

# **Zyzyva: Analisi ed implementazione dell'algoritmo per consenso in sistemi distribuiti**

Giovanni Mormone

`giovanni.mormone@studio.unibo.it`

Fabian Aspee Encina

`fabian.aspeeencina@studio.unibo.it`

Gennaio 2021

Nell'ambito dei sistemi distribuiti, caratterizzati dall'assenza di memoria condivisa, il modo più diffuso che hanno i processi di comunicare tra loro è con uno scambio di messaggi attraverso una rete asincrona. L'assenza di memoria condivisa implica il fatto che ci potrebbero essere delle inconsistenze nello stato dei vari nodi che compongono il sistema con conseguente invio di dati errati all'applicazione che li richiede; altre possibilità dovute all'architettura distribuita possono essere fallimento di un nodo del sistema o il fallimento della rete di comunicazione. Questi errori portano alla necessità di implementare algoritmi per arrivare al consenso, cioè ottenere un accordo sullo stato del sistema tra macchine inaffidabili connesse in una rete asincrona. Nel contesto del consenso ha molta importanza raggiungere il consenso in un sistema che presenta la possibilità di "Byzantine Failure": il fallimento bizantino è relativo al problema che alcuni nodi del sistema potrebbero essere compromessi o comportarsi in maniera arbitraria, mandando quindi valori errati come risposta a una richiesta. Sono dunque nati vari algoritmi per assicurare sistemi con una Byzantine Fault Tolerance (BFT), dove per BFT si indica la capacità di funzionare in modo corretto, e dunque raggiungere il consenso tra i nodi del sistema nonostante la presenza di nodi bizantini. Un contesto in cui questi algoritmi sono molto utilizzati è quello dei sistemi State Machine Replication (SMR), che è un metodo generale per implementare un servizio fault-tolerant replicando i server e coordinando le interazioni del client con le repliche server [20].

# 1 Problema del consenso e fallimenti bizantini

Il consenso tra processi in un sistema distribuito asincrono coinvolge processi che possono essere inaffidabili. Secondo Fischer et al. [14] questi tipi di problemi hanno la possibilità di non raggiungere mai un accordo, diversamente dal caso sincrono, dove Lamport et al. fa riferimento a "The Byzantine Generals Problem" [18], che consente al sistema, in presenza di  $n$  repliche *faulty*, di continuare a funzionare ed arrivare ad un consenso.

## 1.1 Problema del consenso

Il problema del consenso è uno dei problemi fondamentali della computazione distribuita ed è al centro di molti algoritmi per l'elaborazione di dati distribuiti [14]. Sempre secondo Fischer et al., il problema del consenso consiste nel fare in modo che, in presenza di un sistema di processi, tutti i processi corretti raggiungano un accordo su un valore; questo è impossibile in un sistema asincrono, mentre diventa un problema risolvibile in un sistema sincrono, conosciuto come il caso dei "Generali Bizantini" [14]. I processi che costituiscono il sistema possono essere divisi in due gruppi:

- Processi corretti, che sono quindi in possesso dei giusti valori riguardo al sistema e sono raggiungibili e disponibili a comunicare questi valori.
- Processi *faulty*, che possono avere dei valori errati, possono essere irraggiungibili (ad esempio per problemi di rete o di *crash* della macchina su cui sono istanziati) oppure possono assumere comportamenti arbitrari in cui forniscono deliberatamente valori errati (ad esempio in seguito ad attaccanti che prendono possesso del processo).

In base a queste definizioni, in un sistema in cui i processi si scambiano messaggi per proporre dei valori, valori su cui dovranno raggiungere un consenso, devono essere sempre valide le seguenti proprietà, come descritto in [11]

- Terminazione, prima o poi un processo corretto prenderà una decisione.
- Accordo, la decisione presa da ogni processo corretto deve essere la stessa.
- Integrità, se tutti i processi corretti hanno proposto lo stesso valore, allora tutti i processi corretti hanno scelto quel valore.

## 1.2 Fallimenti bizantini

Secondo Lamport et al., un sistema informatico affidabile deve essere in grado di far fronte al guasto di uno dei suoi componenti [18], detto questo, i possibili fallimenti che possiamo avere in un sistema distribuito sono [13]:

- *Fail-stop*: il processore  $p$  esegue correttamente, ma può interrompersi in qualsiasi momento.

- *Omission*: un processore *faulty*  $p$  esegue il suo protocollo correttamente, ma in seguito ad una  $\text{Send}(m, p_j)$ , quando viene eseguito per  $p_i$  potrebbe non mettere  $m$  nel buffer di  $p_j$ .
- *Authenticated Byzantine*: comportamento arbitrario, però i messaggi possono essere firmati col nome del processore che li manda.
- *Byzantine*: comportamento arbitrario ma non ci sono meccanismi di firma.

Allora, Lamport et al. introducono una soluzione per i problemi bizantini in cui dicono che, avendo  $3f + 1$  processori in un sistema, in presenza di fallimenti bizantini il sistema può continuare a funzionare in maniera corretta, sempre che i processori *faulty* non superino  $f$ . Per poter ottenere affidabilità, vengono usate delle firme sui messaggi in modo tale che una replica non riceva dei messaggi falsificati, dato che non si può falsificare la firma di una replica e si può rivelare qualsiasi alterazione su di questa, e qualsiasi replica può verificare l'autenticità di una firma.

### 1.3 Introduzione Algoritmo ZyzyvaSBFT

Come descritto nelle sottosezioni precedenti 1.1 1.2 questi tipi di problemi hanno dato origine a molti algoritmi che provano a risolverli. Uno di questi è Zyzyva [15], algoritmo che viene creato per poter migliorare le prestazioni sia di PBFT [10] che quelle di HQ [12] e Q/U [8]. A differenza dei precedenti algoritmi, Zyzyva si basa sulla speculazione per ridurre il costo ed aumentare le prestazioni del sistema. In Zyzyva le repliche eseguono le richieste proposte da un *Primary* in maniera speculativa, senza dunque effettuare per ogni richiesta una costosa fase di accordo. Per questo le risposte inviate ai *Client* e l'ordine delle richieste eseguite dalle repliche corrette potrebbe differire nel tempo; nonostante questo, le risposte viaggiano con un'*history* tale da permettere al *Client* di rendersi conto che una risposta è stabile. Grazie alla speculazione, Zyzyva permette di completare una richiesta, nel caso migliore, in due fasi del protocollo invece delle solite tre fasi utilizzate negli altri algoritmi di BFT. Zyzyva è composto da tre sottoprotocolli principali:

- *Agreement subprotocol*: sottoprotocollo che si occupa di raggiungere l'accordo tra le repliche per le richieste ricevute. È suddiviso a sua volta in 3 fasi diverse, che sono eseguite in base al numero di risposte corrispondenti ricevute da un *Client*.
- *Viewchange subprotocol*: sottoprotocollo che si occupa di sostituire un *Primary faulty* con una delle altre repliche che costituiscono il sistema. In questa fase deve essere raggiunto un accordo sull'*history* che deve essere adottata da tutte le repliche nella nuova view.
- *Checkpoint subprotocol*: sottoprotocollo che si occupa di salvare dei *checkpoint* ogni  $N$  richieste, *checkpoint* che sono composti dall'*history* comune alla maggioranza delle repliche corrette.

## 2 Obiettivi/Requisiti

### 2.1 Obiettivi

Considerando il problema dei sistemi soggetti a fallimenti bizantini e il proposito del progetto, gli obiettivi saranno:

- Analisi dell'algoritmo
- Implementazione dell'algoritmo
- *Testing* dell'implementazione fatta
- Implementazione di un semplice sistema SMR per verificare la correttezza dell'implementazione

### 2.2 Requisiti

I principali requisiti che ci siamo prefissati sono:

- Semplicità della distribuzione e del *deploy* del sistema, facendo in modo che, piuttosto di far partire un solo artefatto alla volta manualmente, il *deploy* venga effettuato usando solo un comando che fa *deploy* di tutti gli artefatti insieme.
- Pulizia e buona progettazione del codice, cercando di rispettare principi quali DRY, KISS e utilizzando pattern noti della programmazione ad oggetti per permettere una più semplice manutenzione ed estensione del codice.
- Integrazione, laddove possibile, di alcuni aspetti del paradigma funzionale.

### 2.3 Scenari

Gli scenari di utilizzo di questo sistema sono perlopiù orientati a mostrare il funzionamento del protocollo Zyzzyva [15]. Pertanto l'interazione con il sistema è mirata al test della corretta implementazione dello stesso nei vari casi che si possono presentare durante l'esecuzione di un sistema distribuito con possibili fallimenti bizantini, come ad esempio:

- Comportamento arbitrario di una replica, dovuta ad esempio all'attacco da parte di terzi.
- Non raggiungibilità di una replica, dovuta all'assenza di rete o al crash della stessa

### 3 ZyzzyvaSBFT

Come descritto nella sottosezione 1.3, Zyzzyva è un protocollo per gestire i fallimenti bizantini in un sistema distribuito, che però a differenza degli altri offre delle migliori prestazioni poiché è più leggero e in più la quantità di messaggi scambiati tra le repliche è minore. Secondo gli autori *Zyzzyva semplifica lo spazio di progettazione dei servizi replicati BFT avvicinandosi ai limiti inferiori in quasi tutte le metriche chiave* [15], questa citazione viene espressa dopo aver paragonato Zyzzyva con PBFT, Q/U, HQ.

L'algoritmo Zyzzyva è composto da diversi processi, che sono di due tipi:

- *Replica*, che è un processo in esecuzione su un nodo del sistema distribuito, indipendente dalle altre e che comunica con altre repliche e con i *Client* attraverso scambio di messaggi.
- *Client*, che è un processo che si interfaccia con l'applicazione, ne riceve le richieste e le inoltra alle repliche. Si occupa inoltre di raccogliere le risposte speculative delle repliche e di rispondere all'applicazione con queste risposte.

Nelle prossime sottosezioni si procederà alla descrizione dei sottoprotocolli che compongono Zyzzyva, prendendo come modello la spiegazione presente nel paper [16].

#### 3.1 Stato delle repliche

Una Replica  $i$  in Zyzzyva ha uno stato interno composto dai seguenti campi, come mostrato anche in figura 1:

- *Ordered History*, che è composta da tutte le richieste che la replica ha eseguito.
- *Max Commit Certificate*, che è la copia del *Commit Certificate* visto da  $i$  che copre la maggior porzione della sua storia. La parte di *History* che include richieste con un *sequence number*<sup>1</sup> minore o uguale a quello presente nel *Commit Certificate* è detta *Committed History* mentre quella seguente è detta *Speculative History*. Un *Commit Certificate* ha *sequence number*  $n$  quando  $n$  è il più alto *sequence number* presente nella *history* che lo compone.
- *Checkpoint*, che è salvato dalla replica ogni CP\_INTERVAL richieste.
- *Snapshot*, che rappresenta lo stato interno dell'applicazione ed è salvato anch'esso ogni CP\_INTERVAL richieste, insieme al *Checkpoint*.
- *Response Cache*, che contiene una copia dell'ultima richiesta, e della corrispondente risposta, effettuata da ogni *Client*.

Per limitare la dimensione dell'*History*, una replica corretta tronca l'*History* precedente ad un checkpoint e smette di processare nuove richieste dopo averne processate 2 x CP\_INTERVAL a partire dall'ultimo *Checkpoint*.

---

<sup>1</sup>Un sequence number viene assegnato alle richieste dal *Primary*, e rappresenta il momento in cui una richiesta è eseguita

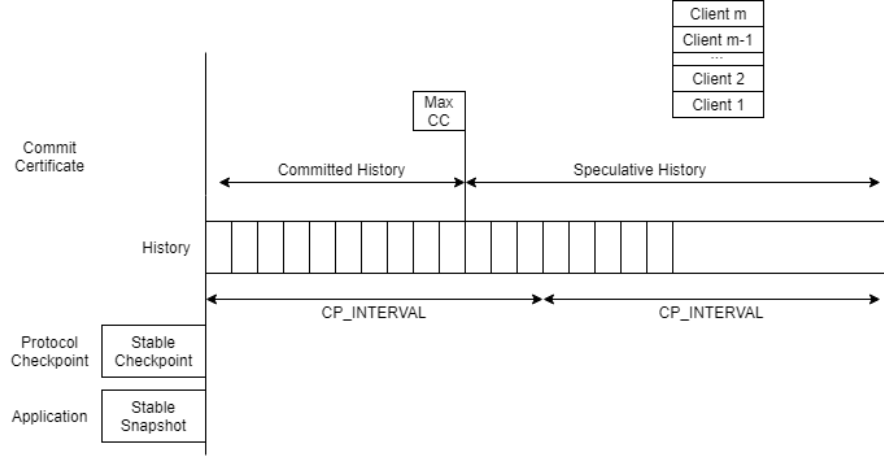


Figura 1: Stato delle repliche

### 3.2 Agreement Subprotocol

L'*agreement subprotocol* in Zyzzyva ha il compito di gestire le richieste dell'utente, che vengono ricevute dal *Client* che le inoltra a un *Primary*<sup>2</sup>, e di generare le risposte da inviare all'utente, risposte che vengono costruite in base alle risposte speculative che ogni replica invia al *Client*. Per poter completare una richiesta le risposte speculative devono essere  $3f + 1$  oppure  $2f + 1$  usando un *local commit*. L'*agreement subprotocol* è formato da 3 casi che sono:

- *fast case*: quando tutti nodi eseguono correttamente e non scade nessun *timer*.
- *two-phase case*: questo caso accade quando al massimo  $f$  repliche falliscono oppure se scadono dei *timer*.
- *view change case*: che può accadere quando il *Primary* è *faulty* oppure se molti *timer* scadono.

La tabella 1 mostra un elenco di tutte le abbreviazioni utilizzate per poter descrivere il protocollo Zyzzyva, che verranno utilizzate nelle sezioni successive. In seguito verranno spiegati in profondità i casi nominati precedentemente.

<sup>2</sup>Il *Primary* è il coordinatore del protocollo che cambia in funzione della *view*, dove il nuovo *Primary* viene eletto in base alla *view*

Label	Meaning
c	Client ID
CC	Commit Certificate
d	Digest (cryptographic 1-way hash) of client request message: $d = H(m)$
i,j	Server IDs
$h_n$	History through sequence number encoded as cryptographic 1-way hash: $h_n = H(h_{n-1}, d)$
m	Message containing client request
$\max_m$	Max sequence number accepted by replica
n	Sequence number
ND	Selection of nondeterministic values needed to execute a request
o	Operation requested by client
OR	Order Request message
POM	Proof Of Misbehavior
r	Application reply to a client operation
t	Timestamp assigned to an operation by a client
v	View number

Tabella 1: Abbreviazioni utilizzate per descrivere il protocollo Zyzzyva

### 3.2.1 Fast case

La figura 2 illustra i passaggi che devono accadere quando tutte le repliche si comportano in modo corretto. Per prima cosa il *Client* manda una richiesta al *Primary*, questo gliela manda a tutte le repliche, le repliche eseguono la richiesta in modo speculativo e rispondono al *Client*<sup>3</sup>. Il *Client* vede che tutte le risposte speculative sono uguali e provengono da  $3f + 1$  repliche e risponde all'applicazione.

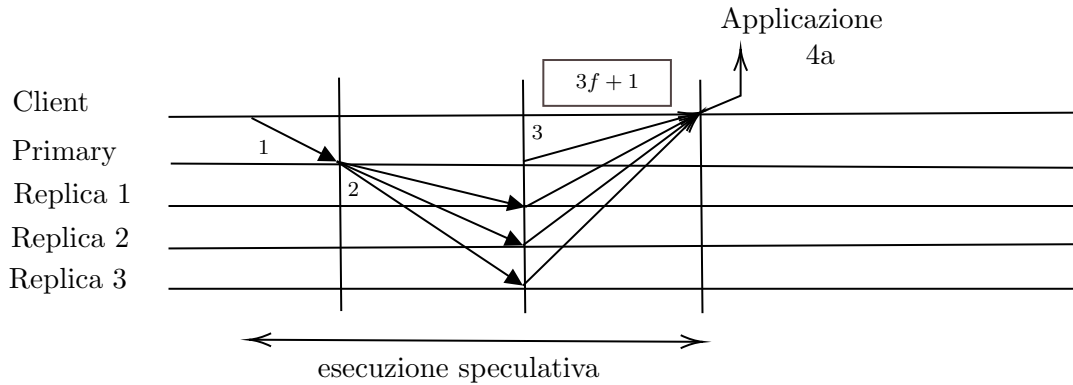


Figura 2: Fast case

<sup>3</sup>Anche il *Primary* deve rispondere al *Client* dopo aver eseguito la richiesta

In seguito si spiegano in modo più approfondito i passaggi eseguiti nel *fast case*.

1. *Client* invia una richiesta al *Primary*

Quando un *Client*  $c$  ha bisogno di richiedere una operazione  $o$  che dovrà essere eseguita dalle repliche, allora il *Client* invia un messaggio  $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$  (1) alla replica che lui pensa sia il *Primary*<sup>4</sup>

2. *Primary* riceve una richiesta, assegna un *sequence number*, e inoltra delle *ordered request* alle repliche.

Il *Primary* nell'attuale *view* è l'unico che ha l'autorità di proporre l'ordine in cui il sistema deve eseguire le richieste, questo lo fa generando dei messaggi *ORDER-REQ* che sono conseguenza dei messaggi *REQUEST* inviati dal *Client*.

Quando un *Primary*  $p$  riceve un messaggio  $m = \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$  da un *Client*  $c$ , il *Primary* deve assegnare un *sequence number*  $n$  alla richiesta nell'attuale *view*  $v$  e inoltrare un messaggio  $\langle \langle \text{ORDER-REQ}, v, n, h_n, d, ND \rangle_{\sigma_p}, m \rangle$  (2) a tutte le repliche.  $n$  e  $v$  indicano il *sequence number* proposto dal *Primary* e la *view* in cui il messaggio  $m$  è stato inviato, digest  $d = H(m)$  è l'hash unidirezionale per  $m$ ,  $h_n = H(h_{n-1}, d)$  è l'hash della *history* e  $ND$  sono un set di valori non deterministici dell'applicazione.

Il *Primary* esegue le azioni precedenti solo se  $t > t_c$ , dove  $t_c$  è il *timestamp* più alto precedentemente ricevuto da  $c$ .

3. La replica riceve una *ordered request*, la esegue in modo speculativo e risponde al *Client*

Quando una replica riceve un messaggio *ORDER-REQ*, questa assume in modo ottimistico che il *Primary* funziona bene e che tutte le altre repliche che stanno funzionando bene riceveranno lo stesso messaggio con lo stesso ordine proposto dal *Primary* e produce una *SPEC-RESPONSE* che manderà al *Client*.

Quando la replica  $i$  riceve *ORDER-REQ* dal *Primary*  $p$ ,  $i$  accetta la *ordered request* solo se  $m$  è una *REQUEST* ben formata,  $d$  è l'hash corretto di  $m$ ,  $v$  è la *view* attuale,  $n = \max_n + 1$  dove  $\max_n$  è l'ultimo *sequence number* nell'*history* di  $i$  e  $h_n = H(h_{n-1}, d)$ . Quando questo messaggio viene accettato  $i$  aggiunge la *ordered request* alla fine della sua *history*, esegue la richiesta usando lo attuale stato dell'applicazione per produrre una risposta  $r$  e infine invia a  $c$  un messaggio  $\langle \langle \text{SPEC-RESPONSE}, v, n, h_n, H(r), c, t \rangle_{\sigma_i}, i, r, OR \rangle^5$  (3).

Una replica può solo accettare ed eseguire speculativamente le richieste in ordine basandosi sul *sequence number*, dato che i messaggi si possono perdere o che il *Primary* può essere *faulty* e può introdurre dei buchi nei *sequence number*, allora la replica  $i$  scarta i messaggi se  $n \leq \max_n + 1$ ; se però  $n > \max_n + 1$  allora  $i$  scarta il messaggio e invia un messaggio  $\langle \text{FILL-HOLE}, v, \max_n, n, i \rangle_{\sigma_i}$  (4) al *Primary* e inizia un *timer*. Quando il *Primary* riceve un messaggio  $\langle \text{FILL-HOLE}, v, k, n, i \rangle_{\sigma_i}$  da una replica  $i$ , il *Primary*  $p$  invia una  $\langle \langle \text{ORDER-REQ}, v, n', h_{n'}, d, ND \rangle_{\sigma_p}, m' \rangle$  per ogni richiesta  $m'$  che  $p$  ha ordinato

<sup>4</sup>Il *Client* sa chi è il *Primary* controllando l'ultima risposta che gli arriva

<sup>5</sup>OR =  $\langle \text{ORDER-REQ}, v, n, h_n, d, ND \rangle_{\sigma_p}$



nell'intervallo di *sequence number*  $k \leq n' \leq n$  nell'attuale *view*, ignorando i messaggi di **FILL-HOLE** da ricevuti da altre *view*. Se la replica  $i$  riceve i messaggi **ORDER-REQ** necessari per coprire i buchi, questa cancella il *timer*. Se il timer scade invece la replica  $i$  rinvia il messaggio **FILL-HOLE** a tutte le altre repliche, setta un *timer* e inizia un *view change* quando questo nuovo *timer* scade. Qualsiasi replica  $j$  che riceve un messaggio **FILL-HOLE** da una replica  $i$  invia il corrispondente messaggio **ORDER-REQ** se ce l'ha. Se nel processo di riempire dei buchi la replica  $i$  riceve dei messaggi **ORDER-REQ** che sono in conflitto tra loro, questi allora formano una *proof of misbehaviour*, che è descritta nel punto 4d della sottosezione 3.2.3

4a. Il *Client* riceve  $3f + 1$  risposte uguali dalle diverse repliche e completa la richiesta.

Quando il *Client* riceve  $3f + 1$  messaggi  $\langle \text{SPEC-RESPONSE}, v, n, h_n, H(r), c, t \rangle_{\sigma_i}, i, r, OR \rangle$  da diverse repliche  $i$ , allora il *Client* determina se questi messaggi **SPEC-RESPONSE** sono identici, dove per sapere se sono identici confrontiamo  $v, n, h_n, H(r), c, t, OR$ , e tutti campi di  $r$ .

Quando non scade nessun *timer* e tutte le  $3f + 1$  risposte ricevute dalle diverse repliche sono identiche, allora queste sono una garanzia sufficiente per assicurare che la risposta alla richiesta fatta in 3.2.1 è frutto di un accordo tra le repliche<sup>6</sup>, ed in particolare le  $3f + 1$  risposte corrette garantiscono che anche in caso di *view change*, la posizione della richiesta nella *history* delle repliche corrette non cambierà.

### 3.2.2 Two-Phase case

Quando la rete, il *Primary* o qualche replica è lenta o è *faulty* il *Client*  $c$  può non ricevere delle risposte uguali da  $3f + 1$  repliche; il *Two-Phase case* si applica quando un *Client* riceve tra  $2f + 1$  e  $3f$  risposte da diverse repliche come si vede nella figura 3

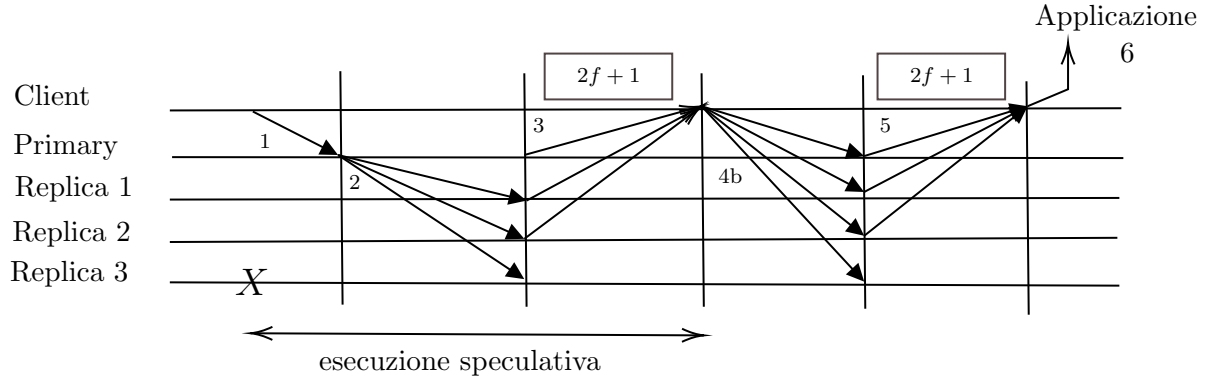


Figura 3: Two phase case

<sup>6</sup>Questo accordo è conosciuto solo dal *Client* in questo momento, dato che le repliche non comunicano tra loro quando eseguono una richiesta.

4b. Il *Client* riceve tra  $2f + 1$  e  $3f$  risposte uguali, crea un *commit certificate*<sup>a</sup> e invia questo a tutte le repliche.

<sup>a</sup>Il *commit certificate* è una prova criptata che una maggioranza di repliche concorda sull'ordine delle richieste fino alla richiesta che il *Client* sta aspettando

Un *Client*  $c$  setta un *timer* quando questo invia una richiesta e, quando questo *timer* scade, se  $c$  ha ricevuto delle risposte speculative da diverse repliche che sono uguali tra loro e sono tra  $2f + 1$  e  $3f$ , allora  $c$  ha una prova che la maggioranza delle repliche corrette sono d'accordo sull'ordine in cui dovrebbe essere processata la richiesta. Le repliche però non sanno di questo quorum, dato che una replica conosce solo la sua decisione locale, fatto che può non garantire che una richiesta venga completata in questo ordine. Prima che un *Client*  $c$  possa accettare questa risposta deve eseguire un step in più per assicurarsi della stabilità della risposta. Il client manda un messaggio  $\langle \text{COMMIT}, c, CC \rangle_{\sigma_c}$  (5) a tutte le repliche, dove  $CC$  è un *commit certificate* composto da una lista di  $2f + 1$  repliche, dalla porzione firmata delle  $2f + 1$  SPEC-RESPONSE e dalle corrispondenti  $2f + 1$  firme delle repliche.

Dato che le  $2f + 1$  risposte che compongono il  $CC$  sono identiche,  $c$  può includere solo una parte firmata del messaggio SPEC-RESPONSE invece di mandare un lista di queste.

5. La replica riceve un messaggio COMMIT dal *Client* che contiene un *commit certificate* e conferma con un messaggio LOCAL-COMMIT.

Quando una replica  $i$  riceve un messaggio  $\langle \text{COMMIT}, c, CC \rangle_{\sigma_c}$  che contiene un *commit certificate*  $CC$  valido, che dimostra che una richiesta deve essere eseguita con un specifico *sequence number* e *history* nella *view* attuale, la replica prima si assicura che la sua *history* locale sia consistente con quella certificata da  $CC$  e in caso affermativo invia un messaggio  $\langle \text{LOCAL-COMMIT}, v, d, h, i, c \rangle_{\sigma_i}$  (6) a  $c$ . Se inoltre il *sequence number* di  $CC$  è maggiore del *sequence number* dell'ultimo  $CC$  salvato la replica salva il  $CC$  contenuto nel messaggio COMMIT.

Se la Ordered History ha dei buchi che appaiono controllando la Ordered History in  $CC$  allora la replica  $i$  manda un messaggio FILL-HOLE come descritto nel punto 3 della sottosezione 3.2.1. Se però le due Ordered History contengono diverse richieste per lo stesso *sequence number*, allora la replica  $i$  inizia il sottoprotocollo *view change*.

6. Il *Client* riceve dei messaggi LOCAL-COMMIT da  $2f + 1$  e completa la richiesta.

Il *Client* rinvia il messaggio COMMIT fino a quando riceve il corrispondente messaggio LOCAL-COMMIT da  $2f + 1$  diverse repliche. Quando il *Client* riceve le  $2f + 1$ <sup>7</sup> risposte, considera la richiesta completata e invia la risposta  $r$  all'applicazione.

<sup>7</sup> $2f + 1$  messaggi LOCAL-COMMIT sono sufficienti per assicurare che un *Client* possa considerare una risposta completata.

Quando un *Client* invia un messaggio COMMIT alle repliche, questo inizia un *timer*. Se il *timer* scade prima che il *Client* riceva  $2f + 1$  messaggi LOCAL-COMMIT, allora il *Client* passa al protocollo descritto nella sottosezione 3.2.3

### 3.2.3 View change case

I casi 4a e 4b permettono a un *Client*  $c$  di completare una richiesta con  $2f + 1$  e  $3f + 1$  risposte uguali, però se il *Primary* o la rete sono *faulty*, il *Client*  $c$  può non ricevere mai dei messaggi SPEC-RESPONSE o LOCAL-COMMIT uguali da  $2f + 1$  repliche. I seguenti casi assicurano che una richiesta dal *Client* verrà completata nell'attuale *view* oppure che una nuova *view* con un nuovo *Primary* verrà inizializzato; in particolare il caso 4c viene scatenato quando un *Client* riceve meno di  $2f + 1$  risposte uguali e il caso 4d accade quando un *Client* riceve delle risposte che indicano inconsistenza da parte del *Primary*.

4c. Il *Client* riceve meno di  $2f + 1$  messaggi SPEC-RESPONSE e rinvia la richiesta a tutte le repliche, che inoltrano la richiesta al *Primary* per assicurarsi che alla richiesta venga assegnato un *sequence number* ed eventualmente venga eseguita.

Quando un *Client* effettua una richiesta, setta un secondo *timer*. Se il secondo *timer* scade prima che la richiesta sia completata, il *Client* sospetta che il *Primary* non stia ordinando le richieste in modo corretto. Questo allora rinvia il messaggio REQUEST a tutte le repliche in modo che possano monitorare l'avanzamento della richiesta e, se il progresso non è soddisfacente, iniziare un *view change*.

*Replica.* Quando una replica  $i$  riceve un messaggio  $\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$  da un *Client*  $c$ , se la richiesta ha un *timestamp* maggiore di quello messo nella Response Cache per quel *Client*,  $i$  invia un  $\langle \text{CONFIRM-REQ}, v, m, i \rangle_{\sigma_i}$ <sup>8</sup> (7) al *Primary*  $p$  e setta un *timer*. Se la replica accetta un messaggio ORDER-REQ per questa richiesta prima che il *timer* scada, processa il messaggio ORDER-REQ come descritto nel punto 3 della sottosezione 3.2.1. Se però il *timer* scade prima che il *Primary* ordini la richiesta allora la replica inizia un *view change*.

*Primary.* Quando riceve un messaggio  $\langle \text{CONFIRM-REQ}, v, m, i \rangle_{\sigma_i}$  dalla replica  $i$ , il *Primary*  $p$  verifica il *timestamp* del *Client* per quella richiesta. Se la richiesta è nuova  $p$  invia un messaggio ORDER-REQ usando un nuovo *sequence number* come descritto nel punto 2 della sottosezione 3.2.1.

Se una replica  $i$  non riceve il messaggio ORDER-REQ dal *Primary* prima che scada il *timer*, la replica invia il messaggio CONFIRM-REQ a tutte le altre repliche e inizia un *view change*. Quando una replica  $j$  riceve un messaggio CONFIRM-REQ dalla replica  $i$ , allora questa invia il corrispondente messaggio ORDER-REQ che ha ricevuto dal *Primary*; se la replica  $j$  non ha ricevuto riceve la ORDER-REQ dal *Primary*, si comporta come se la REQUEST contenuta nel CONFIRM-REQ provenga dal *Client*.

---

<sup>8</sup> $m = \langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$

In più per mantenere una semantica *exactly-once*, la replica mantiene una Response Cache. Se la replica  $i$  riceve una richiesta da un *Client* e questa corrisponde o ha un *timestamp* più basso rispetto a quello presente nella Response Cache per il *Client*  $c$ , allora la replica  $i$  semplicemente rinvia la Response Cache a  $c$ . Lo stesso succede se il *Primary*  $p$  riceve una CONFIRM-REQ da una replica che contiene una REQUEST con un *timestamp* minore rispetto a quello presente attualmente nella Response Cache per il *Client*; in questo caso  $p$  invia a  $i$  il messaggio ORDER-REQ che ha in cache per quel *Client*

4d. Il *Client* riceve delle risposte che indicano un ordine inconsistente dal *Primary* e invia una *proof of misbehaviour* alle repliche, che iniziano un *view change* per cambiare il *Primary* difettoso.

Se un *Client*  $c$  riceve una coppia di messaggi SPEC-RESPONSE che contengono un valido  $OR = \langle \text{ORDER-REQ}, v, n, h_n, d, ND \rangle_{\sigma_j}$  per la stessa richiesta ( $d = H(m)$ ) nella stessa *view* ma con diversi *sequence number*  $n$  o *history*  $h_n$  o  $ND$ , allora questa coppia di messaggi ORDER-REQ costituisce una *Proof Of Misbehaviour (POM)* contro il *Primary*. Quando il *Client* si accorge di un POM,  $c$  invia un messaggio  $\langle \text{POM}, v, POM \rangle_{\sigma_c}$  (8) a tutte le repliche. Quando una replica riceve un valido messaggio POM inizia un *view change* e inoltra il messaggio POM a tutte le altre repliche.

I casi 4b e 4c non escludono 4d; un *Client* può ricevere dei messaggi sufficienti sia per completare una richiesta che per formare una *proof of misbehaviour* contro il *Primary*

### 3.3 Viewchange Subprotocol

#### 3.3.1 Differenze con i classici Viewchange

Il *Viewchange Subprotocol* deve scegliere un nuovo *Primary* quando necessario e deve assicurare che non venga introdotta nessuna modifica in un'*history* che è stata già completata da un *Client*. I tradizionali *Viewchange Subprotocol* richiedono che una replica che partecipi ad un *Viewchange* smetta di accettare qualsiasi messaggio ad eccezione di VIEW-CHANGE, CHECKPOINT e NEW-VIEW. In più per evitare che una replica *faulty* corrompa il sistema, un *Primary* non dovrebbe essere rimosso fino a che almeno una replica corretta partecipi ad un *Viewchange*. Una replica corretta partecipa ad un *Viewchange* in uno dei seguenti due casi:

- Osserva che il *Primary* è *faulty*
- Ha la prova che  $f+1$  repliche partecipano ad un *Viewchange*

Quando una replica partecipa ad un *Viewchange* invia un messaggio VIEW-CHANGE firmato a tutte le repliche, messaggio che include la nuova *view*, il *sequence number* dell'ultimo *stable checkpoint* della replica (insieme ad una prova della sua stabilità) e l'insieme dei *prepare certificates* (equivalenti ai *commit certificate* in Zyzzyva) che una replica ha raccolto. Il tradizionale *Viewchange subprotocol* termina quando il nuovo *Primary*, usando  $2f + 1$  messaggi VIEW-CHANGE da repliche distinte, computa la *History* che tutte le

repliche corrette dovranno adottare nella nuova *view*. Il *Primary* dunque invia un messaggio firmato NEW-VIEW a tutte le repliche, messaggio che include l'*history* calcolata ed una prova della sua validità. Zyzzyva mantiene la struttura di base del protocollo tradizionale, ma si differenzia in due modi che permettono al *Client* di accettare una risposta prima che le repliche sappiano che è stata completata e permettono alle repliche di accettare una risposta dopo due fasi invece delle tradizionali tre. Per prima cosa, per garantire la *liveness*, Zyzzyva introduce una fase "*I hate the primary*" al sottoprotocollo di *Viewchange*. In secondo luogo, per garantire la *safety*, Zyzzyva riduce le condizioni necessarie affinché una richiesta appaia nell'*history* inclusa nel messaggio NEW-VIEW. Poichè in Zyzzyva l'*agreement protocol* garantisce che ogni richiesta completi in una *view* in massimo due fasi, come mostrato nelle figure 2 e 3, è necessario modificare il *Viewchange subprotocol*. Analizzando uno scenario in cui sono presenti  $f$  repliche *faulty*, di cui una è il *Primary*, supponiamo che il *Primary* spinga  $f$  repliche ad iniziare un *Viewchange* ed a smettere di inviare messaggi in questaview. In questa situazione il *Client* potrebbe ricevere solo  $f+1$  risposte dalle rimanenti repliche corrette, non abbastanza da completare la richiesta; in più dato che meno di  $f+1$  repliche richiedono un *Viewchange* non è possibile scegliere un nuovo *Primary* e riottenere *liveness*. In questo caso il problema è dato dal fatto che nei *Viewchange subprotocol* tradizionali una replica può partecipare ad un *Viewchange* anche se questo non garantisce che avverrà effettivamente un *Viewchange*. In Zyzzyva invece una replica non abbandona la *view* attuale a meno che abbia la garanzia che ogni replica corretta faccia lo stesso, forzando una nuova *view* e un nuovo *Primary*. Per garantire questa proprietà Zyzzyva aggiunge una nuova fase al *Viewchange subprotocol*, in modo da rinforzare le condizioni secondo cui una replica smette di partecipare alla *view* corrente. In particolare una replica  $i$  corretta che sospetti che il *Primary* sia *faulty* continua a partecipare alla *view* ma esprime il suo sospetto inviando alle altre repliche un messaggio  $\langle \text{I-HATE-THE-PRIMARY}, v \rangle_{\sigma_i}$  (9). Se  $i$  riceve  $f+1$  messaggi I-HATE-THE-PRIMARY per la *view*  $v$ , allora partecipa ad un *Viewchange* e invia a tutte le repliche un messaggio VIEW-CHANGE che contiene anche la prova che  $f+1$  repliche non si fidano del *Primary* e dunque parteciperanno al *Viewchange*. Una replica che riceve un messaggio VIEW-CHANGE partecipa al *Viewchange*, assicurando così che se una replica corretta partecipa ad un *Viewchange* allora tutte le altre repliche corrette vi parteciperanno prima o poi. Un'altra conseguenza del *Agreement subprotocol* di Zyzzyva è che le repliche potrebbero non sapere mai se una richiesta è stata completata: se un *Client* riceve  $3f + 1$  risposte allora completa la richiesta senza notificarlo alle repliche. Questo rende necessario modificare il *Viewchange subprotocol* in modo da permettere al nuovo *Primary* di calcolare una nuova *History* corretta che includa le richieste eventualmente completate nel *fast path* dell'*Agreement subprotocol*. Il *Viewchange subprotocol* in Zyzzyva è modificato dunque in due modi:

- Le repliche corrette aggiungono al messaggio VIEW-CHANGE tutti i messaggi ORDER-REQ ricevuti a partire dall'ultimo *stable checkpoint*.
- Un nuovo *Primary* corretto estende l'*history* da adottare nella nuova *view* aggiungendo tutte le richieste con un ORDER-REQ il cui *sequence number* sia maggiore del maggior *sequence number* presente nei *commit certificate* presenti nei messaggi

VIEW-CHANGE se queste richieste appaiono in almeno  $f + 1$  dei  $2f + 1$  VIEW-CHANGE che il nuovo *Primary* ha ricevuto.

Questa modifica assicura che il protocollo continui ad onorare l'ordine assegnato alle richieste quando queste vengono completate da un *Client* che ha ricevuto  $3f + 1$  risposte speculative. Questa modifica potrebbe avere però l'effetto di assegnare un ordine ad una richiesta che non è stata ancora completa nella *view* precedente. In particolare una curiosità del protocollo è che, in base a quale gruppo di  $2f + 1$  VIEW-CHANGE viene usato dal *Primary*, potrebbe accadere che per un dato *sequence number* vengano trovate diverse richieste con  $f + 1$  messaggi ORDER-REQ. Questa curiosità però non causa una violazione della *safety*, in quanto possono esistere queste richieste solo se almeno una replica corretta supporta una delle due. In questo caso, nessuna delle due richieste può essere stata completata dal *Client* con  $3f + 1$  risposte corrispondenti e dunque il sistema può assegnare ad una delle due, o a nessuna, quel *sequence number*.

### 3.3.2 Viewchange in Zyzzyva

Il *viewchange subprotocol* in Zyzzyva è suddiviso nei seguenti passi.

VC1. La replica inizia il *view change* inviando una accusa contro il *Primary* a tutte le altre repliche.

La replica  $i$  inizia un *view change* inviando un messaggio  $\langle \text{I-HATE-THE-PRIMARY}, v \rangle_{\sigma_i}$  a tutte le altre repliche, esprimendo il fatto che la replica non è soddisfatta dal comportamento del *Primary*. Nei protocolli precedenti, questo messaggio indicava che la replica  $i$  non avrebbe più partecipato alla *view* corrente, mentre in Zyzzyva è solo un'indicazione che la replica vorrebbe cambiare *view*. Anche dopo aver inviato questo messaggio, infatti, la replica  $i$  continua a partecipare correttamente alla *view* corrente.

VC2. La replica riceve  $f + 1$  accuse che il *Primary* è *faulty* e partecipa al *view change*.

Una replica  $i$  partecipa ad un *view change* che porterà alla *view*  $v + 1$  mandando a tutte le repliche un'accusa contro l'attuale *Primary*, accusa composta da una lista di  $f + 1$  messaggi  $\langle \text{I-HATE-THE-PRIMARY}, v \rangle_{\sigma_j}$  da  $f + 1$  repliche  $j$  distinte e da un messaggio  $\langle \text{VIEW-CHANGE}, v + 1, s, C, CC, O \rangle_{\sigma_i}$  (10).  $O$  è l'*history* della replica  $i$  a partire dall'ultimo *stable checkpoint* con *sequence number*  $s$ .  $C$  è la prova dell'ultimo *stable checkpoint* ed è composto da  $2f + 1$  messaggi *checkpoint*.  $CC$  è il più recente *Commit Certificate* per una richiesta a partire dall'ultimo *view change*.

VC3. La replica riceve  $2f + 1$  messaggi di *view change*.

Quando la replica che sarà il nuovo *Primary* riceve  $2f + 1$  validi messaggi VIEW-CHANGE (inclusendo anche quello mandato da se stesso), allora il nuovo *Primary*  $p$  costruisce il messaggio  $\langle \text{NEW-VIEW}, v + 1, P, G, \sigma_p \rangle_{\sigma_p}$  (11), dove  $P$  è la lista di validi messaggi VIEW-CHANGE ricevuti dal nuovo *Primary* per la *view*  $v + 1$  e  $G$  è la *history* calcolata dal nuovo *Primary*. usando  $P$ , *history* che dovrà essere adottata da tutte le repliche nella nuova *View*. Per calcolare  $G$  il *Primary* procede secondo i seguenti passi:

- Il *Primary* definisce  $min-s$  come il valore dell'ultimo *stable checkpoint* presente nei messaggi *view change* in  $P$ ,  $max-cc$  come il più alto numero di sequenza dei *commit certificate CC* presenti in  $P$ ,  $max-r$  come il più alto numero di sequenza di una richiesta che è potenzialmente stata completata<sup>9</sup> da un *Client* nel *fast path* del sottoprotocollo di *agreement* e  $max-s$  come il più alto numero di sequenza presente nelle *History* all'interno dei messaggi *VIEW-CHANGE* che compongono  $P$ , di modo che  $min-s \leq max-cc \leq max-r \leq max-s$ .
- Il *Primary* aggiunge le richieste completate nella nuova history. Per fare ciò inserisce delle  $\langle \langle \text{ORDER-REQ}, v+1, n, hn, d, ND \rangle_{\sigma_p}, m \rangle$  nella *history* calcolata  $G$  copiando le richieste dall'*history* della replica che ha inviato il *VIEW-CHANGE* con il  $max-cc$ , copiando ogni richiesta il cui *sequence number*  $n$  sia  $min-s < n \leq max-cc$ .
- Il *Primary* aggiunge le richieste potenzialmente completate nella nuova history. Per fare ciò inserisce delle  $\langle \langle \text{ORDER-REQ}, v+1, n, hn, d, ND \rangle_{\sigma_p}, m \rangle$  nella *history* calcolata  $G$  copiando ogni richiesta con *sequence number*  $n$  tale che  $max-cc+1 \leq n \leq max-r$  e che rispettino le seguenti condizioni:
  - La richiesta  $n$  è presente nell'*history* di almeno  $f+1$  repliche distinte con gli stessi *sequence number*  $n$ , *hash* dell'*history*  $hn$ , *hash* della richiesta  $d$ .
  - $h_n = H(h_{n-1}, d)$ .

Questo passaggio è necessario per fare in modo che non si perdano delle richieste che sono state completate quando un *Client* ha ricevuto  $3f+1$  risposte corrispondenti. Va notato infatti che, delle  $3f+1$  repliche che hanno mandato queste risposte corrette, è garantito che almeno  $f+1$  repliche corrette avranno contribuito allo stato raccolto dal nuovo *Primary* durante il *Viewchange subprotocol*, assicurando così che non si perdano richieste quando si passa ad una nuova *view*.

- Per tutte le altre richieste che non ricadono nei gruppi precedenti, il *Primary* ha la certezza che non siano state completate. Procedo dunque ad inserire delle  $\langle \langle \text{ORDER-REQ}, v+1, n, hn, d^{null}, NULL \rangle_{\sigma_p}, NULL \rangle$  nella *history* calcolata  $G$  per ogni richiesta  $n$  tale che  $max-r+1 \leq n \leq max-s$ . Una richiesta nulla verrà semplicemente ignorata quando eseguita.

Quando invece è una replica a ricevere  $2f+1$  messaggi *VIEW-CHANGE* (incluso anche quello mandato da se stesso), fa partire un *timer*. Se la replica non riceve un valido messaggio *NEW-VIEW* dal nuovo *Primary* prima che scada il *timer* inizia un *Viewchange* per la *view*  $v+1$ <sup>10</sup>. La lunghezza del *timer* utilizzato in questa fase cresce esponenzialmente in base al numero di *Viewchange* che falliscono in successione, per essere poi resettato quando un *Viewchange* ha effettivamente successo.

VC4. La replica riceve un valido messaggio *new view*, riconcilia il suo stato locale ed entra nella nuova *view*.

<sup>9</sup>per essere potenzialmente completata, una richiesta deve essere presente in tutte le  $2f+1$  *history* all'interno dei messaggi *VIEW-CHANGE* che compongono  $P$

<sup>10</sup>La Replica passa direttamente allo step VC2 del *Viewchange subprotocol* in questo caso.

Quando ricevono un messaggio NEW-VIEW le repliche, incluso il *Primary*, riconciliano il loro stato locale con quello ricevuto nel messaggio NEW-VIEW, cambiano *view* e iniziano a processare i messaggi nella nuova *view*. In particolare, il *Primary* riconcilia il suo stato locale confrontando la sua *history* con quella  $G$  presente nel messaggio NEW-VIEW. Assumendo come  $max-l$  il *sequence number* dell'ultima richiesta presente nella sua *history* locale:

- Se  $max-l < min-s$ , il *Primary* inserisce il *checkpoint* con *sequence number*  $min-s$  nella sua *history* e in più lo salva come suo *checkpoint*, scarta il contenuto della sua *history* locale e copia il contenuto di  $G$  nella sua *history* a partire dalla richiesta con *sequence number*  $min-s+1$ . Acquisisce inoltre uno Snapshot per  $min-s$  contattando le repliche presenti nei *Checkpoint*. Infine esegue le richieste presenti nella *History* appena creata a partire dalla richiesta con *sequence number*  $min-s+1$ , rispondendo di volta in volta al *Client* che aveva inviato la richiesta. Questo è il caso che sorge quando il *Primary* era rimasto indietro rispetto alle repliche nella *view* precedente.
- Se  $max-l \geq min-s$  e le *digest* dell'*history* presenti nelle richieste con *sequence number*  $max-l$  prese dall'*history* locale e da quella quella  $G$  calcolata sono diverse (oppure se la richiesta con *sequence number*  $max-l$  non è presente in  $G$ ), allora il *Primary* deve fare le stesse azioni spiegate nel punto precedente. Questo è il caso che sorge quando la *history* del *Primary* diverge da quella dello stato globale.
- Se  $max-l \geq min-s$  e le *digest* dell'*history* presenti nelle richieste con *sequence number*  $max-l$  prese dall'*history* locale e da quella quella  $G$  calcolata sono uguali, allora il *Primary* copia il contenuto di  $G$  nella sua *history* a partire dalla richiesta con *sequence number*  $max-l+1$  ed esegue le richieste presenti nell' *history* appena creata a partire dalla richiesta con *sequence number*  $max-l+1$ .

Infine il *Primary* aggiorna il suo *commit certificate* locale usando quello con *sequence number*  $max-cc$  (che è computato usando  $P$  come descritto nello step VC3) ed inizia ad accettare messaggi per la *view*  $v + 1$ .

Quando invece è una replica a ricevere un messaggio NEW-VIEW per la *view*  $v + 1$  verifica che il messaggio sia corretto, dove per corretto si intende un messaggio firmato dal nuovo *Primary*, con validi messaggi VIEW-CHANGE e con una *history*  $G$  computata correttamente dal nuovo *Primary*<sup>11</sup>. Se il messaggio è corretto, riconcilia il suo stato locale ed entra nella nuova *view* eseguendo gli stessi passi, appena descritti, che esegue il *Primary*.

---

<sup>11</sup>per verificare che  $G$  sia corretta, la replica esegue una computazione simile a quella effettuata dal *Primary* nello step VC3



### 3.4 Checkpoint Subprotocol

Il *checkpoint subprotocol* in Zyzzyva è suddiviso nei seguenti passi.

CP1. Quando la replica  $i$  riceve il messaggio *order req* per la richiesta  $CP\_INTERVAL^{th}$  dall'ultimo Checkpoint, la replica invia la risposta speculativa a tutte le altre repliche oltre che al *Client*.

Per efficienza la replica  $i$  non include tutta la risposta speculativa  $r$  ma solo il suo *hash*  $H(r)$

CP2. La replica riceve un *commit certificate* per la richiesta nel  $CP\_INTERVAL^{th}$ , costruisce un messaggio *checkpoint*, e manda il messaggio *checkpoint* a tutte le altre repliche.

Dopo aver ricevuto un *commit certificate* per la  $CP\_INTERVAL^{th}$  richiesta ed averlo processato come descritto nello step 5 dell'*Agreement subprotocol*, la replica  $i$  costruisce un messaggio  $\langle \text{CHECKPOINT}, n, h, a, i \rangle_{\sigma_i}$  (12) e lo manda a tutte le repliche, dove  $n$  è il *sequence number*,  $h$  è la *digest* dell'*History* e  $a$  è la *digest* dell'*application state*. Se non riceve un *commit certificate* per la  $CP\_INTERVAL^{th}$  richiesta, la replica lo crea utilizzando  $2f + 1$  risposte speculative direttamente dalle altre repliche, risposte inviate nello step CP1.

CP3. La replica riceve  $f + 1$  messaggi di *checkpoint* uguali e considera il *checkpoint* stabile.

Dopo aver ricevuto  $f + 1$  messaggi *checkpoint* corrispondenti, la replica  $i$  considera la richiesta stabile, elimina le richieste con *sequence number* minore di  $n$ , dove  $n$  è il *sequence number* della  $CP\_INTERVAL^{th}$  richiesta e salva uno *snapshot* dello stato dell'applicazione.

## 4 Progettazione

Data la natura distribuita del problema e la necessità di gestire diversi comportamenti e funzionalità, il sistema è stato suddiviso in 3 componenti principali:

- **Zyzyva**: è la componente che gestisce tutta la logica del protocollo Zyzyva e contiene lo stato dell'applicazione.
- **ZyzyvagRPC**: è la componente che si occupa di gestire l'interazione tra l'utente ed il sistema. In particolare riceve le richieste dal componente SMRView e le inoltra al componente Zyzyva, per poi mandare il risultato della computazione della richiesta all'utente iniziale.
- **SMRView**: è la componente con cui interagisce l'utente. Fornisce un'interfaccia grafica su cui effettuare delle richieste e comunica con il componente ZyzyvagRPC per inoltrare queste richieste all'applicazione.

Le varie componenti del sistema sono progettate per essere indipendenti le une dalle altre ed essere in esecuzione su nodi diversi della rete e devono pertanto fornire dei meccanismi per permettere la comunicazione e lo scambio di messaggi tra di loro.

### 4.1 Componente Zyzyva

La necessità di utilizzare uno scambio di messaggi asincroni durante le varie fasi del protocollo Zyzyva, ha spinto all'adozione del modello ad attori. Gli attori sono entità reattive che eseguono computazioni solo in risposta alla ricezione di determinati messaggi, messaggi alla cui ricezione sono associati diversi *handler* per la gestione dell'esecuzione. Gli attori permettono inoltre di modellare determinati *behaviour* e di cambiarli in base alla ricezione di determinati messaggi, fattore che permette una migliore divisione dei compiti ed una più semplice suddivisione dei comportamenti nei vari *subprotocol* del protocollo implementato. Per ottenere la distribuzione richiesta dal protocollo Zyzyva, in cui ogni replica è in esecuzione su diversi nodi della rete, il componente Zyzyva del sistema è progettato per essere replicato in rete; ogni replica del componente è in grado di comunicare con le altre. Pertanto il singolo componente Zyzyva è definito come un insieme di attori, insieme che ha al suo interno una gerarchia di attori. La comunicazione possibile è di due tipi:

- Comunicazione intra-replica, in cui gli attori di un insieme possono comunicare tra loro tramite scambio di messaggi.
- Comunicazione inter-replica, in cui attori di insiemi diversi possono comunicare con gli attori di altri insiemi sempre tramite scambio di messaggi.

All'interno del componente Zyzyva sono presenti inoltre la logica di gestione dello stato dell'applicazione, stato che si interfaccia con risorse esterne al sistema, come possono essere file o database, e la logica per la gestione della sicurezza, che si occupa di generare l'*hash digest* quando necessario e fornisce la logica per utilizzare l'*encrypting* asimmetrico

sfruttando l'algoritmo RSA per chiavi pubbliche/private. La gerarchia di attori presente all'interno dell'insieme è la seguente:

- **ReplicaManager**, che è l'attore principale del protocollo Zyzzyva. È in grado di comunicare sia con le altre repliche che con i vari attori client presenti nell'insieme. Inoltre ha al suo interno la seguente gerarchia di attori, utilizzati per gestire lo stato locale dell'applicazione:
  - \* **DatabaseActor**, che è l'attore che si occupa di instradare le varie richieste di modifica o lettura dello stato locale ricevute, richieste che sono inviate come messaggi direttamente dal suo ReplicaManager. Per instradare le richieste ha al suo interno possibili diversi attori figli, figli che risponderanno direttamente al ReplicaManager, ognuno in grado di processare richieste riguardanti tipi di dati diversi<sup>12</sup>:
    - **PersonaActor**, attore che gestisce le richieste riguardanti la struttura dati Persona.
- **ClientManagerActor**, che è l'attore che fa da ponte con il componente ZyzzyvagRPC. Al suo interno ha degli attori figli. **ClientActor**, attori che ricevono le richieste direttamente da ZyzzyvagRPC e che le gestiscono secondo le regole del protocollo Zyzzyva.

È infine presente un attore **Zyzzyva Manager**, attore in esecuzione su un numero variabile di nodi della rete e che si occupa di inizializzare e di mettere in comunicazione i vari insiemi. Ha al suo interno un attore **ClusterListener**, attore che si occupa di notificare l'arrivo di un insieme di attori nella rete. Pertanto l'attore **Zyzzyva Manager** riceve inizialmente messaggi dall'attore ClusterListener, per poi ricevere messaggi da ClientActor e ReplicaManager per iniziarli.

#### 4.1.1 ReplicaManager

L'attore ReplicaManager è il principale attore che implementa il protocollo Zyzzyva. Per poter gestire il protocollo utilizza diversi *Behaviour*, ognuno relativo ai diversi compiti che la Replica in Zyzzyva deve assolvere:

- **Stato iniziale**, in cui una replica esegue le azioni necessarie alla sua inizializzazione.
- **WaitForBrothers**, in cui una replica esegue le azioni necessarie a creare le vie di comunicazione con le altre Repliche e con i ClientActor
- **Replica/Primary**, in cui una replica esegue le azioni descritte nel **Agreement subprotocol** descritto nella sezione 3.2 e nel **Checkpoint subprotocol** descritto nella sezione 3.4.

---

<sup>12</sup>Per lo scopo di test del progetto, è stato sviluppato solo il tipo di dati Persona; per gestire tipi diversi bisogna semplicemente aggiungere un attore in grado di lavorarci.

- **ViewChangeStage**, in cui una replica esegue le azioni descritte nel **Viewchange subprotocol** descritto nella sezione 3.3. Questo *Behaviour* è a sua volta composto da altri *Behaviour*:
  - **SnapshotState**, in cui una Replica esegue le azioni necessarie ad acquisire e salvare in locale uno *Snapshot* dello stato dell'applicazione di altre Repliche corrette.
  - **ReconciliateState**, in cui una replica esegue le eventuali richieste speculative necessarie a riconciliare il suo stato locale, come descritto nello step VC4 della sezione 3.3.2 del **Viewchange subprotocol** descritto nella sezione 3.3.

In seguito sono approfonditi i vari *behaviour*.

**Stato Iniziale** Nel suo stato iniziale una Replica crea il proprio attore figlio DatabaseActor e la sua chiave privata, necessaria alla firma dei messaggi inviati. Successivamente rimane in attesa di un messaggio CLUSTER-READY da parte dell'attore Zyzzyva Manager. Alla ricezione del messaggio CLUSTER-READY, la replica invia all'attore Zyzzyva Manager un messaggio REPLICATION-INIT-MESSAGE e rimane in attesa di un messaggio REPLICATION-NUMBER-MESSAGE, che utilizzerà per settare il suo id interno al protocollo Zyzzyva e la *view* in cui partirà ad eseguire il protocollo. Alla ricezione del REPLICATION-NUMBER-MESSAGE, la replica adotta il *behaviour* **WaitForBrothers**.

**WaitForBrothers** In questo *behaviour* una replica rimane in attesa del messaggio REPLICAS-LIST-MESSAGE, messaggio che utilizzerà per creare i canali di comunicazione con le altre repliche e per salvare le chiavi pubbliche di ogni replica interna al sistema. Il messaggio REPLICAS-LIST-MESSAGE può arrivare in modo diverso in base a quale dei seguenti due scenari hanno portato la replica allo stato **WaitForBrothers**:

- Primo avvio del sistema: quando il sistema viene avviato per la prima volta, la replica riceverà il messaggio REPLICAS-LIST-MESSAGE quando tutte le repliche necessarie all'inizializzazione saranno arrivate nel sistema.
- Arrivo di una replica a sistema già avviato: quando una replica viene inizializzata in ritardo rispetto alle altre, ad esempio in caso di un riavvio dopo un malfunzionamento, prima di ottenere un messaggio REPLICAS-LIST-MESSAGE la replica riceverà un messaggio CLIENT-LIST-MESSAGE, messaggio che le permetterà di ottenere la lista dei ClientActor e delle relative chiavi pubbliche attive nel sistema. Dopo la ricezione del messaggio CLIENT-LIST-MESSAGE, la replica procederà all'invio di un messaggio GET-ANOTHER-REPLICAS al Zyzzyva Manager, che sarà seguito poi dal REPLICAS-LIST-MESSAGE.

Dopo aver eseguito le operazioni richieste dal messaggio REPLICAS-LIST-MESSAGE, la replica passa al *behaviour* **Primary** o **Replica**. La Replica decide quale dei due *behaviour* assumere in base alla *view* in cui si trova.

**Replica/Primary** Nei *behaviour* Replica e *Primary*, la replica esegue l' **Agreement subprotocol** descritto nella sezione 3.2 ed il **Checkpoint subprotocol** descritto nella sezione 3.4. È in grado inoltre di gestire la fase VC1 del **Viewchange subprotocol** descritto nella sezione 3.3<sup>13</sup>. Le principali differenze tra i due *behaviour* sono:

- I messaggi I-HATE-THE-PRIMARY e PROOF-OF-MISBEHAVIOUR non vengono né inviati né ricevuti dal *behaviour* Primary ma solo dal *behaviour* Replica.
- Il *Primary* non ha modo di inviare un messaggio FILL-HOLE, a differenza delle repliche. La gestione della ricezione di un messaggio FILL-HOLE da parte di un *Primary* o di una Replica procede come spiegato nell' *Agreement subprotocol*.

Se in questa fase una Replica riceve  $2f + 1$  I-HATE-THE-PRIMARY, invia a tutte le altre repliche ed al *Primary* un messaggio VIEW-CHANGE-COMMIT e passa al *behaviour* **View-ChangeStage**. Se una Replica o il *Primary* riceve un messaggio VIEW-CHANGE-COMMIT passa anche lei al *behaviour* **ViewChangeStage**.

**ViewChangeStage** In questo *behaviour* una replica esegue i passi VC2, VC3 e VC4 del **Viewchange subprotocol** descritto nella sezione 3.3. Dopo la ricezione da parte del nuovo *Primary* del messaggio NEW-VIEW la replica può assumere diversi comportamenti in base a quanto descritto nello step VC4 del **Viewchange subprotocol**. Se è necessario ottenere uno *snapshot* dello stato dell'applicazione, la replica passa al *behaviour* **SnapshotState** dopo aver inviato alle repliche presenti nel *checkpoint* il messaggio ASK-SNAPSHOT, mentre per riconciliare la propria *history* locale ed il proprio stato locale passa al *behaviour* **ReconciliateState**. Dopo aver terminato il *Viewchange subprotocol*, le repliche tornano al *behaviour* Replica/Primary.

**SnapshotState** Se una Replica assume questo *behaviour*, si limita ad attendere una risposta al messaggio ASK-SNAPSHOT inviato precedentemente, risposta che è rappresentata dal messaggio SNAPSHOT-REPLY. Quando una replica riceve il messaggio SNAPSHOT-REPLY, lo inoltra al suo DatabaseActor, da cui riceverà un messaggio SNAPSHOT-SAVE che conferma l'avvenuto aggiornamento dello stato locale tramite *snapshot* ricevuto da una replica corretta. Dopo questo passaggio, se la replica ha necessità di eseguire delle ORDER-REQ per allinearsi alle altre repliche passa al *behaviour* *ReconciliateState*, altrimenti torna direttamente al *behaviour* Replica/Primary.

**ReconciliateState** Se una Replica assume questo *behaviour*, procederà ad eseguire tutte le ORDER-REQ che le mancano per essere allineata allo stato globale calcolato dal nuovo *Primary* durante il *Viewchange subprotocol*. Dopo aver eseguito tutte le ORDER-REQ necessarie, torna al *behaviour* Replica/Primary.

---

<sup>13</sup>a partire da questa fase la replica firma ogni messaggio che invia, utilizzando la logica di gestione sicurezza, con la propria chiave privata, così da fornire a chiunque li riceva la prova che il messaggio è stato effettivamente inviato da lei.

## 4.2 Componente ZyzzyvagRPC

La componente ZyzzyvagRPC verrà costruita in modo tale da poter accettare sia messaggi da parte del SMRView ed inoltrarli alla componente Zyzzyva che per poter accettare dei messaggi provenienti da Zyzzyva ed inoltrarli verso SMRView, tutto questo in un modo asincrono, permettendo così di sfruttare al massimo le risorse della macchina dove questa componente sarà allocata.

Questa componente avrà due sotto componenti che per semplicità chiameremo *DestinatariogRPC* e *Attore Persona*<sup>14</sup>, ognuna con diverse funzioni. *DestinatariogRPC* avrà il compito di ricevere tutti messaggi provenienti da SMRView e trasformarli in un nuovo messaggio che invierà all'*Attore Persona*, che si occuperà di inoltrarlo a Zyzzyva. Quando Zyzzyva risponde all'*Attore Persona*, questo inoltrerà la risposta ricevuta ad *DestinatariogRPC*, in modo tale che *DestinatariogRPC* possa trasformare questo messaggio e inviarlo a SMRView.

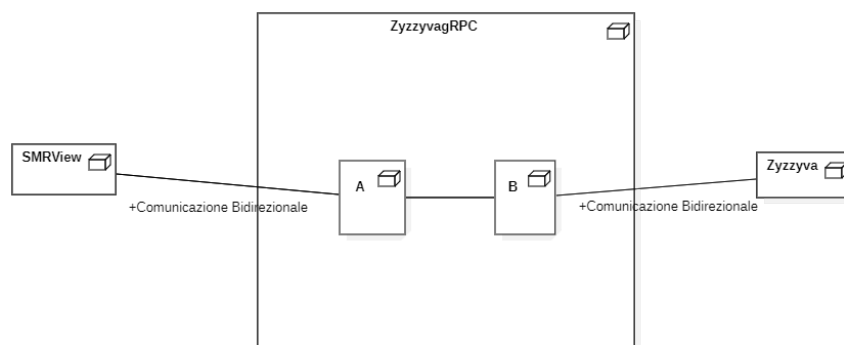


Figura 4: Comunicazione tra *DestinatariogRPC* e l'*Attore Persona*

L'immagine 4 fa vedere l'interazione dall'esterno con *DestinatariogRPC*, l'interazione tra *DestinatariogRPC* e l'*Attore Persona* e l'interazione tra l'*Attore Persona* e Zyzzyva. Inoltre i messaggi che possono essere trattati da ognuna di queste sottocomponenti è descritto in seguito:

I messaggi che può ricevere *DestinatariogRPC* da SMRView sono:

- **WRITE-REQUEST** : permette di realizzare delle operazioni di scrittura su una persona.
- **READ-REQUEST** : permette di realizzare delle operazioni di lettura da una o multiple persone.
- **SET-BYZANTINE-REQUEST** : permette di poter rendere una replica in Zyzzyva difettosa<sup>15</sup>.

<sup>14</sup>*AttorePersona* è un ponte che permette la comunicazione tra *DestinatariogRPC* e Zyzzyva e viceversa

<sup>15</sup>Questa funzionalità esiste per poter testare il sistema, simulando un caso bizantino

I messaggi appena elencati permettono di realizzare delle operazioni su una persona; però per poter rendere tali operazioni possibili, bisogna che *DestinatariogRPC* converta tali messaggi in uno dei messaggi che l'*Attore Persona* è in grado di gestire. I messaggi che *DestinatariogRPC* manda all'*Attore Persona* sono:

- DELETE-PERSONA : è una operazione che fa parte delle operazioni di WRITE-REQUEST e permette di eliminare una persona del sistema.
- INSERT-PERSONA : è una operazione che fa parte delle operazioni di WRITE-REQUEST e permette di inserire una persona del sistema.
- UPDATE-PERSONA : questo messaggio permette di poter aggiornare le informazioni di una persona esistente nel sistema. Fa parte delle operazioni di WRITE-REQUEST.
- READ-PERSONA : questo messaggio permette di poter tornare una persona esistente nel sistema. Fa parte delle operazioni di READ-REQUEST.
- READALL-PERSONA : questo messaggio permette di poter ottenere tutte le persone esistenti nel sistema. Fa parte delle operazioni di READ-REQUEST.
- SET-BYZANTINE-REQUEST : questo messaggio permette di poter rendere una replica nel sistema *faulty*, a differenza degli altri messaggi, questo messaggio è del tipo *request/response*.

Quando l'*Attore Persona* riceve un messaggio da *DestinatariogRPC*, lo inoltra direttamente a Zyzzyva. I messaggi che può ricevere l'*AttorePersona* da Zyzzyva sono:

- DELETE-PERSONA-RESPONSE : questo messaggio contiene il risultato dell'operazione DELETE-PERSONA. Fa parte delle operazioni WRITE-RESPONSE
- INSERT-PERSONA-RESPONSE : questo messaggio contiene il risultato dell'operazione INSERT-PERSONA. Fa parte delle operazioni WRITE-RESPONSE
- UPDATE-PERSONA-RESPONSE : questo messaggio contiene il risultato dell'operazione UPDATE-PERSONA. Fa parte delle operazioni WRITE-RESPONSE
- READ-PERSONA-RESPONSE : questo messaggio contiene il risultato dell'operazione READ-PERSONA. Fa parte delle operazioni READ-RESPONSE
- READALL-PERSONA-RESPONSE : questo messaggio contiene il risultato dell'operazione READALL-PERSONA. Fa parte delle operazioni READ-RESPONSE
- SET-BYZANTINE-RESPONSE : questo messaggio rappresenta la risposta per il messaggio SET-BYZANTINE-RESPONSE, che indica se la operazione è andata a buon fine.

Una volta che Zyzzyva risponde all'*Attore Persona*, questo inoltra il messaggio direttamente a *DestinatarioRPC*, che internamente si occuperà di convertirlo nel formato corrispondente. I messaggi con cui risponde *DestinatarioRPC* verso SMRView sono:

- **WRITE-RESPONSE** : questo messaggio contiene le possibili risposte alle operazioni di **WRITE-REQUEST**.
- **READ-RESPONSES** : questo messaggio contiene le possibili risposte alle operazioni di **READ-REQUEST**.
- **SET-BYZANTINE-RESPONSE** : messaggio che contiene la risposta all'operazione di **SET-BYZANTINE-REQUEST**.

### 4.3 Componente SMRView

Questa componente ha come scopo principale permettere di testare il sistema sia nel caso ideale, cioè quando tutte le repliche hanno un corretto funzionamento, sia nel caso in cui una replica è bizantina. Tutti i messaggi che sono stati descritti nella sottosezione precedente 4.2 vengono creati da questa componente.

SMRView ci permette di eseguire richieste verso sistema, indicando inoltre se l'operazione richiesta è andata a buon fine. Le operazioni appartenenti sia a **WRITE-REQUEST**, che a **READ-REQUEST** verranno gestite in questa componente, così come le loro rispettive risposte. Infine ci permetterà di poter rendere  $n$  repliche bizantine, per permettere un più semplice test del caso in cui al massimo  $f$  repliche diventino bizantine.

### 4.4 Struttura

Il sistema Zyzzyva, come esposto nella sezione 4, è stato suddiviso in 3 componenti principali. L'immagine 5 mostra come questi componenti sono organizzati internamente, esponendo i componenti che li compongono, componenti spiegati nelle sezioni precedenti. L'immagine 6 invece mostra l'architettura finale del sistema una volta che le singole componenti vengono eseguite e rilasciate sulle diverse macchine che compongono il sistema.

L'immagine 6 mostra l'insieme di sistemi che compongono il sistema. L'immagine mostra un sistema composto da 4 Repliche, ognuna rappresentata da un Docker Host in cui è presente un Actor System. Sono presenti inoltre un Docker Host in cui viene eseguito il Componente 4.2 ed un Docker Host in cui è eseguito un Load Balancer per distribuire le richieste tra eventuali multipli Componenti 4.2 (Il Balancer scelto in questa rappresentazione è nginx [6]). È infine mostrata un'istanza del Componente SMRView.



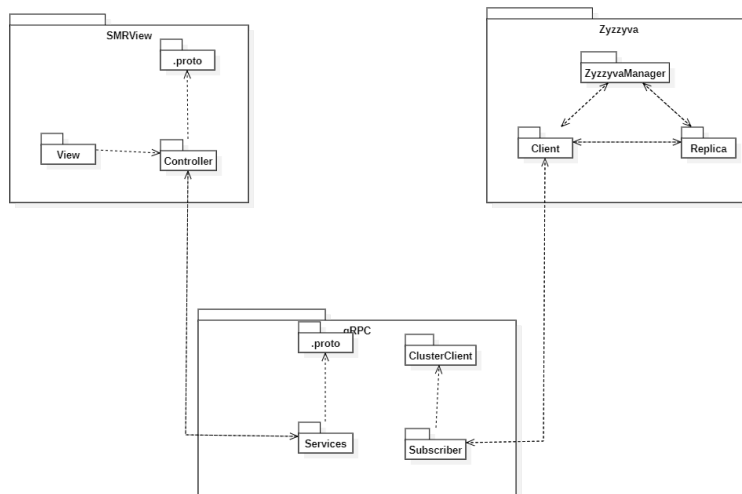


Figura 5: Zyzzyva Package Diagram

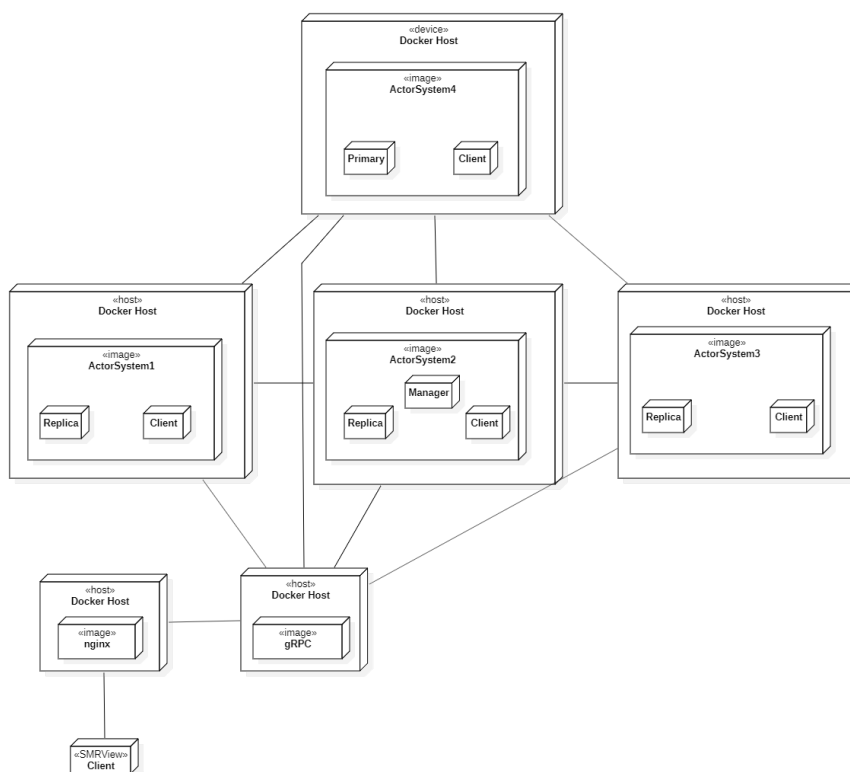


Figura 6: Zyzzyva Deployment Diagram

## 4.5 Interazione

Data la natura distribuita del sistema, le entità che lo compongono interagiscono principalmente tramite uno scambio di messaggi, messaggi che sono per lo più asincroni. L'immagine 7 in particolare rappresenta l'interazione generale tra le varie componenti del sistema, mentre l'immagine 8 mostra l'interazione interna del Componente Zyzzyva, in particolare evidenziando la comunicazione che avviene durante il *fast case* dell'*Agreement protocol* descritto in 2 della paragrafo 3.2. (UML Sequence Diagram)

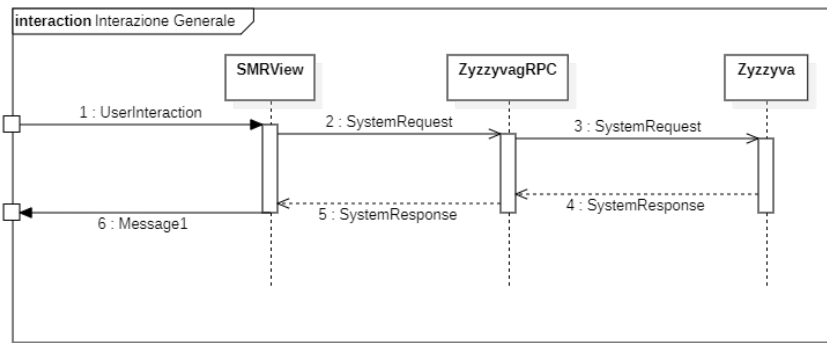


Figura 7: Interazione Generale Zyzzyva.

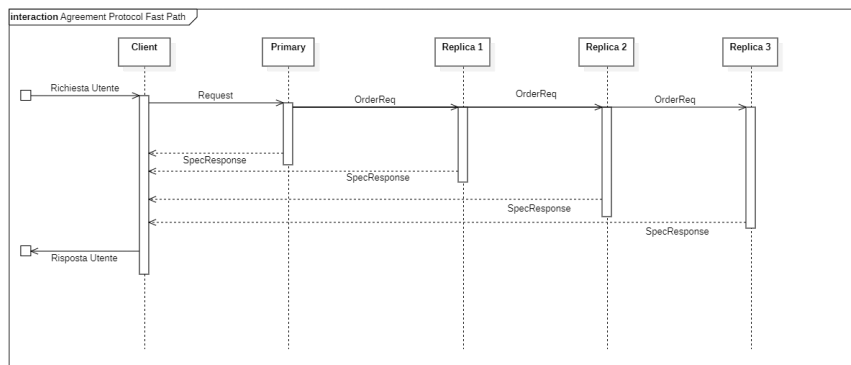


Figura 8: Interazione Agreement Protocol Fast Case.

## 5 Dettagli di Implementazione

In questa sezione vengono individuate tre sottosezioni, in cui nella prima vengono descritte le scelte tecnologiche fatte per sviluppare Zyzyva, nella seconda vengono spiegati alcuni dettagli di implementazione, mentre nell'ultima vengono descritti gli steps necessari per poter produrre la documentazione del progetto.

### 5.1 Scelte Tecnologiche

#### 5.1.1 Linguaggio

Per quanto riguarda il linguaggio in cui il sistema è stato sviluppato, la scelta è ricaduta su C#. Questa scelta è fondata sul grande supporto che riceve il linguaggio, la disponibilità di integrazioni con varie tecnologie, la presenza di documentazione e sul supporto al *cross-platform*.

#### 5.1.2 Google Remote Procedure Call (gRPC)

Per quanto riguarda la comunicazione tra il componente SMRView, in esecuzione sulla macchina locale dell'utente, ed il componente Zyzyva, in esecuzione in rete in maniera distribuita è stato implementato il componente ZyzyvagRPC. ZyzyvagRPC è in grado di comunicare con Zyzyva tramite un attore, mentre per ricevere i messaggi da SMRView utilizza il *framework* gRPC[19]. gRPC è un'implementazione di RPC, inizialmente sviluppata da Google ed ora *open source*. Sebbene l'uso di gRPC abbia bisogno di più codice e sforzi per poterlo implementare dalla parte del *Client*, permette una maggiore flessibilità in confronto al classico REST.

#### 5.1.3 Akka

Per utilizzare il modello ad attori, la scelta che abbiamo fatto è stato per l'utilizzo di akka, in particolare [2]. Akka.net è un *porting* del *framework* akka[1], che nasce come libreria scritta in scala. Akka è una delle implementazioni più conosciute e supportate del paradigma ad attori, che offre diversi moduli come, ad esempio, *Akka Actors* *Akka Cluster* o *Akka Streams*. Il modulo principale, modulo che è stato ampiamente utilizzato nell'implementazione del sistema, è *Akka Actors*, modulo che fornisce un'implementazione del paradigma ad attori. Per quanto riguarda la distribuzione del sistema, abbiamo utilizzato il modulo *Akka Cluster*, che gestisce la formazione di un *Cluster* di *ActorSystem*<sup>16</sup> in grado di comunicare tra loro nella rete in maniera trasparente, senza dover cioè conoscere la posizione fisica del sistema verso cui si vuole comunicare, comunicazione trasparente che è una delle caratteristiche principali del modello ad attori.

---

<sup>16</sup>Gruppo gerarchico di attori che condividono una impostazione comune

#### 5.1.4 Avalonia

Sapendo che *Windows Presentation Foundation* (WPF) [7] è il *framework* più famoso di Microsoft per creare delle interfacce grafiche utilizzando C#, è stata la prima opzione considerata per sviluppare il componente SMRView. Il problema principale di WPF però è che l'applicazione che viene creata è eseguibile solo su sistemi operativi *Windows*. Per poter creare un'interfaccia *cross-platform*, eseguibile su ogni sistema operativo, abbiamo dunque optato per l'utilizzo del *framework* Avalonia, definito dagli autori come "*A cross platform XAML framework for .NET*" e che è utilizzato per creare delle GUI eseguibili sia su sistemi operativi *Windows* che su *Mac* e *Linux*. Per lo sviluppo delle interfacce Avalonia utilizza un dialetto del linguaggio di *markup* XAML, rendendo così il *framework* molto simile a WPF e quindi facile da utilizzare se si hanno delle conoscenze di WPF.

#### 5.1.5 NGINX

Per simulare il fatto di sviluppare un sistema reale, abbiamo scelto di aggiungere al sistema un *load balancer*, per gestire le chiamate web ricevute dall'applicazione. La scelta che abbiamo fatto è stata NGINX, che è tra le altre cose un server HTTP ed un *reverse proxy open source* [6]. L'utilizzo che ne abbiamo fatto è stato quello di *reverse proxy*, che riceve le chiamate da parte dell'utente e le inoltra al componente 4.2, bilanciando le chiamate tra eventuali componenti multipli 4.2 distribuiti nella rete.

### 5.2 Dettagli implementativi

In questa sottosezione verranno spiegate alcune scelte implementative particolari usate nello sviluppo del progetto.

#### 5.2.1 Zyzyva

Per implementare l'insieme di attori descritto nella sezione Progettazione abbiamo sfruttato l'*ActorSystem* di Akka, per cui ogni insieme di attori descritto nella sezione Progettazione è rappresentato da un *ActorSystem*. In seguito verranno elencate le diverse scelte fatte nell'implementazione del componente Zyzyva:

- Per quanto riguarda l'aspetto distribuito del componente, abbiamo utilizzato Akka Cluster<sup>17</sup>, sfruttando le capacità del *framework* di gestire un sistema di vari *ActorSystem* distribuiti nella rete in maniera trasparente al programmatore.
- Per comunicare tra loro gli attori utilizzano degli speciali attori *Router*, attori che si comportano come *Router* e che vengono creati utilizzando il file di configurazione *.hocon* del Cluster.
- Gli attori *ReplicaManager* e *ClientActor* utilizzano un *Router* creato a livello codice, dato che hanno bisogno di distinguere per ogni *view* tra il *Primary* e le Repliche e poter comunicare con loro in maniera adeguata.

---

<sup>17</sup>Per dettagli più precisi si può vedere, Creazione Akka Cluster

- Lo scambio di messaggi tra gli attori avviene utilizzando delle classi immutabili per evitare eventuali *side-effect* dovuti alla modifica dei valori inviati nel messaggio.
- Per gestire la computazione delle `ORDER-REQ` ogni attore `ReplicaManager` utilizza due code, una per le richieste ordinate ricevute dal *Primary* e una per gestire le richieste `ORDER-REQ-FILL-HOLE` ricevute in seguito all'invio di un messaggio `FILL-HOLE`; queste code sono necessarie per garantire che al `DatabaseActor` venga inviata solo una richiesta alla volta. Senza l'utilizzo di queste code, data la natura asincrona degli attori e dello scambio di messaggi, le richieste potrebbero essere eseguite in ordine diverso rispetto al loro inserimento nella *history*, dato che il `DatabaseActor` potrebbe ricevere multipli messaggi `ORDER-REQ` dal suo `ReplicaManager` potenzialmente in ordine diverso rispetto a quanto stabilito dal *Primary*, portando ad un errato comportamento del componente.

### 5.2.2 gRPC

Come descritto nella sottosezione `Componente ZyzzyvagRPC` e `Componente SMRView`, queste due comunicano usando gRPC, dove `SMRView` fa da Client e `ZyzzyvagRPC` fa da Server. La comunicazione tra queste componenti avviene in modo bidirezionale usando un *AsyncStreamReader* e *AsyncStreamWriter*. L'uso di una comunicazione bidirezionale ci permette di evitare di creare una connessione nuova per ogni chiamata effettuata dal Client, riutilizzando la connessione aperta e diminuendo lo spreco di risorse che ci sarebbe creando ogni volta una nuova connessione.

#### ZyzzyvagRPC

- Usa l'*AsyncStreamReader* per leggere le richieste fatte da `SMRView`.
- Usa l'*AsyncStreamWriter* per scrivere le risposte a `SMRView`.
- Usa un *ClusterClient* per comunicare con il componente `Zyzzyva` in modo trasparente sfruttando l'invio di messaggi degli attori.

#### SMRView

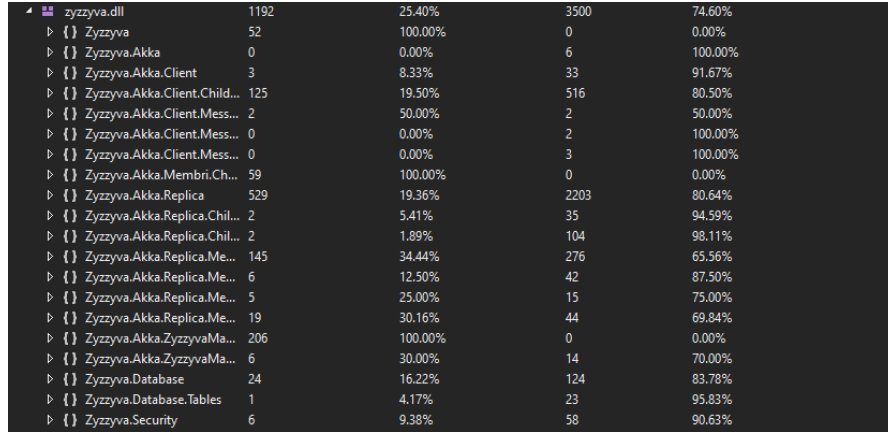
- Usa l'*AsyncStreamReader* per scrivere le richieste a `ZyzzyvagRPC`.
- Usa l'*AsyncStreamWriter* per leggere le risposte fatte da `ZyzzyvagRPC`.
- Evita che la connessione venga chiusa facendo dei *KeepAlivePingDelay* per evitare di chiudere la connessione con `ZyzzyvagRPC`.

### 5.3 Documentazione

Visto che stiamo usando C#, quando il progetto viene compilato il compilatore crea dei file XML per la documentazione del codice. Questi file XML però non sono del tutto user-friendly, motivo per cui abbiamo fatto la scelta di usare DocFX [4], *tool* che ci permette di rendere la documentazione creata dal compilatore più leggibile, creando un progetto che conterrà tutta l'informazione contenuta nei file XML appena descritti. Nel Appendice verrà spiegato il modo in cui si genera la documentazione del progetto. La documentazione del progetto si può trovare alla pagina [zyzzyva.surge.sh](http://zyzzyva.surge.sh)

## 6 Autovalutazione / Validazione

L'immagine seguente mostra la *coverage* raggiunta nei test realizzati sulla componente Componente Zyzzyva.



zyzzyva.dll	1192	25.40%	3500	74.60%
▸ {} Zyzzyva	52	100.00%	0	0.00%
▸ {} Zyzzyva.Akka	0	0.00%	6	100.00%
▸ {} Zyzzyva.Akka.Client	3	8.33%	33	91.67%
▸ {} Zyzzyva.Akka.Client.Child...	125	19.50%	516	80.50%
▸ {} Zyzzyva.Akka.Client.Mess...	2	50.00%	2	50.00%
▸ {} Zyzzyva.Akka.Client.Mess...	0	0.00%	2	100.00%
▸ {} Zyzzyva.Akka.Client.Mess...	0	0.00%	3	100.00%
▸ {} Zyzzyva.Akka.Membri.Ch...	59	100.00%	0	0.00%
▸ {} Zyzzyva.Akka.Replica	529	19.36%	2203	80.64%
▸ {} Zyzzyva.Akka.Replica.Chil...	2	5.41%	35	94.59%
▸ {} Zyzzyva.Akka.Replica.Chil...	2	1.89%	104	98.11%
▸ {} Zyzzyva.Akka.Replica.Me...	145	34.44%	276	65.56%
▸ {} Zyzzyva.Akka.Replica.Me...	6	12.50%	42	87.50%
▸ {} Zyzzyva.Akka.Replica.Me...	5	25.00%	15	75.00%
▸ {} Zyzzyva.Akka.Replica.Me...	19	30.16%	44	69.84%
▸ {} Zyzzyva.Akka.ZyzzyvaMa...	206	100.00%	0	0.00%
▸ {} Zyzzyva.Akka.ZyzzyvaMa...	6	30.00%	14	70.00%
▸ {} Zyzzyva.Database	24	16.22%	124	83.78%
▸ {} Zyzzyva.Database.Tables	1	4.17%	23	95.83%
▸ {} Zyzzyva.Security	6	9.38%	58	90.63%

Figura 9: Zyzzyva Testing

Per poter realizzare il testing di Componente Zyzzyva, si sono realizzati degli Unit Test che sono dei test di tipo *White Box*, in specifico si è fatto utilizzo del framework xUnit, test che ci hanno permesso di poter definire ogni test come un *Fact*, che serve per testare condizioni invarianti. Dopo aver deciso quali erano le componenti principali da testare, si è concordato che sul fatto che alcuni dei blocchi non sarebbero stati testati, come ad esempio il **Main**. Focalizzandoci dunque solo sulla parte considerata da noi più importante, si è deciso di testare gli attori *ReplicaManager* e *ClientActor*. La scelta di questi due particolari attori è dovuta al fatto che, per quanto riguarda il Componente Zyzzyva, essi sono i due attori che incapsulano la logica di funzionamento del protocollo Zyzzyva. Come si può vedere nella immagine 9, la percentuale di *coverage* raggiunta sia nel *ClientActor* che nel *ReplicaManager* è abbastanza soddisfacente, anche basandoci sulle guide linee di google [3]<sup>18</sup>.

<sup>18</sup>Si deve anche tenere in considerazione che ci sono blocchi di codice che fanno riferimento al fatto di permettere far diventare una replica bizantina, comportamento che è stato incluso nel sistema solo per renderlo testabile tramite il 4.3. La possibilità di simulare una replica bizantina infatti viene naturale con l'utilizzo di Unit Test

## 7 Istruzioni di Deployment

Per eseguire l'applicazione, è necessario l'utilizzo di docker. In particolare è stato preparato un file docker-compose per permettere un rapido deploy di tutti i componenti necessari al funzionamento del sistema. Utilizzando questo docker-compose, verranno scaricati da *dockerhub* le immagine già pronte per essere utilizzate. In alternativa, è possibile utilizzare il file docker-compose utilizzato durante lo sviluppo: in questo caso bisogna eseguire il file all'interno della radice del progetto, file che provvederà a creare i contenitori necessari al funzionamento del sistema e li farà partire. Per quanto riguarda la componente SMR di test, sono presenti due versioni dell'applicazione: una per sistemi windows ed una per sistemi linux ubuntu. Per eseguire l'applicazione è sufficiente estrarre il contenuto della versione desiderata ed eseguire il file *SMRViewZyzzva* presente all'interno della cartella estratta. Per poter eseguire il componente *SMRViewZyzzva* direttamente dal progetto, è necessario avere installato dotnet ed eseguire, all'interno della cartella *SMRViewZyzzva*, il comando *dotnet run*. L'ultimo step necessario è l'aggiunta del certificato pfx fornito all'elenco dei certificati trusted del proprio pc: per poter utilizzare grpc tramite https è infatti necessaria la presenza di un certificato ed abbiamo deciso di crearne uno da utilizzare durante lo sviluppo ed il testing dell'applicazione.



## 8 Conclusioni

Lo sviluppo di questo progetto ci ha portato a confrontarci con lo studio, analisi ed implementazione di un meccanismo di consenso in un sistema distribuito con possibilità di fallimenti bizantini. L'implementazione di un algoritmo per la gestione di sistemi di questo tipo si è rivelata più ostica di quanto previsto, portandoci all'utilizzo di un quantitativo di tempo ed energie superiore a quanto inizialmente previsto. Inoltre la spiegazione dell'algoritmo preso in esame risulta poco chiara in alcuni punti dei paper [15] [16] [17] presi in esame per studiarlo, fattore che ci ha costretto ad implementare alcuni meccanismi secondo le nostre interpretazioni della spiegazione. In ultimo, nel paper [9] gli stessi autori dell'algoritmo mettono in luce, a distanza di anni, alcuni problemi riguardo la *safety* dell'algoritmo. Nonostante queste difficoltà riteniamo di aver compreso ed implementato correttamente quanto descritto nei *paper*.

### 8.1 Lavori Futuri

I possibili sviluppi futuri del lavoro sono per la maggior parte orientati al miglioramento delle prestazioni dell'algoritmo, in particolare:

- Utilizzo di un *incremental hashing* laddove necessario, ad esempio per quanto riguarda l'*hashing* della *history*.
- Semplificazione dei meccanismi di firma e verifica dei messaggi per evitare di utilizzare troppo spesso il meccanismo di chiave pubblica-privata fornito da RSA, che con l'aumento dei bit utilizzati per le chiavi peggiora considerevolmente le prestazioni del sistema.
- Miglioramento dell'utilizzo degli *snapshot* per quanto riguarda la fase di *view change*, per aumentare le possibilità che una replica precedentemente bizantina e poi tornata corretta torni consistente con le altre repliche del sistema.
- Aggiunta della possibilità di cambiare il numero di repliche utilizzate dal sistema a *runtime*.

### 8.2 Cosa abbiamo imparato

- Nel corso del progetto abbiamo imparato ad utilizzare in maniera più efficace il modello ad attori. In particolare abbiamo compreso meglio l'utilizzo del *cluster* di akka e dei meccanismi per la comunicazione intra-cluster, come ad esempio i *router*.
- Abbiamo imparato a lavorare in maniera più precisa con docker, in particolare con le varie possibili configurazioni utilizzabili, la comunicazione tra vari container e il *deploy* di sistemi più complessi tramite l'uso di docker-compose.
- Abbiamo imparato, anche se in maniera non approfondita, l'utilizzo di nginx come *load balancer* e delle configurazioni necessarie a farlo funzionare con gRPC.

- Abbiamo imparato ad utilizzare gRPC, soprattutto per quanto riguarda l'uso di *streaming* ed il mantenimento di connessioni http2 sicure e senza interruzioni, per limitare lo spreco di risorse necessario all'apertura di nuove connessioni per ogni operazione.

## Riferimenti bibliografici

- [1] Akka. <https://akka.io/docs/>. Accessed: 2021-01-18.
- [2] Akka.NET. <https://getakka.net/index.html>. Accessed: 2021-01-18.
- [3] Code Coverage Best Practices . <https://testing.googleblog.com/2020/08/code-coverage-best-practices.html>. Accessed: 2021-01-28.
- [4] DocFX An extensible and scalable static documentation generator. <https://dotnet.github.io/docfx/>. Accessed: 2021-01-18.
- [5] gRPC. <https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/grpc>. Accessed: 2021-01-18.
- [6] Nginx Part of F5. <https://www.nginx.com/>. Accessed: 2021-01-18.
- [7] What is Windows Presentation Foundation (WPF .NET). <https://docs.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-5.0>. Accessed: 2021-01-18.
- [8] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.
- [9] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance. *arXiv preprint arXiv:1712.01367*, 2017.
- [10] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.
- [11] Jean Dollimore Coulouris George and Tim Kindberg. The byzantine generals problem. In *Distributed Systems: Concepts and Design*, page 660. Addison-Wesley, 5rd edition, 2001.
- [12] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Sh-rira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190, 2006.
- [13] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.
- [14] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

- [15] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 45–58, 2007.
- [16] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 27(4):1–39, 2010.
- [17] Ramakrishna Kotla, Allen Clement, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. Zyzzyva: Speculative byzantine fault tolerance. *Commun. ACM*, 51(11):86–95, November 2008.
- [18] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the Works of Leslie Lamport*, pages 203–226. 2019.
- [19] Louis Ryan. grpc motivation and design principles, Sep 2015.
- [20] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

## Appendice

Messaggi della Replica:

- $\langle \text{ASK-SNAPSHOT} \rangle$  : messaggio che permette di ottenere uno snapshot dello stato sistema.
- $\langle \text{CONFIRM-REQ-TIMER-END} \rangle$  : messaggio che notifica la fine del timer utilizzato per la confirm-req.
- $\langle \text{FILL-HOLE-TIMER-END}, \text{fillHole} \rangle$  : messaggio che viene inviato quando il timer che è stato settato per la fill-hole scade, il parametro ci permette di poter rinviare la fill-hole a tutte le repliche.
- $\langle \text{FILL-HOLE-TIMER-END-2} \rangle$  : messaggio che indica che il timer settato per la fill-hole quando è stata inviata a tutte le repliche è scaduto, ci permette di poter fare un I-HATE-THE-PRIMARY.
- $\langle \text{ORDER-REQ-FILL-HOLE}, \text{orderReq} \rangle$  : messaggio che viene inviato in risposta ai messaggi FILL-HOLE.
- $\langle \text{RICONCILIATE-MESSAGE} \rangle$  : messaggio che permette di poter richiedere uno snapshot se questo non mi è arrivato oppure se sto per iniziare un riconcilio.
- $\langle \text{REPLICA-SNAPSHOT}, \text{replicasId} \rangle$  : messaggio che contiene le repliche che fanno parte di un CHECKPOINT, in modo tale da poter chiedere il loro snapshot per riconciliare.
- $\langle \text{REVERT-ACTION} \rangle$  : messaggio che permette poter eliminare un'ultima richiesta effettuata da una replica se questa non è stata completata.
- $\langle \text{SNAPSHOT-REPLY}, \text{streamReader} \rangle$  : messaggio inviato dopo che una replica ha inviato un ASK-SNAPSHOT, questo messaggio contiene tutte le informazioni riferite allo snapshot di una replica.
- $\langle \text{SNAPSHOT-SAVE} \rangle$  : messaggio che permette all'attore interno della replica di salvare il nuovo snapshot in modo tale da poter riconciliare in seguito il suo stato.
- $\langle \text{VIEW-CHANGE-COMMIT}, \text{ViewChange}, \text{MyProof}, \rangle$  : messaggio che permette di incapsulare il messaggio VIEW-CHANGE e le proof che fanno riferimento agli I-HATE-THE-PRIMARY.

- Messaggi di DatabaseActor:

- \* Messaggi di PersonaActor:

- $\langle \text{DELETE-PERSONA}, id \rangle$  : questo messaggio permette poter cancellare una persona esistente nel sistema. L'id è il parametro che rappresenta l'identificatore di una persona nel sistema.
- $\langle \text{INSERT-PERSONA}, persona \rangle$  : questo messaggio permette poter inserire una persona nel sistema. Il parametro persona rappresenta la persona da inserire nel sistema.
- $\langle \text{READALL-PERSONA} \rangle$  : questo messaggio che permette di poter ottenere tutte le persone esistenti nel sistema.
- $\langle \text{READ-PERSONA}, id \rangle$  : questo messaggio permette di poter tornare una persona esistente nel sistema. L'id è il parametro che rappresenta l'identificatore di una persona nel sistema.
- $\langle \text{SET-BYZANTINE}, id \rangle$  : questo messaggio permette di poter rendere una replica nel sistema *faulty*. L'id è il parametro che rappresenta l'identificatore di una replica nel sistema.
- $\langle \text{UPDATE-PERSONA}, persona \rangle$  : questo messaggio permette di poter aggiornare l'informazione di una persona esistente nel sistema. La persona è il parametro che contiene tutte le informazioni che si aggiorneranno di una specifica persona.
- $\langle \text{DELETE-PERSONA-RESPONSE}, persone \rangle$  : questo messaggio è la risposta al messaggio DELETE-PERSONA, che torna indietro le *persone* rimanenti nel sistema.
- $\langle \text{INSERT-PERSONA-RESPONSE}, persone \rangle$  : questo messaggio e la risposta al messaggio INSERT-PERSONA, che torna indietro le *persone* nel sistema.
- $\langle \text{READALL-PERSONA-RESPONSE}, persone \rangle$  : questo messaggio è la risposta al messaggio READALL-PERSONA, che torna indietro tutte le *persone* nel sistema.
- $\langle \text{READ-PERSONA-RESPONSE}, persona \rangle$  : questo messaggio è la risposta al messaggio READ-PERSONA, che torna indietro una persona basandosi sul id della richiesta.
- $\langle \text{SET-BYZANTINE-RESPONSE}, byzantine \rangle$  : questo messaggio è la risposta al messaggio SET-BYZANTINE, che indica se una replica è diventata bizantina.
- $\langle \text{UPDATE-PERSONA-RESPONSE}, persone \rangle$  : questo messaggio è la risposta al messaggio UPDATE-PERSONA, che torna indietro gli aggiornamenti effettuati nel sistema.

Messaggi de Zyzzyva Manager:

- $\langle \text{CLIENT-AND-KEY}, actor, key \rangle$  : messaggio che contiene sia i client esistenti nel sistema che la loro chiave pubblica, per permettere ad una replica di poter rispondere ed accettare richieste dal client.
- $\langle \text{CLIENT-INIT-MESSAGE}, key \rangle$  messaggio inviato dall'attore ClientActor per comunicare al Zyzzyva Manager il suo riferimento. *key* rappresenta la chiave pubblica del ClientActor che invia il messaggio.
- $\langle \text{CLIENT-LIST-MESSAGE}, clients \rangle$  : messaggio che contiene tutti client esistenti nel sistema, messaggio utilizzato per permettere ad una replica nuova che sta arrivando, di essere aggiornata con tutti i client già esistenti.
- $\langle \text{CLUSTER-OK-CLIENT} \rangle$  : messaggio che permette di poter indicare ad un client che tutte le repliche sono arrivate e permette al client di proseguire per poter inizializzare lo scambio di messaggi.
- $\langle \text{CLUSTER-READY} \rangle$ , messaggio utilizzato per notificare che tutti i nodi necessari al protocollo sono stati inizializzati.
- $\langle \text{FINAL-NEW-VIEW}, view \rangle$  : messaggio inviato dalle repliche quando queste finiscono un view change, permette di poter informare ad una replica nuova che stia arrivando, quale la attuale view su cui sono.
- $\langle \text{GET-ANOTHER-REPLICAS} \rangle$  : messaggio che permette ad una replica dopo aver ricevuto il messaggio CLIENT-LIST-MESSAGE poter chiedere un identificatore all'amministratore del sistema.
- $\langle \text{POSTMORTEM}, address \rangle$  : messaggio inviato quando una replica sta morendo, permette di poter indicare alle altre repliche e client che una replica è morta e comunque che non bisogna mandare messaggi.
- $\langle \text{REPLICA-ADD}, actorRef \rangle$  : messaggio che viene inviato quando una replica sta iniziando dopo che il sistema è già partito, permette di poter aggiornare le altre repliche indicando che c'è una replica nuova e in più al client, così può comunicare ed inviargli messaggi anche a lui.
- $\text{REPLICA-DEAD} \langle \text{REPLICA-DEAD}, actorRef \rangle$  : messaggio che viene inviato quando una replica sta morendo dopo che il sistema è già partito, permette poter di aggiornare le altre repliche e client, indicando che c'è una replica a cui non mandare dei messaggi.
- $\langle \text{REPLICA-INIT-MESSAGE}, key \rangle$  equivalente del CLIENT-INIT-MESSAGE per l'attore ReplicaManager.
- $\langle \text{REPLICA-NUMBER-MESSAGE}, id, view \rangle$ , messaggio utilizzato dall'attore ReplicaManager per settare il suo stato iniziale. *id* rappresenta l'id che la replica assumerà

all'interno del protocollo Zyzyva e *view* rappresenta la *view* in cui la replica inizierà la computazione.

- $\langle \text{REPLICAS-LIST-MESSAGE}, replicas, maxFailures \rangle$  : messaggio inviato ai client contenendo tutte le repliche esistenti quando il sistema sta partendo, in più permette comunicare ai client quante sono le *maxFailures*.
- $\langle \text{NODE-ARRIVED} \rangle$  : messaggio inviato a Zyzyva Manager che indica che un nodo è appena arrivato al cluster.

Messaggi del *Client* sono:

- $\langle \text{PERSONA-CLIENT}, actor \rangle$  : messaggio inviato da ZyzyvRPC a Zyzyva che fa riferimento alla richiesta di un nuovo client per poter realizzare delle richieste.
- $\langle \text{REPLICA-RESPONSE}, replicaResponses, response \rangle$  : messaggio che contiene la risposta finale fatta per una richiesta e in più contiene le risposte di ogni replica che permette poter vedere il suo stato interno.
- $\langle \text{CREATE-CLIENT-RESPONSE}, actor \rangle$  : messaggio inviato in risposta alla richiesta fatta da Zyzyva gRPC per poter ottenere un nuovo client.
- $\langle \text{REQUEST-TIMER-END-1} \rangle$  : messaggio che viene inviato una volta che il client invia una richiesta al primary e setta un timer per quella richiesta, se quel timer scade allora questo messaggio viene inviato.
- $\langle \text{REQUEST-TIMER-END-2} \rangle$  : messaggio che viene inviato una volta che il client invia una richiesta al primary e setta un secondo timer per quella richiesta, se quel timer scade allora questo messaggio è inviato.
- $\langle \text{LOCAL-COMMIT-TIMER-END} \rangle$  : messaggio che viene inviato quando scade il timer settato all'invio di un commit alle repliche quando una richiesta ha  $2 * maxFailures + 1$  messaggi uguali.

Messaggi gRPC:

- $\langle \text{WRITE-REQUEST}, key \rangle$  : in questo tipo di messaggio vengono sottintese le seguenti operazioni *Insert*, *Update*, *Delete* che fanno riferimento alle operazioni che possono essere eseguite sulla informazione di una persona.
- $\langle \text{READ-REQUEST}, key \rangle$  : in questo tipo di messaggio vengono sottintese le seguenti operazioni *Read*, *ReadAll* che fanno riferimento alle operazioni che permettono ottenere dati da una o multiple persone.
- $\langle \text{SET-BYZANTINE-REQUEST}, key \rangle$  : questo tipo di messaggio permette di poter rendere una replica in Zyzyva difettosa.
- $\langle \text{WRITE-RESPONSE}, key \rangle$  : questo messaggio contiene le possibili risposte alle operazioni WRITE-REQUEST.



- $\langle \text{READ-RESPONSES}, key \rangle$  : questo messaggio contiene le possibili risposte alle operazioni READ-REQUEST.
- $\langle \text{SET-BYZANTINE-RESPONSE}, key \rangle$  : messaggio che contiene la risposta alla operazione SET-BYZANTINE-REQUEST.

Tabelle del database

- $\langle \text{PERSONA}, id, eta, nome, cognome, haMacchina \rangle$  : fa riferimento alla struttura contenuta nel database.

### 8.3 Generazione della Documentazione

La documentazione viene generata ogni volta che il progetto è compilato. Dato che i file XML generati non sono *user-friendly*, abbiamo utilizzato il *tool* DocFX [4], che genera un sito html statico in cui la documentazione viene presentata in maniera leggibile. Per generare dunque questo sito statico è necessario installare il *tool* DocFX [4], spostarsi nella cartella **Documentation** all'interno del progetto ed eseguire i seguenti comandi:

- `docfx build`
- `docfx -serve`

fatto ciò il *tool* genererà il sito statico in locale, a cui si può accedere seguendo il link fornito sulla *command line* (di default localhost:8080).