Realizzare servizi Byzantine Fault-Tolerant e Fault-Scalable: protocollo Query/Update (Q/U)

Final report

Daniele Giulianini
daniele.giulianini3@studio.unibo.it

September 26, 2022

In virtù della sua importanza nell'ambito dei sistemi distribuiti, le soluzioni prodotte dalla comunità scientifica al cosiddetto problema del *consenso*, soprattutto nel contesto dei protocolli *Bizantine Fault-Tolerant* (BFT), sono numerose e variegate. Con l'avvento della tecnologia delle *blockchain* e il vasto riscontro che la sta caratterizzando, l'interesse verso questi protocolli si è ulteriormente rinnovato. Sebbene molteplici, non esiste attualmente una soluzione unanimemente individuata come migliore in ogni contesto, quanto piuttosto un insieme di approcci che meglio si adattano ognuno ad esigenze specifiche. Q/U rappresenta una di queste soluzioni, ossia un protocollo client/server per la realizzazione di servizi BFT distribuiti su una rete asincrona attraverso un'interfaccia costituita da operazioni di query e di update, ciascuna delle quali necessita dell'approvazione da parte di un quorum di nodi prima di essere accettata ed eseguita. Individuato dagli autori come alternativa efficiente a protocolli ben più affermati, Q/U si focalizza sulla *fault-scalability*, ossia la capacità di tollerare un numero crescente di fallimenti senza un significativo decremento delle prestazioni. Oltre che studiare ed approfondire il problema del consenso e gli approcci per affrontarlo nel contesto dei sistemi distribuiti, obiettivo del progetto che questo report accompagna è comprendere e implementare Q/U anche mostrandone il funzionamento mediante la realizzazione di una semplice *Replicated State Machine*.

Contents

1 Contesto

1.1 Tolleranza ai guasti

La tolleranza ai guasti è la proprietà di un sistema che gli consente di continuare ad operare correttamente in presenza di situazioni avverse che affliggono alcuni dei suoi componenti. Con il dispiegamento dei servizi su larga scala, considerare queste eventualità in fase di progettazione è fondamentale per garantire la *dependability* di qualsiasi sistema distribuito, cioè la capacità di fornire un servizio su cui è possibile riporre affidamento in maniera giustificata, non solo per sistemi safety e mission-critical. Un sistema è *t-fault tolerant* se continua a soddisfare la sua specifica a patto che i suoi componenti (siano processi, canali di comunicazioni fra di loro o una combinazione di entrambi) che subiscono guasti siano al massimo t. Tipicamente, un guasto si manifesta come un errore o uno stato del sistema incorretto [25]. Le soluzioni per rendere resiliente ai guasti un sistema distribuito dipendono pesantemente dalla sua modellazione, la quale può avvenire sulla base di diverse prospettive, fra cui:

- Grado di sincronia del sistema, ossia le assunzioni temporali per l'invio dei messaggi, meglio descritto nel seguito;
- Tipologia di guasti che possono affliggerlo;
- Topologia della rete che connette i nodi del sistema distribuito.

La scelta della più appropriata è determinante perché impatta severamente sull'algoritmo con cui affrontare le situazioni di guasto.

1.1.1 Modelli del sistema

Come dimostrano i vari risultati di impossibilità per il problema del consenso in contesti asincroni [15] le strategie per ottenere la resilienza del sistema distribuito non dipendono esclusivamente dal modello di guasto scelto; ne è un ulteriore esempio la maggiore difficoltà, rispetto ad uno scenario sincrono, di verificare la presenza di guasto di un solo processo in assenza di assunzioni sul tempo di trasmissione dei messaggi. Pertanto, risulta utile distinguere fra:

- *Sistemi sincroni*: dove si può assumere un limite superiore sulla trasmissione e sulla durata delle azioni dei processi coinvolti; la costruzione di sistemi sincroni pone requisiti molto spesso insormontabili, come l'analisi della completa infrastruttura del sistema, delle sue limitazioni hardware o software [31].

- *Sistemi asincroni*: per i quali non è prevista nessuna assunzione temporale su processi e canali di comunicazione.

- *Sistemi parzialmente sincroni*: progettati e costruiti in grado di operare in rispetto di requisiti temporali predefiniti e per questo impiegati in varie soluzioni BFT [20, 18], sono modellabili mediante la proprietà di *eventual synchrony*, secondo la quale "il sistema si comporterà prima o poi come un sistema sincrono anche se non c'è certezza su quando accadrà"[31].

1.1.2 Modelli di guasto

Esistono in letteratura diversi modelli di guasto, ossia specifiche più o meno formali della "maniera in cui i componenti di un sistema possono fallire"[21]. Alcuni dei più diffusi, in ordine di severità crescente, sono i seguenti:

- *Crash*: dove un processo che esegue il comportamento atteso può terminare la sua esecuzione senza riprendere a funzionare.

- *Crash-recovery*: per il quale un processo può continuare indefinitamente a sospendersi e riprendere l'esecuzione ed è, per questo, definito instabile[5].

- *Receive omission*: dove un guasto si manifesta attraverso l'invio o la ricezione di un sottoinsieme proprio dei messaggi in invio.

- Guasto *bizantino*, o maligno, *con autenticazione*: dove un processo può esibire qualsiasi comportamento arbitrario, ma non può mentire in merito all'invio di messaggi da parte dei server che si comportano secondo specifica, grazie a meccanismi di autenticazione dei messaggi, come le firme, che li rendono verificabili.

- Guasto *bizantino*, o maligno: dove, invece, questa ultima assunzione decade e un processo può fallire esibendo un comportamento arbitrario.

Oltre ai modelli riferiti ai processi appena elencati, è anche possibile classificare quelli che coinvolgono i canali di comunicazione. Alcuni protocolli assumono, inoltre, modelli ibridi che fanno riferimento alle assunzioni di più modelli di guasto.

1.2 Tolleranza ai guasti di tipo bizantino

La tolleranza ai guasti di tipo bizantino (*Byzantine Fault Tolerance*, BFT) è, banalmente, la capacità di un sistema di tollerare guasti di tipo bizantino, secondo la definizione precedente. Il termine "bizantino" è stato coniato da Lamport che per primo lo formalizzò in riferimento al problema dei "Generali Bizantini"[24] e contribuì ad affrontarlo.

1.2.1 Protocolli BFT

In letteratura si sono distinte varie soluzioni per il problema del consenso in scenari bizantini, fra loro differenti per numerosi aspetti, fra cui il numero di repliche necessarie per tollerare un determinato numero di guasti, le assunzioni sulla sincronia del sistema, la scelta di modellare di un servizio in termini di *Replicated State Machine* (RSM, descritta nel seguito). Fra i primi e più diffusi, nei cosiddetti protocolli *agreement-based* [22, 7, 11], ogni replica processa ogni richiesta eseguendo un broadcast verso le altre repliche, a differenza dell'approccio *quorum-based*, comune a [26, 28], dove invece solo un quorum di repliche processa le richieste limitando la comunicazione server-server ad occasioni limitate. Protocolli successivi, come Zyzzyva[20], combinano le due strategie. Le ultime tendenze volgono verso la modellazione di un servizio non necessariamente deterministico [9], ambito estensivamente studiato ma con modelli di guasti benigni, o all'ulteriore rilassamento di assunzioni temporali [30]. Con l'avvento delle *blockchain*, un rinnovato interesse ha spinto la ricerca verso questi protocolli; tuttavia, l'idea di progettare un protocollo BFT *one-size-fits-all* resta tuttora molto difficile se non impossibile da realizzare in pratica[34]. Siccome ognuno meglio si adatta ad esigenze e caratteristiche specifiche, la scelta del più appropriato va quindi valutata di caso in caso.

1.3 Consenso

Estremamente importante nel contesto dei sistemi distribuiti, il problema del *consenso* consiste nel consentire a due o più entità di convenire su una decisione a partire dai loro valori iniziali. Estensivamente studiato perché necessario per la realizzazione di applicazioni distribuite sicure ed affidabili, la letteratura ne ha prodotto numerose soluzioni nella forma di diversi protocolli per differenti modelli di sistema e di guasto. Più formalmente, il consenso è contraddistinto dalle seguenti proprietà:

- *terminazione*: ogni processo arriverà a *decidere* un valore.
- *integrità*: nessun processo può *decidere* più di un valore.
- *accordo*: due processi corretti non possono *decidere* un valore v diverso.

Il problema del consenso è in realtà declinabile in molte varianti[8], con diverse proprietà specifiche.

Impossibilità in contesto asincrono Proprio l'importante studio che lo ha riguardato ha consentito di identificare per il consenso importanti risultati in merito alla *risolvibilità*: è impossibile, infatti, soddisfare le proprietà elencate in reti completamente asincrone in presenza anche di un singolo guasto [15]. Pertanto, gli approcci presenti in letteratura si sono soffermati sul rilassamento di parte delle assunzioni riguardanti, in particolare, il modello asincrono.

1.4 State Machine Replication

State machine replication (SMR) è un approccio per la costruzione di servizi resilienti ai guasti attraverso la replicazione dei server e il coordinamento delle interazioni dei client [33]. Originalmente introdotto da Lamport [23] e in seguito adottato dalla maggior parte degli approcci BFT presenti in letteratura, si fonda sull'assunzione che un servizio possa essere modellato in termini di una *macchina a stati*.

Dopo essere stato inizializzato allo stato iniziale S0, ogni server riceve una serie di comandi c1, c2, ..., attuando, ogni volta, una transizione di stato deterministica che dipende dallo stato precedente e dal comando eseguito. Così facendo, se due o più repliche saranno avviate con il medesimo stato iniziale, eseguendo la stessa serie di comandi giungeranno allo stesso stato finale. A tale scopo, è pertanto necessario che le repliche raggiungano il consenso sulla sequenza ordinata di comandi: tale problema, è infatti un'istanza del problema del consenso chiamata *Atomic Broadcast*.

Dopo aver inviato i comandi e ricevuto le risposte corrispondenti, i client applicano una funzione all'insieme di risposte allo scopo di mascherare i possibili guasti occorsi alle repliche. Il numero di repliche necessarie dipende dal modello di guasto considerato:

- se il modello di guasto assume solo guasti di tipo *crash* possono essere sufficienti, in base agli approcci specifici, $f + 1$ repliche,
- se ammette anche guasti bizantini, allora il limite minimo di repliche necessarie è $2f + 1$ così da consentire di individuarvi la maggioranza di risposte corrette [33].

1.5 Fault-scalability

Terminologicamente meno popolare rispetto alla *tolleranza ai guasti*, la *fault scalability* è definita dagli autori di Q/U come la capacità di garantire buone prestazioni, con un degradamento graduale, e non repentino, nell'erogazione di un servizio all'aumentare dei guasti dei server che lo erogano. Gli autori propongono di valutarla raffrontando il *throughput* generato dai server (richieste al secondo processate) con il numero di guasti tollerati dai server stessi.

Come mostrato in fig. 1, protocolli *agreement-based* e *quorum-based* esibiscono differenti caratteristiche di *fault-scalability*: per i primi, nonostante ne richiedano in numero minore rispetto alla controparte, l'aumento delle repliche determina un calo generale dello *throughput*, il quale rimane, invece, costante nel caso dei *quorum-based*.

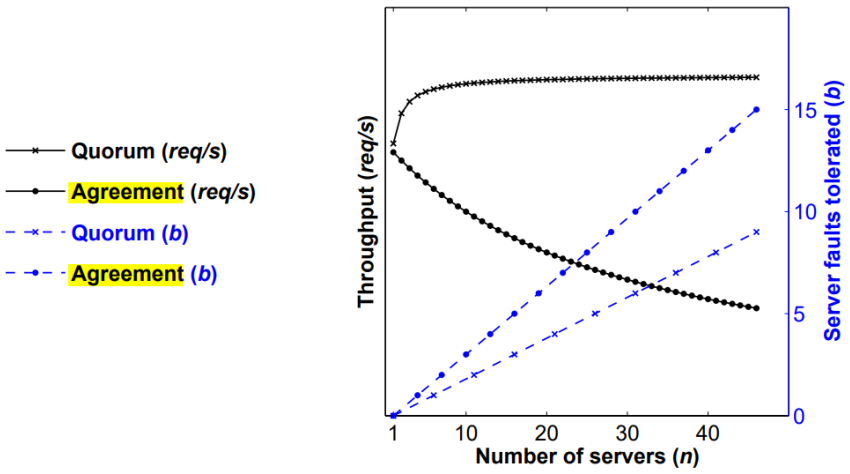


Figure 1: Raffronto delle caratteristiche di *fault-scalability* di protocolli *quorum-based* e *agreement-based*. Sebbene i primi richiedano più repliche all'aumentare dei guasti tollerati (linee blu tratteggiate), il loro *throughput* si mantiene, in teoria, elevato, a differenza dei secondi. Tratto da [4].

2 Protocollo Query/Update

Q/U è un protocollo BFT client/server basato su quorum per la realizzazione di servizi mediante un'interfaccia basata su operazioni. Storicamente succeduto a BFT e PBFT[10], si è distinto rispetto a tali protocolli basati su accordo per il suo focus sulla *fault-scalability* al prezzo di un maggiore numero di repliche ($5b + 1$ per tollerare b guasti bizantini al posto di $3b + 1$) necessarie per il suo funzionamento.

2.1 Modello

Il protocollo si basa su un modello temporale asincrono che non impone assunzioni sul tempo di trasmissione dei messaggi (comunque limitato e non nullo), i quali sono veicolati da canali autenticati punto a punto client-server, inaffidabili, ma rendibili tali attraverso ritrasmissione. Si assume, inoltre, che client e server siano limitati computazionalmente così da non minare l'efficacia delle primitive crittografiche. I client possono esibire un comportamento bizantino, mentre il modello dei guasti adottato per il comportamento dei server combina quello *crash recovery* con quello bizantino, allo stesso modo di [5]. Il *fail-prone system* risultante è indicato come segue: si assume un universo U di server tale che $|U| = n$, entro il quale si distinguono due sottoinsiemi $TS \subseteq 2^U$ e $BS \subseteq 2^U$; ogni server con guasto appartiene a qualche $T \in$TS, ogni server maligno appartiene a qualche $B \in$BS; per cui $B \subseteq T$. Un *quorum system* $QS \subseteq 2^U$ è l'insieme, non vuoto, dei sottoinsiemi di U che si intersecano. Ogni $Q \in QS$ è un *quorum*.

2.2 Proprietà e garanzie

Serializability e progresso il protocollo, nella versione di base con aggiornamento single-object, garantisce aggiornamenti e interrogazioni linearizzabili, e, in un'esecuzione benigna (cioè senza guasti bizantini o di tipo *crash*), la loro esecuzione è anche garantita essere priva di ostruzione. In caso di esecuzioni maligne, la garanzia di progresso decade, ma è possibile integrare meccanismi di mitigazione efficaci.

Fault scalability Come già ricordato, il vantaggio più evidente che Q/U mostra rispetto ai protocolli BFT basati su *agreement* derivante dalla sua natura *quorum-based* è la *fault-scalability*: per questi ultimi, il processamento di una richiesta richiede un broadcast fra le repliche e per questo l'aumento dei guasti tollerati (e il conseguente aumento delle repliche necessarie) si traduce in un *throughput* inferiore dovuto all'overhead di comunicazione.

Efficienza e concurrency atomicity Un altro importante vantaggio di Q/U rispetto ai precursori è il suo approccio ottimistico, ossia l'adozione di meccanismi che, in occasione di periodi senza contesa fra le richieste dei client (quando le repliche non ricevono, cioè, nuove operazioni prima che l' operazione di aggiornamento corrente sia completata) e privi di guasti, permettono di completare interrogazioni ed aggiornamenti in un singolo passo di comunicazione client-server. Questi due scenari, soprattutto il primo, rappresentano i più favorevoli per Q/U[34], rendendolo molto più efficiente degli approcci pessimistici che impiegano, ad esempio, soluzioni centralizzate o lock distribuiti; d'altro canto, il protocollo non è efficiente in situazioni di contesa, che possono, infatti, causare il fallimento delle operazioni di aggiornamento e che rappresentano per questo il più serio suo punto debole [34].

Throughput-scalability Un altro beneficio che la letteratura adduce all'approccio *quorum-based* è la *throughput scalability*, ossia la possibilità di incrementare il *throughput* dei server all'aumentare del numero di repliche oltre alla soglia strettamente necessaria per garantire la fault scalability di fronte allo scenario di fallimento di interesse. L'aumento delle repliche senza specifiche contromisure può, tuttavia, indurre un degrado delle prestazioni definito *"scaleup pitfall"* che gli autori confermano ridurre considerevolmente integrando al funzionamento di base una modifica del protocollo definita *multi-object update*.

Le proprietà sono descritte con più dettaglio nel report che accompagna l'articolo principale [3].

2.3 Funzionamento di base

Il protocollo prevede l'interazione fra più istanze di due categorie di entità:

- client: responsabile della sottomissione delle richieste alle repliche e della restituzione del loro risultato all'utente finale. Rispetto a protocolli *agreement-based*,

ha, inoltre, un fondamentale ruolo nella diffusione della conoscenza fra le repliche, rimpiazzando la comunicazione replica-replica, che in un protocollo *quorum-based* come Q/U è limitata a casi eccezionali.

- server o repliche: mantengono le versioni degli oggetti e processano le richieste dei client.

Mentre il numero dei server necessari per garantire la tolleranza ai guasti è dettato dalle soglie di classificazione definite più avanti, non si pongono vincoli a quello dei client.

Modello dati Q/U espone un'interfaccia basata su operazioni deterministiche in una maniera molto simile all'approccio SMR e che per questo ne facilita l'impiego. Tali operazioni riguardano oggetti replicati su ognuno dei server del sistema. Le operazioni si suddividono in:

- interrogazioni (query): non modificano l'oggetto su cui vengono eseguite.
- aggiornamenti (update): il loro processamento con successo determina una nuova versione dell'oggetto memorizzata dal server.

Ogni operazione viene effettuata attraverso la sottomissione ad opera del client di una richiesta che, se ricevuta e accettata dalla replica, viene invocata sulla versione locale dell'oggetto: in caso di aggiornamento, ne viene quindi generata una nuova che sostituisce la precedente come versione locale. Quest'ultima viene comunque conservata assieme al suo *timestamp logico* all'interno di una struttura chiamata *Replica History* (RH) che rappresenta il suo storico delle versioni, ritornato aggiornato al client a seguito di ogni richiesta. Le RH dei server vengono accumulate dai client dando origine all'*Object History Set* (OHS); dal momento che, viste le assunzioni di asincronia, solo un sottoinsieme delle repliche risponderà, questo "costituisce la visione parziale dei client dello stato globale del sistema"[4].

Prima di essere aggiunto alle RH che compongono l'OHS, il logical timestamp (LT) riferito all'operazione di aggiornamento appena eseguita viene associato a quello relativo all'aggiornamento subito precedente, per questo chiamato *conditioned-on* (ltCo); la RH è, pertanto, una collezione di coppie di LT ciascuna chiamata *candidato* che, delineandone la "catena di condizionamento", stabilisce l'ordinamento totale fra le operazioni. Tale ordinamento è l'oggetto del *consenso* fra le repliche e consente di fornire la garanzia di *strict serializabilty*.

2.3.1 Classificazione

La classificazione è la procedura che determina se l'ultimo candidato di un OHS, è *completo*, *riparabile* o *incompleto*. Dal punto di vista pratico, la sua appartenenza a una di queste categorie è un punto cruciale del protocollo perché determina se eseguire direttamente un'operazione o avviare la procedura di *repair* (descritta alla sezione 2.3.3). La definizione di *repairable set* deriva da quella di quorum: considerato QS l'insieme di tutti i quorum Q, ogni Q definisce un insieme di insiemi repairable $R(Q) \subseteq 2^Q$.

Considerato un insieme di risposte S, le regole di classificazione definiscono i seguenti concetti:

$$classify(S) = \begin{cases} \text{complete}, & \text{if } \exists Q \in QS : Q \subseteq S \\ \text{repairable}, & \text{if } (\forall Q \in QU : Q \subseteq S) \wedge (\exists Q \in QU, R \in R(Q) : R \in S) \\ \text{incomplete}, & \text{otherwise.} \end{cases}$$

L'integrazione di tali regole con una serie di assunzioni matematiche garantisce il corretto funzionamento del protocollo. Q/U non vincola ad uno specifico meccanismo di scelta del quorum; quello impiegato dagli autori è un meccanismo a soglia ricorsiva [27], secondo il quale, definiti un *quorum system* in cui tutti i quorum $Q \in QS$ hanno cardinalità q, tutti i repairable set $R \in R(Q)$ hanno cardinalità r e tutti gli insiemi di repliche che possono esibire un guasto $T \in TS$ sono composti da t elementi, si ha:

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

Da cui derivano le seguenti regole di classificazione:

$$classify(Order) = \begin{cases} \text{complete}, & \text{if } q \leq Order \\ \text{repairable}, & \text{if } r \leq Order < q \\ \text{incomplete}, & \text{otherwise.} \end{cases}$$

che permettono di ricondurre l'operazione da eseguire alla ricezione dell'OHS all'ordine del candidato, cioè al numero di server che lo riportano nella propria risposta.

Schema di Logical Timestamping Q/U adotta uno schema di marcatura temporale dove i timestamp sono una funzione deterministica del conditioned-on OHS e dell'istanza di operazione da eseguire (anche i valori dei suoi parametri contribuiscono all'attribuzione), per questo monotona crescente. Vengono costruiti dalle repliche in risposta ad operazioni di aggiornamento e in questo modo risulta impossibile per un client bizantino sottomettere aggiornamenti differenti con lo stesso timestamp.

Controllo dell'integrità HMAC Per la verifica dell'integrità di fronte a manipolazioni bizantine, Q/U sfrutta un autenticatore che ogni replica include accanto alla propria RH poi contenuta nel conditioned-on OHS inviato dai client nelle richieste. Nello specifico, un autenticatore è una lista di n HMAC[6], uno per ogni replica. Grazie alle proprietà delle funzioni *hash*, il loro impiego rende impossibile la corruzione inosservata delle Replica History. Per risparmiare spazio, gli autenticatori sono generati a partire dall'hash della RH. La scelta di HMAC impatta sulle caratteristiche di *fault-scalability*, dal momento che la loro elaborazione ha un costo proporzionale alla lunghezza dell'input: la valutazione di altre tecniche, come, ad esempio, la *firma digitale*, il cui costo computazionale e spaziale non aumenta con n, dipende dalla scala del sistema e va considerato per grandi dimensioni.

2.3.2 Scenario ottimistico

Client Per l'esecuzione di un'operazione, il client costruisce una richiesta contenente l'OHS risultante dagli scambi precedenti con le repliche, la operazione stessa e i suoi argomenti. Tale OHS è per questo detto conditioned-on OHS e il suo invio rappresenta la propagazione dell'informazione sullo stato globale a ciascuna delle repliche, che non comunicano direttamente fra loro a questo scopo. Una volta inviata, il client rimane in attesa della risposta del server. Le comunicazioni possono non coinvolgere la totalità delle repliche e Q/U è, in tal senso, compatibile con diverse strategie di scelta del quorum. In caso di assenza di contesa e di guasti, l'esecuzione può completare a fronte di un singolo scambio di comunicazione client-server. Tale scenario, definito *ottimistico*, è possibile grazie a due meccanismi assunti dal client:

- caching dell'OHS: a seguito di ogni risposta ricevuta dal quorum, il client mantiene una copia dell'OHS risultante, così da non doverlo richiedere prima dell'esecuzione di aggiornamenti.
- incentivazione della località attraverso l'assegnamento di un quorum preferito per ciascun oggetto, da contattare per l'esecuzione delle operazioni che lo riguardano così evitando di coinvolgere repliche che potenzialmente non dispongono della sua versione più aggiornata.

I requisiti per l'esecuzione ottimistica sono:

- l'assenza di contesa, riscontrabile dalla OHS, che quindi sarà aggiornata, attraverso la procedura di classificazione,
- assenza di fallimenti, che consente al client di comunicare con successo esclusivamente con le repliche del quorum preferito dell'oggetto coinvolto.

Server All'avvio di ogni replica, la rispettiva RH viene inizializzata con un candidato, detto iniziale, che costituisce l'inizio della cosiddetta "catena dei condizionamenti". Alla ricezione di una richiesta, il server procede alla validazione dell'integrità delle RH contenute nell'OHS inviato attraverso il controllo degli autenticatori. La presenza di incongruenze non compromette il processamento della richiesta, ma determina la sostituzione della RH coinvolta con una RH nulla, cioè costituita dal solo candidato iniziale. Segue la classificazione, che valuta se l'OHS è aggiornato: solo in tal caso l'operazione viene invocata sull'oggetto corrispondente all'ultimo candidato che contiene. Se l'operazione è un aggiornamento viene quindi prodotta una nuova versione dell'oggetto che aggiorna la RH della replica, la quale viene infine restituita al client insieme al rispettivo autenticatore, ad un messaggio di successo e alla risposta ottenuta dall'invocazione.

2.3.3 Scenario di contesa (e repair)

Le situazioni di contesa si verificano in occasione di aggiornamenti concorrenti allo stesso oggetto. Tali richieste, poiché fra loro differenti, inducono la generazione di timestamp

differenti che, a sua volta, causa la divergenza delle RH. Se il client non raggiunge un numero sufficiente di risposte concordi, per preservare le proprietà del protocollo, l'aggiornamento deve fallire e riportare le repliche ad uno stato consistente: ciò avviene attraverso la procedura di *repair* (riparazione). Punto fondamentale del protocollo, il repair consiste nell'esecuzione, possibilmente ripetuta, di una sequenza di operazioni di:

- *barrier*: accettate sempre con successo dalle repliche, prevengono il completamento delle operazioni precedentemente sottomesse, così sopprimendole, quando ricevute da un numero sufficiente di server.
- e *copy*: attuate a seguito della soppressione, rilevata tramite OHS, delle operazioni concorrenti, ripristinano la versione dell'oggetto precedente alla contesa, che può essere quella risultante da una delle operazioni concorrenti oppure quella ancora precedente.

La necessità di riparare è individuata dalla procedura di classificazione e non è limitata, nella versione non ottimizzata di Q/U, alla situazione di contesa. In tale scenario, tutti i client autori delle richieste concorrenti incorreranno in repair ed è perciò necessario adottino meccanismi di **backoff** (gli autori hanno implementato una policy di backoff *random exponential*, ma è possibile valutare altri meccanismi) per evitare *livelock*.

2.3.4 Scenario di mancata disponibilità dell'oggetto

Dal momento che, per eseguire un'operazione, ad un client è sufficiente contattare un quorum, cioè un sottoinsieme della totalità di repliche, è possibile che qualcuna di esse si trovi a processare una richiesta senza disporre della versione aggiornata dell'oggetto sul quale l'operazione deve essere invocata. In questo caso, la replica in questione necessita di eseguire *object-sync*, una procedura grazie alla quale provvede a recuperarla contattando i server che, invece, ne dispongono (detti *host-server*), individuabili a partire dal conditioned-on OHS contenuto nella richiesta. Anche in questo caso, varie strategie per la scelta e l'interazione con le repliche da coinvolgere sono possibili.

2.3.5 Scenario in caso di guasti

La presenza di guasti alle repliche può determinare l'esigenza da parte del client di effettuare *probing*, ossia rivedere a tempo di esecuzione la composizione del quorum precedentemente scelto per la sottomissione di un'operazione, e a procedere alla ritrasmissione della richiesta, che verrà, quindi, processata dai nuovi server nella maniera descritta in precedenza, come mostra la fig. 2 (b), dove si assiste ad un unico fallimento. Anche in questo caso, Q/U è compatibile con diversi meccanismi, più o meno avanzati, di *probing* e ritrasmissione.

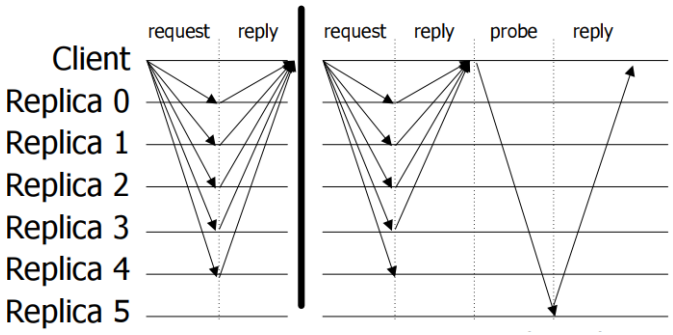


Figure 2: A sinistra (a): sottomissione di un'operazione da parte di un client alle repliche senza guasti e contesa; a destra (b): gestione del guasto della replica 4 con conseguente *probing*. Tratto da [34].

2.4 Ottimizzazioni

Gli autori hanno individuato una serie cospicua di ottimizzazioni al funzionamento di base di Q/U già descritto, che, nella maggior parte dei casi, non ne alterano le proprietà, ma contribuiscono in modo significativo alla sua efficienza.

Inline repair Sebbene fondamentale, la procedura di repair è molto onerosa e, richiedendo una sequenza di operazioni di barrier e copy, rappresenta il punto debole di Q/U in termini di efficienza. È, tuttavia, possibile per una replica, esclusivamente quando:

- l'OHS ricevuto non evidenzia contesa nelle operazioni sottomesse,
- l'ultimo candidato soddisfa la soglia di riparazione,

riparare ottimisticamente (in un singolo passo di comunicazione) la propria RH con l'ultimo candidato dell'OHS, anche di tipo barrier. Tale scenario è comune nel caso di fallimento delle repliche, poiché porta i client a contattarne di nuove, non aggiornate, che risponderanno con una RH differente dalle altre. Tale meccanismo richiede l'integrazione di un controllo lato server e la rivisitazione della procedura di classificazione, che deve assecondare la presenza di due nuovi tipi di operazione (*inline barrier* e *inline method*).

Timestamp compatto Per ridurre lo spazio richiesto per l'allocazione lato client e server dei timestamp e quindi delle RH e dell'OHS, è possibile sostituirli all'interno di un timestamp con una loro rappresentazione quale, ad es., un *collision resistant hash*[17], senza nessuna perdita di informazione di rilievo per il protocollo.

Esecuzione di query ottimistica Qualora:

- il client intenda eseguire un'interrogazione
- l'OHS memorizzato dal client ed inviato alle repliche non fosse aggiornato per via del completamento di aggiornamenti sottomessi nel frattempo da altri client,

le repliche e il client possono includere un meccanismo per completare la richiesta ancora in un singolo passo di comunicazione, contribuendo ulteriormente all'approccio ottimistico di Q/U già descritto. Oltre alla RH aggiornata, nella risposta viene, infatti, incluso il risultato dell'interrogazione poi rilevata dal client attraverso una classificazione aggiuntiva che rende inutile un'ulteriore ritrasmissione.

Quorum preferito Come già ricordato, Q/U è compatibile con diverse politiche di scelta e di coinvolgimento del quorum di repliche contattate dai client. Per promuovere la località d'accesso e quindi l'approccio ottimistico, gli autori suggeriscono di assegnare a ciascun oggetto un quorum preferito, potenzialmente rivedibile dinamicamente, che verrà contattato per primo. Se le repliche contattate sono aggiornate, infatti, la possibilità di ricevere un candidato incompleto si riduce e con essa la necessità di ritrasmissioni. Solo in caso di assenza di risposta entro un certo limite, si procederà al *probing* di altri server, che verosimilmente necessiteranno di eseguire *object-sync*.

Gestione richieste ripetute A causa delle ritrasmissioni necessarie per assecondare il modello di guasto ibrido e alla modifica di inline repair, l'invio alle repliche di richieste ripetute provenienti dallo stesso client, o da client diversi, è una circostanza potenzialmente frequente. Per preservare la semantica del protocollo, in queste situazioni, le repliche devono:

- invocare una sola volta l'operazione sull'oggetto e
- fornire la stessa risposta alla stessa richiesta;

per cui si rende necessaria una modifica esclusivamente lato server per il salvataggio delle stesse ed un controllo per la loro futura ritrasmissione preliminare alla restante logica di processamento.

Componenti bizantine La presenza di componenti bizantine, sia lato client che server, può non garantire la proprietà di liveness. Allo scopo, gli autori di Q/U hanno identificato una serie di meccanismi nell'articolo di presentazione a cui si rimanda per un maggior dettaglio, in grado di garantire il progresso dell'esecuzione anche in tale situazione.

Pruning delle replica history Per il corretto funzionamento del protocollo è sufficiente per i server mantenere solo le porzioni di RH contenenti i timestamp successivi al conditioned-on timestamp relativo all'ultimo aggiornamento accettato dalla replica, eliminando il resto. Ciò si riflette anche sulla riduzione dello spazio necessario al salvataggio delle versioni corrispondenti, che può, tuttavia, impedire ad un server di rispondere alle richieste ripetute o di *object-sync* giunte in ritardo: pertanto, l'ultimo timestamp da considerare è in realtà oggetto di un compromesso fra efficienza e utilizzo di memoria.

Multi object update Si è già discussa la difficoltà del protocollo negli scenari di contesa, dell'esigenza di promuovere l'accesso non concorrente agli oggetti e dei rimedi introdotti per evitare lo "*scaleup pitfall*". A questo scopo, il protocollo può essere ulteriormente esteso per consentire l'aggiornamento atomico di più oggetti con una singola operazione: l'aumento della loro granularità può infatti ridurre le situazioni di contesa fra i client per il loro accesso permettendo di beneficiare più estensivamente dell'approccio ottimistico del protocollo, al prezzo per l'utente di decomporre l'oggetto iniziale nelle sue componenti. Rispetto alla versione a singolo oggetto, tale estensione introduce alcune differenze:

- la richiesta di aggiornamento dei client include l'insieme di oggetti coinvolti e i rispettivi conditioned-on OHS (multi-OHS)
- ciascuna replica richiede l'acquisizione di un lock locale per ognuno degli oggetti coinvolti dall'aggiornamento multiplo, che è atomico
- la procedura di riparazione di un candidato repairable è più complessa: viene dapprima valutato un approccio chiamato *classification by deduction* che tenta di inferire la completezza o meno del candidato in questione sulla base dei singoli OHS, aggiornati, degli oggetti coinvolti, salvo procedere con la riparazione costituita da barrier e copy in caso di mancanza di evidenze. In caso in cui il candidato sia, invece, completo o incompleto la procedura rimane invariata.

In ogni caso, l'ottimizzazione modifica le garanzie fornite dal protocollo sostituendo alla *linearizabiliy* la *strict serializability.*

3 Obbiettivi

L'obbiettivo del progetto è lo studio e l'implementazione del protocollo BFT Query/Update (Q/U), il cui articolo redatto dagli autori in [4] non è accompagnato da un'implementazione attualmente disponibile in rete. Sarà inoltre l'occasione per studiare ed approfondire il problema del consenso e gli approcci per affrontarlo nel contesto dei sistemi distribuiti.

Seppur particolarmente risaltate dagli autori, esulano dagli obbiettivi di progetto l'analisi delle prestazioni di Q/U nelle sue diverse configurazioni e/o il raffronto con quelle di altri protocolli BFT noti in letteratura.

Nello specifico, si intende realizzare:

- una libreria che esponga le funzionalità del protocollo rendendole utilizzabili in diversi contesti, nel modo più *trasparente* possibile e con il maggior controllo sui tipi di dato a tempo di compilazione (*type safety*) per l'utente programmatore. Si ritiene, infatti, fra gli obbiettivi, quello di predisporre una libreria che non induca l'utente in cattive pratiche di programmazione.
- uno scenario di applicazione prototipale che mostri l'attitudine del protocollo alla definizione di un servizio basato sull'approccio SMR, quindi esemplificandone il funzionamento e le potenzialità, come punto di partenza per servizi più complessi.

3.1 Scenari

3.1.1 Libreria

Scenari di utilizzo La libreria deve fornire un punto di accesso alle funzionalità di client e server del protocollo Query/Update agli sviluppatori, consentendo di definire un servizio in termini di una RSM. In particolare, sarà necessario fornire le API per:

- definizioni delle operazioni della SMR che modella il servizio;
- la configurazione e l'avvio del server di autenticazione;
- la configurazione e l'avvio delle repliche;
- l'autenticazione dei client che interagiscono con le repliche;
- la configurazione del client che si interfacciano con le repliche;
- sottomissione di un'operazione di interrogazione o di modifica dello stato dell'oggetto condiviso;
- la terminazione delle repliche, del servizio di autenticazione e del client.

Scenari di funzionamento Per quanto riguarda le situazioni che il protocollo dovrà gestire, saranno almeno le seguenti:

- gestione scenario privo di contesa fra le sottomissioni delle richieste dei client alle repliche;
- gestione scenario di contesa fra le sottomissioni delle richieste dei client alle repliche;
- gestione scenario di disallineamento dello stato fra i server con necessità di sincronizzazione;
- gestione scenario di guasto, di tipo *crash* o bizantino, di una delle repliche contestuale alla ricezione di richieste di client.

3.1.2 Applicazione demo

Come mostrato dai diagrammi di caso d'uso in fig. 3 e 4, l'applicativo dovrà consentire all'utente, in questo caso non necessariamente programmatore software, di interagire con un contatore distribuito tramite un'interfaccia a linea di comando che incapsuli e gli renda trasparente la logica di esecuzione dell'algoritmo. Previa autenticazione e registrazione del client, ottenuta da un server di autenticazione basato su token, l'interazione avverrà attraverso un insieme predefinito di comandi, distinti in operazioni di interazione con un contatore e di gestione del cluster. Le prime verranno ricondotte ad invocazioni client rivolte verso un insieme di repliche consentendo di incrementare il valore del contatore, decrementarlo, reimpostarlo al valore zero e ottenerne lo stato aggiornato. Tramite le seconde, invece, sarà possibile ispezionare lo stato delle repliche e

terminarne l'esecuzione, a patto di evitare il superamento delle soglie di tolleranza del protocollo, circostanza che sarà comunicata all'utente, così come quando l'utente intende arrestare una replica non più in esecuzione. Inoltre, anche la funzionalità di terminazione del sistema nel suo complesso dovrà essere predisposta. In questo modo, l'applicativo fornirà l'opportunità all'utente di verificare l'efficacia del protocollo di fatto realizzando una forma di validazione di alto livello dello stesso.

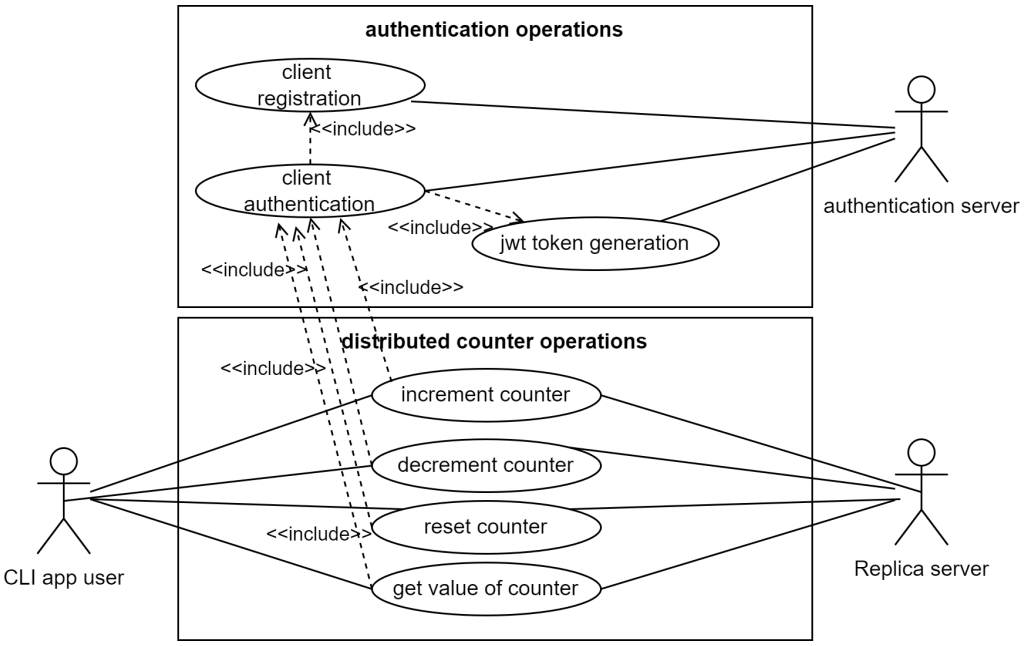


Figure 3: Diagramma UML di caso d'uso che rappresenta le funzionalità della demo riguardanti l'interazione con il contatore, quelle di autenticazione necessarie, e i componenti coinvolti.

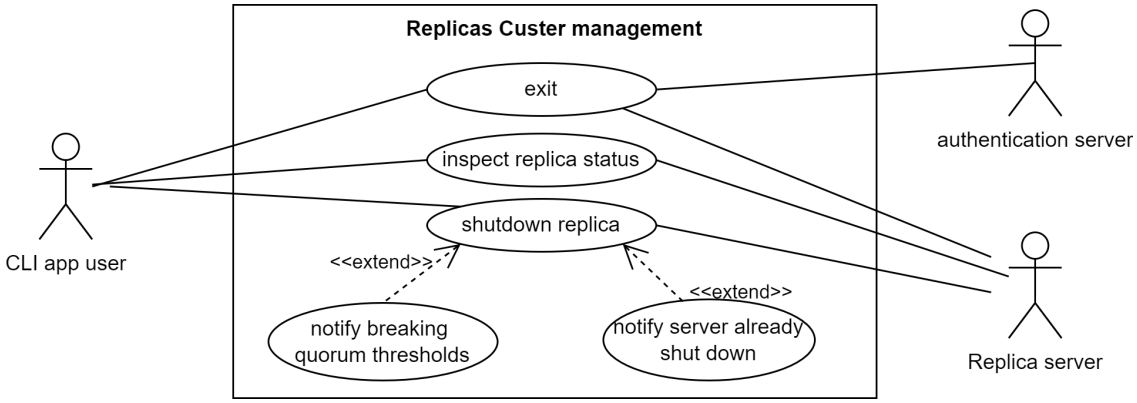


Figure 4: Diagramma UML di caso d'uso che rappresenta le funzionalità della demo riguardanti la gestione del cluster di repliche e i componenti coinvolti.

3.2 Politica di autovalutazione

La rispondenza degli artefatti ai requisiti verrà attestata attraverso test di unità e test di sistema, che, vista la complessità intrinseca nel tema di progetto, avranno un ruolo centrale nel suo sviluppo. In generale, si adotterà un approccio orientato ai test, valutando anche l'impiego di *property-based* testing, meglio descritto nella sezione 7. In secondo luogo, lo scenario applicativo sarà impiegato come ulteriore verifica del suo corretto funzionamento.

La qualità del software prodotto, riportata fra gli obbiettivi, verrà attestata mediante gli strumenti di controllo della qualità descritti alla sezione 10. Anche la rispondenza ai requisiti non funzionali sarà considerata come misura della buona riuscita in questo senso.

4 Analisi dei requisiti

4.1 Requisiti funzionali

Con riferimento alle caratteristiche del protocollo rese note dagli autori e descritte alla sezione precedente, le funzionalità da implementare sono:

4.1.1 Libreria

1. Funzionamento di base:
    a) Interrogazione e single-object update
        i. backoff policy randomica esponenziale
        ii. strategia di quorum di tipo broadcast lato client e server
        iii. object-sync lato server
    b) autenticazione client/server
    c) verifica integrità RH tramite HMAC
2. Ottimizzazioni al funzionamento di base:
    a) esecuzione ottimistica delle query
    b) gestione delle richieste ripetute lato server
    c) ottimizzazione dimensione timestamp
    d) inline repair
    e) pruning delle replica history
    f) timestamp compatto mediante hashing
    g) caching di OHS lato client

4.1.2 Demo

Relativamente allo scenario esemplificativo SMR, si prevede la possibilità per un utente di interagirvi mediante una applicazione a riga di comando, secondo un'interfaccia ben definita, con un contatore distribuito replicato su un insieme di server che mantengono il consenso sul suo stato attraverso il protocollo Q/U. Nello specifico, le funzionalità sono:

1. Sottoporre operazione di incremento del contatore
2. Sottoporre operazione di decremento del contatore
3. Sottoporre operazione di reset del contatore
4. Visualizzazione del valore del contatore
5. Terminare l'esecuzione di uno specifico server, simulando un guasto
6. Terminare l'esecuzione dell'intero cluster
7. Verificare lo stato dei server (ossia se in esecuzione o terminati)

4.2 Requisiti non funzionali

4.2.1 Libreria

Come ulteriori requisiti non funzionali che la libreria dovrà rispettare si aggiungono:

1. Compatibilità con diverse tecnologie, in vista di estensioni future, di:
   a) Serializzazione/deserializzazione
   b) Autenticazione/autorizzazione
   c) Comunicazione (middleware)
2. Predisposizione all'iniezione, in vista di estensioni future, di:
   a) Differente logica di scelta e coinvolgimento quorum lato client
   b) Differente logica di scelta e coinvolgimento server lato server
   c) Differente logica di persistenza lato server
   d) Differente threshold *quorum system*

4.2.2 Demo

Relativamente allo scenario esemplificativo:

1. Compatibilità della logica applicativa con una differente interfaccia, eventualmente grafica.

4.3 Requisiti implementativi

4.3.1 Libreria

Per la libreria non ci sono requisiti di implementazione imposti a priori, ma si valuteranno le tecnologie e gli approcci più convenienti di fronte ai requisiti descritti.

4.3.2 Demo

1. dispiegamento con docker, attraverso Dockerfile per l'automatizzazione delle procedure annesse.

5 Design

5.1 Struttura

5.1.1 Struttura del progetto

La struttura globale del progetto è mostrata in fig. 5. Nell'organizzazione del sistema distribuito si è mantenuto una granularità di modularizzazione fine suddividendolo in unità di compilazione contenute. Vista la complessità del sistema, si è agito in questo modo predisponendolo sin dall'inizio ad eventuali estensioni future in un'ottica di manutenibilità e facilità di gestione. Fra i vantaggi di questo approccio risultano il confinamento delle dipendenze tecnologiche ad un ambito più ridotto che riduce le occasioni di modifica del codice non direttamente coinvolto e un migliore incapsulamento. Il progetto si compone pertanto dei seguenti sotto-progetti:

- qu-core: è il nucleo che definisce il modello delle entità base, le strutture dati e gli algoritmi impiegati da Q/U che prescindono dalla specifica tecnologia di comunicazione, serializzazione, autenticazione o di salvataggio dei dati impiegata e per questo compatibile e riusabile con diverse combinazioni di queste ultime. È il sotto-progetto di più alto livello che non presenta dipendenza da tecnologie esterne e per questo verosimilmente quello meno soggetto a modifiche.

- qu-presentation: definisce gli aspetti di serializzazione e deserializzazione relativi allo strato di presentazione dello stack ISO/OSI per il loro trasferimento sulla rete. Sebbene definisca alcune scelte tecnologiche, la sua progettazione la rende compatibile con diversi meccanismi di serializzazione iniettabili in futuro.

- auth: è un modulo di autenticazione e autorizzazione token-based basato su tecnologia JWT riusabile in diversi contesti e non strettamente legato a questo progetto, modellando genericamente le informazioni di un utente, i suoi ruoli, le sue credenziali.

- qu-communication: contiene le astrazioni per la comunicazione e l'interazione in comune fra client e service, fra cui alcuni concetti riusabili in generale fuori da Q/U, come alcune *stub* lato client. Dipende dalla tecnologia di serializzazione, da qu-core e da qu-auth.

- qu-client: contiene il codice lato client isolandolo da quello lato service. Oltre ad ospitare le funzionalità strettamente relative al protocollo Q/U, contiene un package di più alto livello ancora in fase di prototipo costruito sopra alla semantica del protocollo stesso che modella strutture dati distribuite di alto livello, esemplificando le capacità di personalizzazione e riuso della libreria come base fondante per una logica di applicazione di più alto livello.
- qu-service: ospita la logica server del protocollo, accessibile attraverso rotte, e quella di configurazione, avvio e terminazione delle repliche, oltre a contenere la controparte lato service del modulo di strutture dati distribuite già descritto.
- qu-storage: la logica di memorizzazione delle informazioni è stata isolata in un sotto-progetto specifico per il ruolo centrale che ricopre.
- qu-demo: costituisce una semplice RSM accessibile da un'applicazione a linea di comando che esemplifica il funzionamento della libreria.
- qu-system testing: contiene il codice di test sul sistema integrato completo, che si somma ai test d'unità che accompagnano i singoli sotto-progetti per valutare il funzionamento complessivo del protocollo.

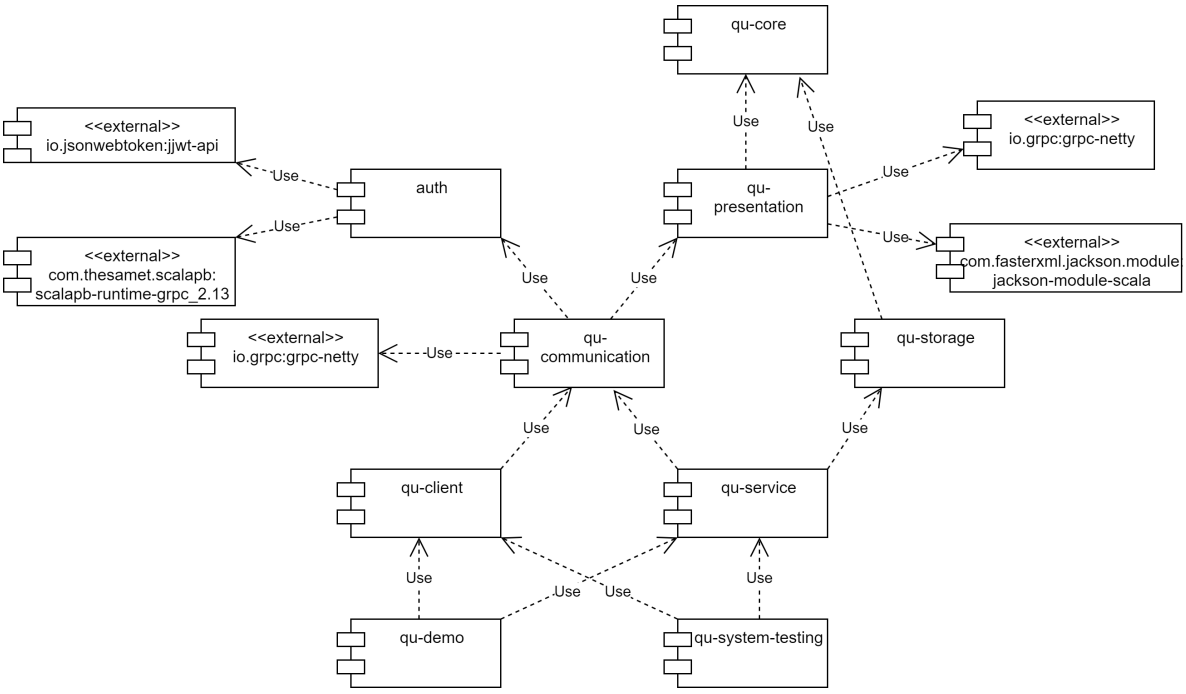


Figure 5: Organizzazione del progetto in sotto-progetti rappresentata attraverso Component Diagram UML.

5.1.2 Struttura del sistema distribuito

La struttura del sistema distribuito, costituito da server di autenticazione, client e repliche Q/U è mostrata in fig. 6. Un servizio di autenticazione resta in ascolto delle richieste di registrazione e autorizzazione attraverso due API distinte. Ogni client include un componente che si occupa di contattare tale server per la sua registrazione ed autenticazione preliminare all'avvio del protocollo vero e proprio. Ciascuna replica comunica con i client e con le restanti repliche; per i primi, espone un'interfaccia a cui sottomettere operazioni; per i secondi, un'interfaccia di richieste di sincronizzazione con cui recuperare una versione dell'oggetto localmente non disponibile (*object-sync*). Ciascuna replica mantiene lo storico dei timestamp corrispondenti alle operazioni invocate sull'oggetto assieme ad un deposito delle sue versioni; i client, invece, memorizzano il cached OHS ottenuto accumulando le RH delle repliche contattate. La strategia di quorum impiegata determina quali e quanti server contattare. Come imposto dal protocollo, le repliche devono essere in numero pari a $5f$, con f numero di guasti tollerati.

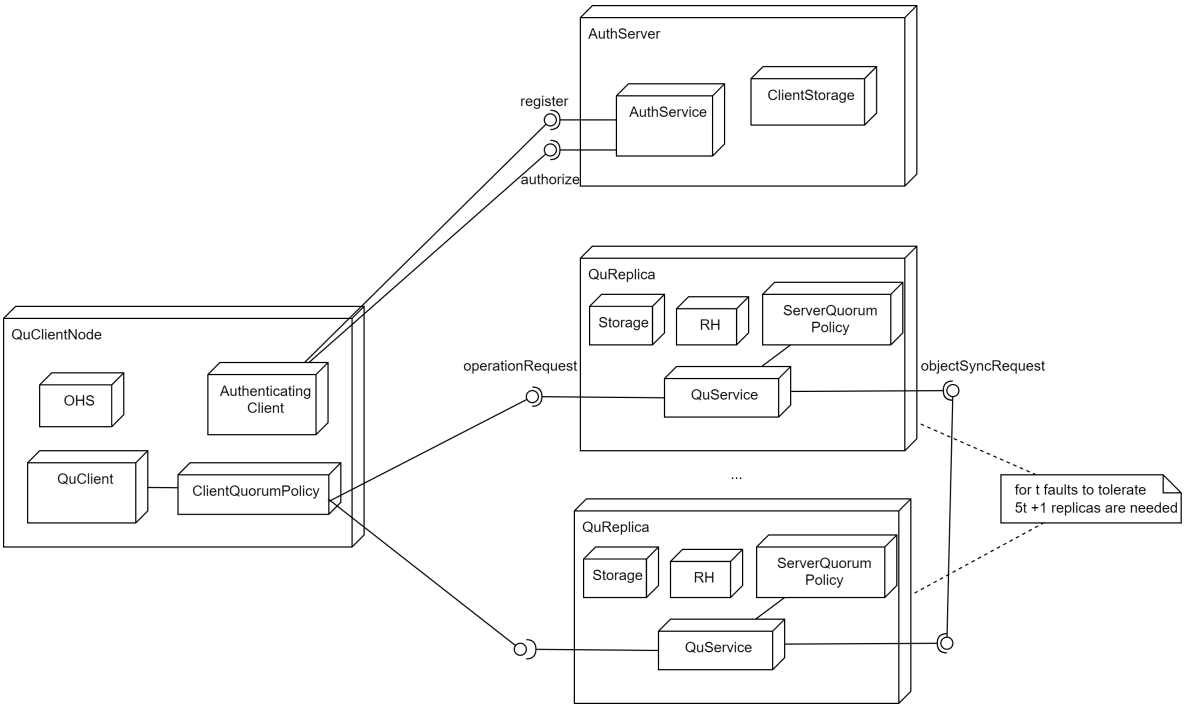


Figure 6: Diagramma UML di deployment del sistema che mostra le principali componenti hardware e software necessarie all'esecuzione del protocollo Q/U.

5.1.3 Struttura modulo qu-core

Modello Per la modellazione dell'insieme di concetti e strutture nucleo di Q/U, al fine di:

- Consentire una diversificazione derivante, ad esempio, dall'impiego di un differente

threshold system (come proposto dagli autori) riusando al contempo una base di comportamento comune e

- impedire di mischiare concetti di una implementazione da concetti di altre (*type-safety*)

  si è fatto riferimento al *polimorfismo a famiglie*, già individuato in letteratura come strumento per "condividere codice comune e preservare la *type-safety*" [19], applicando il *"gradual interface-to-timplementation"* pattern, a fronte del forte accoppiamento fra i concetti, non solo a classi ma ad un framework intero. Al contempo, la logica dei singoli aspetti richiesti dal protocollo è stata suddivisa in moduli, che sono iniettati nella descrizione astratta del modello attraverso *mixin* dando origine alla sua versione definitiva. AbstractQuModel modella, pertanto la descrizione di alto livello dell'insieme dei concetti, a sua volta raffinata progressivamente attraverso i seguenti *mixin*:

- Operations: definisce le astrazioni di operazioni della RSM.
- Crypto: contiene le utilità crittografiche per la verifica dell'integrità dei messaggi.
- OperationTypes: definisce le operazioni per la rappresentazione delle informazioni.
- Hashing: è un modulo per la rappresentazione efficiente di operazioni e OHS.

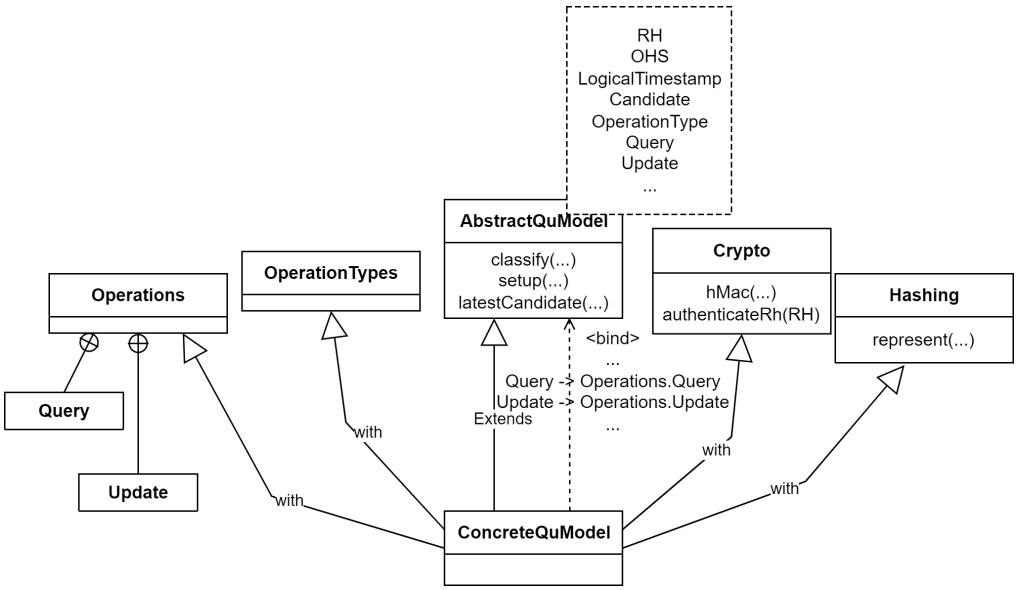


Figure 7: Modellazione della strutturazione delle astrazioni del dominio di Q/U attraverso diagramma delle classi.

Operazioni L'interfaccia basata su operazioni predisposta da Q/U è stata modellata mediante *Command pattern* [16] agevolando la definizione di un servizio attraverso l'approccio SMR. Come mostra la figura 8, il trait Operation presenta un'interfaccia per l'esecuzione di un'operazione che può essere (tramite ereditarietà) di aggiornamento (Update) o di interrogazione (Query). Per assecondare l'obbiettivo di *type safety*, il trait presenta due tipi parametrici che specificano il tipo dell'oggetto su cui è predisposta per essere invocata e il tipo di ritorno atteso dall'invocazione. Il metodo compute, il cui design è influenzato da elementi di programmazione funzionale, non ritorna solo il risultato dell'operazione, ma anche l'oggetto risultante dall'invocazione, immutato in caso di Query. Per:

- incentivare questo vincolo e
- facilitare una maggiore leggibilità

è stata introdotta AbstractOperation che, modellando compute come *Template Method* [16], ne specifica la struttura delegando alle sottoclassi la specifica dei metodi whatToReturn e updateObject, che presenta come astratti. A questo scopo, il trait Query definisce a compile time il vincolo di non modifica dell'oggetto restituendo in updateObject l'oggetto stesso immutato. Come mostra la fig 9, Operations, inoltre, ospita altre operazioni già pronte e riusabili costruite sulla base delle operazioni più semplici e di aiuto nella costruzione di una RSM, come UpdateReturningObject e GetObj, che raffina ulteriormente Query ritornando l'oggetto stesso.

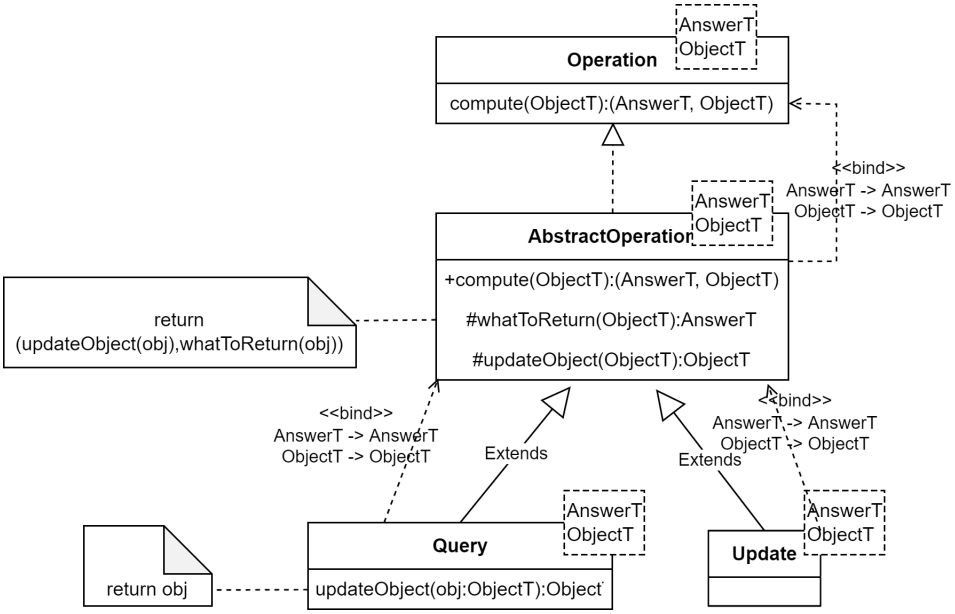


Figure 8: Diagramma UML delle classi che mostra il pattern *Template Method* in combinazione con il *Command pattern* applicati alla gerarchia delle operazioni.

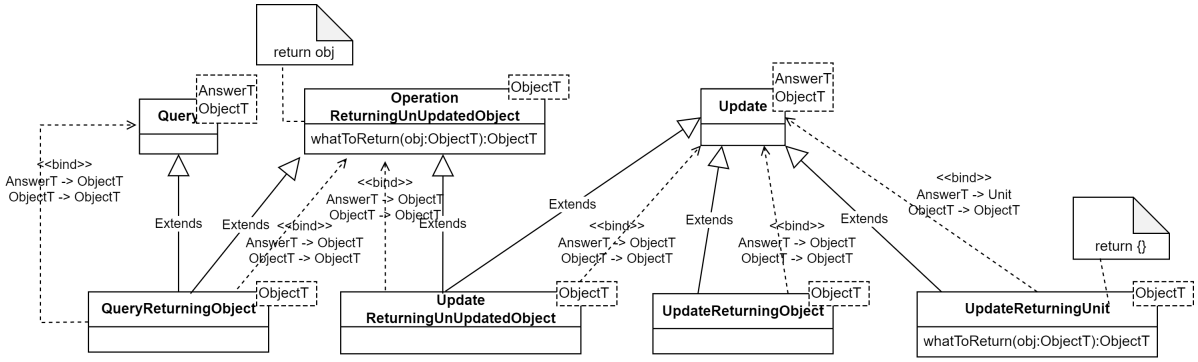


Figure 9: Dettaglio UML delle astrazioni costruite sulla base delle operazioni di base per favorire l'approccio State Machine Replication per la definizione dei servizi, ed appartenenti alla gerarchia di Operations modellata con *Command pattern*.

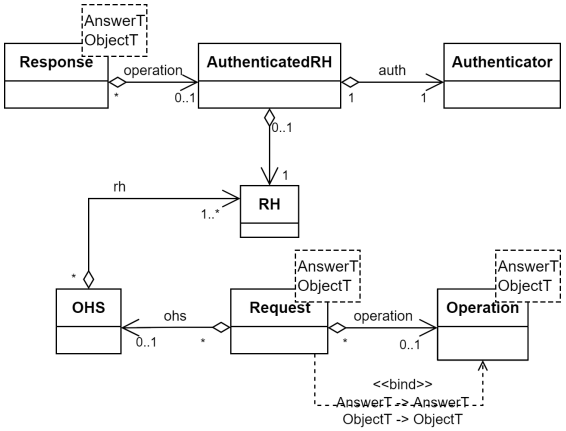
Messaggi Come meglio descritto più avanti, il protocollo prevede una API per l'interazione client-replica, che permette l'esecuzione delle operazioni, e una per la funzionalità di *object-sync* con cui ciascuna replica può recuperare dalle altre una versione di un oggetto che non conserva localmente. I messaggi previsti dal protocollo sono mostrati in fig. 10a e 10b.

- Richiesta di esecuzione operazione (Request): oltre al cached OHS, una richiesta di esecuzione di operazione contiene l'operazione che il client intende eseguire; la sua presenza è opzionale per supportare anche l'invio di operazioni di tipo barrier, inline barrier e copy, necessarie in occasione di repair. Il polimorfismo parametrico consente di ospitare tipi di dato differenti con lo stesso messaggio.

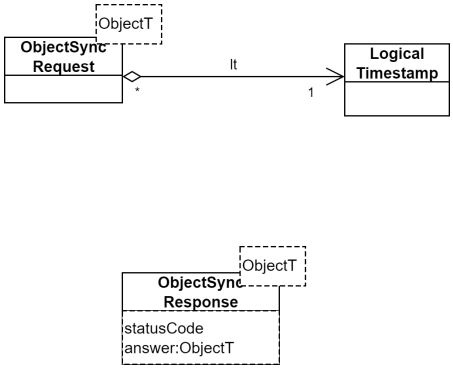
- Risposta alla richiesta di esecuzione dell'operazione (Response): una risposta ospita un codice di stato con cui comunicare la corretta conclusione, o meno, dell'operazione richiesta, la RH fornita degli autenticatori necessari alla sua validazione e il valore di ritorno corrispondente alla operazione.

- Richiesta di una versione dell'oggetto (ObjectSyncRequest): contiene il logical timestamp della versione dell'oggetto richiesta.

- Risposta alla richiesta di *object-sync* (ObjectSyncResponse): oltre ad un codice di stato che comunica eventuali fallimenti, contiene l'oggetto, che può essere assente se la replica contattata non ne dispone.



(a) Modellazione dei messaggi relativi alla sottomissione di un'operazione.



(b) Modellazione dei messaggi necessari per eseguire *object-sync* fra repliche.

5.1.4 Struttura progetto auth

Il progetto auth ospita il servizio che espone le API di registrazione ed autorizzazione per un generale utente, e un client per interagirvi attraverso *Remote Procedure Call* (RPC). Pertanto, come mostra la fig. 11, client e service, rispettivamente modellati da AuthClient e AuthService, condividono la stessa interfaccia, che astrae la distribuzione agli occhi dell'utente, costituita dai metodi register e authorize. Per renderla compatibile con diversi meccanismi di comunicazione o serializzazione (requisiti non funzionali 4.2.1.1.c e 4.2.1.1.b), astraendoli, la logica di *business* è stata isolata nella classe LocalAuthenticator, che modella un servizio di autenticazione in locale; la gestione delle richieste provenienti dalla rete è a invece a carico di AuthService, che, anziché re-implementarla, le delega al LocalAuthenticator di cui si compone: l'astrazione di AuthServer cattura invece gli aspetti di configurazione, avvio e terminazione, presentando la porta su cui ricevere le richieste e i metodi start e shutdown. Per rendere riusabile in più contesti il modulo, le API sono state rese disponibili sia in una versione sincrona che in una versione asincrona che incapsula la computazione in una *Future* e permette all'utente di non bloccare il proprio flusso di esecuzione delegandone la computazione ad un esecutore esterno.

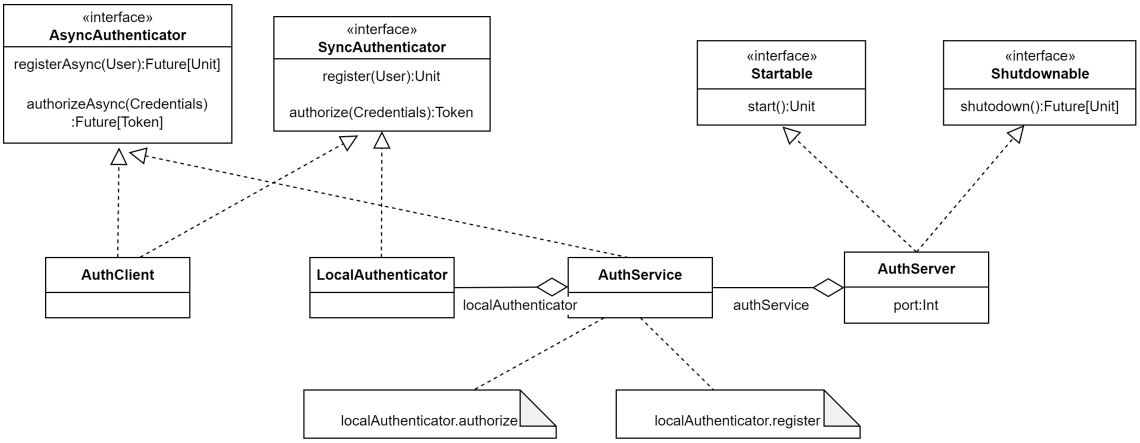


Figure 11: Diagramma delle classi UML per le principali astrazioni contenute nel modulo auth.

5.1.5 Struttura progetto qu-client

Un client Q/U espone attraverso il metodo submit un'interfaccia di sottomissione di operazioni scelte e predisposte dall'utente che incapsula la logica di tolleranza ai guasti e ne ritorna il risultato in modo trasparente. La sua modellazione è rappresentata nell'UML in fig. 12. Il *polimorfismo parametrico* (di O e A) consente di riusare il funzionamento del protocollo per diversi tipi di dato dell'oggetto e degli argomenti e tipi di ritorno delle operazioni invocate su di esso. In particolare, un client può essere impiegato per la sottomissione di aggiornamenti o interrogazioni relativi ad un oggetto di tipo generico O per ritornare risposte generiche in A. Per assecondare il requisito di compatibilità con diverse tecnologie di serializzazione, rimuovendo la duplicazione e non costringendo a riscrivere la logica lato client per la loro futura integrazione, QuClient include un ulteriore tipo generico, l'*higher kinded type* Transportable che definisce il tipo responsabile della serializzazione e deserializzazione delle richieste inviate dal client alle repliche e delle risposte giunte, specificabile dall'utente della libreria.

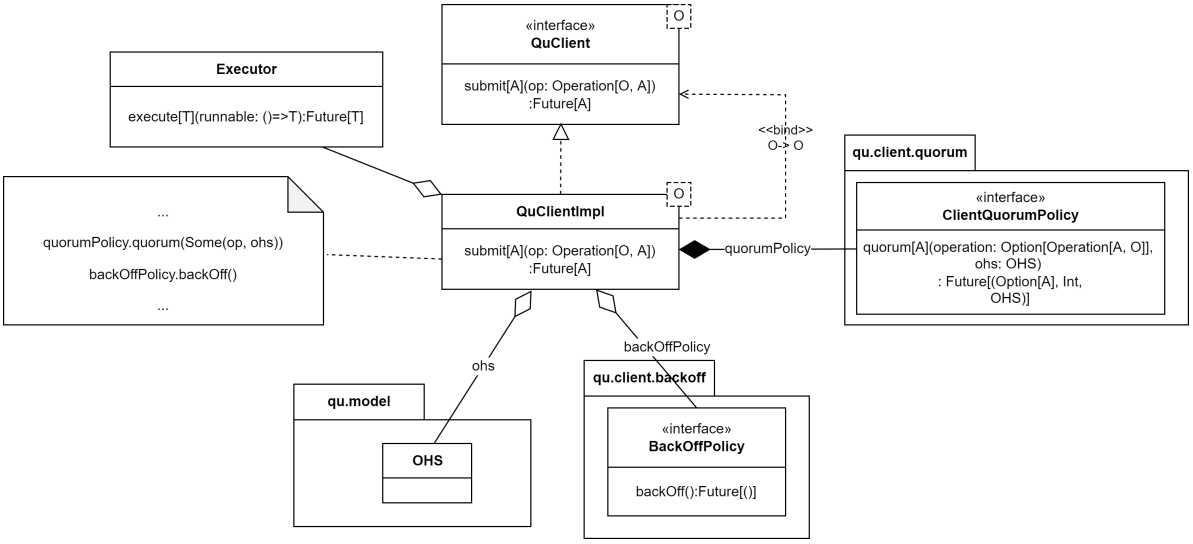


Figure 12: Diagramma UML delle classi che modella ad alto livello un client Q/U.

La classe QuClientImpl implementa QuClient includendo le ottimizzazioni descritte ai requisiti; in particolare, contiene il riferimento al cached OHS.

L'analisi del dominio ha evidenziato la compatibilità di Q/U con diverse strategie di scelta del quorum di repliche da coinvolgere (fra cui il preferred quorum già descritto) e con diverse *strategie* di backoff, potenzialmente configurabili a scelta dell'utente. Per questi motivi e per semplificare la classe QuClientImpl nel rispetto di SRP e DIP[29], anziché tramite ereditarietà, tali differenze di comportamento sono state isolate da QuClient in due interfacce separate (ClientQuorumPolicy e BackOffPolicy) attraverso il pattern *Strategy* [16], a cui QuClientImpl mantiene un riferimento per delegarne le rispettive logiche di funzionamento. Come mostra la fig. 13, al momento la libreria predispone, nei rispettivi package, una BroadcastClientPolicy che prevede di coinvolgere tutte le repliche del cluster a cui sottomettere una richiesta, una backoff policy di tipo esponenziale e una di tipo *random exponential*; in entrambi i casi, future estensioni (già presenti nell'articolo di presentazioni e suggerite dagli autori) saranno quindi iniettabili a posteriori riusando lo stesso client.

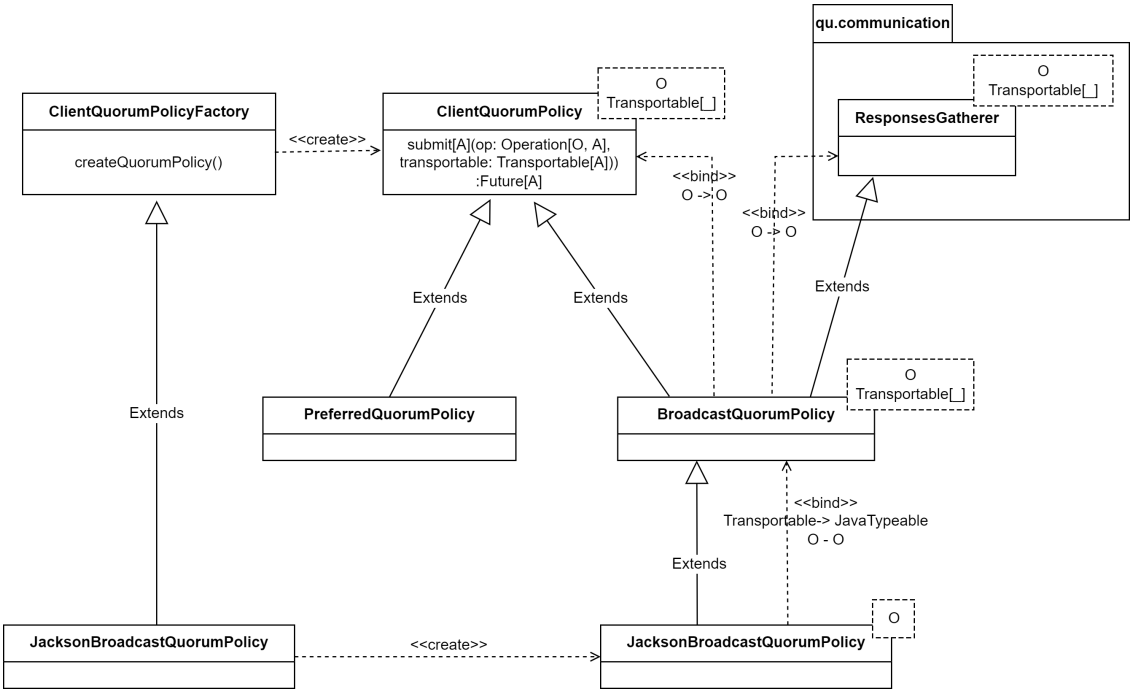


Figure 13: Diagramma UML delle classi che rappresenta la gerarchia delle *Strategy* di ClientQuorumPolicy e le *factory* per istanziarle.

Per agevolare la configurazione, facilitarne la validazione preliminare all'utilizzo, e ridurre la complessità di QuClientImpl, la logica di costruzione è stata disaccoppiata, nel rispetto di SRP[29], da quella di funzionamento e trasferita alla classe QuClient-Builder, responsabile di definire le informazioni sulle repliche (addServer) di configurare le soglie di tolleranza (setThresholds) e di costruirlo (build)[16]. La compatibilità con diverse tecnologie di serializzazione e la presenza di diverse strategie iniettabili nel client identifica in realtà una famiglia di builder (mostrata in fig. 14); per agevolarne la scelta da parte dell'utente programmatore sono state predisposte dei *factory method*[16] di builder che gli consentono di istanziarli ed utilizzarli ignorando i dettagli relativi ai costruttori.

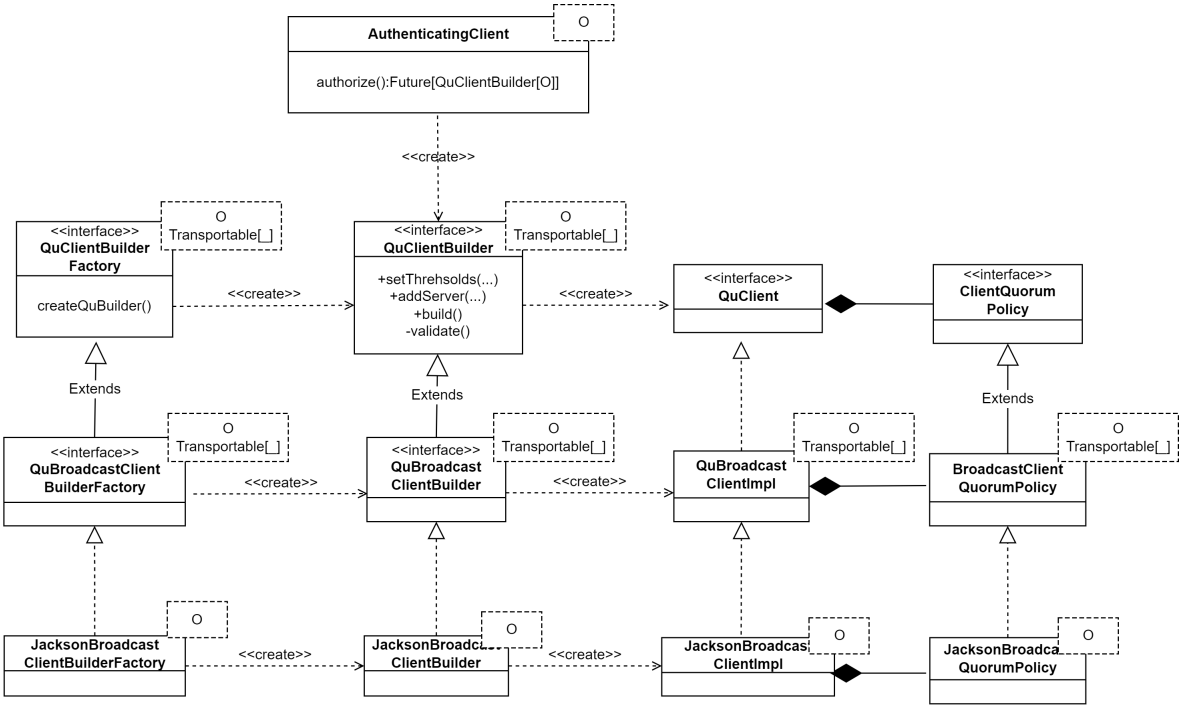


Figure 14: Diagramma UML delle classi che rappresenta la logica creazionale di QuClient attraverso la gerarchia di builder e le i *factory method* annessi.

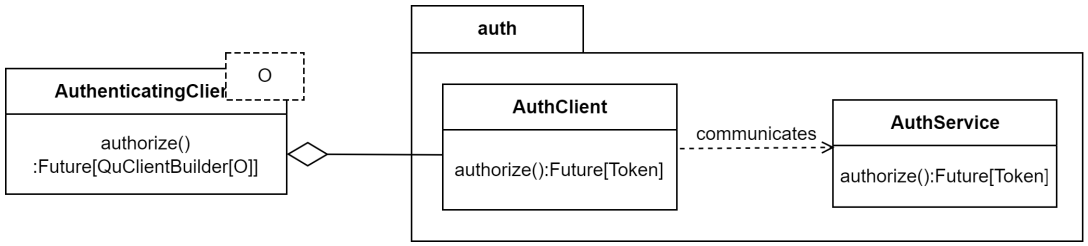


Figure 15: Diagramma UML delle classi che rappresenta l'interfaccia e le astrazioni coinvolti per l'autenticazione e registrazione di un client Q/U.

Al fine di impedire a client non autenticati di sottomettere operazioni alle repliche, le API di autenticazione/registrazione sono state separate da quelle di sottomissione ed isolate nella classe AuthenticatingQuClient (mostrata in fig.15), che incapsula a sua volta un client di autenticazione a cui delegare la sottomissione richieste al servizio del modulo auth. Come mostra la fig. 14, previa registrazione con successo e credenziali corrette, il metodo authorize ritorna il builder con cui configurare il client Q/U vero e proprio.

5.1.6 Struttura progetto qu-service

La modellazione di alto livello di un servizio, responsabile della logica di esecuzione delle repliche di Q/U, è mostrata in fig. 16. QuService permette di replicare un oggetto di tipo generico O ed interagirvi sottoponendo operazioni in maniera faut-tolerant attraverso due rotte:

- OperationsRequest: è il punto di accesso alle richieste dei client e determina, a fronte dell'ispezione dell'OHS contenuto, l'operazione da eseguire sulla versione corrente dell'oggetto;
- ObjectSyncRequest: processa le richieste di sincronizzazione delle altre repliche che le effettuano per recuperare versioni dell'oggetto (individuate mediante logical Timestamp) di cui non dispongono in locale.

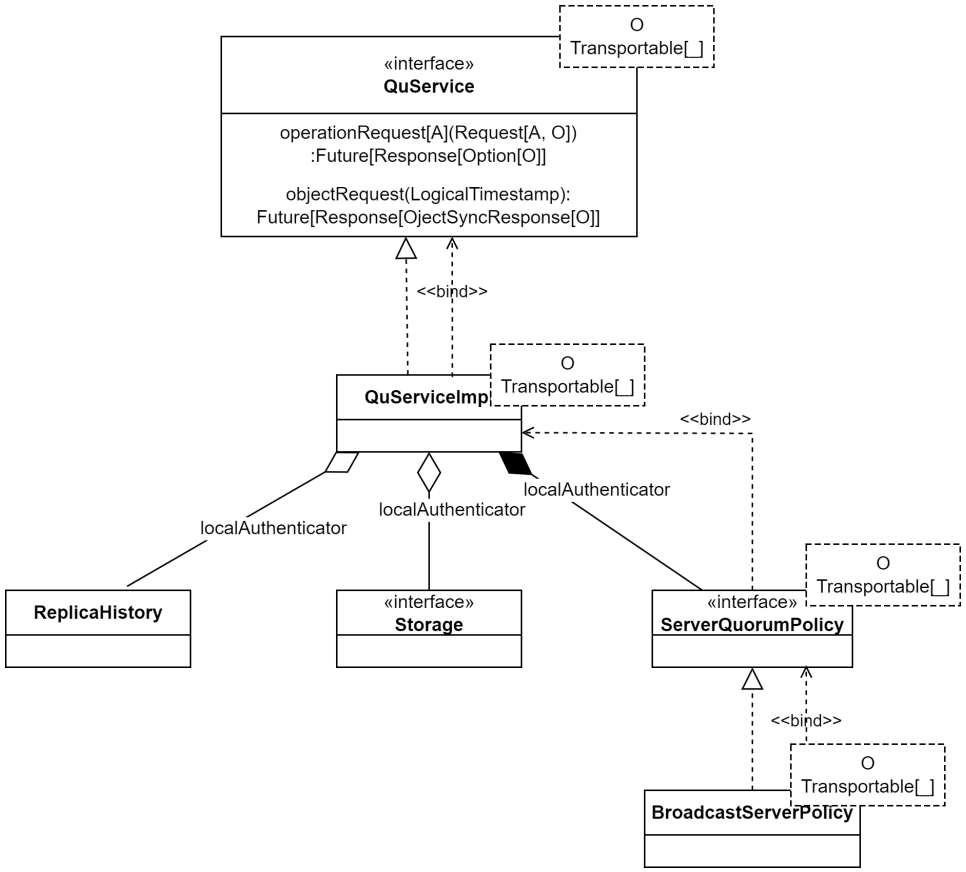


Figure 16: Modellazione di alto livello di un servizio Q/U.

Analogamente a QuClient, per consentire il supporto a formati e tecnologie di serializzazione differenti, l'ulteriore tipo parametrico definisce il tipo responsabile della serializzazione/deserializzazione di richieste e risposte senza re-implementare la logica

in futuro. QuServiceImpl è l'implementazione fornita dalla libreria per QuService che integra le ottimizzazioni descritte nei requisiti. ServerQuorumPolicy è l'interfaccia che definisce la logica di scelta e coinvolgimento delle altre repliche in occasione delle operazioni di *object-sync*. Come per QuClient, visto che Q/U non vincola la restante logica del protocollo alla sua adozione, la si è modellata come una *Strategy* indipendente a cui viene delegata l'interazione con le altre repliche. Al momento, la libreria include una implementazione che effettua un broadcast alle repliche in occasione della sincronizzazione (ServerBroadcastPolicy), ma approcci più intelligenti o efficienti saranno quindi integrabili agevolmente senza la modifica di QuServiceImpl, che rimane inoltre più semplice (SRP). Come descritto dall'articolo, per rendere resiliente ai guasti il protocollo, ciascuna replica mantiene uno storico delle versioni e un deposito degli stati intermedi dell'oggetto. Tali astrazioni sono modellate dalle ulteriori interfacce *Strategy* ReplicaHistory e Storage, rispettivamente definite nei progetti qu-core e qu-storage, di cui QuServiceImpl si compone; anche in questi due casi, quindi, l'integrazione di nuovi meccanismi (in particolare di persistenza) più completi ed efficienti risulta agevolata e apre le porte all'adozione di una moltitudine di comportamenti potenzialmente configurabili dall'utente.

Nel rispetto dei requisiti non funzionali e in analogia al LocalAuthenticator del modulo auth, QuServiceImpl, isolando la *business* logic del protocollo, non vincola la scelta tecnologica per la comunicazione o la scrittura lato server che per questo è stata finora ignorata nella descrizione. A tal proposito, la fig.17 mostra, a titolo di esempio, l'integrazione di un servizio Q/U (QuService) in un servizio gRPC(GrpcQuService) a cui delegare le invocazioni alle due rotte mediante aggregazione. L'estensione di BindableService permette di iniettare il servizio ottenuto in un server gRPC già esistente. Come meglio spiegato alla sezione 9, la configurazione di un GrpcQuService è facilitata da GrpcQuServiceBuilder, che si compone delle *Strategy* presentation.MethodDescriptorFactory e ServerQuorumPolicy, al fine di assecondare la costruzione di servizi realizzati attraverso tecnologie differenti. Per l'aggiunta di servizi con protocolli differenti sarà necessario definire le *factory* corrispondenti e il nuovo *builder* con cui istanziare il nuovo servizio.

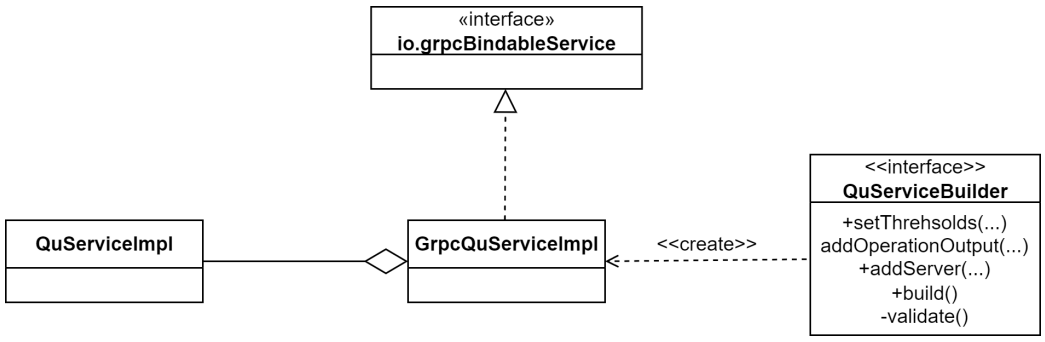


Figure 17: Dettaglio di alto livello dell'iniezione di gRPC come possibile middleware di comunicazione.

Un servizio Q/U prescinde dalle informazioni relative al dispiegamento della replica,

fra cui la porta su cui restare in ascolto e l'interfaccia di avvio e terminazione. Tali informazioni sono modellate da un'interfaccia separata, QuServer, che allo scopo implementa Startable e Shutdownable, come mostra la fig. 18. In particolare, QuServer rappresenta un *Facade* [16] che maschera la complessità relativa alla configurazione delle rotte, alla verifica dell'autenticazione dei client, e, in generale, alla libreria con cui è implementato il servizio (in questo caso, come mostrato in fig. 18, gRPC). Questo approccio ha il vantaggio di ridurre le dipendenze, promuovere l'adozione futura tecnologie di comunicazione differenti (req. 4.2.1.c) senza modificare il codice dell'utente finale e predisporre all'esterno un'interfaccia semplificata. Infine, per fornire un accesso alle API simmetrico a quello predisposto lato client, agevolando la configurazione delle repliche, la libreria integra un builder anche per il QuServer (QuServerBuider), con cui impostare le informazioni necessarie. La compatibilità con tecnologie di serializzazione, di trasporto sulla rete, di autenticazione e meccanismi di quorum differenti identifica in realtà varie famiglie di builder; per favorire la configurazione e scelta del più appropriato l'utente può scegliere la *factory* (QuServerBuilderFactory) più adatta fra le disponibili. Ancora una volta, per l'integrazione di servizi con nuove tecnologie è sufficiente definire le strategie corrispondenti e fornirle come parametro a QuServerBuilder, creando un nuovo builder con cui istanziare le nuove repliche.

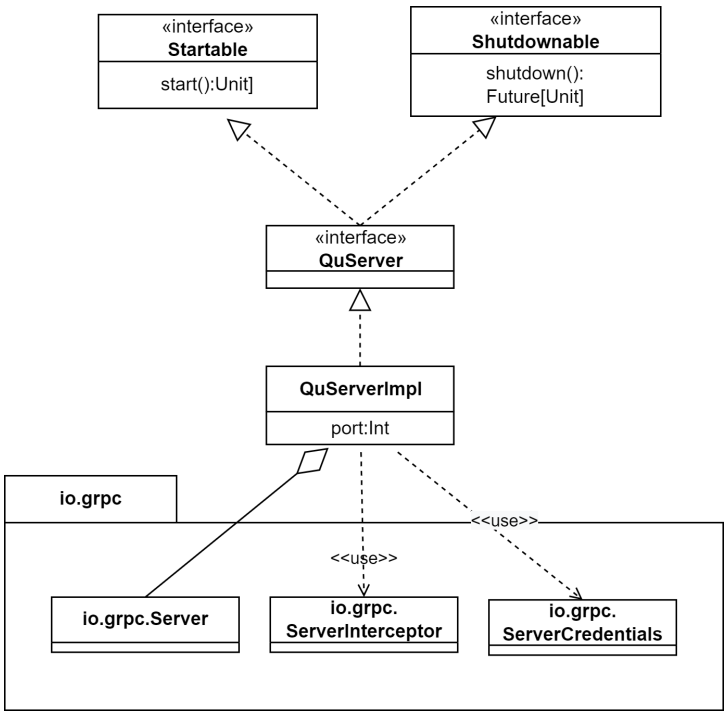


Figure 18: Applicazione del pattern *Facade* nella modellazione di un server Q/U.

5.1.7 Struttura progetto qu-communication

qu-communication isola interfacce e classi comuni a client e service responsabili dell'interazione fra di loro. Allo scopo, contiene un package (qu.client.stub) riusabile in altri contesti costituito da un insieme di stub asincrone lato client per la comunicazione di varie tipologie: intra-processo, distribuita sulla rete, con criptaggio e senza. Una client stub asincrona espone un'interfaccia per l'invio e la ricezione di oggetti di tipo generico. È molto probabile che la creazione di una stub determini la necessità di rilasciare le risorse al termine del suo utilizzo; l'interfaccia ShutdownableAsyncStub cattura questo aspetto estendendola con l'inclusione della logica annessa.

Per agevolarne la costruzione, nascondere i dettagli relativi ai costruttori e consentire la configurazione scegliendo una tra più famiglie di stub, due *Abstract Factory*[16] definiscono i metodi di costruzione corrispondenti alle tipologie elencate, rispettivamente per stub non autenticate e autenticate tramite token JWT. Ancora una volta, le due interfacce, di cui una mostrata in fig. 19, astraggono la tecnologia di serializzazione delegata alla *Strategy* di tipo Transportable: ogni tecnologia identifica una *factory* concreta in grado di istanziare le stub corrispondenti; per l'adozione futura di nuove sarà sufficiente definire la *factory* concreta, eventualmente come *Singleton*[16], in grado di creare le nuove stub corrispondenti.

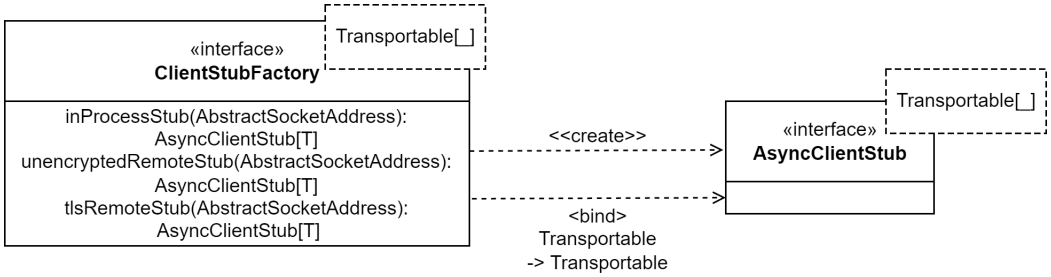


Figure 19: Dettaglio della *Abstract Factory* per la generazione delle stub asincrone non autenticate.

La classe ResponsesGatherer si occupa di inviare le richieste generiche in ReqT e reperire le risposte generiche in RespT dai nodi della rete incapsulando la logica di ritrasmissione (tramite Scheduler) necessaria per far fronte a guasti di tipo *crash*. Per farlo, si serve di stub associate alle repliche corrispondenti, identificate tramite il proprio ID. Anche ResponsesGatherer implementa Shutdownable; per il rilascio delle risorse, si è sfruttato il pattern *Composite* [16], attraverso il quale la richiesta di terminazione (shutdown) viene delegata alle proprie stub, che condividono la stessa interfaccia. Il tipo generico Transportable la rende, infine, compatibile con stub differenti, istanziabili attraverso i metodi della *factory* descritta.

5.1.8 Struttura progetto demo

L'applicazione di demo costituisce un progetto indipendente dalla libreria e a scopo dimostrativo. La sua architettura segue il pattern Model-View-Controller (MVC), come

mostrato in fig. 20. Con questa architettura, possono essere aggiunte in futuro nuove interfacce, anche grafiche, mantenendo invariate le implementazioni di Controller e del model (SmrSystemImpl) già scritte, soddisfacendo il requisito non funzionale corrispondente. Inoltre, anche eventuali modifiche implementative al modello non si riflettono sulla view; pertanto, lo sviluppo di Model e View è potuto proseguire in parallelo.

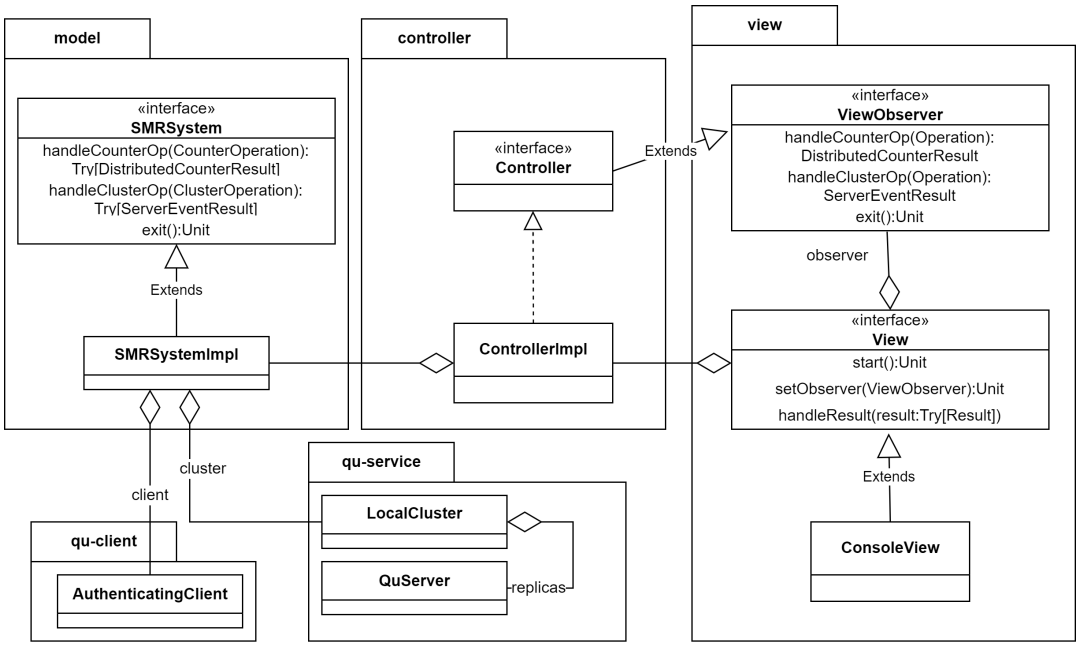


Figure 20: Diagramma UML delle classi che rappresenta l'adozione del pattern architetturale Model-View-Controller nell'organizzazione del sotto-progetto qu-demo, contenente un esempio di SMR.

Il modello è costituito da un sistema composto da un client e un insieme di server (SMRSystem) che replicano un contatore distribuito. La sua interfaccia consente all'utente dell'applicazione di interagire attraverso un client Q/U con le repliche per la sottomissione di operazioni di incremento, decremento e recupero del valore corrente, ma anche di simularne un guasto, recuperarne lo stato e di terminarne l'esecuzione; sebbene Controller sia sufficientemente generale da mascherare repliche realmente distribuite, SmrSystemImpl è un implementazione di tale sistema in locale, e si serve delle astrazioni definite nei sotto-progetti client e server per la realizzazione dell'interfaccia descritta.

Nella variante di MVC adottata, l'interfaccia View non comunica direttamente con il modello; l'interazione tra view e modello è infatti mediata da un Controller responsabile di intercettare gli eventi della View, comandare le modifiche al modello, aggiornare di conseguenza la View. In particolare, per disaccoppiare View e Controller, la loro interazione è stata modellata rendendo Controller un *Observer*[16], registrabile con addObserver, e la view un *Observable*; la generazione di un evento da parte dell'utente, sia esso la sottomissione di operazioni sul contatore (handleCounterOp) o di gestione del

cluster(handleClusterOp) si rifletterà in cascata sull'invocazione del rispettivo metodo del controller. Quando, in risposta, il modello sarà aggiornato, il risultato della modifica (SmrEventResult), di successo o di fallimento, sarà comunicata alla View e gestito di conseguenza.

La View, anch'essa definita nel proprio package, gestisce la logica di ricezione degli input dell'utente e di notifica dei risultati. ConsoleView la estende modellando, infine, una semplice interfaccia a riga di comando. L'input dell'utente viene ricondotto ad uno dei comandi del sealed trait (AbstractCliCmd) attraverso la procedura parse definita nel *companion object*, ed innesca la notifica degli eventi al controller che si rifletterà in cascata sul modello, quindi determinando l'aggiornamento della view, qui solo di tipo testuale.

5.2 Comportamento

Si descrive nel seguito, mediante formalismo UML, la dimensione comportamentale per alcuni componenti di rilievo del sistema.

5.2.1 Comportamento servizio Q/U

Il comportamento di un servizio Q/U alla luce delle ottimizzazioni citate nei requisiti e descritte alla sezione Protocollo **Query/Update** è mostrato in fig. 21. Alla ricezione di una richiesta di un client, si procede alla validazione delle RH dell'OHS contenuto; i rispettivi autenticatori vengono controllati con quelli generati tramite HMAC a partire dalle RH che, in caso di non corrispondenza, vengono sovrascritte con il loro valore iniziale. A partire dall'OHS risultante, una procedura di configurazione determina il timestamp della operazione richiesta, assieme a quello relativo all'operazione che ha determinato la versione dell'oggetto a cui essa si riferisce; quindi, si determina il tipo di operazione di eseguire (se method, barrier, copy, inline method o inline barrier). Prima di procedere, il servizio si assicura di non avere già ricevuto la stessa risposta e, nel caso, risponde al client con la risposta corrispondente, recuperata dal proprio deposito, per poi concludere la comunicazione. Diversamente, si controlla se l'OHS inviato dal client è aggiornato al tempo logico del servizio o se invece quest'ultimo conserva un timestamp posteriore: nel primo caso, se l'operazione è un'interrogazione, si restituisce la versione dell'oggetto risultante dall'operazione (e la risposta annessa) più recente; altrimenti, si conclude la comunicazione inviando la RH aggiornata per consentire al client di riallinearsi. Se, a questo punto, la comunicazione non è ancora stata conclusa e il tipo di operazione che emerge dall'OHS è uno fra copy, inline method e method, il servizio recupera l'oggetto conditioned-on in modo da invocare l'operazione richiesta: se non disponibile localmente, può rivelarsi necessaria un'operazione di *object-sync* con cui richiederlo dalle altre repliche: una volta recuperato, l'operazione può essere invocata. Solo in caso di aggiornamento, si procede quindi ad aggiornare la RH e l'autenticatore, ricalcolando l'HMAC, e a salvare l'oggetto risultante e il valore restituito dall'operazione ritornandoli con successo al client, non prima di aver eliminato i candidati e le versioni locali non più necessarie (pruning).

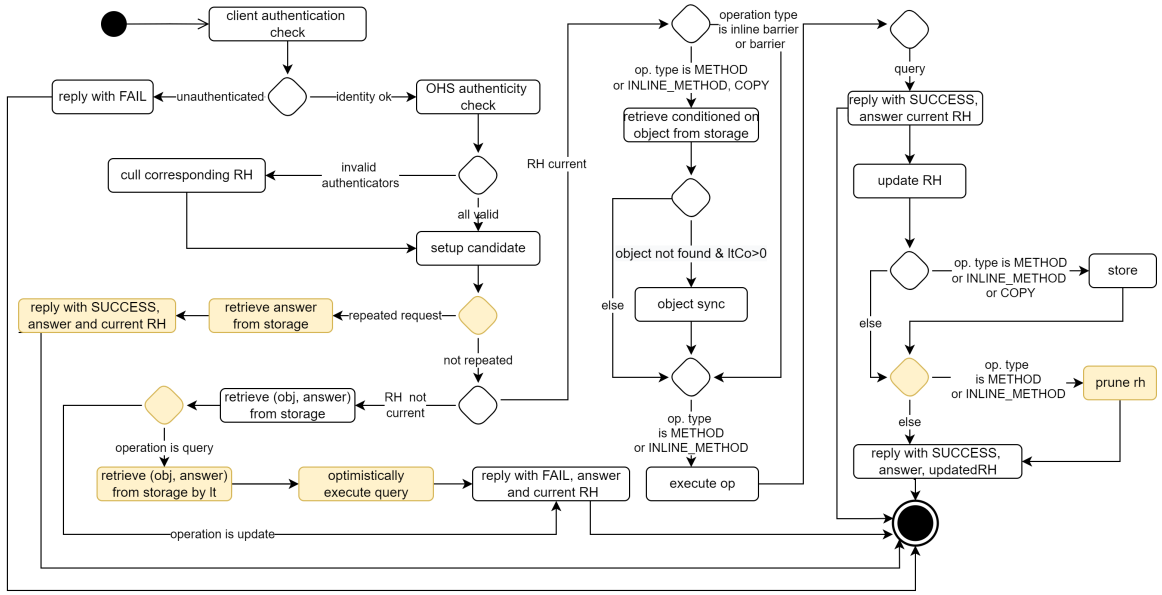


Figure 21: Diagramma UML di attività che mostra il flusso di operazioni per il processamento di una richiesta da parte di un client. Le attività colorate fanno riferimento alle estensioni al funzionamento di base descritte alla sezione 2.

5.3 Interazione

5.3.1 Interazione prevista dal protocollo

Si analizzano nel seguito le interazioni previste dai principali scenari di funzionamento del protocollo realizzato comprensivo delle ottimizzazioni al funzionamento di base, considerando un contatore che espone tre metodi (incremento, decremento e reperimento del valore attuale), e il più piccolo *quorum system* necessario a tollerare un server che subisce un guasto (t=1). Dal formalismo UML utilizzato si può evincere la natura asincrona degli scambi. Inoltre, le RH contenute nelle risposte alle richieste di operazioni non riportano i conditioned-on timestamp (es. [3, 1, 0] anziché [(3,1),(1,0),(0,0)]). $\perp$ indica, invece, l'assenza di un valore, come nel caso del campo operazione in occasione dell'invio di richieste di repair.

Autenticazione e configurazione client La fig. 22 mostra l'interazione necessaria per autenticare e configurare un client preliminarmente all'esecuzione del protocollo. Per usufruire delle API di un client Q/U, l'utente della libreria deve registrarsi e quindi autenticarsi tramite AuthenticatingClient, che incapsula la logica di autenticazione isolandola da quella per l'esecuzione delle operazioni. La richiesta viene quindi delegata ad un AuthClient, un semplice e riusabile client che si occupa di sottometterla al servizio di autenticazione, inoltrandola in modo asincrono attraverso la rete accompagnata dalle credenziali e dall'id del client. Una volta giunte a destinazione, il loro controllo avviene interrogando il deposito mantenuto dal servizio. In caso di mancato riconoscimento

del client si procede restituendo in risposta un errore che viene comunicato in cascata all'utente della libreria. Diversamente, l'avvenuta autenticazione determina la creazione da parte dell'AuthenticatingClient del builder per il client Q/U vero e proprio, poi ritornato all'utente che può interagirvi con una semantica sincrona per la configurazione degli aspetti personalizzabili del protocollo, come i riferimenti delle repliche e le soglie di tolleranza ai guasti. La conclusione della configurazione da parte dell'utente innesca la validazione, la quale rende finalmente disponibile il client Q/U con cui sottomettere le operazioni all'insieme di repliche.

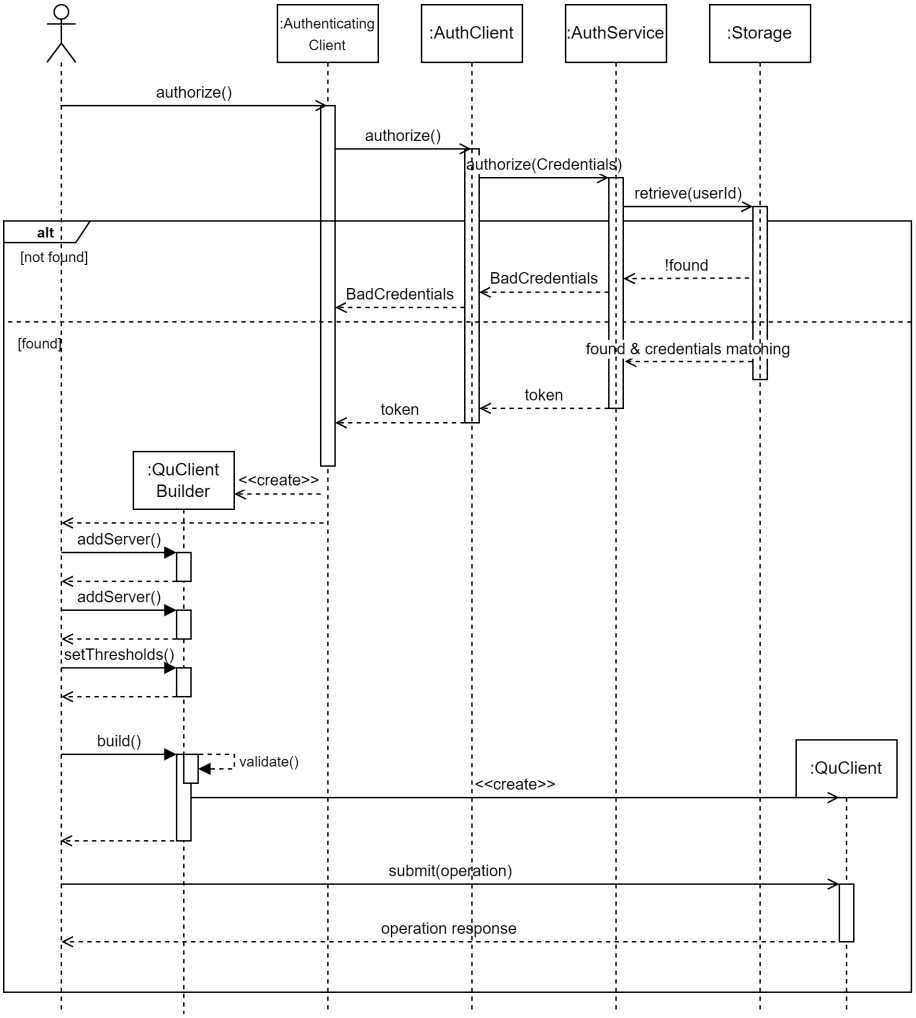


Figure 22: Diagramma UML di sequenza che mostra l'interazione fra i principali componenti del protocollo coinvolti per autenticare e configurare un client Q/U.

Scenario di Interazione senza contesa e senza guasti In figura 23 è mostrata un'esecuzione del protocollo in assenza di concorrenza nelle richieste dei client e di guasti delle repliche. Come si evince, tale scenario rappresenta il più efficiente perché richiede un singolo passo

di comunicazione per la realizzazione di ognuna delle richieste, siano aggiornamenti o interrogazioni. Le risposte delle repliche, che nel secondo caso contengono l'aggiornamento della propria RH, vengono ricevute dalla quorum policy senza necessità di ritrasmissioni, e si riflettono sulla precoce determinazione di un candidato completo e della risposta corrispondente, che viene restituita al client. Come già descritto, l'assenza di guasti, e soprattutto di contesa, costituisce il contesto più favorevole per Q/U dal punto di vista delle prestazioni rispetto ai suoi precursori (BFT e PBFT).

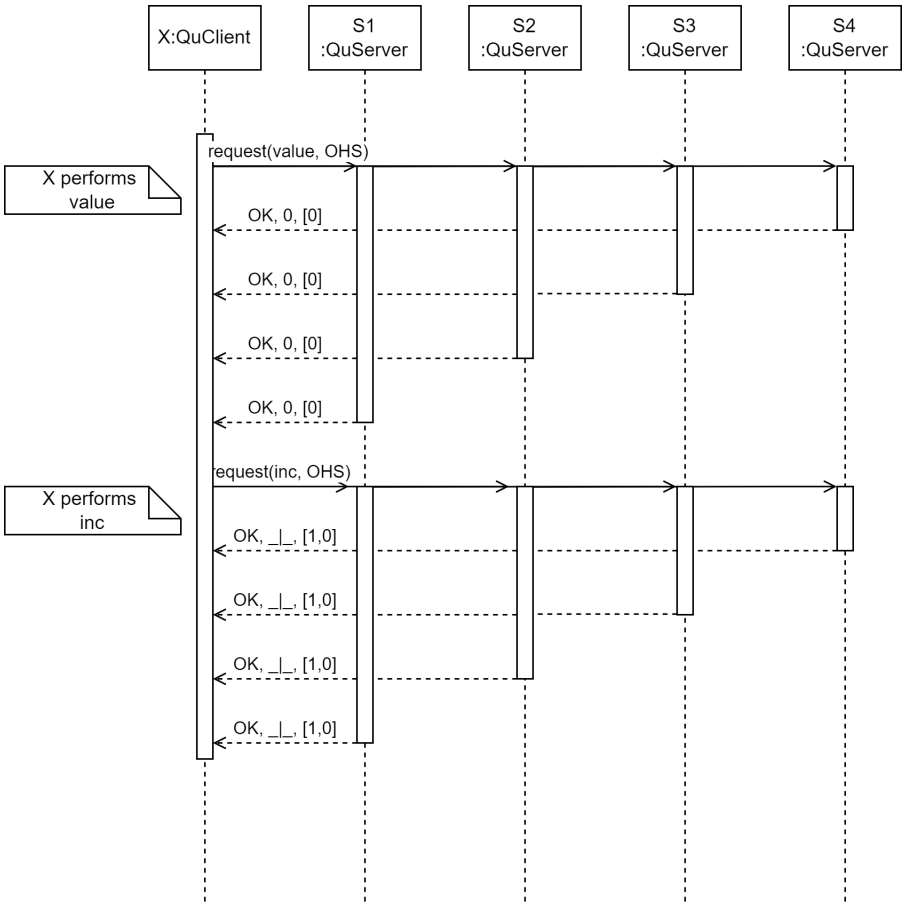


Figure 23: Interazione fra un client autenticato e le repliche Q/U in assenza di contesa nelle richieste sottomesse.

Scenario di interazione con inline repair La fig. 24 mostra la situazione in cui un nuovo client interagisce con un quorum di repliche che non coincide con quello coinvolto dal client precedente (in questo scenario si suppone che la replica S4 non fosse stata coinvolta dal primo aggiornamento da parte del client X); tale situazione può avvenire nel caso in cui la locazione geografica dei client che sottomettono le operazioni differisce sostanzialmente. Nel caso in esame, il client Y, ignaro del precedente aggiornamento eseguito da X, sottomette un'interrogazione a tutti i server: i primi tre catalogano la richiesta come

obsoleta e, dopo aver reperito la risposta dal proprio storage, falliscono (ritornano FAIL) ma, grazie all'ottimizzazione discussa alla sezione 2, eseguono ottimisticamente la query restituendo la propria RH aggiornata, mentre il quarto, non aggiornato, restituisce la sua ancora allo stato iniziale completando la risposta con successo. A causa di un ritardo della trasmissione della risposta di S1, a seguito del primo scambio, il client Y riscontra nell'OHS accumulato dalla quorum policy la presenza di un candidato non completo, bensì repairable, e la contestuale assenza di contesa; pertanto, avvia la riparazione: dopo aver eseguito backoff, procede ad una richiesta di repair, che viene inviata ai server attraverso la quorum policy. Tutti i server tranne il quarto procedono classificando la richiesta come ripetuta (a seguito della "riparazione" che individua l'operazione originale come quella da eseguire) e ritornano con successo grazie alla ottimizzazione specifica introdotta alla sezione 2. Ricevuta da S4, invece, la richiesta innesca la procedura di *object-sync* con cui essa ottiene dalle altre la versione dell'oggetto non posseduta in locale (1); l'ottimizzazione di inline repair consente, inoltre, di riallineare contestualmente la RH di S4 evitando un ulteriore scambio con il client con cui realizzare un'operazione di copy. Dopo aver riparato la RH di S4, un ultimo scambio consente, infine, al client Y di ottenere ottimisticamente il risultato richiesto a seguito della raccolta di risposte in numero sufficiente (q) da parte della quorum policy.

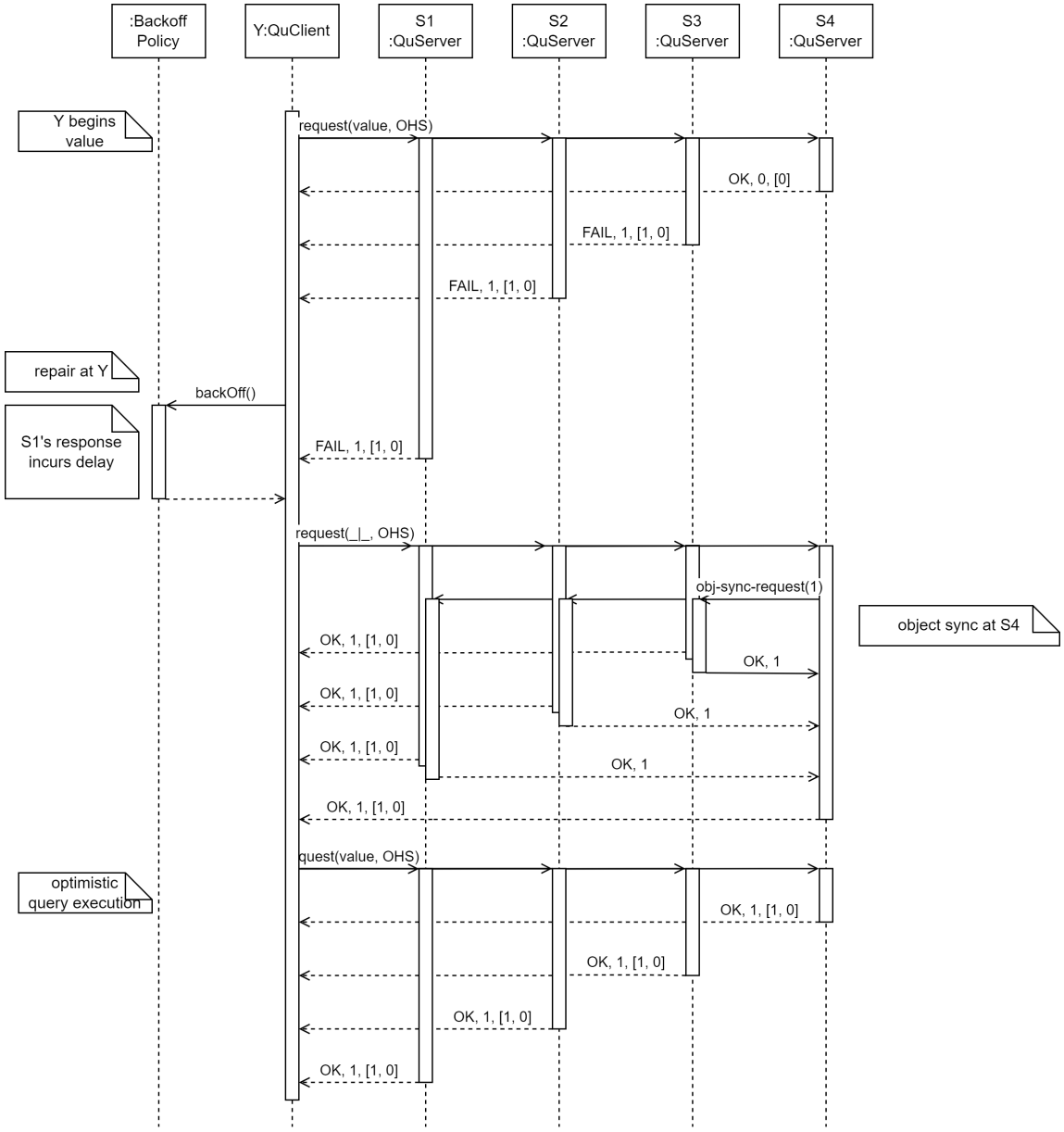


Figure 24: Interazione fra un client autenticato e le repliche Q/U in presenza di disallineamento fra le versioni conservate in locale e conseguente procedura di *object-sync* ed inline repair[1].

[1]Nonostante UML predisponga il frammento combinato *par* per la rappresentazione di flussi concorrenti nel processamento di invocazioni da parte dei classificatori, si è ritenuta in questo contesto più chiara la notazione impiegata, che affianca le linee di vita, come del resto scelto da altri tool.

Scenario di interazione con contesa In fig. 25 è mostrato l'interazione fra le principali componenti del protocollo in risposta ad una situazione di contesa. In questo caso, lo scenario raffigurato è temporalmente successivo a quello sopra descritto. Dopo la sottomissione alle proprie quorum policy delle rispettive operazioni di aggiornamento, fra loro concorrenti, le prime tre repliche ricevono per prima la richiesta del client X, mentre le ultime tre quella del client Y. A causa di un ritardo di trasmissione della risposta da S1, il client X ottiene solo due successi e un fallimento; per lo stesso motivo, il client Y riceve un successo e due fallimenti: entrambi constatano la presenza di un candidato repairable, ma son costretti ad avviare la procedura di repair invocando backoff. Tuttavia, il client X ritorna prima e sottopone, completandole, una richiesta di barrier e una di copy, con cui la contesa viene infine risolta riallineando lo stato delle repliche. Si può, inoltre, notare il pruning delle RH in risposta alla richiesta di incremento (inc) da parte di Y e X, che non presentano il candidato iniziale.

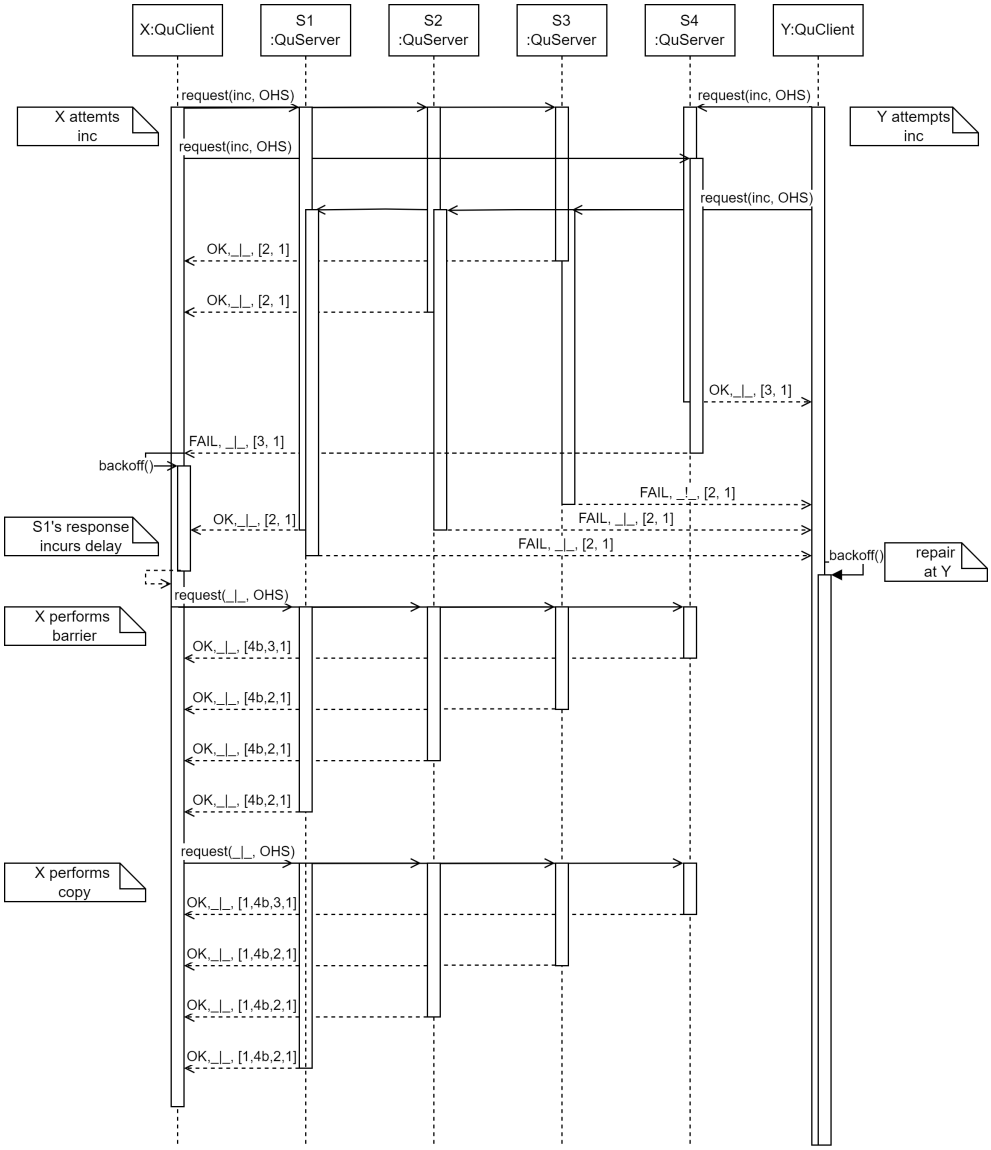


Figure 25: Interazione fra due client autenticati contendenti per l'esecuzione di un'operazione di aggiornamento.

6 Dettagli implementativi

In questa sezione si presentano e discutono le tecnologie e alcuni dei dettagli implementativi che caratterizzano l'implementazione ottenuta dal design sopra descritto.

6.1 Tecnologie utilizzate

scala Linguaggio multi-paradigma che supporta caratteristiche comuni al mondo Object-oriented (OO) e alla programmazione funzionale, scala è stato scelto per il suo *type system* potente ed espressivo, considerato il focus sulla *type safety* del progetto, per la possibilità di adottare una visione OO integrando i vantaggi della programmazione funzionale, oltre che per i suoi costrutti scalabili ed avanzati che consentono di agevolare non solo l'implementazione, ma anche la progettazione di un sistema non banale come un protocollo del consenso, riducendo il gap d'astrazione fra il problema e gli artefatti prodotti. Un'ulteriore motivazione è stata, inoltre, il diffuso e consolidato ecosistema di librerie, sia per il codice di produzione, fra cui gRPC, sia, in particolare, per il testing.

gRPC gRPC è un framework moderno, open source e con un focus sulle prestazioni e l'interoperabilità per la realizzazione di servizi basati sull'astrazione di RPC. Oltre a connettere servizi, dispositivi e applicazioni, include un supporto all'autenticazione, allo streaming asincrono, al load balancing, tracing e health checking. Il suo ruolo nel progetto è centrale. Sulla sua scelta hanno inciso le sue caratteristiche di:

- Interoperabilità: essendo implementati in gRPC, i servizi di autenticazione e di Q/U sono già in grado di rispondere alle richieste provenienti da client scritti in diversi linguaggi, ma anche la logica client può essere facilmente tradotta per assecondare diverse piattaforme.
- Supporto asincrono, con cui predisporre all'utente un'API non bloccante con semantica asincrona, ritenuta l'approccio da scegliere nella costruzione di sistemi distribuiti.
- Efficienza: l'adozione di altri approcci, come REST, è stata valutata, ma si è preferito scegliere, per il nucleo del protocollo, una tecnologia più performante come gRPC su cui costruire, in un secondo momento, viste di più alto livello.
- Compatibilità con proto e protobuf, che ha consentito un più rapido sviluppo attraverso la generazione di codice mediante il primo usufruendo, inoltre, delle prestazioni offerte dal secondo.
- Compatibilità con diversi meccanismi di serializzazione: come descritto nella rispettiva sezione, nel rispetto del requisito 4.2.1.1.a).
- Compatibilità con diversi meccanismi di autenticazione, sia a livello di trasporto che di applicazione, alcuni dei quali già implementati e pronti all'uso, nel rispetto del requisito 4.2.1.1.b).
- il modello di esecuzione: concorde con quello dell'articolo, cioè RPC, e che ha, inoltre, consentito la predisposizione all'utente finale di un'interfaccia *trasparente*, come da obbiettivi di progetto.
- Il supporto a tecniche di load balancing, tracing e health checking, che agevolerà eventuali estensioni future in tali direzioni.

- Supporto attivo e documentazione

jackson-module-scala jackson-module-scala è un modulo add-on per la famosa libreria Jackson che fornisce il supporto per tipi di dato specifici di scala. Compatibile con scala 2.10, 2.11, 2.12 e scala 3, supporta la serializzazione in JSON di Case Class, delle collezioni dello Scala collection framework (Sequence, Maps, Tuple, Option, Enumeration), al netto di qualche problema con Option o Collection quando incapsulano tipi di dato primitivi in scala, affrontato direttamente durante il progetto e già reso noto agli autori [2]. Rispetto ad altre librerie già affermate nell'ecosistema di scala (fra cui circe, PlayJson, ScalaJack) è stata scelta per le buone prestazioni, il supporto attivo e per la (de)serializzazione automatica per mezzo della *reflection*, che, nella stragrande maggioranza, non richiede all'utente la definizione di (de)serializzatori personalizzati[2], nel rispetto dell'obbiettivo di progetto di trasparenza per l'utente. Ha, inoltre, contribuito alla sua adozione lo stato consolidato di Jackson, come del modulo stesso, e la compatibilità con diversi formati di serializzazione nel rispetto del requisito 4.2.1.1.a), fra cui binary JSON (CBOR e Smile), XML, YAML, CSV, che non tutte le concorrenti possono vantare.

io.jwt io.jwt è una libreria per la verifica e la creazione di JSON Web Token (JWT) per progetti android e basati su JVM. Realizzata in java ed open source, è conforme alle specifiche RFC JWT, JWS, JWE, JWK e JWA e fornisce un punto di accesso di alto livello alle utilità di creazione, verifica, firma (JWS) o crittografia (JWE) di JWT mediante gli algoritmi più diffusi e consolidati. Compatibile con altre tecnologie famose, come Jackson e Gson[3], e organizzata in maniera modulare, è un progetto tuttora attivo e ben documentato, che per questo è stato scelto, oltre che per la varietà di meccanismi supportati che aprono le porte ad estensioni future. Il suo impiego nel progetto riguarda la firma dei token nel modulo auth.

ScalaPB ScalaPB è un compilatore protocol buffer (protoc) disponibile come plugin sbt in grado di tradurre specifiche proto per il linguaggio scala. Progetto open source, oltre ad essere pienamente interoperabile con java, è compatibile con proto2 e proto3, consente di generare costrutti nativi in scala come case class e fornisce un supporto alla programmazione funzionale attraverso le *lense*. È compatibile con gRPC, realizzandone un nucleo di utilità per agevolarne la sua adozione in scala, e scala.js. Avendo scelto scala, la scelta di ScalaPB è arrivata di conseguenza, anche di fronte all'impiego futuro delle funzionalità indicate, ma non ad oggi ancora sfruttate. È stata utilizzata nel progetto per l'implementazione del modulo di autenticazione.

[2] `https://selectfrom.dev/jackson-module-scala-overview-and-new-scala-3-reflection-helper-b824be8221f7`
[3] `https://github.com/google/gson`

6.2 Serializzazione e deserializzazione

6.2.1 Passaggio logica di serializzazione con higher kinded type implicito

Per rendere indipendente il protocollo dalla specifica tecnologia, massimizzando il riuso di codice già scritto, la logica di serializzazione e deserializzazione è stata modellata attraverso *higher kinded type* (Transportable), e la sua iniezione nelle classi avviene tramite il meccanismo degli *impliciti*. Questo approccio, combinato con l'uso di Jackson, consente all'utente un accesso più trasparente alle API del protocollo (come da obbiettivo di progetto), perché permette nella stragrande maggioranza dei casi di non dover né definire né fornire esplicitamente la strategia di serializzazione. Avendo impiegato jackson, nelle implementazioni proposte Transportable è stato istanziato a JavaTypeable (ma alcuni equivalenti in altre librerie sono Format in PlayJson[4], l'unione di Encoder e Decoder in circe [5], TypeAdapter in ScalaJack[6] etc.); in ogni caso, JSON non è un vincolo, ma altri formati sono possibili. Per la implementazione della logica di realizzazione si sarebbe potuto valutare anche l'uso dell'astrazione della programmazione funzionale *type class*, realizzando un *polimorfismo ad-hoc*.

6.2.2 Serializzazione e deserializzazione attraverso Marshaller gRPC

Ad alto livello, qu-presentation definisce un modulo contenente la logica di serializzazione e deserializzazione dei messaggi scambiati fra client e server, quindi usata da entrambi.

Nello specifico, la sua struttura dipende dalla libreria gRPC impiegata e verte attorno alla astrazione di io.grpc.MethodDescriptor (MD), ossia la descrizione di un metodo remoto che processa le richieste dei client ritornando risposte (de) serializzate attraverso io.grpc.MethodDescriptor.Marshaller, un'astrazione per la serializzazione e deserializzazione dei messaggi. Marshaller di richieste e risposte devono essere registrati; dal client prima dell'invio, e dal service prima della sua esecuzione, rispettivamente.

Per promuovere l'adozione futura di tecnologie differenti, MethodDescriptorFactory modella una *factory* di MD riusabile in contesti diversi che astrae dalla specifica tecnologia di (de)serializzazione, isolata nel tipo parametrico Transportable ed impiegata, come descritto, anche dagli altri sotto-progetti. Allo scopo, si serve di MarshallerFactory, una *factory* di Marshaller dello stesso tipo a cui delegare la creazione dei Marshaller. La libreria ospita al momento un'implementazione in JSON tramite jackson (JacksonMarshallerFactory), ma per introdurre nuovi approcci o librerie (es. PlayJson) nel rispetto del req. 4.2.1.1.a), sarà sufficiente definire la MethodDescriptorFactory (es. PlayJsonMethodDescriptorFactory) e la MarshallerFactory (es. PlayJsonMarshallerFactory) corrispondenti.

Per evitare di rigenerare i MD inutilmente, è stata inoltre resa disponibile l'ottimizzazione CachingMethodDescriptorFactory, che sfrutta il pattern *Decorator* e *Flyweight*[16] per iniettare in una MethodDescriptorFactory esistente un meccanismo di caching dei MD, indipendente dalla tecnologia sottostante e riusabile a prescindere da essa.

[4] `https://github.com/playframework/play-json`
[5] `https://github.com/circe/circe`
[6] `https://github.com/gzoller/ScalaJack`

Si sottolinea che, sebbene la documentazione riporti come la loro generazione sia spesso a carico di generatori di codice automatici, la definizione manuale di queste astrazioni per quanto riguarda il core di Q/U si è resa necessaria per avere un controllo sui tipi maggiore (obbiettivo di progetto), non altrimenti ottenibile con ProtoBuf. Infatti, l'IDL proto [7] non ammette il concetto di tipo parametrico necessario a veicolare richieste e risposte di tipo diverso [8] (come invece consente il modulo descritto). Come descritto nel seguito, questo vincolo non riguarda il modulo auth, per la cui realizzazione si è potuto quindi sfruttare tutti i vantaggi della generazione automatica.

6.3 Specifica IDL proto per servizio di autenticazione

Anziché codificarla direttamente, gran parte del modulo auth è stata generata a partire dalla specifica di funzionamento proto3 mostrata, in parte, nel listato 1 e che declina le interfacce discusse nella sezione di progettazione. In questo modo, è stato possibile ridurre i tempi di sviluppo, e, grazie alla tecnologia di presentazione ProtoBuf e al rispettivo formato di rappresentazione dei dati, usufruire delle elevate prestazioni che offre, rendendo il modulo interoperabile e riusabile in altri contesti non strettamente vincolati a Q/U, così come su altre piattaforme. In particolare, per favorire l'interoperabilità con gli altri progetti scritti in scala, anche di fronte ad estensioni future, la generazione del codice è stata affidata al generatore ScalaPB, disponibile sotto forma di plugin sbt innescabile in corrispondenza della compilazione.

```
syntax = "proto3";

package qu;

service Auth {
  rpc register (User) returns (RegisterResponse) {}
  rpc authorize (Credentials) returns (Token) {}
}
```

Listing 1: Frammento del listato proto che definisce l'API di un servizio di autenticazione.

6.4 Verifica JWT con Interceptor gRPC

Come descritto alla sezione 2, Q/U richiede l'autenticazione client-server. gRPC include funzionalità di autenticazione a livello di applicazione, già implementate e pronte all'uso ma fondate su meccanismi a livello di trasporto più complessi che richiedono il possesso di una chiave o un certificato, come TLS. Per questa versione della libreria, si è preferito definire una propria policy basata su JWT, lasciando l'introduzione di strategie più avanzate come possibile, e configurabile, estensione futura (la cui fattibilità è stata già valutata).

[7] a differenza di una controparte su .NET

[8] `https://groups.google.com/g/protobuf/c/T8WQwV-Hv-0`

A tal proposito, in gRPC, l'iniezione di un meccanismo di autenticazione di livello di applicazione personalizzato si basa sull'aggiunta da parte della stub lato client di un header di autenticazione che accompagna ogni chiamata remota al servizio gRPC che necessita di validarne l'autenticità del mittente. La verifica lato server viene effettuata da io.grpc.ServerInterceptor che intercetta le richieste in arrivo e può ispezionarne il contenuto, terminandole con fallimento anzi tempo in caso di credenziali invalide o assenti.

6.5 Gestione delle eccezioni

La natura dei sistemi distribuiti è complessa; risulta quindi importante individuare e comunicare con chiarezza eventuali eventi avversi. In caso di eccezione durante l'esecuzione dei propri servizi, gRPC non trasmette ai client le informazioni sul problema né la causa scatenante mascherandoli sotto l'astrazione di io.grpc.StatusRuntimeException, ma fa uso di codice di errore trasmessi con le risposte; la logica di trasformazione di tali codici, tratti dal modello d'errore io.grpc.Status, in eccezioni è stata inglobata nei client. Per una loro più agevole individuazione tramite *pattern matching*, tutte le eccezioni definite sono state, inoltre, modellate in termini di *Case Class*.

6.6 Programmazione asincrona

L'articolo di Q/U presenta il protocollo basando il suo funzionamento su una semantica a RPC sincrona. Per allineare il protocollo, risalente al 2005, allo stato dell'arte per lo sviluppo di sistemi distribuiti e concorrenti, per il quale la *programmazione asincrona* costituisce il paradigma fondamentale e lo standard *de facto*, quella realizzata dalla libreria è una sua **rivisitazione asincrona**. Reso possibile anche grazie al supporto fornito da gRPC, fra i vantaggi di questo approccio non bloccante risultano la migliore responsività derivante dall'ottimizzazione delle risorse e il rialzo del livello di astrazione che permette, sia per il codice utente che quello interno, di rinunciare all'utilizzo tradizionale e diretto dei costrutti *thread*-related e alla maggiore parte dei problemi annessi (*race condition* etc.). La maggior efficienza di un approccio asincrono è dovuta alla prevalenza dell'overhead di comunicazione nell'esecuzione del protocollo, tipica di un contesto distribuito come quello verso cui Q/U è specificatamente rivolto.

In particolare, il supporto all'API asincrona si realizza in scala attraverso il costrutto scala.concurrent.Future, un oggetto che incapsula un valore che diverrà, prima o poi, disponibile. L'esecuzione delle *Future*, in scala, è a carico di ExecutionContext, un'astrazione attiva simile a (e ottenibile da) un Executor di java, di più alto livello dei thread e che ne nasconde in quantità configurabile. Una caratteristica della Future di scala è la sua natura *monadica* che permette di interagirvi attraverso un insieme definito di *combinatori*[12] riusabili. Il vantaggio che si ottiene dal punto di vista stilistico è la possibilità di sfruttare il costrutto *for-comprehension* per una maggiore leggibilità, come mostra la fig.11, che mostra la sottomissione al client di Q/U di alcune operazioni in sequenza.

Un'altra delle novità rispetto a java consiste nel passaggio di tale astrazione in modo *implicito*. Pertanto, per abilitare la logica asincrona, i metodi pubblici di client e service

ritornano *Future* contenenti il risultato corrispondente e si servono di un ExecutionContext implicito la cui presenza in scope è un requisito.

L'impiego di *Future* rappresenta anche la scelta più generale perché consente di riprodurre da essa una semantica bloccante utilizzando costrutti per attenderne, in modo sincrono, il completamento; in scala sono disponibili nel *companion object* Await; il loro uso, anche se generalmente sconsigliato, può essere talvolta necessario.

6.7 Elementi di programmazione funzionale

Seppur basato principalmente su un nucleo object-oriented, per affrontare la complessità di un sistema non banale come un protocollo di consenso, usufruendo dei vantaggi che offre, fra cui una più facile gestione della concorrenza, la maggior leggibilità, incremento del riuso, maggiore confidenza nel correttezza del codice prodotto, la progettazione e l'implementazione del progetto integra aspetti e tecniche tratti dalla *programmazione funzionale* e resi, in particolare, disponibili grazie al supporto di scala. Fra questi, i principali sono i seguenti.

Monadi Oltre a *Future*, scala predispone numerosi costrutti monadici con cui interagire attraverso una serie di combinatori. Per la gestione della possibile assenza di valori all'interno delle richieste e delle risposte trasferite da client e server è stata impiegata Option, anch'essa una monade, il cui principale vantaggio consiste nel "fattorizzare pattern comuni per la gestione dell'errore, attraverso funzioni *Higher-Order*, rinunciando all'usuale boilerplate annesso" [12] che accompagna il codice che deve far fronte a possibili valori null. Anche i generatori impiegati per il *property-based* testing sono a loro volta una monade e la loro composizione è risultata particolarmente agevole mediante *for comprehension*. È stato inoltre impiegato, per la gestione delle eccezioni del modello della demo, un ulteriore costrutto monadico, Try, ossia un contenitore che incapsula una computazione che può ritornare come risultato un valore o una eccezione, il cui uso può essere esteso anche ai moduli restanti come sviluppo futuro.

Ricorsione Piuttosto che costrutti imperativi come il **for** loop, il codice lato client del protocollo che definisce la logica di ritrasmissione delle richieste necessaria a fronteggiare guasti ai componenti del sistema, sfrutta la *ricorsione*, con semantica asincrona (attraverso valori di ritorno di tipo Future), in questo modo più leggibile, anche grazie al costrutto *for-comprehension*.

Folding La sequenzializzazione delle future relative alla terminazione asincrona di una collezione di astrazioni Shutdownable, come, ad esempio, QuorumPolicy, la cui terminazione necessita, a sua volta, della corretta terminazione dei canali di cui si compone, è stata realizzata attraverso *folding* della struttura dati che li contiene, disponibile come metodo nell'astrazione generale di Traversable.

Passaggio strategie Higher-Order Per rendere modulare il progetto predisponendolo al riuso e all'estensione futura, un'attenzione particolare durante il suo sviluppo è stata

rivolta alla *dependency injection* delle strategie con cui abilitare l'inserimento futuro di nuovi meccanismi. L'iniezione di Strategie nei builder relative a QuorumPolicy lato client e server, ad esempio, è stata realizzata attraverso passaggio *Higher-Order* delle *factory* corrispondenti, massimizzando così il riuso e la leggibilità del codice.

Classi immutabili e pattern matching L'utilizzo di classi immutabili, eventualmente con metodi, ma senza *side-effect*, è stato preferito, qualora possibile, per favorire la gestione della *thread-safety*, rimuovere la necessità di *copie difensive*, aumentare la leggibilità anche attraverso l'uso di *pattern matching* e migliorare l'incapsulamento. Ne sono un esempio le eccezioni, la modellazione dei comandi della demo, ImmutableStorage e i builder, per i quali è risultato molto utile impiegare i *default argument* predisposti da scala.

Memoizing e Laziness Per la risoluzione di alcuni problemi di inizializzazione e soprattutto, per migliorare le prestazioni evitando di ri-eseguire inutilmente, in fase di testing, da capo gli scenari di test, si è fatto uso della *lazy evaluation* attraverso la parola chiave *lazy* evitando di rivalutare più volte il valore di una variabile. L'adozione di descrizioni *lazy*, come la *monade IO*, che sostituiscano l'impiego di strutture *eager* come Future per la comunicazione asincrona fra client e service è, inoltre, un possibile sviluppo futuro. La tecnica di *memoization* [12] è stata inoltre applicata per evitare di ri-computare i descrittori dei metodi dei servizi, migliorando le prestazioni visto che verosimilmente un utente sottoporrà più di una volta la stessa operazione al client all'interno della stessa esecuzione.

6.8 Logging

Per agevolare il debugging del sistema distribuito, migliorando la gestione degli scenari di normale funzionamento o di guasto, sia client che server, soprattutto per quanto riguarda la loro comunicazione, in buona parte della base di codice si è fatto uso di logging.

In particolare, si è ritenuto utile implementare alcune funzionalità non già disponibili attraverso le API scelte (java.util.logging), fra cui il logging asincrono, che vi sono state iniettate attraverso il pattern *pimp-my-library*, sfruttando il meccanismo delle conversioni implicite fornito da scala e consentendo così di riusarle senza re-implementarle da zero.

7 Autovalutazione

Ancor più che attraverso la demo, la validazione del codice prodotto è avvenuta attraverso l'uso estensivo di *test automatizzati*. Il loro impiego non è stato limitato a:

- agevolare la rilevazione e quindi risoluzione tempestiva di bug e problemi,
- fornire un controllo di regressione di fronte a *refactoring* o modifiche,

ma anche volto a:

- agire come ulteriore documentazione, considerata la natura di libreria dell'artefatto e quindi la predisposizione all'utilizzo di altri programmatori.
- A guidare, ispirando, o rivisitando, il design, attraverso la visualizzazione del codice dal punto di vista dell'utente.

Si sono redatti sia *unit* che *system* test: oltre ad un sotto-progetto dedicato esclusivamente al *system testing*, dove viene testato il comportamento dell'intero sistema nel suo complesso, ciascun sotto-progetto contiene i test di unità relativi alle sue entità ed astrazioni principali e/o critiche in termini di complessità (di utilizzo o di funzionamento) e predisposizione all'errore.

7.1 Approccio

7.1.1 Behaviour-Driven Development

Sebbene non alla lettera, l'approccio impiegato nello sviluppo del codice si ispira, almeno nella sua componente stilistica, al *Behaviour-Driven Development* (BDD), un raffinamento di *Test-Driven Development* (TDD) nel quale le test suite costituiscono specifiche eseguibili, "ossia specifiche del comportamento del sistema che possono essere eseguite per verificare tale comportamento"[2] poiché l'enfasi del processo è posta sulla comunicazione con gli stakeholder. Ogni test è in realtà una *specifica* del funzionamento dell'oggetto testato.

7.1.2 Testing efficace

Nella stesura delle specifiche si è cercato di perseguire le qualità del software di produzione, cioè la leggibilità, ma anche indipendenza, ripetibilità, tempestività evitando, in particolare, la viscosità, considerata un ostacolo importante al *refactoring* del sistema, condotto continuativamente durante il progetto. Il design modulare della libreria ha consentito una facile *dependency injection* che ha agevolato, in tal senso, di molto il testing. L'aderenza ad una naming convention consistente e alle buone pratiche in termini di testing è stata agevolata dall'impiego di framework consolidati, che ha permesso anche di realizzare testing *property-based* e asincrono. Un contributo importante è arrivato dal *mocking*, con cui isolare le funzionalità, rendere i test deterministici, ridurre i tempi di esecuzione ed agevolare il *behaviour testing*.

7.1.3 Property-based testing

Divenuto famoso nell'ambito della programmazione funzionale con QuickCheck per Haskell, il *property-based* testing si basa sul disaccoppiamento della *specifica* del comportamento del programma da testare, espressa *dichiarativamente* tramite proprietà, rispetto alla *creazione* dei casi di test, delegata ad appositi generatori automatici. Come ricorda [12] fra i vantaggi di un approccio di questo tipo ci sono:

- La *minimizzazione dei casi di test*, grazie ai tentativi attuati dai framework di test di ridurre il campione di test che fallisce.

- *Generazione di casi di test esaustivi*: nel caso di domini limitati, è possibile valutare la validità di una proprietà per tutti i suoi valori, ottenendo quindi una sua *prova* anziché l'assenza di un'evidenza del suo contrario.

7.2 Tecnologie per il testing

L'ampio supporto di cui scala gode in termini di testing ha contribuito alla scelta di tale linguaggio per l'implementazione di Q/U, in vista della complessità intrinseca del protocollo. Si elencano nel seguito le tecnologie utilizzate e una panoramica delle specifiche.

ScalaTest Descritto dagli autori come il più flessibile e popolare nell'ecosistema scala, ScalaTest è uno fra i più completi framework per il testing in ambito scala. Organizzato in modo molto modulare per agevolarne l'estensione, la personalizzazione, la facilità d'utilizzo e la produttività, la caratteristica che più di tutte lo distingue dalle alternative è il supporto a diversi stili di testing, a partire dal mondo JUnit a quello del BDD, passando al *property-based* testing, sotto forma di *Domain Specific Language* (DSL) scala. Fra i suoi concetti basici ci sono quello di test (ogni astrazione con un nome in grado di avviarsi e concludersi con successo o fallimento) e suite (una collezione eseguibile composta da zero o più test). Oltre ad un variegato insieme di asserzioni e *matcher*, ha recentemente introdotto, accanto all'insieme di stili sincroni, le rispettive controparti per facilitare il testing asincrono, predisponendo un'API di alto livello che ne favorisce il determinismo, le prestazioni e la leggibilità, motivi, oltre a quelli elencati, della sua scelta.

ScalaCheck Ispirata dalla libreria QuickCheck del linguaggio funzionale Haskell, ScalaCheck è una libreria per il *property-based* testing di programmi scala e java scritta in scala e compatibile con sbt. Pienamente integrabile con i principali framework di test per scala, ScalaTest e Specs2, ma comunque indipendente, fra i suoi utilizzatori ci sono importanti progetti, come il compilatore Scala e il framework ad attori Akka. Le sue API predispongono generatori già pronti all'uso o la possibilità di configurarne di nuovi, personalizzati. Nonostante attestati problemi nella generazione di dati di test con *shrinking*, che talvolta non rispetta i vincoli prestabiliti, e in cui, peraltro, ci si è imbattuti durante questo progetto e che necessiteranno di una profonda revisione del framework per essere risolti definitivamente [9], la libreria è potente ed espressiva.

ScalaMock ScalaMock è un framework scala nativo e open source per il *mocking*. Disponibile per scala 2.11, 2.12, 2.13, consente di realizzazione stub, mock in scala e scala.js attraverso un'API type safe integrabile con ScalaTest e Specs2. Essendo implementato in scala, fornisce quasi pieno supporto per le funzionalità del linguaggio, ma anche per java, consentendo il *mocking* di classi, case classes, trait, funzioni, operatori, metodi, anche overloaded, con tipi parametrici, in modo type safe. L'accesso alle API per il *mocking* è disponibile attraverso due stili, detti *expectation-first* e *record-than-verify*, con diversi livelli di espressività, in base alle esigenze.

[9] `https://gist.github.com/davidallsopp/f65d73fea8b5e5165fc3`

7.3 Specifiche di unità

7.3.1 Specifiche modulo auth

LocalAuthenticatorSpec Verifica, e specifica, il funzionamento di un autenticatore locale, cioè della *business* logic relativa a registrazione, autorizzazione e generazione di token JWT indipendente dalla tecnologia di serializzazione attraverso cui le richieste vi giungono. Nello specifico, si valuta la corretta gestione delle situazioni di errore (conflitti, credenziali errate).

AuthClient	AuthClient.scala	93	9	62	48	77.42 %
Constants	Constants.scala	22	1	6	6	100.00 %
FutureUtilities	FutureUtilities.scala	34	2	2	2	100.00 %
LocalAuthenticator	LocalAuthenticator.scala	48	3	84	66	78.57 %
AuthServer	AuthServer.scala	42	4	8	8	100.00 %
AuthService	AuthService.scala	41	3	18	16	88.89 %

Figure 26: Report sulla code *coverage*, indicata in percentuale sulla destra, generato dal plugin sbt-coverage per alcune delle classi principali del sotto-progetto auth.

7.3.2 Specifiche modulo qu-core

QuModelSpec È una specifica corposa (di cui un frammento mostrato al listato 2) del funzionamento del nucleo più astratto del protocollo, delle strutture dati e degli algoritmi su cui Q/U basa il suo funzionamento ed indipendenti dalle tecnologie di serializzazione, di autenticazione e di comunicazione. In particolare, contiene la specifica di:

```
describe("A ConcreteLogicalTimestamp") {

  describe("when initialized with a logical time") {
    it("should be classified as previous to a ConcreteLogicalTimestamp
        with a greater logical time") {
      forAll(arbitraryLt) { aLt =>
        forAll(logicalTimestampWithGreaterTimeThan(aLt.time)) { anotherLt
            =>
          anotherLt should be > aLt
        }
      }
    }
    ...
  }
```

Listing 2: Specifica relativa alla verifica delle relazioni di ordinamento per l'astrazione di logical timestamp con *property-based* testing. Le chiamate a forAll incapsulano l'uso di generatori personalizzati.

- Timestamp (ConcreteLogicalTimestamp), in particolare le relazioni di ordinamento e di (dis)uguaglianza precisate dagli autori nell'articolo (di cui si mostra un dettaglio nel listato 2 e che sono alla base del modello temporale di Q/U). Per ottenerne i vantaggi descritti, la specifica è stata disaccoppiata dalla logica di generazione dei dati di test tramite *property-based* testing. Allo scopo, è stato definito un modulo di generatori personalizzati di timestamp per validare i casi più critici.
- Candidati: relativamente alla configurazione dei candidati a partire dagli OHS corrispondenti a tutte i tipi di operazione previsti dal protocollo (disponibili nella *fixture* corrispondente OHSFixture)
- RH: relativamente alle funzionalità di verifica della presenza di un candidato (contains) e di rilevazione dell'ultimo logical timestamp al suo interno, per la RH vuota (emptyRH).
- OHS: che esemplifica, attraverso un OHS corrispondente ad ognuno dei tipi di operazioni previsti dal protocollo, oltre che all'OHS vuoto, le funzionalità di:
  - Classificazione (classify)
  - Determinazione dell'ultimo candidato (latestCandidate)
  - Determinazione dell'ultimo timestamp contenuto (latestTime)
- Rappresentazioni; nello specifico, l'uguaglianza fra le rappresentazioni di OHS o operazioni identiche.

Class	Source file	Lines	Methods	Statements	Invoked	Coverage	
AbstractAbstractQuModel	QuModel.scala	230	10	127	126		99.21 %
ConcreteQuModel.ConcreteLogicalTimestamp	QuModel.scala	258	1	1	1		100.00 %
CryptoMd5Authenticator	Crypto.scala	29	4	12	12		100.00 %
Hashing	Hashing.scala	20	2	7	7		100.00 %
Operations.AbstractOperation	Operations.scala	16	1	3	3		100.00 %
Operations.UpdateReturningUnit	Operations.scala	38	1	1	1		100.00 %

Figure 27: Report sulla code *coverage* generato dal plugin sbt-coverage per alcune delle classi principali del sotto-progetto qu-core.

7.3.3 Specifiche modulo qu-presentation

PresentationSpec Verifica il corretto funzionamento delle classi la cui serializzazione o deserializzazione si è rivelata problematica durante lo sviluppo ed ha richiesto contromisure specifiche.

7.3.4 Specifiche modulo qu-client

QuClientSpec Presenta la specifica di un client Q/U già autenticato (QuClientImpl) illustrandone sia il funzionamento di base, sia quello risultante dalle ottimizzazioni descritte alla sezione 2, e trascurando gli aspetti di interazione con le repliche, trattati

invece in BroadcastClientQuorumPolicySpec. Fra gli altri, si verificano gli scenari di esecuzione ottimistica delle interrogazioni (la cui specifica è mostrata nel listato 3), di riparazione, di sottomissione di aggiornamenti e interrogazioni, soffermandosi sul corretto numero di ritrasmissioni in relazione alle risposte simulate, sulla correttezza dei messaggi scambiati alle repliche e dei valori ritornati all'utente finale. Per verificare l'ordinamento totale, comunque presente nonostante l'esecuzione asincrona, tramite il metodo inSequence, non altrimenti disponibile con lo stile AsyncFunSpec, si è impiegato uno stile sincrono in associazione alle utilità predisposte da ScalaFutures, effettuando così un *behaviour testing* più esaustivo. Per agevolare il testing di una classe così ricca di interazione si è inoltre fatto ricorso alle astrazioni di stub e mock predisposte da scalamock, mostrate nel listato 3, impiegando uno stile expectiation-first, poiché più potente ed espressivo.

```
val queryOp = GetObj[Int]()

describe("when requesting an query operation a response with order < q but
    classified as a " +
   "method (query executed optimistically)") {

   it(" should return the correct answer in a single round of communication") {
     val expectedResponse = initialValue + 1

     queryQuorum.expects(Some(queryOp), emptyOhs(servers), *,
         *).noMoreThanOnce()
       .returning(Future.successful((Some(expectedResponse),
         thresholds.r,
         ohsWithMethodFor(serversKeys))))

     (mockedBackOffPolicy.backOff()(_: ExecutionContext)).expects(*).never()

     whenReady(client.submit[Int](queryOp)) {
       _ should be(expectedResponse)
     }
   }
}
```

Listing 3: Specifica ScalaTest (con stile AnyFunSpec) del comportamento del client in virtù dell'ottimizzazione di esecuzione ottimistica delle interrogazioni. È possibile osservare il *mocking* di quorum e backoff policy con stile expectation-first, con il quale le aspettative vengono registrate prima dell'esecuzione, che permette di verificarne con flessibilità la corretta invocazione (numero di chiamate e argomenti). Si nota, inoltre, l'asserzione sulla risposta ritornata all'utente al completamento della computazione, ottenuta tramite whenReady..

AuthenticatingClient	AuthenticatingClient.scala	44	5	21	20	95.24 %
JacksonQuClientBuilderFactory	JacksonQuClientBuilderFactory.scala	18	1	4	4	100.00 %
OperationOutputNotRegisteredException	OperationOutputNotRegisteredException.scala	16	1	1	0	0.00 %
QuClient	QuClient.scala	30	1	1	1	100.00 %
QuClientBuilder	QuClientBuilder.scala	65	6	38	30	78.95 %
QuClientBuilderFactory	QuClientBuilderFactory.scala	28	1	3	3	100.00 %
QuClientImpl	QuClientImpl.scala	113	8	22	21	95.45 %

Figure 28: Report sulla code *coverage* generato dal plugin sbt-coverage per alcune delle classi principali del sotto-progetto qu-client.

7.3.5 Specifiche modulo qu-service

QuServiceSpec Mostra la specifica di funzionamento di una replica Q/U (QuServer), iniettata nel test attraverso la specifica *fixture* responsabile di avviarne una nuova istanza (e terminarla) prima di (e dopo a) ognuno degli unit test. Come mostra la *coverage* in fig. 29, la specifica copre gran parte delle funzionalità del servizio: dal controllo sull'autenticazione dei client, al corretto processamento delle richieste di *object-sync* provenienti dalle altre repliche, sia in caso in cui l'oggetto richiesto sia disponibile, sia in sua assenza, oltre alla corretta gestione delle richieste contenenti OHS per tutte le tipologie di operazioni (method, inline barrier etc.), al controllo di integrità delle RH ed eventuale *culling* delle RH corrotte. In particolare, si valuta il contenuto delle risposte fornite ai client (codice di stato, valore di ritorno dell'operazione e RH autenticata), la corretta modifica della RH locale e degli autenticatori corrispondenti, oltre che l'interazione con le altre repliche (iniettate come mock ispezionabili per consentire *behaviour testing*, tramite stile expectation-first) esemplificando sia il funzionamento di base di Q/U sia le estensioni (replica pruning, inline repair ed esecuzione ottimistica). Diversamente dalle classi client, si è impiegato uno stile di testing asincrono con cui ScalaTest si prende in carico di eseguire e valutare le asserzioni sulle computazioni, una volta completate.

GrpcQuServiceImpl	GrpcQuServiceImpl.scala	62	4	14	12	85.71 %
JacksonServerBuilderFactory	JacksonServerBuilderFactory.scala	21	1	1	1	100.00 %
JwtAuthorizationServerInterceptor	JwtAuthorizationServerInterceptor.scala	40	2	31	26	83.87 %
LocalQuServerCluster	LocalQuServerCluster.scala	109	3	20	20	100.00 %
LocalQuServerClusterImpl	LocalQuServerCluster.scala	68	7	25	19	76.00 %
QuServer	QuServer.scala	25	1	1	1	100.00 %
QuServerBuilder	QuServerBuilder.scala	90	5	20	18	90.00 %
QuServerImpl	QuServer.scala	67	4	25	24	96.00 %
QuServiceImpl	QuServiceImpl.scala	206	8	223	199	89.24 %

Figure 29: Report sulla code *coverage* generato dal plugin sbt-coverage per alcune delle classi principali del sotto-progetto qu-core.

7.3.6 Specifiche modulo qu-storage

ImmutableStorageSpec Rappresenta la specifica di funzionamento della logica di archiviazione impiegata dai servizi Q/U per depositare le versioni dell'oggetto da replicare e le risposte alle operazioni ricevute. Oltre a mostrarne la natura immutabile, esemplifica il corretto uso dei generici per interagire con il contenitore *eterogeneo* e, in particolare, l'assenza di valori recuperabili quando il contenitore è vuoto, la presenza di un valore precedentemente inserito e l'assenza di valori non precedentemente introdotti al suo interno.

Figure 30: Report sulla code *coverage* generato dal plugin sbt-coverage per alcune delle classi principali del sotto-progetto qu-storage.

7.4 Specifiche di sistema

RemoteAuthenticatorSpec Mostra e verifica il funzionamento di un autenticatore remoto, cioè della corretta interazione fra un client di autenticazione e il servizio in attesa di richieste. Le aspettative sono le stesse dell'autenticatore locale, che infatti estende a sua volta la stessa specifica, contenente le stesse asserzioni. Le due specifiche si distinguono per la logica di avvio e terminazione, isolata nelle rispettive *fixture* in accordo con le buone pratiche di ScalaTest e dove, nel secondo caso, si provvede ad avviare una nuova istanza di server ed un client prima di (e terminarle appena dopo) ogni test della suite.

NonFailingServersInteractionSpec Costituisce la specifica più di alto livello dell'intero sistema e un test di accettazione di fronte all'utente finale della libreria. Mostra infatti l'interazione prevista con la libreria ed ha valore dimostrativo per tutti gli scenari di utilizzo, elencati alla sezione 3, esemplificando l'avvio del server di autenticazione, la scelta delle soglie di quorum, l'autenticazione, l'autorizzazione e la configurazione del client (ciascuna nelle apposite *fixture* iniettate con il pattern suggerito da ScalaTest e rigenerate per ogni esecuzione creando un ambiente di esecuzione vergine), la sottomissione di aggiornamenti e interrogazioni, singole o multiple e la corretta terminazione di tutti i processi. Nel listato 4 si mostra la specifica relativa alla sottomissione di una serie di aggiornamenti seguiti da un'interrogazione.

```
describe("when multiple updates are issued followed by a query") {
  it("should return to client the correct answer") {
      val incrementsCount = 3
      val operations = List.fill(incrementsCount)(Increment())
      for {
        authenticatedQuClient <- quClientAsFuture
        _ <- seqFutures(operations)(authenticatedQuClient.submit(_))
        queryResult <- authenticatedQuClient.submit(GetObj())
      } yield queryResult should be(InitialObject + incrementsCount)
```

```
    }
}
```

Listing 4: Specifica di alto livello del funzionamento della libreria per la sottomissione di alcune operazioni di aggiornamento ed interrogazione in sequenza.

In particolare, NonFailingServersInteractionSpec valida il comportamento di client e server in scenari di esecuzione senza guasti.

OneFailingServerInteractionsSpec Mostra la specifica della libreria, condividendola con NonFailingServersInteractionSpec mediante l'estensione dello stesso trait, ma in uno scenario di esecuzione segnato da guasto di tipo *crash* ad una delle repliche, dispiegandone il minimo numero necessario a tollerarlo (4).

8 Istruzioni per il deployment

Per agevolare la fruizione, le modalità di dispiegamento per libreria e applicazione demo sono distinte.

8.1 Libreria

8.1.1 Prerequisiti

I requisiti software per poter riusare la libreria integrandola nella propria base di codice sono i seguenti:

- scala 2.13.8
- java 11
- sbt 1.6.2
- git

8.1.2 Utenti sbt

Per non renderla accessibile a terzi [10], la libreria non è stata pubblicata su un repository remoto accessibile pubblicamente. Pertanto, i passi di configurazione per riusarla in un progetto sbt sono i seguenti:

1. Clonare il progetto nella cartella di interesse:

   ```
   $ git clone https://gitlab.com/pika-lab/courses/ds
   /projects/ds-project-giulianini-ay1920
   ```

2. Spostarsi dentro alla radice del progetto:

[10] per allinearsi alle politiche di accesso adottate anche per il repository già configurato.

```
$ cd ds-project-giulianini-ay1920
```

3. Pubblicare localmente la libreria:

```
$ sbt publishLocal
```

4. Aggiungere al proprio build.sbt le dipendenze di interesse fra le seguenti:

```
libraryDependencies ++= Seq(
"org.unibo" %% "ds-project-giulianini-ay1920" % "1.0.0",
"org.unibo" %% "quClient" % "1.0.0"
"org.unibo" %% "quService" % "1.0.0"
...
)
```

8.1.3 scaladoc

Per generare la scaladoc:

1. Spostarsi dentro alla radice del progetto:

```
$ cd ds-project-giulianini-ay1920
```

2. Eseguire il comando:

```
$ sbt doc
```

La documentazione HTML relativa ad ogni sotto-progetto verrà generata nella rispettiva cartella target/, il cui percorso è anche visibile dall'output nella console al termine dell'esecuzione del comando.

8.2 Demo

Per l'esecuzione della applicazione che esemplifica il funzionamento del protocollo è stato reso disponibile nella radice del progetto un Dockerfile.

8.2.1 Prerequisiti

- git
- docker

8.2.2 Passi

Pertanto, per il lancio dell'applicazione sono necessari i seguenti passi:

1. Clonare il progetto nella cartella di interesse:

```
$ git clone https://gitlab.com/pika-lab/courses/ds
/projects/ds-project-giulianini-ay1920
```

2. Spostarsi dentro alla radice del progetto:

   ```
   $ cd ds-project-giulianini-ay1920
   ```

3. Dalla radice del progetto, costruire l'immagine dell'applicazione con il comando:

   ```
   $ docker build -t  qu-cli-demo .
   ```

4. Eseguire l'applicazione con:

   ```
   $ docker run -it --name <container-name> qu-cli-demo
   ```

   dove <container-name>è un *placeholder* per il nome scelto.

5. Al termine dell'esecuzione stoppare il container impiegando il nome precedentemente assegnatogli:

   ```
   $ docker rm <container-name>
   ```

In alternativa, sostituendo nei prerequisiti docker con sbt, è possibile:

1. scaricare il repository.

   ```
   $ git clone https://gitlab.com/pika-lab/courses/ds
   /projects/ds-project-giulianini-ay1920
   ```

2. collocarsi nella radice del progetto:

   ```
   $ cd ds-project-giulianini-ay1920
   ```

3. per poi eseguire l'applicazione digitando:

   ```
   $ sbt run
   ```

9 Esempi d'uso

9.1 Libreria

Si propone ora un esempio d'uso della libreria negli scenari individuati alla sezione 3 per la definizione di un servizio fault-tolerant e fault-scalable che mette a disposizione un contatore con cui è possibile interagire attraverso i seguenti metodi:

- Incremento: per incrementarne il valore di uno
- Decremento: per decrementarne il valore di uno

- Reset: per riportarlo al valore iniziale zero
- Value: per recuperare il valore corrente

Il servizio intende tollerare fino ad un massimo di due repliche con guasto di cui uno possibilmente di tipo bizantino, al contempo impiegando il minor numero di repliche necessarie a questo scopo (sei).

Si presentano tre approcci leggermente diversi con cui è possibile interagire con le API del protocollo a seconda delle esigenze.

9.1.1 Utilizzo API gRPC-agnostic

L'approccio "vanilla" consiste nell'utilizzo API della libreria di livello di astrazione intermedio, cioè quelle più espressive senza presupporre la conoscenza di, né la dipendenza diretta da, gRPC, che rimane un dettaglio implementativo. Consente di raggiungere un grado di personalizzazione superiore rispetto al livello di astrazione descritto alla sezione successiva ma inferiore rispetto all'impiego gRPC-aware descritto nel seguito.

Definizione operazioni Come già ricordato, Q/U permette la specificazione di un servizio mediante l'approccio SMR. Pertanto, dopo aver importato qu.model.Quorum SystemThresholdQuModel._, il primo passo per la messa in funzione di un servizio distribuito resiliente ai guasti è quello di definire la sua interfaccia, ossia le operazioni che esporrà all'utente, modellate come case class. Come mostra il listato seguente, è possibile riusare allo scopo le implementazioni riusabili che coprono alcuni dei casi d'uso più comuni.

```
import qu.model.QuorumSystemThresholdQuModel._

object Value extends QueryReturningObject[Int]

case class Increment() extends UpdateReturningUnit[Int] {
  override def updateObject(obj: Int): Int = obj + 1
}

case class Decrement() extends UpdateReturningUnit[Int] {
  override def updateObject(obj: Int): Int = obj - 1
}

case class Reset() extends UpdateReturningUnit[Int] {
  override def updateObject(obj: Int): Int = 0
}
```

Listing 5: Esempio di utilizzo delle API della libreria per la definizione delle operazioni della RSM che descrive il servizio.

Scelta delle soglie di quorum Le soglie di quorum sono reificate nella classe QuorumSystemThresholds, usata per la configurazione sia di client che delle repliche. Come mostra il listato 6, se ci si intende accontentare delle soglie minime per lo scenario di interesse (due guasti (t), di cui uno bizantino (b) nel caso in esame), è sufficiente specificare le soglie relative ai soli guasti (t e b). In alternativa, è possibile ridondare i server specificando anche i rimanenti parametri (q, n, r), soprattutto per usufruire della proprietà di *throughput scalability* accennata alla sezione 1. A tal proposito, è incluso un controllo dell'input in caso di soglie non valide.

```
import qu.model.QuorumSystemThresholds

val thresholds = QuorumSystemThresholds(t = 2, b = 1)
```

Listing 6: Esempio di utilizzo delle API della libreria per la scelta delle soglie di quorum.

Configurazione e avvio server di autenticazione Il protocollo basa il suo funzionamento sull'autenticazione dei client che interagiscono con le repliche. Per la creazione del server di autenticazione (AuthServer) è possibile sfruttare il *factory method* accessibile dal suo *companion object* fornendovi la porta su cui rimanere in ascolto e rendendo disponibile in scope l'ExecutionContext a cui verranno delegate le richieste di registrazione ed autenticazione. Si procede poi al suo avvio tramite il metodo start.

```
val authServerSocketAddr = SocketAddress("localhost", 1006)

import scala.concurrent.ExecutionContext.Implicits.global
import qu.auth.server.AuthServer

val authServer = new AuthServer(authServerSocketAddr.port)
authServer.start()
```

Listing 7: Dettaglio d'uso delle API della libreria per l'avvio di un server di autenticazione.

Configurazione e avvio repliche Per gestire le richieste è necessario definire e istanziare le repliche in numero coerente con la soglia dei guasti da tollerare prescelta. La configurazione di ciascun server avviene per mezzo di QuServerBuilder, ottenibile con il *factory method* corrispondente. Oltre alla porta su cui rimanere in ascolto, è necessario specificare le soglie di tolleranza, l'oggetto (ossia lo stato iniziale della RSM) e la presenza di un ExecutionContext che eseguirà le richieste dei client. Come mostrato nel listato, si procede, quindi, all'aggiunta delle informazioni riguardanti ciascuno degli altri server del cluster a cui la replica partecipa (ip, porta e chiave privata condivisa con il server in questione per la generazione di HMAC per la verifica di integrità delle RH), e del tipo del valore di ritorno delle operazioni che si intende sottomettere: tale settaggio richiede la presenza in scope degli impliciti che definiscono la logica di serializzazione di richieste e risposte, ma la libreria Jackson consente all'utente di non preoccuparsene sfruttando

in automatico le definizione implicite di jackson-module-scala. Se la registrazione delle operazioni non viene effettuata, il client incorre in una OperationOutputNotRegisteredException contestuale alle sottomissioni al server, che compromette il funzionamento.

```
import qu.service.QuServerBuilder
import qu.service.AbstractQuService.ServerInfo

val quReplica1Info = ServerInfo(ip = "localhost", port = 1001, "ks1s1")
val quReplica2Info = ServerInfo(ip = "localhost", port = 1002, "ks1s2")
...
val quReplica6Info = ServerInfo(ip = "localhost", port = 1006, "ks1s6")

val quServer = QuServerBuilder(quReplica1Info, thresholds, 0)
  .addServer(quReplica2Info)
   ...
  .addServer(quReplica6Info)
  .addOperationOutput[Int]() //for Value
  .addOperationOutput[Unit]() //for Inc, Dec, Reset
  .build

quServer.start()
```

Listing 8: Dettaglio d'uso delle API della libreria per la configurazione e l'avvio di una replica.

Autenticazione client Per interagire con le repliche, ciascun client necessita di essere registrato e quindi autenticato. Allo scopo, la libreria suddivide le API client di autenticazione da quelle di sottomissione delle operazioni fornendo le prime attraverso la classe AuthenticatingClient, con la quale registrarsi o autenticarsi previo passaggio al *factory method* corrispondente di indirizzo ip, porta, del server di autenticazione, username e password e presenza dell'ExecutionContext. Come mostrato nel listato seguente, i valori di ritorno modellati come Future abilitano la loro composizione monadica all'interno delle *for comprehension*. In particolare, l'operazione di autenticazione fornisce un QuClientBuilder con cui configurare il client vero e proprio.

```
import qu.client.AuthenticatingClient

val quServer1SocketAddr: SocketAddress = SocketAddress(ip = "localhost", port
    = 1001)
val quServer2SocketAddr: SocketAddress = SocketAddress(ip = "localhost", port
    = 1002)
//...
val quServer6SocketAddr: SocketAddress = SocketAddress(ip = "localhost", port
    = 1006)

val authClient = AuthenticatingClient[Int](
  authServerSocketAddr.ip,
```

```
  authServerSocketAddr.port,
  "username",
  "password")
val authenticatedQuClientBuilder = for {
  _ <- authClient.register()
  builder <- authClient.authorize()
} yield builder
```

Listing 9: Dettaglio d'uso delle API della libreria per la registrazione ed autenticazione di un client Q/U.

Configurazione client La configurazione di un client avviene fornendo al builder le informazioni sulle repliche (ip, porta) e le soglie di tolleranza, che devono coincidere con quelle utilizzate da ognuna di esse. Nel caso in cui le informazioni sui server siano state accumulate in una collezione, è anche possibile la loro registrazione *one-shot* con addServers. Il client con cui interagire con le repliche si ottiene invocando il metodo build che lo restituisce a valle della validazione degli argomenti.

```
val authenticatedQuClient =
  for {
    builder <- authenticatedQuClientBuilder
  }
  yield builder.addServer(quServer1SocketAddr)
    ...
    .addServer(quServer6SocketAddr)
    .withThresholds(thresholds).build
```

Listing 10: Dettaglio d'uso delle API della libreria per la configurazione di un client Q/U.

Sottomissione operazioni La sottomissione delle operazioni, siano di aggiornamento o di interrogazione, al server avviene tramite submit, la cui esecuzione necessita della presenza in scope di un ExecutionContext e degli impliciti (Transportable) contenenti la logica di (de)serializzazione delle operazioni richieste dal protocollo; grazie a Jackson, come per il server, non serve che questi ultimi siano specificati espressamente. Allo stesso modo delle API di AuthenticatingClient, il valore di ritorno Future rende possibile la concatenazione monadica delle invocazioni delle operazioni all'interno di una *for comprehension*, che ha una semantica di sequenzializzazione delle stesse: poiché non bloccante, più chiara e leggibile, si suggerisce questa modalità.

```
for {
  authenticatedQuClient <- authenticatedQuClient
  _ <- authenticatedQuClient.submit[Unit](Increment())
  _ <- authenticatedQuClient.submit[Unit](Reset())
  _ <- authenticatedQuClient.submit[Unit](Increment())
  _ <- authenticatedQuClient.submit[Unit](Increment())
  _ <- authenticatedQuClient.submit[Unit](Decrement())
```

```
  value <- authenticatedQuClient.submit[Int](Value)
} yield println("distributed counter value is:" + value)
```

Listing 11: Dettaglio d'uso delle API client del protocollo per la sottomissione di operazioni con semantica asincrona.

In alternativa, come mostrato nel listato seguente, resta possibile un utilizzo sincrono della API attraverso le utility scala di Await.

```
Await.ready(authenticatedSyncQuClient.submit(Increment()), atMost = maxWait)
Await.ready(authenticatedSyncQuClient.submit(Reset()), atMost = maxWait)
Await.ready(authenticatedSyncQuClient.submit(Increment()), atMost = maxWait)
Await.ready(authenticatedSyncQuClient.submit(Decrement()), atMost = maxWait)
Await.ready(authenticatedSyncQuClient.submit(Reset()), atMost = maxWait)
Await.ready(authenticatedSyncQuClient.submit[Int](Value), atMost = maxWait)
```

Listing 12: Dettaglio d'uso delle API client del protocollo per la sottomissione di operazioni con semantica sincrona.

Terminazione repliche Prima di chiudere l'applicazione, si raccomanda di terminare l'esecuzione delle repliche e del server di autenticazione attraverso il metodo shutdown attendendo il completamento della Future corrispondente, come in listato 13.

```
quServer.shutdown()
```

Listing 13: Dettaglio d'uso delle API per la terminazione asincrona di una replica.

Terminazione client La terminazione è richiesta anche per la classe QuClient e AuthClient, come si mostra al listato seguente:

```
authClient.shutdown()
authenticatedQuClient.map(_.shutdown())
```

Listing 14: Dettaglio d'uso delle API del protocollo per la terminazione di un client Q/U.

9.1.2 Strutture dati già pronte all'uso

Come già descritto, il progetto rende disponibili, nei package qu.client.datastructure e in qu.service.datastructure, delle utilità, lato client e lato service, per la costruzione di alcune strutture dati distribuite e BFT realizzate sopra a Q/U. Il supporto a queste funzionalità riusabili di più alto livello è tuttavia ancora limitato e per questo è stato indicato come possibile sviluppo futuro in 11.1.

Configurazione repliche Rispetto all'uso gRPC-unaware, per la configurazione delle repliche è sufficiente utilizzare la *factory* del builder corrispondente alla struttura di

interesse: come raffigurato al listato 15, l'inserimento delle soglie e degli altri server del cluster rimane invariato, ma non è più necessario registrare le operazioni esposte dal servizio poiché effettuate in automatico dalla libreria.

```
import qu.service.datastructures.RemoteCounterServer

RemoteCounterServer.builder(quReplica1Info.ip, quReplica1Info.port,
    quReplica1Info.keySharedWithMe, thresholds)
    .addServer(quReplica2Info)
    ...
    .addServer(quReplica6Info)
    .build()
```

Listing 15: Dettaglio d'uso delle utilità di più alto livello del protocollo per la configurazione di una replica per un contatore distribuito.

Configurazione client L'interfaccia della struttura dati risulta codificata nei metodi di una classe client che incapsula tutti i dettagli necessari all'interazione con il protocollo (KeyValueStoreClient nell'esempio in esame). Come mostra il listato 16, per la sua creazione si sfrutta un *factory method* a cui fornire lo username e la password del client la necessità di registrarlo (se alla prima interazione con il cluster), le informazioni (ip e porta) sulle repliche del cluster e sul server di autenticazione e le soglie di guasto.

```
import qu.client.datastructures.DistributedCounter

val quServersInfo = Set(quServer1SocketAddr, quServer2SocketAddr /*, ...*/ ,
    quServer6SocketAddr)
val distributedCounter = DistributedCounter("username",
    "password", authServerSocketAddr.ip, authServerSocketAddr.port,
        quServersInfo, thresholds)
```

Listing 16: Dettaglio d'uso delle API di alto livello della libreria per la configurazione di un client Q/U per un contatore distribuito.

Sottomissione operazioni La sottomissione avviene invocando esplicitamente i metodi che la codificano, disponibili in una versione asincrona (che ritorna una future del tipo del risultato desiderato), o sincrona. Il *factory method* consente di definire il (predefinito a 100 millisecondi).

```
for {
    _ <- distributedCounter.incrementAsync()
    _ <- distributedCounter.resetAsync()
    _ <- distributedCounter.incrementAsync()
    _ <- distributedCounter.incrementAsync()
    _ <- distributedCounter.decrementAsync()
    value <- distributedCounter.valueAsync
```

```
} yield println("distributed counter value is:" + value)
```

Listing 17: Dettaglio d'uso delle API di alto livello della libreria, equivalenti a quelle del listato 11, per la sottomissione delle operazioni tramite un client Q/U.

Le rimanenti funzionalità di terminazione del client e di avvio e terminazione delle repliche rimangono invariate.

9.1.3 Utilizzo gRPC-aware lato server

Durante lo sviluppo è emersa la possibile praticità data dall'uso contestuale delle API del protocollo e delle API di gRPC senza un livello di astrazione intermedio, per favorire l'integrazione della libreria con una base di codice gRPC già esistente o per sfruttare il *know-how* già acquisito su gRPC da parte dell'utente sviluppatore. Si tratta dell'utilizzo di più basso livello della libreria, ma con un più alto grado di personalizzazione, oltre che il punto di partenza per alcune possibili estensioni descritte alla sezione 11.1.

Configurazione repliche Per poter integrare le funzionalità di una replica Q/U in un server gRPC (io.grpc.Server) esterno, la libreria fornisce un builder (QuServiceBuilder) con cui istanziare un servizio che implementa io.grpc.BindableService disaccoppiando, quindi, la logica del protocollo da quella di configurazione ed avvio dei server, che diventa maggiormente personalizzabile dall'utente sulla base delle variegate funzionalità predisposte da gRPC.

Oltre a ip, porta, chiave privata della replica che si sta configurando, all'oggetto da replicare, alle soglie di guasto e allo storage per la persistenza delle informazioni, serve indicare:

- la MethodDescriptorFactory responsabile della (de)serializzazione dei messaggi client-server e server-server in entrata e delle corrispondenti risposte in uscita, compatibile con quella impiegata lato client (per dialogare con le JacksonBroadcastClientQuorumPolicy impiegata dal QuClient generato dal QuClientBuilder predefinito, si può usare una JacksonMethodDescriptorFactory, eventualmente con l'ottimizzazione CachingMethodDescriptorFactory), la quale deve basarsi su una tecnologia di serializzazione (es. JSON) condivisa da tutti i server del cluster e da tutti i client.

- la ServerQuorumPolicyFactory, ossia la *factory* Higher-Order di ServerQuorumPolicy impiegata per l'interazione con gli altri server (nello specifiche per le richieste in uscita e corrispondenti risposte in entrata per il service in costruzione) che, per questo, deve condividere la tecnologia di serializzazione scelta per loro (per interagire con i server Q/U generati tramite il QuServerBuilder predefinito si può usare una JacksonBroadcastServerQuorumPolicy).

Non è infatti importante la specifica tecnologia quanto la corrispondenza fra quelle adottate ai due lati della comunicazione.

```
val quService = new QuServiceBuilder(
  methodDescriptorFactory = new JacksonMethodDescriptorFactory with
      CachingMethodDescriptorFactory[JavaTypeable] {},
  policyFactory = JacksonBroadcastBroadcastServerPolicy[Int](_, _),
  ip = quReplica1Info.ip,
  port = quReplica1Info.port,
  privateKey = quReplica1Info.keySharedWithMe,
  obj = 0,
  thresholds = thresholds,
  storage = ImmutableStorage[Int]()).build()
```

Listing 18: Dettaglio d'uso delle API grpc-aware per la configurazione di un servizio Q/U da iniettare in un io.grpc.Server.

L'inserto 19 mostra, dunque, l'iniezione del servizio ottenuto dal builder in un server gRPC; per il suo corretto funzionamento è richiesto specificare, assieme alla porta su cui restare in ascolto, un Interceptor che ispezioni l'header (verificando l'autenticità della sorgente) compatibile con la quorum policy lato client scelta (per JacksonClientQuorumPolicy impiegata dal QuClientBuilder si può usare un JwtAuthorizationServerInterceptor). Come già descritto in design, la libreria fornisce un'implementazione già pronta, ma è possibile implementarne di alternative anche sulla base del meccanismo di autenticazione di interesse.

```
val server = InProcessServerBuilder
    .forName(id(quReplica1Info))
    .intercept(new JwtAuthorizationServerInterceptor())
    .addService(quService)
    .build
```

Listing 19: Dettaglio dell'iniezione di un servizio Q/U in un io.grpc.Server.

Gestione delle repliche La gestione del server avviene, dunque, mediante le API predisposte da gRPC con l'interfaccia io.grpc.Server.

```
server.start()
...
server.shutdown()
```

Listing 20: Dettaglio della gestione di un server gRPC

Gestione client La gestione del client rimane invariata.

9.2 Demo

L'applicazione permette di interagire a linea di comando con un contatore distribuito su un insieme di repliche per le quali è possibile simulare guasti.

Al suo avvio, viene mostrato all'utente l'elenco dei comandi, visionabile in ogni momento attraverso il comando **help**, suddivisi in operazioni di interazione con il contatore e operazioni di gestione delle repliche. Quando la schermata è visibile, il sistema ha già configurato le cinque repliche locali (cinque è la soglia minima per tollerare un guasto *crash*) ed il server di autenticazione con cui è stato già registrato il client Q/U che risulta disponibile per la sottomissione dei comandi.

```
$ docker run -it --name quCliApp qu-cli-demo
Q/U protocol example SMR System
commands summary:
command              argument              description
prof                                       show servers statuses.
value                                      get the value of the
    distributed counter.
help                                       show cli commands list.
reset                                      reset the distributed
    counter.
exit                                       shutdown the SMR system
    .
kill                 <server>              shutdown a single
    server replica for simulating fault.
dec                                        decrement the value of
    the distributed counter.
inc                                        increment the value of
    the distributed counter.
```

Listing 21: Dettaglio del sommario, proiettato al suo avvio, dei comandi resi disponibili dall'applicazione. L'elenco può essere recuperato anche attraverso il comando **help**.

Come mostra il listato 22, per incrementare il valore del contatore è sufficiente sottomettere il comando inc; per decrementarlo, dec. Ogni operazione fornisce un riscontro all'utente che ne conferma l'esecuzione con successo o meno. È anche possibile riportare il contatore al valore zero mediante il comando reset.

```
$ inc
operation completed correctly.
$ inc
operation completed correctly.
$ dec
operation completed correctly.
$ reset
operation completed correctly.
$ inc
```

```
operation completed correctly.
```

Listing 22: Dettaglio della sottomissione in sequenza di alcune operazioni sul contatore distribuito e delle risposte annesse.

Per reperire il valore attuale si digita invece value, il cui responso è riportato nel listato seguente e permette di verificare il corretto funzionamento del protocollo.

```
$ value
operation completed correctly. Updated counter value is: 1
```

Listing 23: Dettaglio della sottomissione, e rispettiva risposta, di un'operazione di recupero del valore del contatore.

Fra le funzionalità di gestione del cluster, c'è la possibilità di ispezionare lo stato di ognuna delle repliche attraverso il comando prof.

```
$ prof
operation completed correctly. Servers statuses are:
localhost:1001 -> active
localhost:1002 -> active
localhost:1003 -> active
localhost:1004 -> active
```

Listing 24: Dettaglio dell'ispezione dello stato delle repliche.

All'avvio, le repliche sono tutte in esecuzione, ma è possibile terminarne una qualunque attraverso il comando **kill** ID, dove ID è l'identificativo del server recuperabile attraverso prof, di fatto simulando una situazione di guasto di tipo *crash*.

```
$ kill localhost:1001
operation completed correctly. Server localhost:1001 stopped.
   Servers statuses are:
localhost:1001 -> shutdown
localhost:1002 -> active
localhost:1003 -> active
localhost:1004 -> active
```

Listing 25: Dettaglio della terminazione di una singola replica per simularne un guasto di tipo *crash*.

Come mostrato il listato seguente, l'applicazione integra un meccanismo di controllo per evitare che il numero di repliche scenda sotto al minimo consentito dal *threshold quorum system* per il rispetto della semantica del protocollo, proibendo all'utente di eccedere con le terminazioni.

```
$ kill localhost:1002
a problem raised up. Quorum System thresholds, which guarantees
    the correct protocol semantics, would be exceeded.
```

Listing 26: Dettaglio del controllo dell'applicazione sul tentativo di terminare repliche in numero eccedente rispetto alle soglie di tolleranza che garantiscono la semantica del protocollo.

L'utente viene informato anche qualora l'ID non fosse corretto o la rispettiva replica fosse stata già terminata, così come in caso di sottomissione di comandi che non rispettano la sintassi (listati seguenti).

```
$ kill localhost:1001
a problem raised up. Server already previously killed, can t
    kill it twice.
```

Listing 27: Dettaglio del tentativo di terminazione di una replica già terminata, con rispettivo messaggio d'errore.

```
$ kill badId
a problem raised up. Server id not valid. Server not existing.
```

Listing 28: Dettaglio del tentativo di terminazione di una replica mediante ID invalido, con rispettivo messaggio d'errore.

```
$ badCmd
command not recognized. Please attain to the syntax, digit help
    to display commands.
```

Listing 29: Dettaglio della sottomissione di un comando che non rispetta la sintassi ed annesso messaggio d'errore.

Per concludere l'esecuzione, è sufficiente digitare **exit**, che si occupa contestualmente della sua *graceful termination* e della liberazione delle risorse allocate.

```
$ exit
quitting...
```

Listing 30: Dettaglio della terminazione dell'applicazione.

10 Workflow di sviluppo

Nonostante lo sviluppo individuale del progetto, ci si è ispirati a **GitLab flow**[1]. Attraverso un approccio *feature-driven* e *test-oriented*, ogni funzionalità di rilievo è stata anticipata dalla pubblicazione della rispettiva **issue**, le quali, per questo, nel loro insieme, forniscono una panoramica delle principali funzionalità del protocollo e della loro corrispondenza con i commit, e proseguita su un branch indipendente. Per la loro pubblicazione sul branch principale si è fatto ricorso al meccanismo delle **merge request**, poi relazionate alle issue corrispondenti all'atto della loro chiusura. Sin dalle fasi iniziali di sviluppo, inoltre, è stata configurata una pipeline *continuous integration/continuous delivery (CI/CD)* per la compilazione e il testing in un primo tempo innescata preliminarmente ad ogni modifica al repository e, successivamente, per far fronte all'esaurimento della quota minuti prevista da GitLab, ad ogni merge request. Ogni modifica al main è stata infatti sottoposta, prima della sua integrazione, alla validazione della pipeline. Al termine dello sviluppo, la versione conclusiva degli artefatti è stata pubblicata su un release branch (v1.0.0) separato e sancita da un tag.

Sono, inoltre, stati impiegati diversi ulteriori strumenti per la produzione, la compilazione, il testing del codice. Oltre a git e **GitLab** per il versioning del codice, si è fatto uso di **sbt** per la *build automation* e per la pubblicazione della scaladoc, in quanto piattaforma di riferimento per il linguaggio scala impiegato. Per la stesura dei test, meglio dettagliati alla sez. specifica, i framework **ScalaTest**, **ScalaCheck** e **ScalaMock** sono stati scelti per ottenere una maggiore espressività e copertura mediante testing *property-based* e testing asincrono. La *coverage* è stata valutata attraverso il plugin **sbt-coverage**. A supporto della qualità degli artefatti, anche in ragione dei requisiti on funzionali, è stato impiegato **Scalastyle**[11], un plugin per il controllo automatizzato del codice basato su regole alla ricerca di potenziali problemi anche consigliato da Martin Oderski, autore di scala. Infine, come già meglio spiegato alla sezione 8, per facilitare il dispiegamento dell'applicazione demo si è ricorso a **docker** rendendo disponibile un **Dockerfile** che la rende fruibile come container che isola le dipendenze necessarie.

11 Conclusioni

Questo documento presenta lo studio e l'implementazione di Q/U, un protocollo presente in letteratura risalente ai primi anni 2000 per la realizzazione di servizi resilienti ai guasti, anche bizantini, attraverso un'interfaccia basata su operazioni e con un focus sulla *fault scalability*, ossia la capacità di garantire un degradamento graduale delle prestazioni nell'erogazione di un servizio all'aumentare dei guasti dei server. Nonostante risalga a quasi vent'anni fa, l'interesse di fornirne un'implementazione al giorno d'oggi è derivato dalla generale constatazione che non esista attualmente un protocollo che superi tutti gli altri in tutte le situazioni, ma che ciascuno eccella in scenari specifici [34] e dalla contestuale assenza di una sua implementazione disponibile in rete. Dopo aver rivisitato la letteratura di riferimento, fra cui l'approccio SMR, il problema del consenso e della tolleranza ai guasti bizantini, sono state presentate le assunzioni e il funzionamento del protocollo nella sua versione più semplice, prima, e delle ottimizzazioni necessarie ad affrontare i suoi punti deboli, poi. Rivisitato alla luce delle tecnologie allo stato dell'arte, come scala e gRPC, e con paradigmi con i quali non era stato inizialmente concepito, come la programmazione asincrona, Q/U è stato quindi proposto nella forma di una libreria riusabile in diversi contesti, nel cui sviluppo ha svolto un ruolo centrale il testing. Si è infine realizzata un'applicazione a linea di comando per la verifica ulteriore del suo funzionamento che ne esemplifica le funzionalità come punto di partenza per la costruzione di servizi più complessi.

11.1 Lavori futuri

Oltre che prototipale, allo stato attuale, l'implementazione non fornisce una copertura totale delle funzionalità descritte dagli autori nell'articolo di riferimento. Inoltre, è possibile integrare o rivedere una serie di funzionalità non direttamente citate nell'articolo. A questo proposito, si descrivono nel seguito le principali estensioni al progetto emerse

[11] `http://www.scalastyle.org/`

durante lo sviluppo. Come già descritto, la loro eventuale integrazione ha condizionato pesantemente il design ed è già stata predisposta attraverso un'architettura e un approccio modulare che incentiva il riuso, nel rispetto dei requisiti non funzionali descritti.

Fra le modifiche suggerite dagli autori si riportano le seguenti, non in ordine di importanza.

Politica di scelta quorum più efficiente Attualmente, la politica di coinvolgimento del quorum, sia per un client che intende sottomettere un'operazione che per un server che necessita, tramite *object-sync*, di una versione di un oggetto di cui non dispone, si limita a coinvolgere tutte le repliche. Il nucleo di Q/U è compatibile con differenti modalità di scelta delle repliche del quorum e logica di ritrasmissione in caso di risposte non pervenute. In tal senso, una strategia particolarmente ragionevole, già discussa, è data dall'assegnazione ad ogni oggetto di un quorum preferito che possa essere contattato per primo per le operazioni che lo riguardano, permettendo così di aumentare le probabilità di un'esecuzione ottimistica. Per la sua attuazione, per la quale gli autori suggeriscono di assegnare un ID all'oggetto, è sufficiente iniettare nel client o nel service già predisposti al riguardo, rispettivamente, una nuova ClientQuorumPolicy e una nuova ServerQuorumPolicy, magari richiedendo un argomento implicito che richieda la definizione della mappatura fra oggetto e repliche "preferite".

Ottimizzazione "update-query contention" Per limitare le situazioni di contesa, che, come scritto più volte, costituiscono la principale ragione di inefficienza per Q/U, e per incentivare l'approccio ottimistico che lo contraddistingue, è possibile integrarvi un'ulteriore ottimizzazione per la quale le interrogazioni non confliggono con aggiornamenti ad esse concorrenti. Accennata in [3], consiste nell'estendere la logica client ispezionando più accuratamente l'OHS risultante dal quorum coinvolto e, in tal caso, nel ritornare la risposta riferita all'ultimo candidato completo anche se la classificazione ne identifica uno successivo incompleto (poiché relativo all'aggiornamento concorrente con l'interrogazione).

Supporto alla memorizzazione più efficiente e durabile L'attuale supporto al salvataggio degli oggetti impiegato dal servizio Q/U è in memoria. Sebbene sia stata implementata la funzionalità di pruning delle replica history che riduce l'occupazione superflua, una possibile estensione riguarda l'integrazione di meccanismi più efficienti da questo punto di vista. Inoltre, per fronteggiare efficientemente i guasti che causano la terminazione inattesa delle repliche, e la conseguente perdita delle rispettive informazioni (RH e versioning degli oggetti), al momento ricostruite attraverso l'interazione con le repliche a tempo di esecuzione, un supporto durabile sarebbe necessario, anche valutando l'adozione di un DBMS e il trade-off fra la dimensione dei dati in memoria e su disco. In entrambi i casi, è sufficiente implementare l'interfaccia Storage arricchendola con nuove implementazioni iniettabili nel servizio Q/U all'atto della costruzione tramite builder. In vista dell'integrazione di nuovi approcci, sarebbe, inoltre, opportuno fornire la possibilità all'utente di configurare il supporto desiderato.

Ottimizzazione "Multi-object update" Nelle dinamiche di Q/U, la già discussa funzionalità di aggiornamento multi-oggetto fornisce un contributo importante in termini di efficienza, permettendo di ridurre le situazioni di contesa. Per la sua implementazione si può contare sulle indicazioni di dettaglio presenti in [3], da cui si evince come gran parte della logica già sviluppata possa essere riutilizzata.

Adozione di meccanismi crittografici più efficienti per la autenticazione dei messaggi Come già descritto in merito all'autenticazione dei messaggi per la verifica di integrità delle RH scambiate, sebbene i meccanismi basati su *crittografia asimmetrica* siano noti per la loro inefficienza[35], il costo richiesto per la generazione di una firma non dipende dalla dimensione dei messaggi (che per gli autenticatori dipende dal numero di repliche), a differenza della tecnica (H)MAC. Visto che, quindi, la loro adozione dipende dalla scala del servizio che si intende rendere resiliente ai guasti, una possibile estensione riguarda l'integrazione di questo meccanismo, o di altri più complessi che garantiscano altre proprietà (come confidenzialità e non-ripudiabilità etc.), quindi offrendo la possibilità all'utente di scegliere il più adatto al suo contesto.

Sono elencate nel seguito alcuni possibili estensioni non direttamente citate dall'articolo di presentazione degli autori.

Servizio di distribuzione chiavi di sessione private Anche mantenendo il meccanismo di autenticazione dei messaggi (cioè verifica dell'integrità e dell'autenticità della sorgente[35]) basato su HMAC e l'uso di JWT per l'autenticazione/autorizzazione dei client già implementati, è opinione diffusa[35] che l'impiego chiavi statiche sia meno sicuro rispetto a quello di chiavi di sessione con una scadenza limitata. Una possibile estensione riguarda, pertanto, l'adozione di un *servizio di distribuzione di chiavi* poi impiegate sia per la generazione del token sia per l'integrità delle RH, come l'algoritmo di *Diffie-Hellmann*[14], oppure di altri meccanismi come la *digital envelope*, da valutare a seconda delle garanzie in termini di sicurezza (un modello in proposito è la triade: *confidenzialità*, *integrità* e *disponibilità* (dall'inglese CIA)[35]) di cui l'utente necessita.

Introdurre tecnologia di serializzazione differente La libreria è predisposta all'integrazione con diverse tecnologie e formati di serializzazione. Attualmente basata su json con jackson, è pertanto possibile estenderla per assecondare le esigenze dell'utente, ad esempio attraverso formati binari più efficienti di quello *human-readable*.

Introdurre un meccanismo di autenticazione a livello di trasporto L'articolo non meglio declina l'accezione di "canale autenticato punto a punto client-server" non specificando se la proprietà di confidenzialità debba essere inclusa. Seguendo la terminologia di alcuni autori che non la includono[13], al momento si è scelto di usare JWT senza un meccanismo di crittografia a livello di trasporto (livello 4 ISO/OSI); è comunque possibile integrare il supporto a TLS o ad altre tecniche già predisposte da gRPC (ALTS etc.) che anche per questo è stata scelta come tecnologia. Questa modifica, sebbene

non espressamente prevista dagli autori potrebbe sostituirsi all'uso di HMAC, poiché la garanzia di integrità coinvolgerebbe il messaggio nella sua interezza e non solo le RH, e rimpiazzare la precedente estensione poiché incapsula già la logica di distribuzione delle chiavi. Operativamente, la sua integrazione è agevole: per il lato client è infatti sufficiente implementare una nuova AsyncStubFactory facendo riferimento ad una delle io.grpc.ChannelCredentials di gRPC ed iniettarla tramite il builder. Per il lato service, basta fornire a QuServer la io.grpc.ServerCredentials corrispondente tramite builder. Fra le diverse configurazioni di TLS, la *mutua autenticazione* permette di autenticare anche il canale server-server aumentando l'affidabilità del protocollo di fronte ad attacchi *man-in-the-middle* [35]. Anche in questo caso, sarebbe interessante fornire all'utente la possibilità di scegliere il meccanismo più appropriato in base alle sue esigenze in termini di triade CIA.

Introdurre un meccanismo di autenticazione/autorizzazione più evoluto Per quanto riguarda le informazioni sui client utilizzate dal protocollo (client id), l'impiego di un meccanismo a livello di trasporto non è sufficiente ma è necessario adottare il supporto all'autenticazione a livello di applicazione (livello 7 ISO/OSI) attualmente costituito dal semplice protocollo token-based contenuto nel modulo auth che fa uso di JWT come formato di token. Resta possibile innestare agevolmente strategie più avanzate (a partire dall'impiego di un JWT con validità temporale limitata fino all'uso di uno Identity and Access Management, IAM) anche solo impiegando quello già predisposto da gRPC (Google Token Based). In alternativa, gRPC è compatibile con ulteriori tecnologie, anche personalizzabili, iniettabili nel progetto implementando una nuova io.grpc.CallCredentials a cui far corrisponde il io.grpc.ServerInterceptor responsabile di estrarne l'id dell'utente a partire dai metadati e integrabile attraverso la dipendenza specifica di QuServerImpl.

Accesso alle funzionalità del protocollo attraverso API REST Per sfruttarne i vantaggi, fra cui l'interoperabilità che garantisce, la popolarità e la comprensione di cui gode e la presenza affermata di strumenti a supporto dello sviluppo che ne consegue, sarebbe interessante predisporre un punto di accesso API REST alle funzionalità delle repliche di Q/U sopra al nucleo di gRPC, modellabile attraverso strumenti OpenAPI come Swagger. Un'implementazione che considerasse direttamente REST come nucleo di partenza attraverso cui esporre le funzionalità delle repliche di Q/U è stata scartata in ragione della sua inefficienza, confermata da altri autori [32] in favore dell'approccio adottato, per il quale diverse viste possono poggiarsi su un nucleo più efficiente basato su gRPC, che fa proprio dell'efficienza una delle sue caratteristiche principali.

Gestione credenziali più accurata Per ragioni di tempo, la gestione delle credenziali nel modulo auth non è stata consapevolmente attuata in maniera sicura. L'impiego di un supporto durevole associato a tecniche di *hashing* e *salting*[35] per la loro memorizzazione e, sia lato client che lato server, di accorgimenti per il trasporto sicuro sulla rete, saranno quindi una priorità di fronte all'uso non prototipale della libreria per mitigare gli attacchi

almeno più comuni a cui sono esposti i meccanismi di autenticazione basati su password, fra cui quelli di tipo *man-in-the-middle* e *offline-dictionary-attack*[35].

Completare le implementazioni per le strutture dati distribuite Come già ricordato alla sezione 9, la libreria predispone, attraverso alcune utilità, anche un livello di astrazione superiore a quello delle API basiche di Q/U con cui semplificare la realizzazione di strutture dati distribuite, ma il suo supporto attuale è minimale e a solo scopo dimostrativo. Una possibile estensione riguarda quindi l'aumento della copertura di queste classi, anche nella direzione di implementare le interfacce appartenenti a collection framework già affermati come quello di scala o java, sfruttando il *know-how* già acquisito da parte dell'utente programmatore e consentendo il loro riuso più trasparente possibile in progetti esistenti.

Abilitare partecipazione dinamica al cluster di repliche Variazioni del carico di lavoro, guasti o minacce, sottopongono i sistemi distribuiti a diverse condizioni durante l'arco del loro funzionamento che influenzano la prestazioni e l'efficacia dei protocolli BFT. L'assenza di meccanismi per la variazione delle soglie tollerate, o per la partecipazione dinamica di repliche dopo l'avvio del cluster, potrebbe costituire un ulteriore punto debole per Q/U. Sarebbe, pertanto, utile valutare la possibile l'integrazione di tecniche che lo adattino dinamicamente a questi cambiamenti, magari considerando l'adozione di intervalli di soglie piuttosto che valori puntuali.

11.2 Conoscenze apprese

Il progetto è stata l'occasione per apprendere nuove conoscenze. Le principali sono le seguenti:

- Organizzare un progetto software per un sistema distribuito in componenti software ben definiti, modularizzando il codice per facilitarne la riusabilità e l'estensione.
- Testing sistema distribuito: per affrontare la complessità del progetto, si è potuto familiarizzare con le principali tecniche di testing, qui applicate in un contesto distribuito, esplorando in particolare *property-based* testing, BDD, *mocking* e *stubbing*, attraverso la scrittura delle specifiche di funzionamento, sia di unità che di sistema, e comprendendone, inoltre, l'importanza.
- modello di sviluppo: non già prima utilizzato, il progetto ha consentito di prendere confidenza con gitlab flow, con l'adozione di un approccio metodologico nel rispetto delle buone pratiche in termini di *build automation* e *continuous integration*, e di acquisire le conoscenze in merito a docker, alla containerizzazione e all'importanza di rendere automatizzabile, quando possibile, le attività delegabili ai calcolatori, come dimostra l'uso di generatori di codice di cui Protoc è un esempio.
- Programmazione asincrona: standard de facto per lo sviluppo di applicazioni distribuite real-world, il progetto ha consentito di consolidare la comprensione di questo paradigma, sia concettualmente che calato nel linguaggio scala.

- Conoscenza di scala: la scelta del linguaggio ha permesso di rafforzare il *know-how* su scala, sui costrutti, in particolare sull'uso dei generici, e sul design e l'implementazione dei pattern di progettazione, sia OO che relativi al mondo della programmazione funzionale.
- Scrittura client e server: si sono rafforzate le competenze in merito alla scrittura di client e server, impiegando l'astrazione di RPC asincrona per astrarre la distribuzione.
- Protocollo consenso: si è potuto studiare ed approfondire il problema del consenso nel contesto dei sistemi distribuiti, capire la sua complessità e i principali approcci e meccanismi per affrontarlo (marche temporali, rilassamento parziale di vincoli sul modello etc.). In particolare, si è potuto approfondire i principi cardine della sicurezza informatica riflettendo sulle possibili minacce che possono affliggere un sistema distribuito, assieme alle tecniche per mitigarle, fra cui l'impiego di token per l'autorizzazione e autenticazione dei nodi della rete o l'uso di funzioni crittografiche per la verifica dell'integrità dei messaggi scambiati.

Riferimenti bibliografici

[1] Introduction to gitlab flow. `https://docs.gitlab.com/ee/topics/gitlab_flow.html`. Accessed:2022-07-19.

[2] Scalatest, simply productive. `https://www.scalatest.org/`. Accessed:2022-07-19.

[3] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Correctness of the read/conditional-write and query/update protocols. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA PARALLEL DATA LABORATORY, 2005.

[4] 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.

[5] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. In *International Symposium on Distributed Computing*, pages 231–245. Springer, 1998.

[6] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *Annual international cryptology conference*, pages 1–15. Springer, 1996.

[7] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[8] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to reliable and secure distributed programming.* Springer Science & Business Media, 2011.

[9] Christian Cachin, Simon Schubert, and Marko Vukolić. Non-determinism in byzantine fault-tolerant replication. *arXiv preprint arXiv:1603.07351*, 2016.

[10] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[11] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[12] Paul Chiusano and Runar Bjarnason. *Functional programming in Scala*. Simon and Schuster, 2014.

[13] Sandro Coretti, Ueli Maurer, and Björn Tackmann. Constructing confidential channels from authenticated channels—public-key encryption revisited. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 134–153. Springer, 2013.

[14] Whitfield Diffie and Martin E Hellman. New directions in cryptography. In *Secure communications and asymmetric cryptosystems*, pages 143–180. Routledge, 2019.

[15] 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.

[16] Erich Gamma, Richard Helm, Ralph Johnson, Ralph E Johnson, John Vlissides, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[17] Shafi Goldwasser and Mihir Bellare. Lecture notes on cryptography. *Summer course "Cryptography and computer security" at MIT*, 1999:1999, 1996.

[18] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376, 2010.

[19] Cay S Horstmann. *Scala for the Impatient*. Pearson Education, 2012.

[20] 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.

[21] Ajay D Kshemkalyani and Mukesh Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

[22] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks (1976)*, 2(2):95–114, 1978.

[23] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):254–280, 1984.

[24] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[25] PA Lee and T Anderson. Dependable computing and fault-tolerant systems, v ol. 3. *Fault Tolerance: Principles and Practice. Springer Verlag, New York*, 753, 1990.

[26] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed computing*, 11(4):203–213, 1998.

[27] Dahlia Malkhi and Michael K Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, 2000.

[28] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal byzantine storage. In *International Symposium on Distributed Computing*, pages 311–325. Springer, 2002.

[29] Robert C Martin. The principles of ood. url: http://butunclebob. com/articles. unclebob. *PrinciplesOfOod (visited on 25/07/2022)*, 2003.

[30] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 31–42, 2016.

[31] Frederico Sabino. Adaptive bft protocols.

[32] Lakshmi Siva Sankar, M Sindhu, and M Sethumadhavan. Survey of consensus protocols on blockchain applications. In *2017 4th international conference on advanced computing and communication systems (ICACCS)*, pages 1–5. IEEE, 2017.

[33] Fred B Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys (CSUR)*, 22(4):299–319, 1990.

[34] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *NSDI*, volume 8, pages 189–204, 2008.

[35] William Stallings, Lawrie Brown, Michael D Bauer, and Michael Howard. *Computer security: principles and practice*, volume 2. Pearson Upper Saddle River, 2012.