Strategie di Elezione del Leader per la Coerenza e l'Affidabilità nei Sistemi Cloud Distribuiti: Analisi degli Algoritmi ad Anello e di Raft

Pierfrancesco Lijoi
Magistrale Ingegneria Informatica
Anno Accademico 2023-2024
Roma, Italia
pierfrancesco.lijoi@libero.it

***Abstract* —**Nei sistemi cloud distribuiti, l'elezione del leader riveste un ruolo cruciale per garantire la coerenza e l'affidabilità delle operazioni. Due degli algoritmi tra i più utilizzati per questo scopo, sono l'algoritmo di elezione ad anello e l'algoritmo di elezione nell'algoritmo del consenso di Raft. Nel seguente articolo, verranno esaminati più approfonditamente entrambi gli algoritmi, analizzandone le caratteristiche, le prestazioni e le considerazioni di implementazione. La comprensione di questi algoritmi è essenziale per progettare e gestire sistemi cloud distribuiti affidabili e scalabili.

Parole chiave: *elezione leader, algoritmo di elezione ad anello, sistemi distribuiti, progettazione, scalabilità, robustezza, performance.*

I. Introduzione

Nel contesto sempre più complesso dei sistemi cloud distribuiti, l'elezione del leader emerge come un processo fondamentale per garantire la stabilità e l'affidabilità delle operazioni. Questo processo, che coinvolge la selezione di un nodo principale tra i partecipanti del sistema, riveste un ruolo critico nel coordinare le attività e nell'assicurare la coerenza delle operazioni all'interno dell'infrastruttura distribuita. L'importanza dell'elezione del leader è particolarmente evidente in ambienti in cui la distribuzione su larga scala di risorse computazionali richiede una gestione efficiente e resiliente. Due degli algoritmi più significativi utilizzati per questo scopo sono, l' algoritmo di elezione ad anello e l'algoritmo di elezione di Raft. L'elezione ad anello, basata su una topologia circolare dei nodi, offre una soluzione scalabile e robusta per determinare il leader del sistema. L'algoritmo di consenso di Raft, al contrario, si basa sull'idea di un'elezione del leader per stabilire la leadership all'interno di un cluster distribuito. Il processo di elezione del leader è una delle fasi cruciali dell'algoritmo di Raft, che mira a garantire un consenso affidabile tra i nodi distribuiti per il corretto funzionamento del sistema.

II. Condizioni Necessarie per l'Utilizzo degli Algortimi di Elezione

A. Condizioni Necessarie per l'Algoritmo di Elezione ad Anello

Prima di avviare l'algoritmo di elezione ad anello, è fondamentale garantire che i nodi del sistema siano organizzati in una topologia logica di forma circolare. Tale topologia permette il corretto fluire dei messaggi di elezione tra i nodi del cluster. È possibile identificare in modo chiaro e immediatamente percepibile i seguenti aspetti necessari per la corretta esecuzione e funzionamento:

A.1. Topologia dell'Anello:
L'anello logico deve essere costituito da nodi che mantengano una comunicazione affidabile tra di loro. È necessario che ogni nodo abbia almeno la conoscenza del suo diretto successivo nell'anello per garantire la corretta circolazione dei messaggi di elezione. Tuttavia, non è essenziale che ogni nodo abbia una conoscenza completa di tutti i nodi presenti nella struttura circolare.

A.2. Identificazione Unica dei Nodi:
Ogni nodo del sistema distribuito deve essere identificato in maniera univoca mediante un proprio codice identificativo. Questo assicura che ogni nodo possa essere distinguibile all'interno del cluster e sia possibile garantire l'unicità del leader.

A.3. Servizio di Registrazione
Per consentire ad un nodo di entrare nell' overlay ad anello, è necessario introdurre un elemento di centralizzazione, che potrà essere esterno alla struttura logica oppure interno, ovvero che possa essere un nodo già presente nella rete. In ogni caso va fornito un punto di accesso sempre presente ed attivo per permettere al nodo di inserirsi nell'anello nella rispettiva posizione.

A.4. Costruzione della Topologia ad Anello
Tale struttura può essere realizzata applicando un ordinamento dei nodi in base ai rispettivi codici identificativi univoci. Tale condizione non è necessaria per la corretta esecuzione dell'algoritmo, perché la struttura logica può essere creata in base anche all'ordinamento temporale spontaneo con cui i nodi sono entrati a far parte della struttura logica ad anello. La struttura ad anello deve avere una cardinalità maggiore di uno per poter essere applicata in un contesto distribuito, se quest'ultima condizione non viene rispettata, non si necessita dell'utilizzo dell'algoritmo poiché non è applicato in un contesto distribuito.

A.5. Comunicazione
Per garantire la corretta esecuzione dell'algoritmo e che le proprietà di Liveness e Safety vengano rispettate, è fondamentale definire una comunicazione affidabile tra i nodi monodirezionale uguale per ogni componente dell'anello logico. Tale condizione consentirà il fluire della comunicazione in un sola direzione, che permetterà all'algoritmo di convergere verso l'elezione del singolo

leader. Nel contesto specifico l'assunzione di Liveness e Safety assumono i seguenti significati:

- Liveness: soltanto un nodo, non guasto, con più alto codice identificativo univoco, sarà eletto come leader
- Safety: l'algoritmo, se tutte le condizioni inziali verranno rispettate, convergerà all'elezione del singolo leader.

A.6. Gestione dei nodi guasti
Durante il processo di elezione o comunicazione del leader eletto, nel flusso comunicativo a anello, se un componente deve trasmettere informazioni al suo nodo successivo e quest'ultimo è guasto, sarà necessario stabilire una comunicazione con il nodo immediatamente successivo al nodo guasto, utilizzando le informazioni precedentemente acquisite dal nodo stesso riguardo alla configurazione dell'overlay circolare. Non è necessario che il nodo fallito comporti un aggiornamento della conoscenza dei nodi sulla struttura logica, può essere scelto di mantenere le sue informazioni in memoria, ignorando quindi il fallimento temporaneo del nodo considerato, oppure di eseguire un aggiornamento delle stesse e scartare le sue informazioni per essere contattato.

A.7. L'Obiettivo
Lo scopo dell'elezione è eleggere un nodo appartenente alla rete logica ad anello non guasto con il rispettivo valore più alto del codice identificativo. Quindi, con tale condizione, il risultato non dipende da chi avvierà il processo di elezione ,poiché più processi di elezione forniranno lo stesso risultato del leader eletto.

B. Condizioni Necessarie per l' Algoritmo di Elezione in Raft

Prima di avviare l'algoritmo di elezione di Raft, è fondamentale garantire la definizione di un cluster di nodi, tra cui avverrà l'algoritmo di elezione del leader. È possibile identificare in modo chiaro e immediatamente percepibile i seguenti aspetti necessari per la corretta esecuzione e funzionamento:

B.1. Composizione del Cluster:
È necessario definire l'insieme dei nodi che dovranno eleggere e far riferimento al leader eletto all'interno del cluster. Il cluster deve avere una cardinalità maggiore di uno per poter essere applicata in un contesto distribuito. Se quest'ultima condizione non viene rispettata, non si necessità dell'utilizzo dell'algoritmo poiché non è applicato in un contesto distribuito.

B.2. Identificazione Unica dei Nodi:
Ogni nodo del sistema distribuito deve essere identificato in maniera univoca mediante un proprio codice identificativo. Questo, assicura che ogni nodo possa essere distinguibile all'interno del cluster e sia possibile garantire l'unicità del leader.

B.3. Servizio di Registrazione
Per consentire ad un nodo di entrare nel cluster dei nodi, è necessario introdurre un elemento di centralizzazione, che potrà essere esterno alla struttura logica oppure interno, ovvero che potrà essere un nodo già presente nella rete. In ogni caso va fornito un punto di accesso sempre presente ed attivo per permettere al nodo di inserirsi nell'anello nella rispettiva posizione.

B.4. Timer Randomici
Il nodo dovrà definire due timer randomici. Il primo timer, denominato "Timer di elezione", verrà utilizzato per attendere i messaggi di Heart beat da parte del leader, inviati per notificare il nodo che è ancora attivo. Inoltre, sarà utilizzato anche per definire il tempo massimo della durata dell'elezione avviata da un nodo candidato per essere il leader. Il secondo timer viene denominato come "timer Heart beat" poiché è il timer con il quale il leader definisce la frequenza di invio del messaggio di Heart beat per notificare a tutti gli altri nodi presenti nel cluster che è ancora in funzione.

B.5. Comunicazione
Per garantire la corretta esecuzione dell'algoritmo e che le proprietà di Liveness e Safety vengano rispettate, è fondamentale definire una comunicazione affidabile tra i nodi per ogni componente del cluster di nodi. Tale condizione consentirà la convergenza dell'algoritmo e un elezione del leader. Nel contesto specifico l'assunzione di Liveness e Safety assumono i seguenti significati:

- Liveness: soltanto un nodo, non guasto, con più alto codice identificativo univoco, sarà eletto come leader
- Safety: l'algoritmo, se tutte le condizioni inziali verranno rispettate, convergerà all'elezione del singolo leader.

B.6. Meccanismo di voto
Tutti i nodi del cluster avranno lo stesso valore inziale del turno di elezione, definito "Term". Ogni nodo, potrà votare per l'elezione di un leader una sola volta durante lo stesso term. Ogni nuovo processo di elezione, comporterà l'incremento del turno di elezione, in tal modo viene garantito che solo un leader venga eletto.

B.7. Gestione dei nodi guasti
È necessario considerare e prevedere la possibilità che un nodo subisca un fallimento. Tale situazione, può essere ignorata dagli altri nodi proseguendo con la comunicazione dei messaggi agli altri nodi ancora attivi oppure si può gestire un meccanismo di propagazione del fallimento del nodo così che tutti i componenti del cluster eliminino le informazioni inerenti ad esso.

B.8. L'Obiettivo
Lo scopo dell'elezione è eleggere un nodo appartenente al cluster, non guasto, che abbia ottenuto più della metà dei voti nello stesso turno di elezione per essere eletto leader. Quindi, con tale condizione, il risultato dell'elezione di un leader sarà sempre univoco.

III. Descrizione del Funzionamento degli Algortimi di Elezione

La descrizione degli algoritmi di elezione rispetterà scrupolosamente le condizioni necessarie precedentemente delineate. In particolare, se si è stabilita la possibilità di vari comportamenti in risposta a una specifica condizione, sarà definito un comportamento inequivocabilmente chiaro e coerente che si armonizza senza conflitti con le altre condizioni stabilite.

A. Descrizione dell'Algorimo di Elezione ad *Anello*

Nel contesto dei sistemi distribuiti, l'algoritmo di elezione del leader ad anello inizia con la registrazione dei nodi al server di registrazione, esterno alla struttura nodale, incaricato di registrare la presenza dei nodi all'interno dell'anello. Dopo aver raccolto le informazioni di registrazione del nuovo nodo, il server le organizzerà in base al loro codice identificativo e le trasmetterà al nuovo nodo e agli altri nodi già presenti nella struttura, garantendo così una lista sempre aggiornata. Il server provvederà a inviare aggiornamenti sulla presenza dei nodi dopo ciascuna nuova registrazione, assicurando che ogni nodo abbia una conoscenza completa e aggiornata degli altri componenti della rete logica strutturata. La struttura dell'anello sarà definita dall'ordinamento dei codici identificativi dei nodi eseguito dal server di registrazione. L'algoritmo di elezione ad anello viene attivato quando un nodo rileva il fallimento del leader corrente. In tal caso, il nodo avvia un processo di elezione inviando un messaggio al suo nodo successivo seguendo il flusso monodirezionale stabilito nella struttura dell'anello. Se il nodo destinatario del messaggio è inattivo a causa di un guasto, il mittente contatterà direttamente il successivo nella sequenza, continuando finché non trova un nodo attivo. Se tutti i nodi risultano inattivi e l'unico nodo attivo è lo stesso che ha avviato l'elezione, non si rispetta la condizione necessaria di avere un sistema distribuito. Il messaggio di elezione contiene i codici identificativi univoci di tutti i nodi che partecipano all'elezione del leader. L'elezione termina quando il nodo che ha avviato il processo trova il proprio codice identificativo nel messaggio, indicando che il messaggio ha completato un giro completo nell'anello. Il nodo eleggerà come leader il nodo con il codice identificativo più elevato all'interno del messaggio. Una volta eletto, il nuovo leader informerà tutti gli altri nodi presenti nell'anello del risultato dell'elezione, sempre seguendo il flusso monodirezionale della struttura ad anello. In caso di guasto del leader, l'algoritmo si riattiverà affinché l'intera procedura possa essere ripetuta. Inoltre, questo algoritmo non è in grado di gestire la partizione di rete.

Per valutare il costo computazionale di un algoritmo, è necessario considerare diverse operazioni e il loro impatto sulle risorse del sistema. Nel caso dell'algoritmo di elezione ad anello, il costo computazionale è principalmente determinato dall'invio e dalla ricezione dei messaggi tra i nodi. Supponendo che ogni nodo debba inviare un messaggio a ogni altro nodo nell'anello per avviare il processo di elezione, abbiamo un totale di N messaggi inviati. Inoltre, per ciascun messaggio inviato, il mittente deve aspettare una risposta o determinare se il nodo successivo è inattivo, il che richiede un ulteriore scambio di messaggi. Pertanto, si riscontra un totale di circa 2N messaggi scambiati per l'intero processo di elezione. Si può concludere , che il costo di computazione dell'algoritmo del contesto osservato è dell'ordine di O(2N) messaggi.

Tuttavia, va notato che questo è un costo asintotico e potrebbe variare a seconda dell'implementazione specifica e delle caratteristiche della rete. Ad esempio, se la rete ha ritardi o perdite di pacchetti, potrebbe essere necessario un maggiore numero di scambi di messaggi per completare con successo il processo di elezione. Inoltre, fattori come la latenza di rete e la capacità dei nodi di elaborare i messaggi influenzeranno il tempo complessivo richiesto per l'algoritmo.

B. Descrizione dell'Algorimo di Elezione in Raft

L'algoritmo di elezione del leader in Raft, fondamentale per la coerenza e l'efficienza dei sistemi distribuiti, si basa su un processo accurato e distribuito che coinvolge l'intero cluster di nodi. Prima di entrare in dettaglio sull'algoritmo stesso, è importante comprendere il contesto in cui opera. Ogni nodo, al momento della sua attivazione, stabilisce una connessione con un server di registrazione esterno che funge da registro centrale per tutti i nodi nell'anello. Questo server raccoglie e organizza le informazioni di registrazione dei nodi, garantendo che ogni nodo abbia una visione completa e aggiornata della struttura dell'anello.

Nell'algoritmo di elezione di Raft, ogni nodo può trovarsi in uno dei seguenti tre stati: candidato , follower o leader. Lo stato "candidato" si verifica quando un nodo, non ricevendo l'heartbeat dal leader entro un intervallo di tempo casuale, presume che il leader sia fallito e inizia un processo di elezione. Durante questo processo, il nodo candidato invia messaggi di voto agli altri nodi e, se riesce a ottenere la maggioranza dei voti, diventa il nuovo leader. Tuttavia, se nessun nodo riceve la maggioranza dei voti, o se un altro nodo vince l'elezione, il nodo candidato può rimanere in questo stato e avviare un nuovo processo di elezione.

Lo stato "follower" è quello in cui il nodo è un seguace del leader e attende passivamente l'heartbeat dal leader corrente. Se il nodo non riceve l'heartbeat entro un certo intervallo di tempo, passa nuovamente allo stato candidato per avviare un nuovo processo di elezione. Infine, lo stato di leader è raggiunto quando un nodo, mentre è in stato candidato e ha ottenuto la maggioranza dei voti, diventa il nuovo leader. Una volta eletto, il leader genera periodicamente un heartbeat che invia a tutti i nodi follower nell'anello, consentendo loro di conoscere il leader attuale e mantenere la coerenza nel sistema distribuito.

Per garantire l'integrità del processo di elezione e prevenire la formazione di partizioni di rete, ogni nodo nel cluster mantiene un valore chiamato "Term", che indica il turno corrente dell'elezione. Ogni volta che viene avviato un processo di elezione, il Term viene incrementato e ogni nodo può votare una sola volta durante il proprio Term. Inoltre, il Term è incluso nell'heartbeat inviato dal leader, consentendo ai nodi di identificare il leader corrente e garantire la coerenza nel cluster. Questo approccio offre vantaggi significativi in termini di coerenza e disponibilità del sistema, consentendo ai nodi di eleggere un nuovo leader in modo affidabile e prevenire la formazione di partizioni di rete. Tuttavia, è importante considerare anche i costi associati a questo algoritmo; in particolare l'overhead di comunicazione generato dai messaggi di elezione e heartbeat, che possono aumentare in caso di grandi cluster con molti nodi. Inoltre, l'algoritmo può essere influenzato da nodi "bizantini", che potrebbero comportarsi in modo non corretto durante il processo di elezione e compromettere la coerenza del

sistema. Nonostante questi svantaggi potenziali, l'algoritmo di elezione del leader in Raft rimane una scelta popolare per la sua semplicità e affidabilità nel garantire la coerenza nei sistemi distribuiti.
Per quanto riguarda il costo computazionale, l'algoritmo di elezione di Raft comporta un costo O(2N) a causa dello scambio di messaggi durante il processo di elezione e l'invio periodico di heartbeat. Questo costo aumenta linearmente con il numero di nodi nel cluster (N), poiché ogni nodo deve comunicare con gli altri nodi durante l'elezione e durante il mantenimento del leader attivo. Tuttavia, è importante notare che questo costo è accettabile nella maggior parte dei casi e garantisce un'elevata affidabilità e coerenza nel sistema distribuito.

IV. Implementazione degli Algoritmi di Elezione nel Contesto di Sistemi Distribuiti

Definite le condizioni necessarie e il funzionamento degli algoritmi presi in analisi in questo articolo è possibile definire: scelta delle tecnologie, architettura generale di sistema, considerazioni sulla scalabilità, testing e debugging.

A. Scelta delle Tecnologie

L'obiettivo di questa sezione è chiarire e motivare le scelte intraprese nella scelta delle tecnologie, per l'implementazione degli algoritmi di elezione

1) Liguaggio di Programmazione

La scelta del linguaggio Go per l'implementazione dei sistemi distribuiti è motivata da diversi fattori fondamentali. In primo luogo, Go offre un supporto integrato alla concorrenza tramite GoRoutine e canali, essenziali per gestire efficacemente le attività parallele in ambienti distribuiti. Inoltre, la leggerezza ed efficienza delle risorse di Go lo rendono ideale per sistemi che devono massimizzare l'utilizzo delle risorse disponibili. La sua sintassi semplice e leggibile promuove la chiarezza del codice, facilitando la manutenibilità e il debug nel tempo. Inoltre, Go fornisce un'ampia gamma di librerie standard per la comunicazione di rete, semplificando lo sviluppo di applicazioni distribuite. Infine, la presenza di una vasta comunità di sviluppatori attivi e un ecosistema in crescita assicurano un supporto continuo e risorse abbondanti per affrontare le sfide tipiche dei sistemi distribuiti. Questi fattori combinati rendono Go, una scelta ragionevole e vantaggiosa per progetti distribuiti che richiedono scalabilità, affidabilità e manutenibilità.
Inoltre, si sottolinea che entrambi gli algoritmi di elezione sono stati implementati con un ampio utilizzo di go-routine; questo perché l'implementazione asincrona tramite go-routine in Go offre un'elegante soluzione per l'algoritmo di elezione del leader nei sistemi distribuiti. Questo approccio migliora l'efficienza computazionale grazie alla gestione concorrente delle attività, senza bloccare il flusso di esecuzione. Inoltre, permette una comunicazione efficiente tra i nodi e sfrutta al massimo le risorse hardware del sistema.

2) Tipologia di Comunicazione

La decisione di adottare gRPC in Go per la comunicazione affidabile nei sistemi distribuiti è fondamentale anche per la sua scalabilità e robustezza. Inoltre, gRPC offre un meccanismo di comunicazione altamente efficiente e performante basato su HTTP/2, che consente lo streaming bidirezionale e la gestione automatica della serializzazione dei dati. Questo non solo ottimizza la larghezza di banda e riduce la latenza, ma supporta anche carichi di lavoro distribuiti su larga scala. La sua capacità di generare codice multipiattaforma favorisce un'implementazione omogenea e interoperabile su una vasta gamma di piattaforme e linguaggi di programmazione, il che è essenziale per sistemi distribuiti complessi che coinvolgono componenti eterogenee. Inoltre, gRPC integra la definizione di interfaccia chiara e strutturata offerta da Protocol Buffers (protobuf), agevolando la comunicazione affidabile e la manutenibilità del codice anche in scenari di sviluppo distribuito e in team eterogenei. Inoltre, la crescente adozione e il supporto attivo della comunità contribuiscono a una rapida evoluzione e miglioramento di gRPC, consolidando la sua posizione come soluzione affidabile, scalabile e robusta per le esigenze di comunicazione nei sistemi distribuiti implementati in Go.

3) Cloud Provider

La scelta di AWS per implementare un cluster di nodi per l'algoritmo di elezione è motivata dalla sua affidabilità, scalabilità e facilità di gestione. AWS offre un ambiente altamente affidabile e scalabile, con strumenti di gestione e monitoraggio avanzati. La presenza globale dei data center AWS garantisce una bassa latenza e alta disponibilità. Anche se inizialmente verranno utilizzate solo alcune funzionalità, AWS fornisce una base solida per l'apprendimento e la futura espansione del sistema distribuito.

B. Architettura Generale di Sistema e Distribuzione

Nell'architettura generale, per l'implementazione di un algoritmo di elezione distribuito su AWS, si fa uso di un'istanza EC2 per ospitare i nodi del sistema distribuito. L'istanza EC2 fornisce l'infrastruttura computazionale necessaria per il funzionamento del cluster, consentendo un ambiente flessibile e scalabile in cui è possibile adattare le risorse di calcolo alle specifiche esigenze dell'applicazione. Attraverso l'impiego di Docker, viene garantito l'isolamento e la portabilità delle componenti del sistema distribuito, poiché ogni container rappresenta un nodo nell'algoritmo di elezione. L'uso dei container facilita il confezionamento e la distribuzione dei nodi del cluster, agevolando un deployment rapido e semplificando la gestione delle risorse. Per avviare l'insieme di container, si fa ricorso a Docker Compose all'interno della stessa istanza EC2, evitando così la necessità di utilizzare soluzioni più complesse come Swarm o Kubernetes, che sono tipicamente impiegate per la gestione di nodi distribuiti su macchine separate. La comunicazione tra i nodi del cluster avviene mediante gRPC, un protocollo che garantisce un efficiente e affidabile scambio di messaggi, grazie alla sua capacità di esporre e implementare procedure come servizi remoti tra i componenti distribuiti.
Grazie a questo approccio, si realizza un sistema distribuito altamente scalabile, affidabile e disponibile, capace di gestire carichi di lavoro complessi e di adattarsi dinamicamente alle mutevoli esigenze dell'applicazione.

C. Considerazione sulla Scalabilità

La scalabilità in un progetto relativo ai sistemi distribuiti, come quello descritto, riveste un'importanza cruciale durante la fase di progettazione e implementazione. È essenziale, inizialmente, affrontare la questione della scalabilità a livello architetturale del sistema. La progettazione di un'architettura che agevoli l'aggiunta di nuovi nodi al cluster in modo efficiente e senza interruzioni del servizio è fondamentale. L'utilizzo dei container per confezionare e distribuire i nodi del cluster assicura una maggiore portabilità e flessibilità nell'aggiornamento e nell'espansione del sistema. Inoltre, l'impiego di gRPC per la comunicazione tra i nodi offre una notevole scalabilità orizzontale, permettendo di gestire un numero crescente di richieste senza compromettere le prestazioni del sistema. Il gRPC agevola la definizione di procedure remote tramite il protocollo HTTP/2, garantendo un'efficace gestione delle chiamate remote e una maggiore efficienza nel trasporto dei dati. La containerizzazione dei nodi del cluster consente un'efficiente isolamento e gestione delle risorse di calcolo e di rete, semplificando il deployment e il provisioning del sistema su larga scala.
Inoltre, la capacità di eseguire più istanze del cluster su diverse macchine fisiche o virtuali, pur considerando che il progetto sia stato sviluppato e distribuito sui container presenti sull'unica istanza EC2 tramite Docker Compose, aggiunge un livello di ridondanza e affidabilità al sistema. Questa disposizione permette di gestire carichi di lavoro intensi e di fronteggiare eventuali guasti hardware o software senza che ciò comporti interruzioni del servizio. In sintesi, mediante questo approccio, si realizza comunque un sistema distribuito altamente scalabile, affidabile e disponibile, capace di gestire in modo efficace carichi di lavoro complessi e di adattarsi dinamicamente alle mutevoli esigenze dell'applicazione. Nel contesto di questo progetto, si evidenzia la presenza di un elemento centralizzato di rilievo, ossia il "Registry", il quale potrebbe potenzialmente generare punti critici e possibili guasti. Questa eventualità può essere mitigata attraverso la replica dei registry e il mantenimento di una stretta coerenza tra le informazioni contenute in tali repliche, mediante l'attacco di un volume containerizzato ai registry e la sincronizzazione della struttura dei file al loro interno. Inoltre, si potrebbe considerare l'implementazione di un load balancer, il quale, mediante un algoritmo di assegnazione equa, distribuisce le richieste in ingresso tra i diversi registry. Tuttavia, si evidenzia che questa soluzione non è stata adottata o gestita, poiché va al di là delle specifiche richieste del progetto.

D. Test e debugging

Prima di procedere con il deployment containerizzato all'interno dell'istanza EC2, è stato necessario svolgere una fase di test e debugging. Questa fase è stata condotta in seguito allo sviluppo degli algoritmi, una volta osservato il comportamento e individuate eventuali criticità degli stessi in ambiente locale. Solo dopo aver accertato la robustezza dell'implementazione, questa è stata distribuita sui container all'interno dell'istanza EC2.
Durante questo processo, sono stati eseguiti vari test sugli algoritmi di elezione ad anello e di Raft. Per l'algoritmo ad anello, sono stati effettuati test per verificarne il corretto funzionamento durante l'elezione, nel rispetto delle specifiche dell'algoritmo. Inoltre, è stata verificata la gestione di una possibile situazione di doppia elezione contemporanea, garantendo che portasse allo stesso risultato anche nel caso in cui il container leader fosse in una stato di guasto. Per quanto riguarda l'algoritmo di Raft, i test hanno riguardato la corretta esecuzione durante l'elezione e il comportamento adeguato del nuovo processo di elezione in caso di interruzione del container leader. Successivamente, è stata valutata la resilienza alla partizione di rete, avviando un container leader eletto nel termine più antico e verificando che rimanesse il leader più recente eletto nel termine di valore maggiore. Ciò dimostra la robustezza dell'algoritmo di Raft anche in situazioni di partizione di rete.
In aggiunta, nell'ultimo algoritmo in esame, si evidenzia l'utilizzo di timer dell'ordine dei secondi anziché dei millisecondi. Questa scelta è stata motivata dalla necessità di osservare più agevolmente il comportamento dell'algoritmo. È importante notare che, sebbene i timer operino su un ordine di grandezza nettamente superiore rispetto alla versione originale, essi presentano un'intersezione non nulla. Questo garantisce che nessun aspetto del comportamento dell'algoritmo venga trascurato, permettendo così di valutarne la robustezza in ogni scenario possibile. È da sottolineare che anche nella versione distribuita ufficiale è mantenuto questo stesso ordine di grandezza dei timer, assicurando coerenza e uniformità nell'esecuzione dell'algoritmo.

Pseudocodice Algoritmo di Elezione ad Anello

Qui di seguito gli pseudocodici della struttura del codice per il peer e il registry nell'algoritmo di elezione ad Anello:

Algorithm 1: Pseudocodice Peer

```
main() avviaServer();
avviaElezioni();
controllaLeader();
Funzione avviaServer:
    // Avvia il server gRPC sulla porta specificata
Funzione avviaElezioni:
    // Avvia un timer per eseguire l'elezione ad anello
       periodicamente
    while true do
        if LeaderPeerAddress è vuoto e len(jsonListPeer) ¿ 1 then
            posizione ← trovaMe();
            trovaConnessioneValida(posizione, []string);
        end
        attendi ΔT secondi end
    Funzione controllaLeader:
        // Controlla periodicamente se il leader è ancora
           attivo
        while true do
            if LeaderPeerAddress non è vuoto e len(jsonListPeer) ¿ 1
             then
                if ¬ controllaLeaderAttivo() then
                    LeaderPeerAddress ← "";
                end
            end
            attendi ΔT' secondi end
        Funzione trovaConnessioneValida(posizione int, listaElezioni
         []string):
            peerSuccessivo ← (posizione + 1) modulo len(jsonListPeer);
            while non trovi un peer valido do
                if peerSuccessivo non è l'host corrente then
                    connessione ←
                     connettiA(jsonListPeer[peerSuccessivo]);
                    if connessione ≠ nil then
                        inviaMessaggioElezioni(connessione,
                         listaElezioni);
                        esci dal ciclo;
                    end
                end
                peerSuccessivo ← (peerSuccessivo + 1) modulo
                 len(jsonListPeer);
            end
        Funzione inviaMessaggioElezioni(connessione,
         listaElezioni):
            // Invia un messaggio di elezione al peer tramite
               la connessione
        Funzione controllaLeaderAttivo ():
            // Controlla se il leader è ancora attivo tramite
               una richiesta al leader stesso
        Funzione trovaMe ():
            // Trova la posizione dell'host corrente
               nell'anello
        Funzione connettiA(peer):
            // Tenta di connettersi al peer specificato
```

Algorithm 1: Pseudocodice Registry

```
main() avviaServer();
Funzione avviaServer:
    // Avvia il server gRPC
    // Gestisce le richieste dei client e coordina l'elezione
       dell'anello
    // Inizia ad ascoltare sulla porta specificata
    // Registra i servizi di elezione dell'anello
Funzione SendRequest(ctx context.Context, item
 *interfaccia.JSONClientRegistration):
    // Gestisce le richieste di registrazione dei client
       nell'anello
    // Verifica se l'elemento esiste già nella lista
    // Aggiunge l'elemento alla lista e la ordina
    // Invia aggiornamenti ai peer nell'anello
Funzione
 update_peer_informazion(hostnameOld string, portOld string):
    // Aggiorna le informazioni dei peer nell'anello
    // Invia aggiornamenti a tutti i peer tranne quello
       specificato
Funzione inviaUpdateAlPeer(hostname string, port string):
    // Invia un messaggio di aggiornamento al peer
       specificato
    // Connessione al serverRingElection gRPC
```

Pseudocodice Algoritmo di Elezione di Raft

Qui di seguito gli pseudocodici della struttura del codice per il peer e il registry nell'algoritmo di elezione di Raft:

Algorithm 1: Pseudocodice Peer

```
main() avviaServer();
genesiRaft();
Funzione avviaServer:
    // Avvia il server gRPC
Funzione genesiRaft:
    // Gestisci i messaggi dai canali e le azioni appropriate
    while true do
        // Seleziona il comportamento in base al canale
        <-FollowerChannel // Gestisci il comportamento da
           follower
        handleFollower();
        <-ElectionChannel // Gestisci il comportamento di
           elezione
        handleElectionMessages();
        <-HeartbeatChannel // Gestisci l'invio dell'heartbeat
        handleHeartbeat();
    end
Funzione handleElectionMessages:
    // Genera un timer di elezione casuale e aumenta il term
    // Invia richieste di voto ai peer
    // Interrompi se il timer scade prima
Funzione sendRequestVoteToPeers:
    // Incrementa il voto per se stesso
    // Invia richieste di voto ai peer
Funzione connettiEInviaVoto(peer):
    // Connettiti al peer specificato e invia la richiesta di
       voto
Funzione handleHeartbeat:
    // Gestisci l'invio dell'heartbeat ai peer
Funzione inviaHeartbeat:
    // Invia l'heartbeat ai peer
Funzione connettiEInviaHeartbeat(peer):
    // Connettiti al peer specificato e invia l'heartbeat
Funzione handleFollower:
    // Gestisci il comportamento da follower
    // Attendi l'arrivo degli heartbeats o il termine del
       timer di elezione
    // Avvia nuove elezioni se il timer scade prima
Funzione handleElectionTimeout:
    // Avvia nuove elezioni
```

Algorithm 1: Pseudocodice Registry

```
funzione main():
    iniziaAscolto();
funzione iniziaAscolto():
    // Avvia l'ascolto sulla porta specificata
    // Crea un nuovo server gRPC e registra il servizio
       RaftElection
    // Inizia ad ascoltare per le richieste in arrivo
    // Gestisce gli errori durante l'ascolto
    // Gestisce gli errori durante il servizio
funzione gestisciErroreAscolto(err errore):
    // Stampa l'errore durante l'ascolto
funzione gestisciErroreServe(err errore):
    // Stampa l'errore durante il servizio
funzione aggiornaInformazioniPeerRaft(hostnameOld string,
 portOld string):
    // Aggiorna le informazioni dei peer nell'anello Raft
    // Invia aggiornamenti a tutti i peer tranne quello
       specificato
    // Gestisce gli errori durante l'invio degli
       aggiornamenti
funzione inviaAggiornamentoTuttiPeerRaft(hostname, port
 string):
    // Invia un aggiornamento a tutti i peer nell'anello Raft
    // Connettiti al peer specificato
    // Crea un oggetto JSONListPeer con i dati aggiornati
    // Invia l'aggiornamento tramite la connessione gRPC
    // Gestisce gli errori durante l'invio dell'aggiornamento
```

RIFERIMENTI

Si fanno riferimenti ai seguenti articoli dove vengono analizzati ed elaborati gli algoritmi trattati:

- In Search of an Understandable Consensus Algorithm (Diego Ongaro, John Ousterhout), Anno: 2014
- Consensus: Bridging Theory and Practice (Diego Ongaro), Anno: 2014
- Message Efficient RingLeader Election in Distributed Systems (Syed Muhammad Bilal, Muhammad Saeed, Sajjad Haider), Anno: 2018
- The Ring Algorithm (Chang and Roberts Algorithm) (E. Chang, R. Roberts), Anno: 1979
- An Improved Leader Election Algorithm for Distributed Systems (Fuyao Liu, Chunxiao Jiang, Xiaoyu Song), Anno: 2012