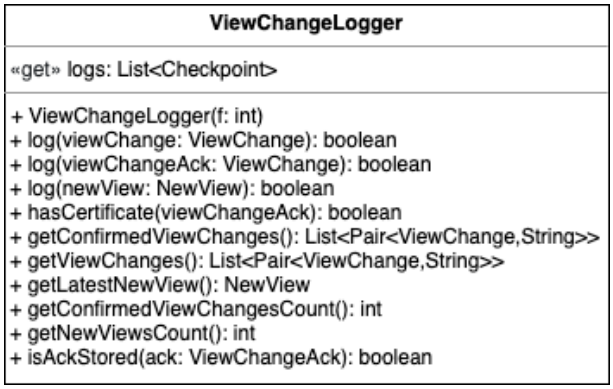

Figura 34: Classe ViewChangeLogger.

8.4 Client

Una generica entità client viene rappresentata da una classe la cui esecuzione attraversa quattro fasi principali. All'avvio, il client deve reperire le informazioni di configurazione leggendole da uno specifico file *configuration.json* ed inizializzare l'oggeto *BFTClient*. Specificando l'intero i che identifica l'entità e l'oggetto *ClientConfiguration* costruito mediante lettura del file, vengono inizializzate tutte le strutture necessarie alla comunicazione e all'avanzamento del protocollo.

Terminata la fase di configurazione, la seconda fase vede il client richiedere dei parametri all'utente al fine di creare una richiesta da inviare alle repliche. Possono essere create richieste per le 5 operazioni supportate dalle repliche, ovvero *ADDITION*, *SUBTRACTION*, *DIVISION*, *MULTIPLICATION* e *READ*. Per le 4 operazioni la cui esecuzione potrebbe provocare una modifica dello stato del servizio, viene richiesto anche un valore che fungerà da operando di destra nell'operazione sullo stato. I client in questione non sono pensati come entità proattive, per cui nessuna richiesta verrà inviata alle repliche senza una interazione dell'utente con il client.

Quando questo avviene, il client entra in possesso di una *Request* e passa alla fase di invio, facendo uso della *sendRequest* di *BFTClient*.

A questo punto il client entra nella fase di attesa, in cui deve solo attendere sufficienti risposte delle repliche per la sua richiesta e memorizzare il risultato corretto dell'operazione. Ogni messaggio che viene ricevuto deve essere processato da *BFTClient* verificando di volta in volta se le risposte possono far parte di un certificato che attesti la correttezza del risultato. L'attesa delle risposte non è infinita ma, come specificato da PBFT, ogni client possiede un timer interno alla cui scadenza viene effettuata una ulteriore trasmissione della richiesta alle repliche, tale e quale all'originale. Questo viene effettuato mediante un *ClientThread* del quale è possibile verificare l'avanzamento e attendere la teminazione.

Quando le risposte ricevute compongono un certificato, il client termina la propria attesa e torna alla fase di creazione della richiesta.

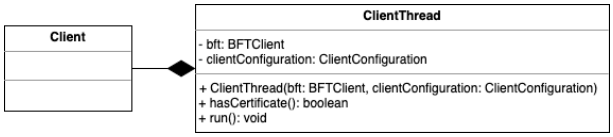

Figura 35: Classi Client e ClientThread.

8.5 Server

La classe Server rappresenta l'entità del sistema che, dopo una prima fase di configurazione, attende passivamente l'arrivo di messaggi dai client o dalle altre repliche.

Come detto, anche i server attraversano fase iniziale di configurazione nella quale, parallelamente ai client, leggono le informazioni memorizzate all'interno di un file *configuration.json* per costruire un *ServerConfiguration*.

Al termine della configurazione, il server si mette in attesa che qualche qualche entità client o server gli invii dei messaggi. I messaggi ricevuti da una replica che provengono dai client o da altre repliche, vengono processati dall'oggetto BFTServer, il quale li gestirà in base alla fase di avanzamento del sistema o alla tipologia del messaggio.

Solo i messaggi che la replica riceve dalle entità SOB sono gestiti direttamente dall'entità Client, senza essere tramessi a BFTServer in quanto non devono intaccare il flusso dei messaggi scambiati per l'esecuzione del protocollo PBFT, anche se potrebbero influire sul comportamento del sistema. Le tipologie di messaggio che un server può ricevere dal SOB sono 2: *RESTART_SERVER* per il riavvio forzato della replica e *CORRUPT_SERVER* per renderla fallata e fargli inviare messaggi non corretti.

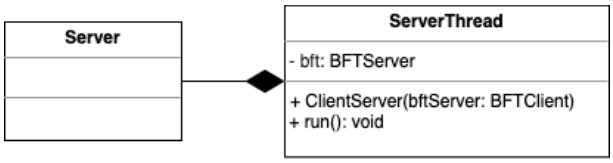

Figura 36: Classi Server e ServerThread.

8.6 SOB

Il componente SOB, nome che rappresenta un acronimo di **Simulation OBject**, rappresenta un'entità del sistema esterna al protocollo utile ad innescare eventi che interessano le componenti Client e Server. Questo componente è rappresentato da una semplicissima classe con funzionamento similare al Client con la differenza che, rappresentando appunto una classe completamente autonoma e distaccata dal resto (soprattutto dal protocollo PBFT), permetterà solo l'invio di messaggi mirati a client o repliche senza attendere risposte o innescare ritrasmissioni di messaggi. Anche questa entità effettua, come prima cosa nella sua esecuzione, la lettura da file delle informazioni di configurazione per poter comunicare con tutte le altre entità.

Successivamente è l'utente che dovrà immettere comandi per poter innescare l'invio di messaggi a client e repliche: le operazioni consentite sono *RESTART_CLIENT*, *RESTART_SERVER*, *CORRUPT_SERVER*, *SHUTDOWN_CLIENT* e *SHUTDOWN_SERVER*.

8.7 Comunicazione tra entità

In modo parallelo a quanto descritto dall'algoritmo PBFT, Client e Server attivi del sistema devono comunicare tra loro in modalità *point-to-point* o *multicast*. In entrambi i casi la comunicazione viene gestita mediante Socket UDP, le quali permettono di immettere in rete pacchetti indipendenti senza garanzia di ordinamento e di consegna. Siccome l'algoritmo è già progettato per far fronte all'instabilità della rete e alla perdita o al mancato ordinamento dei messaggi, UDP rappresenta una scelta calzante che permette inoltre di risparmiare eventuali overhead dovuti ad un approccio TCP. Allo stato attuale il sistema è costruito per supportare al massimo 7 client e 7 repliche e per essere eseguito completamente in locale, perciò per la comunicazione è sufficiente far utilizzare ad ogni entità attiva una porta differente della macchina, anche se è possibile rendere il sistema completamente distributo effettuando poche semplici modifiche.

Client	1	2	3	4	5	6	7
Porta	4450	4451	4452	4453	4454	4455	4456

Tabella 1: Mappatura porte delle entità Client.

Server	1	2	3	4	5	6	7
Porta	4470	4471	4472	4473	4474	4475	4476

Tabella 2: Mappatura porte delle entità Server.

8.7.1 Comunicazioni unicast e multicast

L'approccio utilizzato per l'implementazione delle trasmissioni multicast non segue dettagliatamente quanto definito nell'algoritmo. PBFT include all'interno di un messaggio multicast un *authenticator*, ovvero un array contentente per ogni destinatario j il corrispondente MAC, delegando all'entità che lo riceve il recupero della entry corretta dall'array per effettuare il controllo di autenticità. Nell'approccio proposto, per praticità, non si effettua un invio multicast nel senso proprio del termine, ma piuttosto viene inviato un insieme di messaggi point-to-point ad ogni singolo destinatario con incluso lo specifico MAC.

9 Dettagli implementativi

In questa sezione verrà analizzato più nello specifico il comportamento di alcune classi o di porzioni di codice.

9.1 Client e ClientThread

Come accennato nella sezione precedente la classe *Client* utilizza al suo interno una classe *ClientThread* per la gestione dell'attesa delle risposte. La scelta di realizzare questa fase con un thread di esecuzione autonomo deriva dalla necessità di monitorare l'avanzamento ed, eventualmente, effettuare ritrasmissioni di messaggi alla scadenza del timer t. Dopo l'invio del messaggio di REQUEST, quindi, il main thread della classe *Client* crea e avvia una nuova istanza di *ClientThread*, per poi attenderne la terminazione per un tempo pari alla durata del timer. Al suo interno, la classe *ClientThread* si mette in attesa di messaggi di REPLY sulla propria socket mediante una *receive* fino a che l'insieme dei messaggi ricevuti non costituisce un certificato valido ad attestare la correttezza del risultato.

A questo punto l'esecuzione potrebbe avere due esiti differenti:

- I messaggi ricevuti sono sufficienti ad ottenere un certificato, quindi il thread prosegue terminando la propria esecuzione in un tempo inferiore a t.
- Trascorso un tempo t, il numero di messaggi processati non è sufficiente ad ottenere un certificato, quindi il *Client* riprende la sua esecuzione e interrompe il thread.

In entrambe le situazioni, quando il controllo dell'esecuzione torna al main thread nella classe *Client*, questa dovrà controllare se il certificato è stato ottenuto. In caso negativo deve essere incrementato il valore del timer per effettuare una ritrasmissione del messaggio, mentre nel caso positivo il timer dovrà essere in ogni caso resettato e il client potrà rimettersi in attesa dell'input dell'utente per la creazione della richiesta successiva.

9.1.1 BFTClient

Ogni messaggio ricevuto dalle socket effettuando una chiamata bloccante alla *receive* viene gestito da un oggetto *BFTClient* mediante la *byzInvoke* che accetta oggetti di tipo *Message*, effettuando al suo interno i controlli previsti dall'algoritmo: (1) il messaggio deve essere di tipo REPLY, (2) deve essere autenticato, (3) con corretto timestamp e (4) il *clientNumber* contenuto all'interno deve corrispondere all'identificativo del client. Il messaggio può essere ignorato se una delle precedenti condizioni non è rispettata oppure se è già stato ottenuto un certificato, quindi in questi casi la procedura di gestione termina. Nel caso in cui venga accettato, il messaggio deve essere loggato e, se il certificato è ottenuto, lo stato deve essere modificato con il risultato.

9.2 Server e ServerThread

A differenza del Client, che utilizza un thread per la sola parte di attesa e controllo dei messaggi, il *Server* demanda tutte le attività inerenti il protocollo ad un *ServerThread* autonomo. Questa scelta implementativa è stata adottata al fine di porre le basi per l'implementazione del meccanismo di PROACTIVE RECOVERY, attualmente mancante. Il *Server* che avvia un *ServerThread* infatti resta attualmente in attesa infinita della terminazione del thread, ma impostando un'attesa limitata sarebbe possibile interromperlo per poi riavviarlo ripristinando lo stato. Inoltre, a seguito del riavvio, potrebbe essere implementata anche la procedura di Fetch dello stato, anch'essa attualmente mancante.

Il *ServerThread* ha un comportamento iterativo che prevede l'attesa di un messaggio e la sua gestione. La sua esecuzione inizia quindi mettendosi in attesa di ricevere messaggi effettuando una chiamata alla funzione *receive*. Come accennato nelle sezioni precedenti, la replica potrebbe ricevere messaggi di tipo *Message* oppure di tipo *SOBMessage*.

- **Message**: quando la replica riceve un messaggio di questo tipo ne affida la gestione ad un oggetto *BFTServer* mediante chiamata al suo metodo *byzInvoke* che lo gestirà in base al suo *MessageType* effettuando i dovuti controlli.

- **SOBMessage**: i messaggi di questa tipologia servono ad innescare azioni specifiche (riavvio o corruzione) sul server. Questi messaggi non prendono parte all'avanzamento del protocollo, per cui vengono gestiti direttamente dal *Server* che esegue l'operazione richiesta nel momento esatto in cui il messaggio viene ricevuto.

Lo stato di un BFTServer potrebbe temporanemente rendere alcuni messaggi non processabili, i quali vengono salvati in un coda interna che viene interrogata dal *ServerThread*. La coda sarà sicuramente vuota alla prima iterazione, quindi il flusso si fermerà sulla *receive*.

È comunque prudente controllare come prima cosa la presenza di messaggi accodati siccome, in caso di riavvio, potrebbero essere presenti messaggi non ancora processati dalla precedente esecuzione che rimarrebbero bloccati se il flusso si fermasse continuamente in attesa di messaggi dalle socket. Questa soluzione instaura una priorità ai messaggi in coda nei confronti di quelli in arrivo dalla socket, dotata comunque di una propria coda interna di messaggi. La definizione di questa priorità consente inoltre di stabilire un ordinamento dei messaggi: considerando l'arrivo di un messaggio nella socket a tempo t, se sono già presenti messaggi in coda significa che questi hanno raggiunto la replica in un momento $t' < t$.

9.2.1 BFTServer

Come accennato in precedenza, il comportamento di un *BFTServer* è dipendente dal tipo di messaggio che di volta in volta riceve. Siccome in PBFT ogni messaggio ha una gestione diversa dalle altre, anche in questo caso in base al *MessageType* del *Message* viene eseguito un handler interno specifico per quella tipologia. Alcuni handler condividono quasi interamente una fase iniziale di controllo sul messaggio per verificarne correttezza e autenticità.

9.2.1.1 handleRequest

La gestione di una REQUEST ricalca in pieno le funzionalità descritte dall'algoritmo PBFT, effettuando una verifica iniziale sull'autenticazione (mediante mac) ed una esecuzione differenziata per primary e backup. Il primary ha l'onere di assegnare il sequence number, che sarà 0 in caso di corruzione, ed inviare le PRE-PREPARE (anche in caso di ritrasmissione). Una replica di backup, invece, dovrà solo loggare il messaggio o effettuare una ritrasmissione del messaggio di REPLY corrispondente.

9.2.1.2 handlePrePrepare

Anche la gestione di una PRE-PREPARE è fedele all'algoritmo PBFT ma in questo caso, dopo i normali controlli, viene gestita l'eventualità che il sequence number sia pari a 0. In questo caso la replica invia un messaggio di VIEWCHANGE alle altre ed entra nella fase VIEWCHANGE_PHASE. Nel caso contrario controlla se la REQUEST relativa è contenuta nei log al fine di inviare le PREPARE alle altre repliche.

9.2.1.3 handlePrepare

Un messaggio di PREPARE viene inizialmente i controllato per quanto concerne autenticazione, numero di vista e sequence number, dopodichè, in caso di esito positivo, verrà inserito nei log. Se il messaggio va a comporre un certificato di prepare, la replica è autorizzata ad inviare i COMMIT alle altre repliche, altrimenti l'handler termina.

9.2.1.4 handleCommit

Un messaggio di COMMIT, alla pari di quelli di PRE-PREPARE e PREPARE, viene inizialmente controllato e loggato in caso di esito positivo. Se il messaggio compone un certificato di commit, deve essere anche verificato che la relativa richiesta possa essere eseguita, ovvero che siano eseguite tutte le richieste con sequence number inferiore. In tal caso viene recuperata dai log la Richiesta e l'operazione in essa contenuta viene eseguita aggiornando lo stato di conseguenza, dopodichè viene inviata la REPLY con il risultato al client.

In seguito è necessario anche verificare la possibilità di produrre un checkpoint, semplicemente controllando se il sequence number della richiesta sia divisibile per l'intervallo K. In caso affermativo la replica invierà alle altre un messaggio di CHECKPOINT. La versione attuale del sistema genera il primo checkpoint in corrispondenza del messaggio con sequence number 1, mentre per i successivi rispetta comunque l'intervallo K per il quale viene utilizzato un valore abbastanza piccolo (es. 5).

9.2.1.5 handleCheckpoint

Un CHECKPOINT viene controllato solo per quanto riguarda la sua autenticazione, loggandolo in caso di esito positivo. Nel momento in cui il messaggio compone un certificato, la replica procede con la pulizia dei propri log per i sequence number inferiori a quello registrato come checkpoint.

9.2.1.6 handleViewChange

Una replica che riceve una VIEWCHANGE deve innanzitutto verificarne l'autenticazione, dopodichè dovrà aggiornare la sua vista attuale impostandola pari al numero comunicato nel messaggio. A questo punto il messaggio può essere accettato solo se il nuovo numero di vista è maggiore o uguale ad ognuno dei sequence number per cui sono state inviate le PRE-PREPARE e PREPARE contenute nei log. La replica deve entrare nella fase VIEWCHANGE_PHASE e preparare il proprio VIEWCHANGE_ACK da inviare al nuovo primary. Nel caso la replica sia il primary, procede solo con la memorizzazione dell'ack nei propri log.

9.2.1.7 handleViewChangeAck

Una replica che riceve un VIEWCHANGE_ACK deve verificarne l'autenticazione e loggarlo. Poi, nel caso componga un certificato, tentare un esecuzione della *decisionProcedure* cercando di costruire la nuova vista.

9.2.1.8 handleNewView

Alla ricezione di un messaggio di NEWVIEW la replica, dopo averne verificato l'autenticazione, controlla che tutti i messaggi di VIEWCHANGE siano loggati e, in caso affermativo, procede nuovamente all'esecuzione della *decisionProcedure* per verificare se può accettare le informazioni comunicate dal client.

9.3 Gestione dei messaggi

In questa sezione verrannno forniti ulteriori dettagli sulla gestione dei messaggi che una replica invia e riceve e sulla loro memorizzazione sottoforma di log.

9.3.1 Autenticazione

L'autenticazione dei messaggi avviene mediante la classe *Authenticator*, che permette la computazione di mac e digest mediante i metodi *computeMac* e *computeDigest* che utilizzano rispettivamente gli algoritmi SHA-512 e SHA-256. La classe permette inoltre di verificare l'uguaglianza di un mac ricevuto con uno computato al momento mediante il metodo *authenticate*.

9.3.2 Gestione dei log

Un oggetto *Log* deve contenere al suo interno tutte le informazioni relative alla richiesta, compresi tutti i messaggi che la replica ha prodotto o ricevuto per quella richiesta, in modo tale da poter effettuare in ogni momento qualsiasi controllo su stato di avanzamento ed emissione di certificati.

Un *Logger* funge sostanzialmente da contenitore di oggetti di tipo *Log*, permettendo di inserirne di nuovi, ottenere informazioni e aggiornare quelli presenti. Entrando nello specifico dell'inserimento e della modifica, mediante le funzioni di *log* è possibile memorizzare le differenti tipologie di messaggi. Qualora il messaggio sia di tipo *Request*, verrà inserito un nuovo log specificandone sequence number, vista di appartenenza e digest. In tutti gli altri casi (*Pre-Prepare*,

Prepare, *Commit* e *Reply*) il messaggio verrà inserito nell'opportuno log controllandone sequence number, vista e digest.

9.3.3 Gestione dei checkpoint

Un oggetto *CheckpointLog* serve a tenere traccia dello stato di un checkpoint del sistema. Al suo interno devono essere memorizzati tutti i messaggi di tipo *Checkpoint* ricevuti, oltre al sequence number associato e al digest del checkpoint stesso.

Un *CheckpointLogger*, in modo simile ad un *Logger*, funge da contenitore di oggetti di tipo *CheckpointLog* permettendo di inserirne di nuovi, ottenere informazioni e aggiornare quelli presenti. Richiamando la funzione *log* è possibile memorizzare un nuovo *Checkpoint*, ma in questo caso verranno effettuati dei controlli di esitenza. Nel caso sia già presente un *CheckpointLog* per cui sequence number e digest corrispondono a quelli del messaggio da memorizzare, questo viene aggiunto all'insieme dei *Checkpoint* ricevuti per quella occorrenza. Al contrario, se non c'è ancora nessun messaggio per il quale sequence number e digest corrispondono viene creato un nuovo *CheckpointLog*.

9.3.4 Gestione dei log di ViewChange

Il *ViewChangeLogger* viene utilizzato per memorizzare al suo interno tutti i messaggi che le repliche producono o ricevono durante la procedura di View Change. In questo oggetto verranno inseriti, durante l'avanzamento della procedura, messaggi di ViewChange, di ViewChangeAck e di NewView grazie alle apposite funzioni di *log*. Durante le operazioni di *log*, viene inoltre effettuato l'aggiornamento dell'insieme S, fondamentale nella fase di *Decision Procedure*, in cui vengono inseriti tutti i messaggi di ViewChange per i quali la replica ha ottenuto un certificato.

10 Note finali

In questa sezione finale verranno descritti i tool utilizzati per lo sviluppo del progetto, verrà descritta un'esecuzione tipo del sistema e riportate alcune conclusioni rigurardanti tutto il progetto.

10.1 Tool utilizzati

Come accennato, il sistema è stato sviluppato in linguaggio Java facendo uso di **Git** come sistema di controllo delle versioni e **Gradle** per quanto riguarda la build automation e la gestione delle dipendenze.

10.1.1 Git

Il versioning dell'intero progetto è tracciato mediante il software Git e memorizzato in un repository dedicato di GitLab[1]. Il workflow di riferimento per lo sviluppo del sistema è *GitFlow*, anche se la parte realizzativa del progetto sia stata unipersonale e abbia seguito uno sviluppo lineare che non ha richiesto l'utilizzo di branch secondari.

10.1.2 Gradle

Gradle permette una gestione semplificata delle dipendenze nel progetto. All'interno dei file *build.gradle.kts*, sia dell'intero progetto che di ciascun package, vengono definite le principali direttive per la build del progetto e la risoluzione delle dipendenze.

- **Plugins**: in questa sezione vengono definiti i plugin che vengono utilizzati dai singoli sotto-progetti. Tutti i package fanno uso del plugin *java-library*, mentre i soli package *server*, *client* e *sob* fanno uso del plugin *application* che consente l'esecuzione di un progetto a riga di comando.
- **Repositories**: in questa sezione vengono definiti i repository che devono essere utilizzati per la risoluzione delle dipendenze in ciascun sotto-progetto. In questo caso, ciascun package fa uso di **Maven Central** a questo scopo.
- **Dependencies**: in questa sezione vengono definite tutte le dipendenze dei singoli sotto-progetti. Un sotto-progetto può essere dipendente, per esempio, da un altro sotto-progetto o da librerie esterne.
- **Application**: in questa sezione vengono definite le rispettive classi *main* per ciascun sotto-progetto, qualora presenti.

Di seguito si riporta in forma esemplificativa, il contenuto del file *build.gradle.kts* del package *sob*.

```
plugins {
    application
    'java-library'
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter:5.9.0")
    api ("com.google.code.gson:gson:2.9.0") //dipendenza dalla libreria gson
    api (project(":common")) //dipendenza dal sotto-progetto common
}

application {
    mainClass.set("it.unibo.ds.bft.sob.Sob")
```

[1] https://gitlab.com/pika-lab/courses/ds/projects/ds-project-perugini-ay2122

```
}

tasks.getByName<JavaExec>("run") {
    standardInput = System.`in`
}
```

In questo specifico esempio viene inoltre definita una ulteriore proprietà per uno specifico task del sotto-progetto. Nello specifico viene espresso che per il task *run* lo standard input del systema *System.`in`* debba essere utilizzato come standard input del sotto-progetto.

10.2 Esecuzione del sistema

Come già detto, il sistema non offre allo stato attuale particolari tecnologie per il supporto al deployment, che deve quindi avvenire in maniera completamente manuale.

10.2.1 Client

Per avviare un'istanza del client è necessario eseguire il seguente comando

```
$ gradle client:run --args=#
```

sostituendo a # l'identificatore del client che si intende avviare. L'identificatore scelto determina la porta che l'istanza utilizzerà per le comunicazioni mediante socket, quindi in una esecuzione locale non è possibile mettere in esecuzione due client con stesso identificatore in quanto al massimo un processo potrà utilizzare tale porta.

Ad inizio esecuzione il client legge le configurazioni dal file *resources/configuration.json* e tenta di inizializzare le socket. Se tutto va a buon fine il client, con stato iniziale pari a 0, si mette in attesa di un'input dell'utente per l'invio di una richiesta.

```
[CONFIG]: Reading configuration....
[CLIENT-1]: INIT STATUS (0.000000).
[BOOT]: Service started on port 4450.
[INPUT]: Input your request: [OPERATION] [OPERAND].
```

Figura 37: Avvio del client.

Le richieste possono essere create specificando a riga di comando l'*OPERAZIONE* seguita dall'*OPERANDO*, come schematizzato nella seguente tabella.

	Forma breve	**Forma estesa**	**Risultato atteso**
Addizione	ADD N	ADDITION N	*STATO* + N
Sottrazione	SUB N	SUBTRACTION N	*STATO* - N
Moltiplicazione	MUL N	MULTIPLICATION N	*STATO* * N
Divisione	DIV N	DIVISION N	*STATO* / N
Lettura	READ	READ	*STATO*

Tabella 3: Sintassi breve ed estesa delle operazioni.

Se la sintassi è corretta e il client riesce a creare una richiesta dall'input inserito, questa viene inviata ai server in modalità multicast e il client si mette in attesa di risposte.

```
[CONFIG]: Reading configuration....
[CLIENT-1]: INIT STATUS (0.000000).
[BOOT]: Service started on port 4450.
[INPUT]: Input your request: [OPERATION] [OPERAND].
READ
[CLIENT]: Message sent to replicas.
[CLIENT]: Waiting for responses....
```

```
[CONFIG]: Reading configuration....
[CLIENT-1]: INIT STATUS (0.000000).
[BOOT]: Service started on port 4450.
[INPUT]: Input your request: [OPERATION] [OPERAND].
ADD 1
[CLIENT]: Message sent to replicas.
[CLIENT]: Waiting for responses....
```

Figura 38: Creazione ed invio di richieste.

Alla ricezione di una risposta il client effettua tutte le operazioni già descritte in precedenza, producendo anche un output utile all'utente per conoscere l'esito.

```
[CLIENT-1]: Message type: REPLY.
[CLIENT-1]: MESSAGE (10517931936682) IS AUTHENTICATED.
[CLIENT-1]: TIMESTAMP IS CORRECT.
[SERVER]: Message handled!.
```

Figura 39: Ricezione di una risposta.

Quando le risposte ricevute sono sufficienti all'emissione di un certificato il client considera corretto il risultato e aggiorna il proprio stato, sempre riportando tutte le informazioni in output.

```
[CLIENT-1]: Message type: REPLY.
[CLIENT-1]: MESSAGE (10517931936682) IS AUTHENTICATED.
[CLIENT-1]: TIMESTAMP IS CORRECT.
[CLIENT-1]: GOT A CERTIFICATE FOR (10517931936682).
[CLIENT-1]: NEW STATUS (1.000000).
[SERVER]: Message handled!.
```

Figura 40: Ricezione di una risposta con emissione di un certificato.

```
[CLIENT]: Waiting for responses....
[CLIENT-1]: Message type: REPLY.
[CLIENT-1]: MESSAGE (10517931936682) IS AUTHENTICATED.
[CLIENT-1]: ALREADY HAVE A CERTIFICATE FOR THIS MESSAGE (10517931936682).
[SERVER]: Message handled!.
```

Figura 41: Ricezione di una risposta con certificato già emesso.

10.2.2 Server

Per avviare un'istanza del server è necessario eseguire il seguente comando

```
1  $ gradle server:run --args=#
```

L'identificatore scelto determina la porta che l'istanza utilizzerà per le comunicazioni mediante socket, quindi in una esecuzione locale non è possibile mettere contemporaneamente in esecuzione due server con stesso identificatore.

Ad inizio esecuzione il server legge le configurazioni dal file *resources/configuration.json* e tenta di inizializzare le socket. Se tutto va a buon fine il server si mette in attesa di un messaggio con stato iniziale pari a 0.

```
[CONFIG]: Reading configuration....
[BOOT]: Service started on port 4470.
[SERVER-1]: Waiting for messages....
```

Figura 42: Avvio del server.

Alla ricezione di una Request da parte di un client, la replica effettua tutte le operazioni già descritte in precedenza, producendo anche output utile all'utente per conoscere l'andamento delle operazioni. L'output a video differisce in base al ruolo attuale della replica, primary o backup.

```
[SERVER-1]: NEW MESSAGE RECEIVED FROM :4450.
[BFT]: BYZ_INVOKE: REQUEST.
[SERVER-1]: Message type: REQUEST.
[SERVER-1]: MESSAGE IS AUTHENTICATED.
[SERVER-1]: NEW REQUEST.
[SERVER-1]: PREPREPARE SENT TO REPLICAS.
[SERVER-1]: PRE-PREPARES FOR SEQ.NUM. 1 SENT TO REPLICAS.
[SERVER-1]: REQUEST LOGGED.
[SERVER]: Message handled!.
[SERVER-1]: Waiting for messages....
```

Figura 43: Primary: ricezione Request.

```
[SERVER-3]: NEW MESSAGE RECEIVED FROM :4450.
[BFT]: BYZ_INVOKE: REQUEST.
[SERVER-3]: Message type: REQUEST.
[SERVER-3]: MESSAGE IS AUTHENTICATED.
[SERVER-3]: NEW REQUEST.
[SERVER-3]: REQUEST LOGGED.
[SERVER]: Message handled!.
[SERVER-3]: Waiting for messages....
```

Figura 44: Backup: ricezione Request.

Alla ricezione di una Pre-Prepare, ogni replica di backup effettua tutte le operazioni già descritte in precedenza, producendo anche un output utile all'utente. Se l'esito di tutti i controlli ha esito positivo vengono inviati i messaggi di Prepare alle altre repliche.

```
[SERVER-2]: NEW MESSAGE RECEIVED FROM :4470.
[BFT]: BYZ_INVOKE: PREPREPARE.
[SERVER-2]: Message type: PREPREPARE.
[SERVER-2]: SEQUENCE NUMBER: 1.
[SERVER-2]: VIEW: 1.
[CHECK]: MESSAGE IS AUTHENTICATED.
[CHECK]: MESSAGE IS FOR CURRENT VIEW.
[CHECK]: MESSAGE IS BETWEEN h AND H.
[SERVER-2]: Message ACCEPTED.
[SERVER-2]: MESSAGE FOUND IN LOG.
[SERVER-2]: PREPARE SENT TO REPLICAS.
[SERVER-2]: PREPARES SENT TO REPLICAS.
[SERVER-2]: PREPREPARE LOGGED.
[SERVER-2]: PREPARE LOGGED.
[SERVER]: Message handled!.
[SERVER-2]: Waiting for messages....
```

Figura 45: Ricezione di Pre-Prepare.

Alla ricezione di una Prepare da parte di un client, la replica effettua tutte le operazioni già descritte in precedenza, producendo anche un output che differisce nel caso in cui venga emesso un certificato di prepare. Quando la Prepare va a formare un certificato, la replica procede con l'invio dei messaggi di commit.

```
[SERVER-1]: NEW MESSAGE RECEIVED FROM :4471.
[BFT]: BYZ_INVOKE: PREPARE.
[SERVER-1]: Message type: PREPARE.
[SERVER-1]: Seq num: 1.
[CHECK]: MESSAGE IS AUTHENTICATED.
[CHECK]: MESSAGE IS FOR CURRENT VIEW.
[CHECK]: MESSAGE IS BETWEEN h AND H.
[SERVER-1]: Message ACCEPTED.
[SERVER-1]: PREPARE LOGGED.
[SERVER-1]: NO PREPARED CERTIFICATE YET.
[SERVER]: Message handled!.
[SERVER-1]: Waiting for messages....
```

Figura 46: Ricezione di Prepare senza emissione di certificato.

```
[SERVER-1]: NEW MESSAGE RECEIVED FROM :4472.
[BFT]: BYZ_INVOKE: PREPARE.
[SERVER-1]: Message type: PREPARE.
[SERVER-1]: Seq num: 1.
[CHECK]: MESSAGE IS AUTHENTICATED.
[CHECK]: MESSAGE IS FOR CURRENT VIEW.
[CHECK]: MESSAGE IS BETWEEN h AND H.
[SERVER-1]: Message ACCEPTED.
[SERVER-1]: PREPARE LOGGED.
[SERVER-1]: GOT A PREPARED CERTIFICATE.
[SERVER-1]: COMMIT SENT TO REPLICAS.
[SERVER-1]: COMMIT LOGGED.
[SERVER]: Message handled!.
[SERVER-1]: Waiting for messages....
```

Figura 47: Ricezione di una Prepare con emissione di certificato ed invio del commit.

Alla ricezione di un messaggio di Commit da parte di un client, la replica effettua tutte le operazioni già descritte in precedenza. L'output prodotto a video sarà differente nel caso in cui venga emesso un certificato di commit. Quando il messaggio di commit va a formare un certificato, la replica procede con l'esecuzione dell'operazione richiesta e con l'invio della Reply al client.

```
[SERVER-1]: NEW MESSAGE RECEIVED FROM :4471.
[BFT]: BYZ_INVOKE: COMMIT.
[SERVER-1]: Message type: COMMIT.
[SERVER-1]: SEQUENCE NUMBER: 1.
[SERVER-1]: VIEW: 1.
[CHECK]: COMMIT IS AUTHENTICATED.
[CHECK]: COMMIT IS FOR CURRENT VIEW.
[CHECK]: COMMIT IS BETWEEN h AND H.
[SERVER-1]: COMMIT ACCEPTED.
[SERVER-1]: NO COMMITTED CERTIFICATE YET.
[SERVER]: Message handled!.
[SERVER-1]: Waiting for messages....
```

Figura 48: Ricezione di Commit senza emissione di certificato.

```
[SERVER-1]: NEW MESSAGE RECEIVED FROM :4472.
[BFT]: BYZ_INVOKE: COMMIT.
[SERVER-1]: Message type: COMMIT.
[SERVER-1]: SEQUENCE NUMBER: 1.
[SERVER-1]: VIEW: 1.
[CHECK]: COMMIT IS AUTHENTICATED.
[CHECK]: COMMIT IS FOR CURRENT VIEW.
[CHECK]: COMMIT IS BETWEEN h AND H.
[SERVER-1]: COMMIT ACCEPTED.
[SERVER-1]: GOT A COMMITTED CERTIFICATE.
[SERVER-1]: NEXT EXECUTABLE = 1.
[SERVER-1]: SEQNUM = 1.
[SERVER-1]: EXECUTABLE.
[SERVER-1]: NEW STATUS (0.000000).
[SERVER-1]: REPLY SENT TO CLIENT (:4450).
[SERVER-1]: LOG CHECKPOINT MESSAGE.
[SERVER-1]: CHECKPOINT SENT TO REPLICAS.
[SERVER-1]: MULTICAST CHECKPOINT MESSAGE.
[SERVER]: Message handled!.
[SERVER-1]: Waiting for messages....
```

Figura 49: Ricezione di Commit con emissione di certificato ed invio della risposta.

10.2.2.1 Checkpoint

Nel caso in cui il sequence number n associato alla richiesta appena eseguita sia un multiplo dell'*intervallo di Checkpoint* K, le repliche procedono con la creazione di un checkpoint per n. Anche in questo caso, alla ricezione del messaggio, verrà prodotto un output che consente di monitorarne l'avanzamento e che sarà differente quando il checkpoint diventa stabile.

```
[SERVER-1]: NEW MESSAGE RECEIVED FROM :4471.
[BFT]: BYZ_INVOKE: CHECKPOINT.
[SERVER-1]: Message type: CHECKPOINT.
[SERVER-1]: SEQUENCE NUMBER: 1.
[CHECKPOINT]: MESSAGE IS AUTHENTICATED.
[CHECKPOINT]: CHECKPOINT LOGGED.
[SERVER]: Message handled!.
[SERVER-1]: Waiting for messages....
```

Figura 50: Ricezione checkpoint.

```
[SERVER-1]: NEW MESSAGE RECEIVED FROM :4473.
[BFT]: BYZ_INVOKE: CHECKPOINT.
[SERVER-1]: Message type: CHECKPOINT.
[SERVER-1]: SEQUENCE NUMBER: 1.
[CHECKPOINT]: MESSAGE IS AUTHENTICATED.
[CHECKPOINT]: CHECKPOINT LOGGED.
[CHECKPOINT]: CHECKPOINT FOR SEQ.NUM. 1 IS STABLE.
[CHECKPOINT]: OLD CHECKPOINTS REMOVED.
[SERVER]: Message handled!.
[SERVER-1]: Waiting for messages....
```

Figura 51: Ricezione checkpoint stabile.

10.2.2.2 View Change

Un primary corrotto invierà alle repliche dei messaggi di Pre-Prepare contenenti un sequence number errato pari a 0. Una replica che riceve una Pre-Prepare con sequence number pari a 0 avvia immediatamente la procedura di View Change, in quanto capisce che il primary è fallato, entrando in VIEWCHANGE_PHASE ed inviando un messaggio di ViewChange alle altre repliche.

La replica che riceve un messaggio di ViewChange deve innanzitutto controllare che possa essere accettato e, in caso affermativo, invia alla replica mittente un messaggio di tipo ViewChangeAck.

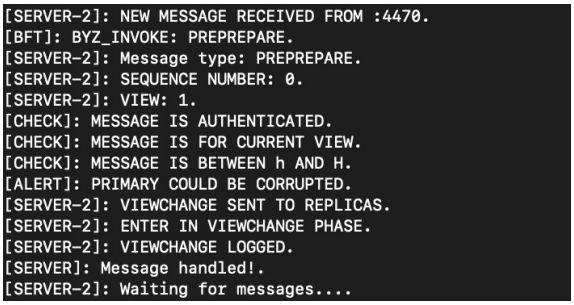

Figura 52: Invio e ricezione dei messaggi di View Change.

Al momento della ricezione di un messaggio di ViewChangeAck, una replica deve come prima cosa loggarlo e poi controllare se il messaggio può far parte di un certificato. In caso affermativo la replica avvia la *Decision Procedure* al fine di determinare una serie di richieste che permettano la ripresa delle normali operazioni. La replica eseguirà la procedura ogni volta che un ViewChangeAck compone un certificato per la corrispondente ViewChange, fino a quando non riesce a determinare tutte le richieste necessarie. In tal caso, effetturerà l'invio dei messaggi di NewView.

Figura 53: Invio e ricezione dei messaggi di View Change.

Quando una replica riceve un messaggio di NewView deve controllare innanzitutto di possedere tutte le relative ViewChange, dopodichè eseguirà la *Decision Procedure* per verificare le informazioni comunicate dal nuovo primary.

Se le informazioni coincidono, tutte le richieste comunicate vengono aggiunte ai log e per ciascuna viene inviato un messaggio di Prepare. Dopodichè la replica esce dalla VIEWCHANGE_PHASE e riprende le normali operazioni.

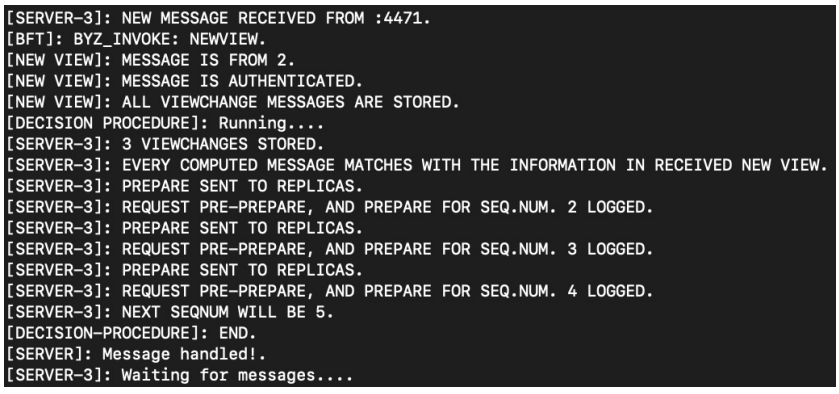

Figura 54: Gestione del messaggio di NewView.

10.2.3 SOB

Per avviare un'istanza SOB è necessario eseguire il seguente comando

```
$ gradle sob:run
```

Il SOB non richiede argomenti in quanto possiede una unica scelta per la porta su cui mettersi in esecuzione, ovvero la porta ***4440***, di conseguenza non è possibile avviare due entità contemporaneamente.

Ad inizio esecuzione il SOB legge le configurazioni dal file *resources/configuration.json* e tenta di inizializzare le socket. Se tutto va a buon fine il SOB si mette in attesa di un'input dell'utente per l'invio di un messaggio.

```
[CONFIG]: Reading configuration....
[CONFIG]: Setting up sockets....
[CONFIG]: Socket address is localhost.
[CONFIG]: Socket port is 4400.
[INPUT]: Input your request: [TYPE] [ENTITY].
```

Figura 55: Avvio del SOB.

I messaggi possono essere create specificando a riga di comando l'*OPERAZIONE* seguita dall'*ENTITÁ*, come schematizzato nella seguente tabella.

	Forma breve	**Forma estesa**
Corruzione server	COR SER N	CORRUPT SERVER N
Riavvio server	RES SER N	RESTART SERVER N
Spegnimento server	SHU SER N	SHUTDOWN SERVER N
Riavvio client	RES CLI N	RESTART CLIENT N
Spegnimento client	SHU CLI N	SHUTDOUWN CLIENT N

Tabella 4: Sintassi breve ed estesa per i messaggi SOB.

Il SOB non attenderà risposte di conferma al messaggio inviato, ma per l'utente sarà possibile controllare l'esito della gestione del messaggio monitorando i log a video di client o server.

10.3 Sviluppi futuri

Come già detto, il sistema sviluppato riporta alcuni *rilassamenti* e non rappresenta in maniera fedele tutti gli aspetti e i vincoli imposti dal sistema teorico descritto nel paper. Sarebbe sicuramente interessante continuare con lo sviluppo del sistema andando ad emulare dettagliatamente le caratteristiche di PBFT, anche successivamente e in circostante non puramente didattiche come quelle per le quali è invece stato realizzato.

Uno degli aspetti principali su cui si potrebbe focalizzare una versione 2.0 potrebbe essere il meccanismo di Proactive Recovery, contestualmente al quale dovrebbero essere implementati tutti i processi collaterali come *Key Exchange, salvataggio, recupero e controllo dello stato* che non sono stati inclusi nell'attuale versione per motivi legati al tempo, ma per i quali sono state poste delle basi.

La realizzazione di un completo *proof of concept* di PBFT potrebbe essere stimolante al fine di poter mettere alla prova il sistema stesso, ad esempio con un *p.o.c.* di una semplice Blockchain.

10.4 Conclusioni

L'algoritmo di consenso PBFT è stato uno dei primi algoritmi di consenso progettati per la gestione di una rete con nodi distribuiti e trova ancora oggi applicazione in molte reti blockchain private e ibride, oltre che in molti sistemi che richiedono alta sicurezza e affidabilità. Questo è dovuto sicuramente ai suoi principali punti di forza rispetto ad altri approcci:

- **Sicurezza:** l'approccio *in tre fasi* adottato da PBFT garantisce che anche se alcuni nodi all'interno della rete sono fallati, la maggioranza dei nodi corretti sarà in grado di raggiungere il consenso su una decisione. Inoltre, siccome ciascun nodo all'interno della rete possiede un identificativo univoco che lo distingue dagli altri ed una chiave condivisa con gli altri nodi, è possibile verificare che ogni messaggio scambiato nel processo di consenso provenga da un certo nodo e che quindi solo i nodi autorizzati partecipino al processo.

- **Affidabilità:** grazie all'uso di *Quorum System*, PBFT garantisce che la maggioranza dei nodi nel quorum riescano a raggiungere il consenso e validare le transazioni anche nel caso in cui una singola replica sia compromessa o non disponibile. Inoltre, se la maggioranza di nodi nel quorum è rappresentata da nodi corretti, questi riusciranno ad impedire ad una singola replica fallata di compromettere la rete o di effettuare transazioni false.

- **Throughput:** l'uso dei *Quorum System* impatta indirettamente, ma in modo positivo, sul throughput del sistema. In approcci come *Proof of Work*, *Proof of Stake* o altri basati su ricompensa, ai nodi della rete viene richiesto di eseguire molti calcoli al fine di garantire la loro attendibilità all'interno della rete. La conseguenza di questo approccio è che i nodi potrebbero essere congestionati dai calcoli, facendo ridurre il numero di richieste evase in un certo lasso temporale. In PBFT l'interazione tra i nodi ha invece un fine coordinativo e grazie ai Quorum si riduce la necessità di calcoli, permettendo di ottenere un throughput maggiore.

Tuttavia, PBFT porta con sè anche alcuni limiti:

- **Complessità:** nonostante sotto l'aspetto computazionale PBFT non risulti eccessivamente complesso, può essere considerato tale se si considerano le sue fasi di implementazione e gestione. L'implementazione di PBFT richiede una certa conoscenza delle tecnologie di rete e della crittografia, il che può rendere la sua implementazione più complessa rispetto ad altri algoritmi di consenso, mentre la gestione di una rete basata su PBFT richiede un'attenta pianificazione e monitoraggio, in particolare per quanto riguarda la gestione dei nodi e la risoluzione dei problemi tecnici.

- **Scalabilità:** l'architettura di PBFT è progettata per funzionare in modo maggiormente efficiente in una rete di dimensioni ridotte, ovvero con pochi nodi attivi che partecipano al processo di consenso. Quando si cerca di implementare PBFT in una rete aumentando il numero n di nodi, l'utilizzo della banda aumenta in modo direttamente proporzionale ad n in quanto sarà maggiore il numero di messaggi scambiati dalle repliche per raggiungere il consenso.

- **Velocità:** l'aumento del numero di nodi all'interno di una rete PBFT impatta di conseguenza in modo negativo sulla velocità. Siccome ogni nodo deve partecipare attivamente inviando messaggi verso tutti gli altri e raccogliendo parte delle risposte, all'aumento del numero dei nodi e del traffico nella rete potrebbe verificarsi un aumento della latenza nella ricezione dei messaggi con conseguente rallentamento delle operazioni.

Inoltre un sistema PBFT potrebbe essere compromesso da un *Sybil Attack*, un attacco in cui il processo di consenso viene stravolto mediante la creazione di identità multiple nel sistema. Poichè questo algoritmo è basato sulla proprietà che la maggioranza dei nodi partecipanti siano onesti, se un intrusore riuscisse ad aggirare il meccanismo di identità univoca e a creare molte identità false (dette *nodi Sybil*) all'interno della rete potrebbe avere la capacità di controllare la maggioranza dei nodi. In questo modo il processo di consenso risulterebbe appunto ribaltato, permettendo all'intrusore di inviare messaggi corrotti e compromettere la integrità del sistema.

I limiti e le debolezze appena descritti, rendono PBFT maggiormente adatto e sicuro in ambienti privati per i seguenti motivi:

- *Sicurezza*: se i nodi che partecipano al processo di consenso sono controllati da enti fidati, viene ridotto il rischio di attacchi Sybil e di compromissione.

- *Controllo*: in un ambiente privato gli operatori possono controllare e gestire i nodi in modo più diretto ed efficiente, rendendo più semplice la gestione di eventuali problemi tecnici o di sicurezza.

- *Prestazioni*: siccome l'algoritmo offre maggiore efficienza con un numero limitato di nodi, in un ambiente dove la dimensione della rete è controllabile le prestazioni e la velocità di elaborazione delle transazioni saranno generalmente elevate.

Al contrario, in ambienti pubblici e decentralizzati, dove il numero di nodi è elevato e potrebbe crescere senza un controllo diretto, potrebbero essere necessari altri algoritmi di consenso per garantire la sicurezza e l'affidabilità della rete.