Progetto di Sistemi Distribuiti

Dott. Borsoi Diego
Dott. Callegari Filippo
DMIF, University of Udine, Italy
(also the authors are distributed)

Version $\frac{1}{4}$, 10 maggio 2021

Indice

Capitolo 1

Introduzione

Il progetto in questione riguarda la creazione di un sistema distribuito per la comunicazione fra dispositivi all'interno di una rete sparsa, tramite l'utilizzo di eventi.

1.1 Descrizione del problema

Ogni dispositivo corrisponde ad un nodo della rete ed è caratterizzato da:

- **Id** : numero univoco del nodo.
- **Stato** : tupla di variabili che rappresentano delle specifiche caratteristiche del nodo (es. temperatura di un sensore, stato di accensione di una termocoppia, ecc).
- **Regole** : insieme di regole del tipo ECA (Event Condition Action) che possono attivarsi a seguito di un evento inviato al nodo. Queste regole possono essere di due tipi: locali, l'azione modifica solamente lo stato del nodo in cui si attiva, oppure globale, l'azione viene inviata a tutti i nodi della rete perché venga letta ed, in caso la valutazione della guardia associata sia positiva, eseguita.

$$\{event;\ condition;\ action \mid \text{if } guard \text{ then } action\}$$

Lo stato della rete si evolverà ogni qual volta un evento verrà attivato, andando a sua volta ad innescare eventuali nuovi eventi e creando quindi una sequenza di azioni a cascata.

1.2 Struttura dell'implementazione

La rete in questione ha una struttura a mesh sparsa (cioè ogni nodo sarà al più connesso a un numero di nodi molto basso, rispetto alla totalità). I nodi sono idempotenti in modo tale da avere un sistema fortemente decentralizzato. Le

varie comunicazioni fra i nodi sono eseguite al di sopra di connessioni TCP, in tal modo possiamo garantire la consegna di ogni messaggio nell'ordine prestabilito. Per quanto concerne invece le comunicazioni riguardanti il sistema di *heartbeat* (3.2), queste vengono eseguite utilizzando connessioni UDP.

1.3 Trasparenze

Di seguito sono descritte le trasparenze che concergono e sono implementate dal sistema:

Trasparenza ai fallimenti: nel momento in cui un nodo fallisce/si disconnette, il resto della rete continua a funzionare normalmente.

Trasparenza alla scalabilità: la rete può espandersi in dimensione senza che il funzionamento dei nodi vari.

Trasparenza alla mobilità: un nodo può spostarsi all'interno della rete senza che il funzionameto suo e degli altri nodi vada a modificarsi.

1.4 Algoritmi

Il sistema implementa solamente due algoritmi:

- **Flooding Algorithm:** viene usato inizialmente per la comunicazione di un'azione a tutta la rete nel momento in cui in un nodo una regola globale o di transazione viene attivata.
- **Lamport clock (modificato):** viene utilizzata una versione modificata del Lamport clock per identificare i vari $flood$ eseguiti; questo clock viene incrementato solamente dall'invio (o ricezione) di un messasggio, e non dalle azioni interne ad un nodo;
- **Distributed Spanning Tree:** viene usato per ridurre il traffico di rete quando la rete si è "stabilizzata". Questo algoritmo mira a creare un albero di copertura nella rete.

Il sistema non implementa particolari algoritmi essendo che si vuole realizzare una rete distribuita dove ogni nodo conosce esclusivamente i vicini ed evolve il suo stato solamente a causa di eventi ricevuti tramite dei messaggi.

1.5 Testing

Per testare il sistema verrà utilizzata un'entità *Ambiente*, la quale simulerà:

- la creazione iniziale della rete, caricando da dei file appositi la struttura degli stati, la lista di regole e la conformazione della rete
- la scoperta di nuove connessioni

- variazioni di variabili legate all'ambiente (es. temperatura registrata da un sensore)
- fallimenti di nodi
- ritardi nell'invio di messaggi fra nodi

Ogni modulo verrà testato singolarmente ed infine verranno eseguiti dei test completi del sistema.

1.6 Piano di sviluppo

Le future fasi di sviluppo seguiranno il seguente ordine:

1. Riunione con il committente per convalidare la risoluzione del problema
2. Implementazione ambiente virtuale per la gestione dei nodi
3. Implementazione della struttura del nodo
4. Implementazione sistema *heartbeat*
5. Implementazione del sistema algoritmico
6. Test totale
7. Validazione

Capitolo 2

Analisi

In questo capitolo vengono descritti nel dettaglio requisiti funzionali e non funzionali della soluzione proposta.

2.1 Requisiti Funzionali

I requisiti funzionali individuati sono:

- **Categorizzazione di un nodo:** ogni nodo ha un tipo il quale ne identifica lo stato e le sue regole;
- **Modifica delle regole di un nodo:** ogni tipo di nodo può avere le sue regole, codificabili attraverso la programmazione dello stesso;
- **Modifica dello stato di un nodo:** ogni evento permette di avere o degli *effetti locali*, degli *effetti globali* o degli *effetti transizionali*:
  - ***effetti locali***: la regola va a modificare lo stato interno;
  - ***effetti globali***: la regola può modificare lo stato delle variabili interne, e può generare un evento sugli altri nodi;
  - ***effetti transizionali***: regola simile a quelle globali, ma con la differenza che l'esecuzione avverrà in tutti i nodi coinvolti "contemporaneamente";
- **Aggiunta dinamica di un nodo:** un nodo può essere aggiunto alla rete in qualsiasi momento senza perturbarne la dinamicità, limitando l'aggiornamento ai vicini a cui si collega.
- **Esecuzione di un'azione ricevuta dai vicini:** nel momento in cui un nodo riceve un messaggio dai propri vicini esso va a verificare la soddisfacibilità della guardia (se presente) e nel caso di una valutazione positiva viene eseguita l'azione associata, andando quindi a modificare il proprio stato.

- **Attivazione di una regola:** ogni qual volta avviene un cambiamento nello stato di un nodo, viene eseguito un controllo delle regole, per vedere se gli eventi generati possano attivare una o più regole del nodo; nel caso in cui una regola venga attivata, in base al tipo (locale, globale o transazione) viene portata a termine l'azione corrispondente.

2.2 Requisiti Non Funzionali

I requisiti non funzionali individuati sono:

- **Decentralizzazione:** nessun nodo ha il controllo dell'ordine degli eventi, grazie al fatto che ogni nodo è idempotente;
- **Tolleranza ai guasti:** poiché tutti i nodi sono idempotenti, nel momento in cui un nodo si scollega dalla rete, la rete rimanente continua ad operare normalmente;
- **Etereogenità:** fintanto che i nodi aggiunti utilizzano il protocollo descritto, qualunque nodo di qualunque tipo (hardware o categoria) potrà essere aggiunto alla rete;
- **Scalabilità:** l'aggiunta dinamica dei nodi alla rete permette di scalare orizzontalmente con estrema facilità;
- **Trasparenze:** le trasparenze implementate sono quelle descritte al capitolo precedente (paragrafo 1.3).

Capitolo 3

Progetto

In questo capitolo vengono descritti in modo più approfondito l'achitettura del progetto, i moduli, i protocolli e gli algoritmi utilizzati.

3.1 Architettura logica

Essendo tutti i nodi costruiti al medesimo modo, di seguito presentiamo la struttura di uno singolo di essi.

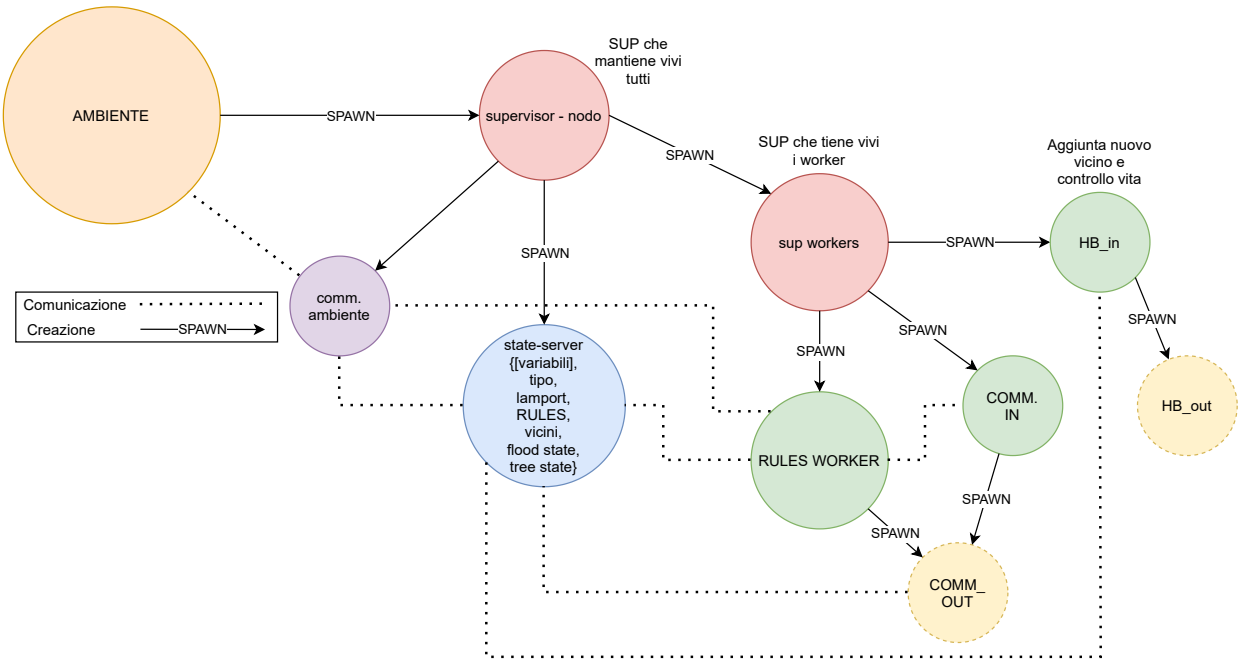

Figura 3.1: Struttura gerarchica dei moduli di un nodo e visualizzazione delle connessioni fra di essi.

Come si può vedere dalla figura 3.1, ogni nodo è formato dai seguenti moduli:

- **Supervisor nodo:** modulo che cerca di manterene sempre operativi gli altri moduli interni. Se questo componente si riavvia equivarrebbe ad un riavvio del nodo, e quindi la conseguente perdita della modifica agli stati interni. I moduli da lui controllati sono:
  - **State server:** questo modulo manterrà tutte le variabili locali al nodo, che verranno modificate durante l'operatività dello stesso. Queste variabili sono:
    - stato del nodo;
    - lista delle regole associate al nodo;
    - tipo del nodo;
    - stato delle waves;
    - informazioni sui nodi vicini;
    - id univoco del nodo;
  - **Supervisor dei workers:** modulo che si occupa di gestire i moduli a lui dipendenti, riavviandoli in caso di "crash". Questi sono:
    - **heartbeat sense:** si occupa di controllare la "vitalità" dei vicini, di istanziare le connessioni con essi e di mantenere la "spanning tree" della rete. Ogni qual volta che deve comunicare con dei vicini affini, creerà un processo effimero (HB_out);
    - **communication in:** si occupa di gestire tutti i messaggi in ingresso relativi al nodo in questione;
    - **rules worker:** si occupa di applicare le azioni ricevute dai nodi vicini ed eventualmente eseguire una delle regole a lui locali al momento dell'attivazione; lui si occuperà anche di propagare le regole che generano degli eventi verso gli altri nodi, interpellando il modulo "*communication out*" (modulo apposito per l'invio dei messaggi ai nodi vicini).

Come è stato accennato in precedenza verrà sviluppato un ulteriore modulo chiamato "***Ambiente***": questo modulo permette di simulare le interazioni che avverrebbero nel mondo reale. Nel dettaglio, le funzionalità del modulo sono:

- il "discovery" dei vicini;
- cambiamento delle variabili non dipendenti dalle regole (temperatura dell'ambiente/GPS/...);
- simulazioni di disservizi di rete;
- simulazione di guasti (temporanei o non) di un nodo;
- topologia della rete.

Dal punto di vista del nodo ci troviamo quindi costretti ad aggiungere un ulteriore modulo fittizio (***comunicazione ambiente***) per permettere la comunicazione con l'ambiente.

3.2 Protocolli ed algoritmi

Di seguito verranno descriti nel dettaglio i vari protocolli ed algoritmi utilizzati.

Controllo e attivazione delle regole

Nel momento in cui il sistema riceve un'azione da eseguire (dopo aver controllato la validità e che appartenga ad una wave non ancora ricevuta) si innesca la seguente serie di azioni:

1. il modulo *communication IN* invia l'azione da eseguire al modulo *rules worker*;
2. quest'ultimo utilizza una funzione dello *state server* per modificare lo stato del nodo in accordo all'azione ricevuta;
3. il *rules worker* esegue quindi un controllo sulle regole andando ad identificare quali possono essere attivate dalla modifica appena eseguita;
4. per ciascuna regola che viene attivata viene testata la condizione e in caso di risultato positivo:
   - se la regola è del tipo *locale*, viene eseguita l'azione associata
   - se invece la regola è del tipo *globale*, viene eseguita l'azione associata (in caso di guardia con valutazione positiva) e viene passata ad un processo di *communication OUT*, istanziato appositamente, che genera una nuova wave di messaggi inviando ai vicini la nuova azione.

Heartbeat

Il protocollo di "heartbeat" serve per mantenere consistente lo stato dei vicini di un nodo. Questo infatti controlla la loro vitalità e sarà componente chiave per l'aggiunta di un nuovo nodo.

L'algoritmo si suddivide quindi in tre componenti:

- *ECHO*: similmente al protocollo ICMP, si fa una richiesta di echo al vicino. Se questa "*ECHO_RQS*" andrà a buon fine, il primo nodo che istanzia una richiesta di "*ECHO*" richeverà un pacchetto di "*ECHO_RPL*". Definiamo come τ il tempo che intercorre tra un messaggio di *ECHO* ed un'altro. Se un nodo non rispondesse entro 2τ ad una *ECHO_RQS*, questo verrà considerato come non più collegato. Ogni *ECHO_RPL* conterrà il clock attuale del vicino.

- *ADD_NEW_ND*: similmente al protocollo DHCP, nella fase di aggiunta di un nodo alla rete, il nuovo nodo si annuncia al suo vicino "fisico", chiedendo le informazioni essenziali per poter partecipare attivamente alla rete. Questo sarà spiegato più in dettaglio nella prossima sezione.
- *TREE_STATE*: questa componente si focalizza sulla gestione dell'albero di copertura della rete. Sarà analizzato in dettaglio nei paragrafi successivi.

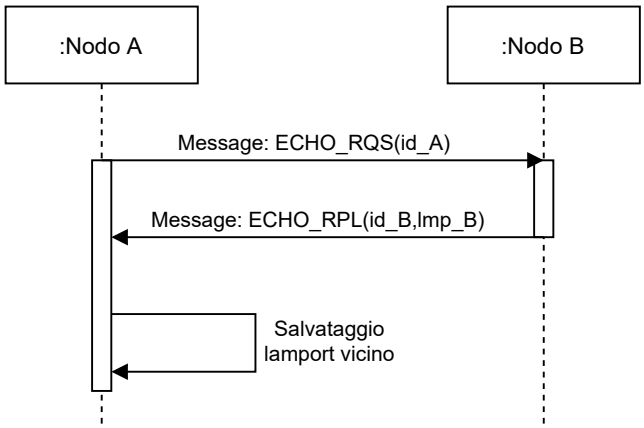

Figura 3.2: Sequence diagram dei messagi usati per il sistema di *heartbeat*.

Aggiunta di un nuovo nodo

L'aggiunta di un nodo è una parte complicata del sistema: bisogna tener conto della possibilità che la rete si partizioni. Questo si può verificare nel caso in cui un nodo si riavvii. Il partizionamento della rete è visto come caso particolare di aggiunta di un nodo alla rete.

L'aggiunta di un nuovo nodo si compone dei seguenti passi:

1. *ADD_NEW_ND*: il nuovo nodo manda la richiesta di aggiunta alla rete a tutti i suoi vicini inviando il proprio id;
2. *ADD_NEW_NB*: il nodo che deve aggiungere il nuovo nodo risponde con un messaggio contenente $(clock, id)$.

A questo punto, una volta che per ogni vicino ho i suoi parametri di clock, i casi possibili sono 2:

1. **tutti i nodi hanno medesimo clock:** imposto il mio clock al clock comune;
2. **esiste un clock massimo:** imposto il mio clock al valore massimo, e rispondo con *UPD_LMP* a tutti i miei vicini che non hanno il clock al

massimo. Questi a loro volta manderanno a tutti i loro vicini, con clock diverso da quello scelto, il nuovo valore.

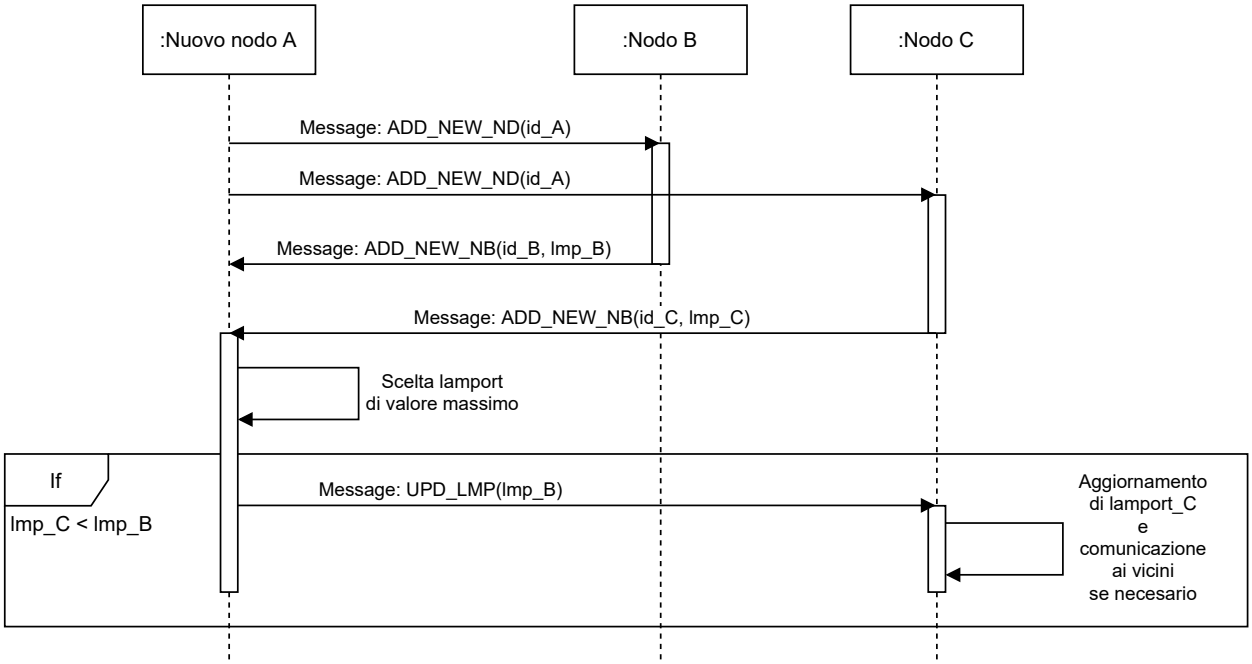

Figura 3.3: Sequence diagram dei messagi usati per laggiunta di un nuovo nodo alla rete.

Two-phases Rules (Transazioni)

Questo è un protocollo definito come "two-phases commit", ovvero permette l'esecuzione in contemporanea su più parti di almeno un'azione. Questo protocollo è stato modificato rispetto l'originale, in quanto, non avendo l'obbligo di avere sempre uno stato consistente tra i nodi, ci permette di semplificarlo. Le fasi sono quindi le seguenti:

1. *discovery*: troviamo tutti i nodi nella rete che sono interessati ad eseguire la transazione. Il nodo inizializzatore manda nella rete un messaggio di richiesta di transazione, il quale contiene delle condizioni per partecipare. Se un nodo è interessato, risponderà all'inizializzatore dicendogli di essere interessato.

2. *start*: il nodo inizializzatore darà inizio alla transazione per i nodi interessati.

All'interno di questo protocollo, come possiamo vedere nell'immagine 3.4, vengono tenuti dei timeout di controllo, al fine di non rimanere bloccato nello

stato di transizione: se un nodo decide di partecipare alla transazione, allora non potrà eseguire altre regole.

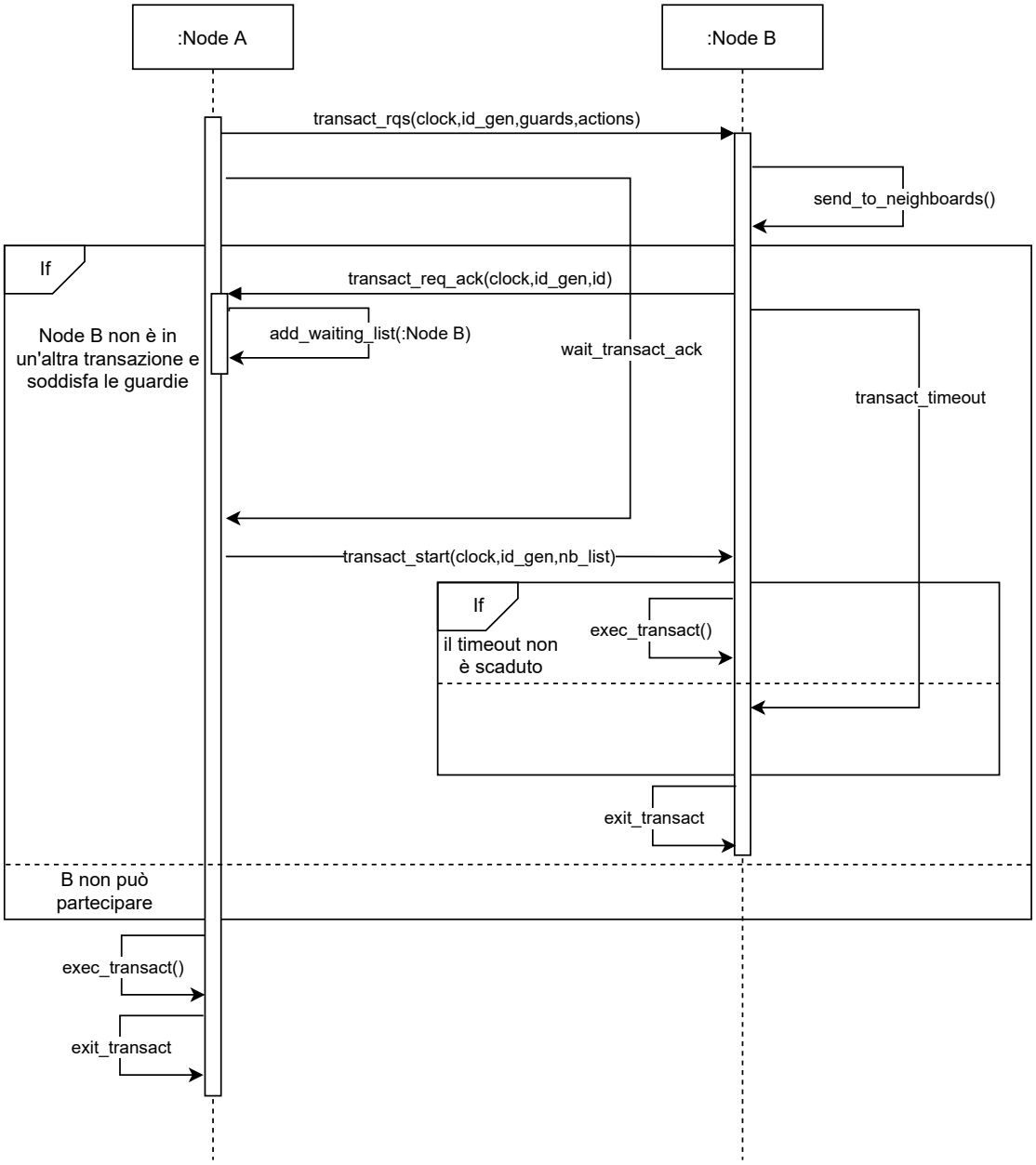

Figura 3.4: Sequence diagram del protocollo adottato per le regole di transaction.

Algoritmo: Flooding

Ogni qual volta si crea una regola che genera un evento "globale", ogni nodo spedirà ai suoi vicini un messaggio contenenta un'azione da eseguire. Questi, una volta ricevuto, aggiorneranno il loro "wave count" (tenuto dal clock, spiegato successivamente), e passerà all'esecuzione dell'istruzione condizionata contenuto nel messaggio. Al fine di evitar la presenza di messaggi vecchi nella rete, ogni messaggio conterrà la coppia $(clock, id\_gen)$: questo verrà salvato localmente nel nodo ricevente, e verrà mantenuto in memoria al fine di verificare se un messaggio ricevuto non è già stato eseguito. Ogni nodo quindi spedirà una copia del messaggio a tutti i suoi vicini (a patto non l'abbia già ricevuto in passato), meno a quello da cui l'ha ricevuto. Queste strategie applicate faranno si che i messaggi circolanti nella rete siano il minor numero possibile, e nell'eventuale creazioni di cicli nella rete, dovuta alla topologia della stessa, siano soppressi alla prima occasione utile.

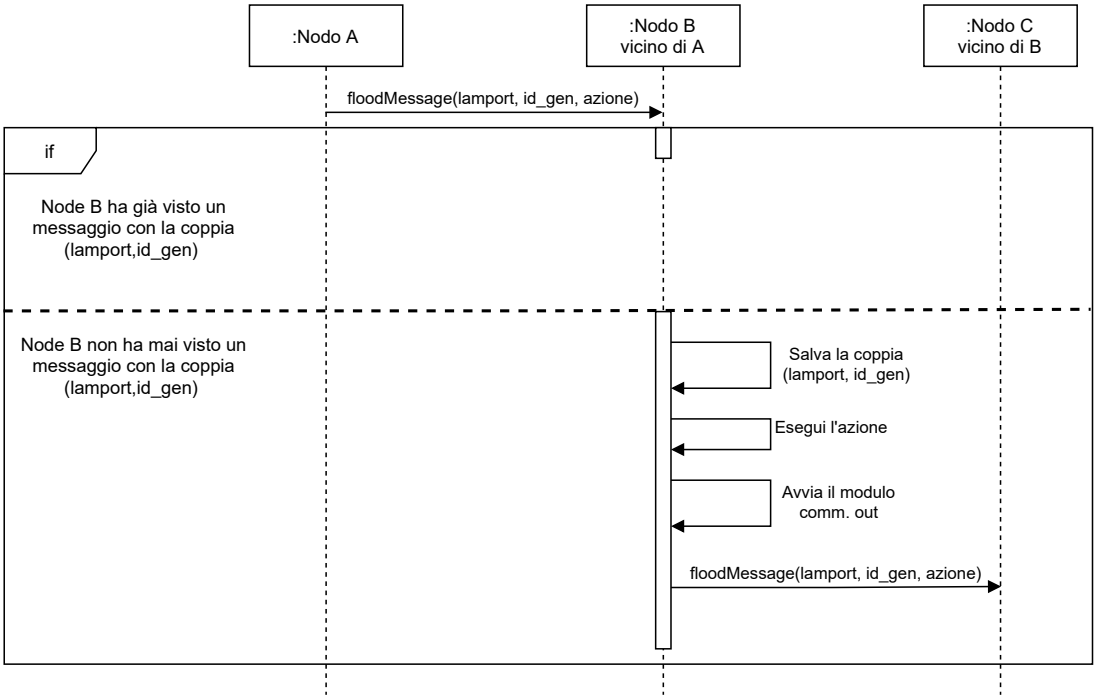

Figura 3.5: Sequence diagram dei messaggi usati per l'algoritmo di flooding.

Algoritmo: Lamport modificato

Il "Clock di Lamport" ci permette di dare una certa conseguenza temporale alle azioni. Questo sarà un numero intero crescente, ed identificherà ogni wave generata. Alla generazione di una wave ogni nodo userà il suo clock interno,

incrementato, per identificarla. Questo ci permette, come già abbiamo spiegato, di controllare la propagazione dei messaggi. Sia quindi Lmp_i il clock della nuova wave ricevuta e Lmp_n il clock del nodo corrente. Le azioni intraprendibili sono:

- $Lmp_i = Lmp_n + 1$: eseguo immediatamente l'azione in essa contenuta ed aggiorno il valore di Lmp_n;
- $Lmp_i > Lmp_n + 1$: aspetto un preimpostato timeout (es: 5τ) prima di eseguire l'azione di Lmp_i, in modo tale da poter ricevere le wave mancanti, e quindi mantenere l'ordine causale delle azioni;
- $Lmp_i < Lmp_n$: vado ad eseguire l'azione solo nel caso in cui tutte le variabili coinvolte non siano già state modificate da una wave ad essa successiva, cioè con $clock > Lmp_i$.

Algoritmo: Distribuited Spanning Tree

L'albero di copertura (o spanning tree) è un protocolo distribuito tra i nodi ideato per limitare il numero di messaggi nella rete. Questo protocollo risulta essere una versione modificate el più noto *STB* (IEEE 802.1aq). Tutto questo protocollo viene eseguito sempre dal sistema di heartbeat, in quanto strettamente correlati.

Questo protocollo si divide in 3 fasi:

- ***discovery***: in questa fase, il nodo che si sta connettendo esplora la rete, e dice a tutti di essere la radice. A seconda delle condizioni, esplicitate in dettagli nel capitolo successivo, potrà effettivamente esserlo o meno;
- ***listening***: in questa fase, l'albero risulta stabile. Ogni $\bar{\tau}$ la radice "eletta" manda dei pacchetti di keep alive ai figli. Nel momento in cui un la radice cesserà la sua permanenza nella rete, si passerà alla fase successiva;
- ***transiction***: in questa fase, per almeno uno componente della rete, la radice risulta non più raggiungibile. In questa fase i nodi coinvolti, non possono accettare la vecchia radice. Questa fase avviene quando un nodo perde la sua "radice", o quando lo spanning tree perde la radice che lo genera.

La scelta della radice avviene in maniera deterministica, sfruttando l'identificatore univoco di cui ogni nodo è dotato. Definiamo $< saved\_root, saved\_len, saved\_nh >$ la tupla che identifica i dati di radice attualmente salvata, la mia attuale distanza e attraverso quale vicino riesco a raggiungerla, mentre definiamo come $< root, len, nh >$ la tupla proposta dai miei vicini. A questo punto la scelta per un nodo della sua radice avviene come segue:

- $saved\_root > root$: la radice proposta è inferiore alla mia e sarà scelta come mia nuova radice;
- $saved\_root = root \wedge saved\_len + 1 < len$: il mio vicino è ha una distanza peggiore della mia, lo avviso che il mio percorso potrebbe essere migliore;

- $saved\_root = root \wedge saved\_len > len + 1$: la mia distanza è peggiore di quella proposta, mi sposto sul mio vicino, in maniera di esser più vicino alla radice;
- $saved\_root = root \wedge saved\_len = len + 1 \wedge nh\_saved > nh$: per questioni di determinismo, se la distanza e la radice proposta sono identiche, scelgo come "porta" verso la radice il mio vicino con id minore degli altri.

A questo punto possiamo scegliere anche a quali dei nostri possiamo inviare i messaggi: poichè ogni nodo è informato se è un nodo scelto o meno, se inoltriamo i messaggi solo ai vicini in cui sono stato scelto o al vicino che io ho scelto, posso raggiungere tutta la rete con un numero minimo di messaggi. A tal fine ogni vicino potrà assumere 3 vari stati:

- *root_port*: vicino scelto per raggiungere la radice;
- *active*: vicino che mi ha scelto per raggiungere la radice;
- *disabled*: vicino che non dipende da me.

3.3 Architettura fisica e deployment

Per quanto riguarda l'architettura fisica è necessario l'utilizzo di microcalcolatori, come dei "Raspberry" o degli "Arduino". Ogni nodo corrisponderebbe fisicamente ad una di queste schede, avendo quindi la possibilità d'utilizzare più nodi per il medesimo "apparato".

Non è strettamente necessario l'ausilio di microcalcolatori: potrei concentrare all'interno di un singolo calcolatore più nodi, a patto che siano gestiti in maniera consona.

Come descritto in precedenza, questi comunicherebbero con protocollo TCP ed UDP, non interessandoci quindi di tutta la rete sottostante.

Visto l'esempio a cui abbiamo pensato, si ritiene ideale l'utilizzo di connessioni wireless.

3.4 Piano di sviluppo

Le **funzionalità di base** che verranno implementate sono:

- comunicazione tra nodi;
- sistema di flooding;
- sistema di heartbeat;
- gestione dell'aggiunta di un nodo alla rete;
- gestione del riavvio di un nodo nella rete;
- impostazione iniziale dei nodi (ambiente);

- programmazione dei nodi;
- sistema basico d'esecuzione delle regole.

Sono state inoltre individuate le seguenti **funzionalità avanzate**:

- sistema avanzato d'esecuzione delle regole;
- salvataggio dello stato del nodo su files interni al controllore;
- riprogrammazione dinamica del nodo;
- implementazione di un *calculus* locale.

Capitolo 4

Implementazione

In questo capitolo tratteremo le scelte implementative avvenute nel corso del progetto. ci focalizzaremo man mano nei vari dettagli, cercando di spiegarne le motivazioni delle varianti adottate dai vari algoritmi o protocolli standard.

4.1 Software e hardware

Per l'implementazione non si è seguita una vera e propria scelta hardware, ma si è tenuto conto i vincoli, quali scarsa capacità computazionale e scarsa memoria.

A livello software la scelta è ricaduta su Erlang, in quanto linguaggio fortemente orientato ai threads, funzionale e con spiccata capacità alle connessioni. Altra peculiarità sono le librerie messe a disposizione dallo stesso tramite OTP (Open Telecom Platform). Di queste librerie sfrutteremo fortemente dei "behavior module", quali *gen_server* e *supervisor*, i quali danno a disposizione l'astrazione del meccanismo del client/server e di supervisione dei threads del processo in vita. Essendo fortemente orientato ai threads e funzionale per poter utilizzare dei timer siamo dovuti ricorrere ad un particolare meccanismo: una funzione messa a disposizione da erlang, chiamata *send_after*, che ci permette di inviare un messaggio ad un modulo passato come parametro dopo un intervallo di tempo specificabile, in questo modo siamo riusciti ad evitare il blocco dell'esecuzione di un processo. I moduli sviluppati più interessanti saranno quindi descritti in dettaglio nei paragrafi successivi.

4.2 Utilizzo dei "Supervisors"

Come già accennato, il meccanismo dei supervisor permette di gestire tutti i threads che verranno creati per i vari processi del nodo. Come si può notare in figura 3.1, abbiamo due supervisor: uno principale che rappresenta il vero e proprio nodo ed un secondo per mantenere la vitalità di tutti i processi interni legati alla comunicazione (dal modulo "rules_worker" al più semplice "HB_out").

Il primo supervisor, generato dal modulo *"supervisor_nodo"*, provvede a sopperire alla necessità di mantenere in vita tutti quei dati essenziali al funzionamento del nodo. Al suo interno infatti verranno istanziate le tabelle "ets" dove vengono salvate tutte le informazioni relative alle funzionalità del nodo (lo stato dell'albero, il clock, le regole,...), verrà istanziato il thread designato alla comunicazione con l'ambiente, e per finire anche il thread che si occuperà di manipolare in maniera *sicura* i parametri del nodo. Ultimo, ma non per importanza, verrà generato come figlio del supervisor generale del nodo, il supervisor dei "workers", che si occuperà di mantenere in vita i thread per l'heartbeat e per chi interpreterà le regole.

Codice d'esempio si trova nell'appendice A.

4.3 Heartbeat (module: HB_in)

Come descritto precedentemente, nel modulo *HB_in* possiamo trovare tutti i meccanismi legati alla gestione della rete e delle sue funzionalità come tale. Nello specifico, questo si occupa solamente di controllare la "vitalità" di un nodo e di creare un albero di copertura della rete stabile, con tutti i suoi protocolli annessi.

Heartbeat - consistenza della rete

La prima funzionalità (appunto, garantire la consistenza della rete), serve a garantire che se un nodo a me vicino si spenga o diventi non più raggiungibile, allora non devo più considerarlo come mio vicino. La consistenza della rete avviene tramite alcune variabili di stato interne al modulo, identificabili dalla tupla:

$$< neighb\_clocks, neighb\_state >$$

Il primo valore della tupla identifica una mappa di tipo $< id, clock >$, e mi serve nel momento in cui ho un aggiornamento del clock interno dovuto al ricongiungimento di una rete partizionata: per cercar di garantire la consistenza delle wave di regole che si propagheranno nella rete, dovrò cercare di allineare tutti i componenti della rete al medesimo clock. Il secondo valore della tupla identifica una mappa del tipo $< id, stato >$: per controllare la vitalità di un nodo devo infatti verificarne la sua presenza attraverso il meccanismo di "echo" descritto nel paragrafo 3.2. Alla creazione del nodo suppongo che tutti i miei vicini siano "vivi", e sono tali fino a prova contraria (ovvero quando non rispondono più agli echo). Gli *stati* assumibili dai vicini in questa mappa quindi saranno:

$$stato(\eta : neightboard) = \begin{cases} alive, & \text{se } \eta \text{ ha risposto all'ultimo echo} \\ maybe\_death, & \text{se } \eta \text{ non ha risposto all'ultimo echo} \\ death, & \text{se } \eta \text{ non ha risposto agli ultimi due echo} \end{cases}$$

con η vicino di ogni nodo.
Si rimanda all'appendice B per ulteriori esemplificazioni.

Un vicino si dichiara *maybe_death* dopo $\tau = 5s$, e *death* dopo $2 * \tau$.

Heartbeat - spanning tree

Come abbiamo già accennato in precedenza, il protocollo di spanning tree prende spunto dal più famoso *spanning tree protocol* adottato dai bridge.

Questo protocollo basa il suo funzionamento sull'interscambio di tuple:

1. $< tree\_state, < root, len, nh >>$: questa tupla indica lo stato dell'albero per il nodo che la propaga. Essa infatti contiene chi è la radice per quel nodo, quanto essa è distante e chi è che manda fuori il pacchetto;
2. $< tree\_ack, id >$: questo pacchetto avvisa il nodo vicino che verrà usato per "instradare" pacchetti nella rete;
3. $< tree\_rm\_rp, id >$; avvisa la mia vecchia *root_port* che non la utilizzo più per instradare pacchetti.

Per ulteriori delucidazioni su come vengono intraprese le scelte si rimanda al paragrafo 3.2.

La necessità di avvisare i nodi vicini che li usiamo come root port o meno nasce dalla particolare situazione in cui ci troviamo: poiché non abbiamo un vero e proprio ambiente dove un nodo non designato al raggiungimento della radice comunica con tutti i nodi di quella zona, siamo costretti ad avvisare se un nodo è una root_port o meno. Questo passaggio complica la rete, in quanto crea un "overhead" di messaggi normalmente non contemplato dal protocollo originale.

Ulteriormente, poichè tutti i nodi non possono accorgersi in "tempo reale" della situazione delle loro "porte" (dato che sono tutte connessioni punto-punto), si richiede l'utilizzo di ulteriori clock per la correttezza dell'albero: uno per i timeout della vitalità della radice ed uno per evitare di utilizzare immediatamente una radice che si ipotizzava fosse morta. L'utilizzo di questi clock creano altri messaggi di saturazione della "rete emulata", desincronizzano i vari nodi, e peggiorandone le performance, imponendo necessariamente una potenza di calcolo più grande in quanto non sia una rete fisica.

Nel nostro specifico caso, viene fissato $\bar{\tau} = 2s$, come timeout $2 * \bar{\tau}$, come timeout per accettare di nuovo la vecchia radice $3 * \bar{\tau}$.

Heartbeat - comunicazioni in uscita

Per le comunicazioni in uscita, come possiamo notare da alcune linee di codice dell'appendice B, interpelliamo il modulo *HB_out*: questo modulo gestisce le comunicazioni in uscita, compresi gli errori di comunicazione. Parametri richiesti per la comunicazione verso l'esterno sono:

- *pacchetto*: il messaggio già confezionato da inviare
- *vicini*: una lista iterabile di vicini a cui mandarlo (per precisione, al loro hearbeart).

Questo thread viene direttamente creato dal modulo di *HB_in*, ed appena esaurisce la lista cessa la sua esistenza.

Si rimanda al file in `src/HB_out.erl` per vedere in dettaglio il suo funzionamento.

4.4 Comunicazione coi vicini (module: comm_IN)

La comunicazione con i vicini delle regole da eseguire avviene tramite l'utilizzo dei moduli *comm_IN* e *comm_OUT*.

comm_IN : questo modulo si trova perennemente in uno stato di ricezione, in modo tale da processare in continuazione tutti i messaggi inviati dai vicini. In particolare il modulo in questione gestisce l'algoritmo di *flooding*; per fare ciò genera al suo avvio una tabella (*ets*) per il salvataggio ed il controllo dei messaggi. Nel momento in cui arriva un messaggio di *flood* questo viene controllato, tramite la tabella, per decidere se sia un nuovo messaggio oppure uno già visto in precedenza: nel secondo caso il messaggio viene ignorato, altrimenti viene processato tramite il modulo apposito (*rules_worker*) e poi inviato ai vicini, escludendo chi l'aveva inviato.

comm_OUT : questo modulo è strutturato quasi identicamente a quello di *HB_OUT*, viene infatti utilizzato da *comm_IN* per inviare messaggi ai vicini gestendo possibili errori di comunicazione e alla conclusione cessa di esistere.

4.5 Memoria dei parametri (module: state_server)

Ogni nodo, come esplicitato nei requisiti funzionali (paragrado 2.1), necessita d'essere programmabile e d'avere un insieme di variabili di stato proprie. Oltre a questo, abbiamo necessità di conoscere e di poter interpretare delle regole definite all'avvio del nodo. Inoltre, poiché abbiamo a che fare con un ambiente virtualizzato, necessitiamo di sapere quali sono i vicini con cui collabora.

Tutte queste informazioni, come già esplicitato nel capitolo precedente e nei paragrafi precedenti, sono salvate all'interno di uno "state server", il quale opera su delle tabelle persistenti (a meno di un riavvio totale di un nodo), create e fornite dal supervisor del nodo, al fine di non aver problematiche in caso di fallimenti.

Lo state server è incentrato in un'ottica client/server: per questo motivo il suo behavior module è *gen_server*.

Lo state server si occupa quindi di accedere in maniera sicura e non concorrenziale alle variabili a suo carico, al fine di non generare situazioni di deadlock o starvation.

Lo stato dello *state_server* sarà quindi composto dalla tupla

$$< vars\_table, rules\_table, neighb\_table, node\_params\_table >$$

dove ogni campo equivale a:

- *vars_table*: tabella delle variabili;
- *rules_table*: tabella delle regole;
- *nb_table*: tabella dello stato dei vicini;
- *node_params_table*: tabella dei parametri interni al nodo.

I valori associati alla terza componente sono strettamente correlati a quanto affrontato nel paragrafo 4.3: al suo interno troveremo delle tuple con i valori legati allo spanning tree per cui sapremo se inoltrare o meno un messaggio a quel nodo. L'inoltro di un messaggio verso un nodo avviene solo nel caso in cui il vicino sia in stato *active* o *root_port*.

I valori associati al quarto membro della tupla sono invece legati ad informazioni chiave per identificare il nodo di appartenenza. Al suo interno possiamo trovare alcune voci, quali ad esempio: il suo identificativo univoco, che tipo di nodo è per la rete, quante *"rules wave"* ha visto, il suo clock interno e chi è la sua radice. Queste informazioni sono identificative ed univoche per un nodo, e sono il suo stato interno.

Il significato puntuale della prima e della seconda componente di questa tupla verranno affrontati in miglior dettaglio nel paragrafo 4.6.

Poiché lo state server è l'unico a poter manipolare le variabili del nodo, al suo interno avrà un piccolo "interprete" per manipolare e controllare le variabili. Questo avviene in maniera ricorsiva mediante l'utilizzo di una funzione ricorsiva chiamata "check_condition" (si rimanda all'appendice C per ulteriori delucidazioni).

4.6 Esecutore dei comandi (module: rules_worker)

Questo modulo, come accennato nei paragrafi precedenti, si occupa dello smaltimento delle azioni ricevute dai nodi vicini (tramite il modulo *communication in*) e di gestire la logica di esecuzione delle varie regole che possono venir attivate.

Il modulo fornisce una funzione apposita (*exec_action*) per permettere ad un qualsiasi altro modulo del nodo (in questo caso solamente *communication in* e *communication ambiente*) di inviare ad esso un'azione da eseguire. Una volta ricevuta un'azione, viene eseguito un controllo del valore del clock legato ad essa: se il clock ricevuto fosse maggiore del valore aspettatosi dal nodo (cioè il valore di clock salvato + 1), l'azione viene inserita in una lista di attesa (*on_timer_hold*, presente nello stato interno del modulo) nella quale rimarrà finché non vengono eseguite le azioni mancanti, che quindi porterebbero il valore atteso dal nodo a quello in questione, oppure finché non viene a scadere un timer apposito istanziato nel momento in cui l'azione veniva aggiunta alla lista. Nel momento in cui un'azione ricevuta possiede un valore di clock minore o uguale al valore atteso dal nodo, oppure è stata appena estratta dalla

lista *on_timer_hold*, viene inserita in una coda di esecuzione (*action_queue*) aggiornando il valore di clock del nodo in accordo; questa coda a questo punto verrà smaltita da una funzione apposita del nodo, il quale quindi potrà continuare ad eseguire i calcoli necessari senza bloccare la possibilità di ricevere ulteriori azioni dai nodi vicini.

Per processare le varie azioni inserite nella coda viene utilizzato un messaggio apposito internamente al modulo (*handle_next_action*): appena prima di inserire una nuova azione nella coda viene controllato se questa sia vuota, in caso affermativo il modulo invia a se stesso il messaggio in questione, altrimenti no. Questo permette di avere sempre solamente un messaggio di questo tipo nella coda di ricezione del modulo e garantisce che la coda delle azioni contenga almeno un elemento quando andiamo a processare questo messaggio. Nel momento in cui finisco di processare un'azione estratta dalla testa della coda, vado nuovamente a controllare se questa sia vuota per decidere se inviarmi ulteriormente il messaggio oppure no.

Esecuzione di un'azione

Finché la coda *action_queue* non è vuota, vado ad estrarre la prossima azione da processare dalla sua testa. A questo punto viene utilizzata una funzione apposita messa a disposizione dallo *state_server* la quale mi permette di inviare l'azione da eseguire e ricevere in ritorno una lista di regole che si sono attivate. Questa funzione va a controllare se le variabili presenti nella guardia e nelle vere e proprie istruzioni dell'azione siano eseguibili, cioé controlla inizialmente che i nomi delle variabili siano presenti e poi che il valore di ultima modifica salvato per ogni variabile non sia maggiore o uguale al valore di clock legato all'azione in questione. In caso di risultato positivo viene controllato il valore della guardia e se questa è soddisfatta vengono eseguite le varie istruzioni di modifica delle variabili. Una volta eseguita l'azione lo *state_server* cerca quindi tutte le regole che vengono attivate dalle modificihe appena eseguite (cioè se tutti i trigger siano stati attivati) e le restituisce sotto forma di lista.

A questo punto le regole che sono state attivate devono essere processate prima di eseguire una nuova azione, perciò vengono inserite in una coda prioritaria *priority_queue* che verrà sempre elaborata a step come per l'*action_queue* ma avendo precedenza su quest'ultima.

Infine viene eseguito un controllo della lista *on_timer_hold* per estrarre tutte quelle azioni che stavano aspettando l'esecuzione di un'azione con valore di clock uguale a quello appena processato, per poi venir inserite nell'*action_queue*.

Propagazione delle regole

Ad ogni passo processo la regola presente in testa alla coda *priority_queue* finché non è vuota. Una volta estratta la regola viene eseguito un controllo sulla condizione presente tramite l'utilizzo di una funzione apposita dello *state_server*: questa inanzitutto controlla se le variabili presenti esistano realmente e se non siano già state modificate da un clock di valore maggiore, infine

esegue il controllo vero e prorio della condizione, restituendo se essa è soddisfatta oppure no. In questo caso, diversamente dalle azioni esterne, il clock della regola può andare a modificare variabili con valore di clock di ultima modifica uguale, questo perché il caso in cui più regole vadano ad utilizzare la medesima variabile allo stesso step (per modifiche o condizioni) deve essere una situazione possibile.

A questo punto la regola viene processata in base al suo tipo:

Locale l'azione associata viene trattata come fosse una normale azione da eseguire: viene quindi chiamata l'apposita funzione dello *state_server*, il quale restituisce la lista di regole attivate che a loro volta verranno inserite nella *priority_queue*. Queste nuove regole devono essere inserite in testa alla coda, in questo modo elaboro prima gli effetti causati dalla regola appena processata, piuttosto che le altre regole già presenti.

Globale in questo caso l'azione associata non deve essere eseguita internamente dal nodo in questione, ma deve essere propagata ai vicini creando un nuovo *flooding* utilizzando il valore di clock adeguato.

Transazione in questo caso la regola genera una transazione, perciò inizio inviando il messaggio di richiesta. A questo punto lo *rules_worker* deve controllare se è anch'esso un partecipante della transazione, cioè la guardia dell'azione associata è soddisfatta: in caso affermativo viene bloccato il processamento delle due code per dedicarsi totalmente ad essa; se invece non è un partecipante, può procedere con il normale processamente, gestendo però in simultanea lo svolgimento della transazione. La gestione dello stato del modulo viene eseguita tramite l'utilizzo di una variabile di stato che indica in quale momento della transazione mi trovo:

- $\{none\}$: non mi trovo in una transazione.
- $\{started\_transaction, Transact\_clock, Partecipants, Action\}$: ho iniziato una nuova transazione, quindi aspetto le risposte dei nodi che vogliono partecipare aggiungendoli alla lista $Partecipants$ finché non scade il timer associato.
- $\{waiting\_commit, Id\_gen, Transact\_clock, Action\}$: ho risposto ad una richiesta di transazione dicendo di voler partecipare, aspetto il messaggio di commit o la fine del timer apposito.

Nel caso in cui la transazione vada a buon fine, l'azione viene eseguita utilizzando la medesima sequenza di comandi usata nei casi precedenti: utilizzo la funzione fornita dallo *state_server*, ricevo le regole che si sono attivate e le inserisco in *priority_queue* ed infine aggiorno *on_timer_hold*.

4.7 Ambiente e la comunicazione con i nodi

funzionalità dell'ambiente come:

```
%% client functions
-export([ignore_neighb/2, kill_node/1, ...(?)]).

-record(ambiente_state, {
  graph,     %ets del grafo
  id_spwn,
  comm_spwn,
  id_sup_node
}).
```

come comunico con il nodo?
comm_ambiente aiutaci tu!

```
%% client functions
-export([add_neighb/2, ignore_neighb/2]).

-record(comm_ambiente_state, {
  name,
  server_name,
  rules_worker_name,
  hb_name
}).

init([Name, Server_name, Rules_worker_name, HB_name, Id]) ->
  ambiente ! {nodo_avviato, Name, {Id, HB_name}},  % IMPORTANTE: l'ambiente deve essere registrato sotto il
  self() ! {start_discovery},
  {ok, #comm_ambiente_state{name = Name, server_name = Server_name, rules_worker_name = Rules_worker_name,

handle_cast({add_neighb, Node = {_Node_ID, _Node_HB_name}}, State = #comm_ambiente_state{server_name = Ser
  state_server:add_neighb(Server, Node),
  {noreply, State};
handle_cast({ignore_neighb, Neighb}, State = #comm_ambiente_state{server_name = Server}) ->
  state_server:ignore_neighb(Server, Neighb),
  {noreply, State};
handle_cast(_Request, State = #comm_ambiente_state{}) ->
  {noreply, State}.

handle_info({start_discovery}, State = #comm_ambiente_state{server_name = SN, hb_name = HBN}) ->
  % richiedo all'ambiente di inviarmi la lista dei vicini
  receive
    {discover_neighbs, _Neighbs_list = []} ->
      ok;
    {discover_neighbs, Neighbs_list = [_ | _]} ->
      io:format("Ricevuta lista di vicini: ~p.~n", [Neighbs_list]),
      state_server:add_neighbs(SN, Neighbs_list)
  end,
```

```
34    HBN ! {neighb_ready},
35    {noreply, State};
```

Appendice A

Supervisors

```
init({Id, Tipo}) ->
  State_tables = create_table(Id, Tipo),
  Server_name = list_to_atom(atom_to_list(Id) ++ "_server"),
  Rules_worker_name = list_to_atom(atom_to_list(Id) ++ "_rules_worker"),
  Comm_ambiente_name = list_to_atom(atom_to_list(Id) ++ "_comm_ambiente"),
  HB_name = list_to_atom(atom_to_list(Id) ++ "_heartbeat_in"),
  MaxRestart = 1,
  MaxRestartPeriod = 5,
  SupFlags = #{strategy => one_for_one,
               intensity => MaxRestart, period => MaxRestartPeriod},
  ChildSpecs = [
    #{id => state_server,
      start => {state_server, start_link, [Server_name, Id, State_tables]},
      restart => permanent,
      shutdown => infinity,
      type => worker,
      modules => [state_server]},
    #{id => sup_workers,
      start => {supervisor_workers, start_link,
       [Id, Server_name, Rules_worker_name, HB_name]},
      restart => permanent,
      shutdown => infinity,
      type => supervisor,
      modules => [supervisor_workers]},
    #{id => comm_ambiente,
      start => {comm_ambiente, start_link,
       [Comm_ambiente_name, Server_name, Rules_worker_name, HB_name, Id]},
      restart => permanent,
      shutdown => infinity,
      type => worker,
      modules => [comm_ambiente]}
  ],
  {ok, {SupFlags, ChildSpecs}}.
```

Appendice B

Heartbeat

```
-record(hb_state, {
  id,                %il mio id
  hb_name,           %il mio hearbeat
  server_name,       %il mio state server
  neighb_clocks,     %mappa dei vicini
  neighb_state,      %mappa dello stato dei vicini
  i_am_root,         %se sono la root true
  is_root_alive,     %se la radice è viva è true
  oldroot            %se la radice è morta per un po non posso riaccettarla
}).

start_link(Id, Server_name, HB_name) ->
  Pid = spawn_link(?MODULE, init, [Id, Server_name, HB_name]),
  {ok, Pid}.

init(Id, Server_name, HB_name) ->
  register(HB_name, self()),
  State = #hb_state{id = Id, hb_name = HB_name, server_name = Server_name},
  {ok, Clock} = state_server:get_clock(Server_name),
  case Clock of
    -1 ->
      receive
        {neighb_ready} -> % aspetto che il mio comm_ambiente abbia aggiornato la lista dei vicini
          ok
      end;
    _ -> ok
  end,
  {ok, Neighbs} = state_server:get_neighb_hb(Server_name),
  case {Clock, Neighbs} of
    {-1, []} -> % siamo gli unici nella rete
      state_server:update_clock(Server_name, 0),
      Neighbs_clocks = maps:new();
    {-1, _} -> %non ho ancora controllato i vicini
      Neighbs_clocks = enter_network(Neighbs, State);
    _ -> %il supervisor mi ha resuscitato
      Neighbs_clocks = maps:from_list([{Node, -1} || Node <- Neighbs])
  end,
  Neighbs_state = maps:from_list([{Key, alive} || Key <- maps:keys(Neighbs_clocks)]),
  % fingo la fine di un timer per dare inizio al protocollo, essendo is_root_alive inizializzato a false
  self() ! {start_tree},    %parto con l'albero
  self() ! {start_echo},    %parto con gli echo
```

```
42    listen(State#hb_state{neighb_clocks = Neighbs_clocks,
43                          neighb_state = Neighbs_state,
44                          i_am_root = false,
45                          is_root_alive = false,
46                          oldroot = undefined}).
47
48  [...]
49
50  enter_network(Neighbs, _State = #hb_state{id = Id, hb_name = HB_name, server_name = Server_name}) ->
51    spawn(hb_OUT, init, [Server_name, {add_new_nd, Id, HB_name}, Neighbs]), % invia un messaggio ad ogni vici
52
53    Neighbs_clocks = maps:from_list([{Node, -1} || Node <- Neighbs]),  % crea una mappa per il salvataggio de
54    New_neighbs_clocks = wait_for_all_neighbs(Neighbs_clocks, Server_name),
55    check_clock_values(New_neighbs_clocks, HB_name, Server_name),
56    New_neighbs_clocks.
57
58  [...]
59
60      {add_new_nd, Id_sender, Id_hb_sender} ->
61  %%      io:format("~p: Ricevuta richiesta connessione alla rete di ~p.~n", [Id, Id_sender]),
62        state_server:add_neighb(Server_name, {Id_sender, Id_hb_sender}),
63        {ok, Clock} = state_server:get_clock(Server_name),
64        try
65          Id_hb_sender ! {add_new_nb, HB_name, Clock}
66        catch
67          _:_ -> ok
68        end,
69  % aggiorno le due mappe usate presenti nello stato
70        New_NC = maps:put(Id_hb_sender, Clock, NC),
71        New_NS = maps:put(Id_hb_sender, alive, NS),
72        listen(State#hb_state{neighb_clocks = New_NC, neighb_state = New_NS});
73
74  [...]
75
76      {start_tree} ->
77        io:format("~p: Sono entrato in start_tree.~n", [Id]),
78        {ok, {Id_root, Dist, _ID_RP}} = state_server:get_tree_state(Server_name),
79        {ok, Neighbs_hb} = state_server:get_neighb_hb(Server_name),
80        spawn(hb_OUT, init, [Server_name, {tree_state, HB_name, {Id_root, Dist, Id}}, Neighbs_hb]),
81        if
82          Id_root == Id ->
83            New_Im_root = true,
84            self() ! {tree_root_keep_alive_timer_ended};
85          true ->
86            New_Im_root = false
87        end,
88        listen(State#hb_state{i_am_root = New_Im_root, is_root_alive = false});
89  [...]
```

```
90
91      {tree_state, HB_sender, {Id_root, Dist, Id_sender}} -> % messaggio di aggiornamento della radice dell'albero
92        io:format("~p: Ricevuto tree_state: ~p.~n", [Id, {Id_root, Dist, Id_sender}]),
93        {ok, {Saved_root, Saved_dist, Saved_RP}} = state_server:get_tree_state(Server_name),
94        {ok, Neighbs_map} = state_server:get_neighb_map(Server_name), % mappa con elementi {Id_nodo -> HB_nodo}
95        Neighbs_hb = maps:values(Neighbs_map),
96        if
97          Saved_root < Id_root ->
98            spawn(hb_OUT, init, [Server_name, {tree_state, HB_name, {Saved_root, Saved_dist, Id}}, [HB_sender]]),
99            New_Im_root = Im_root,
100           New_is_alive = Root_alive;
101         (Saved_root == Id_root) andalso (Saved_dist + 1 < Dist) ->
102           spawn(hb_OUT, init, [Server_name, {tree_state, HB_name, {Saved_root, Saved_dist, Id}}, [HB_sender]]),
103           New_Im_root = Im_root,
104           New_is_alive = Root_alive;
105         (Saved_root > Id_root) andalso (Oldroot =/= Id_root) ->
106           % aggiorno lo lo stato dell'albero salvato
107           state_server:set_tree_state(Server_name, {Id_root, Dist + 1, Id_sender}),
108 % avviso la nuova route port che la uso come tale
109           spawn(hb_OUT, init, [Server_name, {tree_ack, Id}, [HB_sender]]),
110 % avviso la vecchia route port che non la uso più
111           if
112             Saved_RP == Id ->
113               ok;
114             true ->
115               spawn(hb_OUT, init, [Server_name, {tree_rm_rp, Id}, [maps:get(Saved_RP, Neighbs_map)]])
116           end,
117 % avviso i vicini che ho cambiato porta
118           spawn(hb_OUT, init, [Server_name,
119               {tree_state, HB_name, {Id_root, Dist + 1, Id}},
120               Neighbs_hb -- [HB_sender]]),
121           New_Im_root = false,
122           New_is_alive = false,
123           erlang:send_after(4000, self(), {tree_keep_alive_timer_ended, Id_root});
124         ((Saved_root == Id_root) andalso (Saved_dist > Dist + 1))
125           orelse
126           ((Saved_root == Id_root) andalso (Saved_dist == Dist + 1) andalso (Id_sender < Saved_RP)) ->
127 % aggiorno lo lo stato dell'albero salvato
128           state_server:set_tree_state(Server_name, {Id_root, Dist + 1, Id_sender}),
129 % avviso la nuova route port che la uso come tale
130           spawn(hb_OUT, init, [Server_name, {tree_ack, Id}, [HB_sender]]),
131 % avviso la vecchia route port che non la uso più
132           spawn(hb_OUT, init, [Server_name, {tree_rm_rp, Id}, [maps:get(Saved_RP, Neighbs_map)]]),
133 % avviso i vicini che ho cambiato porta
134           spawn(hb_OUT, init, [Server_name,
135               {tree_state, HB_name, {Id_root, Dist + 1, Id}},
136               Neighbs_hb -- [HB_sender]]),
137           New_Im_root = false,
```

```
138             New_is_alive = Root_alive;
139           true ->
140             New_Im_root = Im_root,
141             New_is_alive = Root_alive
142         end,
143         listen(State#hb_state{i_am_root = New_Im_root, is_root_alive = New_is_alive});
```

Appendice C

Memoria dei parametri

```
%% API
-export([start_link/3]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2, terminate/2, code_change/3]).

%% client functions
-export([exec_action/3, exec_action_from_local_rule/3, check_rule_cond/3, check_trans_guard/3]).
-export([update_clock/2, get_clock/1, get_rules/1]).
-export([get_neighb/1, get_neighb_hb/1, add_neighb/2, add_neighbs/2, rm_neighb/2, rm_neighb_with_hb/2, check_neighb/2,
-export([get_tree_state/1, reset_tree_state/1, set_tree_state/2, set_tree_active_port/2, rm_tree_active_port/2]).
-export([get_active_neighb/1, get_active_neighb_hb/1]).
-export([ignore_neighb/2, get_ignored_neighb_hb/1]).

-record(server_state, {
  id,
  vars_table,
  rules_table,
  neighb_table,
  node_params_table,
  lost_connections
}).

%%%===================================================================
%%% API
%%%===================================================================

start_link(Name, Id, State_tables) when is_atom(Name) ->
  gen_server:start_link({local, Name}, ?MODULE, [Id, State_tables], []);

[...]

check_condition(Cond, VT, PT) ->
  case Cond of      %operazioni binarie
    {} ->
      true;
    {Op, Var1, Var2} -> % operazioni di arità 2: lt, lte, gt, gte, eq, neq
      % nel caso in cui le variabili siano atomi (quindi vere e proprie variabili) devo ottenere il valore corrispondent
      % altrimenti sono dei semplici numeri e quindi li uso così come sono
      Real_var1 = if
                    is_atom(Var1) ->
```

```
42                      [{Var1, Var1_value, _Var1_clock}] = ets:lookup(VT, Var1),
43                      Var1_value;
44                    true ->
45                      Var1
46                  end,
47        Real_var2 = if
48                      is_atom(Var2) ->
49                        [{Var2, Var2_value, _Var2_clock}] = ets:lookup(VT, Var2),
50                        Var2_value;
51                      true ->
52                        Var2
53                  end,
54        % ritorna il valore corrispondente all'operazione eseguita sulle variabili
55        case Op of
56          lt ->
57            Real_var1 < Real_var2;
58          lte ->
59            Real_var1 =< Real_var2;
60          gt ->
61            Real_var1 > Real_var2;
62          gte ->
63            Real_var1 >= Real_var2;
64          eq ->
65            Real_var1 == Real_var2;
66          neq ->
67            Real_var1 =/= Real_var2;
68  %%        land ->
69  %%          check_condition(Real_var1,VT,PT) andalso check_condition(Real_var2,VT,PT);
70  %%        lor ->
71  %%          check_condition(Real_var1,VT,PT) orelse check_condition(Real_var2,VT,PT);
72          _ ->
73            io:format("State server - check_condition condizione non riconosciuta: ~p.~n", [Cond]),
74            false
75        end;
76      {OpU, Tpc} ->         %operazioni unarie
77        case OpU of
78          tpe ->
79            [{tipo, Tpc}] == ets:lookup(PT, tipo);
80          ntp ->
81            [{tipo, Tpc}] =/= ets:lookup(PT, tipo);
82  %%        lnot ->
83  %%          Real_var1 = if
84  %%                        is_atom(Var1) ->
85  %%                          [{Var1, Var1_value, _Var1_clock}] = ets:lookup(VT, Var1),
86  %%                          Var1_value;
87  %%                        true ->
88  %%                          Var1
89  %%                      end,
```

```
90  %%          check_condition(Real_var1, VT, PT);
91          _ ->
92            io:format("State server - check_condition condizione non riconosciuta: ~p.~n", [Cond]),
93            false
94        end;
95      _ ->
96        io:format("State server - check_condition condizione non riconosciuta: ~p.~n", [Cond]),
97        false
98    end.
```