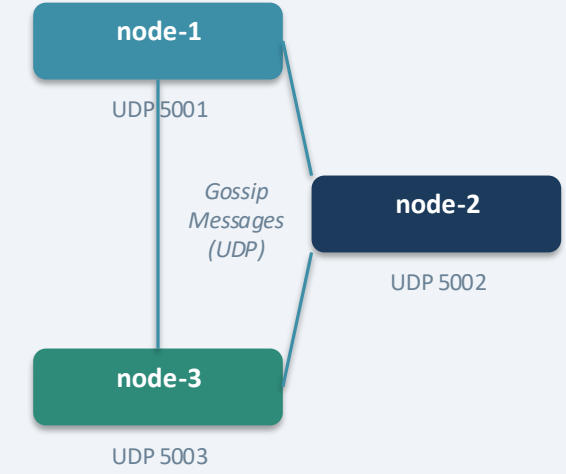
Gossip-Based Distributed Monitoring System

Sistema di monitoring decentralizzato basato su protocolli epidemici

Presentazione progetto · DS 2025/2026

Thomas Testa



SWIM-inspired | asyncio

Python 3.11 | Docker

Problema affrontato e obiettivi di sistema

Un servizio di monitoring distribuito che resta operativo con guasti e disconnessioni

Perché distribuire?

- monitor centralizzato(Approcio classico) = single point of failure
- crash failures non osservabili direttamente: inferiti via heartbeat timeout
- rete inaffidabile: packet loss, ritardi, partizioni
- nessuna memoria condivisa, nessun clock sincronizzato
- scalabilità: O(k) messaggi per nodo per round

Modello funzionale

Node ▶ Gossip ▶ Peers

- ogni nodo mantiene Membership Table locale
- heartbeat incrementale ogni gossip round
- push e push-pull dissemination via UDP
- stati: ALIVE → SUSPECT → DEAD
- eventual consistency, no shared DB

AP / BASE | Eventual consistency | peer-to-peer

Modellazione del sistema

Componenti, connettori, dati e stile architetturale

Componenti (Nodo)

- Network Layer (UDP socket)
- Gossip Engine (fanout k)
- Membership Manager
- Failure Detector
- FastAPI layer (REST)

Connettori

- UDP datagram (gossip push/pull)
- JSON serialization
- heartbeat periodico
- FastAPI REST (osservabilità)
- asyncio event loop

Dati

- node_id, heartbeat, incarnation, status: ALIVE/SUSPECT/DEAD, updated_at timestamp
- **Gossip Message** (JSON array)
- **Rumor Event** (rumor_id, origin)

Domain events: Gossip Round Triggered · Heartbeat Incremented · Membership Update Merged · Node Suspected · Node Declared Dead · Rumor Generated / Received

Membership Management

Ogni nodo mantiene una visione locale del cluster

Membership Table (per nodo)

`node_id`	identificatore univoco del nodo
`heartbeat`	contatore monotono incrementale
`incarnation`	versione al riavvio del nodo
`status`	ALIVE / SUSPECT / DEAD
`updated_at`	timestamp ultimo aggiornamento

Merge Policy

R sostituisce L se (equazione dal report):

```
(R.hb > L.hb)  V  (R.inc > L.inc)  V
(R.status = DEAD)
```

1. **Higher Heartbeat**
   attività più recente vince
2. **Higher Incarnation**
   nodo riavviato → stato fresco vince (refuta SUSPECT)
3. **Status = DEAD**
   stato terminale monotono — propaga sempre

Garanzia: due nodi che si sincronizzano raggiungeranno sempre lo stesso stato (eventual consistency).

Gossip Dissemination Protocol

Push e push-pull epidemic dissemination — scelta AP/BASE con eventual consistency

Gossip round — push mode

1. incrementa heartbeat locale
2. seleziona k peer casuali (fanout)
3. serializza Membership Table → JSON
4. invia via UDP ai peer selezionati
5. peer ricevente esegue merge (policy)

Push vs Push-Pull

Push

nodo invia la propria tabella— semplice, basso overhead

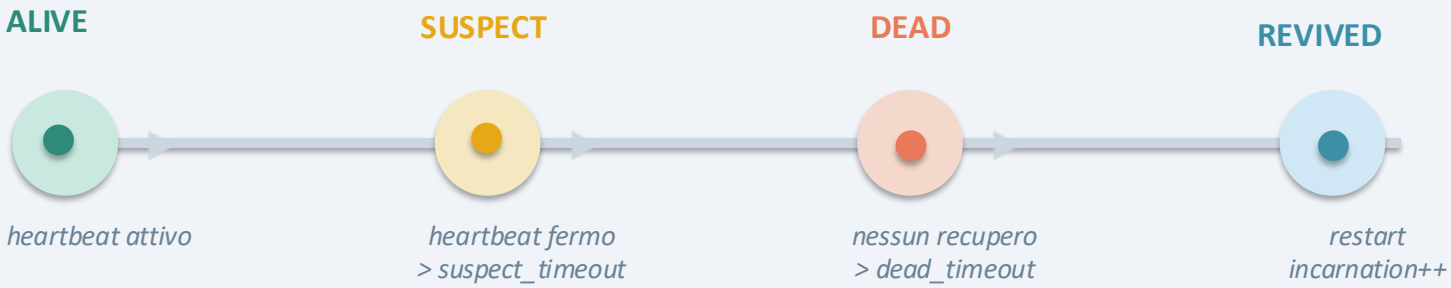
Push-Pull

il peer risponde con la propria vista — anti-entropy bidirezionale, convergenza più rapida

AP | BASE | Eventual consistency | O(k) msg/round

Failure Detection

Heartbeat, timeout e transizioni di stato ispirate a SWIM



Meccanismo

- heartbeat incrementato ogni gossip round
- peers tracciano updated_at di ogni entry
- suspect_timeout: 5.0 s (default)
- dead_timeout: 10.0 s (default)
- incarnation++ al riavvio refuta SUSPECT

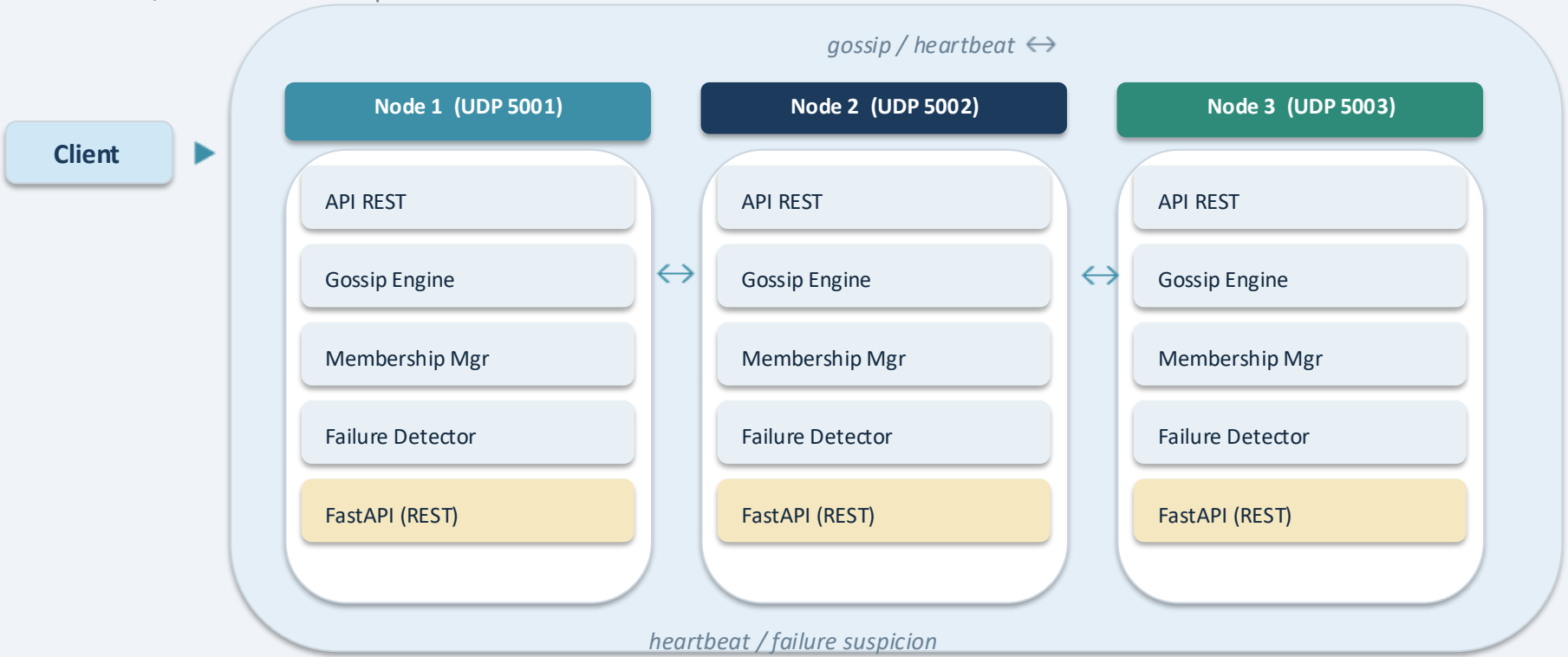
Dati sperimentali (crash node-3)

Observer	SUSPECT	DEAD
node-1	6.16 s	12.18 s
node-2	6.08 s	12.10 s

Latenze vicine ai timeout configurati — failure detection deterministica

Architettura run-time

Nodi attivi, sincronizzazione periodica e osservabilità



Scelte progettuali e alternative scartate

Perché gossip + SWIM invece di coordinazione forte

Opzione	Vantaggi	Perché non scelta / esito
Consenso forte (SMR / Raft)	ordine globale, consistenza forte	latenza alta, disponibilità ridotta sotto partizione, complessità inutile per monitoring
CRDT + replica asincrona	merge naturale, alta disponibilità	*scelta adottata: gossip + merge policy equivalente a CRDT semantics*
Logical / vector clocks	tracciano causalità e concorrenza	non necessari: heartbeat counter + incarnation rendono l'ordine non critico
Code mobility	sposta computazione sui nodi	non usata: il progetto muove stato e membership, non codice

Gossip Arena — Validazione sperimentale

3 nodi, eseguiti come processi Python locali via loopback UDP

Gossip Arena

- inietta rumor sintetici in un nodo qualsiasi
- misura copertura: % nodi raggiunti per round
- traccia velocità di propagazione e hop count
- testa robustezza con packet loss configurabile

```
curl -X POST
http://localhost:8001/arena/inject
  -d '{"rumor_id": "r42"}'
```

Esperimenti condotti:

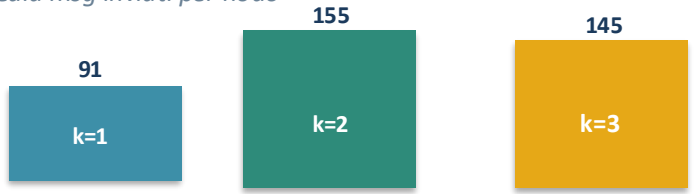
- Convergenza baseline (rete affidabile)
- Failure detection: crash di node-3
- Packet loss 30% (NET_DROP_RATE=0.3)
- Fanout tradeoff: k ∈ {1, 2, 3}

Risultati — Tabella overhead (msg inviati)

Fanout	Node-1 Sent	Node-2 Sent	Node-3 Sent
k = 1	99	89	84
k = 2	158	158	148
k = 3	148	148	138

Nota: con 3 nodi, k=3 ≈ k=2 (bounded by available peers)

Media msg inviati per nodo



Risultati e conclusioni

Obiettivi raggiunti, tradeoff chiave e sviluppi futuri

Risultati principali

- Eventual consistency — tutti i nodi convergono
- Failure detection corretta, no false positives
- Robustezza con 30% packet loss UDP
- O(k) messaggi per nodo per round
- fanout k → convergenza più rapida
- fanout k → maggiore traffico di rete
- push-pull migliora anti-entropy rispetto a push

Sviluppi futuri

- SWIM indirect probing (ping-req)
- adaptive failure detection (ϕ-accrual detector)
- advanced peer sampling (HyParView)
- large-scale deployment (50-200+ nodi)
- dynamic peer discovery (no static seeds)
- Kubernetes-native deployment
- CRDT-based replicated state

Il progetto dimostra come epidemic dissemination e coordinazione decentralizzata possano offrire monitoring scalabile e fault-tolerant in ambienti asincroni e inaffidabili — senza alcun coordinatore centrale.