# Gossiping per System Utilization e Job Sharing in un sistema P2P

SDCC 2024/2025

Alessandro Lori
Macroarea di Ingegneria Informatica
Università degli Studi di Roma Tor Vergata
Roma, Italia
alessandrolori179@gmail.com

## ABSTRACT

Questo documento presenta l'architettura, le soluzioni adottate e l'implementazione di un sistema P2P decentralizzato che diffonde informazioni tra i nodi attraverso Gossiping per System Utilization e Job Sharing. Il sistema è stato sviluppato in Golang e va a trattare i guasti dei nodi della rete e le leave volontarie cercando di mantenere la rete il più resiliente possibile.

## INTRODUZIONE

I sistemi distribuiti sono fondamentali ad oggi per lo sviluppo digitale su larga scala portando ad un dislocamento geografico di server che comporta sfide per tenere tutti i dispositivi aggiornati e responsivi alle sollecitazioni. Prendendo come esempio una render farm distribuita è cruciale la coordinazione tra i server che ne fanno parte. Tale coordinazione implica la costante interazione tra questi server per sapere condizioni di carico, disponibilità e problematiche. Ogni nodo di un sistema distribuito, per ottenere o diffondere informazioni, non può permettersi di inondare la rete con messaggi broadcast dato che andrebbe a saturare la banda di rete rendendo totalmente inefficiente il sistema. In questa trattazione si presenta una soluzione che sfrutta un protocollo di comunicazione distribuita che permette ai nodi di scambiare informazioni, simili a quelle scambiate nelle render farm, correttamente senza sovraccaricare la rete mantenendo contemporaneamente una resilienza ai guasti che potrebbero risulta in Single Point Of Failure (SPOF) se non trattati adeguatamente. Il protocollo adottato in questa trattazione è il Gossiping, anche detto protocollo Epidemico, attraverso comunicazioni tra i nodi effettuate tramite framework gRPC che ci consente comunicazioni bidirezionali e basse latenze di rete.

### I. Protocollo Gossiping

È un protocollo per la comunicazione distribuita in cui ogni nodo periodicamente scambia delle conoscenze con un sottogruppo di nodi tra quelli conosciuti dal mittente. Un nodo invia delle informazioni ad un gruppo casuale di nodi che conosce e i nodi riceventi l'informazione la propagheranno con stesso criterio a loro volta. Questa soluzione non porta ad una propagazione immediata dell'informazione ma allo stesso tempo permette semplicità di gestione, nessun controllo centralizzato, basso utilizzo di banda di rete e resilienza ai guasti data dalla ridondanza dei messaggi inviati da nodo a nodo.

Nell'architettura presentata è stato adottato un Gossiping di tipo Anti-Entropy, ovvero una tipologia di comunicazione che mira alla consistenza di informazioni completa per i nodi del sistema. Il protocollo Anti-Entropy offre più modalità per lo scambio di informazioni, la modalità adottata per l'architettura sviluppata è di tipo PUSH-PULL: un nodo oltre ad inviare le proprie informazioni richiede anche quelle in possesso del ricevente, in questo modo i nodi ottengono consistenza più velocemente a fronte di un impatto sulla rete minimo, con un costo di circa $O(log_2 N)$ round per far arrivare l'informazione ad N nodi.

## ARCHITETTURA

L'architettura implementata cerca di rimanere il più possibile affine ad eventi e casistiche che potrebbero accadere realmente in un sistema distribuito come quello di una render farm, per poter realizzare tale scopo è stata simulata l'esecuzione di una rete P2P nella quale ogni nodo:

- Scambia con gli altri nodi informazioni di CPU, memoria e GPU propri e degli altri nodi conosciuti, aggiornando così la Membership List, ovvero la lista dei nodi conosciuti con i relativi dettagli.
- Genera dei job caratterizzati da un carico CPU, MEM, GPU che invia ad un nodo libero.
- È soggetto ad eventi di Fault.
- È soggetto ad eventi di Leave volontarie.
- Effettua delle ping randomiche per controllare lo stato di vita degli altri nodi.
- Possiede una lista di Friends&Affinity.

Questa scelta permette di rappresentare tutti i principali avvenimenti in un sistema distribuito dall'assegnazione dei job che vanno a sovraccaricare i nodi che li ricevono all'uscita di un nodo volontaria tramite Leave o involontaria tramite Fault.

Per poter rendere il sistema più responsivo, i nodi più consistenti e la rete meno carica sono stati usati più tipologie di messaggi in funzione della comunicazione che i nodi devono effettuare:

- ExchangeAvail, di tipo Anti-Entropy dove il nodo scambia in modalità push-pull le sue informazioni e quelle degli altri nodi che conosce con un altro nodo.

- SWIM (Scalable Weakly-consistent Infection-style Membership), messaggi di ping per monitorare lo stato di vita di un nodo.
- Probe, Cancel e Commit (Scheduling), con i quali il nodo sonda alcuni dei suoi vicini per poi decidere a chi affidare il job, annullare il job da affidare, confermare la corretta esecuzione del job.
- Piggyback, il nodo sfrutta i metadati di un messaggio gRPC per allegare informazioni rapide.

Lo stato di vita di un nodo è descritto uno dei 3 stati possibili:

- ALIVE: il nodo è vivo e risponde correttamente agli stimoli.
- SUSPECT: il nodo non ha risposto ad uno stimolo.
- DEAD: il nodo non ha risposto a stimoli di accertamento sul suo stato.

La rete P2P è descritta da nodi Seed e nodi Peer:

- Seed: nodi responsabili del punto di accesso alla rete P2P, forniscono al nodo entrante una lista di nodi presenti nel sistema.
- Peer: nodi partecipanti alla rete, tramite i nodi Seed ottengono una prima lista di nodi della rete da poter contattare.

Per il deployment del sistema è stato impiegato l'uso di Docker e Docker Compose, dove ogni nodo è rappresentato da un container, permettendo di eseguire ogni nodo in un ambiente isolato e di emulare la corretta interazione di macchine indipendenti come quelle di una render farm distribuita.

## DETTAGLI DELL'ARCHITETTURA

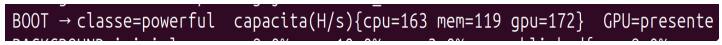
L'architettura prevede un massimo di 20 nodi di cui 3 Seed e 17 Peer, questi ultimi devono connettersi almeno ad un singolo Seed per poter entrare nel sistema. I Seed oltre a rappresentare un punto di entrata nella rete sono esattamente come i Peer, hanno stesse funzionalità e sono soggetti agli stessi eventi come Fault o Leave volontarie.

L'architettura è stata sviluppata affinché potesse affrontare un'alta varianza di situazioni andando a ricreare un contesto il più realistico possibile, nel quale gli eventi sono descritti da alcune probabilità intrinseche di ogni nodo. Grazie al sistema di probabilità implementate l'architettura non richiede comandi manuali per provocare eventi bensì tutto è automatizzato all'avvio, cercando, come già menzionato, di riprodurre il comportamento di un sistema distribuito di tipo job sharing.

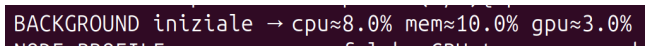
### I. Generazione dei Nodi

Ogni nodo quando viene generato acquisisce delle caratteristiche personali che ne descrivono il comportamento lungo tutta la durata della simulazione. Queste caratteristiche sono descritte da bucket di probabilità che permettono varianza sui nodi:

- Ogni nodo ha una sua potenza computazionale con la quale contribuire al sistema e che descrive la potenza di CPU, Memoria e GPU eventuale. Un nodo può essere:
  - Debole con probabilità 20%
  - Medio con probabilità 50%
  - Forte con probabilità 30%

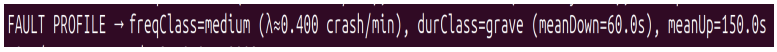



- Ogni nodo nasce con un carico di base rappresentante operazioni non afferenti al sistema o carichi dovuti al sistema operativo. I carichi sono in funzione della potenza del nodo: meno il nodo è potente, più percentuale di componente sarà utilizzata per il carico di base.




- Ogni nodo può andare in Fault e quindi lasciare il sistema senza risposta da un momento all'altro senza avvisare nessuno, simulando un crash reale. Viene utilizzata una probabilità derivata da un bucket per descrivere la frequenza con la quale i Fault possono avvenire:
  - Alta: il 15% dei nodi subirà 1.2 Fault/min in media.
  - Media: il 35% dei nodi subirà 0.4 Fault/min in media.
  - Bassa: il 40% dei nodi subirà 0.1 Fault/min in media.
  - Nessuna: il 10% dei nodi non subirà Fault.

  Viene utilizzata una seconda probabilità derivata da un secondo bucket per descrivere la gravità e la durata del Fault prima che il nodo effettui la riconnessione.
  - Grave: il 15% dei nodi ha un Fault della durata di 60 secondi in media.
  - Medio: il 50% dei nodi ha un Fault della durata di 20 secondi in media.
  - Piccolo: il 35% dei nodi ha un Fault della durata di 5 secondi in media.




- Ogni nodo può effettuare una Leave volontaria comunicando la sua uscita spontaneamente per poi interrompere le attività nel sistema. Una ulteriore probabilità derivata da un bucket descrive la frequenza con la quale il nodo effettua una Leave volontaria:
  - Alta: il 10% dei nodi effettua 0.5 Leave/min in media.
  - Media: il 20% dei nodi effettua 0.15 Leave/min in media.
  - Bassa: il 40% dei nodi effettua 0.05 Leave/min in media.
  - Nessuna: il 30% dei nodi non effettua Leave.

  Utilizzando una seconda probabilità derivata da un secondo bucket descrive la durata della Leave volontaria prima che il nodo effettui la riconnessione.
  - Lunga: il 15% dei nodi effettua una Leave della durata di 45 secondi in media.

- Media: il 35% dei nodi effettua una Leave della durata di 15 secondi in media.
- Corta: il 50% dei nodi effettua una Leave della durata di 5 secondi in media.

```
LEAVE PROFILE → freqClass=low (λ≈0.050 leave/min), durClass=short (meanAway=5.0s), meanUp=1200.0s
```

- Ogni nodo può generare dei job e inviarli agli altri nodi disponibili. Utilizzando una probabilità derivata da un bucket si descrive la frequenza con la quale il nodo genera un job:
  - Alta: il 25% dei nodi genera 16 job/min in media.
  - Media: il 50% dei nodi genera 12 job/min in media.
  - Bassa: il 25% dei nodi genera 8 job/min in media.

  Con una ulteriore probabilità derivata da un secondo bucket si descrive la durata del job generato:
  - Lungo: il 10% dei job generati ha una durata di 20 secondi in media.
  - Medio: il 30% dei job generati ha una durata di 10 secondi in media.
  - Corto: il 60% dei job generati ha una durata di 5 secondi in media.

```
JOB GENERATION: job_type=MEM_HEAVY  mix=General:0.10, CPU_int:0.10, MEM_int:0.70, GPU_int:0.10
```

```
rate=normal*x1.00  mean_interarrival=5.0s
```

**II. Tipi di Job**

Ogni nodo genera job di dimensioni diverse che scambia con gli altri nodi andando ad aumentare l'utilizzazione delle loro componenti. Ogni job generato ha una richiesta di risorse diversa strettamente legata alla natura del nodo che lo genera, utilizzando la stessa probabilità i nodi possono essere propensi a generare job di tipo CPU_HEAVY, MEM_HEAVY, GPU_HEAVY o GENERAL. Trattandosi di una tendenza, non è detto che il nodo generi job solo di quel tipo; infatti, tale comportamento ha il 70% di probabilità di essere rispettata, mentre il restante 30% è equidistribuito sulle rimanenti 3 nature del job. Più precisamente, ogni natura del job ha un bucket di altre probabilità che fanno variare il carico delle altre componenti ed eventualmente creano job ibridi: se un nodo genera un job di tipo CPU_HEAVY allora il carico sulla CPU sarà del 70% circa, mentre sulle altre componenti entrano in azione altre probabilità:

- Per il 60% di probabilità la componente avrà un carico del 4% circa.
- Per il 30% di probabilità la componente avrà un carico del 12% circa.
- Per il 10% di probabilità la componente avrà un carico del 52% circa.

Per i job di tipo GENERAL, invece, abbiamo che il carico di ogni componente può variare fino al 50% massimo, cercando di emulare job equilibrati e più uniformi sulla richiesta.

Il meccanismo di probabilità utilizzato viene ulteriormente variato da un jitter massimo del 20%. Questa configurazione permette di rappresentare un'elevata varianza di richieste e quindi di simulare contesti di job sharing generali, non solo di render farm.

**III. Probe, Cancel, Commit - Scheduling dei Job**

Una volta generato un job, il nodo deve occuparsi di trovare un suo simile al quale inviarlo. Affinché la rete non venga rallentata da richieste broadcast provenienti da ogni nodo per inviare job, è stato adottato un meccanismo di invio dei job a fasi ispirato dal paper *"Issues of Implementing Random Walk and Gossip Based Resource Discovery Protocols in P2P MANETs & Suggestions for Improvement"*. Per poter assegnare il job, il nodo sonda tramite PROBE al più 3 nodi che conosce mandando una richiesta di disponibilità e un'anteprima del peso del job affinché i nodi che ricevono tale richiesta possano valutare le proprie disponibilità di sistema e rispondere adeguatamente.

```
2025/10/07 14:18:30 [01:06:53.787] [node-cbd7ae756c4d] [INFO] PROBE ORDER (3):
  #1 node-36c3f847ebbb load=85.9% stale busy=0s addr=127.0.0.1:9006 [cpu=14% mem=86% gpu=29%]
  #2 node-5366b44faf55 load=10.2% stale busy=0s addr=127.0.0.1:9004 [cpu=8% mem=10% gpu=-]
  #3 node-3187b9cd9846 load=16.1% stale busy=0s addr=127.0.0.1:9005 [cpu=14% mem=16% gpu=-]
```

Quando il nodo ricevente accetta, si riserva uno slot per attendere l'invio del job vero e proprio affinché eventuali altre richieste possano essere rifiutate e non creare sovraccarico su se stesso. Quando il nodo mittente riceve un'accettazione del job manda una CANCEL agli altri nodi contattati e invia il job al nodo che ha accettato.

```
COORD COMMIT ✖ target=node-36c3f847ebbb job=job-9b721b54
```

Il nodo che riceve il job attraverso COMMIT inizia la computazione e, una volta terminata, rende il risultato al mittente.

```
[01:08:40.388] [node-cbd7ae756c4d] [INFO] COMMIT ← job=job-777c1012 cpu=5.3% mem=16.2% gpu=15.6% dur=5.097s
[01:08:40.389] [node-cbd7ae756c4d] [INFO] JOB START → job=job-777c1012 cpu=5.3% mem=16.2% gpu=15.6% dur=5.097s
[01:08:40.390] [node-cbd7ae756c4d] [INFO] COMMIT ✓ job=job-777c1012
```

```
[01:08:45.524] [node-cbd7ae756c4d] [INFO] JOB END    → job=job-777c1012
```

Il nodo mittente ricevuto il risultato confermerà la ricezione.

```
COORD COMMIT ✓ target=seed@127.0.0.1:9002 job=job-777c1012 cpu=5.3% mem=16.2% gpu=15.6% dur=5.097546922s
```

Questa struttura permette al sistema di rimanere leggero a livello di rete migliorando le prestazioni generali del sistema in modo da permettere ai nodi di effettuare pochi messaggi ma diretti.

**IV. Friends&Affinity**

Al fine di alleggerire ulteriormente la rete e di migliorarne le prestazioni dell'architettura, per ogni nodo è stata aggiunta un sistema di Friends&Pheromoni, che da adesso in poi chiameremo Friends&Affinity, ispirato dal paper *"Improving Resource Discovery and Query Routing in Peer-to-Peer Data Sharing Systems Using Gossip Style and ACO Algorithm"*. Questa aggiunta permette ai nodi di riconoscere chi è più adatto ad un determinato job rendendolo suo amico in modo da preferire quel nodo al momento di consegna dei job. Questo comportamento potrebbe creare però un effetto isolazionista nei confronti dei nodi

rimanenti della rete rischiando di far convergere più nodi a richiedere disponibilità di un singolo molto potente. Per scongiurare questo effetto isolazionista ed effetto gregge, ogni nodo ha una tabella di affinità e reputazione dove ogni nodo ha un punteggio.

Il valore punteggio non rimane stabile nel tempo ci sono vari fattori che portano all'incremento o decremento dello stesso:

- Decadimento naturale nel tempo che forza il punteggio a convergere a zero

```
[Affinity/Rep] decay class=0 peer=node-7d5fdee557bd  -0.016 -> -0.015 (factor=0.919630)
[Affinity/Rep] decay class=0 peer=node-db588bde2d68  -0.017 -> -0.016 (factor=0.919630)
[Affinity/Rep] decay class=2 peer=node-7d5fdee557bd  -0.258 -> -0.237 (factor=0.919630)
[Affinity/Rep] decay class=2 peer=node-db588bde2d68  -0.271 -> -0.249 (factor=0.919630)
[Affinity/Rep] decay class=1 peer=node-db588bde2d68  -0.000 -> -0.000 (factor=0.919630)
[Affinity/Rep] decay class=1 peer=node-7d5fdee557bd  0.022 -> 0.020 (factor=0.919630)
[Affinity/Rep] decay class=3 peer=node-7d5fdee557bd  0.240 -> 0.221 (factor=0.919630)
[Affinity/Rep] decay class=3 peer=node-db588bde2d68  -0.060 -> -0.055 (factor=0.919630)
```

- Decremento dovuto al rifiuto di un job

```
2025/10/07 15:35:21 [01:10:13.780] [node-a0d6c97dbae5] [INFO] COORD COOL-OFF ← set after RPC-ERROR addr=127.0.0.1:9005 cooloff=1.1s
2025/10/07 15:35:21 [01:10:13.781] [node-a0d6c97dbae5] [INFO] PROBE → node-7d5fdee557bd refuse (reason=dial_error) job=job-362859ed
[Affinity/Rep] class=1 peer=node-7d5fdee557bd delta=-0.50 reason=probe_refuse  |  raw: 0.096 -> -0.404 (norm=0.386)
```

- Incremento dovuto all'accettazione di un job

```
[Affinity/Rep] class=3 peer=node-7d5fdee557bd delta=+0.50 reason=probe_accept  |  raw: 0.241 -> 0.741 (norm=0.383)
```

- Incremento dovuto all'esecuzione corretta di un job

```
[Affinity/Rep] class=1 peer=node-7d5fdee557bd delta=+2.00 reason=commit_ok  |  raw: 4.513 -> 6.513 (norm=0.768)
```

Per ogni classe di job i nodi salvano una lista di amicizie con i relativi punteggi riconoscendo chi è più affine per quella classe di job, per un massimo di 4 nodi amici per classe.

```
Friends (max=4) & Reputation (60s HL)
- Class=3 (GENERAL)
  Friends: [node-7d5fdee557bd node-db588bde2d68]
  RepTop: [node-7d5fdee557bd: raw=1.66 norm=0.44, node-db588bde2d68: raw=-0.41 norm=0.31]
- Class=0 (CPU_ONLY)
  Friends: [node-db588bde2d68 node-7d5fdee557bd]
  RepTop: [node-7d5fdee557bd: raw=-0.11 norm=0.33, node-db588bde2d68: raw=-0.12 norm=0.33]
- Class=2 (MEM_HEAVY)
  Friends: [node-db588bde2d68 node-7d5fdee557bd]
  RepTop: [node-7d5fdee557bd: raw=-0.03 norm=0.33, node-db588bde2d68: raw=-0.97 norm=0.27]
- Class=1 (GPU_HEAVY)
  Friends: [node-7d5fdee557bd]
  RepTop: [node-7d5fdee557bd: raw=5.41 norm=0.69, node-db588bde2d68: raw=-0.00 norm=0.33]
```

Per poter ulteriormente portare varianza tra i nodi e far si che non convergono ad un singolo nodo potente vi è un meccanismo di COOLOFF: quando un nodo rifiuta un job, o lo committa, viene applicato un COOLOFF per il quale il nodo mittente non potrà contattare nuovamente il nodo in questione preferendo quindi altri nodi per eventuali altri job.

### V. SWIM - Ping di Stato

Per poter monitorare lo stato di vita di ogni nodo nel sistema non c'è un organismo centrale, quindi i nodi devono organizzarsi per poter scoprire chi ha avuto un guasto e non risponde alle varie sollecitazioni. Anche in questo caso i nodi non possono mandare messaggi broadcast sperando di ricevere risposta da tutti i nodi, la rete andrebbe ad appesantirsi e di conseguenza le prestazioni del sistema calerebbero. I nodi effettuano un PING diretto ad al più 2 nodi che devono rispondere entro una finestra di tempo per essere considerati ancora ALIVE.

```
[node-c7e1e42f3bc2] [INFO] SWIM PING → to=127.0.0.1:9004
[node-c7e1e42f3bc2] [INFO] SWIM PING ← to=127.0.0.1:9004 ok=true
[node-c7e1e42f3bc2] [INFO] SWIM PING → to=127.0.0.1:9006
[node-c7e1e42f3bc2] [INFO] SWIM PING ← to=127.0.0.1:9006 ok=true
```

Questi messaggi di PING derivano da un approccio stile SWIM, diretta su pochi nodi ma precisi al fine di scoprire informazioni sia in caso di risposta ricevuta che mancata.

Quando un nodo che invia una PING non riceve risposta chiede ad al più 2 nodi vicini vivi che conosce di effettuare una PINGREQ (ping-request), se il richiedente della PINGREQ riceve esito positivo perde ogni dubbio nei confronti del nodo iniziale che non ha risposto.

```
[WARN] SWIM PING → to=127.0.0.1:9002 dial fail
[INFO] SWIM PINGREQ → helper=127.0.0.1:9005 target=127.0.0.1:9002
[INFO] SWIM PINGREQ ← helper=127.0.0.1:9005 ok=true
```

Se invece il nodo richiedente riceve un esito negativo dalla PINGREQ marca il nodo come SUSPECT.

```
SWIM PING → to=127.0.0.1:9006 dial fail
SWIM PINGREQ → helper=127.0.0.1:9004 target=127.0.0.1:9006
```

```
SWIM PINGREQ ← helper=127.0.0.1:9004 ok=false (rpc error: code = DeadlineExceeded desc = context deadline exceeded)
SWIM PINGREQ → helper=127.0.0.1:9005 target=127.0.0.1:9006
```

```
SWIM PINGREQ ← helper=127.0.0.1:9005 ok=false (rpc error: code = Unavailable desc = error reading from server: EOF)
SWIM ALIVE → SUSPECT  peer=node-67d048426c61 (127.0.0.1:9006)  reason=no-ack direct+indirect
```

Dopo un lasso di tempo ulteriore di circa 6-7 secondi il nodo passa dallo stato SUSPECT allo stato DEAD e viene inoltre comunicata la sua morte nella rete.

```
SWIM still SUSPECT  peer=node-67d048426c61 (127.0.0.1:9006) reason=no-ack direct+indirect
SWIM SUSPECT → DEAD  peer=node-67d048426c61 (127.0.0.1:9006)  reason=timeout
SWIM UPDATE SEND → 1 death(s) to=127.0.0.1:9005  ids=node-67d048426c61
```

Quando un nodo marcato come SUSPECT o DEAD riprende le sue attività viene riportato come ALIVE.

### V. Fault

Uno degli eventi che porta un nodo ad essere SUSPECT o DEAD è una Fault. Durante la Fault il nodo si spegne senza preavviso andando in CRASH lasciando qualsiasi tipo di job o comunicazione in sospeso, può capitare che alcuni log vengano stampati dopo i log di Fault, questo è dovuto ad altre go-routine in background che sono andate in stampa dopo ma che in realtà sono state concluse prima.

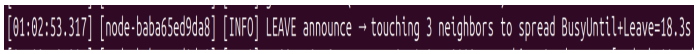
```
[01:05:14.956] [node-5e13eb9ff666] [WARN] FAULTS → DOWN
[01:05:14.957] [node-5e13eb9ff666] [WARN] FAULT ↓ CRASH – stop SWIM + AE + Reporter + gRPC
[01:05:14.958] [node-5e13eb9ff666] [WARN] Reporter stopped
[01:05:14.958] [node-5e13eb9ff666] [WARN] Anti-Entropy engine stopped
[01:05:14.969] [node-5e13eb9ff666] [WARN] gRPC fermato (server e listener chiusi)
[01:05:15.519] [node-5e13eb9ff666] [INFO] FAULTS → UP
[01:05:15.521] [node-5e13eb9ff666] [WARN] FAULT ↑ RECOVERY – restart SWIM + AE + Reporter + gRPC
[01:05:15.524] [node-5e13eb9ff666] [INFO] Peer non-seed su 127.0.0.1:9004 (Ping/PingReq/ExchangeAvail/Probe/Commit/Cancel)
```

Una volta che il nodo si riprende dal Fault con una RECOVERY effettua la riconnessione, riprendendo le sue normali attività che lo renderanno di nuovo visibile e vivo alle connessioni degli altri nodi.

Per simulare correttamente il Fault vengono fermati tutti gli engine e i server gRPC.

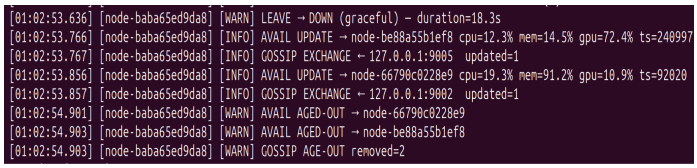
### VI. Leave

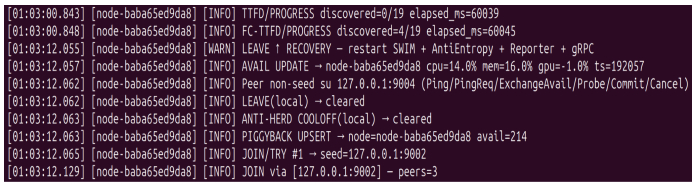
Quando un nodo effettua una disconnessione dalla rete tramite una Leave effettua una disconnessione volontaria, non dovuta a crash o malfunzionamenti. Durante una Leave, prima di lasciare definitivamente la rete, il nodo comunica al più a 3 nodi che conosce la sua disconnessione e successivamente va in DOWN.









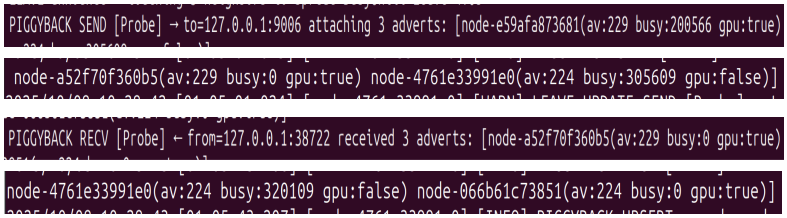

Successivamente alla DOWN, terminato il periodo di disconnessione, il nodo si riconnetterà alla rete riprendendo le normali attività previste. Anche nel caso della Leave, eventuali stampe nel log dopo la disconnessione del nodo dalla rete sono dovute a delle go-routine che sono andate in stampa dopo il DOWN ma che si sono concluse prima.

Per simulare correttamente la Leave vengono fermati tutti gli engine e i server gRPC.

### VII. Piggyback

gRPC mette a disposizione dei metadata nella parte header e trailer del messaggio HTTP/2 dei messaggi sfruttabili per allegare piccoli payload con informazioni minimali per un massimo consigliato di 1-2KB. Il messaggio vero e proprio non subisce alterazioni, le informazioni allegate ai metadata sono ulteriori ed eventuali. La tecnica di allegare piccole informazioni nei metadata dei messaggi è detta Piggyback, usata soprattutto nell'ambito delle telecomunicazioni. L'implementazione di tale tecnica all'interno dell'architettura è ispirata dal paper *"Using Gossip for Dynamic Resource Discovery"*. All'interno di questi metadata i nodi riescono ad allegare informazioni di utilizzazione del sistema, un flag che specifica la presenza della GPU e un campo busy che specifica per quanto tempo il nodo sarà occupato con un job in millisecondi promuovendo così un comportamento anti-gregge, ogni campo appartiene ad un nodo appartenente alla Membership List del mittente per un massimo di 3 nodi conosciuti. L'utilizzazione del sistema condivisa all'interno dei metadata è una quantizzazione ad un singolo valore tra tutte le componenti del nodo stimata per eccesso su un valore di 255 byte unit8 (senza segno) che indica quanto, in generale, il nodo è libero. Un carico di sistema dello 0% sarà indicato da un Avail pari a 255, se avessimo un carico del 50% avremmo 128 e così via.

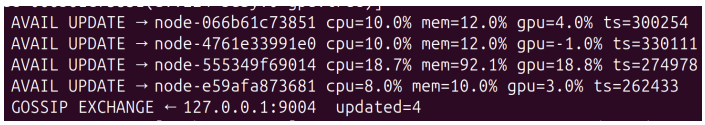
Questi allegati Piggyback sono presenti in tutti i messaggi del sistema, dai Probe per sondare e dare dei job, alle PING e PINGREQ per capire lo stato di vita degli altri nodi, fino alla LEAVE. Il caso della LEAVE si ha che che quando un nodo annuncia la sua disconnessione invia agli altri nodi un messaggio il cui contenuto risiede prevalentemente nella porzione relativi ai metadata; in questo modo riesce ad annunciare agli altri nodi la sua uscita dalla rete e allo stesso tempo la propria conoscenza della rete, rendendo il peso del messaggio irrisorio.




### VIII. ExchangeAvail - Gossip Esplicito

Tutte le tipologie di comunicazione viste fino ad ora sono Gossip Esplicito solo per il particolare messaggio che vogliono inviare, ma non lo sono per quanto riguarda lo scambio di conoscenze sull'utilizzazione degli altri nodi facenti parte della loro Membership List, bensì effettuano un Gossip Implicito dato dagli allegati di Piggyback. Per effettuare Gossip Esplicito proprio sull'utilizzazione dei nodi, questi, effettuano dei messaggi di tipo GOSSIP EXCHANGE, attuando a pieno il metodo PUSH-PULL, inviando e ricevendo informazioni di massimo altri 8 nodi appartenenti alla propria lista.

Quando il nodo riceve un'informazione, inerente ad un determinato nodo menzionato nel messaggio, la confronta con quella che possiede: se ha un timestamp più recente esegue una AVAIL UPDATE dove aggiorna le informazioni che conosce, altrimenti la ignora.




## SCOPERTA DELLA RETE

Come menzionato in precedenza, quando un nuovo nodo entra nella rete aggiunge alla propria Membership List un sottogruppo di nodi già presenti dal peer al quale si connette, per un massimo di 8 nodi comunicati, i restanti nodi della rete dovranno essere scoperti. Il nodo viene a sapere dell'esistenza dei restanti nodi

della rete attraverso gli scambi che avvengono con i pochi nodi iniziali dati dal seed, stessa cosa vale per tutti i nodi che non conoscono il nuovo nodo. Ci sono due modi con i quali possiamo identificare la scoperta di un nuovo nodo:

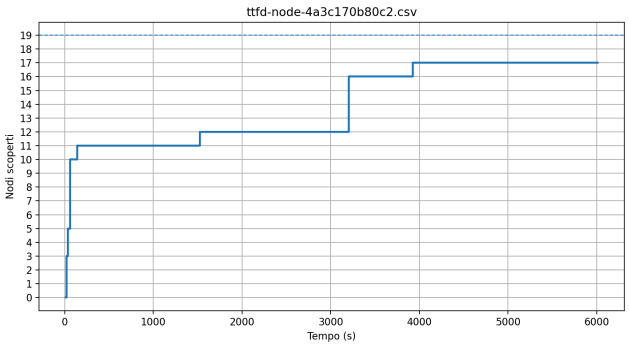
- TTFD (Time To First Data), quando il nodo ha scoperto l'esistenza di un altro nodo ottenendo anche l'utilizzazione delle sue componenti CPU, MEM, GPU.
- FC-TTFD (First Contact TTFD), quando il nodo ha scoperto l'esistenza di un altro nodo ma senza ottenere informazioni inerenti all'utilizzazione delle sue componenti.

Questi identificativi mostrano da una parte la conoscenza parziale di un nodo nel sistema con il quale sarà difficile interagire a livello di job non conoscendo le sue disponibilità, dall'altra una conoscenza completa che permette la piena aggiunta alla propria Membership List.

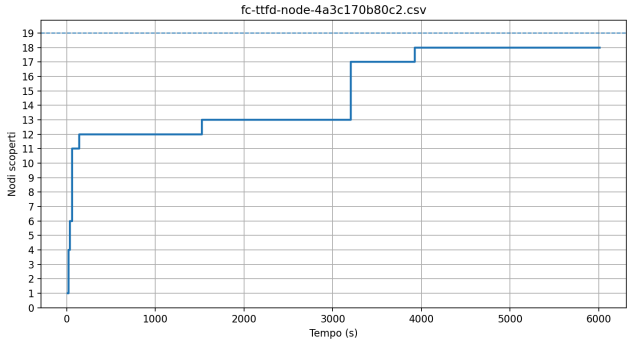
```
FC-TTFD/PROGRESS discovered=7/19 elapsed_ms=360021
TTFD/PROGRESS discovered=5/19 elapsed_ms=360020
```

Come menzionato precedentemente, l'architettura sviluppata supporta fino a 20 nodi di cui 3 Seed. Nei grafici sottostanti possiamo vedere l'evoluzione della scoperta dell'intero sistema per il 20esimo nodo, la configurazione adottata è la connessione del 20esimo nodo ad un singolo Seed con i restanti 19 nodi già nel sistema ed in attività.

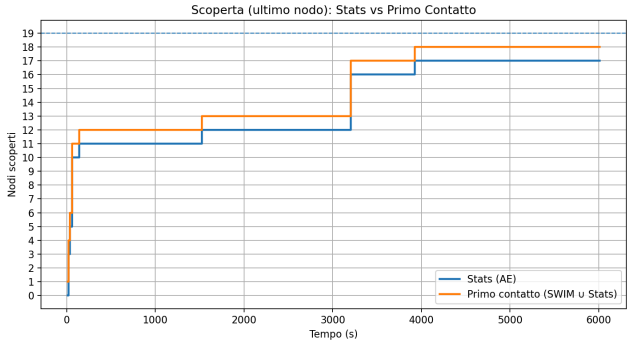
Il primo grafico mostra come attraverso la tipologia di contatto TTFD il 20esimo nodo ottenga a pochi secondi dalla sua connessione nella rete più della metà dei nodi già presenti, per poi nel tempo arrivare a 17 nodi su 19 presenti nella sua Membership List:




Il secondo grafico mostra come attraverso la tipologia di contatto FC-TTFD il 20esimo nodo arrivi a conoscere un nodo in più rispetto alla tipologia precedente, ma senza conoscere le sue caratteristiche di utilizzazione nel dettaglio.




Nel terzo grafico possiamo vedere la differenza di scoperta tra le due tipologie già menzionate, in arancione la tipologia FC-TTFD e in blu la tipologia TTFD:




Come visibile su tutti i grafici, il nodo non arriva a conoscere tutti i 19 nodi della rete. Questo comportamento può essere dovuto ad un'eventuale alta probabilità di Fault e/o Leave del nodo mancante che pur essendo connesso alla rete durante la simulazione continua a ricevere crash improvvisi o ad effettuare delle disconnessioni volontarie.

Il fatto che il nodo arrivi a coprire quasi tutti i nodi è indice di un'implementazione anti-gregge e anti-isolazionista prevalentemente corretta.

Il nodo arriva a conoscere quasi tutti i suoi simili nella rete prima dei 4000 secondi di attività circa, questo può essere dovuto anche alla connessione al singolo Seed effettuata per entrare nella rete, se avesse comunicato con più Seed è molto probabile che il tempo per arrivare allo stesso numero di nodi scoperti sarebbe stato meno.

## OSSERVAZIONI E MIGLIORAMENTI

L'architettura implementata richiede Seed fissi che rappresentano quindi degli SPOF nel momento in cui sono in crash e nuovi nodi cercano di entrare nella rete. Per risolvere questo inconveniente si potrebbe implementare un sistema di elezione tra i nodi per far si che un Peer diventi Seed sostituendo quello andato in crash, in modo da garantire una continuazione alle join del sistema.

Il sistema potrebbe essere reso autoadattativo in funzione del comportamento medio osservabile attraverso l'uso di intelligenza

artificiale, ad esempio adattando il numero massimo di informazioni inviabili nei messaggi in funzione del carico di rete medio osservato.

Alcuni nodi potrebbero avere una tendenza maggiore a generare job piuttosto che ad eseguirli, per gestire tale evenienza si potrebbe pensare ad un sistema a punti per il quale più job completi più job puoi generare, scagionando così un'eventuale discrepanza o la presenza di nodi malevoli.

Attualmente tra i nodi avviene uno scambio di job interi, per migliorare l'esecuzione del singolo job si potrebbe frammentare la sua esecuzione tra più nodi, come per il rendering di un'animazione si fa con i frame, affinché più nodi possano prendere in esecuzione più frammenti di job contemporaneamente. Inoltre, la rappresentazione ibrida dei job potrebbe dare vita a tipologie di carico rilevanti in ambito di ricerca odierni, un job con alta richiesta di MEM e di GPU potrebbe rappresentare un carico simile a quello per il Machine Learning distribuito.

## CONCLUSIONI

In conclusione possiamo osservare che l'architettura implementata rappresenta correttamente un sistema P2P per System Utilization e Job Sharing attraverso l'uso di gRPC per la comunicazione tra nodi e del protocollo Gossiping per la diffusione delle informazioni nella rete. La presenza della Friends&Affinity e del Piggyback ha permesso maggiore efficienza per l'invio di job e novità nel sistema andando ad alleggerire il carico in rete, mentre la presenza di meccanismi anti-gregge e anti-isolazionismo si è rivelata cruciale per la decentralizzazione del sistema evitando così la creazione cluster interni, funzionalità evidenti osservando i grafici per la scoperta della rete da parte di un nodo. L'alta varianza di probabilità e bucket di probabilità ha permesso di caratterizzare il sistema con nodi diversi aventi nature e tendenze diverse come realisticamente avverrebbe in un sistema distribuito a contributo open source.

## REFERENCE

[1] Issues of Implementing Random Walk and Gossip Based Resource Discovery Protocols in P2P MANETs & Suggestions for Improvement - Ajay Arunachalam, Ohm Sornil.

[2.0 - 2.1] Improving Resource Discovery and Query Routing in Peer-to-Peer Data Sharing Systems Using Gossip Style and ACO Algorithm - Hamdi Hanane, Benchikha Fouzia

[3] Using Gossip for Dynamic Resource Discovery - Eric Simonton, Byung Kyu Choi, and Steven Seidel