Gossip-Based Service Discovery and Failure Detection

Leonardo Polidori (0357314)
Università Tor Vergata
leonardo.polidori@students.uniroma2.eu

***Abstract*—Il *service discovery* richiede diffusione rapida ma sostenibile. Il *flooding* garantisce copertura deterministica ma scala male; i protocolli *epidemici* accettano copertura probabilistica con costo per nodo quasi costante e buona resilienza a perdita e churn. Presentiamo un sistema leggero in Go con control plane bimodale: (i) *heartbeats* one–hop (light/full) per liveness e snapshot, (ii) *rumor mongering* multi–hop per lookup ed eventi di guasto con (TTL, B, F) e dedup, (iii) round *anti–entropy* push–pull opzionali per riconciliazione. Il registry è *seed-only* e resta fuori dal data path dei lookup. Usiamo Docker/Compose per avviare topologie ripetibili, esplorare i parametri e iniettare guasti..**

***Index Terms*—gossip, service discovery, failure detection, anti-entropy, rumor spreading, Docker, Go**

I. Introduzione

Nei sistemi distribuiti, la scoperta dei servizi deve bilanciare latenza, costo di rete e robustezza. I registry centralizzati semplificano la coerenza ma introducono un single point of failure; il *flooding* offre copertura elevata a fronte di traffico esplosivo. I protocolli epidemici (gossip) spostano l'equilibrio: accettano copertura probabilistica in cambio di un costo per nodo quasi costante e migliore tolleranza a perdita e churn.

Adottiamo un **control plane bimodale**: heartbeat *one-hop* (light/full) per liveness e snapshot locali; rumor *multi-hop* con (TTL, B, F) e dedup per propagare solo gli eventi che devono diffondersi (lookup e transizioni `Suspect/Dead/Leave`); una fase opzionale di *anti–entropy* (push–pull) per riconciliare divergenze. Il registry resta *seed-only* e fuori dal data path dei lookup.

Contributi:

- Architettura *seed-only* con diffusione epidemica di membership e servizi.
- *Lookup* epidemico con TTL/fanout/budget e negative cache con invalidazione a tempo.
- Failure detector a heartbeat con rumor di stato e promozione parametrizzabile (timeout/quorum).
- Metodologia sperimentale in Docker/Compose.

II. Fondamenti: Gossip, Rumor, Anti-Entropy, Failure Detection

A. Epidemic Gossiping (push, pull, push–pull)

Nel gossip ogni nodo scambia periodicamente piccoli frammenti di stato con vicini (log n) scelti casualmente. Le tre varianti canoniche sono: *push* (il mittente invia lo stato), *pull* (il mittente chiede lo stato) e *push–pull* (scambio bidirezionale). Su grafi sufficientemente connessi, la diffusione di un aggiornamento avviene in $O(\log n)$ round; l'overhead per–nodo resta quasi costante e controllato dal fanout per round B e dai periodi dei timer, piuttosto che dalla dimensione del sistema n [1].

B. Rumor mongering e regole di arresto

Il *rumor mongering* (o *rumor spreading*) diffonde eventi "caldi" (es. richieste di lookup, notifiche di stato) in modo probabilistico/limitato. Per evitare tempeste di messaggi si usano tre manopole:

- **TTL** (hop massimi): limita la profondità della propagazione;
- **Fanout** B: numero di vicini contattati ad ogni hop;
- **Budget** F: limite ai reinvii per *rumor ID* (dedup) o, in varianti equivalenti, una probabilità di continuazione p [2].

Ogni nodo mantiene la memoria dei rumor visti con un contatore; superato F smette di inoltrare. Questo taglia la ridondanza rispetto al flooding mantenendo alta la probabilità che almeno un cammino raggiunga un provider. *Nota.* Nel seguito useremo "**rumor**" per le propagazioni event-driven (lookup, `Suspect/Dead/Leave`) e "**anti–entropy**" per la riconciliazione periodica push–pull. Per le proprietà di complessità e copertura rimandiamo alle formulazioni classiche [1], [2].

C. Anti–entropy (riparazione periodica)

L'anti–entropy è una riconciliazione periodica *time–driven* dello stato tra coppie casuali di nodi: ci si scambia un *digest* (riassunto compatto di versioni/epoch) e, solo se emergono divergenze, si trasferisce lo stato mancante via *push–pull*. A differenza del rumor (che è *event–driven*), l'anti–entropy garantisce *eventual consistency* [1], [3].

D. Bimodalità: rumor veloce + riconciliazione

Combinando rumor mongering (per gli eventi) e anti–entropy (per lo stato) si ottiene una diffusione *bimodale*: propagazione rapida e probabilistica quando serve reattività, con una riparazione periodica che chiude i buchi residui e abbatte la varianza. È il pattern adottato qui: heartbeat/light/full e (opzionale) repair periodico per allineare membership/servizi, rumor per lookup e segnalazioni di guasto.

E. Failure detection a heartbeat

In sistemi parzialmente sincroni, heartbeat periodici aggiornano la *liveness*; il mancato arrivo entro una soglia induce stato di *suspicion*, e il protrarsi oltre un secondo timeout porta

allo stato *dead*. Parametri troppo aggressivi aumentano i falsi positivi; parametri conservativi rallentano la rimozione dei nodi guasti. Nel progetto si usa un FailureDetector a soglia temporale, con rumor epidemici `Suspect/Dead/Leave` che si propagano con le stesse manopole (B, F, TTL) del rumor; l'anti–entropy e gli heartbeat *full* contribuiscono a ripristinare/propagare la vista corretta [4]. Inoltre adottiamo un **quorum dinamico** per la promozione `Suspect`→`Dead`, calcolato come maggioranza di *peer vivi entro* `DEAD_TIMEOUT`.

III. Docker per la sperimentazione

Docker è una piattaforma di containerizzazione che permette agli sviluppatori di impacchettare applicazioni e tutte le loro dipendenze in unità standardizzate chiamate container. Questi container possono essere eseguiti in modo rapido, affidabile e coerente su qualsiasi macchina, sia locale che nel cloud, semplificando la creazione, il test e il rilascio del software e garantendo l'isolamento e l'indipendenza dell'applicazione dall'ambiente sottostante. Nel progetto lo usiamo per (i) descrivere la topologia in modo dichiarativo con `docker compose` (cfr. [6]), (ii) scalare localmente il numero di nodi per esplorare i parametri (B, F, TTL), e (iii) iniettare guasti controllati (stop/restore dei container) per misurare l'impatto su gossip e failure detection (cfr. [5]).

La topologia del progetto comprende: (i) un *registry* minimale usato solo per fornire *un* peer di bootstrap (TCP 9000), esplicitamente escluso dal *data path* dei lookup; (ii) N nodi gossip che comunicano su UDP (porte 9001–9006) e ospitano i provider; (iii) un *client* effimero che esegue il *lookup* (UDP 9009). Tutti i container condividono la stessa rete `bridge` di Compose e si risolvono per nome (`registry`, `node1`, ...). Parametri e *feature flags* (heartbeat, repair, caching) sono esternalizzati in `.env` per rendere gli esperimenti ripetibili; i file `compose` e il codice sono disponibili nell'artifact del progetto [7].

IV. Progetto SDCC: obiettivi, flussi e scelte

A. Obiettivi

Obiettivi: (i) service discovery decentralizzato dopo un bootstrap minimo; (ii) robustezza a churn/guasti;

B. Architettura e flussi (sintesi)

Il registry è solo *seed*. L'overlay e la vista emergono via gossip.

- **HB light (one–hop):** piggyback di peer–hints + metadati (epoch, svcver); aggiorna `lastSeen`.
- **HB full:** snapshot (servizi+peer) quando serve coerenza più forte (es. cambio servizi) e come risposta ai repair.
- **Lookup rumor:** il client diffonde (TTL, B, F); il primo provider risponde *direct-to-origin*. Negative cache temporanea sul miss.
- **Failure handling:** oltre `SUSPECT_TIMEOUT` si emette `Suspect`. La promozione a `Dead` avviene *solo* al raggiungimento del **quorum dinamico** $\lfloor(\mathsf{alivePeerCount}() + 1)/2\rfloor + 1$, dove alivePeerCount conta i peer con *HB* entro `DEAD_TIMEOUT` (alive window) escluso il nodo stesso. Si applicano *tombstone* e rumor `Dead`. `Leave` applica direttamente `Dead+tombstone`. Il revival è accettato solo con metadati più recenti della tombstone.
- **Repair (anti–entropy) opzionale:** scambio push–pull on–demand (o periodico a flag) basato su digest verso un sottoinsieme casuale di vicini vivi.

a) Flussi in breve.: **Bootstrap:** seed dal registry, avvio HB light e ampliamento via piggyback. **Mutazioni servizi:** `ADD/DEL` → bump (epoch/ver) → HB full push. **Lookup:** dedup per `reqId`, risposta se provider vivo, forward a B vicini (TTL–), miss → neg-cache. **FD:** `Suspect` dopo `SUSPECT_TIMEOUT`, `Dead` per quorum; filtro di freschezza evita `Dead` se arriva un HB recente. **Repair:** HB full su richiesta o periodico se abilitato.

C. Feature flags e parametri runtime

a) Sintesi.:

- **`LearnFromHB`**: se attivo, gli *HB(full)* ricevuti aggiornano *Service View* e *Peer View*; se disattivo, gli HB(full) sono ignorati (resta solo la liveness dagli *HB(light)*).
- **`LearnFromLookup`**: se attivo, il *client originator* apprende il mapping `service` → `provider` dalla prima `LookupResponse` e lo memorizza localmente.
- **`RepairEnabled`**: abilita l'*anti–entropy* periodica (push–pull); efficace solo con `LearnFromHB=true` (il ricevente deve accettare i *full*).

b) Interazioni pratiche.:

- `RepairEnabled=true` & `LearnFromHB=true`: repair *efficace*, i tick push–pull periodici allineano le viste.
- `RepairEnabled=true` & `LearnFromHB=false`: repair *inefficace*, i *full* circolano ma il ricevente non apprende.
- `LearnFromLookup=true`: la prima risposta al *lookup* aggiorna la mappa locale *service→provider* del richiedente, riducendo la latenza dei lookup successivi.
- `LearnFromLookup=false` & `LearnFromHB=false`: nessun apprendimento passivo dei provider; la discovery dipende solo da risposte dirette o aggiornamenti espliciti.
- **Negative cache** (`SDCC_LOOKUP_NEGCACHE_TTL`): quando un rumor di lookup esaurisce il *TTL* senza trovare un provider, il nodo mette il servizio in *neg-cache* per un intervallo finito e lascia cadere richieste uguali fino a scadenza; la cache si invalida allo scadere del TTL.

c) Altri flag e timer (riassunto).:

- **Rumor-mongering (FD):** `SDCC_FD_B` = fanout B; `SDCC_FD_F` = budget F per *rumor ID*; `SDCC_FD_T` = *TTL* (hop).
- **Heartbeats:** `SDCC_HB_LIGHT_EVERY` (periodo *light*); `SDCC_HB_FULL_EVERY` (periodo *full*); `SDCC_HB_LIGHT_MAX_HINTS` (max peer–hints nei *light*riduce il payload degli HB(light) ma rallenta la diffusione dei peer via hint).
- **Failure detector (timeout):** `SDCC_SUSPECT_TIMEOUT` e `SDCC_DEAD_TIMEOUT` per le soglie *Suspect*/*Dead*.

- **Repair (anti–entropy):** `SDCC_REPAIR_ENABLED` abilita il push–pull periodico; `SDCC_REPAIR_EVERY` è l'intervallo.
- **Lookup:** `SDCC_LOOKUP_TTL` = profondità del rumor di lookup; `SDCC_LOOKUP_NEGCACHE_TTL` = durata della *negative cache*.
- **Osservabilità e client:** `SDCC_CLUSTER_LOG_EVERY` = periodo del log "`>> Cluster ...`"; `SDCC_CLIENT_DEADLINE` = deadline della modalità *client one-shot* per attendere la `LookupResponse`.
- **Registry bootstrap:** `SDCC_REGISTRY_MAX_ATTEMPTS` = tentativi massimi di bootstrap verso il registry.
- **RPC (servizi demo):** `SDCC_RPC_A`, `SDCC_RPC_B` controllano i parametri dei servizi aritmetici d'esempio.

V. Ambiente sperimentale in Docker

A. Topologia e deployment

Usiamo `docker compose` per avviare una topologia minimale e ripetibile:

Tutti i container stanno sulla rete `bridge` di Compose (DNS interno: `registry`, `node1`, ...). L'aggiunta/rimozione di servizi avviene tramite file di controllo (`ADD`/`DEL`) e innesca un gossip *full*. Timer e flag sono in `.env` per consentire sweep controllati.

VI. Pseudo-codice dei flussi chiave

Algorithm 1 Lookup rumor (lato nodo)

[1]
on LOOKUP(reqId, $svc, ttl, B, F, origin$) **from** s
if dedup[reqId] $\geq F$ **then return**
end if
dedup[reqId] $\leftarrow$ dedup[reqId] $+ 1$
if knownProvider(svc) **and** isAlive(provider) **then**
  if dedup[reqId] $== 1$ **then**
    send LOOKUPRESPONSE(reqId, provider) **to** origin
  end if
  return ▷ rispondo e *mi fermo* (niente forward)
end if
if $ttl == 0$ **then**
  negativeCache.insert(svc, now $+ \Delta$) ▷ $\Delta =$ `LOOKUP_NEGCACHE_TTL`
  return
end if
$ttl \leftarrow ttl - 1$
$E \leftarrow \{s, \text{origin}, \text{self}\}$ ▷ evita eco e ping–pong
$N \leftarrow$ randomAliveNeighbors(B, exclude $= E$)
for all $n \in N$ **do**
  forward LOOKUP(reqId, $svc, ttl, B, F, origin$) **to** n
end for

Algorithm 2 Heartbeat light/full e failure rumors (compatto)

[1]
every T_ℓ: **send** HB_LIGHT($hints, meta$) **to** randomFanout() ▷ $\approx \log n$
every T_f: **send** HB_FULL($services, peers, meta$) **to** randomFanout()
on HB from p: $lastSeen[p] \leftarrow now$
periodicamente:
  $need \leftarrow \left\lfloor \frac{\text{alivePeerCount}() + 1}{2} \right\rfloor + 1$
for all p **do**
  if $now - lastSeen[p] >$ `SUSPECT_TIMEOUT` **and** $state[p] = Alive$ **then**
    rumor(SUSPECT(p),B, F, TTL); **quickProbe**(p)
  end if
  if suspectVotes$[p] \geq need$ **and not** recentHB(p) **then**
    $state[p] \leftarrow Dead$; purgeProviders(p); tombstone(p)
    rumor(DEAD(p),B, F, TTL)
  end if
end for
on LEAVE from p:
  $state[p] \leftarrow Dead$; purgeProviders(p); tombstone(p); **rumor**(DEAD(p),B, F, TTL)
on RUMOR(ev, p, ttl, B, F, id) **from** s:
if dedup$[id] \geq F$ **or** $ttl < 0$ **then return**
end if
dedup$[id] \leftarrow$ dedup$[id] + 1$
if $ev =$ SUSPECT **and not** recentHB(p) **then** addVote(p, s)
end if
if $ev \in \{$DEAD, LEAVE$\}$ **and not** recentHB(p) **then**
  $state[p] \leftarrow Dead$; purgeProviders(p); tombstone(p)
end if
if $ttl > 0$ **then**
  $ttl \leftarrow ttl - 1$; **forward** RUMOR(ev, p, ttl, B, F, id) **to** B random neighbors $\setminus \{s, p, \text{self}\}$
end if

VII. Discussione critica

Riassumiamo i principali trade–off progettuali con rischi e contromisure operative.

- **Negative cache.** *Pro*: contiene tempeste su miss ripetuti. *Contro*: rischio *starvation* se un servizio nasce subito dopo il miss.
- **Failure detector (HB one–hop).** *Pro*: semplice e prevedibile. *Contro*: falsi positivi sotto jitter/loss; sensibilità a skew. *Mitigazione*: timeouts proporzionali a HB(light); quorum per promuovere `Dead`.
- **Rumor per eventi** (B, F, TTL)**.** *Pro*: copertura rapida con costo per nodo quasi costante. *Contro*: copertura insufficiente su overlay sparsi o traffico eccessivo. *Mitigazione*: tuning per profilo di rete [1].
- **Anti–entropy (repair) push–pull.** *Pro*: chiude buchi informativi; garantisce convergenza eventuale. *Contro*: overhead periodico in quiete. *Mitigazione*: $T_{\text{repair}} \gg T_{\text{HB(full)}}$.
- **Servizi *owner–only*.** *Pro*: implementazione semplice; coerenza locale forte; misure più pulite. *Contro*: disponibilità

più bassa; latenza più variabile; interazione più delicata con la negative cache..
- **Schema bimodale (gossip + repair).** *Pro*: reattività agli eventi + riconciliazione periodica; robustezza a loss/churn. *Contro*: tuning delicato di intervalli e parametri.
- **Quorum per `Suspect`→`Dead`.** *Pro*: riduce falsi positivi e rumor isolati. *Contro*: rilevazione più lenta; rischio di stallo in partizioni piccole.

VIII. Minacce alla validità

Sicurezza (minacce & mitigazioni):
- *DoS/DDoS sul control-plane:* flood di join/ping/ping-req e gossip che saturano CPU/banda. **Mitigazione:** rate limiting per peer, circuit-breaker, code prioritarie, verifiche firma prima di lavori costosi, fanout e payload massimi.
- *Amplificazione/Reflection (UDP):* il nodo risponde più di quanto riceve. **Mitigazione:** cookie di ammissione/challenge-response, no risposte ad indirizzi non verificati.
- *Spoofing:* identità/nodo falsi o molte identità per inquinare il cluster. **Mitigazione:** NodeID = hash della chiave pubblica, mTLS/PKI (SPIFFE/SPIRE), join token/whitelist.
- *MITM:* intercettazione e modifica dei messaggi. **Mitigazione:** canale autenticato e cifrato.
- *Replay/Downgrade:* reinvio di record vecchi o versioni inferiori. **Mitigazione:** versioni monotone (Lamport/contatore), TTL brevi, nonce/anti-replay cache; i tombstone prevalgono.

IX. Campagna sperimentale

A. Setup e metodologia

Gli esperimenti sono stati eseguiti usando la Topologia introdotta nella sezione V Ambiente Sperimentale in Docker. La distribuzione dei servizi nei log di riferimento è:
- `node1`: `sum` `node3`: `div` `node4`: `mul` `node2`, `node5`: `sub`.

I parametri, caricati da `.env`, salvo diversa indicazione, sono:
- Rumor-mongering FD: B=3, F=2, T=3; HB light/full: 3s/9s.
- Timeouts FD: `Suspect`= 30s, `Dead`= 36s.
- Lookup: `TTL`= 3, `LEARN_FROM_LOOKUP`∈ {`true`}, `LEARN_FROM_HB`∈ {`true`}.
- Repair anti–entropy: `REPAIR_ENABLED`∈ {`false`}, periodo 30s.

B. Casi sperimentali

Abbiamo variato alcuni flag e alcune configurazioni dei nodi:

a) E1 — Baseline (HB-learn + Lookup-learn, no repair): I client eseguono lookup con `TTL=3`, B=3, F=2. Osservazioni:
- **Latenza:** risposta rapida; nei log di esempio la `LookupResponse` arriva nella stessa seconda della `TX Lookup`.
- **Cache locale:** il client aggiorna la mappa (*"registry updated: svc → provider"*) e invoca l'RPC.
- **HB light/full:** propagano metadati e snapshot servizi; aggiunte/rimozioni forzano `full` immediato.

b) E2 — Disabilitare l'apprendimento da HB (`learnFromHB=false`): Ogni nodo *non* chiede il `HB(full)` quando riceve `HB(light)` sospetti/obsoleti; nei log si vede: *"[REPAIR] skip: learnHB=false → non richiedo HB(full) a ..."*.
- **Effetto:** la convergenza della *service view* dipende dai rumor di lookup e dalle risposte (se `learnFromLookup=true`).
- **Discovery:** il lookup continua a funzionare (il provider risponde direttamente al client), ma l'allineamento passivo via HB è ridotto.

c) E3 – Crash (fail-stop).: Azione: `docker kill -s SIGKILL nodeX`. *Atteso*: entro `SUSPECT_TIMEOUT` rumor `Suspect`, quindi `Dead` per quorum. Durante la tombstone, rumor/HB obsoleti sono ignorati.

d) E4 – Leave volontario.: Azione: `docker stop nodeY` (il processo invia `Leave`). *Atteso*: applicazione immediata di `Dead+tombstone` senza fase `Suspect`; propagazione rumor con (B, F, TTL).

e) E5 – Rientro.: Azione: riavvio di `nodeX` con metadati (epoch/ver) maggiori della tombstone. *Atteso*: `HB(full)` accettato, transizione da guardia a `Alive`; in caso contrario HB scartati fino a bump dell'incarnation.

f) E6 – Join a caldo.: Con registry: nodo richiede un seed (`registry:9000`) e avvia HB/light; vista converge in pochi cicli. *Senza registry*: bootstrap diretto su un vicino vivo (`--bootstrap-peer=node2:9002`); semantica identica, niente SPOF sul path dei lookup.

X. Conclusioni e lavori futuri

Abbiamo descritto un service discovery decentralizzato in Go che unisce rumor mongering, anti–entropy e failure detection a heartbeat con orchestrazione Docker. **Lavori futuri:**
- Difesa contro DoS/DDoS, spoofing, MITM, replay.
- Auto–tuning di gossip/FD guidato da misure online di RTT e perdita.
- Scalabilità/robustezza: stress test a n elevato.

References

[1] A. Demers et al., "Epidemic algorithms for replicated database maintenance," 1988.
[2] R. Karp et al., "Randomized rumor spreading," 2000.
[3] W. Vogels, "Eventually Consistent," 2009.
[4] N. Hayashibara et al., "The ϕ-Accrual Failure Detector," 2004.
[5] Docker Inc., "Docker Engine Overview," https://docs.docker.com/engine/.
[6] Docker Inc., "Compose file reference (Compose Specification)," https://docs.docker.com/compose/compose-file/.
[7] L. Polidori, "SDCC_Gossiping (artifact e sorgenti)," https://github.com/Polidori1999/ProgettoSDCC.