Implementazione di un Sistema Gossip-based per distance estimation and failure detection

Maccari Gianluca

gianlu.maccari99@gmail.com

Università degli Studi di Roma Tor Vergata

Abstract

In questo documento viene descritta l'implementazione di un sistema distribuito, decentralizzato e scalabile che attraverso protocolli di gossiping stima le distanze tra i nodi di una rete ed individua eventuali fallimenti di tali nodi. Per la stima delle distanze viene utilizzata una implementazione semplificata dell'algoritmo di Vivaldi, mentre per l'individuazione di failure si sfruttano delle comunicazioni peer to peer basate su gossip.

1 Introduzione

Il sistema è formato da N nodi e da un service registry. Gli N nodi si registrano al server per ottenere il proprio identificatore, segnalare il proprio indirizzo IP agli altri nodi e per ricevere gli indirizzi dei nodi che possono contattare.

Poiché l'algoritmo di Vivaldi utilizza un sistema di coordinate, ogni nodo mantiene le proprie coordinate e quelle dei nodi della rete. Per stimare le distanze, ogni nodo utilizza la distanza euclidea con tali coordinate. Lo scambio delle informazioni sulle coordinate e delle distanze misurate avviene secondo un modello *Anti-Entropy* con strategia pull, in cui un nodo sceglie randomicamente un altro nodo e il nodo contattato risponde con le informazioni necessarie.

Per diffondere le informazioni sui fallimenti nella rete, i nodi sfruttano un protocollo di gossip. Per il sistema implementato si può selezionare, tramite file di configurazione, tra Bimodal Multicast Gossip e Blind Counter Rumor Mongering Gossip.

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.

2 Background

L'algoritmo di Vivaldi è un algoritmo decentralizzato che crea un sistema di coordinate sintetiche sopra una rete fisica di nodi, con l'obiettivo di approssimare con precisione il tempo di trasmissione tra coppie di nodi con le distanze nello spazio di coordinate generato. I nodi sfruttano i tempi di trasmissione dei messaggi osservati per aggiornare le proprie coordinate in funzione anche delle coordinate del nodo destinazione dei messaggi.

Lo scambio di messaggi segue il modello Anti-entropy. Inoltre, si sfrutta la strategia pull, ovvero le informazioni saranno inviate solamente dal nodo contattato.

Il Bimodal Multicast è un protocollo di gossiping diviso in due fasi. La prima fase, Message Distribution, sfrutta l'invio di un messaggio multicast non affidabile per diffondere il nuovo aggiornamento. La seconda fase, Gossip Repair, si basa sull'utilizzo di un digest contenente le informazioni di tutti gli aggiornamenti passati di cui un nodo è a conoscenza. Il digest viene allegato ad ogni messaggio in modo che un nodo ricevente possa individuare aggiornamenti persi.

Il Blind Counter Rumor Mongering diffonde le informazioni periodicamente tenendo conto di quante volte un aggiornamento è stato inoltrato e di quali nodi hanno ricevuto tale aggiornamento. L'algoritmo possiede due parametri: il primo per contare quante volte il nodo ha ricevuto l'aggiornamento e il secondo indica quanti nodi si possono contattare per iterazione.

3 Design

3.1 Strumenti

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 [3] dove è salvato il progetto, sono riportate tutte le informazioni per set up e lancio del software.

Un tool fondamentale per il testing del software è Netem, il quale fornisce servizi di emulazione di reti come emulazione ritardi di rete e perdite di pacchetti. Attenzione, i ritardi introdotti sono unidirezionali e vengono applicati solo in fase di invio di messaggi.

3.2 Assunzioni

Il software è stato progettato e implementato basandosi su tre assunzioni:

- I ritardi di rete devono essere costanti o non devono subire variazioni eccessive durante l'attività del software. Se così non fosse, si potrebbe osservare un tempo di convergenza dell'algoritmo di Vivaldi maggiore, e minore precisione dei risultati. Tuttavia, il sistema è stato testato anche in tali situazioni riportando discreti risultati;

- Il software implementato e la rete dei nodi devono essere correttamente configurati. I parametri del sistema riportati nel file di configurazione, consentono un'estesa configurazione che potrebbe, in casi particolari, portare ad oscillazioni o divergenza dei risultati. Pertanto, la configurazione deve essere redatta in modo opportuno considerando anche la rete di cui si ha a disposizione;
- Considerando tutte le tuple di tre nodi possibili nella rete, i ritardi di rete devono rispettare la disuguaglianza triangolare per il 95% di tali tuple. Questa assunzione è in linea con i risultati osservati in [1] in cui i dataset utilizzati presentano per meno del 5% violazioni della disuguaglianza.

3.3 Dettagli Implementativi

3.3.1 Tipologie messaggi. Nel sistema si identificano due tipologie di messaggi:

- **Vivaldi Message**: messaggio contenente le informazioni scambiate da due nodi per eseguire l'algoritmo di Vivaldi. Per i messaggi di richiesta informazioni, questo messaggio riporta: id, porta, coordinate, errore e (se presente) il digest del sender. Per i messaggi di risposta, riporta le precedenti informazioni del receiver e anche una map contenente le coordinate di altri nodi della rete;
- **Gossip Message**: messaggio contenente le informazioni di un fault nel sistema. Riporta le seguenti informazioni: id e porta del sender, id del nodo fault e digest. Il campo digest viene utilizzato per riportare le informazioni di un Vivaldi message riguardo un fault, alla componente software dedicata al gossiping.

Lo scambio di messaggi utilizza la serializzazione e deserializzazione fornita dallo standard JSON e le API fornite dal package Net.

3.3.2 Liste dei nodi. Per mantenere le informazioni dei nodi della rete sono implementate tre liste:

- due slice di *node*, la prima è una lista di nodi attivi e la seconda è una lista di nodi fault. I nodi riportati in queste due slice sono nodi che il nodo attuale può direttamente contattare;
- una map di puntatori a *vivaldiNodeInfo*, lista di tutte le coordinate dei nodi della rete. In questa lista compaiono nodi che non possono essere contattati e quindi nodi che non compaiono nelle due slice precedenti.

Node è una struct che contiene: id del nodo, indirizzo del nodo, lo stato del nodo, il numero di tentativi di contatto rimasti (contatti consecutivi, prima che venga segnalato fault) e l'ultimo round trip time osservato per il nodo. Lo stato può assumere tre valori: 0 non conosciuto, nodo che non è mai stato contattato; 1 attivo, nodo che è stato contattato e presente nella lista dei nodi attivi; 2 disattivo, nodo fault e presente nella lista di nodi fault.
VivaldiNodeInfo è una struct con campi:

- stato del nodo, 1 se il nodo è contattabile e quindi presente in una delle due slice di sopra, 2 se il nodo non è contattabile;
- un mutex, utile per gestire la concorrenza delle scritture;
- una variabile di tipo *coordinates*, che contiene le coordinate del nodo, l'errore e l'ultimo round trip time osservato.

L'identificatore del nodo a cui è associata una struct *vivaldiNodeInfo*, corrisponde alla chiave corrispondente nella map. La ripetizione dell'ultimo round trip time serve per un accesso più veloce a tale valore, poiché costituisce input per l'algoritmo di Vivaldi ma anche valore usato come timeout nelle comunicazioni tra nodi.

3.3.3 Separazione sottoreti. Il sistema può essere applicato per lavorare anche con reti formate da sottoreti differenti. La presenza delle tre liste riportate precedentemente e l'esistenza di nodi con cui non è possibile comunicare, permette una separazione tra nodi di differenti sottoreti. Come viene descritto più avanti, un nodo in una sottorete non può comunicare con un nodo di un'altra sottorete, tuttavia conosceranno le coordinate l'uno dell'altro. Però, le prestazioni e i risultati di Vivaldi sono peggiori del caso di una rete priva di sottoreti.
Per verificare questo comportamento è stata inserita la variabile *IGNORE_IDS*. Con questo parametro è possibile specificare insiemi di nodi che non possono comunicare tra loro, simulando sottoreti differenti.

4 Algoritmi e Protocolli

4.1 Algoritmo di Vivaldi

L'algoritmo di Vivaldi implementato nel progetto segue la descrizione di [1] nella versione semplificata, con coordinate tridimensionali e utilizzando la distanza euclidea.
L'implementazione segue lo pseudo codice riportato nella figura 1. Le uniche modifiche riguardano l'ordine di esecuzione dei passi.

vivaldi(rtt, x_j, e_j)
// *Sample weight balances local and remote error. (1)*
$w = e_i/(e_i + e_j)$

// *Compute relative error of this sample. (2)*
$e_s = \left|\|x_i - x_j\| - rtt\right|/rtt$

// *Update weighted moving average of local error. (3)*
$e_i = e_s \times c_e \times w + e_i \times (1 - c_e \times w)$

// *Update local coordinates. (4)*
$\delta = c_c \times w$
$x_i = x_i + \delta \times \left(rtt - \|x_i - x_j\|\right) \times \mathrm{u}(x_i - x_j)$

Figura 1: Pseudo codice dell'algoritmo di Vivaldi presente in [1]

La distanza euclidea, norma tra i vettori x_i e x_j dello pseudo codice, viene utilizzata per calcolare l'errore relativo, passo (2). L'errore relativo viene utilizzato per aggiornare l'errore del nodo corrente, passo (3).
L'errore del nodo viene aggiornato con una media mobile pesata, dove il peso corrisponde all'errore locale precedente diviso la somma di tale errore con l'errore remoto, inviato dal nodo contattato. Inoltre, per regolare questo aggiornamento, viene utilizzato il parametro *Precision Weight*. Dunque, l'esito del passo (3) impatta la prossima iterazione dell'algoritmo di Vivaldi.
Il parametro weight, peso della media mobile pesata dell'errore, viene anche utilizzato per calcolare il parametro delta, passo (4). Poiché il delta rappresenta l'*Adaptive Timestep*, questo valore controlla l'aggiornamento delle coordinate del nodo. Un valore troppo alto per il delta comporta un aggiornamento troppo violento delle coordinate che con lo scorrere del tempo potrebbe comportare divergenza. D'altra parte, un valore piccolo potrebbe rallentare la convergenza. Il valore di delta può essere anche regolato con un parametro configurabile, lo *Scale Factor*.
Infine, nel passo (4) si aggiornano le coordinate del nodo utilizzando il valore precedente sommato all'errore sulla distanza pesato con il delta. Inoltre, l'incremento delle coordinate viene moltiplicato per il vettore direzione, fondamentale per indirizzare l'incremento delle coordinate nella giusta direzione. Ricordiamo che l'algoritmo di Vivaldi può essere paragonato ad un sistema di molle in cui si cerca di trovare l'equilibrio delle forze. Il vettore direzione indica dove spostare le coordinate nell'aggiornamento per avvicinarsi alle coordinate di equilibrio.

L'algoritmo viene avviato quando un nodo riceve la risposta ad un messaggio di Vivaldi; quindi, quando si può misurare il round trip time effettivo con una comunicazione tra nodi. Tuttavia, non tutti i nodi potrebbero comunicare tra loro. Quindi alcune coppie di nodi non possono calcolare l'effettivo round trip time che li distanzia. Per ovviare a questo problema è implementata la funzione *unreachableHandler*(). Fissiamo due nodi che non possono comunicare direttamente: nodo 1 e nodo 2. Per avviare Vivaldi, occorre calcolare un round trip time artificiale ma significativo. Per fare questo si sfruttano nodi intermedi e i round trip time che i due nodi osservano con i nodi intermedi. Utilizzando i nodi 1 e 2, con un nodo intermedio, possiamo immaginare un triangolo avente come lati i ritardi tra i tre nodi (come riportato in figura 2).

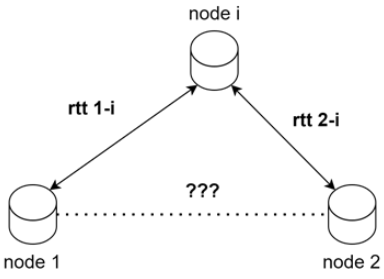

Figura 2: Esempio di nodi non comunicanti e utilizzo disuguaglianza triangolare.

A questo punto si sfrutta la disuguaglianza triangolare tra i lati del triangolo per restringere il valore massimo e il valore minimo che può assumere il round trip time tra i nodi 1 e 2 (come riportato in figura 3). Sfruttando più nodi intermedi, possiamo restringere l'intervallo ulteriormente. Inoltre, utilizzare più nodi intermedi, ci permette di ovviare il problema di violazioni della disuguaglianza. Infine, come round trip time artificiale si utilizza il valore medio tra valore massimo e valore minimo ottenuti. Con tale valore artificiale i due nodi non comunicanti possono avviare l'algoritmo di Vivaldi.

$$\min(rtt1\text{-}i, rtt2\text{-}i) < rtt1\text{-}2 < \max(rtt1\text{-}i, rtt2\text{-}i)$$

Figura 3: Disuguaglianza triangolare relativa alla figura 2

Se ignorassimo il problema, la convergenza sarebbe garantita ma le distanze osservabili tra nodi non comunicanti sarebbero randomiche. La soluzione implementata è greedy ma semplice.

Infine, un nodo contattato invia al sender anche le coordinate dei nodi che conosce selezionandoli randomicamente. Il sender aggiorna le coordinate di tali nodi senza avviare l'algoritmo di Vivaldi.

4.2 Algoritmi per failure detection

Entrambi gli algoritmi sono realizzati come struct che implementano una interface *FaultGossiper*, la quale espone tre metodi: *GossipFault*(), metodo che inoltra il fault ad altri nodi; *HandleGossipFaultMessage*(), metodo che si occupa di gestire un messaggio di gossip appena ricevuto e di gestire le informazioni dei fault ricevuti; *ReviveNode*(), metodo che va a rimuovere tutte le strutture dati per gestire un fault, di un nodo segnalato come fault ma attualmente vivo.

4.2.1 Blind Counter Rumor Mongering

Come detto in precedenza, questo algoritmo si basa su due parametri: *GOSSIP_MAX_NEIGHBOR*, o parametro b, che indica quanti nodi contattare per ogni iterazione; *GOSSIP_MAX_ITERATION*, o parametro f, che indica quante volte il messaggio può essere inoltrato o ricevuto dal nodo attuale.
L'algoritmo in questione mantiene una lista di struct di tipo *BlindInfoStruct*. Questa struct contiene l'id del nodo fault, una slice di nodi a cui inviare il messaggio gossip per il fault corrente e i valori attuali dei parametri b e f. La slice di nodi è fondamentale per mantenere traccia di quali nodi sono a conoscenza del fault. Quando un nodo riceve un messaggio di gossip da un certo sender, rimuove il sender dalla lista di nodi riguardanti il fault riportato nel messaggio. Quindi il gossiping di un fault termina quando tutti i nodi della struct sono stati notificati o se si raggiunge un numero di iterazioni di gossiping pari al parametro f. Inoltre, terminato il gossiping di un fault, la struct viene rimossa dalla lista di struct. La lista delle struct è gestita dal file *blindCounterStructhandler.go*.

Il gossiping viene avviato quando un nodo nota un fault, e gli altri nodi si uniscono al gossiping appena ricevono il messaggio di gossip oppure se anch'essi notano il fault.

4.2.2 Bimodal Multicast

Questo algoritmo è il meno adatto al sistema e la sua implementazione non corrisponde alla definizione teorica. L'algoritmo dovrebbe inviare un messaggio multicast non affidabile nella fase di Multicast, mentre nella seconda fase si effettua Gossip Repair. In questa implementazione, nella fase di Multicast, il nodo che inizia il gossiping invia un messaggio affidabile (tramite l'utilizzo di TCP) a tutti i nodi che può contattare (eventualmente, a tutti i nodi della sua stessa sottorete). La seconda fase viene sfruttata per la diffusione del fault ad eventuali nodi con cui il nodo iniziale non può comunicare. Infatti, questi ultimi nodi, tramite le informazioni riportate nel digest, possono essere aggiornati su fault per cui non hanno ricevuto un messaggio di gossip. Il digest viene gestito dal file *digest.go*, ed è implementato da una slice di identificatori e poi riportato nei messaggi come una stringa contenente tutti gli id dei nodi fault separati dal carattere "/".

4.3 Protocollo Lazzarus

Secondo le assunzioni effettuate ad inizio progetto, i nodi non dovrebbero subire variazioni eccessive dei ritardi o packet loss. Tuttavia, è stata implementata una soluzione ad un problema interessante. Per come è progettato il sistema, un nodo che non riesce a rispondere ai messaggi di Vivaldi o che non riesce ad inviarli, a causa di perdite messaggi o eccessivi ritardi, potrebbe finire in una situazione in cui tutti i nodi assumono tale nodo fault e il nodo stesso assume tutti i nodi come fault. Tale nodo si troverebbe nella situazione in cui non potrebbe comunicare con nessun altro, poiché assume che sia rimasto solo. Per ovviare a questo problema è stata implementata una funzione che si attiva soltanto quando la lista dei nodi attivi di un nodo sia vuota. In tal caso il nodo cambia lo stato di tutti i nodi fault a stato attivo. Rendendo tutti i nodi attivi, il nodo tenterà il normale invio di messaggi di Vivaldi, in questo modo anche gli altri nodi possono cambiare lo stato del nodo vittima ad attivo. La funzione in questione è *tryLazzarus*() e tale comportamento è configurabile con due parametri: *LAZZARUS_TRY*, parametro che indica per quanti tentativi consecutivi può eseguire la funzione e se si raggiunge tale valore, il nodo termina la sua esecuzione*; LAZZARUS_INTERVAL,* parametro che indica l'intervallo di tempo tra una esecuzione della funzione ed una successiva, in caso la precedente sia fallimentare.

5 Testing

Ogni nodo stampa con una certa frequenza: la lista dei nodi attivi e fault, seguita dalla lista delle coordinate dei nodi della rete e infine le distanze con tali nodi, misurate e calcolate con le coordinate.

Per accedere a tali output, dopo aver eseguito uno script di test, eseguire il comando *sudo docker logs <container-name>*.

La frequenza della stampa dei risultati, per ogni test, può essere configurata con la variabile *ITERATION_PRINT*.

5.1.1 SimpleNetem.sh. Nel primo test si assegna ad ogni nodo un ritardo fisso tramite il tool Netem. I seguenti risultati sono generati dopo 20 secondi di esecuzione dal nodo 1.

```
[PEER 1] active nodes
nodo id: 2  stato: 1  rtt: 27
nodo id: 3  stato: 1  rtt: 30
nodo id: 4  stato: 1  rtt: 25
nodo id: 5  stato: 1  rtt: 20
[PEER 1] fault nodes
None

[PEER 1] my coordinates: 6.25, -7.87, 5.52
[PEER 1] NODES COORDINATES
nodo: 2, state: 1 coordinate: 21.96, 8.90, -3.23
nodo: 3, state: 1 coordinate: 14.55, -0.30, 35.85
nodo: 4, state: 1 coordinate: 11.11, -30.06, 2.81
nodo: 5, state: 1 coordinate: -4.56, 3.33, 6.08

[PEER 1] DISTANCES
nodo: 2, rtt misurato: 27  rtt calcolato: 24.58
nodo: 3, rtt misurato: 30  rtt calcolato: 32.35
nodo: 4, rtt misurato: 25  rtt calcolato: 22.87
nodo: 5, rtt misurato: 20  rtt calcolato: 15.58
```

5.1.2 CrashNode.sh. In questo test viene eseguito lo script crashNode.sh, in cui vengono assegnati ritardi fissi e dopo alcuni secondi viene eseguito il comando *sudo docker kill node2*, il quale arresta l'esecuzione del container nodo2 simulando un crash.

I seguenti risultati sono generati dal nodo 1 prima del crash. Si può notare come il nodo due sia ancora attivo.

```
[PEER 1] active nodes
nodo id: 2  stato: 1  rtt: 27
nodo id: 3  stato: 1  rtt: 30
nodo id: 4  stato: 1  rtt: 25
nodo id: 5  stato: 1  rtt: 20
[PEER 1] fault nodes
None
```

I successivi risultati sono generati dal nodo 1 dopo il crash del nodo 2. Si può notare dalle prime righe che il nodo 1 nota una interruzione dell'esecuzione di *sendVivaldiMessage*(), dovuta da un timeout scaduto durante la comunicazione con il nodo 2. Tuttavia, il parametro *FAULT_MAX_RETRY* è impostato a 3, quindi non viene assunto node 2 come fault ma viene decrementato il suo valore *Retry*.

Nella terza riga si può notare che il nodo 1 riceve un messaggio di gossip dal nodo 5 segnalante il fault di nodo 2. BCRM indica che il

tipo id gossip utilizzato nella segnalazione dei fault è il Blind Counter Rumor Mongering. Si può notare che successivamente al messaggio di fault, il nodo 2 viene segnalato come fault.

[PEER 1] sendVivaldiMessage()--> timeout scaduto per 2
[PEER 1] time out expired for node: 2 retry left: 2
[PEER 1] BCRM, received gossip message from: 5 fault node: 2
[PEER 1] BCRM, gossip from: 5 about fault node: 2 added to my knowledge

[PEER 1] active nodes
nodo id: 3 stato: 1 rtt: 30
nodo id: 4 stato: 1 rtt: 25
nodo id: 5 stato: 1 rtt: 20
[PEER 1] fault nodes
nodo id: 2 stato: 2

5.1.3 SimpleNetem.sh con nodi non comunicanti. In questo test si esegue lo script *simpleNetem.sh* impostando il parametro *IGNORE_IDS* in modo tale che i nodi 1, 2 e 3 non possano comunicare direttamente. L'output è stampato da nodo 1 e si può osservare che i nodi 2 e 3 non compaiono né tra i nodi attivi né tra i nodi fault. Tuttavia, i nodi sono presenti nella lista delle coordinate dei nodi, dove sono segnalati come *unreachable* e compaiono anche i round trip time artificiali.

[PEER 1] active nodes
nodo id: 4 stato: 1 rtt: 25
nodo id: 5 stato: 1 rtt: 20
[PEER 1] fault nodes
None

[PEER 1] my coordinates: 12.50, 32.74, 34.90
[PEER 1] NODES COORDINATES
nodo: 4, state: 1 coordinate: 39.84, 15.54, 20.51
nodo: 5, state: 1 coordinate: 12.15, 18.46, 19.53
nodo unreachable: 2, state: 2 coordinate: -12.49, 27.36, 29.07
nodo unreachable: 3, state: 2 coordinate: -2.87, 6.91, 7.69

[PEER 1] DISTANCES
nodo: 4, rtt misurato: 25 rtt calcolato: 35.35
nodo: 5, rtt misurato: 20 rtt calcolato: 20.99
nodo non reachable: 2, rtt artificiale: 27.00 rtt calcolato (con coordinate): 26.22
nodo non reachable: 3, rtt artificiale: 30.00 rtt calcolato (con coordinate): 40.54

5.2 VariableNetem.sh. Il seguente test è effettuato eseguendo lo script *variableNetem.sh*. Nel test vengono assegnati dei ritardi che possono subire variazioni randomiche. Si può notare come le distanze calcolate tramite le coordinate si discostano significativamente rispetto ai valori osservati nelle comunicazioni.

[PEER 1] DISTANCES
nodo: 2, rtt misurato: 50 rtt calcolato: 60.50
nodo: 3, rtt misurato: 53 rtt calcolato: 58.23
nodo: 4, rtt misurato: 49 rtt calcolato: 49.17
nodo: 5, rtt misurato: 38 rtt calcolato: 22.61

I risultati di questo test potrebbero sembrare non accettabili. Tuttavia, occorre sottolineare che le variazioni dei ritardi riportate nello script di test possono decrementare o incrementare il ritardo base. Ad esempio, per il nodo 2 il comando *Netem* termina con *delay 34ms 7ms*, il che significa che il nodo 2 può subire un ritardo in fase di invio messaggio che varia tra 27 ms e 41 ms. Inoltre, i ritardi devono essere considerati a coppie di nodi. Ovvero il nodo 1 ha componente di ritardo *delay 20ms 5ms*; quindi, nella comunicazione tra nodo 1 e 2 possiamo osservare un ritardo che oscilla tra 42 ms e 67 ms. I risultati ottenuti sono fortemente influenzati dalle variazioni dei ritardi e dunque sono accettabili.

5.3 PacketLossTest.sh e lazzarus. Il seguente test è effettuato lanciando lo script *packetLossTest.sh*. Lo script assegna inizialmente dei ritardi fissi. Successivamente imposta un valore di packet loss per il nodo 2 pari a 100%. Il nodo, quindi, non può inviare alcun messaggio, tuttavia, rimane ancora attivo. I seguenti messaggi di stampa sono generati dal nodo 1 che osserva dei timeout scaduti per il nodo 2. Quindi avvia il gossiping, di conseguenza la rete assume il nodo 2 fault.

[PEER 1] sendVivaldiMessage()--> timeout scaduto per 2
[PEER 1] time out expired for node: 2 retry left: 1
[PEER 1] sendVivaldiMessage()--> timeout scaduto per 2
[PEER 1] time out expired for node: 2 no retry left. Fault node!
[PEER 1] BCRM, sending gossip message to: 3 about fault node: 2
[PEER 1] BCRM, sending gossip message to: 4 about fault node: 2
[PEER 1] BCRM, gossip iteration done
[PEER 1] BCRM, sending gossip message to: 5 about fault node: 2
[PEER 1] BCRM, gossip iteration done
[PEER 1] active nodes
nodo id: 3 stato: 1 rtt: 30
nodo id: 4 stato: 1 rtt: 25
nodo id: 5 stato: 1 rtt: 10
[PEER 1] fault nodes
nodo id: 2 stato: 2

Il nodo 2 osserva dei timeout per tutti i nodi; quindi, assume che tutti i nodi siano fault. A questo punto, poiché la lista di nodi attivi del nodo 2 è vuota, avvia il protocollo Lazzarus; quindi, rende tutti i nodi attivi e tenterà l'invio di messaggi Vivaldi. I valori dei round trip time stampati dipendono dai messaggi Vivaldi inviati dopo l'esecuzione della funzione *tryLazzarus*().

[PEER 2] TRYING LAZZARUS PROTOCOL
[PEER 2] active nodes
nodo id: 1 stato: 1 rtt: -1
nodo id: 5 stato: 1 rtt: 0
nodo id: 3 stato: 1 rtt: 20
nodo id: 4 stato: 1 rtt: 15

[PEER 2] fault nodes
None

Infine, possiamo osservare i risultati del nodo 1 dopo che la funzione 2 ha attivato il protocollo Lazzarus. La prima riga viene stampata dalla componente gossiper. Notiamo come il nodo 2 successivamente viene riportato tra i nodi attivi.

[PEER 1] BCRM, LAZZARUS node: 2
[PEER 1] active nodes
nodo id: 3 stato: 1 rtt: 30
nodo id: 4 stato: 1 rtt: 25
nodo id: 5 stato: 1 rtt: 10
nodo id: 2 stato: 1 rtt: 10
[PEER 1] fault nodes
None

6 Possibili estensioni

Il sistema può essere esteso con alcune modifiche per ottenere risultati più precisi e più stabili nel tempo. Nell'implementazione, infatti, la scelta dei nodi da contattare o delle coordinate da allegare ad un messaggio Vivaldi di risposta, sono randomiche. Si potrebbe implementare un criterio di selezione di nodi meno recentemente contattati o meno conosciuti.

Un'altra possibile estensione potrebbe essere quella di implementare un sistema che decrementi il valore del delta nell'algoritmo di Vivaldi con l'avanzare delle iterazioni. Una possibile implementazione potrebbe essere un delta decadente, come il *learning rate dacay*, concetto del paradigma del Machine Learning. In questo modo, andando avanti con le iterazioni, gli aggiornamenti apportati alle coordinate sarebbero meno significativi e i risultati di Vivaldi sarebbero più stabili.

Un'ultima possibile estensione potrebbe essere quella di interrompere l'esecuzione dell'algoritmo di Vivaldi quando le coordinate ottenute, stimato adeguatamente le distanze tra nodi.

7 Conclusioni

7.1 Distance node estimation. I test dimostrano come il progetto sia funzionale e ottenga risultati accettabili. Applicando il sistema in reti dove le assunzioni fatte in fase di progettazione sono rispettate, è possibile osservare migliori risultati, ma anche non rispettando le assunzioni, i risultati sono accettabili.

7.2 Failure Detection. Il sistema riesce correttamente ad individuare le failure del sistema e a diffondere l'occorrenza di tali eventi in modo rapido. Inoltre, i parametri di configurazione permettono un servizio di failure detection flessibile rispetto ai requisiti del caso applicativo.

References

[1] Frank Dabek, Russ Cox, Frans Kaashoek, Robert Morris. *Vivaldi: A Decentralized Network Coordinate System*. MIT CSAIL Cambridge, MA.

[2] Ruosi Cheng, Yu Wang. *A Survey on Network Coordinate Systems*. Luoyang Electronic Equipment Test Center of China, Luoyang, China. Henan University of Engineering, Xinzheng, China

[3] Maccari Gianluca. Repository GitHub: https://github.com/jack-mack15/SDCC_repo.git