Gossip-Based Service Discovery and Failure Detection in Distributed Systems

Marchionni Edoardo
edoardo.marchionni@students.uniroma2.eu
matricola n: 0359614

***Abstract*—In questo documento vengono illustrati la progettazione e l'implementazione di un sistema distribuito, decentralizzato e scalabile, basato sul protocollo Gossiping Push-pull, con l'obiettivo di diffondere informazioni all'interno di una rete di nodi. Il progetto si concentra su due aspetti principali: i meccanismi di failure detection, che coprono tutti gli eventuali malfunzionamenti di un nodo, e la gestione dei servizi a essi associati, comprendendo le operazioni di aggiunta, rimozione e discovery.**

I. Introduzione

Il sistema è formato da N nodi tutti appartenenti alla stessa rete; il cui accesso è legato all'entità Registry, che si occupa di fornire una sotto-lista casuale di nodi che appartengono già ad essa. Ogni nodo detiene informazioni su tutti i nodi conosciuti; in particolare sul loro stato(Alive, Suspect, Dead) e sui servizi che offrono.
La comunicazione tra i nodi è implementata con un protocollo Gossiping di tipo Push-Pull ad intervalli di tempo, per garantire una convergenza maggiore delle informazioni.
Ogni messaggio inviato nel sistema ha una grandezza massima con la quale si intende il numero di eventi che verranno disseminati nel sistema, in modo da garantire un invio rapido senza sovraccaricare la rete e i nodi. Questo approccio decentralizzato è particolarmente adatto per ambienti di rete su larga scala, dove i guasti dei nodi sono inevitabili e la necessità di scalare facilmente il sistema è essenziale. Infatti, l'utilizzo di gossip permette una propagazione efficace delle informazioni, riducendo i costi di comunicazione e garantendo la robustezza della rete, anche in caso di nodi intermittenti o guasti.
Il service registry introduce un elemento di centralizzazione contrapposto al requisito di decentralizzazione del sistema; tuttavia, tale elemento viene sfruttato per informare, un nodo appena unito alla rete, quali sono gli altri nodi della rete e come contattarli. Una volta terminata l'interazione con il service registry, il nodo non interagirà in nessuna altra occasione con tale elemento.
Il progetto si propone di sviluppare un sistema in grado di gestire dinamicamente la disponibilità dei servizi e la gestione dei guasti, garantendo al contempo un'efficiente propagazione delle informazioni in una rete altamente distribuita. I principali obiettivi sono la scoperta dei servizi e il rilevamento dei guasti, due aspetti fondamentali per il corretto funzionamento e la resilienza di un sistema distribuito.

II. Background e Strumenti

A. Protocollo Gossiping

Il protocollo Gossiping è un meccanismo di comunicazione probabilistico, utilizzato per diffondere informazioni in sistemi distribuiti. Esso si basa sull'idea di scambiare informazioni tra nodi selezionati casualmente, in modo da ridurre l'overhead nella rete e garantendo una diffusione convergente nel tempo. Nel meccanismo di **Push-Pull Gossiping**, quello adottato in questo progetto, ogni nodo ne seleziona casualmente altri con cui scambiare informazioni. In questo approccio, il nodo invia prima le informazioni ai nodi target (Push), per poi richiedere l'invio del loro set di informazioni indietro (Pull). Questo scambio attivo permette una diffusione più rapida delle informazioni nella rete, in quanto ogni nodo non solo aggiorna gli altri, ma riceve anche aggiornamenti in cambio.
Gossiping vs flooding: Si preferisce utilizzare un protocollo come il gossiping, per la comunicazione in sistemi distruibiti fortemente scalabili, anzichè il flooding, poichè il primo riduce significativamente il numero di messaggi inviati grazie alla sua natura probabilistica; mentre il secondo prevede l'invio di tutte le informazioni appartenenti ad un nodo a tutti i nodi conosciuti. Pur essendo un protocollo semplice presenta un costo elevato in termini di risorse poiché ogni messaggio viene inoltrato a tutti i vicini, generando una notevole quantità di traffico di rete. La **scalabilità** del flooding è limitata, specialmente in reti molto grandi, dove il numero di messaggi da inviare cresce esponenzialmente con il numero di nodi, portanto molto overhead nella rete. In sintesi, mentre il flooding, essendo deterministico, riesce a garantire una completa diffusione delle informazioni entro pochi round, il gossiping risulta essere più efficiente in termini di risorse, ma con un trade-off in termini di **affidabilità e velocità di diffusione** che lo rendono più scalabile e quindi più adatto a reti con un elevato numero di nodi.

B. EC2 e Docker

Il software è implementato in linguaggio Golang ed è eseguito in container la cui orchestrazione è gestita grazie a Docker Compose. Il deployment si può effettuare sia in locale sia su una istanza remota EC2 dei servizi Amazon AWS. Sulla pagina GitHub dove è salvato il progetto, sono riportate tutte le informazioni per set up e lancio del software.

III. Design

A. Architettura Logica

L' architettura proposta è composta da 2 tipologie di container docker principali:
Registry: componente centralizzato dedicato solo alla fase di bootstrap dei nodi. Espone un api per il join dei nodi che restituisice un sotto lista randomica contenente gli ID già presenti nella rete e come contattarli.
Node: componente decentralizzato che, dopo il bootstrap dal Registry, mantiene *membership* e registro dei servizi e li dissemina via gossip *push–pull* su UDP. Esegue *heartbeat* e failure detection a timeout con quorum tra nodi per l'emissione dei death certifiacte. Inoltre espone le api per la ricezione di richieste per i servizi esposti (se li espone), ed ha anche un meccanismo di service discovery. Entrambe le tipologie di componenti sono configurabili tramite la modifica delle variabili a loro assegnate nel file .env in cui solo elecante le variabili assegante ad ogni componente replicato(nel caso dei nodi) e al componente centralizzato Registry.

B. Gossip

Il protocollo Gossip viene attivato periodicamente e si appoggia a tre loop principali:

- *gossipLoop* si occupa di avviare periodicamente, secondo il ***GOSSIP_INTERVAL***, un nuovo round richiamando sendGossip(): calcola un fanout dinamico, seleziona casualmente un sottoinsieme di peer non DEAD, costruisce un messaggio con il digest di *membership* e *services* (con eventuale riporto di voti e certificati) e lo invia via UDP; i destinatari fondono le novità nella propria vista e, se il messaggio non è una reply, rispondono solo con le differenze utili (*push–pull*) riducendo il traffico superfluo.
- *heartbeatLoop* si occupa di incrementare periodicamente, secondo l'***HEARTBEAT_INTERVAL*** il contatore monotono contenuto nell'entry della membership corrispondente a se stesso. Questo valore poi sarà riportato nei vari round di gossip segnalando che il nodo è ancora nel suo stato Alive. Solo il nodo proprietario della entry può aggiornare questo valore;

1) Scelta dei target: La selezione dei target inizia con la scansione della membership, la quale viene filtrata mantenendo solo i peer Alive; successivamente viene calcolato dinamicamente il numero dei target $k = \lfloor \log_2(\text{alive}) \rfloor + 1$ (con fallback a 1). Infine dalla lista dei peer alive ne vengono estratti k, questi sono i target a cui, nel prossimo round di gossip, verrano inviate le informazioni e conseguemente se ne riceveranno ulteriori seguendo il Push-Pull. L a conseguente operazione di invio viene protratta tramite UDP.

2) Definizione, composizione e gestione dei Gossip message: Il Gossip Message è una struttura JSON che incapsula tutte le informazioni di stato dei nodi e quelle dei servizi scambiate durante i vari round. Ogni messaggio contiene i seguenti campi:

from_id, *return_addr* (per le reply), *is_reply*, il digest di *membership* (lista aggiornamenti della membership), e quello dei *services*, ed eventuali liste di *votes* e *certs* (per i Death certificate).
Il digest di **membership** è costituito da una lista di *MemberSummary*, ordinata prioritariamente in base allo stato (nodi ALIVE prima), poi per *incarnation* e infine per *heartbeat* maggiore, limitata al numero massimo definito dalla variabile d'ambiente ***MAX_DIGEST_PEERS***.
Il digest di **services** è una lista di *ServiceAnnouncement* (servizio, instance_id, nodo, indirizzo, versione, TTL e stato attuale *Up*/ *Tombstone*), ordinata dando precedenza alle istanze attive, poi per versione con taglio a ***MAX_SERVICE_DIGEST***.
I digest di voti e di Death certificate vengono costruiti ad ogni round scartando quelli scaduti in modo da non diffondere troppo a lungo nella rete gli stessi voti troppe volte creando ridondanza. entrambi hanno una lunghezza limite definita dalla variabile d'ambiente corrispondente: ***MAX_VOTE_DIGEST*** e ***MAX_CERT_DIGEST***.
Durante la comunicazione i messaggi vengono costruiti tramite la funzione *buildMessage()* che include i digest aggiornati e un numero limitato di voti e certificati in base ai budget definiti dalle variabili nominate sopra. Il flag *is_reply* distingue le risposte dai messaggi di gossip ordinari, mentre *return_addr* serve a identificare l'indirizzo di ritorno per eventuali reply ottimizzate.

Alla ricezione di un Gossip Message, il nodo ricevente applica prima i certificati e i voti ricevuti, quindi filtra dal digest eventuali entry obsolete (*anti-resurrezione*), fonde le informazioni di *membership* e di *services* nella vista locale e, se il messaggio non è una reply, costruisce una risposta tramite *buildReplyDiff()* contenente solo le novità non già note al mittente.

Questa strategia *push–pull*, attivata periodicamente secondo il valore di ***GOSSIP_INTERVAL***, consente una disseminazione bidirezionale efficiente, riducendo il traffico e garantendo la convergenza eventuale della rete.

3) Merge delle informazioni:

- **Membership**: La fusione, svolta nella funzione *mergeMembership()*, integra eventuali nuovi mittenti e, per ciascun membro già noto, applica l'aggiornamento solo se la coppia $\langle incarnation, heartbeat \rangle$ ricevuta è strettamente più recente di quella locale; in tal caso imposta `State = ALIVE`, aggiorna `LastSeen` e, se necessario, allinea indirizzo/porta. Gli update obsoleti o duplicati vengono ignorati (le entry più vecchie di un certificato di morte attivo sono filtrate in ingresso).
- Services: La fusione delle informazioni di servizio (*mergeServices()*) usa versioni monotone per chiave $\langle \text{service}, \text{instance} \rangle$ (lastSvcVer) per scartare vecchie versioni del servizio; ogni annuncio ricevuto imposta Up/Tombstone, TTLSeconds e la scadenza locale `ExpiresAt`. Gli aggiornamenti con versione non strettamente minore di quella attuale sono ignorati.

C. Failure Detection

Il sistema di failure detection viene implementato tramite una rilevazione basata su timeout e disseminazione delle transizioni tramite gossip. Questo meccanismo è integrato principalmente nel lopp chiamato **suspicionLoop()** che si occupa di controllare periodicamente, secondo il ***SUSPICION_INTERVAL***, il campo lastseen di ogni membro appartenente alla membership per, eventualmente, considerare un cambiamento di stato dei singoli nodi. Nel caso non si ricevano aggiornamenti entro un Δ (***SUSPECT_TIMEOUT***) di tempo lo stato di un nuovo viene prima cambiato da $Alive \xrightarrow{transition} Supect$; e nel caso non si ricevino aggiornamenti per un tempo molto elevato (***DEAD_TIMEOUT***) allora il nodo verrà considerato morto scatenando $Supect \xrightarrow{transition} Dead$. Infine questo loop si occupa anche dell'emissione dei voti locali e dell'eliminazione dei servizi non più, considerati, raggiungibili tramite la funzione *pruneExpiredServices()*;

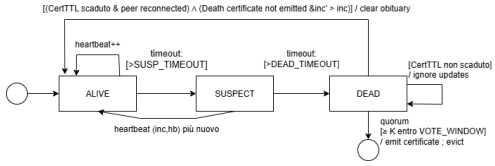

Fig. 1. Macchina a stati meccanismo failure detection

1) Heartbeat e Incarnation monotone: Ogni nodo incrementa periodicamente un contatore *heartbeat* nella propria entry di membership; l'update viene diffuso nei round di gossip. Il merge accetta un update solo se la coppia $\langle inc, hb\rangle$ ricevuta è strettamente più nuova di quella locale.

2) SuspectTransition: La *suspicion* percorre periodicamente la membership e, superata la soglia `SUSPECT_TIMEOUT` sul $\Delta = now - LastSeen$, porta lo stato del peer da ALIVE a SUSPECT. Questo passaggio riduce i falsi positivi dovuti peer intermittenti o con una latenza di rete elevata, poichè evita che la transizione a Dead avvenga troppo in fretta diffondendo informazioni false o non coerenti con il resto della rete.

3) Dead Transition: Se il Δ supera `DEAD_TIMEOUT`, il peer transita a DEAD. Contestualmente si congela l'evidenza $\langle inc, hb\rangle$ osservata in DEAD, che sarà riportata nel voto locale e, tramite gossip, contribuirà al raggiungimento del quorum per l'emissione di un certificato di morte.

4) Voti e Death Certificate: Alla prima osservazione di DEAD su un peer X, il nodo emette un voto locale contenente la prova di morte $\langle inc, hb\rangle$ e la sua firma di *witness*. I voti sono aggregati in bucket con finestra di validità (`VoteWindow`) e priorità di diffusione in modo che non vengano creati duplicati, o rimangano troppo a lungo, bucket dei voti. Raccogliendo voti da witness distinti, al raggiungimento di $Quorum\_K$, viene emesso un *Death Certificate* anch'esso con una durata limitata (`CertTTL`) e priorità dedicata alla diffusione. Un certificato appena viene ricevuto viene gestito nel seguente modo:

1) *eviction* dell'entry target dalla membership;
2) attivazione del filtro *anti-resurrezione* che scarta update meno recenti di $\langle inc, hb\rangle$ finché il certificato è valido.

5) Recover: Gestiamo il rientro di un peer intermittente o riavviato senza introdurre rumore dato da viste precendeti né continui falsi positivi.

- Rientro senza certificato in circolazione Se il peer, durante i round di gossip, recupera informazioni di sé marcate come DEAD, e ancora non è stato emesso un *Death Certificate* valido; allora esegue un *self-bump* di incarnation e diffonde una coppia $\langle inc, hb\rangle = \langle inc_{\text{prev}}{+}1,\ 0\rangle$. Alla ricezione di questa evidenza strettamente più nuova, gli altri nodi aggiornano la membership riportandolo a ALIVE evitando la fase di votazione e di conseguente eviction.
- Rientro con certificato attivo Se in rete circola un *Death Certificate* valido sul peer, la riammissione non è immediatamente possibile: ogni nodo che riceve tale certificato applica un filtro *anti-resurrezione* che scarta qualunque update su quel peer non strettamente più nuovo dell'evidenza contenuta nel certificato. In questo scenario, il nodo attende la scadenza del certificato e quindi riparte con l'aspetto di un nodo appena aggiunto, annunciando $\langle inc, hb\rangle = \langle 1,\ 0\rangle$ e procedendo normalmente con la diffusione del proprio stato.

a) Perché non può "smentire" un certificato attivo: Il *Death Certificate* rappresenta un accordo di rete a tempo determinato (certTTL) che congela lo stato del peer per garantire coerenza globale durante partizioni; inoltre garantisce stabilità delle informazioni evitando rimbalzi DEAD/ALIVE dovuti a peer intermittenti.Per questo motivo, finché il certificato è valido, gli update provenienti dal peer non possono riabilitarlo; solo la sua scadenza rimuove il filtro e consente il rientro.

D. Gestione dei servizi

I servizi sono gestiti come *istanze* identificate da $\langle service, instance\_id\rangle$, con stato `Up`/`Tombstone`, versione monotona (`ver`) e scadenza locale (`TTL`). Solo il nodo proprietario può aggiornarle; i peer applicano gli update ricevuti via gossip solo se la versione è strettamente più nuova.

1) Tipologie di servizi ed inizializzazione: Nel sistema coesistono due classi di servizi:

- *Servizi interni*: attualmente il solo `calc` (somma/sottrazione). Questo servizio essere inizializzato (i) all'avvio, se indicato all'internod della variabile d'ambiente `SERVICES_DEFAULT`, oppure (ii) alla prima chiamata `/service/register`. La registrazione crea/aggiorna l'istanza locale e dissemina l'aggiunta via gossip.
- *Servizi esterni*: applicazioni non inclusi nel nodo. Si registrano tramite `/service/register`; il nodo rende disponibile l'istanza sull'indirizzo fornito e ne diffonde lo stato via gossip fintanto che si ricevano informazioni sul servizio prima della sua scadenza; a quel punto verrà considerato non disponibile ed aggiornato lo stato come *Tombstone*.

2) Service API:

- **POST /service/register** Crea/aggiorna un'istanza locale: `service`, `instance_id`, `addr`, `ttl_seconds`. Imposta `Up=true`, incrementa `ver`, rinnova `ExpiresAt`.
- **POST /service/deregister** Marca `Tombstone` un'istanza locale (`Up=false`, `ver++`).
- **GET /services/local** Elenco delle istanze locali con metadati.

3) Service-Discovery: L'endpoint $/discover?service = NAME$ restituisce, sulla base della vista corrente del nodo, solo le istanze che soddisfano *tutte* le seguenti condizioni:

- stato $Up = true$;
- TTL non scaduto (now $<$ ExpiresAt);
- nodo ospitante in stato ALIVE nella membership;

l'insieme risultante, che può anche essere vuoto rappresenta i target in cui è possibile invocare il servizio cercato.

E. Garbage Collector

Il nodo esegue due routine di *garbage collection*, una per i servizi e una per i metadati di failure detection, così da mantenere la vista locale compatta e priva di informazioni scadute.

a) Servizi: Ad ogni tick del `suspicionLoop` il nodo invoca `pruneExpiredServices()`. Per ogni istanza $\langle service, instance\_id \rangle$:

1) se è `Up` e non `Tombstone` ma ha superato la scadenza locale `ExpiresAt`, viene portata a `DOWN`; se l'istanza è locale, il nodo incrementa la `ver` in modo monotono prima della transizione, così che l'aggiornamento sia accettato in gossip;
2) qualunque istanza la cui `LastUpdated` è più vecchia di $2\times$`TTL` viene rimossa dalla mappa dei servizi. Questo evita di mantenere per lungo tempo annunci non più rinfrescati e limita la memoria.

b) Voti e Death certificate: Ad ogni tick del gossipLoop il nodo invoca gcDeathMeta(): i bucket di voti scaduti oltre VoteWindow vengono eliminati e i Death Certificate oltre CertTTL vengono rimossi. In questo modo si impedisce che i digest si gonfino con metadati obsoleti e si limita la finestra del filtro anti-resurrezione.

c) Pulizia in caso di eviction: Quando un certificato valido causa l'*eviction* di un peer, la routine elimina anche tutte le istanze di servizio ospitate da quel peer, garantendo che la discovery non restituisca istanze non valide.

IV. CONCLUSIONI

Abbiamo progettato e implementato un servizio gossip-based per service discovery e failure detection con:

- Disseminazione push–pull su UDP e fanout dinamico log2(N)+1, che bilancia copertura e overhead.
- Failure detector a timeout integrato con quorum e Death Certificate a scadenza (anti-resurrezione), con rimozione coerente dei servizi dei nodi evicted.
- API HTTP leggere per registrare/deregistrare istanze e per la discovery, con filtri di validità (TTL, stato del nodo).
- Garbage collection per servizi e metadati di voti/certificati.

Il design rispetta i requisiti di decentralizzazione (a eccezione del solo bootstrap via registry), scalabilità e fault tolerance richiesti dal progetto, ed è configurabile via env e deployabile con Docker Compose in locale o su EC2.

V. LAVORI FUTURI

Il sistema può essere esteso e migliorato in vari modi:

1) *Registry centralizzato scalabile*: Manteniamo il registry come punto di bootstrap ma lo rendiamo affidabile e scalabile eseguendone 2–3 istanze dietro un load balancer. Per robustezza operativa il registry effettua snapshot leggeri su disco per un ripristino istantaneo in caso di guasti. In questo modo rimane "centralizzato" dal punto di vista logico, ma non è più un singolo punto di fallimento e garantisce un bootstrap affidabile anche sotto guasti o picchi di carico.
2) *Suspicion timeout dinamico*: un numero elevato di *incarnation* indica riavvii/intermittenze oppure collegamenti ad alta latenza (p.es. distanza geografica). Con un `SUSPECT_TIMEOUT` fisso quel nodo finirebbe spesso in `DEAD`, ostacolando comunicazione e diffusione degli aggiornamenti. Una soluzione quindi potrebbe essere l'inserimento di un timeout *adattivo*: se l'*inc* è alto, la soglia viene estesa temporaneamente (per un numero limitato di tentativi) così da ridurre i falsi positivi.
3) *Autenticazione dei messaggi gossip (HMAC)*: si può anche pensare di introdurre l'autenticazione dei frame UDP: ogni messaggio gossip porterà un tag HMAC-SHA256 calcolato su $header||payload||seqno$, dove *seqno* è un contatore monotono per mittente (o un nonce univoco) per mitigare i replay da parte di peer malevoli. Ogni coppia di nodi condividerà una chiave simmetrica $K_{i,j}$, derivata durante la fase di bootstrap tramite handshake, che include anche la derivazione delle chiavi. In ricezione di un Gossip Message, il nodo identificherà il mittente, selezionerà $K_{i,j}$ e verificherà l'HMAC, validando anche seqno/nonce per scartare duplicati e replay. I pacchetti con MAC invalido o seqno non valido verranno scartati.