Implementazione di Algoritmi di Elezione in un contesto Distribuito

Progetto SDCC 2023-2024
Andrea Tozzi
Macroarea di Ingegneria
Corso di laurea in Ingegneria Informatica
Università degli Studi di Roma Tor Vergata
andreatozzi199@gmail.com

ABSTRACT

L'obiettivo di questo documento è descrivere l'architettura, le scelte progettuali e l'implementazione di due algoritmi di elezione. Questi sono fondamentali per garantire la corretta elezione di un leader in un sistema distribuito a seguito del crash del precedente leader, assicurando la coerenza e l'affidabilità del sistema. L'elezione di un leader è un processo cruciale poiché esso è responsabile della gestione delle operazioni critiche, della coordinazione tra i nodi nel sistema e del mantenimento della consistenza dei dati. Senza un meccanismo di elezione robusto, un sistema distribuito può diventare incoerente, inefficiente e vulnerabile ai guasti. Nel corso di questa trattazione verranno presentati l'algoritmo Bully e Raft. Chiameremo occasionalmente il "leader" anche con il nome di "coordinator". Un nodo può diventare inattivo a seguito di un "crash", chiamato anche "failure" all'interno della trattazione.

INTRODUZIONE

In questo progetto si è assunto di essere in un sistema sincrono, dove i tempi di comunicazione e di elaborazione dei messaggi tra i nodi sono limitati e noti. Questo garantisce la corretta applicazione degli algoritmi scelti che sono applicabili su un sistema sincrono. Nel contesto specifico c'è bisogno di garantire le proprietà di:
Liveness: soltanto un nodo, non guasto, sarà eletto come leader.
Safety: l'algoritmo, porterà all'elezione di un singolo leader all'interno di una rete.
Per garantire tali proprietà è necessario utilizzare una comunicazione affidabile tra i nodi che comunicano all'interno della rete distribuita tramite scambio di messaggi. Essi sono rappresentati all'interno di questa trattazione in maiuscolo, ad esempio: ELEZIONE è un messaggio di elezione inviato da un nodo verso un altro. Nel sistema che si è sviluppato, i nodi comunicano esclusivamente durante le fasi di elezione di un nuovo leader e il meccanismo di failure detection, in modo da concentrarsi sul corretto sviluppo degli algoritmi di elezione. Questo non genera alcuna differenza nell'elezione di nuovi leader, nel momento in cui si volesse utilizzare il sistema sviluppato per ulteriori scopi basterà aggiungere le funzionalità desiderate. Le reti di nodi in analisi sono di tipo "fully connected", in cui ogni nodo è a conoscenza di tutti gli altri nodi nella rete e vi comunica per selezionare il nodo coordinatore. Entrambi gli algoritmi sono correttamente funzionanti in questo tipo di reti, ma solamente Raft può essere adottato anche in reti non completamente connessi, risultando inoltre tollerante alle partizioni di rete. In un sistema distribuito, in alcuni contesti, i termini "processo" e "nodo" possono essere usati in modo intercambiabile, ma è importante distinguere tra i due.
Processo: Un'unità di esecuzione che può essere avviata, fermata e gestita indipendentemente.
Nodo: L'entità fisica o virtuale che ospita uno o più processi.

Algoritmo Bully:

Spesso utilizzato per la sua semplicità ed efficienza in reti con un numero limitato di processi, in cui si vuole mantenere in ogni istante come leader il processo con ID superiore tra gli attivi. Se un processo P(i) rileva l'assenza del nodo coordinatore, promuove una nuova elezione seguendo questi passaggi:

1) P(i) invia ELEZIONE a tutti i processi con ID maggiore del suo.

2) Se nessun nodo contattato risponde, P(i) si autoproclama vincitore, invia COORDINATOR a tutti i processi con ID minore del suo e diventa il nuovo leader. Se P(i) riceve almeno uno STOP allora lascerà il posto a nodi con ID superiori del suo.

Se un processo P(k) con (k > i) riceve ELEZIONE da P(i), risponde con STOP per dichiarare di essere attivo e indice una nuova elezione per cercare di diventare leader. Se un processo P(k) riceve COORDINATOR da P(i), lo riconosce come nuovo leader della rete. Per appurare che nessuno abbia inviato STOP, P(i) deve attendere un tempo pari all'RTT massimo ammissibile nella rete (noto grazie all'assunzione di sistema sincrono).

Il costo dell'algoritmo in termini di numero di messaggi scambiati è:
Caso ottimo: il processo non guasto con ID più alto è colui che si accorge per primo della failure del coordinatore; esso può eleggersi subito come coordinatore e inviare (N-2) messaggi di proclamazione. In breve, si hanno O(N) messaggi.
Caso pessimo: il processo con ID più basso è colui che rileva per primo il crash del leader; esso invia (N-2) messaggi agli altri processi, ciascuno dei quali a sua volta indice una nuova elezione. In totale si hanno O(N^2) messaggi.

Algoritmo Raft:

Per primo distinguiamo i seguenti concetti:

1) Mandato: il periodo durante il quale un leader è in carica, è identificato da un numero incrementale. Durante un mandato, il leader invia periodicamente HEARTBEAT ai follower come meccanismo di failure detection. Quando un nodo rileva il crash del leader, avvia un'elezione per iniziare un nuovo mandato.

2) Stato in cui un nodo può trovarsi:
   - Follower: Un nodo che segue il leader corrente. Risponde a HEARTBEAT e REQUESTVOTE aggiornando il proprio mandato corrente in base ai messaggi ricevuti.
   - Candidate: Un nodo che avvia un'elezione quando non riceve comunicazioni dal leader per un periodo di tempo randomico chiamato "time-out elettorale". Il time-out è di dimensione non fissa per impedire elezioni simultanee da più nodi. Un candidato invia REQUESTVOTE a tutti i nodi per ottenere i voti necessari a diventare leader.
   - Leader: Un nodo che ha vinto un'elezione. Invia HEARTBEAT ai follower per mantenere la leadership.

Il processo di elezione nell'algoritmo Raft segue i passi:

1.Time-out Elettorale: Se un follower non riceve HEARTBEAT dal leader entro un certo periodo temporale, diventa un candidato e avvia un'elezione.

2.Incremento del Mandato: Il candidato incrementa il proprio mandato corrente e vota per sé stesso.

3.Richiesta di Voto: Il candidato invia REQUESTVOTE agli altri nodi nel cluster.

4.Risposta ai Messaggi di Voto: I nodi rispondono con un messaggio di REQUESTVOTE:

- Se il mandato del candidato è maggiore del mandato del nodo ricevente, il nodo ricevente aggiorna il proprio mandato e vota per il candidato.
- Se il nodo ricevente ha già votato per un altro candidato o il mandato del candidato è minore del mandato del nodo ricevente, il voto viene rifiutato.

5.Vittoria dell'Elezione: Se il candidato riceve la maggioranza dei voti, diventa il nuovo leader e inizia il processo di failure detection inviando periodicamente HEARTBEAT ai follower.

Il costo dell'algoritmo in termini di numero di messaggi scambiati è: O(2N) a causa dello scambio di messaggi durante il processo di elezione.

DESIGN DELLA SOLUZIONE

Per l'implementare gli algoritmi descritti è stato utilizzato il linguaggio Go, grazie al suo supporto nativo per le chiamate RPC. L'invocazione di una procedura remota rappresenta un messaggio inviato da un nodo ad un altro. L'utilizzo di Docker consente di creare, distribuire ed eseguire applicazioni all'interno di container (ambienti isolati che includono tutto il necessario per eseguire il software). Docker Compose è stato utilizzato per gestire e orchestrare facilmente le applicazioni multi-container su un singolo host. L'applicazione è stata eseguita su un'istanza EC2 di Amazon Web Services per eseguire lo script di Docker Compose e avviare il sistema distribuito. Per i primi test in locale è stata definita una funzione per simulare il crash e la recovery dei nodi, mentre per l'implementazione su container, il crash è stato simulato manualmente mandando in pausa o riavviando uno specifico container.

DETTAGLI DELLA SOLUZIONE

È stato implementato un Server Registry per gestire la registrazione dei nodi e permettere a ciascuno di loro di conoscere gli altri nodi e ottenere le informazioni necessarie per contattarli. L' indirizzo e la porta di ascolto del Server sono fissi e conosciuti da tutti i nodi tramite un file di configurazione. Questo componente consente ai nodi di registrarsi ottenendo un proprio ID e l'elenco dei nodi presenti, facilitando la comunicazione e il coordinamento tra i nodi. Il server, inoltre, rende il sistema facilmente scalabile, non necessitando di interconnettere ogni singola componente del sistema prima dell'esecuzione. Per semplicità di implementazione è stata assunta l'impossibilità di crash del server di registrazione. È importante sottolineare come i nodi interagiscano con il server registry solo in fase di avvio e non per comunicare eventuali fallimenti riscontrati o per aggiornare l'ID dell'attuale leader. Questa scelta è stata adottata al fine di ridurre al minimo le interazioni tra nodi e server registry, riducendo inoltre le sue responsabilità, poiché essendo centralizzato, rappresenterebbe un punto critico in un contesto in cui possono riscontrare fallimenti.

Per far fronte all'assunzione di un sistema sincrono è stato impostato manualmente un valore per il parametro maxRTT (Round-Trip Time massimo) per definire il limite di tempo entro il quale i messaggi devono essere inviati e ricevuti. Questo valore è cruciale per mantenere la sincronizzazione tra i nodi e garantire che le decisioni effettuate siano corrette, nel rispetto di quanto stabilito dagli algoritmi in analisi.
Nella fase di verifica è stata inizialmente definita una funzione, eseguita in una go routine, che simula il crash di un nodo per valutare il corretto funzionamento degli algoritmi. Per velocità di implementazione, tale funzione di crash emulation ha il solo compito di interrompere e riavviare il listener delle chiamate RPC, tramite una probabilità definita e modificabile da file di configurazione. Con il suo utilizzo si sono potute riscontrate numerose situazioni di malfunzionamento, questo perché la funzione genera una maggiore variabilità di eventi nel sistema rispetto alla disattivazione e attivazione manuale dei nodi. Tale funzione può essere disabilitata, per simulare manualmente il crash dei nodi.

Un nodo, a seguito di un riavvio deve riottenere il suo precedente ID nella rete, perché registrandosi nuovamente otterrebbe un ID diverso. Con l'aggiunta di un file di log all'interno del container si è potuto differenziare il primo avvio di un container con l'avvio in fase di recovery (ritorno da un crash). Se il file è vuoto, il nodo eseguito all'interno del container deve registrarsi per la prima volta e ottenere un nuovo ID, altrimenti il suo ID è già noto e contatta il Server Registry per aggiornare i suoi valori come indirizzo e porta e per ottenere la lista dei nodi in rete che potrebbe essere variata durante il periodo di inattività.

Per implementare gli algoritmi si è posta attenzione nell'utilizzo di mutex per evitare accessi contemporanei alle variabili di stato: election e hbState, che descrivono rispettivamente l'esistenza o meno di un processo attivo di elezione e heartbeat. Questo previene conflitti, impedendo che più processi di ciascun tipo siano attivi contemporaneamente.

Dettagli implementativi dell'algoritmo Bully:

La fase di recovery viene effettuata come prima operazione nel metodo main che recupera il precedente ID all'interno del file di log, se non presente l'ID è impostato al valore "-1". Viene successivamente creata una entità nodo che sfrutta i metodi dell'algoritmo Bully e invoca la funzione 'start'.

La funzione `start` espone i metodi per le chiamate RPC e crea un listener RPC per accettare connessioni in arrivo. Le chiamate a procedura remota rappresentano i messaggi, previsti dall'algoritmo, scambiati tra i nodi. Successivamente, il nodo si connette al server di registrazione e invia una richiesta per aggiornare le sue informazioni ricevere un ID in caso di primo avvio del nodo.
A seguito della registrazione, viene richiesta la lista dei nodi attualmente registrati nella rete, e poiché non conosce il leader (informazione non mantenuta dal server registry), avvia un'elezione. Questo ha un duplice vantaggio:

1. Se il nodo ha l'ID più alto tra i nodi attivi: Deve diventare il nuovo leader, e senza una nuova elezione questo non sarebbe possibile.

2. I nodi già presenti nella rete vengono a conoscenza del nuovo ID poiché contattati da COORDINATOR del nuovo nodo (con ID maggiore), ottenendo una conoscenza completa della rete ad ogni nuovo ingresso, senza bisogno di contattare periodicamente il server registry per aggiornamenti.

Infine, è invocata la funzione `startHeartBeatRoutine` per avviare la routine di HeartBeat, solo se non attiva, evitando che più routine di

HeartBeat siano attive contemporaneamente. La stessa precauzione è adottata per l'avvio del processo di elezione tramite "startElectionBully", in quanto non possono essere attive due o più elezioni sullo stesso nodo. Terminata la fase di avvio, il nodo rimane in attesa di "messaggi" come: ELECTION, COORDINATOR, o STOP da parte dei nodi nella rete, ai quali risponderà come previsto dall'algoritmo.

In seguito al crash del leader, per costruzione dell'algoritmo, ogni nodo invia ELECTION unicamente ai nodi con ID superiore al suo, che genera quindi un elevato numero (al crescere dell'ID) di ELECTION ricevuti. Per far fronte ad un eccessivo invio di messaggi nella rete è stata aggiunta una variabile binaria "stopped". Essa può cambiare valore solo una volta nel periodo di tempo che separa il crash del leader e l'elezione del nuovo, utile ad evitare che un nodo con ID non massimo possa avviare un'elezione più di una volta. Prendiamo in esempio il nodo 2 che avvia un'elezione dopo il crash del leader (nodo 5). Se il nodo 4 invia un messaggio STOP, il nodo 2 interrompe l'elezione. Tuttavia, se un altro nodo (nodo 1) invia un messaggio ELECTION, il nodo 2 non deve avviare un'altra elezione, poiché è già stato interrotto una volta.

Una ulteriore precauzione è stata adottata nel processo di elezione del nodo con ID massimo. Una volta avviata un'elezione, il nodo con ID massimo non incomincia la fase di invio ELECTION in quanto non ci sono nodi con ID superiore al suo, procedendo immediatamente con l'invio di COORDINATOR a tutti i nodi. Questo passaggio immediato alla fase di vincita dell'elezione può portare all'avvio di elezioni in serie, che avranno tutte lo stesso esito positivo, con conseguente invio ripetuto di COORDINATOR non necessari. Per evitare che questo accada è stata aggiunta una attesa di maxRTT prima di diventare leader per attendere ulteriori messaggi ELECTION da parte di più nodi nella rete ed impedire di avviare ulteriori elezioni "in serie".

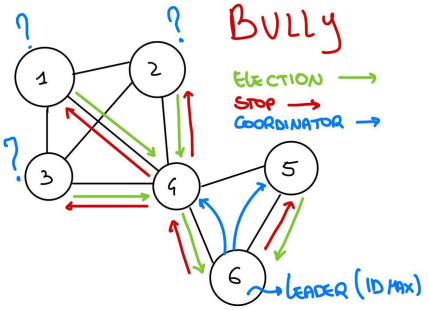
Dettagli implementativi dell'algoritmo Raft:

La fase di recovery e di start sono invariate rispetto l'algoritmo Bully. L'algoritmo richiede che un nodo ottenga la maggioranza dei voti per diventare leader. Se si considerano anche nodi non attivi, il conteggio dei voti potrebbe risultare insufficiente per raggiungere la maggioranza, anche se i nodi attivi hanno votato a favore. Si assicura che la maggioranza sia calcolata correttamente in base ai nodi effettivamente presenti e funzionanti nella rete (raggiungibili in fase di REQUESTVOTE). Questo approccio è stato adottato per garantire la correttezza del processo di elezione. In un sistema in cui i nodi possono fallire in qualsiasi momento utilizzare il numero totale di nodi nella rete potrebbe portare a situazioni in cui un nodo tenta di ottenere la maggioranza di voti, che non verrà mai raggiunta da nessuno dei nodi, rendendo l'elezione inefficace o addirittura senza successo. Per eseguire ad ogni elezione il conteggio dei nodi attivi, il nodo crea un canale per raccogliere gli errori dalle goroutine che inviano REQUESTVOTE. Ogni goroutine invia una richiesta di voto a un nodo specifico e invia il risultato (errore o nil) nel canale. Un'altra goroutine chiude il canale solo dopo che tutte le goroutine che inviano richieste di voto hanno terminato. Raccolti i risultati dal canale verranno conteggiate solo le risposte senza errori.

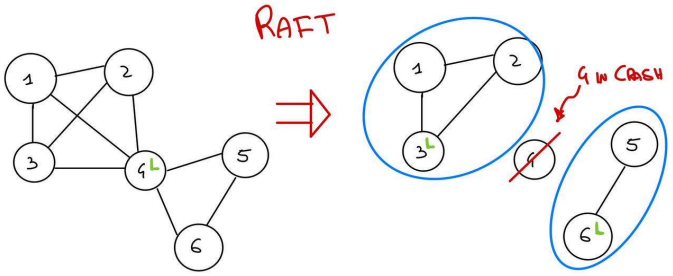
Quando un REQUESTVOTE viene inviato, il nodo ottiene la risposta tramite il campo reply della procedura remota, a differenza di quanto previsto dall'algoritmo, che prevede l'attesa di ricezione di un REQUESTVOTE con il responso dal nodo contattato. Questa scelta diminuisce il numero totale di messaggi scambiati in rete e riduce la complessità implementativa del nodo. Tale semplificazione è stata adottata solo nell'algoritmo Raft, in quanto nell'algoritmo Bully si è deciso di attendere le risposte STOP ai messaggi ELECTION, senza utilizzare la struttura di ritorno della chiamata a procedura remota.

Reti parzialmente connesse e Partizionamento della rete:

L'algoritmo Bully non può funzionare in una rete parzialmente connessa: l'algoritmo si basa sul presupposto che tutti i nodi siano in grado di comunicare tra loro, per eleggere il nuovo coordinatore. Se alcuni nodi (come 1, 2 e 3) non possono contattare o essere contattati dal leader, essi non sapranno mai chi è il coordinatore eletto. Questo genera una situazione di inconsistenza nella rete, dove alcuni nodi potrebbero continuare a funzionare come se non esistesse un coordinatore.



È possibile, in una rete non completamente connessa, che si verifichino scenari in cui la rete è divisa in sottogruppi indipendenti di nodi. In tali situazioni, i nodi che rimangono in comunicazione tra loro possono tentare di eleggere un nuovo leader. L'algoritmo Raft consente l'elezione di leader indipendenti in ciascuna delle sottoreti, portando a situazioni in cui potrebbero esistere più leader contemporaneamente in parti diverse della rete.
Ad esempio, se una rete di 6 nodi si divide in due partizioni una con 3 nodi e l'altra con 2 nodi a seguito del crash del nodo Leader 4 che fungeva da ponte, ogni partizione può eleggere il proprio leader, a condizione che la maggioranza dei nodi attivi all'interno di quella partizione voti a favore. Questo meccanismo di elezione è intrinsecamente progettato per consentire l'operatività continua della rete anche in presenza di partizioni, garantendo che i nodi possano sempre cercare di stabilire un leader all'interno del loro subset di nodi. Il nodo 4, tornando operativo, rileva la presenza di due leader, iniziando così una nuova elezione che vincerà per maggioranza di voti, ristabilendo la condizione di Safety.



Per testare l'algoritmo Raft in tali situazioni è stata aggiunta una modalità di personalizzazione della rete nel server registry, piuttosto che costruire una rete completamente connessa. Con la modalità "manualTopology" si può definire una matrice di adiacenza che rappresenta la topologia della rete, consentendo di specificare quali nodi possono comunicare tra loro. Il metodo remoto del server registry ora utilizza la matrice di adiacenza per restituire al nodo (i) che richiede la lista dei nodi nella rete, unicamente i nodi con coefficiente (uno) nella riga i-esima.

TEST e DEBUGGING

Fase progettuale avviata in seguito allo sviluppo degli algoritmi, per poter osservare eventuali criticità degli stessi in ambiente locale. I primi test sono avvenuti in locale tramite uno script bash che esegue un server registry e successivamente un numero n (a scelta) di nodi, ognuno in una in una console separata, così da osservare e controllare al meglio il comportamento di ciascun nodo grazie al log su stdout. Una volta che il corretto comportamento degli algoritmi a seguito di interventi manuali è stato verificato, è stata attivata la funzione di crash simulation, per poter osservare il comportamento in una situazione più dinamica e imprevedibile, più simile alla realtà, grazie alla quale sono state scovate e risolte numerose criticità. Le esecuzioni effettuate hanno avuto una durata di ore, con un massimo di tre (con dimensione di rete massima pari a 20 nodi) all'interno delle quali, ad intervalli regolari, si osservavano e valutavano le seguenti proprietà:

- Per Bully che il nodo leader sia, in ogni istante unico e con ID maggiore tra i nodi attivi.
- Per Raft che il leader sia il nodo con numero di mandato maggiore.
- Per entrambi, in ogni istante deve sempre essere presente un leader o in caso contrario, almeno una elezione attiva.

Nei test effettuati eseguendo ogni nodo su un container, tutti gestiti e connessi tramite un docker-compose.yml si è cercato di garantire che la funzionalità di crash recovery fosse correttamente funzionante, verificando che il contenuto del file di log fosse pari all' ID del nodo. A seguito di una interruzione e riattivazione manuale si è constatato che il nodo abbia recuperato il suo procedente ID, prima di effettuare l'iscrizione al server registry, tornando così ad occupare la posizione precedente nella rete. Ottenere un nuovo ID avrebbe portato il nodo ad ottenere l'ID maggiore della rete, e lasciare il precedente inutilizzato.

Solo dopo aver accertato la corretta implementazione, il codice è stato distribuito sui container in una istanza offerta del servizio EC2 di AWS, all'interno della quale sono stati effettuati gli stessi test sopra descritti. Tali test sono stati necessari per accertare il giusto dimensionamento dell'istanza EC2.

DISCUSSIONE

Server Registry

Nel contesto di un sistema distribuito, il server registry gioca un ruolo fondamentale in quanto gestisce la registrazione dei nodi e facilita il loro coordinamento. Tuttavia, essendo un componente centralizzato, rappresenta un punto critico per la scalabilità e la resilienza dell'intero sistema. La scelta di centralizzare questa funzionalità semplifica l'implementazione, ma espone il sistema a possibili vulnerabilità in caso di crash o malfunzionamenti del server registry (non previsti in questo studio). Essendo responsabile della registrazione iniziale dei nodi, il server registry fornisce a ciascun nodo un ID univoco e mantiene un elenco dei nodi attivi nella rete. Tuttavia, una volta completata la fase di avvio, i nodi non dipendono più dal server registry per il loro funzionamento regolare. Questo design riduce il carico sul server e limita le interazioni non essenziali, ma non elimina completamente i rischi associati al suo crash. Se il server registry dovesse fallire, diversamente da come previsto, la registrazione di nuovi nodi e la gestione degli ID risulterebbero impossibili fino al ripristino del servizio. Per garantire la robustezza del sistema anche in presenza di guasti del server registry, potrebbero essere adottate alcune soluzioni alternative:

1. Replica del server registry: Una possibile soluzione è la replica del server registry su più nodi, creando così una configurazione distribuita che elimina il single point of failure. In questo scenario, ogni replica del server registry deve essere sincronizzata con le altre per mantenere una vista coerente dell'intero sistema.
2. Persistenza dei dati del server registry: Un'altra soluzione potrebbe consistere nel mantenere una copia persistente delle informazioni del server registry. In caso di crash, il server potrebbe essere ripristinato con lo stato più recente, limitando così l'interruzione del servizio.

Algoritmo di crash simulation:

La funzione è stata realizzata per una breve fase di verifica (in locale) degli algoritmi in una fase preliminare del processo di sviluppo. Essa non simula a pieno il processo di crash di un nodo, in quanto, è responsabile della sola interruzione e riattivazione della possibilità di comunicare del nodo. Essa, quindi, non simula l'intero processo che separa la failure del nodo e la sua riattivazione.
Tale scelta è dovuta principalmente a:

1. Facilità di implementazione.
2. Mancanza di una funzione di recovery dello stato, implementata solo successivamente, che avrebbe complicato l'implementazione della funzione di crash simulation.

Un algoritmo così realizzato porta situazioni indesiderate durante i processi di elezione, ad esempio: Un nodo che ha interrotto le comunicazioni non verrà a conoscenza di un nuovo ingresso nella rete, avvenuto tramite il primo messaggio ELECTION per la scoperta dell'attuale leader, e in seguito alla sua riattivazione, non contatterà il nuovo nodo, che potrebbe portare situazioni indesiderate. Tuttavia, questo comportamento è opportunamente gestito (come descritto nella fase di test) nel momento in cui un nodo dovesse riscontrare un vero crash, con successiva fase di recovery e aggiornamento presso il server registry.

Timer/Intervalli adottati:

I timer e gli intervalli adottati negli algoritmi di elezione dei leader sono stati scelti nell'ordine dei secondi per facilitare la comprensione e la lettura dei messaggi delle stampe di output. Sebbene questi intervalli non siano realistici per un sistema di produzione, essi permettono di osservare e analizzare il comportamento degli algoritmi in un contesto di sviluppo e test. Il tempo massimo di RTT è impostato ad 1 secondo, il delay tra l'invio dei messaggi di HeartBeat è 2 secondi, e il range possibile per il time-out elettorale di Raft va da 5 a 15 secondi. Si è comunque osservato come, impostando dei tempi nell'ordine delle centinaia di millisecondi, i due algoritmi continuino a funzionare regolarmente.

Confronto

L'algoritmo Bully risulta più semplice da implementare per via della sua struttura lineare: i nodi con ID più basso inviano messaggi di elezione e quelli con ID più alto rispondono, evitando complessità come la gestione di stati intermedi o time-out variabili. Raft, invece, presenta più stati dei nodi (follower, candidate, leader) e gestisce time-out casuali per prevenire conflitti elettorali, aumentando la complessità generale.

La principale e unica difficoltà riscontrata nell'implementazione dell'algoritmo Bully risiede nella gestione dei molteplici messaggi ricevuti dai nodi al crescere del loro ID e al diminuire dell'ID del nodo che per primo scopre il failure del leader. Tali messaggi se non opportunamente gestiti possono portare ad elezioni in serie ripetute, con un conseguente invio eccesivo di messaggi nella rete. Mentre per l'algoritmo Raft, la principale difficoltà è stata riscontrata nel codificare ogni singola possibilità di interazione tra i nodi. Piccoli cambiamenti nel codice, generano un comportamento estremamente instabile.

L'utilità del meccanismo di failure detection adottato dall'algoritmo Bully, che prevede l'invio di un messaggio ad intervalli regolari da parte dei nodi verso il Leader, è da valutare in relazione alle caratteristiche della rete. In scenari in cui i failure del leader sono rari, questo meccanismo causa un overhead non necessario. Si potrebbe quindi pensare di delegare in tali casi, il meccanismo di failure detection all'interno delle normali interazioni tra nodi e leader, ma ciò porterebbe comunque a ritardare la scoperta del failure fin quando un nodo non tenta di comunicare (per necessità) con il leader.

Questo ritardo può rallentare la reazione al fallimento e il relativo soddisfacimento della richiesta del nodo, specialmente in situazioni dove il leader non è chiamato a operare spesso.

L'algoritmo Raft invece è basato su un meccanismo di failure detection più conservativo in termini di messaggi inviati, il suo time-out elettorale consente nella maggioranza dei casi ad un solo nodo di rilevare il failure del leader, incominciando un'elezione senza concorrenti, velocizzando quindi il tempo necessario a rimpiazzare il leader. Inoltre, il numero di messaggi scambiati per un'elezione è linearmente dipendente alla dimensione della rete, permettendo di scalare senza eccessive difficoltà a differenza dell'algoritmo Bully. Potrebbe essere interessante considerare, in reti con alta stabilità, una variante in cui il carico dovuto al meccanismo di failure detection è spostato sui nodi, lasciando che siano essi stessi ad inviare Heartbeat al leader ad ogni time-out elettorale (casuale). Ciò permetterebbe di ridurre il totale di messaggi scambiati nell'unità di tempo, con lo svantaggio di dover ritardare la scoperta del failure.

L'algoritmo Bully è più tollerante ai guasti, un nodo può rimanere da solo nella rete senza rischio di riscontrare malfunzionamenti nelle elezioni. L'algoritmo Raft è meno tollerante, nello specifico, il suo meccanismo di elezione prevede, prima di diventare leader, un numero di voti positivi maggiore della metà della dimensione di rete. Questo genera ovvi problemi in caso il numero totale di nodi in crash, sia maggiore della metà della dimensione della rete.

Conclusioni

In reti piccole e stabili, dove le interazioni col leader sono poco frequenti, il meccanismo reattivo di Bully può risultare vantaggioso in termini di overhead. Tuttavia, rischia di diventare inefficiente con elezioni ripetute in caso di failure frequenti.
In reti distribuite su larga scala, il meccanismo di heartbeat di Raft combinato con l'utilizzo dei mandati, si dimostra più efficiente e scalabile, riducendo il rischio di cicli elettorali non necessari e garantendo un sistema di failure detection rapido e strutturato.

In sintesi, Bully è più leggero nelle reti piccole e meno attive, ma può generare troppe elezioni e quindi overhead in caso di failure ripetuti, rimanendo un algoritmo di facile implementazione. Raft, invece, bilancia meglio failure detection e overhead grazie al meccanismo di heartbeat, risultando più adatto a sistemi distribuiti di grandi dimensioni e frequenti failure, con non pochi ostacoli implementativi.