Università di Bologna
Ingegneria e Scienze Informatiche LM

Progetto di Sistemi Distribuiti

HoneyBadgerBFT: studio, implementazione e utilizzo dell'algoritmo di consenso

---

Giannini Andrea
andrea.giannini10@studio.unibo.it

Guiducci Linda
linda.guiducci@studio.unibo.it

Magnini Matteo
matteo.magnini@studio.unibo.it

9 agosto 2020

Indice

Elenco delle figure

Elenco delle tabelle

Sommario

Nell'ambito dei sistemi distribuiti il problema del **consenso** ricopre un ruolo di primaria importanza. In questo report il problema del consenso viene dapprima formalmente introdotto e successivamente analizzato in base ai vari modelli e soluzioni proposti in letteratura. Viene poi presentato un algoritmo di consenso in grado di operare in modo completamente *asincrono* e in presenza di nodi *bizantini*: **Honey Badger Byzantine Fault Tolerant** (abbreviato semplicemente in *Honey Badger o HoneyBadgerBFT*). L'algoritmo viene dapprima analizzato e documentato sotto l'aspetto teorico, poi implementato in linguaggio **Scala** per mezzo del *framework* **Akka** sotto forma di libreria utilizzabile in applicazioni terze. È fornita un'applicazione per testare le funzionalità della libreria che permette all'utente di instanziare ed osservare lo stato di avanzamento dell'algoritmo *HoneyBadgerBFT*.

Capitolo 1

Consenso

Nell'ambito dei sistemi distribuiti il problema del **consenso** ricopre un ruolo di primaria importanza. È infatti fondamentale disporre di algoritmi in grado di raggiungere una comune intesa tra parti differenti su un insieme di valori proposti per la realizzazione di applicazioni distribuite.

Formalmente chiameremo le varie parti che devono conseguire il consenso *processi* (o *nodi*), distinguendo tra *processi corretti*, il cui comportamento non è alterato, e *processi faulty* o più semplicemente *fault*, il cui comportamento è alterato rispetto a quanto atteso.

Il problema assume varie declinazioni a seconda delle ipotesi relative allo scenario di esecuzione. In generale si deve sempre tenere in considerazione la possibilità che uno o più processi smettano di funzionare improvvisamente, per via di un *crash*, in questo caso si parla di *crash failure*. Inoltre si possono considerare scenari nei quali i processi assumano un comportamento alterato arbitrario, si parla allora di *processi bizantini* o *fault bizantini*.

Il consenso è il problema di raggiungere un accordo tra un insieme di processi su un valore dopo che uno o più processi ne hanno proposto uno. Più formalmente: sia P l'insieme di processi $\{p_1,...,p_N\}$ che devono raggiungere il consenso. Ogni processo p_i è inizializzato in uno stato di *indecisione* e propone un singolo valore v_i. I processi comunicano tra di loro scambiandosi i valori. Successivamente ogni processo p_i imposta il valore della variabile di decisione d_i entrando nello stato di *decisione*. Il consenso gode delle seguenti proprietà: [5]

- **Terminazione (Termination)**, ogni processo corretto prima o poi deve decidere un valore;
- **Integrità (Integrity)** se tutti i processi corretti hanno proposto lo stesso valore v, allora ogni processo corretto deve decidere v;
- **Intesa (Agreement)**, ogni processo corretto deve essere d'accordo sullo stesso valore.

Grazie a queste proprietà, il consenso è utilizzato quando non è possibile avere *trust* dei processi istanziati sui nodi della rete, il che è la situazione classica nel contesto dei sistemi distribuiti. Ciò è dovuto a:

- *crash* di alcuni nodi della rete;
- *bugs* all'interno dei processi;
- attaccanti che prendono controllo di parte dei processi per trarne un profitto;
- ritardi della rete nella consegna dei messaggi.

1.1 State Machine Replication

La necessità di sviluppare algoritmi di consenso per sistemi distribuiti nasce dalla creazione delle *State Machine Replication*. Queste sono state pensate per evitare i problemi dei servizi con architetture centralizzate, cioè quei servizi che si affidano ad un singolo server centrale. Grazie ad esse infatti è possibile eseguire la stessa macchina a stati (non necessariamente finiti) su molteplici processori indipendenti distribuiti e in parallelo, così da realizzare un sistema che sia:

- *Fault tolerant* a qualsiasi tipo di *fault*, che sia uno spegnimento, un *crash*, un'interruzione, un *bug*, ecc;
- Disponibile e reattivo;
- Replicato sia nei dati che nel software e incorruttibile.

Questo perché si fa affidamento sul fatto che, esistendoci più repliche, anche se ne venisse a meno una perché compromessa, la maggior parte delle altre repliche rimarrebbe disponibile e non corrotta.

Per macchina a stati generalmente si intende un programma comprendente proprie variabili interne, che ne rappresentano lo stato, e propri comandi, che invece possono cambiare lo stato. Quindi per ogni comando deve corrispondere una computazione deterministica. Queste possono essere di due tipi:

- **Stateless:** la computazione del comando non va a cambiare lo stato e ricevendo lo stesso input restituirà sempre lo stesso output;
- **Stateful**: in questo caso invece per avere lo stesso output è necessario che la computazione abbia, oltre agli stessi input, anche lo stesso stato di partenza.

In entrambi i casi, essendo computazioni deterministiche, possono essere replicate ed eseguite in parallelo, però nel secondo caso sono necessarie ulteriori condizioni, cioè che tutte le repliche partano dallo stesso stato iniziale e che tutti i comandi siano sottoposti alle repliche con gli stessi input e nello stesso ordine. In questo modo le repliche attraverseranno la stessa sequenza di stati e manterranno la cosiddetta coerenza di stato.

Tuttavia questo tipo di condizioni in un sistema distribuito non sono facili da raggiungere perché possono incorrere diversi problemi. Quando una replica riceve un comando ovviamente questo va comunicato anche alle altre repliche per poterle mantenere tutte allineate allo stesso stato. Qui sorgono le prime problematiche: il messaggio potrebbe andare perso, corrotto, riordinato o duplicato nel transito in rete e le repliche potrebbero di conseguenza ritrovarsi in stati diversi.

Un altro fattore che complica l'allineamento delle macchine è la mancanza di un tempo globale tra di esse. Infatti, essendo difficile sincronizzare i loro *clock*, questo conduce a delle difficoltà nell'ordinare gli eventi avvenuti su repliche differenti.

Tutti questi problemi hanno portato alla definizione del teorema **CAP**, che dimostra come non possano coesistere tutte le caratteristiche desiderate in uno stesso sistema.

Enunciato teorema CAP Un qualsiasi sistema distribuito non può garantire simultaneamente le tre seguenti proprietà:

- **Coerenza (Consistency)**: è la proprietà che assicura che tutti i nodi del sistema possiedano la stessa copia aggiornata dei dati (stato);

- **Disponibilità (Availability)**: indica che il sistema è attivo, accessibile e pronto a ricevere input e a fornire output corretti in un tempo conforme alle attese;
- **Tolleranza di partizione (Partition tolerance)**: assicura che anche nel caso in cui un gruppo di nodi crollasse per qualche motivo, il sistema continuerebbe ad operare correttamente.

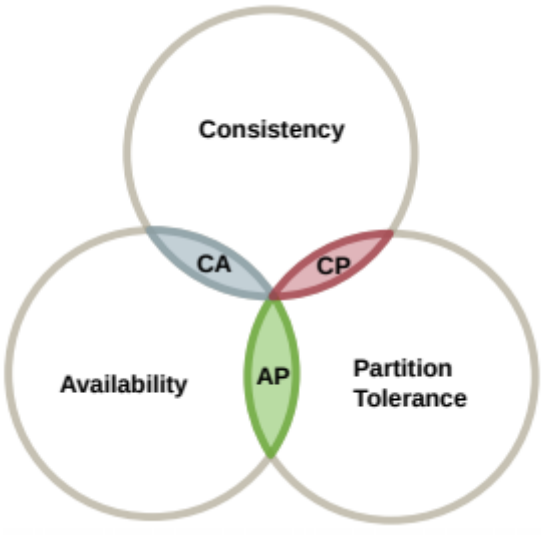

Figura 1.1: Rappresentazione grafica teorema CAP.

Grazie proprio agli algoritmi di consenso è possibile ottenere le proprietà di *availability* e *partition tolerance*, e di raggiungere anche la *consistency* anche se successivamente, dopo una quantità finita di tempo.

1.1.1 Fault Tolerance

La *fault tolerance* è la capacità di un sistema di non subire danni anche in presenza di *fault*. Per *fault* si intendono tutti quei comportamenti non coerenti alle specifiche. Il determinismo della macchina a stati è fondamentale per questa proprietà perché grazie ad esso è possibile determinare se un componente è *faulty* semplicemente confrontando gli output e lo stato con quello degli altri componenti. Da questa affermazione possiamo dedurre che il numero minimo di repliche per fare in modo che un sistema distribuito sia *fault tolerant* è di tre. Non ne basterebbero due perché altrimenti confrontando gli output non si capirebbe quale delle due repliche sia andata in *fault*. É ovvio però che un sistema a tre repliche riesca a reggere solo un nodo in *fault*.

I comportamenti *faulty* possono essere suddivisi principalmente in due categorie: *Fail-Stop Failure* e *Byzantine Failure*.

Fail-Stop Failure

Sono quei tipi di fallimenti a cui la replica risponde assumendo uno stato che permette alle altre repliche di accorgersi del problema e di fermarlo. Se si garantisce che la replica in *fault* si ferma e non restituisce un output teoricamente richiederebbe solo $F + 1$ repliche, con F uguale al numero di repliche che possono andare in *fault* contemporaneamente.

Byzantine Failure

Il nome deriva dal dilemma dei Generali Bizantini posto da Leslie Lamport nel 1982 [6]:

> Un gruppo di generali a capo di diverse sezioni dell'esercito bizantino sta pianificando di attaccare una città. L'unico modo a loro disposizione per comunicare è mediante messaggeri, e hanno bisogno di concordare il momento dell'attacco per vincere: raggiungendo un accordo l'attacco riuscirà e la città sarà conquistata, mentre non riuscire a determinare un momento univoco per agire porterebbe ad un attacco frammentato e fallimentare. Potrebbero esserci traditori tra di loro, i quali comunicherebbero messaggi discordanti per minare la riuscita dell'operazione: è quindi necessario stabilire un sistema affidabile per raggiungere il consenso sulle tempistiche di attacco.

Analogamente in un sistema distribuito potrebbero esserci repliche che potrebbero avere comportamenti malevoli o involontariamente incoerenti. Generalmente si può supporre che bastino $2F + 1$ repliche nel sistema per poter supportare questo tipo di *fault* perché basterebbe che le repliche corrette siano in maggioranza, ma non è sempre così.

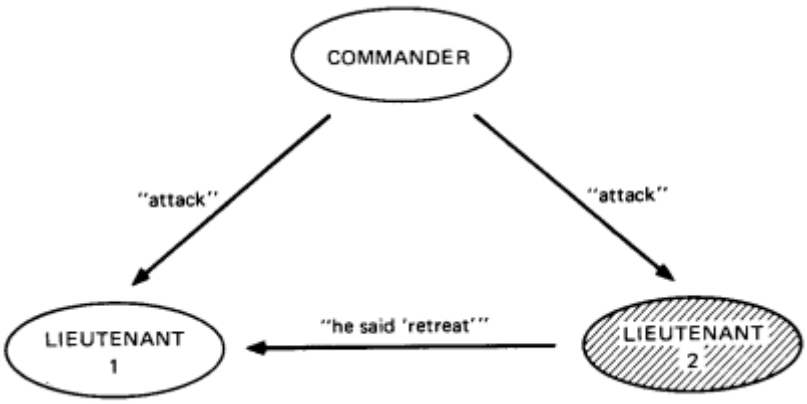

Figura 1.2: Rappresentazione grafica dei Byzantine Failure con un generale traditore.

Prendendo come esempio il caso con tre generali, stando a ciò che è stato detto in precedenza, dovrebbe reggere alla presenza di un traditore. Infatti, come si può vedere nella figura 1.2, nonostante il generale 2 tradisca riportando al generale 1 di aver ricevuto l'ordine di ritirarsi dal comandante, tutti dovranno obbedire all'ordine di attaccare, perché il generale 1 pur ricevendo ordini contrastanti dovrà obbedire all'ordine del comandante.

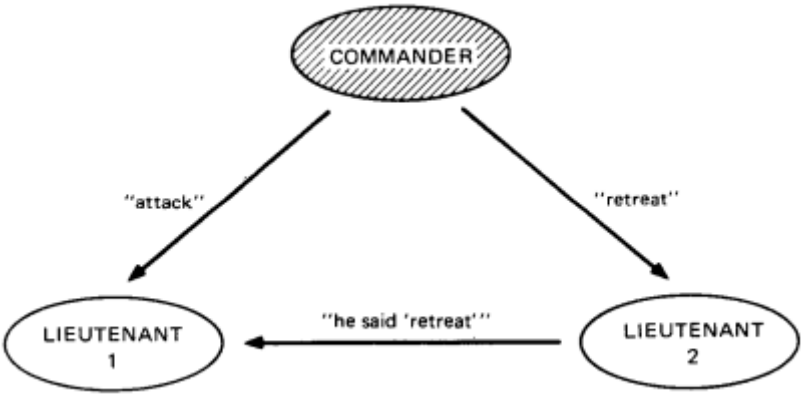

Figura 1.3: Rappresentazione grafica dei Byzantine Failure con il comandante traditore.

Se invece il traditore fosse il comandante e mandasse ordini differenti agli altri generali come in figura 1.3, si verrebbe a creare la situazione indesiderata in cui i due generali si attengono ad ordini differenti.

Dato questo risultato possiamo affermare che, nei casi in cui si mette in conto che le repliche potrebbero assumere comportamenti malevoli come in questi ultimi esempi, non ci sono soluzioni con meno di $3F + 1$ generali che riescano a contrastare F traditori.

1.2 Ipotesi temporale

Quando si analizzano algoritmi di consenso bisogna considerare sotto quali assunzioni operano. Quelle inerenti al tempo sono dette *ipotesi temporali* e riguardano le assunzioni sui tempi di consegna dei messaggi tra i vari processi. Un algoritmo progettato per operare sotto determinate ipotesi temporali non può fare altrettanto sotto ipotesi meno stringenti.

Le principali ipotesi temporali in cui ricadono gli algoritmi di consenso sono (in ordine dalla più stringente alla più lasca):

- **(Full) Sinchrony**: l'assunzione più forte che si possa fare, che consiste nel considerare che ogni messaggio scambiato tra nodi corretti della rete sia consegnato entro un certo tempo Δ (dove Δ è una misura di tempo reale). Tale intervallo diviene un parametro del protocollo e va quindi tarato opportunamente;
- **Eventual synchrony**: modello nel quale il rispetto del limite del tempo di consegna di un messaggio Δ è garantito solo dopo un certo istante (sconosciuto), noto come *Global Stabilization Time*;
- **Partial synchrony**: assunzione che si basa sempre su un massimo tempo di consegna di un messaggio, ma nella quale tale parametro Δ risulta sconosciuto, ovvero il protocollo non è in grado di usarlo come parametro;
- **Weak synchrony**: ipotesi fondata sull'avere un Δ che varia nel tempo, ovvero assume un valore crescente rispetto a t, ma dove tale crescita non può essere più veloce di una funzione polinomiale;
- **Asynchrony**: l'assunzione temporale più debole, ovvero data dal fatto che tutti i messaggi sono prima o poi consegnati, ma non è possibile fare altre ipotesi sul tempo di consegna.

I nodi in una rete asincrona non possono quindi fare affidamento sul tempo reale, ma possono basarsi soltanto sull'ordine dei messaggi che ricevono. É ovviamente lo scenario più complesso da gestire, poiché si basa su un ambiente privo di limiti e nel quale un attaccante potrebbe consegnare messaggi in ogni momento e in qualsiasi ordine.

I primi quattro casi, essendo basati su un parametro (Δ), dipendono fortemente da esso e in base al suo valore (o alla sua crescita) variano in efficienza e progresso dell'algoritmo. Infatti, avere un valore di Δ piccolo o che cresce lentamente permetterà maggiore velocità di avanzamento, ma d'altra parte potrebbe portare a performance peggiori date dal fatto che c'è più sensibilità ad eventuali latenze di rete (portando a frequenti situazioni di rielezioni del leader quando non necessario ad esempio). Nel caso in cui invece Δ fosse troppo grande oppure crescesse troppo rapidamente nel tempo, si avrebbe il problema di un recupero lento da una situazione di partizione di rete intermittente.

Uno dei vantaggi del considerare un ambiente asincrono è quindi il fatto di non avere alcun parametro di timeout da tarare, ma il progresso avviene ogni volta che i messaggi vengono consegnati, indipendentemente dal tempo. É necessario però un modo per caratterizzare lo scorrere degli eventi e un certo tipo di ordine anche in questo secondo caso. L'approccio standard a questo proposito è assegnare ad ogni messaggio un numero legato al *round* in cui quel messaggio è stato consegnato, soggetto alla condizione che ogni messaggio scambiato tra nodi al round $t-1$ deve essere consegnato prima che un messaggio di round $t+1$ sia inviato.

I protocolli non basati su ipotesi temporali di rete sincrona, anche debole, hanno il vantaggio ulteriore di essere più adatti per ambienti decentralizzati come quelli delle criptovalute. In essi infatti i collegamenti di rete possono talvolta essere irraggiungibili, la velocità di rete variabile, e potrebbero esserci latenze introdotte da eventuali attaccanti. Utilizzare in questi contesti protocolli (debolmente) sincroni potrebbe portare al fallimento della proprietà di *liveness* qualora le assunzioni fossero violate, ad esempio per la presenza di nodi *malicious*.

Anche qualora le condizioni di sincronia fossero soddisfatte, però, potrebbero esserci altri problemi, come basso *throughoutput* quando la rete sottostante fosse irraggiungibile oppure lenta ripresa dopo partizioni di rete transitorie.

1.2.1 Algoritmi

In base alle differenti ipotesi temporali e alla gestione o meno dei *fault* bizantini, sono stati proposti in letteratura vari algoritmi di consenso. La seguente tabella non è completa, riporta solo gli algoritmi più famosi.

	Crash fault	**Byzantine fault**
Synchrony		Dolev-Strong
Weak synchrony	Paxos, Raft	PBFT
Asynchrony		SINTRA, HBBFT

Tabella 1.1: Tassonomia dei principali algoritmi di consenso.

L'ipotesi temporale e il successo nella gestione dei bizantini definiscono lo spazio di operatività dell'algoritmo.

Tale spazio è più ostile se ci si colloca più a destra o più in basso nelle celle della tabella. Lo spazio di operatività non può quindi comprendere queste celle. Se, per qualche motivo, venissero a mancare le condizioni sotto le quali un algoritmo opera, questo fallirebbe il consenso.

Paxos e **Raft** operano correttamente fin quanto la rete non introduce dei ritardi improvvisi e prolungati nella comunicazione. Inoltre, se un attaccante si impossessasse di un numero sufficientemente elevato di nodi, potrebbe influenzare l'algoritmo a suo piacimento. Soprattutto per questo ultimo motivo questi algoritmi non sono idonei a lavorare in contesti in cui ci sono elevati interessi a impedire il consenso, un esempio su tutti è il dominio finanziario.

PBFT, che rientra sempre nella famiglia dei *leader-based*, è stato il primo algoritmo di consenso robusto ai *fault bizantini* che ha assunzioni sincrone deboli. Seppur robusto alla corruzione dei nodi della rete, è vulnerabile ad attacchi che riescano ad introdurre mirati ritardi nelle comunicazioni oppure con comportamento di rete impredicibile. Sotto queste condizioni l'algoritmo fallisce o le sue prestazioni crollano poiché la sua proprietà di *liveness* è strettamente dipendente dall'assunzione di sincronia [9].

SINTRA [2] è un algoritmo che opera in contesto asincrono ed è robusto ai *fault bizantini*, fa uso del protocollo *Atomic Broadcast* [1]. Queste proprietà lo rendono un buon candidato per applicazioni distribuite di *cryptocurrency*; tuttavia la complessità asintotica di comunicazione (*bits/transaction*) del protocollo di *Atomic Broadcast* risulta essere $O(N^2)$, dove N è il numero di nodi della rete.

HBBFT vuole essere un miglioramento di SINTRA: operare nelle stesse condizioni ambientali ma con prestazioni migliori. In particolare, il protocollo di *Atomic Broadcast* è sostituito dal *Common Subset Agreement* (ACS, 2.2.3) che permette di ridurre il costo di comunicazione a $O(N)$ sfruttando anche una tecnica di *batching*. Queste caratteristiche rendono *HoneyBadgerBFT* un ottimo candidato per applicazioni di *cryptocurrency*.

Capitolo 2

HoneyBadgerBFT

2.1 Contesto

L'obiettivo degli autori dell'algoritmo è quello di costruire una *replicated state machine* dove dei *client* generano e sottomettono delle transazioni e una rete di nodi li riceve e li processa. Astraendo da dettagli applicativi specifici, è sufficiente costruire un *registro (log)* che sia globalmente consistente, totalmente ordinato e al quale sia possibile solo aggiungere transazioni. Una primitiva che garantisce ciò è detta in letteratura *total order, atomic broadcast o blockchain.*

Come anticipato nel precedente capitolo, *HoneyBadgerBFT* è un algoritmo di consenso robusto ai *fault* bizantini e che opera con comunicazione asincrona. Queste due caratteristiche lo rendono particolarmente interessante in quanto è idoneo a scenari ostili. Un esempio su tutti è un'applicazione per transazioni finanziarie, nella quale un potenziale attaccante è interessato a compromettere i nodi della rete per trarne qualche profitto. La robustezza ai *fault* bizantini mette al sicuro l'utente da noti possibili attacchi, in particolare:

- **Sybil**: l'attaccante crea un numero molto alto di nodi alterati all'interno del sistema per acquisire più influenza su di esso. Fintanto che il numero di processi *faulty* è inferiore ad un terzo del totale l'attacco non ha successo;
- **Censura**: la capacità di un attaccante di impedire la pubblicazione di un determinato valore nel registro. Tale attacco è impedito grazie ad un livello crittografico che impedisce ai processi bizantini di sapere in anticipo il valore sul quale si sta raggiungendo il consenso.

HoneyBadgerBFT si colloca nello scenario di *permissioned blockchain*, contesto nel quale le transazioni possono essere effettuate da *client* arbitrari, ma i nodi responsabili dell'avanzamento dell'algoritmo sono costanti. Il modello teorico dell'algoritmo prevede le seguenti assunzioni:

- **Rete completamente asincrona**. Ogni coppia di nodi è connessa da un canale affidabile, autenticato, che non perde messaggi. Tuttavia, l'ordine con cui i messaggi sono consegnati potrebbe essere determinato da un attaccante, ma ogni messaggio è prima o poi consegnato. Poiché la rete può avere un ritardo arbitrario, si assume anche che i nodi abbiano un *buffer* illimitato. Questa assunzione permette all'algoritmo di procedere nel consenso non appena i messaggi sono consegnati. In questo modo si è completamente svincolati dal tempo reale e dalla topologia della rete sottostante;
- **Fault bizantino statico**. L'attaccante può prendere il controllo fino a f nodi *faulty*. Il *lower bound* è $3f + 1 \leqslant N$, dove N è il numero totale di nodi;

- **Inizializzazione accreditata**. Si assume per semplicità che i nodi interagiscano con un distributore affidabile per ottenere chiavi pubbliche e private.

2.2 Algoritmo

HoneyBadgerBFT è un algoritmo modulare, quindi composto da più sotto componenti o protocolli. Ogni protocollo può eseguire diverse istanze di sotto protocolli. Un protocollo esterno fornisce l'input e riceve l'output da un sotto protocollo. Ogni componente garantisce determinate proprietà, le quali nell'insieme forniscono le caratteristiche generali precedentemente descritte dell'algoritmo.

N nodi ricevono come input le transazioni dai *client* e le conservano nei loro *buffer* illimitati. Il protocollo procede in epoche, per ogni epoca viene processato un insieme di transazioni denominato *batch* che viene aggiunto in coda al registro. All'inizio di ogni epoca i nodi selezionano un sottoinsieme di transazioni dal loro *buffer*, secondo una data politica. Questo sottoinsieme viene dato in input al modulo relativo al protocollo di consenso. Alla fine del protocollo viene individuato l'insieme di transazioni per la data epoca. Il consenso è ottenuto grazie ad un'istanza del protocollo **Asynchronous Common Subset** (ACS, 2.2.3). ACS è ottenuto dall'uso di altre due protocolli: **Reliable Broadcast** (RBC o RB, 2.2.4) e **Asynchronous Binary Agreement** (ABA o BA, 2.2.5).

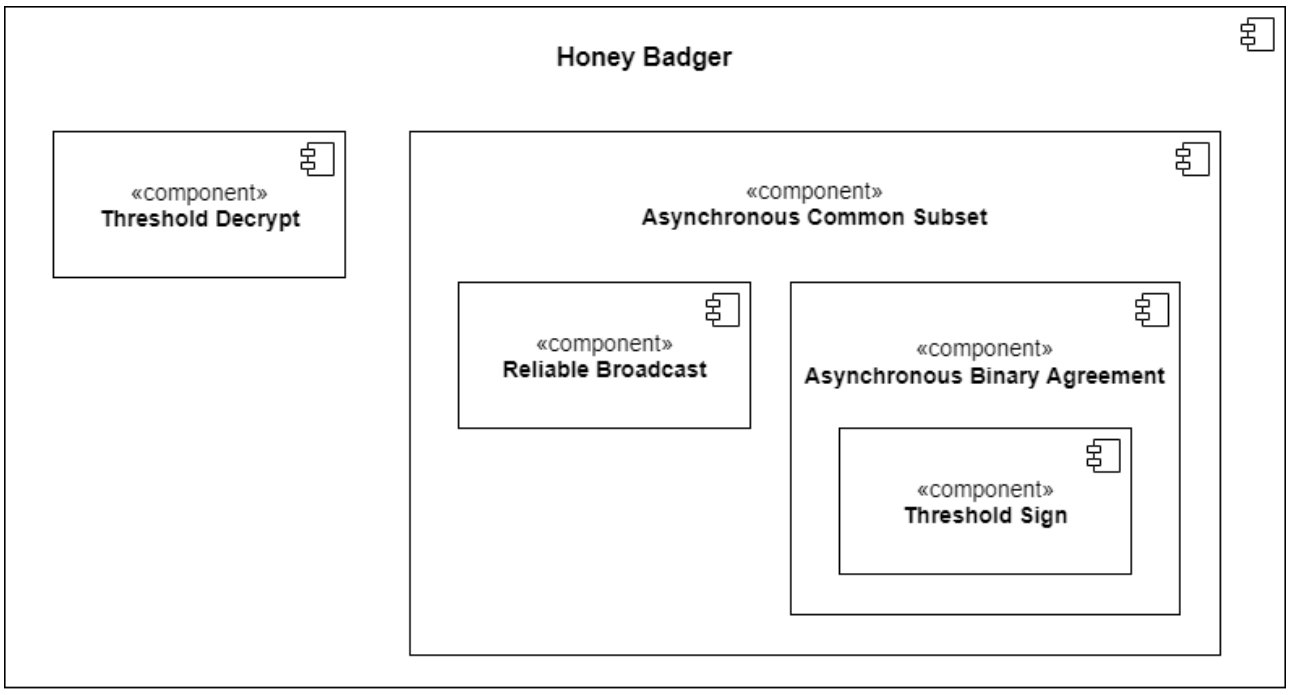

Figura 2.1: Diagramma UML delle componenti dell'algoritmo HoneyBadgerBFT.

L'algoritmo non termina mai: gestisce un continuo *stream* di transazioni provenienti dai *client* e continua a produrre nuovi *batch* da questi. Ogni nodo corretto darà in output lo stesso *batch* per ogni epoca, un *batch* contiene almeno le transazioni di $N - f$ nodi.

Nelle seguenti sezioni sono descritti in modo approfondito i componenti di *HoneyBadgerBFT*.

2.2.1 Honey Badger

L'*Honey Badger* è un algoritmo di gestione, che si occupa di tenere traccia delle epoche e di criptare e decriptare i messaggi. L'input che viene fornito a questo protocollo è un insieme di transizioni scelte in maniera randomica dalla coda dei vari nodi. In particolare, vengono scelte un numero pari a B/N transizioni dalle prime B della coda, dove con B indichiamo la *batch size*. Questo tipo di selezione è dovuta al fatto di voler ridurre drasticamente il numero di transazioni duplicate, poiché più nodi potrebbero essere connessi allo stesso *client* e avere le stesse transazioni in coda.

Quando l'algoritmo *Honey Badger* riceve le transazioni in input, cripta tali informazioni tramite la *threshold encryption*, con lo scopo di proteggere i messaggi da attacchi di censura. In questo modo, infatti, un eventuale attaccante non è in grado di capire quale nodo ha proposto quale transazione e non può quindi sopprimerne la proposta. Dopo averli criptati, l'algoritmo

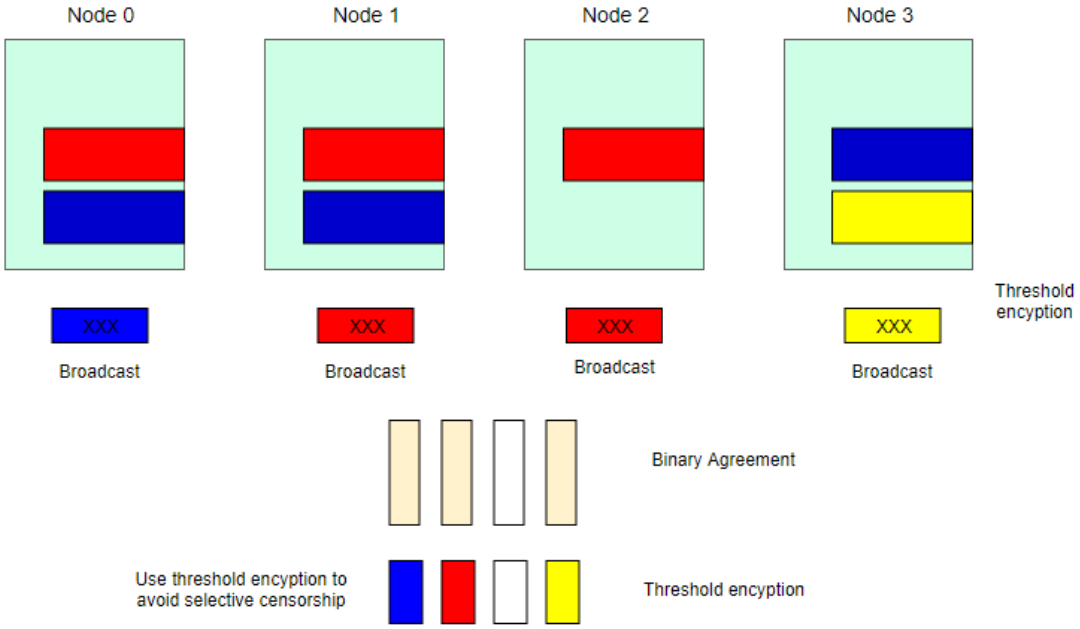

Figura 2.2: Schematizzazione della proposta dei valori da parte dei nodi e loro criptazione

Fonte: Congresso di presentazione del protocollo [8].

invia poi i messaggi all'ACS. Tale protocollo torna poi una lista con le transazioni accettate dai nodi, ancora in forma criptata, all'*Honey Badger*. La lista contiene al massimo un contributo da ogni nodo e provenienti da almeno 2/3 dei nodi. L'algoritmo prende questo risultato e lo decripta, sempre sfruttando la *Threshold encryption*; in particolare è necessario che almeno $f+1$ nodi collaborino per poter effettuare tale operazione di *decrypt*. Così facendo si ha garanzia che, se il numero di bizantini è inferiore o uguale ad f, le transazioni che si decriptano sono effettivamente quelle approvate dall'ACS. Infine, vengono eliminati eventuali duplicati e si rimuovono dalla coda le transazioni date in output dall'algoritmo.

Algorithm 1 Comportamento di un processo corretto di Honey Badger

Require: $id \leftarrow processIdentifier$
Require: $B \leftarrow batchSize$
Require: $N \leftarrow processesNumber$
Require: $PK \leftarrow pubblicKey$
Require: $SK_{id} \leftarrow secretKey$
Require: $buffer \leftarrow \{\}$ ▷ coda delle transazioni in input
Require: $round \leftarrow 0$
1: **repeat**
2: $proposed \leftarrow Random(B/N)$ ▷ prende B/N transazioni dalla testa B del buffer
3: $x \leftarrow Encrypt(PK, proposed)$
4: $ACS.input(round, x)$
5: $output \leftarrow ACS.output()$ ▷ aspetta l'output delle transazioni dall'ACS
6: **for all** transazione t_i in output **do**
7: $e_i \leftarrow DecryptShare(SK_{id}, t_i)$
8: $Broadcast(Dec(round, i, id, e_i))$
9: ▷ aspetta di ricevere f + 1 messaggi $Dec(round, i, id, e_{i,k}))$
10: $y_k \leftarrow Decrypt(PK, \{(k, e_{i,k})\})$
11: **end for**
12: $block_{round} \leftarrow Sort(\{y_j\})$
13: $buffer \leftarrow buffer - block_{round}$
14: **until** per ogni epoca

2.2.2 Threshold Encryption

L'*HoneyBadgerBFT* usa la crittografia a soglia (*Threshold Encryption*) per evitare che un singolo nodo possa influenzare la scelta delle transazioni o manomettere il loro contenuto durante l'algoritmo di consenso.

Nella tradizionale crittografia a chiave pubblica la capacità di firmare messaggi e decriptare crittogrammi è centralizzata nel detentore della chiave segreta. Nel caso della *Threshold Encryption* questa capacità viene decentralizzata "dividendo" la chiave segreta in più parti e dandone una parte per ogni nodo del sistema. In questo modo i nodi per decriptare crittogrammi e applicare firme digitali sono costretti a collaborare, perché, come suggerisce il nome, per poter completare l'operazione è necessario che venga eseguita almeno da $t + 1$ nodi, dove t è la cosiddetta soglia. Ovviamente per raggiungerla basta la collaborazione di una qualsiasi combinazione di nodi attivi.

All'interno del *Honey Badger* viene usato questo algoritmo per due casi:

- per assicurarsi che i nodi non sappiano il contenuto delle transazioni proposte dagli altri nodi finché non viene raggiunto il consenso;
- nell'algoritmo *Binary Agreement* per creare una firma condivisa da cui poter ricavare un numero casuale per decidere il risultato del *coin flip* che deciderà le sorti delle transazioni che non hanno raggiunto i 2/3 dell'*agreement*.

Nel primo caso avviene ciò che è mostrato nella figura 2.3:

1. Alice una volta selezionato il set di transazioni da proporre lo cripta con la chiave pubblica e lo invia agli altri nodi;

2. I nodi che ricevono il crittogramma lo decriptano con la loro parte di chiave privata ottenendo così come output il *decryption share*;

3. I nodi arrivati fin qui si scambiano i *decryption share* a vicenda;

4. Una volta ottenuti almeno $t + 1$ *decryption share* (ad esempio in questo caso t equivale ad 1 quindi sono bastati solo Bob e Carol) i nodi possono usarli come input insieme al crittogramma alla funzione di decriptaggio per avere come output il set di transazioni.

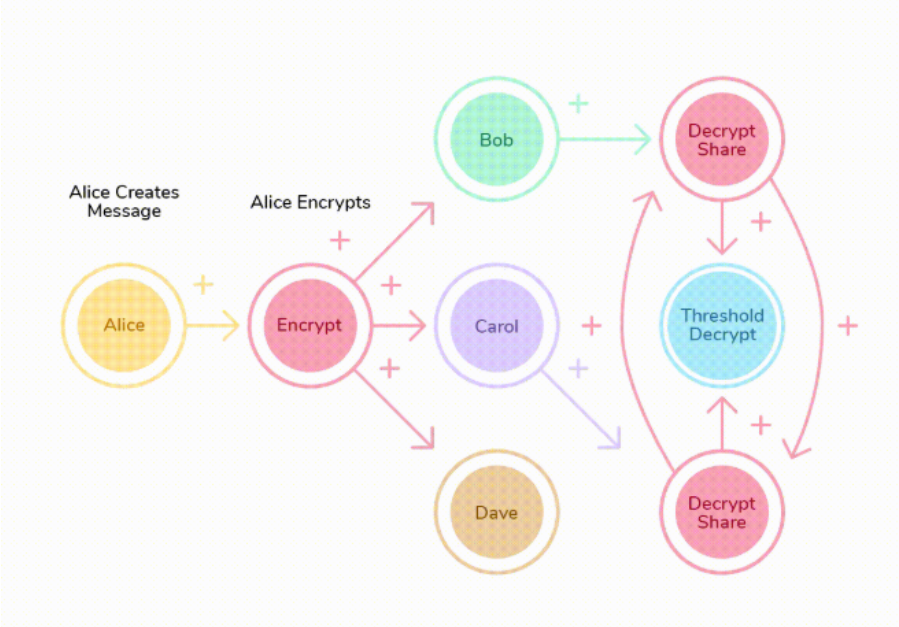

Figura 2.3: Schematizzazione della Threshold Encryption.

Nell'algoritmo di *Binary Agreement* avviene invece l'inverso, come viene mostrato nella figura 2.4:

1. Ogni nodo crea un *signature share* differente criptando un messaggio M, ognuno con la propria chiave segreta;

2. I nodi si scambiano i *signature share*;

3. Una volta che i nodi ricevono almeno $t + 1$ *Signature Share* possono combinarle andando a creare la firma completa;

4. Grazie alla firma completa adesso qualunque nodo può verificare la sua veridicità usando la chiave pubblica, così da avere la conferma che il messaggio M è stato firmato da più di t nodi.

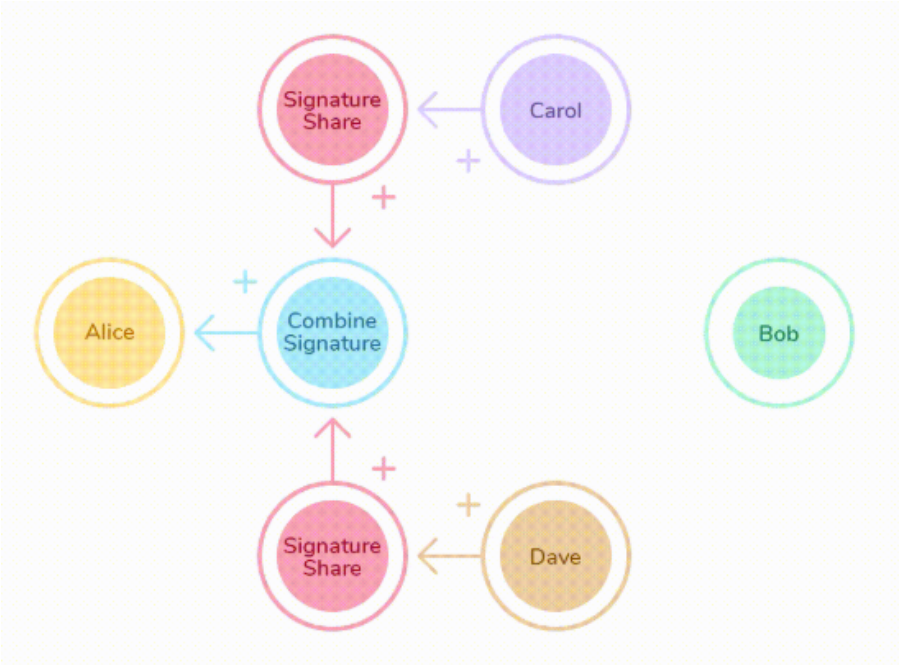

Figura 2.4: Schematizzazione della Threshold Signature.

2.2.3 Asynchronous Common Subset

L'*Asynchronous Common Subset* è il componente principale dell'algoritmo *HoneyBadgerBFT*: è dove avviene il vero e proprio consenso. Tale algoritmo, di cui viene creata un'istanza per ogni nodo, permette ad ognuno di essi di proporre un valore e garantisce che tutti i nodi abbiano lo stesso output, ovvero un vettore contenente i valori proposti da almeno $N - 2f$ nodi corretti, dove N equivale al numero di nodi della rete e f al massimo numero possibile di nodi *faulty*, con $f = (N - 1)/3$.

Tale processo invia quindi un valore (criptato) a tutti i nodi della rete e riceve valori dagli altri nodi. Per effettuare questo scambio di sfrutta il *Reliable Broadcast* (approfondito in sezione 2.2.4). Il processo istanzia anche un altro algoritmo, per ogni nodo, chiamato *Binary Agreement* (approfondito in sezione 2.2.5). Esso è un meccanismo che permette di decidere se includere o meno una transazione a quelle accettate dai nodi.

Il protocollo ACS presenta alcune importanti proprietà:

- **Validity**: se un nodo corretto ha come output un set v, allora $\mid v \mid \geq N - f$ e v contiene gli input di almeno $N - 2f$ nodi corretti;
- **Agreement**: se un nodo corretto ha come output v, allora ogni nodo ha come output v;
- **Totality**: se $N - f$ nodi corretti ricevono un input, allora tutti i nodi corretti producono un output.

Più dettagliatamente, il protocollo può essere descritto in due fasi. Nella prima, ogni nodo P_i sfrutta il *Reliable Broadcast* per propagare i valori da esso proposti agli altri nodi. Nella seconda fase invece, N istanze concorrenti di *Bizantine Binary Agreement* sono usate per accordarsi su un vettore di bit $\{b_j\}_{j\in[1,..,N]}$, dove $b_j = 1$ indica che il valore proposto da P_j è incluso nel set finale. In particolare, il voto `Yes` (1) indica che la maggioranza dei nodi ha ricevuto il valore v attraverso il *Reliable Broadcast*, il voto `No` (0) indica invece il contrario. L'unione di tutti i

valori per i quali il risultato dei BA era `Yes` è il risultato finale dell'algoritmo e permette il fatto che tutti i nodi onesti conoscano i valori su cui hanno raggiunto l'accordo.

Algorithm 2 Comportamento di un processo corretto di Asyncronous Common Subset

Require: $id \leftarrow processIdentifier$
Require: $RB \leftarrow \{RB_i\}$ ▷ insieme dei processi di reliable broadcast
Require: $BA \leftarrow \{BA_i\}$ ▷ insieme dei processi di binary agreement

1: **repeat**
2: ▷ quando riceve input v dal processo di Honey Badger
3: $transactions \leftarrow v$
4: $RB_{id}.input(transactions)$
5: ▷ quando riceve output v_i da RB_i
6: **if** non è stato dato input a BA_i **then**
7: $BA_i.input(\texttt{Yes})$
8: **end if**
9: ▷ quando riceve $N - f$ `Yes` da BA_i differenti
10: **for all** BA_i a cui non è stato dato input **do**
11: $BA_i.input(\texttt{No})$
12: **end for**
13: ▷ quando sono ricevuti tutti gli output dai BA_i
14: aspetta le risposte di tutti i RB_i i cui BA_i hanno risposto con `Yes`
15: ▷ quando tutti i RB_i i cui BA_i hanno dato output `Yes` hanno dato output
16: da in output l'unione dell'output dei suddetti RB_i
17: **until** per ogni epoca

Un approccio naïve di implementazione dell'algoritmo farebbe sì che ogni nodo debba aspettare che le prime $N - f$ istanze di *Broadcast* abbiano completato e poi proporre 1 per le istanze di BA corrispondenti e 0 per le rimanenti. Tuttavia, i nodi corretti potrebbero osservare tali completamenti avvenire in ordine differente. Dal momento che il BA garantisce solo che l'output sia 1 solo se tutti i nodi corretti propongono 1 all'unanimità, è possibile che il vettore di bit risultante sia vuoto. Per evitare tale problema, i nodi si astengono dal proporre 0 fino a che non sono certi che il vettore finale abbia settati almeno $N - f$ bit.

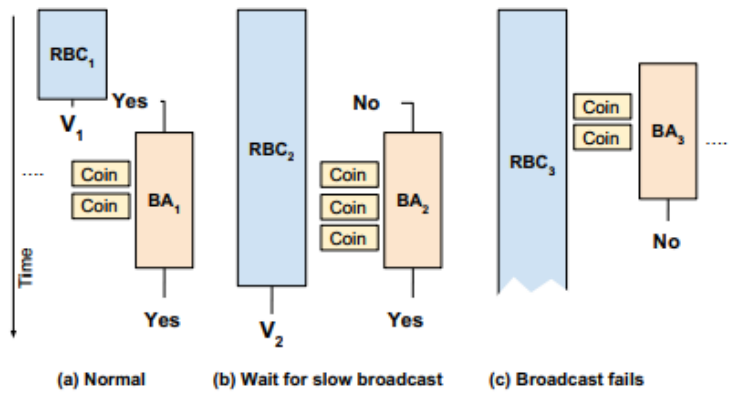

Figura 2.5: Possibili scenari d'esecuzione del protocollo ACS

Fonte: paper di riferimento [9].

In Figura 2.5 sono riportati vari esempi d'esecuzione dell'algoritmo ACS. In particolare, ogni esecuzione del protocollo coinvolge N istanze concorrenti di RBC, così come N istanze di BA, che a turno usano un numero fissato di *common coins* (vedi sezione 2.2.2). Possono realizzarsi 3 principali scenari (visti dal punto di vista del nodo 0):

- **Ordinario**: il nodo 0 riceve in input il valore v1 (proposto dal nodo 1) dal RBC_1. Il nodo 0 dà allora voto `Yes` a BA_1, il quale dà in output `Yes`.
- **Broadcast lento**: il protocollo RBC_2 impiega troppo tempo a completare e nel frattempo il nodo 0 ha già ricevuto $N - f$ voti `Yes`, quindi vota `No` per BA_2. Tuttavia, altri nodi hanno visto BA_2 completare con successo, di conseguenza esso risulta in `Yes` e il nodo 0 si mette in attesa del valore $v2$.
- **Fallimento del Broadcast**: BA_3 conclude con `No` prima del completamento di RBC_3.

2.2.4 Reliable Broadcast

Lo scopo di tale protocollo è quello di distribuire il valore proposto da un nodo al resto dei nodi della rete. Tale valore viene dato quindi come input a un nodo e come output da tutte le istanze di RB di tutti gli altri nodi. Il protocollo garantisce che ogni nodo abbia lo stesso output, anche in presenza di nodi *faulty*.

In particolare, vengono usati due principali sotto-protocolli:

1. **Reed-Solomon coding**. Esso permette di dividere un messaggio in un numero n di parti e riuscire poi a ricostruirlo usando k di esse (con $k < n$). Può essere ad esempio possibile dividere il messaggio in un numero di parti pari ai nodi della rete e scambiare quindi le parti tra tutti i nodi, riuscendo a risalire al messaggio originale anche in presenza di nodi *faulty*, che tentano attacchi di censura evitando di inviare la loro parte del messaggio agli altri nodi. Tale protocollo permette di risparmiare in larghezza di banda durante lo scambio di messaggi: come si vede in Figura 2.6, mentre nel protocollo standard il leader invia l'intero blocco a tutti i nodi, usando larghezza di banda nell'ordine di $O(N \cdot B)$, nel *broadcast* efficiente il leader manda una parte del blocco ai nodi che sono poi responsabili di effettuare gli scambi delle parti, con larghezza di banda usata nell'ordine $O(B)$;

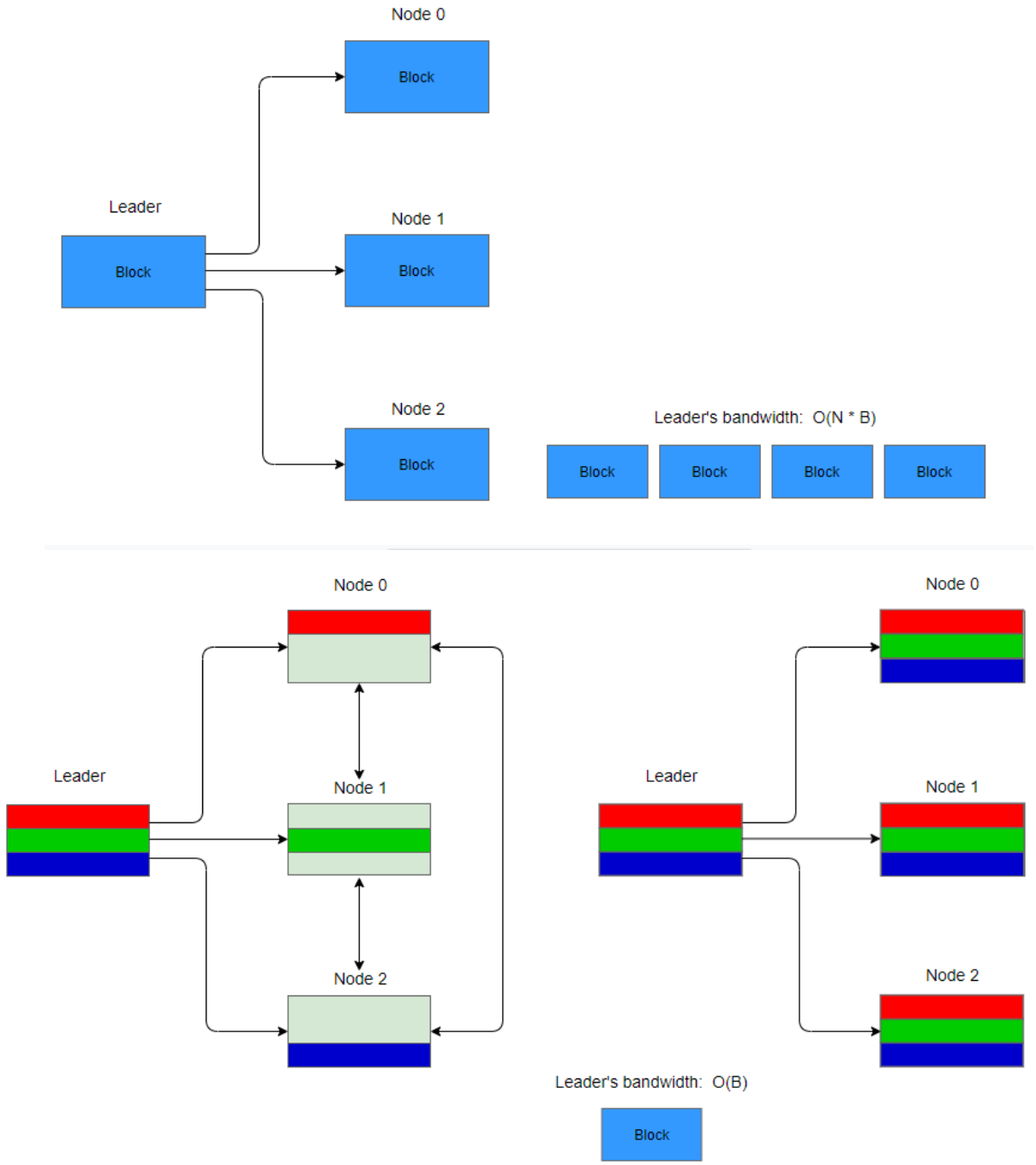

Figura 2.6: Scambio di messaggi nello Standard Reliable Broadcast e nell'Efficient Reliable Broadcast a confronto

Fonte: Congresso di presentazione del protocollo [8].

2. **Merkle Tree**. L'obiettivo di tale componente è di verificare la validità del contenuto dei messaggi scambiati tra nodi. Esso permette di verificare il fatto che una certa parte di messaggio appartenga o meno a quello di partenza, sfruttando delle piccole *prove*. I nodi provvedono quindi a scambiarsi tali *prove* per essere certi del fatto che lo stesso messaggio sia stato mandato a tutte le istanze di RB nella rete, senza essere modificato da nodi *malicious*. Il funzionamento di questo componente in particolare consiste nel calcolare dapprima gli *hash* delle varie parti in cui è suddiviso il messaggio e ricalcolare poi gli *hash*

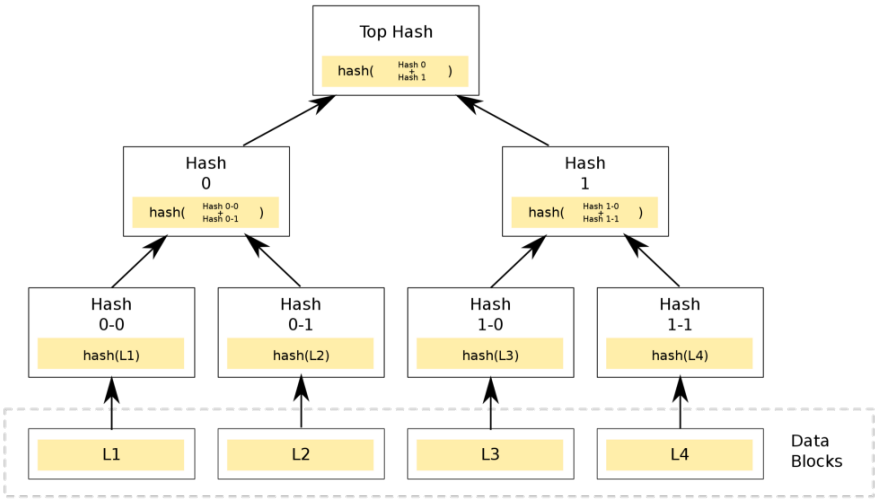

Figura 2.7: Esempio di Merkle Tree: avendo ricevuto ad esempio il blocco L4, è necessario conoscere solo l'hash `1-0` e l'hash `0` per verificare se il blocco L4 è valido, ovvero poter ricostruire l'hash alla radice dell'albero.

Fonte: `https://en.wikipedia.org/wiki/Merkle_tree`

delle varie parti concatenate a due a due. Si crea in questo modo un albero alla cui radice si ha un *hash* globale (*roothash*), sul quale si basa il procedimento di verifica della validità delle singole parti. Un esempio è riportato in figura 2.7.

Le proprietà che deve soddisfare il *Reliable Broadcast* sono le seguenti:

- **Agreement**: se due nodi corretti distribuiscono valori v e v', allora $v = v'$;
- **Totality**: se un nodo corretto distribuisce valore v, allora tutti i nodi corretti distribuiscono lo stesso valore v;
- **Validity**: se il mittente è corretto e ha come input v, allora tutti i nodi corretti distribuiscono il valore v.

Il funzionamento del protocollo è il seguente: *(i)* il mittente applica l'algoritmo di *erasure coding* all'input e ne ricava così N blocchi; *(ii)* computa poi il *Merkle Tree* a partire da tali blocchi e salva il *roothash* dell'albero; *(iii)* a questo punto, invia a tutti i nodi un messaggio `VAL` contenente un blocco, il *roothash* del *Merkle Tree* e il corrispondente *branch* dell'albero. *(iv)* Ogni nodo, una volta ricevuto tale messaggio, inoltra il blocco ricevuto agli altri in un messaggio `ECHO`. *(v)* Alla ricezione di messaggio `ECHO`, esso viene validato tramite il *Merkle Tree* e quando almeno $N - f$ messaggi `ECHO` sono stati ricevuti e considerati validi, viene inviato agli altri nodi un messaggio `READY`, se non lo si era già fatto. *(vi)* Quando un nodo riceve almeno $f + 1$ messaggi `READY`, lo invia a sua volta, a meno che non lo avesse già fatto precedentemente. *(vii)* Quando un nodo riceve almeno $2f + 1$ messaggi `READY`, e $N - 2f$ messaggi `ECHO`, decodifica v a partire dalle parti ricevute nei messaggi `ECHO`, ordinate in base all'indice dei nodi da cui sono state ricevute. Dopo aver ricostruito l'intero *Merkle Tree*, se il *roothash* ottenuto coincide con quello ricevuto dal nodo leader all'inizio dell'algoritmo, allora il nodo dà il valore decodificato in output, altrimenti scarta il messaggio.

Algorithm 3 Reliable Broadcast (per nodo P_i, con mittente P_{sender})

1: $blocks \leftarrow \{\}$
2: **if** arrivo messaggio SUBMIT(v) (a P_{sender}) **then**
3: $\{s_j\}_{j \in |N|} \leftarrow$ divisione valore v in blocchi tramite $(N - 2f, N)-$ erasure coding
4: $h \leftarrow$ roothash Merkle Tree calcolato sui blocchi s_j
5: invio messaggio VAL(h, b_j, s_j) a ogni nodo P_j (b_j = j-esimo branch del Merkle Tree)
6: **end if**
7: **if** arrivo messaggio VAL(h, b_j, s_j) (da P_{sender}) **then**
8: multicast messaggio ECHO(h, b_j, s_j)
9: **end if**
10: **if** arrivo messaggio ECHO(h, b_j, s_j) (da un nodo P_j) **then**
11: **if** b_j = branch valido per il Merkle Treee con roothash h e foglia s_j **then**
12: $blocks \leftarrow blocks + s_j$
13: **else** scarta messaggio
14: **end if**
15: **end if**
16: **if** arrivo di $N - f$ messaggi ECHO($h, \cdot, \cdot$) validi da nodi diversi della rete **then**
17: **if** non ancora inviato messaggio READY **then** multicast messaggio READY(h)
18: **end if**
19: **end if**
20: **if** arrivo di $f + 1$ messaggi READY(h) che matchano **then**
21: **if** non ancora inviato messaggio READY **then** multicast messaggio READY(h)
22: **end if**
23: **end if**
24: **if** arrivo di $2f + 1$ messaggi READY(h) che matchano **then** aspetta $N - 2f$ messaggi ECHO, poi:
25: riordina i valori s_j
26: $d \leftarrow$ decodifica valore v dalle parti s_j ricevute
27: $h' \leftarrow$ roothash Merkle Tree ricomputato
28: **if** $h' \neq$ h **then** scarta il messaggio
29: **else return** valore decodificato
30: **end if**
31: **end if**

2.2.5 Binary Agreement

Binary Agreement è il principale *building block* di *Honey Badger*: è una semplice primitiva che permette ai processi di accordarsi sul valore di un singolo bit. Per convenzione faremo riferimento ai valori binari con i nomi di `Yes` e `No`. Il protocollo è in grado di tollerare fino a $N/3$ processi bizantini e gode delle seguenti proprietà [10]:

- **Consenso (Agreement)**, se un processo corretto dà in output il valore b, allora ogni processo corretto da in output b;
- **Validità (Validity)**, un valore su cui si è raggiunto consenso è stato proposto da un processo corretto;
- **Unicità (One-shot)**, un processo corretto decide al più una sola volta;

- **Terminazione (Termination)**, tutti i processi corretti decidono. In particolare se tutti i processi corretti ricevono un input, allora tutti i processi corretti danno in output un bit.

Tali proprietà garantiscono che se un valore è proposto unicamente da processi *faulty* questo non può essere deciso. Cioè se tutti i processi corretti propongono b, allora $\neg b$ non può essere deciso.

Il contesto di utilizzo di *Binary Agreement* è identico a quello dell'intero algoritmo *Honey Badger*, ovvero si assume l'ipotesi di asincronia e il numero massimo di processi bizantini è $N/3$. Si assume inoltre che la comunicazione tra i processi avvenga in modalità *peer-to-peer*, pertanto un processo non può personificarne un altro. Tuttavia un processo bizantino può modificare la schedulazione di consegna dei messaggi, senza però intaccare l'affidabilità della rete.

N processi propongono un valore, `Yes` o `No`, e devono accordarsi sul quale scegliere convergendo solo su uno solo dei valori proposti. I processi corretti propongono un valore (`EST`) e inviano un messaggio a tutti gli altri processi (`broadcast_EST`) contenente il valore stesso e l'identificativo del turno. Un processo corretto, che si vede recapitare almeno $N/3+1$ messaggi da processi differenti contenenti il valore b, se non ha precedentemente inviato b, fa (`broadcast_EST`) di b. Questo (`broadcast_EST`) di eco viene effettuato solo una volta. Un processo corretto, che si vede recapitare almeno $2 * N/3 + 1$ messaggi da processi differenti contenenti il valore b, aggiunge tale valore nell'insieme (`bin`) dei possibili valori su cui convergere. Non appena un processo corretto vede il proprio `bin` non più vuoto effettua un nuovo (`broadcast_AUX`) di un valore (`AUX`) appartenente all'insieme. È importante tenere presente che nel frattempo altri messaggi di (`broadcast_EST`) possono arrivare e contribuire a modificare `bin`. Non appena arrivano $N - f$, con ($f = N/3$), messaggi (`broadcast_AUX`) da differenti processi contenenti un insieme di valori (`vals`) che è sottoinsieme di `bin` si calcola il valore `c` del *common coin* 2.2.6 e:

- se tale sottoinsieme contiene un solo valore (`AUX`), questo viene confrontato con `c`.
  - Se i valori sono uguali, `AUX` è il valore che i processi corretti danno in output.
  - Altrimenti si inizia un nuovo turno nel quale `EST` assume il valore di `AUX` fino a che il valore `c` non sarà uguale ad `AUX`.
- se il sottoinsieme contiene sia `Yes` che `No`, si inizia un nuovo turno nel quale `EST` assume il valore di `c`.

Algorithm 4 Comportamento di un processo corretto di Binary Agreement

Require: il processo riceve l'input b

```
1:  EST ← b
2:  repeat
3:      bin ← ∅
4:      broadcast_EST(EST)
5:      ▷ quando sono ricevuti f + 1 broadcast_EST(b)
6:      if b non è stato inviato then
7:          broadcast_EST(b)
8:      end if
9:      ▷ quando sono ricevuti 2f + 1 broadcast_EST(b)
10:     bin ← bin ∪ b
11:     while bin = ∅ do
12:         aspetta che bin ≠ ∅
13:     end while
14:     broadcast_AUX(v)
15:     ▷ v è un valore appartenente a bin
16:     ▷ quando sono ricevuti N - f broadcast_AUX(b) tali che i valori appartengano a bin
17:     vals ←valori arrivati con broadcast_AUX
18:     c ← getCoin()
19:     ▷ se vals è formato da un solo valore
20:     if vals = {b} then
21:         EST ← b
22:         if b = c then
23:             output(b)
24:             ▷ consenso raggiunto
25:         end if
26:     else
27:         EST ← c
28:     end if
29: until non si raggiunge il consenso
```

Facendo riferimento all'algoritmo 4, la prima fase (linee 1-13) è la fase di scambio del valore di input, ogni processo corretto informa gli altri processi del proprio valore di stima `EST`. Anche la seconda fase (linee 14-16) è una fase di scambio, ogni processo corretto invia un valore del proprio `bin` mentre un processo bizantino potrebbe inviare un valore arbitrario. Ciò garantisce la proprietà di validità, cioè che solo un valore proposto da un nodo corretto può essere scelto da *Binary Agreement*. L'ultima fase (linee 17-28) è una computazione locale che permette di decidere l'output, oppure di ripetere l'algoritmo con un nuovo turno qualora più di $f + 1$ processi hanno proposto sia `Yes` che `No`. Il *common coin* vine utilizzato per scegliere quale valore proporre nello scenario in cui siano leciti entrambi i valori.

Nello scenario di asincronia con la presenza di processi *faulty* è necessario l'utilizzo di una potenza computazionale aggiuntiva per poter risolvere il consenso[4]. Tale componente può essere ottenuto tramite la casualità, assunzioni sull'ordinamento dei messaggi, un servizio di individuazione dei fallimenti o assunzioni di sincronia. In questo caso il componente è un *common coin*, cioè un oggetto distribuito che consegna la stessa sequenza binaria casuale ad ogni processo (maggiori dettagli in 2.2.2). Con l'aggiunta di maggiore potenza computazionale da parte del *common coin* la proprietà di terminazione non è più deterministica. Tuttavia per

ogni processo corretto p_i vale che:

$$\lim_{t \to +\infty} (\text{Probabilità}[p_i \text{ decide al turno } t]) = 1$$

Il numero di turni attesi per giungere al consenso è costante ed equivale a 4 fintanto che i processi bizantini sono in numero inferiore ad un terzo dei processi totali.

Costo dell'algoritmo (c è il numero di processi corretti):

- se i processi corretti propongono lo stesso valore ogni turno richiede due passi di comunicazione (un `broadcast_EST` e un `broadcast_AUX`) e il numero atteso di turni è 2. Di conseguenza il numero totale di messaggi inviati da processi corretti è $2cN$;
- se i processi corretti propongono valori differenti ogni turno richiede tre passi di comunicazione (due `broadcast_EST` e un `broadcast_AUX`) e il numero atteso di turni è 4. Di conseguenza il numero totale di messaggi inviati da processi corretti è $4cN$.

2.2.6 Common Coin

Il *common coin* è un oggetto distribuito dalle seguenti proprietà:

- se $f + 1$ processi chiamano la funzione `GetCoin`, allora ogni processo riceverà prima o poi un valore comune;
- tale valore può essere 0 o 1 con pari probabilità e non può essere influenzato da un avversario;
- fino a quando almeno un processo non chiama `GetCoin`, nessuna informazione riguardo al valore è nota all'avversario.

Un *common coin* può essere realizzato tramite l'utilizzo di un unico schema di *threshold encryption* per N processi con f bizantini. L'utilizzo di tale schema richiede la distribuzione delle chiavi private e della chiave pubblica ad ognuno degli N processi. Si suppone che la fase di distribuzione sia effettuata tramite un'entità affidabile.

Algorithm 5 Comportamento di un processo corretto di Common Coin

Require: $sid \leftarrow$ nome del common coin
Require: $pk \leftarrow$ chiave pubblica
Require: $sk_i \leftarrow$ chiave privata ▷ Le chiavi sono distribuite da un'entità affidabile.

1: ▷ Quando viene chiamato `GetCoin`
2: Broadcast $\texttt{ThresholdSign}_{pk}(sk_i, \texttt{sid})$
3: ▷ Quando si ricevono $f + 1$ firme prova a combinarle
4: $sig \leftarrow \texttt{ThresholdCombine}_{pk}(\texttt{j}, s_j)$
5: **if** $\texttt{ThresholdVerify}_{pk}(\texttt{sid})$ **then**
6: invia sig
7: **end if**

Dato un messaggio m un processo i-esimo utilizza la propria chiave segreta, sk_i, per creare una *share signature*. Date $f + 1$ *share signature* del messaggio m chiunque può combinarle producendo una una *valid signature* tramite la chiave pubblica pk. Con meno di $f + 1$ *share* un avversario non ha informazioni del contenuto. Vale inoltre la proprietà di unicità per cui data una chiave pubblica pk, esiste un'unica *valid signature* per ogni messaggio m.

Capitolo 3

Obiettivi e requisiti

La libreria che ci si propone di realizzare cercherà di essere più attinente possibile all'articolo di riferimento [9], in modo che la collaborazione tra i vari moduli da realizzare risulti agevole e le interazioni corrette. La libreria dovrà pertanto essere robusta rispetto alla presenza di un eventuale avversario che introduca ritardo nelle comunicazioni o modifiche ai messaggi inviati da alcuni nodi, facendo in modo che ciò non impatti il raggiungimento del consenso. Verrà poi implementato un semplice esempio d'uso della libreria, ovvero una simulazione di un contesto bancario in cui le transazioni vengono proposte da vari nodi della rete e tramite l'algoritmo vengono accettate dagli altri nodi della rete.

Riassumendo quindi i principali requisiti che tale sistema dovrà soddisfare, essi sono:

- Permettere la proposta di valori (transazioni bancarie nel caso specifico) da qualsiasi nodo della rete;
- Garantire il raggiungimento del consenso (output delle stesse transazioni) anche in presenza di nodi *faulty*, *crash* e ritardi di rete;
- Non raggiungere il consenso in caso di presenza di nodi in *crash* in numero superiore a 1/3 dei nodi della rete;
- In presenza di nodi bizantini in numero superiore a 1/3 del totale le proprietà dell'algoritmo non sono più mantenute, in base alla gravità del comportamento bizantino può verificarsi sia una *gracefull degradetion* sia un'interruzione immediata;
- Fare in modo che un nodo eventualmente rimasto indietro nei round di consenso, una volta fatta la proposta, si riporti allo stesso stato degli altri nodi.

Si ricorda che per le ipotesi assunte nel modello di esecuzione dell'algoritmo, un nodo della rete non può in alcun modo falsificare la propria identità e assumere quella di un altro nodo.

3.1 Scenarios

In Figura 3.1 è riportato un semplice scenario d'esecuzione: i nodi propongono un valore, il quale viene diffuso agli altri nodi della rete, che sono responsabili del decidere se aggiungere o meno tale valore al set di quelli che saranno dati in output. Nello scenario presentato tutti i valori proposti sono accettati dai nodi.

In Figura 3.2 è invece riportato il caso di un nodo *faulty*, il cui valore non viene acettato dagli altri nodi a cui arriva la proposta.

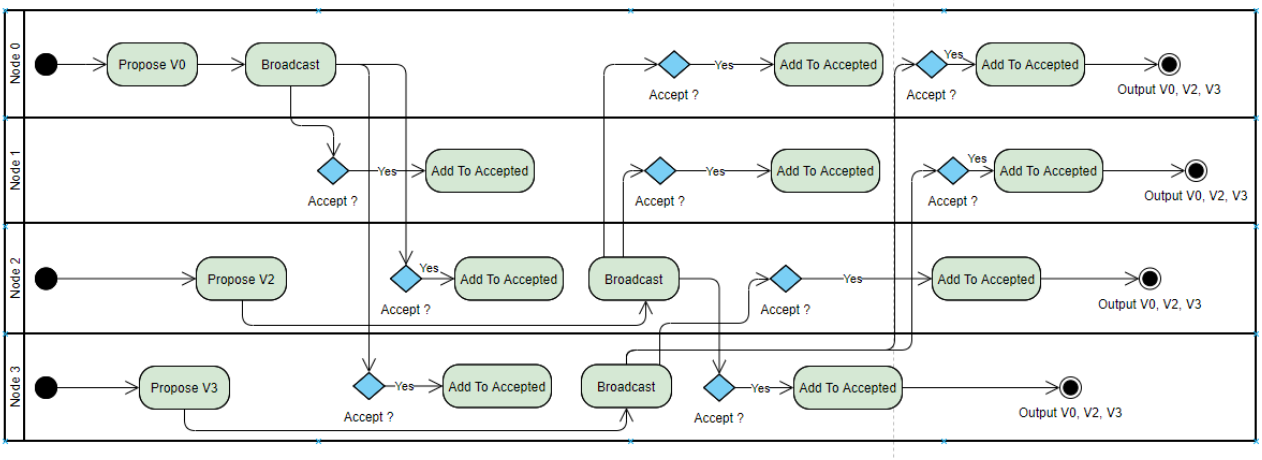

Figura 3.1: Semplice scenario in cui tutti i valori proposti sono accettati.

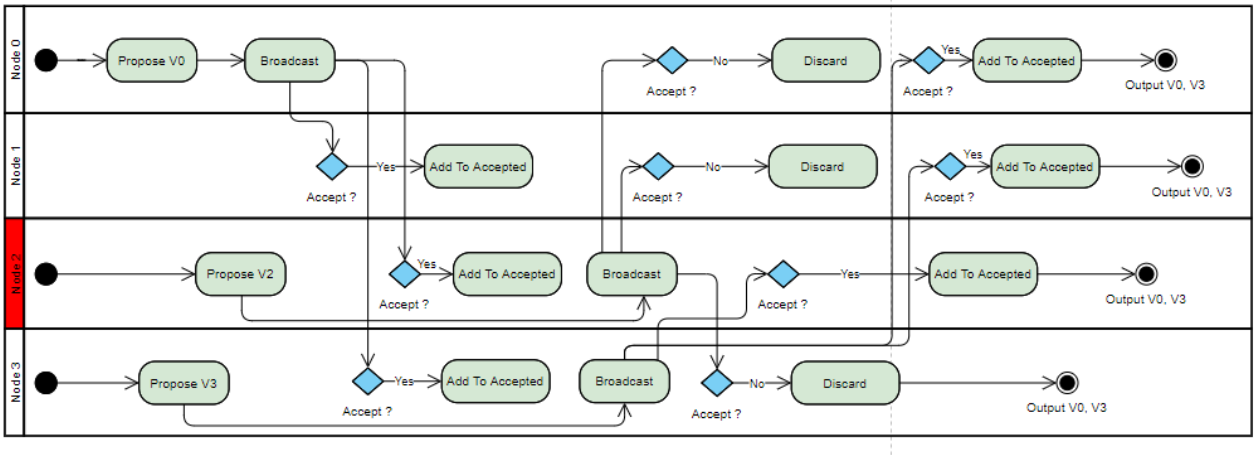

Figura 3.2: Scenario in cui non tutti i valori proposti sono accettati.

3.2 Metodi di autovalutazione

Il sistema sarà valutato in termini di **qualità** ed **efficacia** tramite un approccio *test-driven*, ovvero tramite test realizzati man mano insieme all'implementazione dei vari moduli. Per la natura dell'algoritmo, composto da moduli che lavorano sempre più ad alto livello sfruttando quelli sottostanti, questo approccio è stato molto utile e significativo ed ha aiutato uno sviluppo agevole e una migliore compartimentazione di eventuali problemi riscontrati.

I dettagli riguardanti il test sono approfonditi in sezione 6.

3.3 Analisi dei requisiti

Data la natura asincrona che le comunicazioni interne alla rete necessitano si è scelto di utilizzare l'astrazione dell'**attore** per incapsulare i vari protocolli istanziati. Gli attori comunicano tra loro per mezzo di scambio di messaggi, che sono recapitati appunto in maniera asincrona. Un attore per tutta la sua durata rimane sempre reattivo alla ricezione di particolari messaggi, al cui evento associa un *handler* di esecuzione.

Tramite la definizione di *behaviour* inoltre è stato possibile gestire gli attori come delle macchine a stati, ovvero che in base alla ricezione di certi messaggi cambiassero il loro compor-

tamento (ad esempio passando dallo stato di inizializzazione al comportamento di default). Il fatto di lavorare con i messaggi ha avuto inoltre un altro vantaggio, ovvero il fatto di poter ricostruire uno "storico" dei messaggi inviati, permettendo il recapito di essi a un nodo che venga inizializzato in ritardo. In questo modo esso rimane al passo con gli altri nodi della rete, poiché i messaggi inviati prima che esso fosse reattivo non vanno persi.

Infine, il paradigma ad attori ha permesso la facile simulazione di scenari come *delay* nella rete (ritardando l'invio dei messaggi), comportamenti bizantini (creando appositi *behaviour*) e *crash* (inviando un messaggio di `PoisonPill` all'attore).

Capitolo 4

Implementazione libreria

4.1 Design

Seguendo la filosofia del modello ad attori, secondo la quale "tutto è attore" (entro certi limiti di buon senso), ogni processo degli algoritmi descritti è implementato come attore.

Attori appartenenti alla stessa istanza di un algoritmo sono detti appartenenti allo stesso *sistema*. Gli attori possono comunicare unicamente con attori dello stesso sistema per tramite di un *router* che gestisce i loro messaggi.

Il *router* è unico per sistema ed è anch'esso implementato in forma di attore. La sua utilità è quella di fornire un ulteriore livello di astrazione, comodo per effettuare *debug* ed introdurre ritardi nella consegna dei messaggi nonché garantire le assunzioni sulle proprietà della rete. Il *router* gestisce i messaggi in modo persistente, possiede internamente un buffer per conservare ogni messaggio in arrivo così da permettere ad attori che si registrano in ritardo di non perderli. I dettagli implementativi del *router* sono descritti nella sezione 4.1.7.

Alternativamente un attore può comunicare direttamente con l'attore del protocollo sottostante o, viceversa, al protocollo soprastante. Ad esempio un attore di ACS può direttamente fornire input ai suoi sotto processi di RB e BA e da questi ricevere output.

Il codice è organizzato a moduli in modo analogo alla suddivisione teorica illustrata nel capitolo precedente. I moduli hanno la seguente organizzazione gerarchica:

Honey Badger, modulo contenitore di tutti gli altri moduli e dell'attore HB.

Asynchronous Common Subset, contiene l'attore ACS e i moduli dei restanti protocolli;

Binary Agreement, contiene l'implementazione dell'attore di BA e dell'attore CC;

Reliable Broadcast, contiene l'implementazione dell'attore RB.

Threshold Encryption, contiene le implementazioni degli algoritmi crittografici;

Utility, contiene le definizioni dei messaggi, l'attore router e metodi di configurazione usati per lo più nei test.

4.1.1 Honey Badger

Un processo di *Honey Badger* è un attore Akka. L'attore è caratterizzato da un identificativo univoco. L'attore HB comunica attraverso scambio di messaggi con gli altri attori HB per

mezzo di un *router* e direttamente con la propria istanza di ACS. Inoltre un attore HB riceve come input esterno le transazioni che sono somministrate dai *client*.

Durante ogni turno dell'algoritmo, un attore HB crea un singolo attore di ACS, responsabile della creazione di N attori per i protocolli di RB, BA e CC. Tali attori sono tutti caratterizzati dallo stesso identificativo dell'attore di HB, tuttavia ognuno di essi appartiene ad un differente sistema. Quando sono creati viene loro passato il riferimento al corrispettivo attore *router*, responsabile di un singolo sistema, che permetterà la comunicazione tra attori generati dagli altri HB. Una volta creati i CC, i loro riferimenti vengono passati in creazione ai corrispettivi BA.

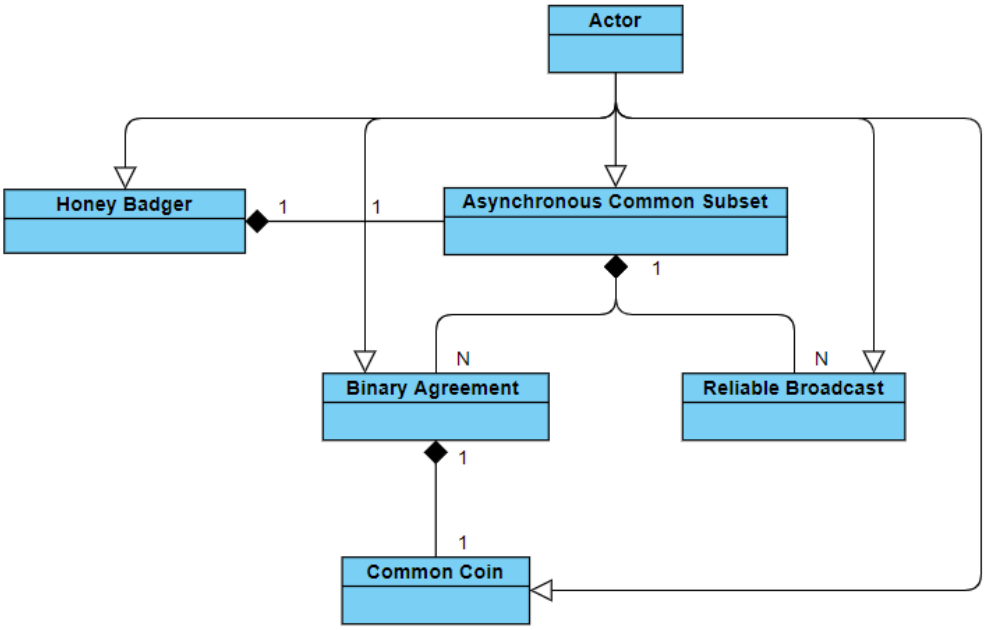

Figura 4.1: Diagramma UML delle classi per Honey Badger

L'architettura finale derivata da un'istanza di HB è quella riportata in figura 4.1. Nel diagramma 4.2 per semplicità di rappresentazione non sono rappresentati i messaggi di aggiunta, proposta e output delle transazioni dei processi di HB_1, HB_2 e HB_3.

Il funzionamento è il seguente: *(i)* un *client* invia un insieme di transazioni al nodo HB_0. *(ii)* L'attore verifica se la propria coda delle transazioni ha lunghezza uguale o superiore a *batch size* (nel diagramma ciò è affermativo). *(iii)* A questo punto HB_0 cripta le transazioni selezionate e le fornisce come input all'ACS_0. *(iv)* Quando l'algoritmo di ACS termina, ACS_0 restituisce a HB_0 le transazioni (criptate) che sono state approvate (provenienti sia da HB_0 che da altri nodi). *(v)* Nella fase finale, i vari attori di HB devono collaborare tra loro per decriptare le transazioni utilizzando la crittografia a soglia e scambiandosi messaggi `DEC` contenenti gli *share*. *(vi)* Non appena $f + 1$ *share* corretti sono ricevuti, le transazioni sono decriptate e pubblicate.

A livello implementativo, le transazioni sono limitate a 16 byte di lunghezza. Questo perché durante la fase di *batching* vengono concatenate le transazioni proposte in un'unica transazione più grande. La transazione risultante ha lunghezza 64 byte: le transazioni proposte dai nodi sono quindi sempre 4. La lunghezza di 64 byte è imposta dall'utilizzo della funzione *hash SHA512* durante le operazioni di criptaggio e decriptaggio. Per avere un numero maggiore di transazioni proposte per singolo nodo si può diminuire la lunghezza delle transazioni, tuttavia

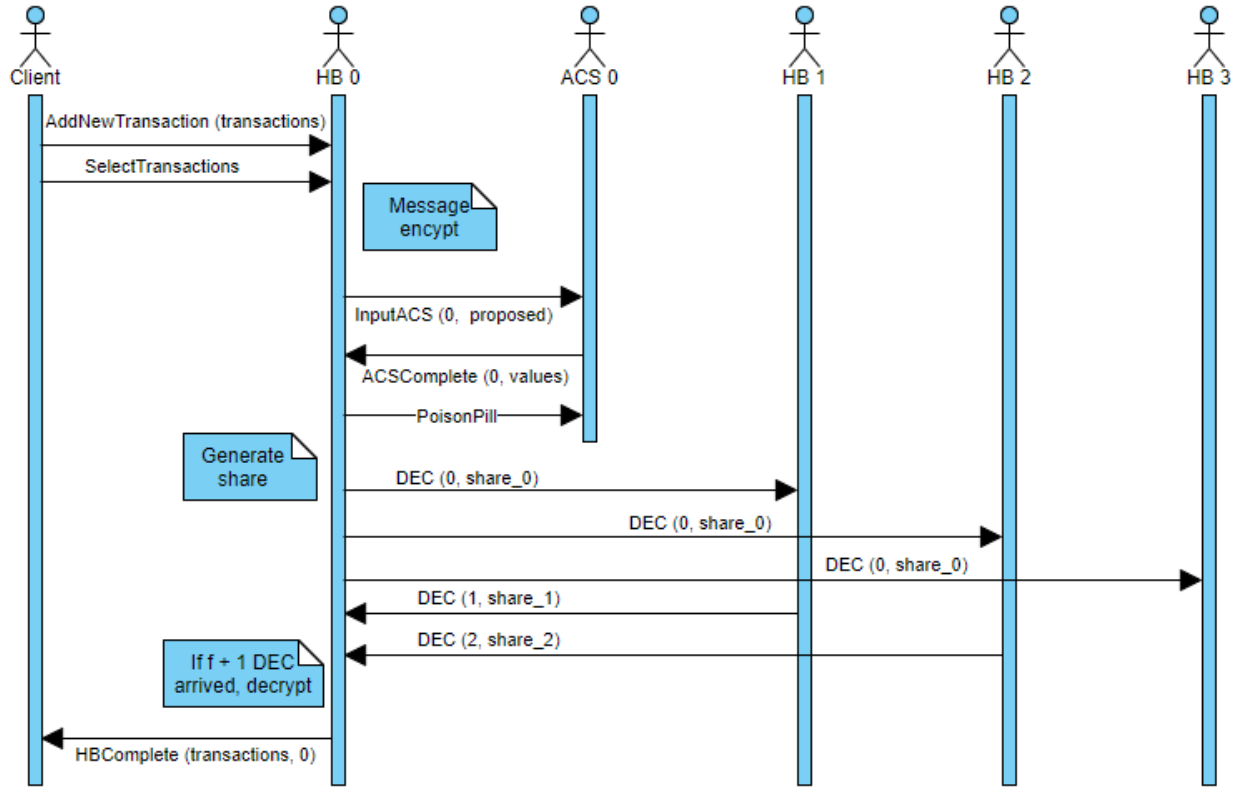

Figura 4.2: Diagramma UML di sequenza per Honey Badger.

deve rimanere valido il vincolo per cui la transazione risultato della concatenazione deve avere lunghezza di 64 byte.

Nome Messaggio	**Argomenti**	**Scopo**
InitializeHB		Inizializzazione processo di HB.
AddNewTransaction	transactions: List[String]	Aggiunta di nuove transazioni alla coda del processo.
SelectTransactions		Selezione delle transazioni da proporre agli altri nodi (inizio dell'algoritmo se la coda è abbastanza piena).
DEC	round: Int acsId: Int shares: Map[Int, Share]	Messaggio inviato internamente tra i processi di HB per scambiarsi i relativi *shares*.
KillHoneyBager		Terminazione del processo di HB.
HBComplete	value: List[String] id: Int round: Int	Completamento di un round dell'algoritmo e relativo output.

Tabella 4.1: Messaggi di Honey Badger.

4.1.2 Asynchronous Common Subset

Un processo di *Asynchronous Common Subset* è un attore Akka. L'attore è caratterizzato da un identificativo univoco corrispondente allo stesso identificativo del processo di HB. L'attore comunica per scambio di messaggi direttamente con il corrispettivo attore HB e con gli attori di BA e RB da lui creati.

(i) l'ACS viene inizializzato esso inizializza a sua volta gli attori RB e BA attendendo che questa operazione vada a buon fine. *(ii)* Successivamente l'attore inizia il suo comportamento operativo gestendo i messaggi specifici del protocollo. *(iii)* A fronte di un messaggio di input da parte dell'HB l'ACS con identificativo i provvede a inviare il messaggio di input, con il valore contenuto nel primo messaggio, all'attore RB_i. *(iv)* Se invece il messaggio proviene dall'attore RB_j, a seconda di quanti messaggi di input ai BA sono stati inviati, invia una proposta `true` o `false` a BA_j *(v)* Quando tutti i BA hanno dato risposta e tutti i RB i cui corrispettivi BA hanno dato risposta `true` hanno risposto, l'ACS da in output all'attore HB l'insieme delle transazioni approvate.

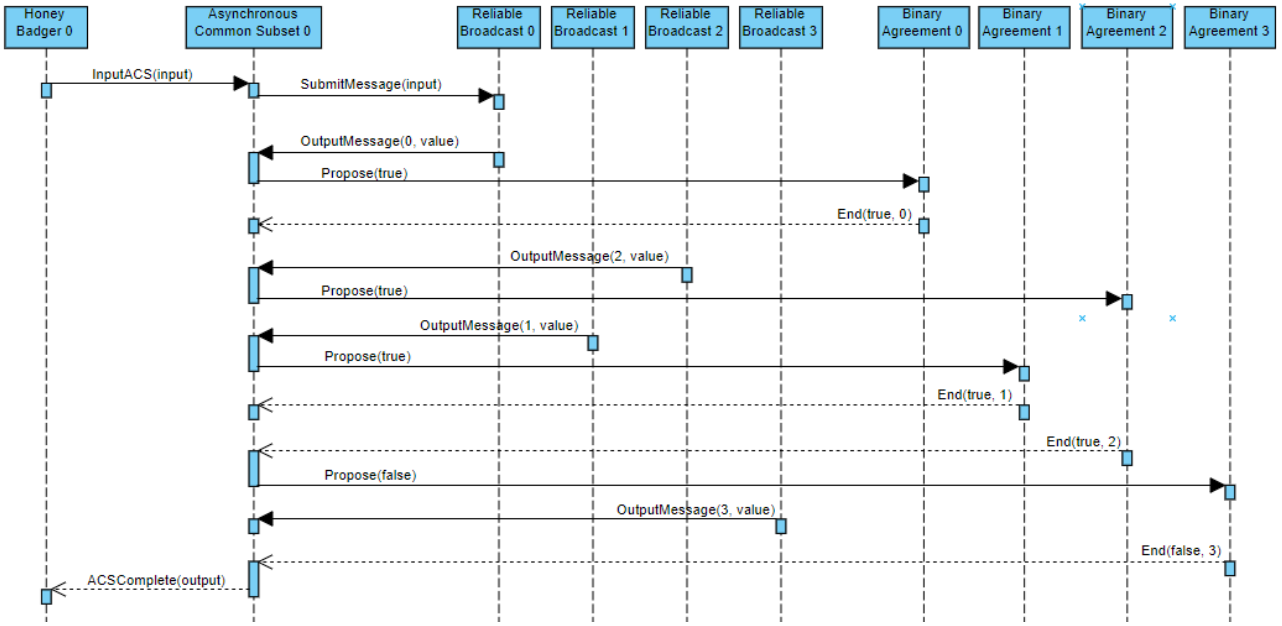

Figura 4.3: Diagramma UML di sequenza per uno scenario di esecuzione di ACS

Nel diagramma 4.3 è presente uno scenario di esecuzione con 4 attori (quindi è consentito 1 bizantino).

Nome Messaggio	**Argomenti**	**Scopo**
InitializeACS		Inizializzazione processo di ACS.
InputACS	id: Int input: Array[Byte]	Input ricevuto dal processi di HB con stesso id, dà inizio al protocollo.
ACSComplete	value: Map[Int, Array[Byte]]	Completamento dell'algoritmo e relativo output.

Tabella 4.2: Messaggi di Asinchronous Common Subset.

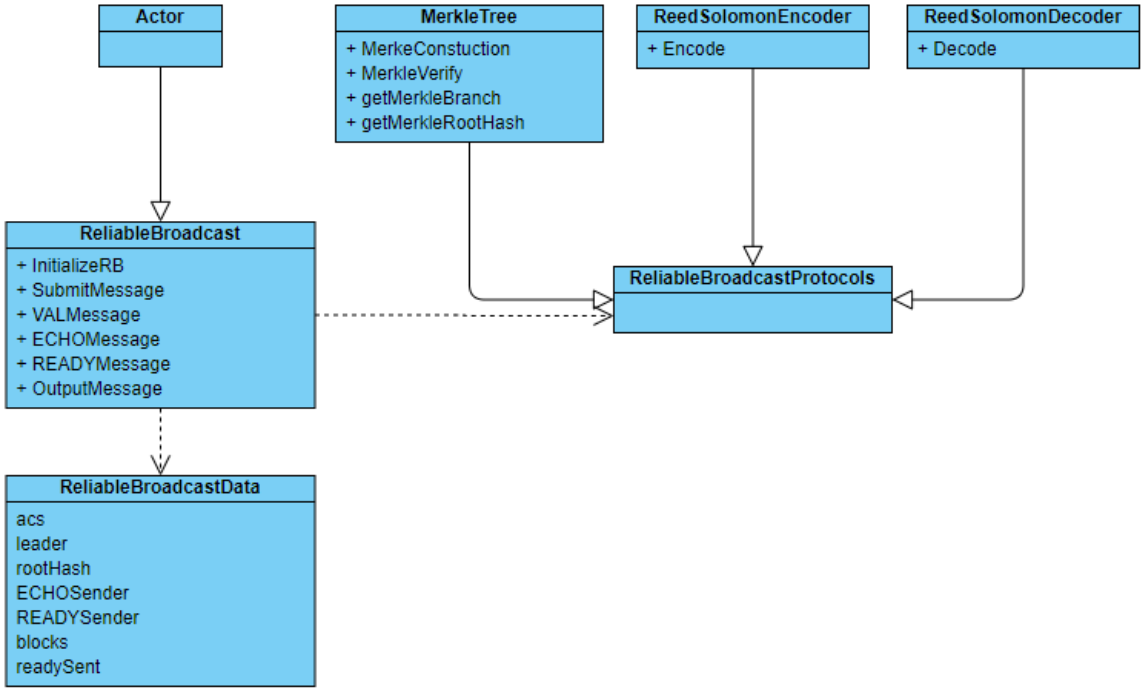

Figura 4.4: Diagramma UML delle classi del Reliable Broadcast.

4.1.3 Reliable Broadcast

Ogni istanza del processo di *Reliable Broadcast* corrisponde a un attore. Ogni processo ha un id univoco che lo identifica all'interno dello stesso sistema. Si crea quindi una sorta di matrice di processi, che comunicano tra loro all'interno dello stesso sistema e inviano il loro output al processo di ACS con loro stesso id.

In questo modo è possibile far sì che lo stesso valore sia scambiato tra RB dello stesso sistema e che essi riportino in output tale valore al loro ACS di riferimento, con il risultato che tutti i processi di ACS ricevano gli stessi valori.

Il funzionamento dell'algoritmo è gestito tramite una successione di *behaviour*: *(i)* inizialmente l'entità si trova nello stato di inizializzazione, nel quale rimane finché non viene effettivamente inizializzato dal processo ACS da cui dipende; entra quindi nel comportamento di default, che gli permette di gestire i vari messaggi in entrata. *(ii)* Quando vengono ricevuti almeno $N - f$ messaggi `ECHO` o $f + 1$ messaggi `READY`, il processo invia a sua volta un messaggio `READY`, dopo cui non può più inviare più alcun messaggio agli altri processi di RB, ma attendere solo di riceverne abbastanza per poter decodificare il valore e inviare l'output finale. *(iii)* Una volta fatto ciò, entra nello stato di output, ovvero nel quale non reagisce più ad eventuali altri messaggi in entrata, ma attende che il processo padre (ACS) lo faccia terminare. *(iv)* Ad un nuovo round vengono poi reinstanziati nuovi processi di RB da parte dell'ACS di riferimento.

L'ordine di arrivo dei messaggi non è importante, e ciò si adatta bene alla natura asincrona dello scambio di messaggi tra attori. L'unica situazione a cui bisogna fare attenzione è però l'eventuale arrivo di un messaggio `ECHO` prima del relativo messaggio VAL (che riportano lo stesso *roothash*). In questo caso infatti potrebbe accadere che un messaggio `ECHO` valido venga erroneamente scartato perché non considerato valido. Si è quindi decido di far sì che in caso non sia ancora stato ricevuto un messaggio VAL, l'attore si reinvii ogni messaggio `ECHO` o `READY` eventualmente in anticipo.

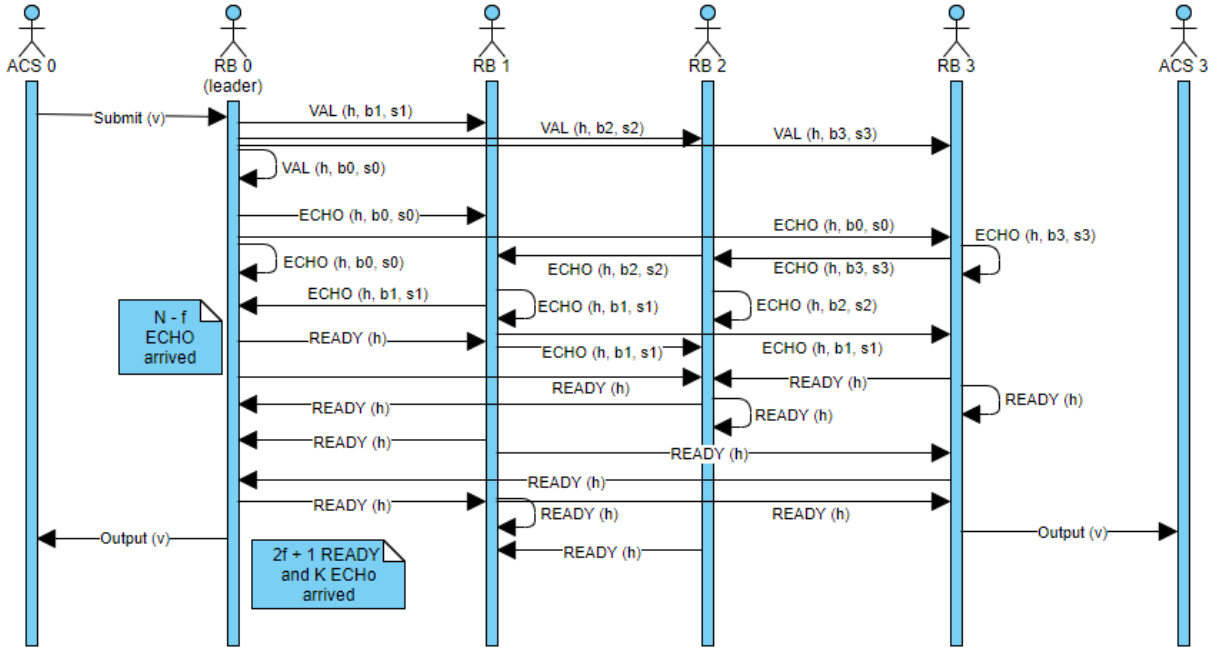

Figura 4.5: Diagramma UML di sequenza per uno scenario di esecuzione di RB.

In figura 4.5 sono riportati tutti i messaggi scambiati dai processi di RB per la distribuzione dei valori. É importante notare che ogni processo invia agli altri solo la sua parte di valore, e ogni volta che riceve una parte di messaggio da un altro processo, dopo averlo verificato e validato, lo raccoglie internamente aggiornando il proprio stato. Così facendo alla fine del protocollo sarà in grado di ricostruire il valore originale, anche in presenza di processi *faulty*. Qualora infatti ci fossero processi bizantini, se essi:

- evitassero di distribuire il loro valore, sarebbe comunque possibile ricostruire il messaggio originale grazie alle proprietà del *Reed Solomon encoding*;
- modificassero il valore inviato agli altri processi, sarebbe possibile accorgersene a causa della mancata validazione del *roothash* del *Merkle Tree* ricostruito con tale valore.

Nome Messaggio	Argomenti	Scopo
InitializeRB		Inizializzazione processo di RB.
Submit	value: Array[Byte]	Sottomissione del valore da distribuire agli altri processi, proveniente da ACS.
VAL	value: Array[Byte] branch: List[String] roothash: String	Ricezione del valore da parte del leader. Con il valore sono inviati anche il roothash e il brach del Merkle Tree che permette di validare il massaggio.
ECHO	value: Array[Byte] senderId: Int branch: List[String] roothash: String	Messaggio inviato alla ricezione di un messaggio VAL, che permette di distribuire il proprio blocco agli altri processi di RB. Contiene sempre le informazioni necessarie alla validazione del messaggio.
READY	roothash: String	Messaggio inviato alla ricezione di almeno $N - f$ messaggi ECHO.
Output	senderId: Int value: Array[Byte]	Output del valore, una volta ricevuti $2f + 1$ messaggi READY, ottenuto dalla decodifica delle varie parti provenienti da almeno $N - 2f$ processi di RB.

Tabella 4.3: Messaggi di Reliable Broadcast.

4.1.4 Binary Agreement

Un processo dell'algoritmo di *Binary Agreement* è un attore Akka. Un sistema di *Binary Agreement* è un insieme di processi BA. All'interno dello stesso sistema i processi BA comunicano tra loro tramite messaggi, attraverso un attore chiamato *router*, con lo scopo di raggiungere il consenso. Ogni processo è identificato da un numero univoco $id_i = 0, ..., N - 1$ all'interno dello stesso sistema. I processi di BA possiedono un ulteriore identificativo per il sistema di appartenenza. L'attore BA è in grado di inviare e ricevere messaggi prestabiliti con gli altri attori del sistema. Un attore BA riceve l'input sotto forma di messaggio da parte di un attore ACS. Terminato l'algoritmo di consenso, il processo BA da in output sotto forma di messaggio il valore risultante al processo ACS.

Durante la fase di creazione dell'attore ogni processo BA riceve il riferimento al proprio *common coin*. La descrizione dettagliata del funzionamento del *common coin* è nella successiva sezione 4.1.5. Ogni processo BA comunica solo con il corrispettivo CC, il CC persiste al termine dell'algoritmo di BA. Non vi è dunque necessità di reinizializzare o redistribuire i CC.

Comportamento e stato sono separati. L'attore BA e l'attore CC incapsulano il comportamento che i processi devono tenere durante l'esecuzione dei rispettivi algoritmi. Le classi *BinaryAgreementData* e *CommonCoinData* invece incapsulano le strutture dati necessarie per il corretto funzionamento.

Nello schema in 4.6 si è usato il formalismo dei metodi negli attori per indicare i messaggi ai quali è associata un'elaborazione. Sono stati esclusi dallo schema i metodi delle classi dati in quanto non fondamentali alla comprensione del comportamento degli attori.

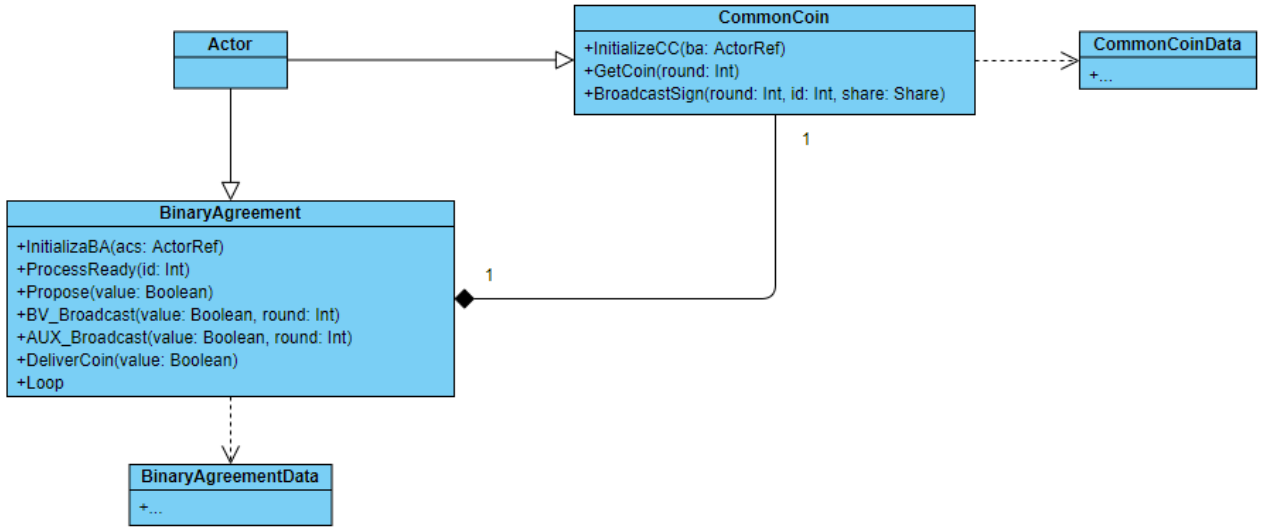

Figura 4.6: Diagramma UML delle classi del Binary Agreement.

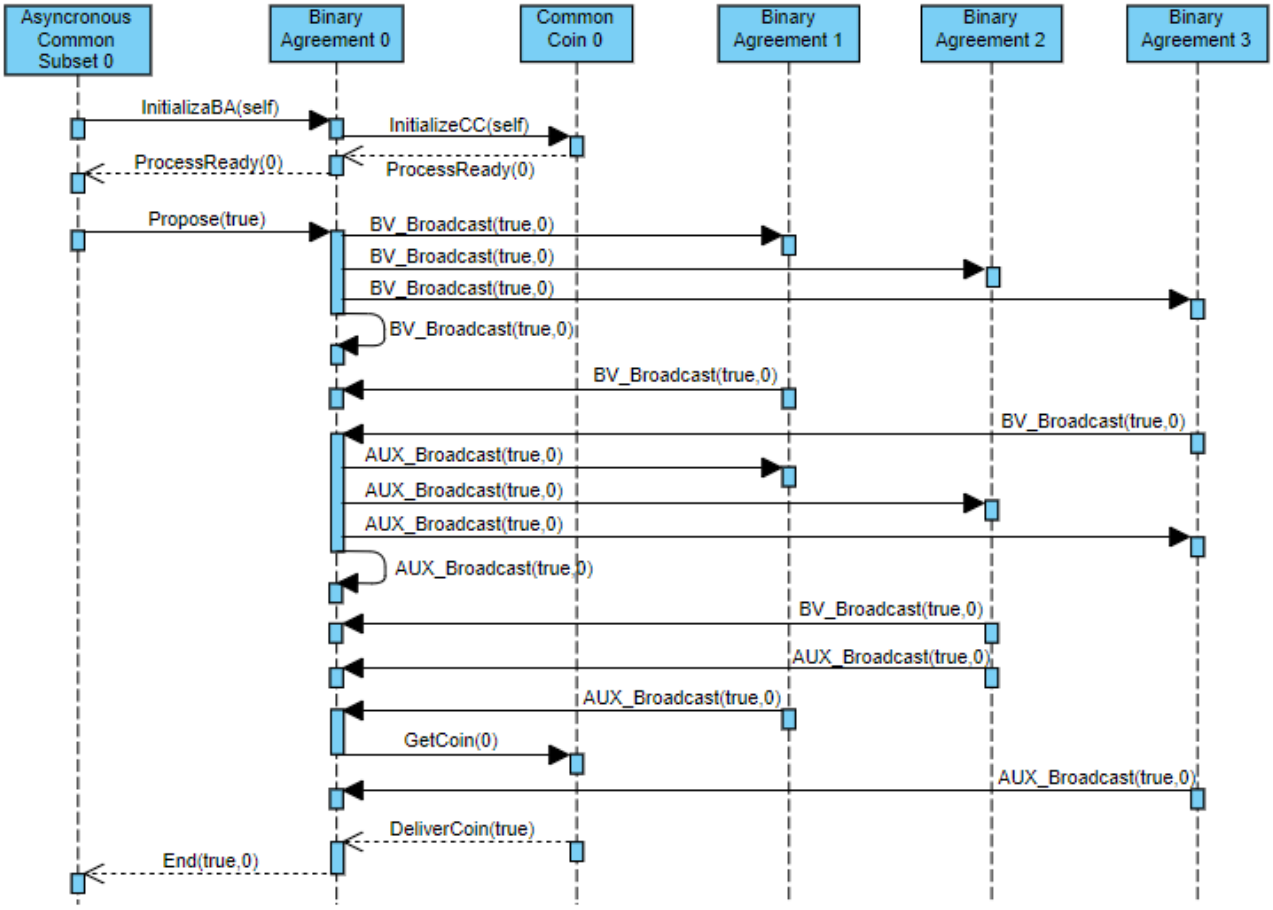

Figura 4.7: Diagramma UML di sequenza per uno scenario di esecuzione di BA.

Il diagramma di sequenza 4.7 mostra lo scambio di messaggi nella fase di inizializzazione del processo di BA_0 assieme al suo CC. Successivamente l'attore riceve un messaggio di input dall'ACS e viene illustrato un possibile scenario di esecuzione dell'algoritmo. Per migliorare la presentazione del diagramma è stato omesso l'attore *router*, facendo comunicare gli attori di

BA direttamente tra di loro. Inoltre sono stati nascosti i messaggi di input degli altri BA e i messaggi che si scambiano tra loro, assieme ai messaggi dei vari CC.

Nello scenario proposto, con 4 nodi (quindi è consentito 1 bizantino), *(i)* l'ACS da in input il valore `true` al BA_0 iniziando così l'algoritmo di *Binary Agreement*. *(ii)* Immediatamente l'attore effettua il *broadcast* del valore con gli altri processi, BA_1, BA_2 e BA_3, compreso se stesso. *(iii)* All'arrivo di $2f + 1$ messaggi BA_0 inizia il secondo *broadcast*, il messaggio di BA_2 arriva in ritardo. *(iv)* Quando giungono $N - f$ messaggi, BA_3 arriva in ritardo, BA_0 chiede a CC_0 il valore del *coin*. *(v)* Non appena più di f CC hanno ricevuto la richiesta del valore del *coin* dal corrispettivo BA, questi ritornano il valore comune. *(vi)* All'arrivo del valore, supponendo che sia `true`, BA_0 invia il messaggio di fine protocollo all'ACS contenente tale valore.

Tutti i messaggi del diagramma di sequenza sono asincroni, le frecce tratteggiate evidenziano messaggi asincroni di output a fronte di un input ricevuto.

Nella tabella 4.4 sono riportati tutti i possibili messaggi che sono scambiati tra le entità coinvolte nel completamento dell'algoritmo di *Binary Agreement*.

Nome Messaggio	**Argomenti**	**Scopo**
InitializeBA	acs: ActorRef	Inizializza l'attore di BA con il riferimento all'attore di ACS.
ProcessReady	id: Int	Il CC invia il messaggio una volta inizializzato.
Propose	value: Boolean	L'ACS da in input un valore booleano.
BV_Broadcast	value: Boolean round: Int	Gli attori di BA fanno broadcast del proprio valore di stima, oppure di un valore ricevuto più di f volte ma non ancora inviato.
AUX_Broadcast	value: Boolean round: Int	Gli attori di BA effettuano il secondo broadcast per inserire i valori nell'insieme dei possibili valori su cui convergere.
Deliver_Coin	value: Boolean	Il CC invia il messaggio al corrispettivo BA col valore del coin.
Loop		L'attore di BA invia a se stesso questo messaggio per ripetere l'algoritmo con un turno aggiuntivo.
End	value: Boolean systemId: Int	Il BA invia il valore sul quale si è raggiunto il consenso all'ACS.
InitializaCC	ba: ActorRef	Inizializza l'attore di CC con il riferimento all'attore di BA.
GetCoin	round: Int	Il BA invia il messaggio al corrispettivo CC per conoscere il valore del coin.
BroadcastSign	round: Int id: Int share: Share	Messaggio scambiato dai coin di uno stesso sistema per poter condividere e raccogliere i contributi individuali e così inviare lo stesso valore per tutti ai corrispettivi BA.

Tabella 4.4: Messaggi di Binary Agreement e Common Coin.

4.1.5 Common Coin

Un *common coin* è un attore Akka. Ogni attore di *Binary Agreement* ha un riferimento ad un unico CC, attraverso il quale può richiedere un valore binario in uno specifico turno del

protocollo stesso di BA. Per ottenere tale valore il *common coin* deve collaborare con gli altri CC dei processi di BA appartenenti allo stesso sistema. Per fare ciò ogni attore di CC ha un riferimento al *router* di quel sistema.

Il *common coin* viene creato da un processo di ACS e inizializzato con un identificativo, uguale a quello del corrispettivo BA, e un *sid*, cioè un *session identifier*. Il *sid* viene generato ad ogni turno dell'algoritmo di HB ed è utilizzato per decidere il valore binario da inviare al BA. *Common coin* corretti appartenenti allo stesso sistema hanno lo stesso *sid*. Applicando una funzione di *hash* ad un valore derivante dal *sid* e dal turno indicato nel messaggio `getCoin`, si seleziona l'ultimo bit del risultato della funzione e lo si invia al BA.

Al fine di garantire la proprietà per cui un *common coin* possa restituire un valore al BA solo dopo che almeno $f + 1$ CC hanno ricevuto il messaggio di `getCoin` con stesso valore di turno viene utilizzata la *threshold encryption*. In particolare ogni CC ha una propria chiave privata e una chiave pubblica. Al momento della ricezione del messaggio di `getCoin`, l'attore firma un valore in funzione del proprio *sid* e del turno con la propria chiave privata detto *share* e lo invia agli altri CC. Quando sono ricevuti almeno $f + 1$ *share* l'attore prova a combinarle e a verificare che il risultato di questa operazione sia equivalente al proprio *sid*. Se la verifica ha successo si calcola l'*hash* di un valore funzione del *sid* e del turno. Il risultato di questa operazione diventa il nuovo *sid*, mentre l'ultimo bit viene inviato al BA.

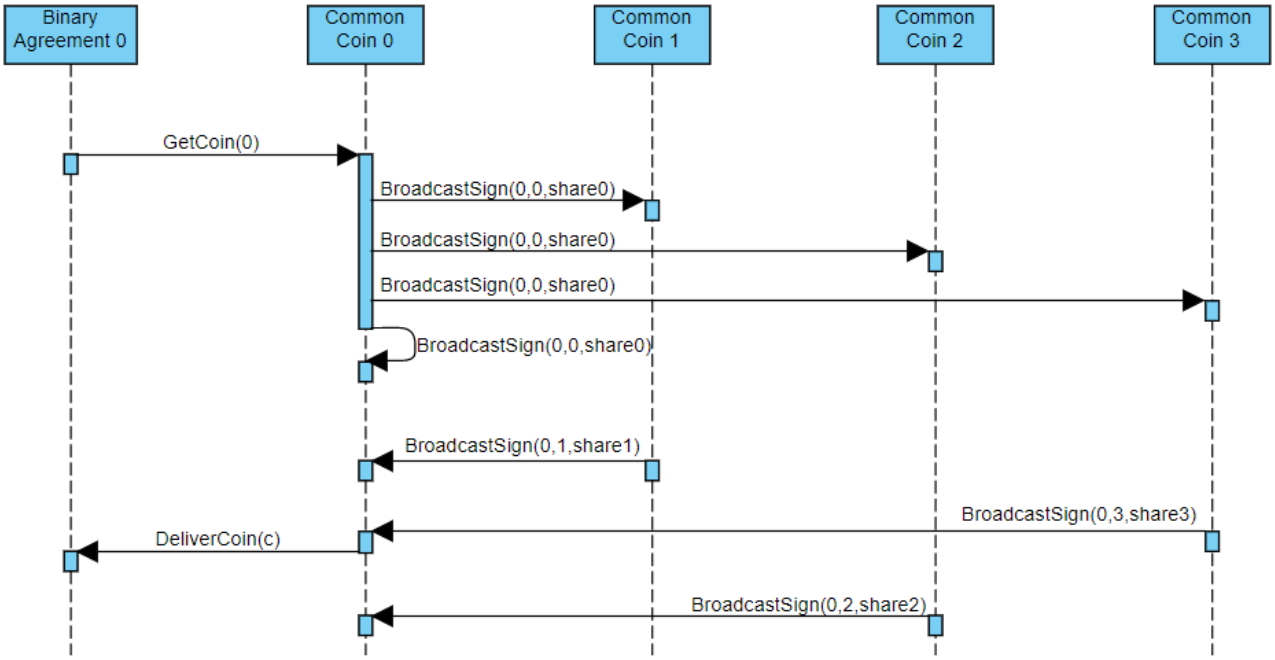

Figura 4.8: Diagramma UML di sequenza per uno scenario di esecuzione di CC.

Nello scenario proposto nel diagramma 4.8 il processo di BA_0 chiama `GetCoin` con numero di turno 0. Il CC_0 invia lo *share* generato dal proprio *sid* e dal valore del turno agli altri *common coin*. Non appena anche gli altri *common coin* ricevono dal corrispettivo *binary agreement* il messaggio `GetCoin` con turno 0, questi inviano il proprio *share* a tutti. Nel diagramma il CC_2 è in ritardo rispetto agli altri e il CC_0 riesce a verificare gli *share* di CC_0, CC_1 e CC_3 prima del suo arrivo. Invia quindi a BA_0 il valore comune.

4.1.6 Threshold Encryption

Per permettere la *Threshold Encryption* è stata sviluppata un'apposita libreria utilizzabile come *black box* nel criptaggio e decriptaggio. La sua implementazione ha necessitato dell'aggiunta al progetto dei *jar* della libreria **jPBC** [3], un *porting* in Java della libreria **Pairing-Based Cryptography (PBC)** [7]. Questa libreria permette di eseguire tutte le operazioni matematiche alla base dei sistemi di criptografia *pairing-based*.

Per rendere più adatto allo stile di programmazione di Scala e più semplice l'uso di certe funzioni è stato creato un *wrapper* implicito che faccia da DSL per la libreria, chiamato *ScalaPBC*. Grazie ad esso è possibile usare gli operatori anche nel codice per eseguire le operazioni matematiche e di ottenere gli elementi risultanti senza andare a modificare il contenuto di quelli di partenza.

All'interno del progetto la *Threshold Encryption* è fondamentale per mantenere le proprietà dell'*Honey Badger* in due punti con due modalità d'uso differenti:

- **Encryption**: all'interno dell'attore *HoneyBadger* avvengono le quatto operazioni fondamentali per l'*encryption* necessarie per evitare che un eventuale attaccante possa vedere, omettere o modificare le transazioni finché non si è raggiunto il consenso:
  - **Criptaggio** del contenuto delle transazioni selezionate;
  - **Decriptaggio** del crittogramma ottenuto con il completamento dell'*Asynchronous Common Subset* e conseguente creazione del proprio *Decrypt Share*;
  - **Verifica** del *Decrypt Share* che corrisponda al relativo crittogramma;
  - **Combinazione** dei *Decrypt Share* per completare il decriptaggio dopo averne ricevuti almeno t (valore soglia) dagli altri *HoneyBadger*, ottenendo così le transazioni.
- **Signature**: all'interno dell'attore del *CommonCoin* per assicurarsi che il valore del *coin flip* coincida per tutti i nodi dello stesso sistema e per farlo ci sono tre operazioni:
  - **Firma** di un messaggio univoco tra tutti i *CommonCoin* e creazione del *Signature Share*;
  - **Verifica** del *Signature Share* per confermare che sia la firma del relativo messaggio e che corrisponda a quella del nodo dichiarato al suo interno;
  - **Combinazione** dei *Signature Share* per formare una firma univoca per tutti i nodi dello stesso sistema; così il *CommonCoin* potrà prendere l'ultimo bit per avere il risultato del *coin flip*.

Come si può vedere nella figura 4.9, al livello di interfacce i vari metodi sono stati divisi in base alla modalità d'uso e al tipo di chiave necessaria per l'esecuzione dell'operazione. Nonostante siano due modalità d'uso differenti si basano comunque sulle stesse chiavi, per cui, anche se al livello di interfacce i metodi sono stati separati, le chiavi ereditano sia la parte di *Encryption* che quella di *Signature*.

Per quanto riguarda la creazione delle chiavi avviene a partire dal numero di nodi presenti e dal valore *soglia*, cioè quel valore che indica quanti *share* differenti sono necessari per completare il processo di decriptaggio e di firma. Oltre a questi valori è possibile anche impostare i parametri per la creazione della curva, da cui deriveranno i valori delle chiavi (di default una curva di tipo A), e la lunghezza dei messaggi da criptare. Questa deve essere fissa perché nel criptaggio viene fatta un'operazione di *or esclusivo* bit a bit tra il messaggio da criptare e un *hash* di un altro elemento. Quindi la lunghezza del messaggio determinerà la funzione di *hash* che verrà

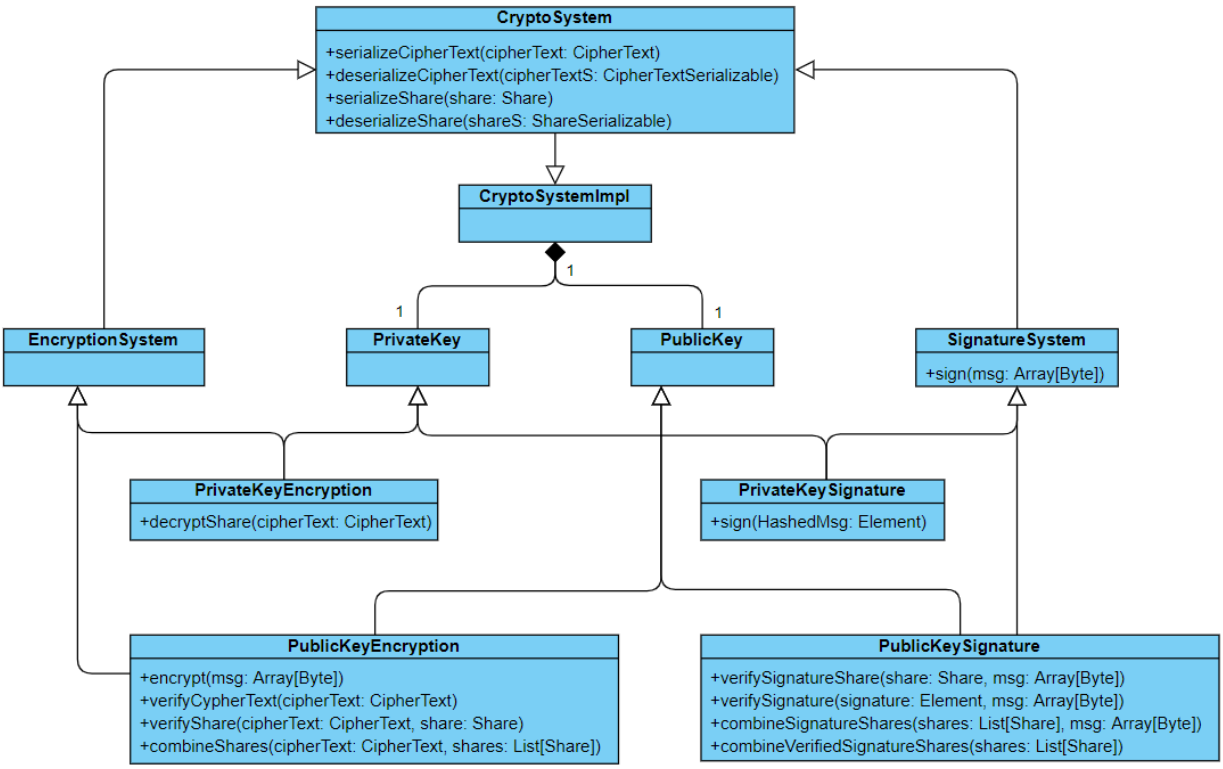

Figura 4.9: Diagramma UML delle classi per la Threshold Encryption.

usata: ad esempio nel caso sia impostata a 32 byte, valore impostato di default, verrà usata come funzione *hash* SHA256.

In questo progetto la creazione delle chiavi è stata svolta in modo centralizzato per poi distribuirle ai vari *Honey Badger*. Però ad essere rigorosi andrebbe fatta in modo distribuito, così da evitare che un singolo nodo venga a conoscenza di tutte le chiavi private e che vengano distribuite in rete.

In realtà ciò che viene distribuito sono i **CryptoSystem**, oggetti che incapsulano al proprio interno i dati della chiave pubblica (*PublicKey*) e della chiave privata (*PrivateKey*), permettendo così di chiamare tutte le funzioni di criptaggio. Nel *CryptoSystem* sono stati aggiunti anche i metodi per permettere la serializzazione e deserializzazione dei crittogrammi e degli *share*, rendendo possibile così il loro invio in rete.

4.1.7 Router

Il *router* è un attore Akka che permette una comunicazione persistente tra processi dello stesso protocollo. La sua funzione principale è quella di garantire che ogni messaggio inviato da attori a lui registrati sia ridirezionato all'attore destinatario del messaggio. Se il destinatario del messaggio non è ancora registrato, quando questo effettuerà la registrazione al *router* riceverà i messaggi a lui destinati. Il *router* inoltre può essere utilizzato per introdurre ritardi nelle comunicazioni in modo da simulare un ritardo di rete.

Il *router* non prevede una fase di autenticazione degli attori che si registrano, non per problemi di fattibilità, ma semplicemente perché dalle assunzioni della rete un processo non può in alcun modo falsificare la propria identità e spacciarsi per un altro.

Un attore si registra presso il *router* tramite il messaggio `RegisterToRouter`, indicando il proprio identificativo e il turno dell'*Honey Badger*. Un attore registrato presso il *router* può inviare a tutti gli altri attori registrati con stesso turno di *Honey Badger* (e a quelli che si registreranno) un messaggio relativo al proprio protocollo tramite il *router* con il messaggio `BroadcastToRouter`. Nel protocollo di *Reliable Broadcast* un processo ha necessità di inviare un singolo messaggio a uno specifico destinatario. Per permettere ciò un attore invia il messaggio tramite `OneMessageToRouter` indicando l'identificativo del destinatario.

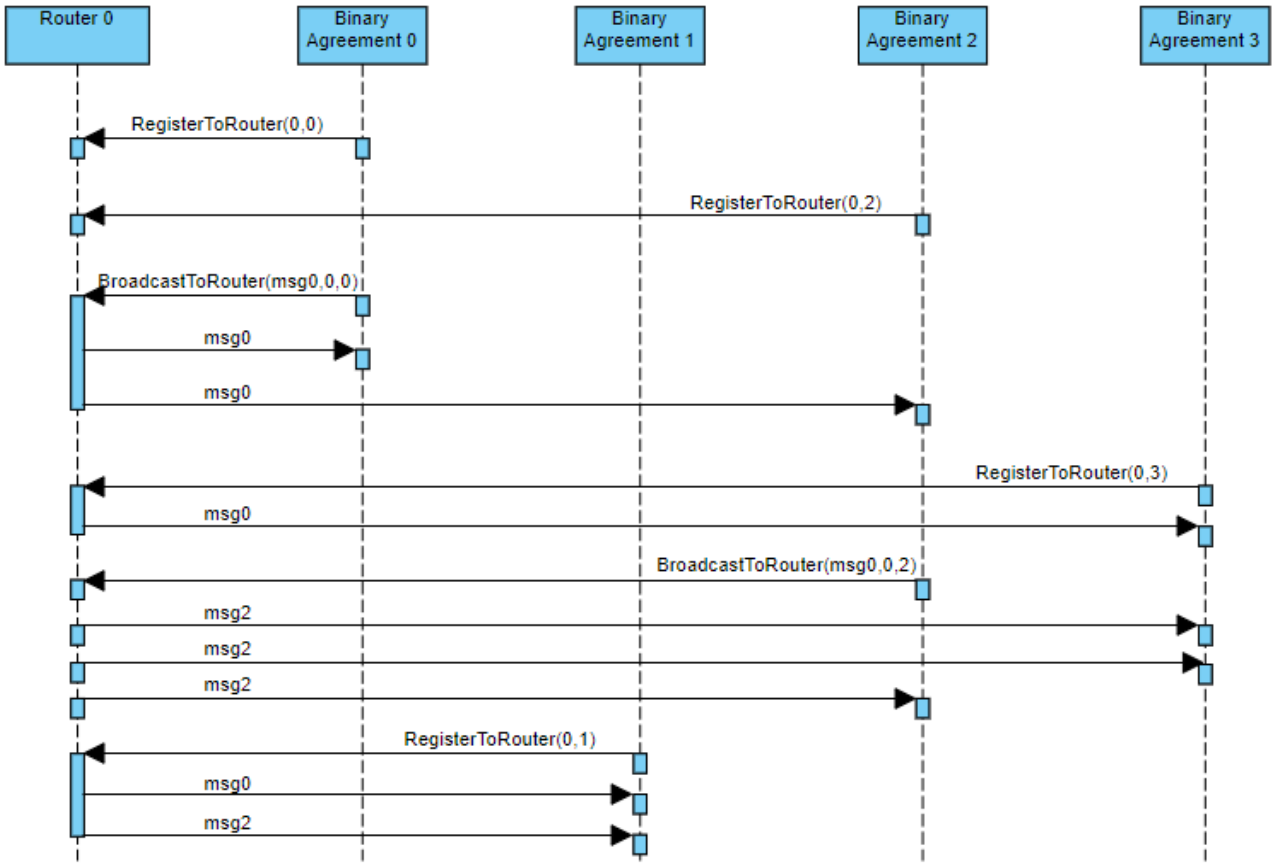

Figura 4.10: Diagramma UML di sequenza per uno scenario di esecuzione del router.

Nel diagramma in figura 4.10 è rappresentato uno scenario in cui il *router* responsabile del sistema 0 dei *binary agreement* riceve messaggi di registrazione per il primo turno dell'algoritmo dell'*Honey Badger*. Inizialmente si registrano BA_0 e BA_2. BA_0 invia un messaggio all'interno di `BroadcastToRouter` così tutti gli attori attualmente registrati ricevono il messaggio, cioè BA_0 e BA_2. Successivamente si registra BA_3 che si vede ricevere il messaggio inviato da BA_0. BA_2 invia un messaggio all'interno di `BroadcastToRouter` così tutti gli attori attualmente registrati ricevono il messaggio, cioè BA_0, BA_2 e BA_3. Infine si registra BA_1 che si vede ricevere il messaggio inviato da BA_0 e BA_2.

Nome Messaggio	Argomenti	Scopo
RegisterToRouter	round: Int id: Int	Registrazione di un processo al router, per un certo round.
BroadcastToRouter	msg: Message round: Int id: Int	Broadcast del messaggio a tutti i processi registrati.
OneMessageToRouter	msg: Message destId: Int round: Int id: Int	Invio di un messaggio tramite il router da un processo ad un altro tra quelli registrati.

Tabella 4.5: Messaggi di Router.

4.2 Linguaggio e Framework

Per l'implementazione sia della libreria che dell'applicazione di test si è scelto di utilizzare il linguaggio **Scala**. La scelta è stata dettata principalmente dal fatto che il *framework* su cui ci si è basati e che sembrava più appropriato è **Akka**, che trova nel linguaggio Scala il principale supporto.

4.2.1 Scala

Scala è un linguaggio che segue la filosofia della programmazione *object-oriented*, secondo cui "*all is an object*", aggiungendo ad essa un supporto di programmazione funzionale. L'uso di tale linguaggio si è mostrato molto efficiente per vari motivi, tra i quali il fatto di poter definire funzioni di *utility* e piccoli *dsl* per compiere operazioni in modo più agile. Il fatto che Scala sia un linguaggio ancora "di nicchia" non è stato un limite data la sua completa *java-compatibility*, che ha permesso l'uso di librerie esterne scritte in Java, qualora non fosse possibile reperirne specifiche per Scala.

4.2.2 Akka

Il *framework* Akka ha permesso di gestire la programmazione concorrente e distribuita in maniera totalmente asincrona e *message-based*. Le principali *feature* che caratterizzano l'astrazione dell'*attore* sono:

- **Immutabilità dello stato interno.** A tale proposito, sono state create apposite *case class* in modo da non dover mai modificare direttamente lo stato interno dell'attore, ma lavorare sempre con aggiornamento dei *behaviour* e passaggio di oggetti immutabili;
- **Scambio di messaggi.** I messaggi sono alla base della definizione del comportamento dell'attore e della gestione delle interazioni tra attori diversi. I messaggi si configurano pertanto come una sorta di interfaccia che definisce gli eventi che una certa entità deve gestire.

- **Organizzazione gerarchica.** Gli attori sono organizzati gerarchicamente, in base a come sono creati: un attore creato da un altro è dipendente da esso, di conseguenza l'attore "padre" può agire come supervisore degli attori figli e gestire così loro eventuali *failure*.
- **Modularità**. La modularità dell'applicazione è data dall'utilizzo di *actorSystem*, ovvero dei "canali" che permettono agli attori di comunicare. É un modo per rispettare il *single responsibility principle* e mantenere il codice pulito, modulare e più facile da mantenere;
- **Trasparenza di località**. Lo scambio di messaggi avviene senza necessità di sapere da dove i messaggi provengano: il mittente potrebbe risiedere nella stessa *jvm* del ricevente come in una separata (nello stesso nodo o in un altro).

4.3 Dettagli implementativi

4.3.1 ScalaDoc

Per rendere la libreria più leggibile e fruibile per una possibile estensione e riuso si è cercato durante lo sviluppo di mantenerla sufficientemente commentata. In particolare, è possibile generare la **ScalaDoc** del progetto digitando, internamente alla cartella principale, il comando

```
./gradlew scalaDoc
```

Essa inoltre è consultabile direttamente all'indirizzo **scala-honey-badger.surge.sh**.

Capitolo 5

Applicazione di test: Savanna

L'applicazione di test prende il nome di **Savanna**, cioè savana in inglese. Il nome è un chiaro riferimento all'habitat nel quale il tasso del miele vive prevalentemente.

Scopo di *Savanna* è quello di utilizzare la libreria *Honey Badger* mostrando all'utente, per mezzo di una chiara interfaccia grafica, l'avanzamento dell'algoritmo.

5.0.1 Architettura

Savanna è funzionalmente e graficamente molto semplice. A livello architetturale si compone di un model che fa utilizzo diretto della libreria e un'interfaccia grafica composta da due finestre: un pannello di controllo (*control panel*) e una *dashboard*. Sia i due componenti grafici, sia il model, sono implementati come attori Akka e comunicano tra loro attraverso messaggi.

Il punto di avvio di *Savanna* è il *control panel*, attraverso il quale l'utente può configurare i parametri dell'algoritmo e lanciarlo. Al lancio il *control panel* crea il model e la relativa *dashboard*. Attraverso la *dashboard* l'utente inserisce nuove transazioni e modifica lo stato dei nodi dell'*honey badger*. La *dashboard* cattura gli input dell'utente e aggiorna il model. Il model applica i cambiamenti agendo direttamente sui nodi dell'*honey badger* tramite i messaggi della libreria. Quando i nodi dell'*honey badger* danno in output un insieme di transazioni queste vengono ricevute dal model che provvede ad aggiornare il proprio stato e ad inviare le modifiche alla *dashboard*.

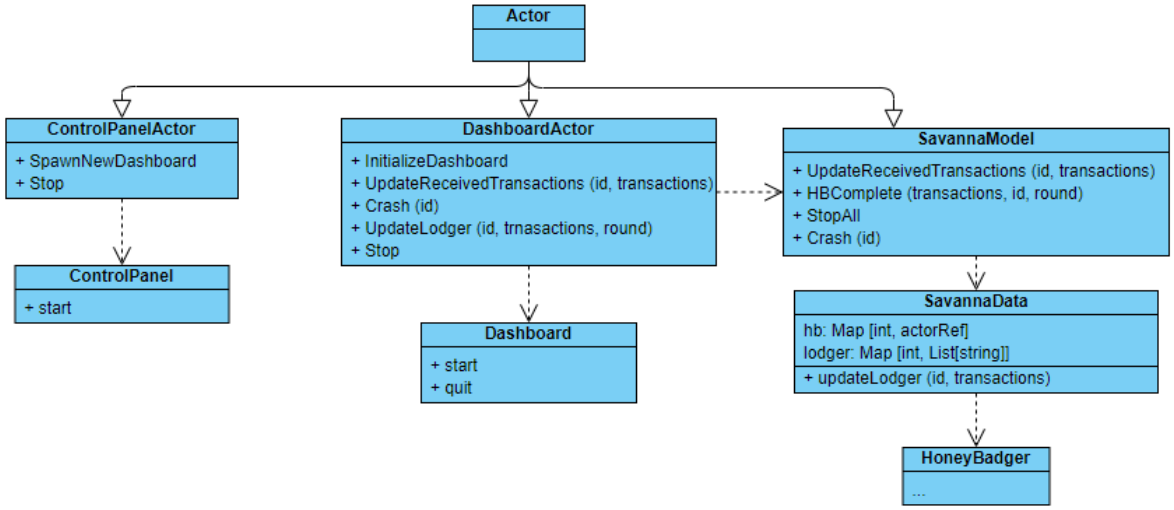

Figura 5.1: Diagramma UML delle classi dell'applicazione di test Savanna.

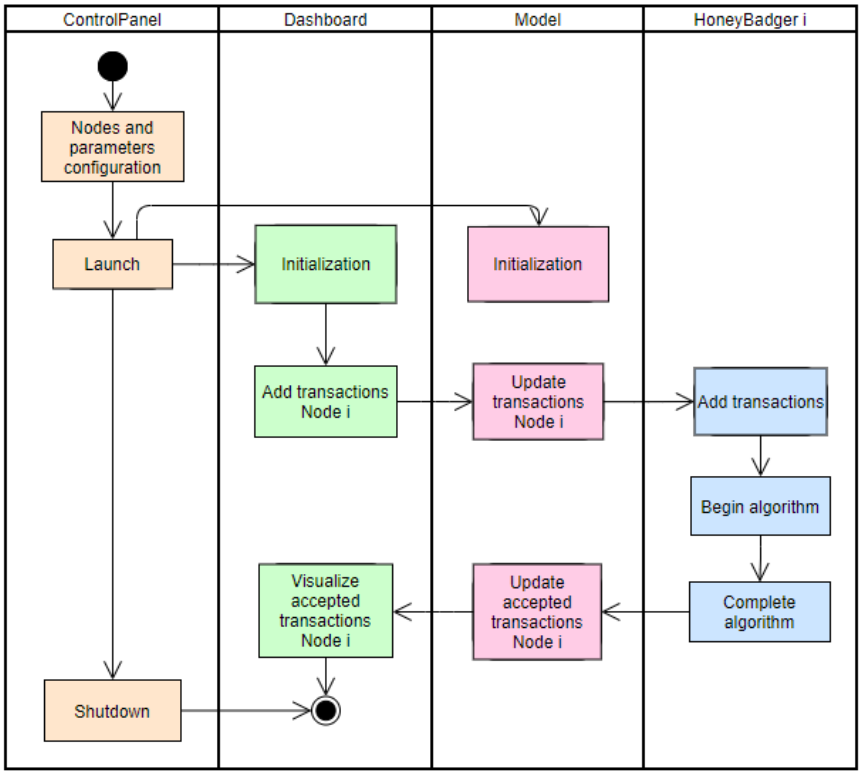

Figura 5.2: Diagramma UML delle attività dell'applicazione di test Savanna.

5.0.2 Funzionalità e Interfaccia Grafica

Avviata l'applicazione l'utente interagisce con il *control panel*. Il *control panel* (Figura 5.3) indica nella parte superiore all'utente quanti nodi di *honey badger* si stanno per instanziare.

Nella parte centrale sono visualizzati i singoli nodi e la loro tipologia, mentre in quella inferiore vengono riportati i valori di: ritardo di rete, *batch size* e numero di bizantini. I nodi bizantini si comportano esattamente come i nodi corretti con l'unica differenza di proporre sempre `false` ai propri processi di *binary agreement*. Ciò comporta una diminuzione delle transazioni approvate se il numero di bizantini è superiore ad f. Se i bizantini tendono a $2f$, la probabilità che le transazioni che i nodi corretti propongono è prossima a 0. Se il loro numero supera $2f$ nessuna transazione verrà mai approvata.
Dal *control panel* l'utente può:

- inserire un nuovo nodo per mezzo del bottone "+" in alto a sinistra;
- modificare lo stato di un nodo attraverso i bottoni "correct" e "byzantine";
- eliminare un nodo attraverso il bottone "-" (non può eliminare se i nodi sono 4);
- indicare un ritardo di rete attraverso i bottoni "+" e "-" a destra della corrispettiva label (il ritardo è compreso in un intervallo tra 0 e 2000 millisecondi con un passo di 100 millisecondi);
- indicare il valore della *batch size* attraverso i bottoni "+" e "-" a destra della corrispettiva label (non meno di 4);

- iniziare l'algoritmo attraverso il bottone "start" in basso a sinistra;
- terminare l'algoritmo attraverso il bottone "shutdown" a destra del bottone "start".

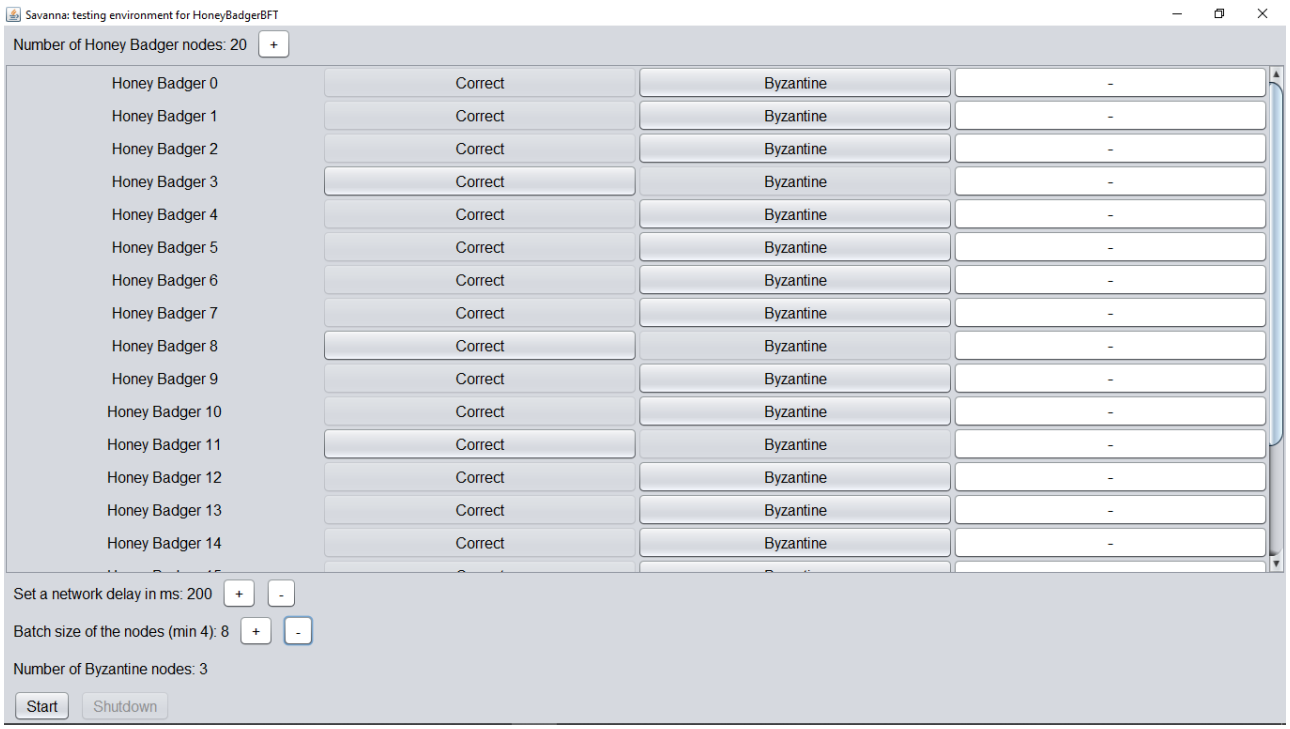

Figura 5.3: Schermata del pannello di controllo.

Quando si avvia l'algoritmo viene creata la corrispettiva *dashboard* (Figura 5.4) nella quale è possibile visualizzare lo stato di ogni nodo dell'*honey badger*.

La *dashboard*, nella sua parte centrale, contiene delle sotto-finestre riassuntive dello stato di uno specifico nodo dell'*honey badger*. Nella parte superiore di una sotto finestra è rappresentato l'identificativo del nodo assieme all'ultimo turno di algoritmo eseguito. Il colore verde scuro indica che il nodo è un nodo corretto. Il colore rosso significa che il nodo è un nodo bizantino, mentre il verde chiaro che il nodo è in *crash*. La parte centrale della sotto finestra è suddivisa in due aree. Quella di sinistra mostra le transazioni ricevute dal nodo e non ancora pubblicate, sopra ad essa è riportato il numero esatto. Quella di destra mostra le transazioni pubblicate, sopra ad essa è riportato il numero esatto. Questo ultimo numero deve essere il medesimo, assieme alle transazioni, per tutti i nodi corretti dello allo stesso turno.

Nella parte inferiore della *dashboard* sono riepilogati il numero totale di nodi, quelli bizantini, quelli in *crash* e il ritardo della rete.

Attraverso la *dashboard* l'utente può:

- Aggiungere transazioni autogenerate a tutti i nodi dell'*honey badger* attraverso il bottone "add transaction" in alto a sinistra;
- Cancellare dalla finestra le transazioni pubblicate fin ora per ogni nodo con il bottone "clear all" in alto a destra;
- Mandare in *crash* uno specifico nodo per mezzo del bottone "X" in alto a destra nella sotto-finestra relativa al nodo stesso;

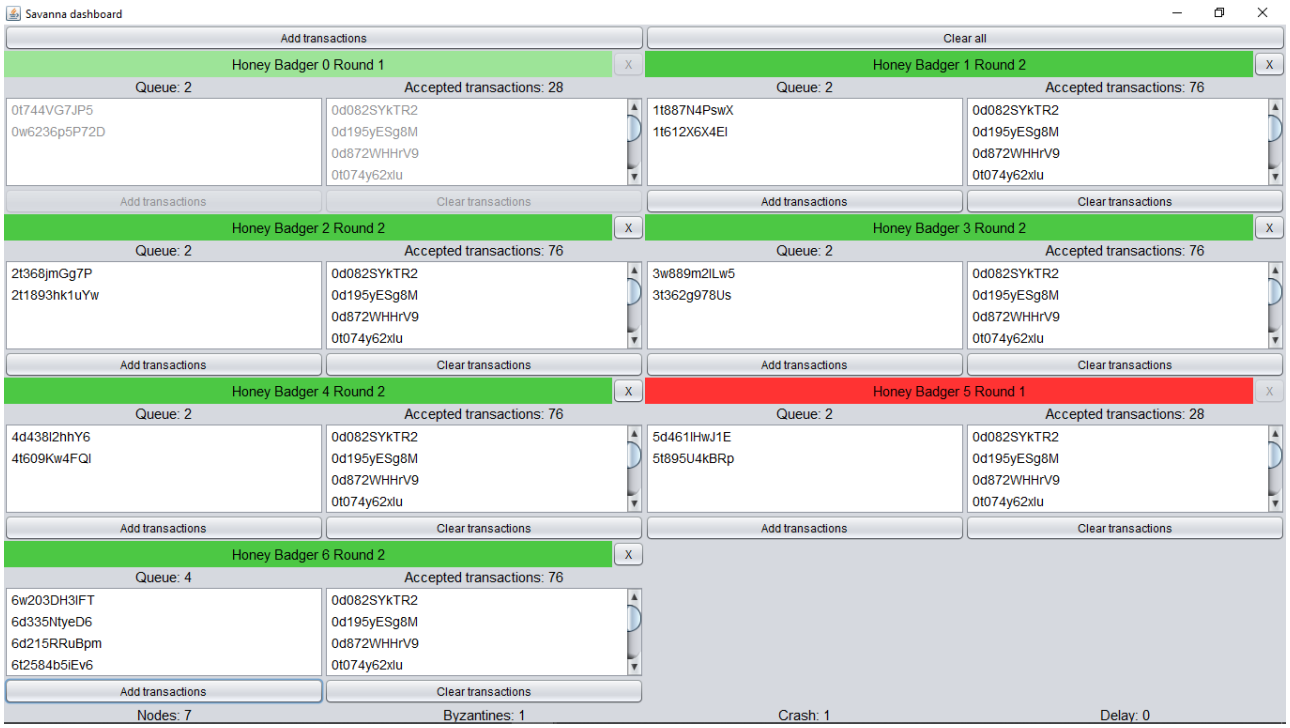

Figura 5.4: Schermata della dashboard.

- Aggiungere specifiche transazioni autogenerate in un unico nodo tramite il bottone “add transaction” in basso a sinistra nella sotto-finestra relativa al nodo stesso;
- Cancellare dalla finestra le transazioni pubblicate finora di un unico nodo con il bottone “clear all” in basso a destra nella sotto-finestra relativa al nodo stesso;

Non appena l’utente mette in *crash* più di f nodi l’algoritmo di *honey badger* è impossibilitato a compiere progressi. Se il numero di nodi in *crash* è inferiore ad f l’algoritmo continuerà comunque a procedere, salvo che il numero di bizantini non sia troppo elevato.

Nota: l’esecuzione dell’*Honey Badger* avviene tramite attori in esecuzione all’interno della stessa macchina. Poiché il numero di messaggi scambiati tra i diversi protocolli è quadratico rispetto al numero di nodi instanziati, si suggerisce di utilizzare numeri contenuti durante l’esecuzione dell’applicazione.

Capitolo 6

Validazione e test

I test ricoprono un ruolo di primaria importanza poiché devono verificare la correttezza delle proprietà dei singoli protocolli. Se anche solo un protocollo non rispetta una delle sue proprietà, l'intero *Honey Badger* non è in grado di funzionare in modo corretto.

Data la natura modulare dell'algoritmo, anche i test sono stati sviluppati in modo similare, rispecchiando l'architettura di *framework*. Ogni singolo test è mirato a verificare una particolare proprietà di un protocollo. La costruzione dei test ha avuto un approccio *bottom-up*, cioè dai protocolli di più basso livello a quelli di alto livello.

I test inerenti i protocolli di *Common Coin, Binary Agreement, Reliable Broadcast, Asynchronous Common Subset e Honey Badger* sono implementati come test **TestKit** di Akka. Essendo i processi che implementano i vari protocolli degli attori Akka, questa tipologia è la più idonea. Quelli riguardanti le funzioni crittografiche, invece, viene fatto uso solo di *scalatest*.

Il protocollo che realizza i *Common Coin* è non deterministico. Di conseguenza non si può sapere di quanti turni ha bisogno il protocollo di *Binary Agreement* per giungere al consenso. Questo implica che non è possibile stabilire a priori quanto tempo è necessario per il suo completamento. Se il protocollo è ben implementato è valida la proprietà di terminazione, prima o poi si giunge al consenso su un valore, ma non si sa quando. Ciò comporta l'impostazione di un tempo di *timeout* dei test che sia ragionevolmente alto per permettere il completamento, ma al contempo che non sia eccessivo poiché è necessario catturare possibili situazioni di stallo dovuti a bug.

L'introduzione di un tempo di *timeout* può far fallire dei test che potenzialmente non dovrebbero. Il fallimento può essere dovuto ad un effettivo bug, oppure ad un lancio sfortunato del test nel quale i processi hanno impiegato più tempo del dovuto per ottenere un risultato. Riuscire a distinguere le due situazioni è un compito difficile, possibile solo tramite l'introduzione di numerose stampe di controllo, poi successivamente rimosse terminata la fase di test.

Per rendere automatizzata la procedura di test inoltre è stata inserita una *pipeline* sul *repository* del progetto, in modo da avere sempre modo di controllare che ogni modifica apportata non causasse eventuali fallimenti di altri componenti. Data la forte interoperabilità dei moduli questo approccio è stato molto utile.

Durante lo sviluppo della libreria si è fatto utilizzo delle *pipeline* di GitLab. I test sono stati automaticamente eseguiti, attraverso i task di Gradle, ad ogni *push* sul *repository* del progetto.

6.1 Threshold Encryption

Il numero totale di test svolti è 18. Il test del *Threshold Encryption* prevede la verifica delle seguenti proprietà:

- Validità del crittogramma creato nella fase di criptaggio;
- Validità del *decrypt share* creato nella fase di decriptaggio;
- Uguaglianza tra il messaggio d'origine e il messaggio ottenuto combinando almeno t (valore soglia) differenti *decrypt share* del crittogramma del messaggio d'origine;
- Validità del *signature share* creato nella fase di firma;
- Validità della firma ottenuta dalla combinazione di almeno t *signature share* differenti;
- Uguaglianza tra firme ottenute combinando *signature share* di nodi differenti.

Sono stati svolti anche dei test per verificare l'invalidità delle firme e dei crittogrammi ottenuti con un numero di *share* inferiore rispetto alla soglia, ma sono stati messi sotto commento perché attualmente all'interno del codice è presente un controllo che evita di combinare gli *share* in tali situazioni.

Package	Class, %	Method, %	Line, %
honey_badger.threshold_encryption	61,9% (13/ 21)	80,7% (67/ 83)	86,9% (106/ 122)

Class ▲	Class, %	Method, %	Line, %
CipherText	50% (1/ 2)	50% (1/ 2)	50% (1/ 2)
CipherTextSerializable	50% (1/ 2)	50% (1/ 2)	50% (1/ 2)
CryptoSystem	66,7% (2/ 3)	80% (20/ 25)	87,2% (34/ 39)
MsgLength	100% (1/ 1)	100% (3/ 3)	100% (3/ 3)
PrivateKey	50% (1/ 2)	80% (4/ 5)	80% (4/ 5)
PublicKey	50% (1/ 2)	95,2% (20/ 21)	97,4% (38/ 39)
ScalaPBC	100% (5/ 5)	81% (17/ 21)	85,7% (24/ 28)
Share	50% (1/ 2)	50% (1/ 2)	50% (1/ 2)
ShareSerializable	0% (0/ 2)	0% (0/ 2)	0% (0/ 2)

Figura 6.1: Coverage dei test effettuata tramite IntelliJ per la Threshold Encryption.

6.2 Router

Il numero di test è di 6. Il test del *Router* prevede la verifica delle seguenti proprietà:

- Tutti i processi registrati ricevono immediatamente i messaggi di tipo `Broadcast`, quelli che sono in ritardo li ricevono non appena si registrano;
- Tutti i processi registrati ricevono immediatamente i messaggi singoli a loro indirizzati, quelli che sono in ritardo li ricevono non appena si registrano;
- Verifica delle due precedenti proprietà contemporaneamente.

Package	Class, %	Method, %	Line, %
honey_badger.utility	48,6% (36/ 74)	53,9% (55/ 102)	56,8% (117/ 206)

Class ▲	Class, %	Method, %	Line, %
Configuration	60% (3/ 5)	47,4% (9/ 19)	44,7% (42/ 94)
MessageInstance	43,8% (28/ 64)	43,8% (28/ 64)	43,8% (28/ 64)
Router	100% (3/ 3)	100% (9/ 9)	100% (27/ 27)
RouterData	100% (2/ 2)	90% (9/ 10)	95,2% (20/ 21)

Figura 6.2: Coverage dei test effettuata tramite IntelliJ per il Router.

6.3 Binary Agreement

6.3.1 Common Coin

Il corretto funzionamento del *Common Coin*, oltre che dalla corretta implementazione del relativo attore, dipende a sua volta dal corretto funzionamento dei test crittografici e del *router*. Il numero totale di test è di 7. Il test del *Common Coin* prevede la verifica delle seguenti proprietà:

- terminazione con stesso valore con n processi corretti;
- terminazione con stesso valore se vengono effettuate almeno $f + 1$ chiamate `getCoin`;
- non terminazione se vengono effettuate meno di $f + 1$ chiamate `getCoin`.

6.3.2 Binary Agreement

Il corretto funzionamento del *Binary Agreement* dipende a sua volta dal corretto funzionamento del *Common Coin* e del *router*. Il numero totale di test è di 22. Il test del *Binary Agreement* prevede la verifica delle seguenti proprietà:

- raggiunta del consenso con n processi corretti;
- raggiunta del consenso con n processi corretti con $n - f$ chiamate `Propose`;
- raggiunta del consenso con valore `true` se meno di $f + 1$ processi hanno proposto `false`;
- non terminazione se vengono effettuate meno di $f + 1$ chiamate `Propose`;
- terminare con un qualsiasi valore se vengono effettuate almeno $f + 1$ chiamate `Propose` di entrambi i valori;
- terminazione corretta di tutti i processi anche se f processi ritardatari iniziano il consenso quando gli altri $n - f$ hanno già terminato;
- terminazione corretta in presenza di f *crash*.

Package	Class, %	Method, %	Line, %
honey_badger.asynchronous_common_subset.binary_agreement	100% (13/ 13)	98,3% (57/ 58)	97% (161/ 166)

Class	Class, %	Method, %	Line, %
BinaryAgreement	100% (5/ 5)	100% (11/ 11)	95,6% (65/ 68)
BinaryAgreementData	100% (2/ 2)	95,5% (21/ 22)	97,5% (39/ 40)
CommonCoin	100% (4/ 4)	100% (10/ 10)	100% (33/ 33)
CommonCoinData	100% (2/ 2)	100% (15/ 15)	96% (24/ 25)

Figura 6.3: Coverage dei test effettuata tramite IntelliJ per il Binary Agreement.

6.4 Reliable Broadcast

L'algoritmo di *Reliable Broadcast* si basa sul corretto funzionamento dei due algoritmi su cui fa affidamento: *Merke Tree* e *Reed Solomon encoding*. I test sviluppati (11 in tutto) pertanto verificano dapprima tali protocolli e poi il *broadcast* completo, in particolare:

- Corretto funzionamento in caso di totalità dei nodi corretti;
- Corretto funzionamento in caso di presenza di f *crash*;
- Non terminazione in presenza di $f + 1$ *crash* (ricostruzione valore corretto grazie al *Reed Sollomon decoding*);
- Corretto funzionamento in caso di presenza di f nodi bizantini che modificano il valore distribuito agli altri nodi (valori scorretti scartati grazie alla *Merke Tree verification*);
- Non terminazione in presenza di $f + 1$ nodi *faulty*.

Package	Class, %	Method, %	Line, %
honey_badger.asynchronous_common_subset.reliable_broadcast	78,6% (11/ 14)	96% (72/ 75)	93% (172/ 185)

Class	Class, %	Method, %	Line, %
ReliableBroadcast	100% (5/ 5)	100% (21/ 21)	88,4% (76/ 86)
ReliableBroadcastData	100% (2/ 2)	100% (15/ 15)	100% (16/ 16)
ReliableBroadcastProtocols	57,1% (4/ 7)	92,3% (36/ 39)	96,4% (80/ 83)

Figura 6.4: Coverage dei test effettuata tramite IntelliJ per il Reliable Broadcast.

6.5 Asynchronous Common Subset

Il corretto funzionamento del *Asynchronous Common Subset* dipende a sua volta dal corretto funzionamento del *Binary Agreement*, del *Reliable Broadcast* e del *router*. Il numero totale di test è di 10. Il test del *Asynchronous Common Subset* prevede la verifica delle seguenti proprietà:

- Terminazione corretta con n processi corretti;
- terminazione corretta di tutti i processi anche se f processi ritardatari iniziano il consenso quando gli altri $n - f$ hanno già terminato;
- terminazione corretta in presenza di f *crash*.

Package	Class, %	Method, %	Line, %
honey_badger.asynchronous_common_subset	85,7% (6/ 7)	86,8% (33/ 38)	86,3% (82/ 95)

Class	Class, %	Method, %	Line, %
AsynchronousCommonSubset	80% (4/ 5)	75% (15/ 20)	82,4% (61/ 74)
AsynchronousCommonSubsetData	100% (2/ 2)	100% (18/ 18)	100% (21/ 21)

Figura 6.5: Coverage dei test effettuata tramite IntelliJ per l'Asynchronous Common Subset.

6.6 Honey Badger

Il corretto funzionamento di *Honey Badger* dipende a sua volta dal corretto funzionamento del *Asynchronous Common Subset* e del *router*. Il numero totale di test è di 13. Il test del *Honey Badger* prevede la verifica delle seguenti proprietà:

- Terminazione corretta con n processi corretti;
- Terminazione corretta con n processi corretti nel successivo turno dell'algoritmo;
- Terminazione corretta di tutti i processi in presenza di input differente, i processi pubblicano comunque lo stesso insieme di transazioni;
- Terminazione corretta con $n - f$ processi corretti e f ritardatari;
- Terminazione corretta anche in presenza d f processi bizantini che propongono sempre `false` in tutti i sotto sistemi di *Binary Agreement*;
- terminazione corretta in presenza di f *crash*.

Package	Class, %	Method, %	Line, %
honey_badger	100% (6/ 6)	97,1% (34/ 35)	93,6% (88/ 94)

Class	Class, %	Method, %	Line, %
HoneyBadger	100% (4/ 4)	94,1% (16/ 17)	92% (69/ 75)
HoneyBadgerData	100% (2/ 2)	100% (18/ 18)	100% (19/ 19)

Figura 6.6: *Coverage* dei test effettuata tramite IntelliJ per l'Honey Badger.

Il numero totale di test di tutta la libreria è di 87.

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	63,7% (86/ 135)	82,1% (321/ 391)	85,5% (743/ 869)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
honey_badger	100% (6/ 6)	97,1% (34/ 35)	93,6% (88/ 94)
honey_badger.asynchronous_common_subset	85,7% (6/ 7)	86,8% (33/ 38)	86,3% (82/ 95)
honey_badger.asynchronous_common_subset.binary_agreement	100% (13/ 13)	98,3% (57/ 58)	97% (161/ 166)
honey_badger.asynchronous_common_subset.reliable_broadcast	78,6% (11/ 14)	96% (72/ 75)	93,5% (173/ 185)
honey_badger.threshold_encryption	61,9% (13/ 21)	80,7% (67/ 83)	86,9% (106/ 122)
honey_badger.utility	50% (37/ 74)	56,9% (58/ 102)	64,3% (133/ 207)

Figura 6.7: *Coverage* globale dei test generata tramite IntelliJ.

Capitolo 7

Istruzioni per il deployment

La corretta compilazione ed esecuzione del progetto necessita l'installazione di `Scala 2.12.8` o versioni superiori.

Per permettere un facile *deployment* si è fatto uso di **gradle**:

- Per fare la *build* del progetto e lanciare i test, dopo aver clonato la *repository* di GitLab, è necessario posizionarsi all'interno della directory principale del progetto e lanciare il comando

  ```
  ./gradlew
  ```

- Per lanciare invece l'applicazione di test è necessario sempre dalla directory principale lanciare il comando

  ```
  ./gradlew app
  ```

  e dall'interfaccia che si aprirà sarà possibile selezionare il numero e il tipo di nodi della rete e impostare eventuali ritardi di rete.

Capitolo 8

Conclusioni

In questo report è stato presentato il problema del consenso facendo particolare riferimento ai vincoli e assunzioni su cui la sua esecuzione si basa. Si è poi descritto in modo molto dettagliato l'algoritmo *HoneyBadgerBFT*, dalle condizioni dei sistemi in cui è in grado di operare al funzionamento teorico dei singoli protocolli di cui si compone. Infine si è fornita una sua implementazione sotto forma di libreria Scala mediante l'utilizzo del *framework* Akka. Si è inoltre sviluppata una semplice applicazione per l'utilizzo della libreria.

Tutto questo lavoro, oltre allo studio dell'algoritmo mediante numerosi articoli, è stato molto impegnativo, richiedendo più tempo ed energie di quanto inizialmente immaginato. Il lavoro è stato suddiviso come segue:

- Studio dell'algoritmo, ogni membro del gruppo ha studiato autonomamente l'intero algoritmo attraverso l'articolo originale [9] per circa due settimane. Lo studio è poi proseguito individualmente in modo più dettagliato per i protocolli in base alla suddivisione iniziale che ci eravamo prefissati;

- Implementazione di *HoneyBadgerBFT*, ha richiesto la maggior parte del tempo dedicato all'intero progetto, in particolare la fase di *debug*. Inizialmente si sono sviluppati i protocolli più a basso livello:

  – Giannini Andrea, *threshold encryption*;
  – Guiducci Linda, *reliable broadcast*;
  – Magnini Matteo, *binary agreement* con *common coin* e il *router*.

  Successivamente, visto il tempo necessario per la *threshold encryption*, Guiducci e Magnini hanno iniziato il protocollo di *Asynchronous Common Subset*. L'*honey badger* e il *debug* sono stati fatti da tutti i membri. Per quanto riguarda l'applicazione di test *Savanna*, il *model* è stato implementato principalmente da Giannini, mentre la *view* da Guiducci e Magnini.

- Fase di test, i test della libreria per i singoli protocolli sono stati scritti rispettivamente da chi li aveva in carico. Tuttavia, vista l'alta complessità e importanza che ricoprono, tutti i membri hanno fornito un contributo trasversale.

8.1 Lavori futuri

L'attuale implementazione presenta alcuni limiti che potranno essere rimossi in futuro. Il limite maggiore è quello legato alla lunghezza prefissata delle transazioni. A livello applicativo il

limite di lunghezza non è un problema poiché un messaggio troppo lungo può essere scomposto in più messaggi logicamente collegati tra loro. Tuttavia a livello di *batching* ciò influisce sulle prestazioni. La soluzione consiste nell'utilizzare delle primitive crittografiche differenti.

Il secondo limite è l'utilizzo e l'assenza di autenticazione nei *router*. I *router* sono solo un espediente comodo per simulare una comunicazione persistente ed affidabile tra gli attori. Per questo motivo un utilizzo diretto di questa libreria in applicazioni che non siano a scopo dimostrativo o di ricerca è sconsigliato.

Si potrebbe infine aggiungere un protocollo di *Distributed Key Generation* che non richiede un *trusted dealer* iniziale per la distribuzione delle chiavi, ma che permette ai nodi di creare una chiave segreta collettivamente.

8.2 Cosa abbiamo imparato

- Nel corso dell'intero progetto i membri del gruppo hanno approfondito le loro conoscenze sul linguaggio Scala e sul *framework* Akka;
- Si è appreso in modo approfondito un recente e complesso algoritmo di consenso;
- Si sono potenziate *soft skill* quali:
  - predisposizione a lavorare in gruppo;
  - *management* del carico di lavoro;
  - progettazione da remoto causa coronavirus.

8.3 Note stilistiche

- Il *corsivo* è stato utilizzato per termini di lingua inglese non di uso comune nella lingua italiana (es. input e output non in corsivo ma *batch size* e *framework* sì);
- L'*enfasi* è stata usata su termini a cui si voleva dare attenzione nel discorso;
- Il **grassetto** è stato usato negli elenchi puntati quando le prime parole fanno riferimento al nome proprio di una proprietà o di un algoritmo. È stato usato anche per termini a cui si voleva dare particolare risalto nel discorso;
- Il `monospace` è stato usato per indicare termini che fanno riferimento a variabili o metodi in pseudocodice o diagrammi;
- La notazione matematica è stata usata per formule e valori;
- Nelle *caption* delle immagini e tabelle non sono applicati alteratori;
- In generale si è preferito usare il termine inglese, anche se una sua traduzione in italiano è esistente, per una maggiore aderenza alla letteratura che è in lingua inglese.

Bibliografia

[1] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. *Secure and Efficient Asynchronous Broadcast Protocols*, volume 2139, pages 524–541. Springer, advances in cryptology — crypto 2001. crypto 2001. lecture notes in computer science edition, 2001.

[2] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the internet. June 2002.

[3] Angelo De Caro and Vincenzo Iovino. jPBC: Java pairing based cryptography. In *Proceedings of the 16th IEEE Symposium on Computers and Communications, ISCC 2011*, pages 850–855, Kerkyra, Corfu, Greece, June 28 - July 1, 2011. IEEE.

[4] Fischer, Michael J., Lynch, Nancy A., Paterson, and Michael S. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[5] Coulouris George, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design*, page 660. Addison-Wesley, 5rd edition, 2001.

[6] Lamport Leslie, Shostak Robert, and Pease Marshall. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[7] Ben Lynn. PBC Library: The pairing-based cryptography library. `https://crypto.stanford.edu/pbc/`.

[8] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. Ccs 2016 - the honey badger of bft protocols. `https://www.youtube.com/watch?v=Qone4j1hCt8`.

[9] Andrew Miller, Yu Xia, Kyle Croman, Elaine Shi, and Dawn Song. The honey badger of bft protocols. 2016.

[10] Achour Mostéfaoui, Hamouma Moumen, and Michel Raynal. Signature-free asynchronous byzantine consensus with $t < n/3$ and $o(n^2)$ messages. In *PODC '14: Proceedings of the 2014 ACM symposium on Principles of distributed computing (Paris, France, July 2014)*, pages 2–9, New York, NY, United States, 2014. Association for Computing Machinery.

[11] Schneider and Fred B. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.