Progetto di Sistemi Distribuiti

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

Version $\frac{1}{4}$, 7 maggio 2021

Indice

Indice **a**

1 Introduzione **1**
1.1 Descrizione del problema 1
1.2 Struttura dell'implementazione 1
1.3 Trasparenze 2
1.4 Algoritmi 2
1.5 Testing 2
1.6 Piano di sviluppo 3

2 Analisi **5**
2.1 Requisiti Funzionali 5
2.2 Requisiti Non Funzionali 6

3 Progetto **7**
3.1 Architettura logica 7
3.2 Protocolli ed algoritmi 9
3.3 Architettura fisica e deployment 15
3.4 Piano di sviluppo 15

4 Implementazione **17**
4.1 Software e hardware 17
4.2 Utilizzo dei "Supervisors" 17
4.3 Heartbeat (module: HB_in) 18
4.4 Server dei parametri (module: state_server) 18
4.5 Esecutore dei comandi (module: rules_worker) 18
4.6 Ambiente e la comunicazione con i nodi 18

A Supervisors **19**

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.

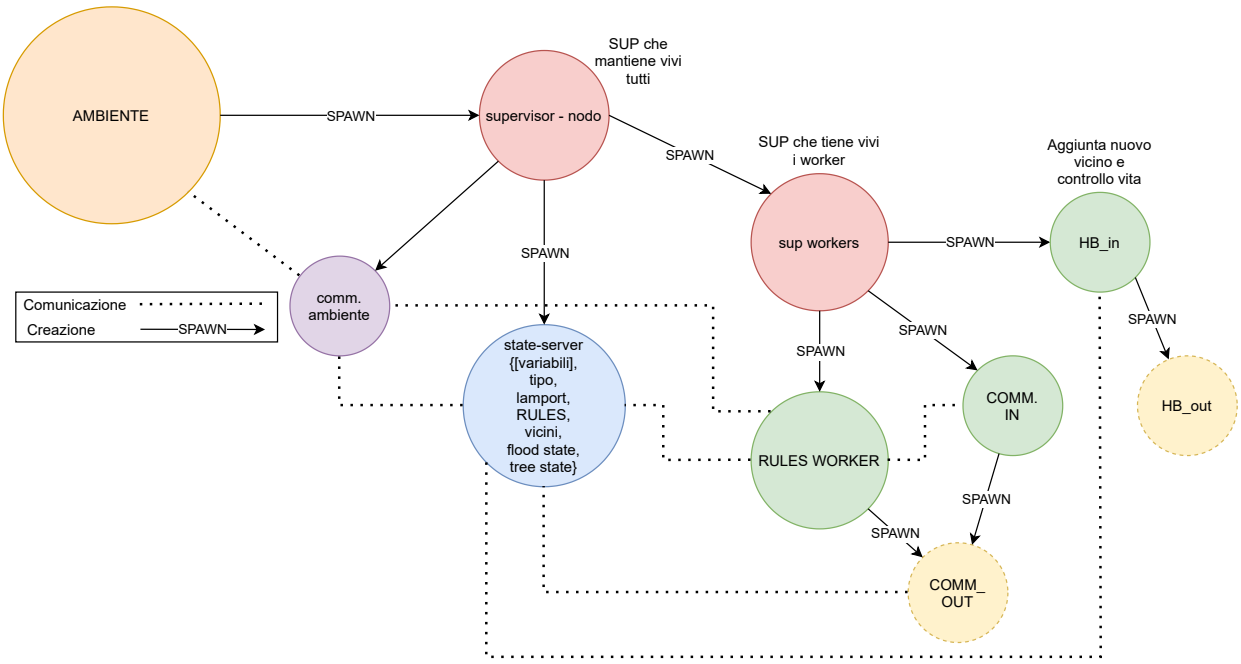

Figura 3.1: Struttura gerarchica dei moduli di un nodo e visualizzazione delle connessioni fra di essi.

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.

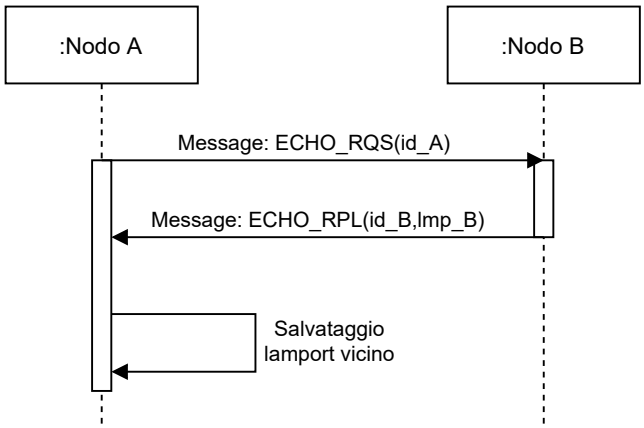

Figura 3.2: Sequence diagram dei messagi usati per il sistema di *heartbeat*.

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.

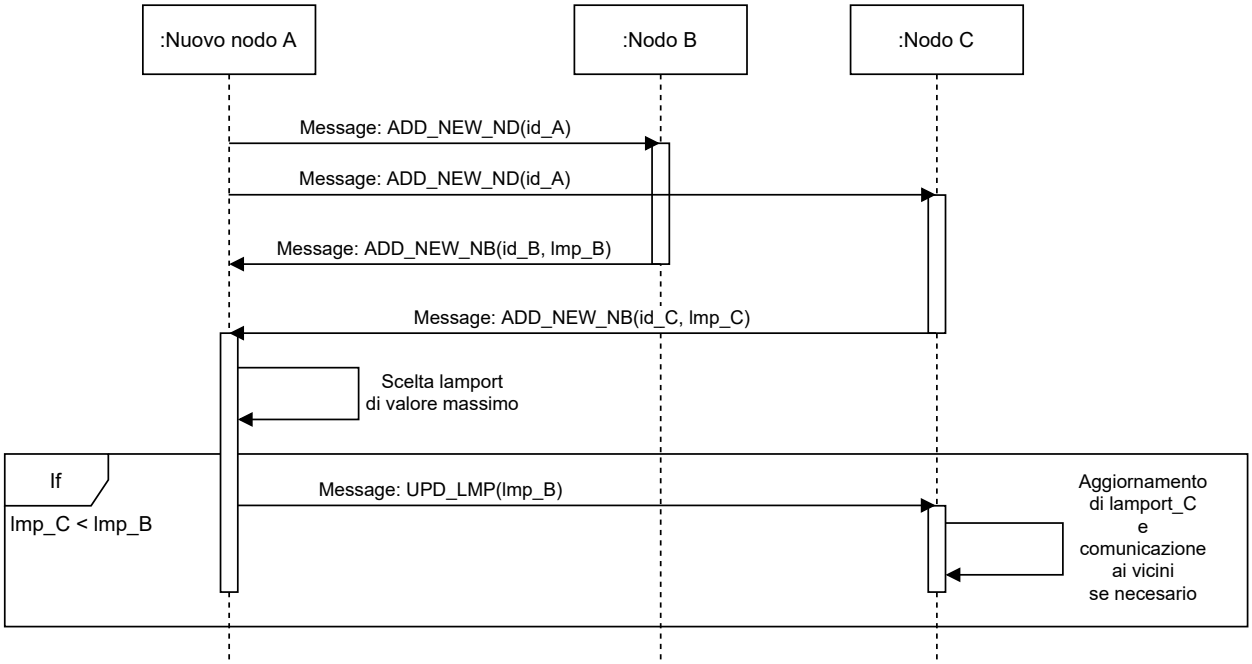

Figura 3.3: Sequence diagram dei messagi usati per laggiunta di un nuovo nodo alla rete.

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.

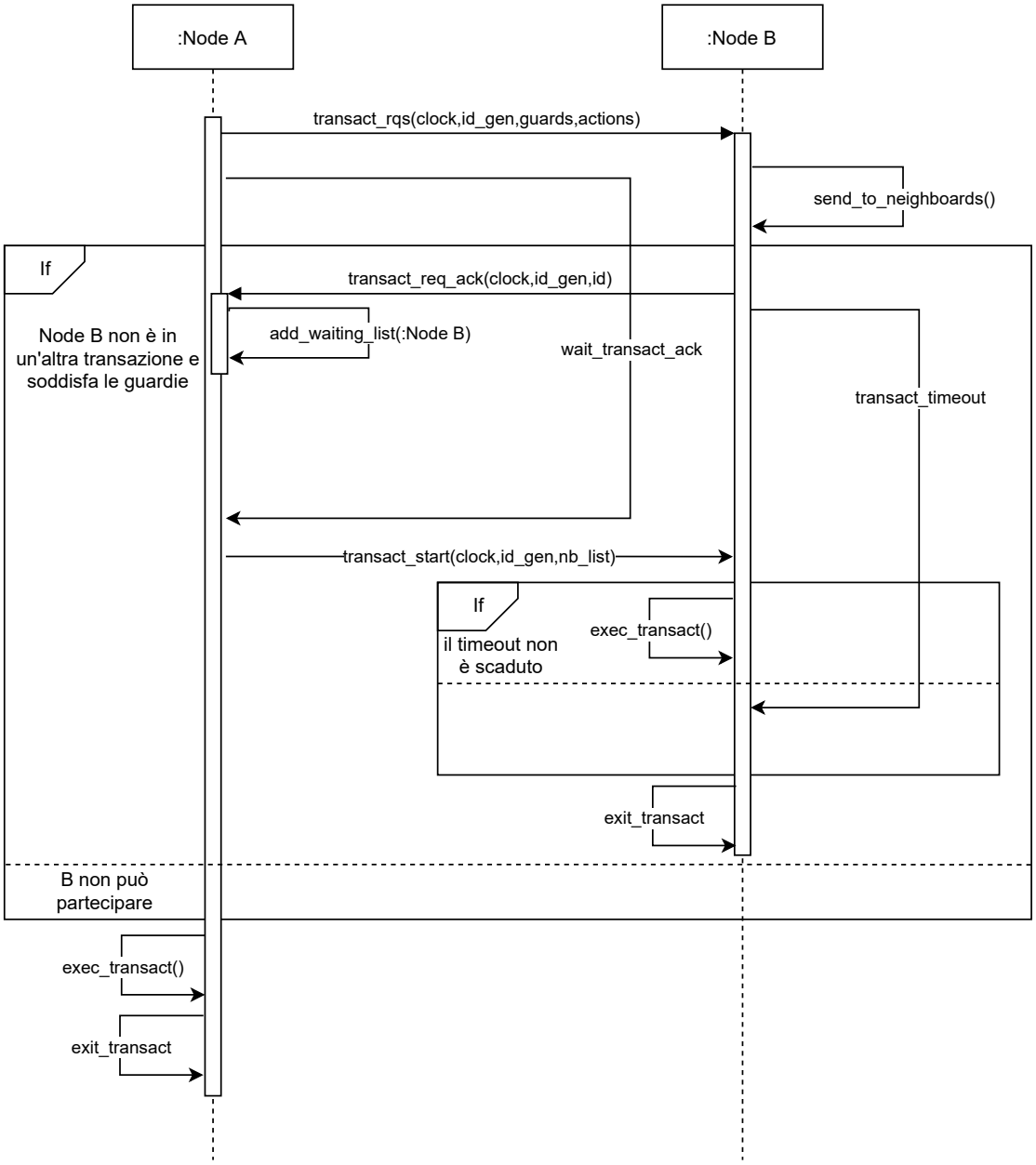

Figura 3.4: Sequence diagram del protocollo adottato per le regole di transaction.

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.

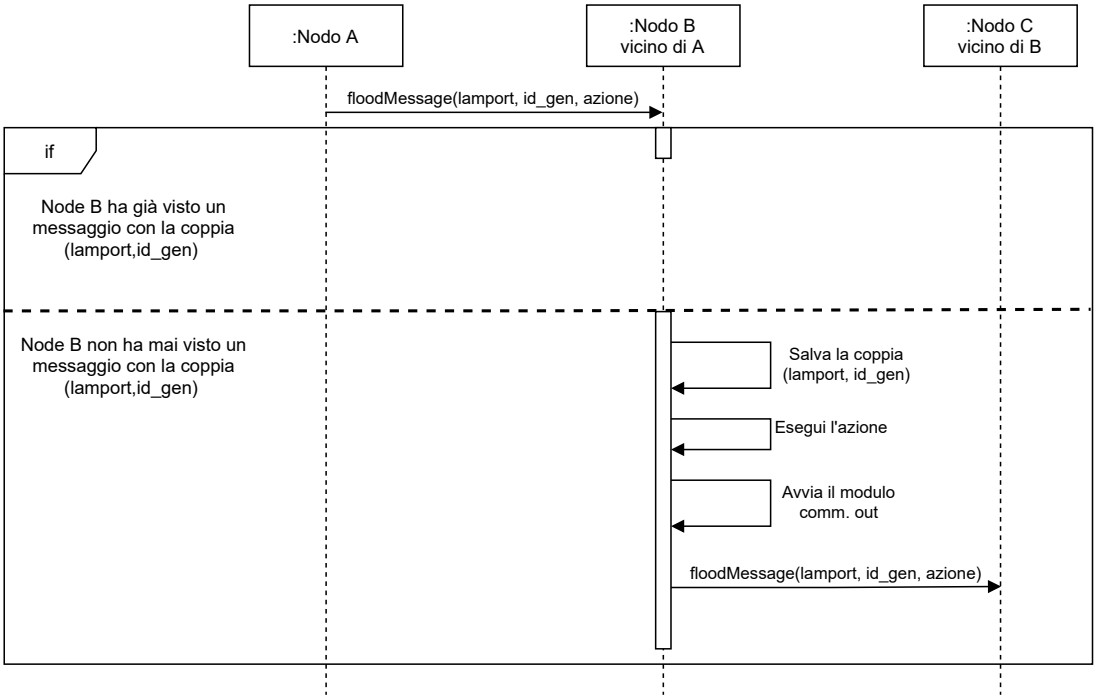

Figura 3.5: Sequence diagram dei messaggi usati per l'algoritmo di flooding.

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. 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 per mantenere la vitalità di tutti i processi interni (dal modulo di "state_server" 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 verrà istanziata la tabella "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)

4.4 Server dei parametri (module: state_server)

4.5 Esecutore dei comandi (module: rules_worker)

4.6 Ambiente e la comunicazione con i nodi

Appendice A

Supervisors

Di seguito estratto di codice per il supervisor del nodo.

```
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}}.
```