Università di Pisa

FACOLTÀ DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica

Realizzazione e simulazione di un algoritmo per il rilevamento delle intrusioni in sistemi robotici mobili

Candidato:
Rossi Alessio
Matricola 271578

Relatori:
Prof. Gianluca Dini
Prof. Antonio Bicchi
Ing. Adriano Fagiolini

Anno Accademico 2008–2009

Sommario

Questa tesi affronta il problema della realizzazione di un sistema di rilevamento di intrusione per sistemi robotici mobili. In particolare, si considerano scenari applicativi in cui ogni robot deve pianificare il proprio movimento, secondo un insieme di regole di cooperazione date, al fine di evitare collisioni con altri robot mobili vicini.

Per raggiungere questo scopo, i robot possono avvalersi delle misure fornite da alcuni sensori di bordo e, qualora necessario, di informazioni ricevute mediante lo scambio di un messaggio con un vicino. In questo contesto, consideriamo il potenziale rischio della presenza di uno o più robot, o intrusi, che pianificano il proprio movimento senza rispettare le regole di cooperazione, al fine di deteriorare le prestazioni del sistema o addirittura di provocare collisioni "intelligenti".

La letteratura scientifica a riguardo è estremamente recente e, in particolare, ha portato alla definizione dell'architettura di un sistema di rilevamento di intrusione completamente distribuito. In tale architettura, ogni robot monitorizza i propri vicini, per stabilire se il loro comportamento è conforme alle regole di cooperazione date, e scambia queste informazioni con gli altri robot monitoranti in modo da raggiungere un consenso. Tuttavia, il precedente lavoro non considera aspetti legati alla comunicazione, che invece devono essere accuratamente analizzati e verificati prima di una reale implementazione del sistema stesso.

Il contributo di questa tesi ha riguardato infatti la simulazione e realizzazione del sistema di rilevamento di intrusione in questione, attraverso un protocollo di comunicazione che è stato verificato mediante lo strumento Omnet++ e il suo Mobility Framework.

In questo modo si è dimostrato che il protocollo di consenso è in grado

di rilevare robot non cooperativi anche in condizioni reali di funzionamento.

Indice

1 Introduzione **1**

2 Lo stato dell'arte **3**

3 Realizzazione dell'algoritmo in Omnet++ **7**

4 Valutazione dei risultati **11**

4.1 Simulazione in ambiente reale senza perdita 11

4.2 Simulazione in ambiente reale con perdita 15

4.3 Confronto tra diverse topologie 17

5 Conclusioni **28**

A Sorgenti **29**

A.1 Reputation Manager 29

A.1.1 ReputationManager.h 29

A.1.2 ReputationManager.cc 30

A.2 Monitor 32

A.2.1 Monitor.h 32

A.2.2 Monitor.cc 33

A.3 Mobility Module 36

A.3.1 RMobility.h 36

A.4 Agent Host 39

A.4.1 AgentHost.ned 39

A.5 Network 40

A.5.1 Network.ned 40

B Installazione Omnet++ e Mobility Framework **43**
B.1 GNU/Linux Debian unstable 368 (architettura 32 bit) 43
B.2 Installazione Mobility Framework (architettura 32 bit) 43

C Struttura della directory **45**

Elenco delle figure

2.1 Topologia con 5 agenti 4
2.2 Topologia con 5 agenti e relative manovre 5

3.1 Moduli semplici e moduli complessi 7
3.2 Aree libere visibili dall'agente 3 8
3.3 Timeline 9
3.4 Schema di comunicazione 9

4.1 Topologia a 5 agenti 11
4.2 Topologia a 5 agenti: n_{min} per coprire la rete 12
4.3 Topologia a 5 agenti: 0s 13
4.4 Topologia a 5 agenti: 35s 13
4.5 Topologia a 5 agenti: 70s 13
4.6 Topologia a 5 agenti: 82s 13
4.7 Topologia a 5 agenti: 94s 14
4.8 Topologia a 5 agenti: 107s 14
4.9 Topologia a 5 agenti: 120s 14
4.10 Topologia a 5 agenti: reputazione di 2 assegnata da 3 con $p = 0$ e $n = 2$ 15
4.11 Topologia a 5 agenti: numero di messaggi ricevuti con $p = 0.5$ e $n = 2$ 16
4.12 Topologia a 5 agenti: numero di messaggi ricevuti con $p = 0.5$ e $n = 4$ 17
4.13 Topologia a 5 agenti: numero di messaggi persi con $p = 0.5$ e $n = 4$ 18

4.14 Topologia a 5 agenti: reputazione di 2 assegnata da 3 con $p = 0.5$ e $n = 4$ 18
4.15 Topologia a 9 agenti 19
4.16 Topologia a 8 agenti 20
4.17 Topologia a 7 agenti 21
4.18 Topologia a 6 agenti 22
4.19 Topologia a 5 agenti 23
4.20 Topologia a 4 agenti 24
4.21 Topologia a 3 agenti 25
4.22 Topologia a 2 agenti 26

C.1 Schema di compilazione, link tra files e esecuzione 46

Elenco delle tabelle

4.1 Parametri di configurazione: $p = 0.00$ e $n = 2$ 12
4.2 Parametri di configurazione: $p = 0.50$ e $n = 2$ 15
4.3 Parametri di configurazione: $p = 0.50$ e $n = 4$ 16
4.4 Tabella riassuntiva 27

Capitolo 1

Introduzione

Negli ultimi anni, la diffusione sempre maggiore di sistemi robotici e del principio "divide et impera" per la risoluzione dei problemi, hanno dato molto spazio all'attività di ricerca, principalmente nell'ambito della realizzazione di sistemi intelligenti distribuiti.

Questa tipologia di sistemi è caratterizzata dalla cooperazione di più agenti robotici che, in seguito ad una fase di sensing dell'ambiente circostante, iniziano uno scambio di messaggi con i propri vicini. In questo modo, le informazioni raccolte da un agente si propagano e diventano parte della conoscenza di tutti. Ogni agente è adesso in grado di analizzare l'ambiente e trarre delle conclusioni che, essendo le stesse per tutti, portano ad un consenso tra i sistemi robotici.

Questo protocollo può risultare utile e necessario in molti ambienti e situazioni, vediamone alcuni esempi. Una possibile applicazione si ha nell'ambiente industriale, dove i robot, trasportando oggetti o materiali per corridoi e scaffalature, dovranno evitarsi l'un l'altro oltre a fare attenzione a ostacoli non previsti. Un altro tipico esempio può essere l'ambiente di un "mall", dove i robot si muovono tra le corsie trasportando oggetti, fermandosi a caricarne di nuovi, evitando in ogni istante di collidere tra loro o con altri ostacoli.

Soprattutto a causa dell'interazione diretta o indiretta con le persone, essi devono garantire di operare in assoluta sicurezza, ovvero, devono essere in grado di gestire situazioni inattese, devono essere tolleranti a malfunzionamenti causati da guasti e rilevare eventuali agenti malevoli che hanno come

obiettivo l'arrecare danni agli altri.

Valutando la possibilità della presenza di un agente malevolo nell'ambiente, poiché guasto o manomesso, il software in esecuzione sui robot dovrà essere in grado di rilevarlo mediante un algoritmo distribuito di rilevamento di intrusioni.

Un software di questo tipo è stato descritto ed implementato per un migliore studio del problema in due pubblicazioni, CASE07 ed ICRA08 e ripreso come base di partenza per il presente lavoro. Il sistema di simulazione è il simulatore "MASS" (Multi-Agent System Simulator), il quale, implementa l'intero protocollo ipotizzando un canale di comunicazione ideale. In particolare, nell'attività di scambio di messaggi tra agenti robotici, si suppone di essere in presenza di un mezzo che non introduce ne' ritardi ne' perdite di informazioni.

Con il presente lavoro ci poniamo l'obiettivo di dimostrare il raggiungimento del consenso tra i sistemi cooperativi in presenza di un canale reale, ovvero, in presenza di un canale che introduce ritardi di trasmissione e perdite di messaggi.

I capitoli principali, seguiti da alcune appendici sono i seguenti:

Lo stato dell'arte , si descrive la base di partenza del presente lavoro.

Realizzazione dell'algoritmo con Omnet++ , si descrive la realizzazione dell'algoritmo.

Valutazione dei risultati in questo capitolo si effettuano le simulazioni e si analizzano i risultati ottenuti.

Capitolo 2

Lo stato dell'arte

Come precedentemente anticipato negli articoli, è presente e realizzato un protocollo di consenso in grado di essere eseguito in modo distribuito e di rilevare intrusioni o manomissioni di agenti mobili.

Facendo riferimento al simulatore MASS, il quale simula il protocollo in questione, vediamone il funzionamento più nel dettaglio.

Per semplificare l'analisi della situazione, consideriamo un'autostrada a tre corsie ed un unico senso di marcia dato che già conosciamo, intuitivamente, il set di regole secondo cui un gruppo di agenti può muoversi in questo ambiente. Un agente è un singolo sistema robotico che si muove ed interagisce con gli altri.

Ad ogni istante, un agente può scegliere, sulla base di un set di regole prefissate, una manovra da eseguire tra quelle presenti nel proprio insieme (che chiameremo R). In particolare, nel caso dell'autostrada, l'insieme delle manovre consentite è riportato di seguito:

Right: rientro da un sorpasso

Left: sorpasso

Brake: frenata

Fast: accelerare fino alla velocità massima consentita

La scelta della nuova manovra da schedulare non avviene a caso, ma sulla base di formalismi, eventi e sottoeventi, che, combinati tra loro, determinano la transizione da uno stato, inteso come stato di esecuzione di una manovra,

ad un altro. Tutti questi formalismi sono testabili su un software. Vediamo, per ciascuna manovra, quali sono gli eventi che determinano una transizione da una manovra ad un'altra. Ogni evento sarà composto da sottoeventi:

Right: non sono nella corsia più interna && la corsia interna è libera

Left: non sono nella corsia più esterna && ho un ostacolo di fronte a me && la corsia esterna è libera

Brake: ho un ostacolo di fronte a me && non posso eseguire ne' una manovra di LEFT ne' una manovra di RIGHT

Fast: non ho ostacoli di fronte a me && non sussistono le condizioni che determinano una manovra di LEFT oppure una manovra di RIGHT

Supponiamo adesso di avere la situazione riportata in figura e supponiamo inoltre che tutti gli agenti stiano funzionando correttamente secondo le regole del protocollo.

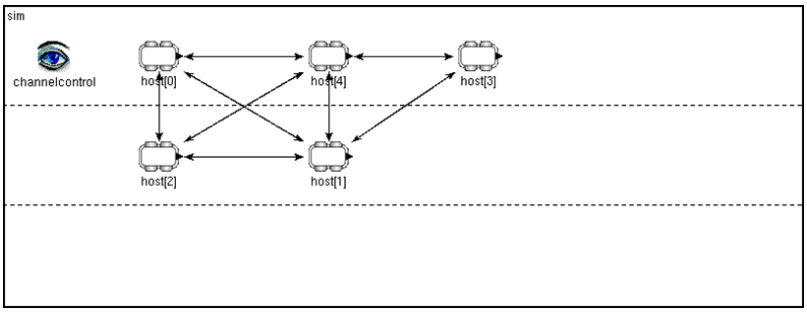


Figura 2.1: Topologia con 5 agenti

In questo particolare caso, gli agenti scheduleranno le seguenti manovre:

0: FAST

1: RIGHT

2: RIGHT

3: RIGHT

4: FAST

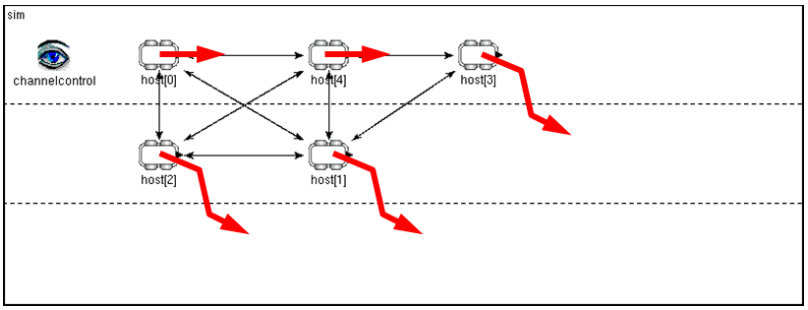


Figura 2.2: Topologia con 5 agenti e relative manovre

come possiamo vedere dalla Figura 2.2.

I sistemi robotici, oltre a muoversi secondo il proprio set di regole R, predefinito, eseguono un protocollo di consenso, mediante il quale possono monitorarsi a vicenda e quindi, sulla base delle proprie conoscenze sull'ambiente, riuscire a capire cosa li circonda e rilevare eventuali comportamenti errati da parte di altri agenti. I comportamenti errati, ovvero, comportamenti che non rispettano il set di regole R, possono essere dovuti a malfunzionamenti di sensori, di software oppure alla presenza di agenti malevoli il cui unico scopo è quello di creare confusione ed arrecare danno.

Il protocollo che andremo ad analizzare fa uso di messaggi di tipo broadcast inviati in modo sincronizzato, ipotizzando la loro immediata e corretta ricezione.

Lo scambio di messaggi avviene nell'intervallo temporale T, una o più volte a seconda della topologia della rete. Un messaggio scambiato è composto essenzialmente da due parti:

- una lista di coordinate, ognuna delle quali identifica la posizione di un altro agente nell'ambiente
- una lista di aree visibili dall'agente che rileva i vicini mediante i propri sensori

Ogni agente, dopo aver ricevuto i messaggi dei suoi vicini, li prende uno ad uno e li fonde con le informazioni che già possiede, ovvero, fonde la propria lista di posizioni e la propria lista di aree con quelle ricevute. Una volta ottenute le liste complete, ovvero, dopo essere arrivati ad avere lo stesso livello

di conoscenza su ogni agente, è quindi possibile avviare dei monitor software in grado di assegnare delle reputazioni agli altri agenti che conoscono.

Sulla base di quanto detto finora, la valutazione delle reputazioni avviene periodicamente su ogni agente. Inoltre possiamo dedurre che l'aspetto della comunicazione non è stato trattato in modo approfondito, ovvero, è stato semplicemente considerato ideale.

L'obiettivo da raggiungere in questo lavoro consiste nella valutazione delle prestazioni del protocollo di consenso in caso di canale non più ideale, ma reale e simulato da Omnet++ con Mobility Framework.

Capitolo 3

Realizzazione dell'algoritmo in Omnet++

Omnet++ abbinato al Mobility framework, fornendo un'architettura di sviluppo completamente diversa rispetto a quella presente nel simulatore MASS, ha reso necessario il bisogno di riprogrammare alcune parti del programma di simulazione.

L'esigenza di riprogrammazione si è resa necessaria anche dal punto di vista logico. Mentre il vecchio simulatore MASS è costituito da classi c++ che implementano le diverse funzionalità, Omnet++ e Mobility fw, mediante il loro framework, ci forniscono un livello di astrazione logica maggiore offrendo una vista a moduli, ovvero, trattando determinate classi c++ come componenti modulari che vanno a comporre l'architettura di un agente.

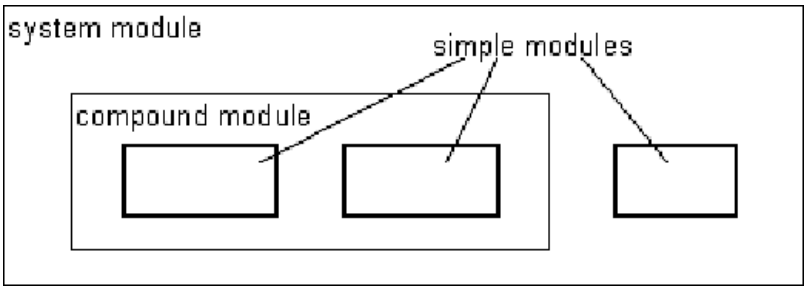


Figura 3.1: Moduli semplici e moduli complessi

Mediante un ulteriore file di descrizione di architettura di tipo .ned, canale, ambiente ed agenti vengono legati insieme al fine di definire e costruire l'intera network di simulazione.

Inizialmente è stato necessario ridefinire il modulo di movimento, ovvero, dato il set di regole R, sono state riscritti gli eventi e le condizioni di schedulazione delle manovre, ma solamente a livello implementativo. Il cambiamento degli eventi relativi al movimento, intesi in senso strettamente tecnico ed implementativo, uniti alla separazione logica dei componenti degli agenti, ha portato alla necessità di reimplementare anche il sistema di valutazione e gestione delle reputazioni.

Con l'approccio a moduli isolati che ricreano un ambiente reale con sistemi robotici reali, non esiste un'entità esterna che possa conoscere tutto ciò che accade in ogni luogo ed in ogni istante e che quindi possa valutare statistiche e funzinamento del protocollo. Da qui è nata l'esigenza di riprogrammare buona parte dei moduli.

Tutto questo ha portato anche alla necessità di lasciare un po' da parte l'approccio puramente teorico che era stato alla base del MASS e di riuscire a trovare un compromesso tra teoria e realizzabilità. In particolare, per quanto rigurda questo punto, facciamo riferimento alle aree, chiamate anche ipotesi, che nel simulatore MASS erano composte da aree irregolari dell'ambiente e che adesso sono state discretizzate.

Con queste modifiche è adesso possibile, per ogni agente, creare un messaggio analogo a quello utilizzato precedentemente. La differenza rispetto al passato, sta nel fatto che le ipotesi saranno costituite da una lista di coordinate che identificano un'area chiamata più comodamente cella.

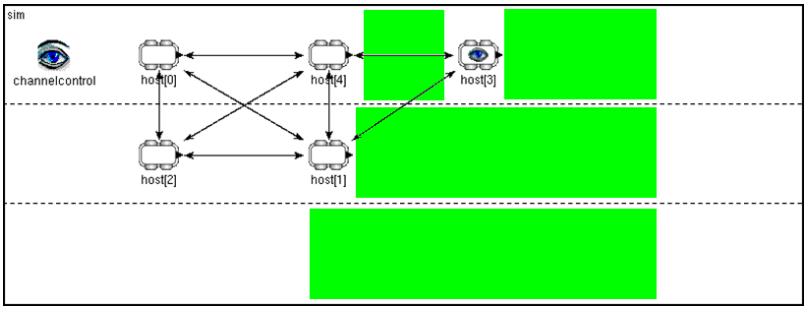


Figura 3.2: Aree libere visibili dall'agente 3

Nella corrente implementazione dell'algoritmo di intrusion detection si utilizzano i moduli di comunicazione del protocollo CSMA p-persistent forniti dal Mobility framework. In particolare, l'idea del CSMA p-persistent è

"listen before talk", questo significa che una stazione, prima di tentare una trasmissione, ascolterà il canale alla ricerca di un segnale. Se il canale è rilevato occupato, allora sarà calcolato un intervallo di tempo casuale prima di ascoltare nuovamente. Solamente quando il canale sarà rilevato in stato "idle", il messaggio sarà inviato.

Lo scambio di messaggi avviene con protocollo CSMA p-persistent all'interno di uno slot temporale T con frequenza calcolata in base al numero n di messaggi che vogliamo inviare. Da come possiamo vedere in Figura 3.3, ciascuno di questi intervalli è suddiviso in sottointervalli di ampiezza t_0 calcolati come $f(T, n)$.

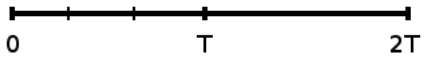


Figura 3.3: Timeline

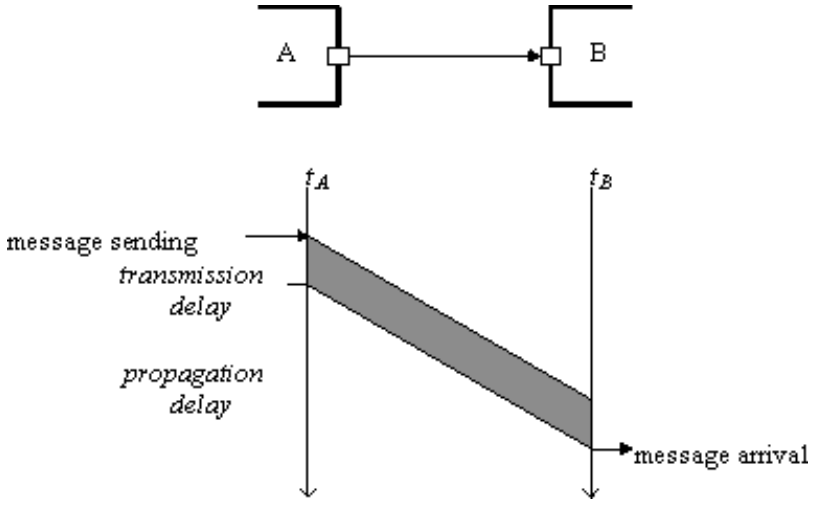


Figura 3.4: Schema di comunicazione

Ogni agente, periodicamente ed in modo asincrono rispetto agli altri, all'interno di un certo slot temporale di durata pari a T secondi chiamato passo di consenso, effettuerà più invii del proprio messaggio, ogni volta aggiornato con le informazioni che ha ricevuto dai propri vicini. Ogni T secondi, quindi, andrà in esecuzione su ogni agente, l'algoritmo di consenso con lo scopo

di rilevare eventuali comportamenti non corretti da parte degli altri sistemi robotici che sta monitorando ed assegnare a loro una reputazione coerente.

Sfruttando Omnet++ ed avendo quindi un valido supporto per quanto riguarda la simulazione e l'analisi della comunicazione, è possibile valutare le prestazioni del protocollo di consenso.

Supponendo che il nostro sistema sia in esecuzione in un ambiente dove la comunicazione simulata è di tipo reale, vogliamo dimostrare che anche in presenza di ritardi e disturbi gli agenti robotici sono ancora in grado di raggiungere il consenso.

Capitolo 4

Valutazione dei risultati

4.1 Simulazione in ambiente reale senza perdita

Al fine di effettuare una valutazione completa del protocollo, è bene iniziare da un'analisi del sistema in condizioni di canale reale senza perdite, ovvero, in presenza di un canale che apporta solamente ritardi di trasmissione e quindi con una probabilità di perdita di dati nulla.

Consideriamo inizialmente un sistema composto da cinque agenti robotici disposti come in Figura 4.1.

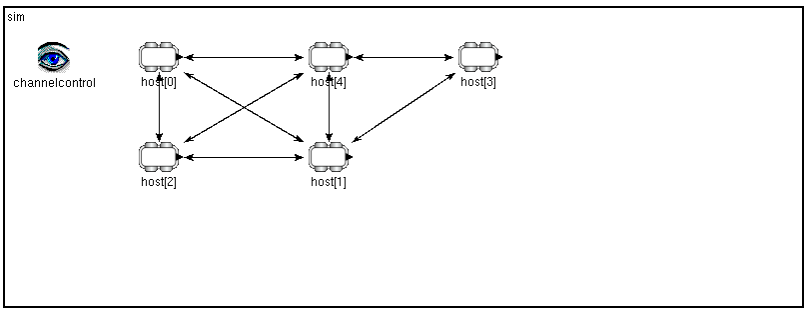


Figura 4.1: Topologia a 5 agenti

Supponiamo che la potenza di trasmissione, pari a 3mW, permetta ad ogni agente di scambiare messaggi solamente con gli agenti immediatamente vicini, che l'intervallo di consensus T sia pari a 1 secondo e che il canale sia senza perdite.

Avendo quindi una percentuale di perdita di pacchetti nulla, vediamo che per ottenere il consenso tra gli agenti robotici, è sufficiente che ciascun robot invii un numero di messaggi pari all'ordine del grafo associato alla topologia.

Il grafo, ottenuto considerando ogni agente come un nodo ed ogni freccia rappresentativa della comunicazione come un arco, ci mostra che in questo caso, è sufficiente inviare due messaggi per fare propagare le informazioni di un agente all'agente più lontano nella rete. Imponiamo perciò n pari a 2.

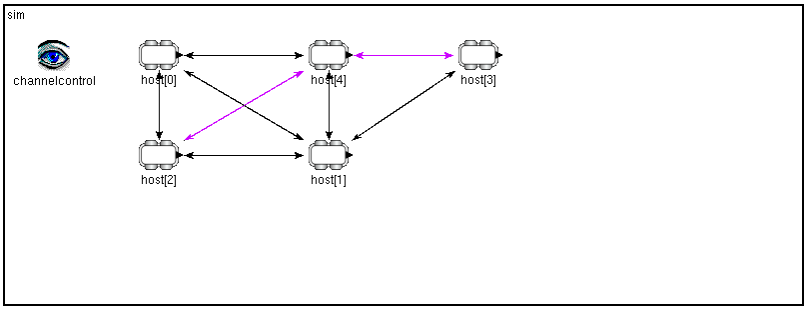


Figura 4.2: Topologia a 5 agenti: n_{min} per coprire la rete

Supponiamo inoltre che l'agente etichettato come numero 2 sia non cooperativo e violi la regola RIGHT. Per violazione della regola si intende il non schedulare questa manovra nonostante le condizioni del sistema siano tali da renderla necessaria.

Topologia	5 agenti
p	0.00
n	2
T	1s
Agente non cooperativo	2
Agente monitor	3

Tabella 4.1: Parametri di configurazione: $p = 0.00$ e $n = 2$

Passando alla simulazione dell'ambiente appena descritto, possiamo osservare il comportamento degli agenti robotici dalla Figura 4.3 alla Figura 4.9.

Tutti gli agenti, in queste condizioni, sono in grado di rilevare l'agente non cooperativo. Nella Figura 4.10 è possibile osservare la reputazione dell'agente 2 assegnata dall'agente 3.

Per convenzione, associamo alle reputazioni dei valori numerici, in particolare:

Cooperativo: 2

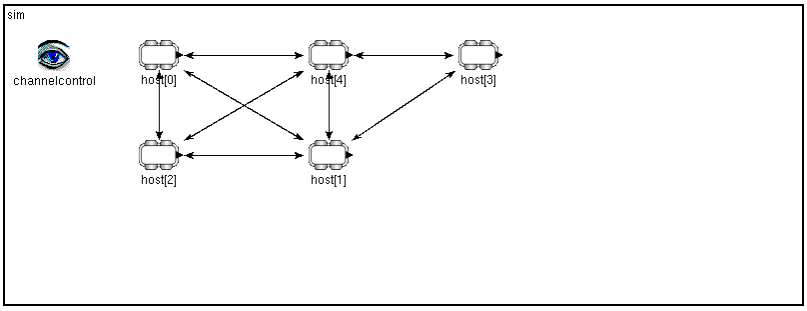


Figura 4.3: Topologia a 5 agenti: 0s

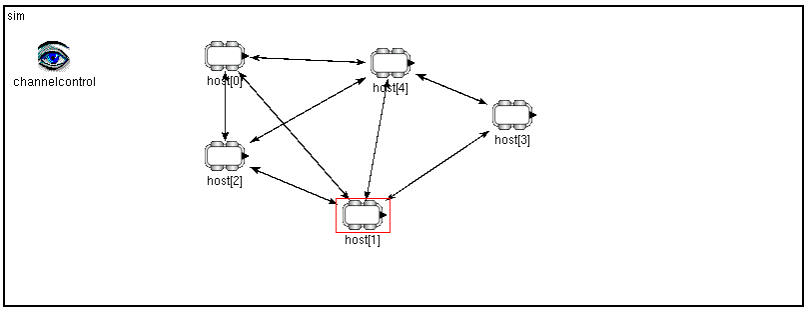


Figura 4.4: Topologia a 5 agenti: 35s

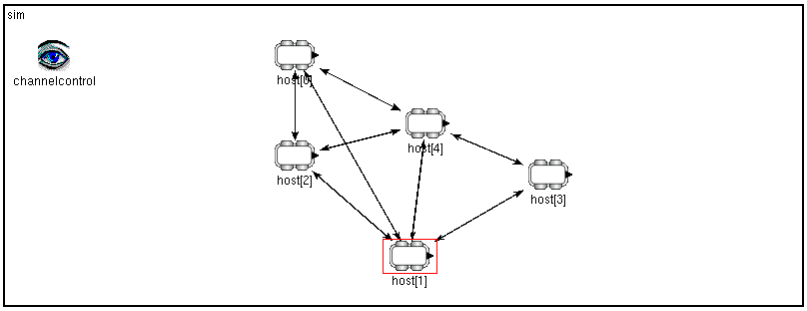


Figura 4.5: Topologia a 5 agenti: 70s

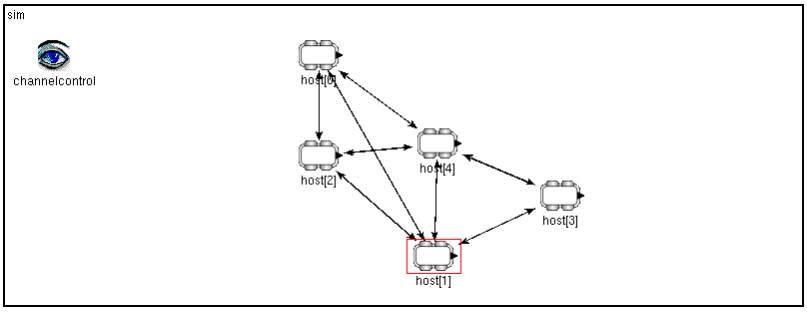


Figura 4.6: Topologia a 5 agenti: 82s

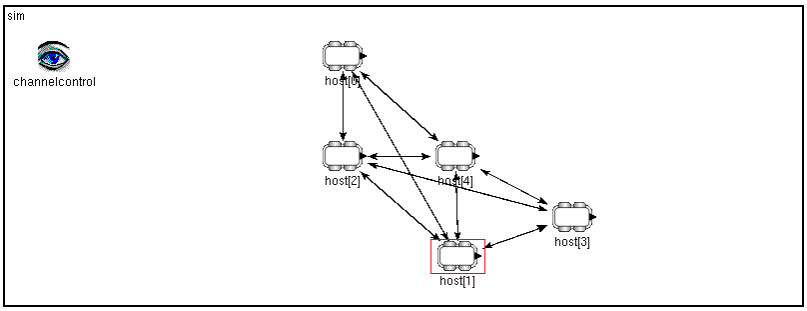


Figura 4.7: Topologia a 5 agenti: 94s

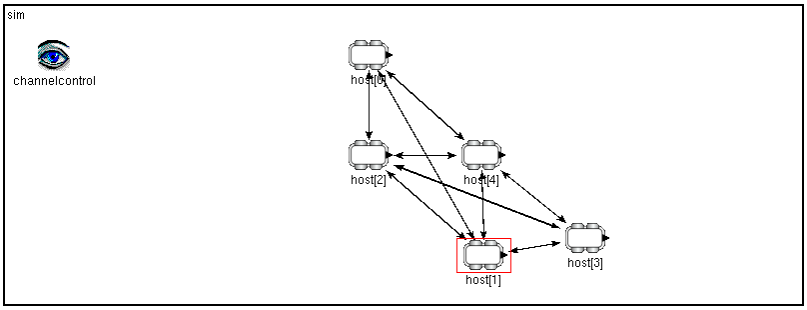


Figura 4.8: Topologia a 5 agenti: 107s

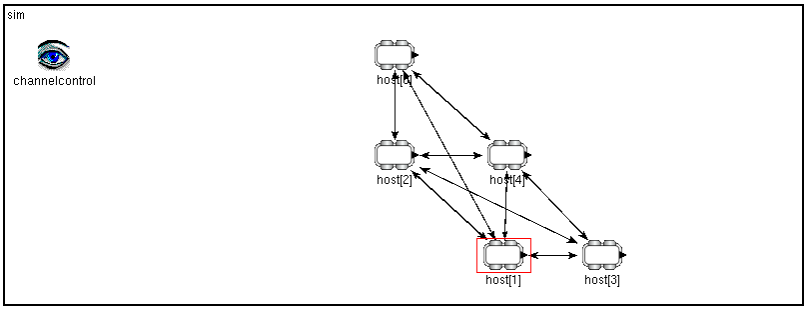


Figura 4.9: Topologia a 5 agenti: 120s

Incerto: 1

Non cooperativo: 0

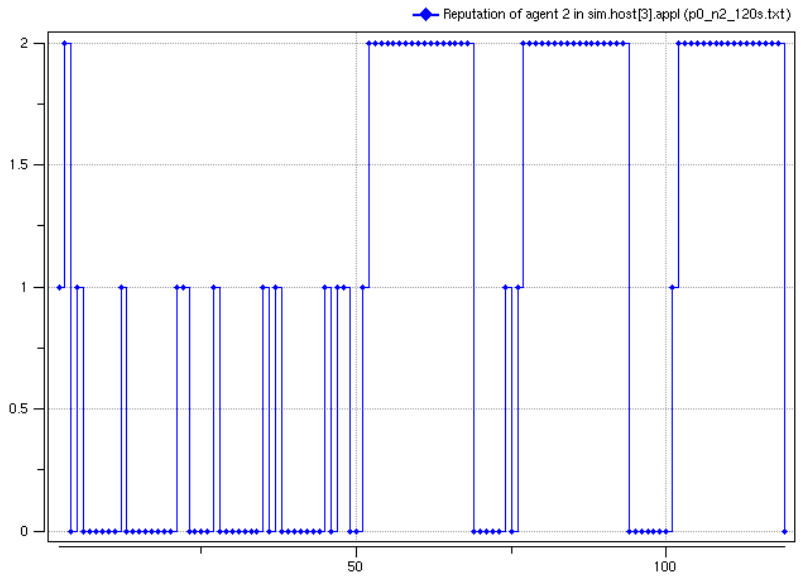


Figura 4.10: Topologia a 5 agenti: reputazione di 2 assegnata da 3 con $p = 0$ e $n = 2$

4.2 Simulazione in ambiente reale con perdita

Supponendo adesso che il canale presenti elevati disturbi, tali da far innalzare la percentuale di perdita al 50%, andiamo ad eseguire la precedente simulazione.

Topologia	5 agenti
p	0.50
n	2
T	1s
Agente non cooperativo	2
Agente monitor	3

Tabella 4.2: Parametri di configurazione: $p = 0.50$ e $n = 2$

Il nostro obiettivo è verificare di poter ottenere di nuovo il consenso tra gli agenti.Come possiamo osservare dalla Figura 4.11 che indica il numero

di pacchetti ricevuti da ogni nodo in ogni step di consenso, ovvero ogni T secondi, l'elevata percentuale di perdita imposta dal canale non consente di far ricevere correttamente il numero minimo di messaggi necessario ad ottenere il consenso.

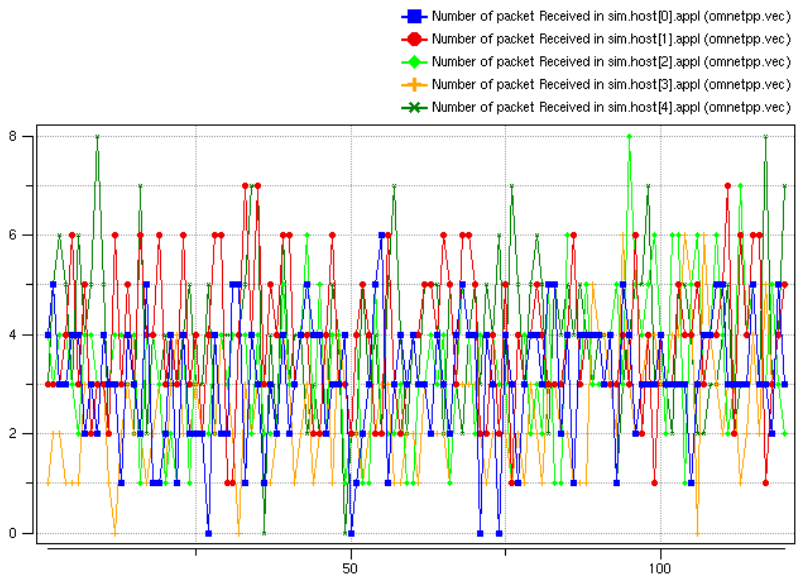


Figura 4.11: Topologia a 5 agenti: numero di messaggi ricevuti con $p = 0.5$ e $n = 2$

Possiamo a questo punto intervenire incrementando il valore del parametro n fino a 4.

Topologia	5 agenti
p	0.50
n	4
T	1s
Agente non cooperativo	2
Agente monitor	3

Tabella 4.3: Parametri di configurazione: $p = 0.50$ e $n = 4$

Terminata la simulazione, in questo caso, vediamo dalla Figura 4.12 che il numero di pacchetti ricevuti è almeno pari a 2, come detto prima, pari al numero minimo di messaggi necessario per ottenere il consenso in questa topologia.

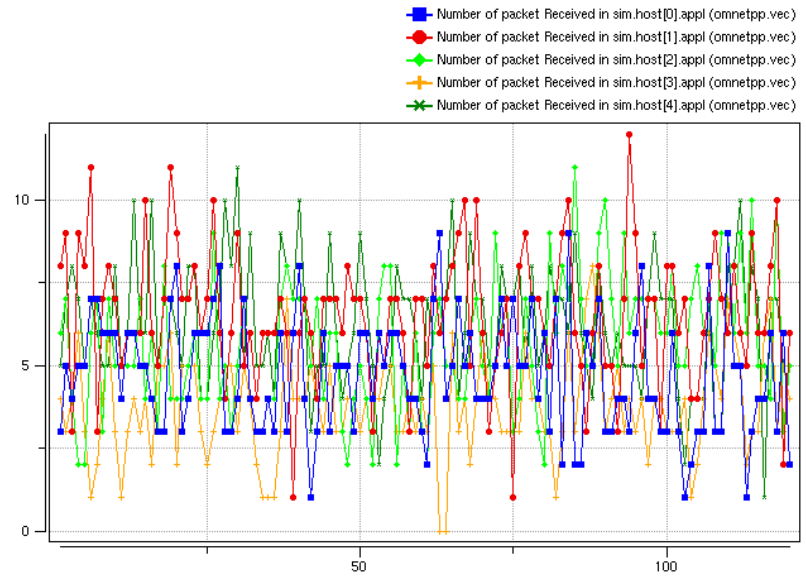


Figura 4.12: Topologia a 5 agenti: numero di messaggi ricevuti con $p = 0.5$ e $n = 4$

Nella Figura 4.13 vediamo l'elevato numero di pacchetti persi in ogni intervallo.

Dalla rappresentazione della reputazione mostrata in Figura 4.14, vediamo che siamo ancora in grado di rilevare la non cooperatività dell'agente 2.

I risultati ci mostrano come sia possibile raggiungere il consenso e rilevare gli agenti malevoli anche in situazioni in cui la comunicazione è notevolmente difficoltosa a causa dell'elevata percentuale di perdita imposta dal canale.

4.3 Confronto tra diverse topologie

Un'altra serie di simulazioni interessante ai fini della valutazione delle prestazioni del protocollo, riguarda l'andamento di n_{min} tale per cui, fissata la topologia, si ottiene il consenso al variare di p, la percentuale di perdita. Supponendo anche in questo caso T pari a 1 secondo, consideriamo, in successione, 8 topologie che si differenziano per il numero di collegamenti che gli agenti hanno tra loro.

9 agenti: Figura 4.15. Data l'elevata ridondanza delle connessioni tra i sis-

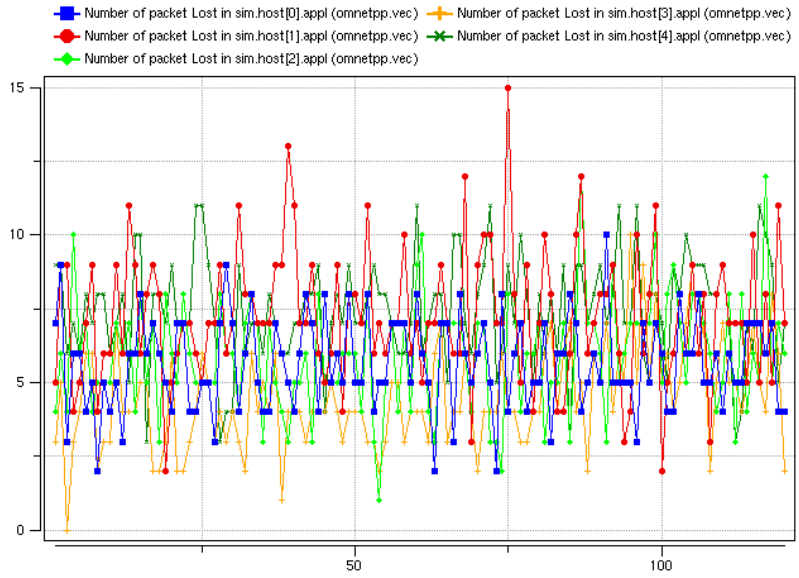


Figura 4.13: Topologia a 5 agenti: numero di messaggi persi con $p = 0.5$ e $n = 4$

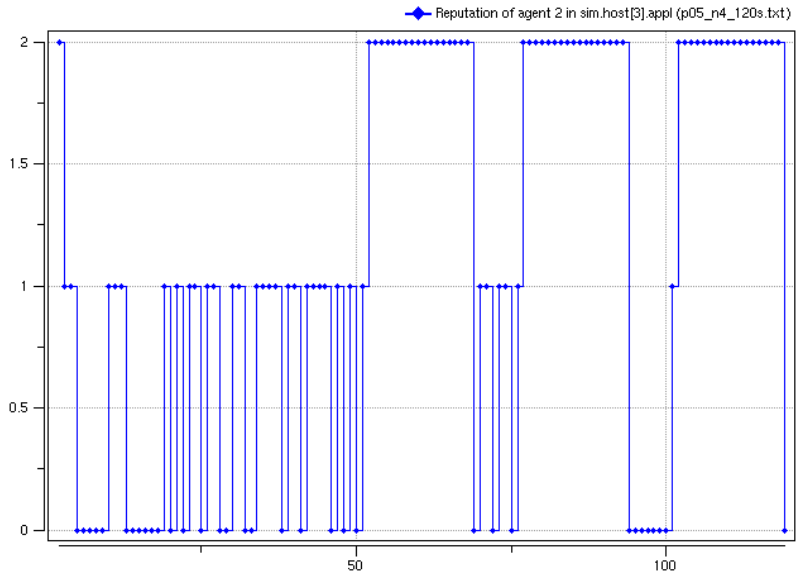


Figura 4.14: Topologia a 5 agenti: reputazione di 2 assegnata da 3 con $p = 0.5$ e $n = 4$

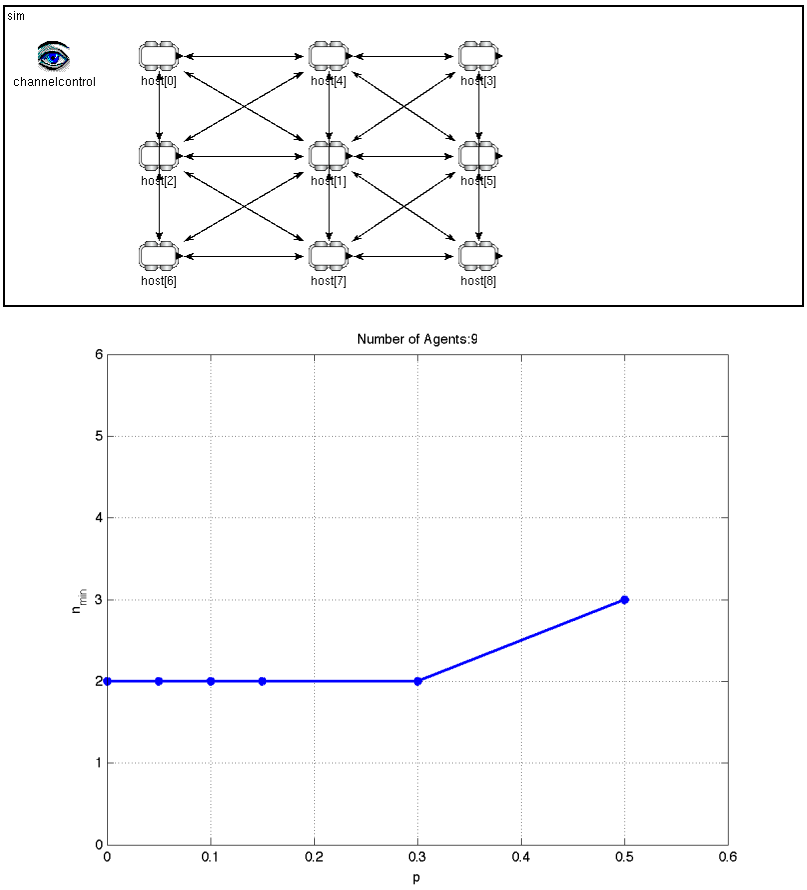


Figura 4.15: Topologia a 9 agenti

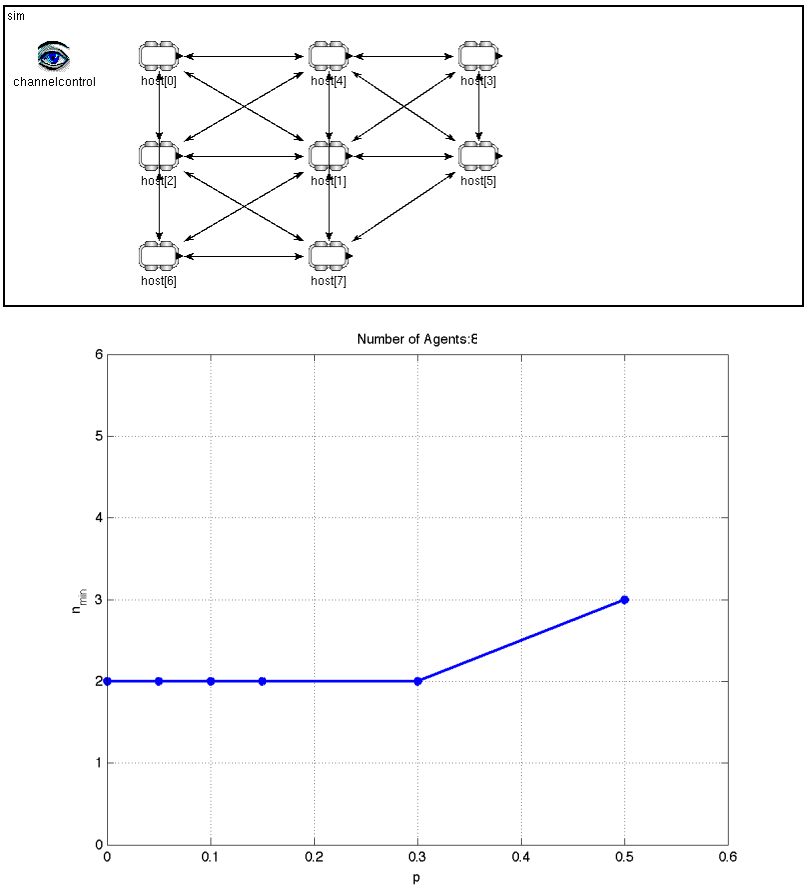
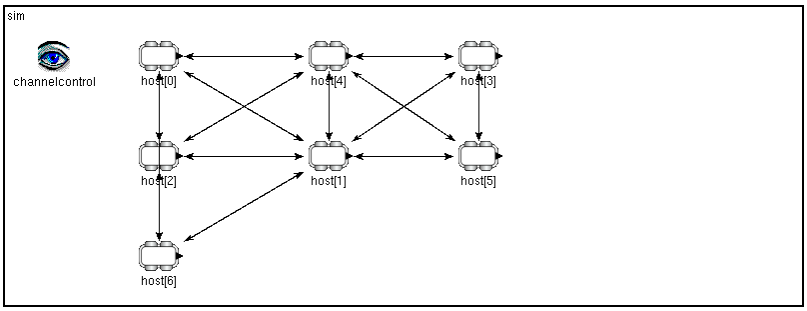


Figura 4.16: Topologia a 8 agenti



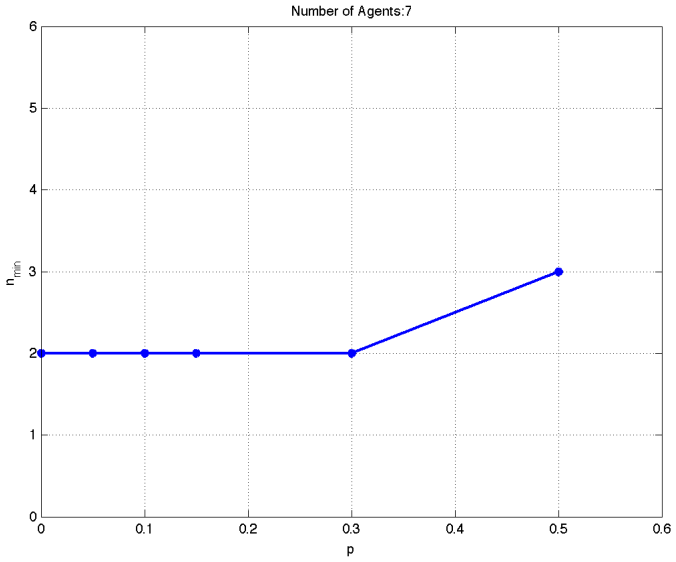
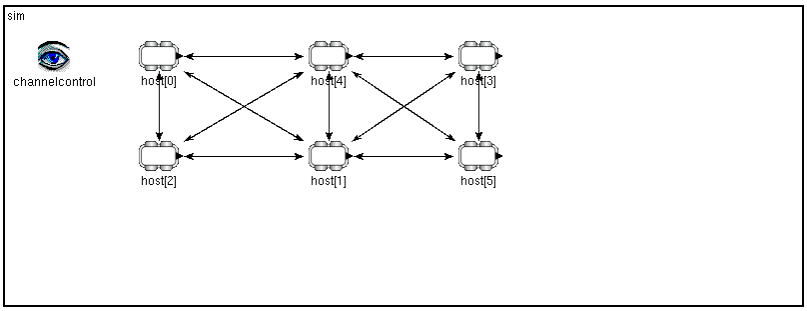


Figura 4.17: Topologia a 7 agenti



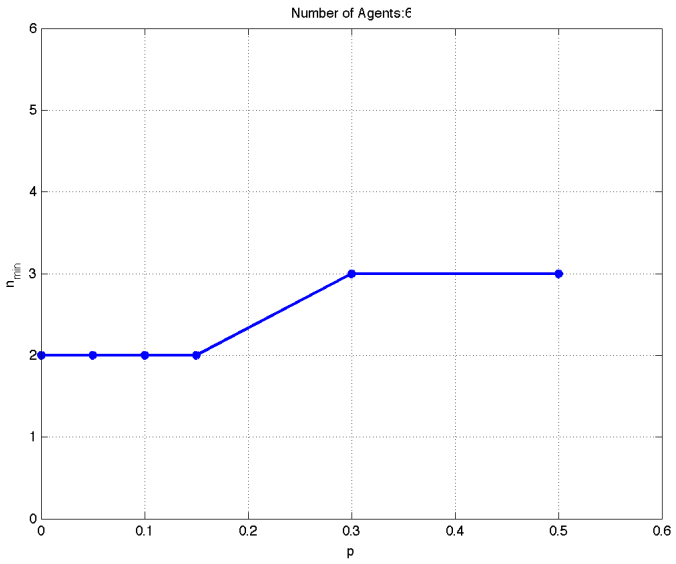
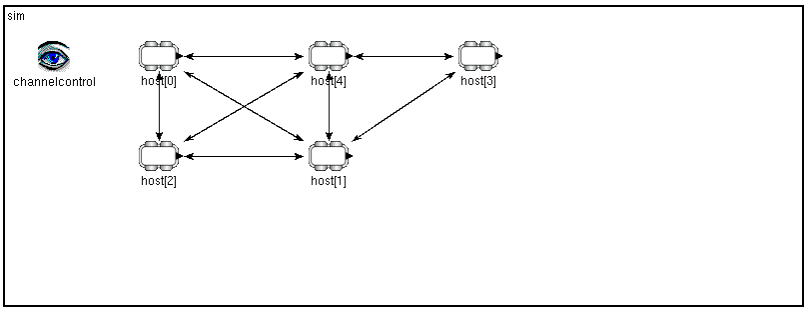


Figura 4.18: Topologia a 6 agenti



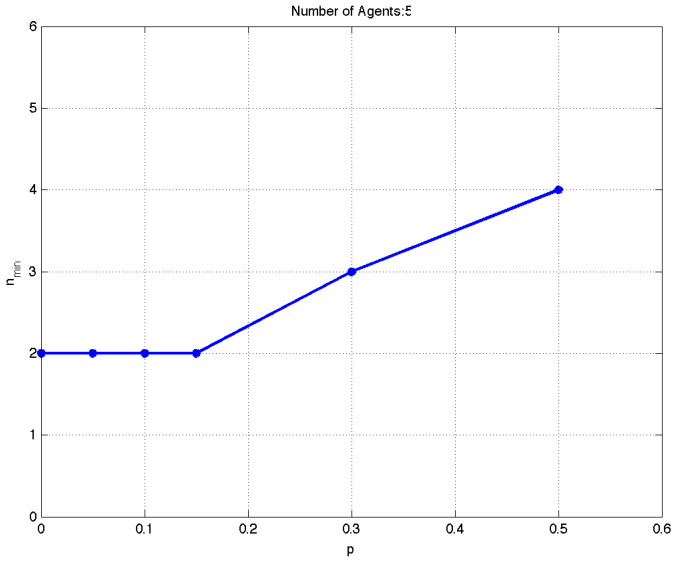
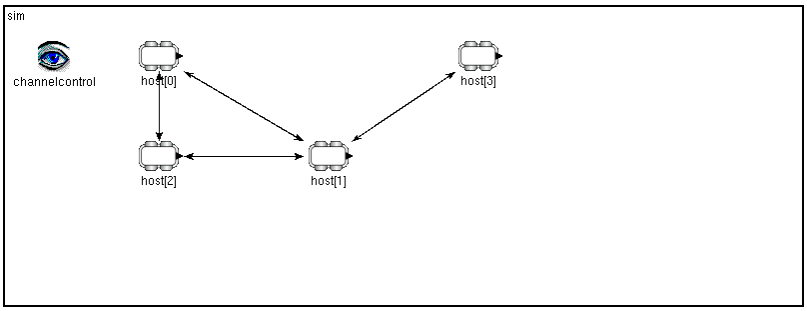


Figura 4.19: Topologia a 5 agenti



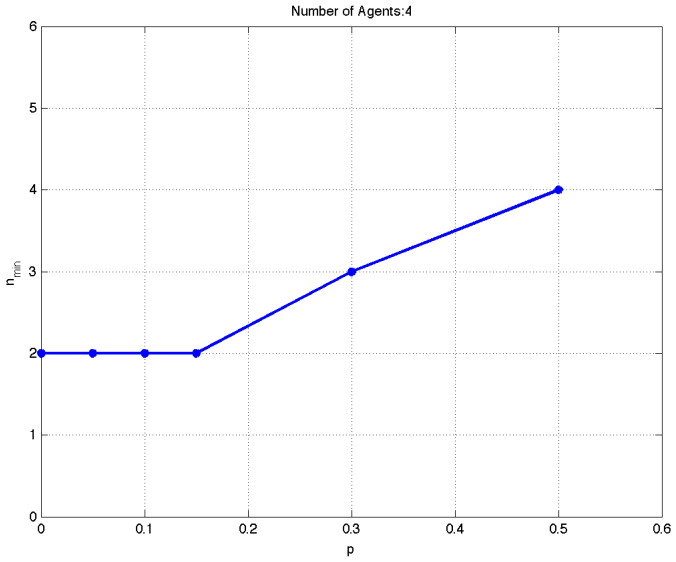
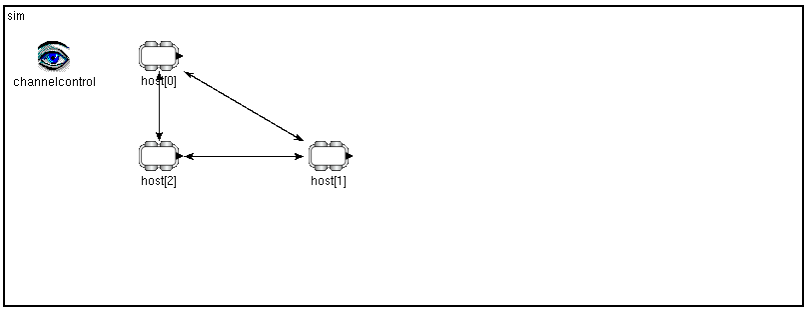


Figura 4.20: Topologia a 4 agenti



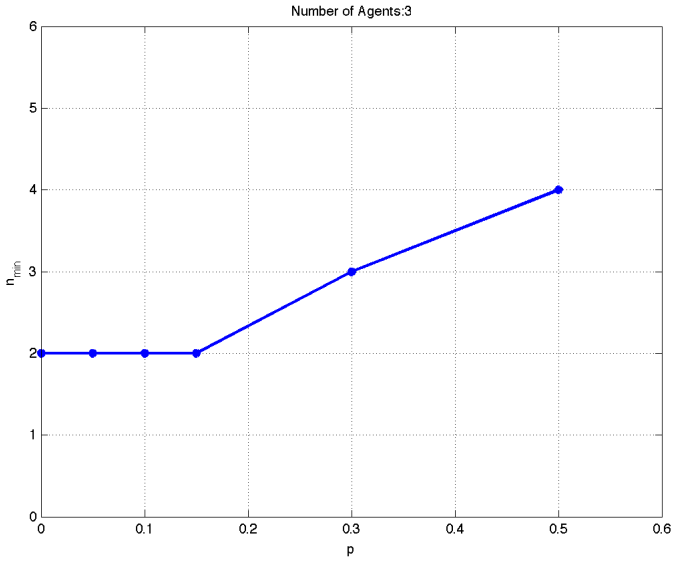


Figura 4.21: Topologia a 3 agenti

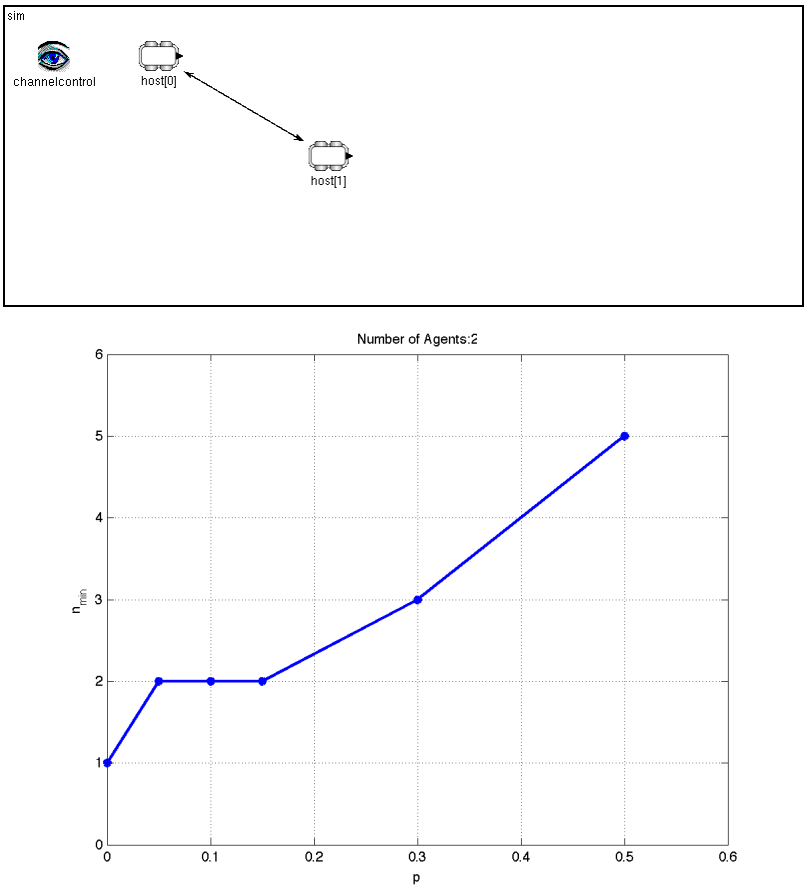


Figura 4.22: Topologia a 2 agenti

temi robotici, vediamo che è sufficiente n_{min} piccolo al fine di ottenere il consenso.

8 agenti: Figura 4.16.

7 agenti: Figura 4.17.

6 agenti Figura 4.18.

5 agenti: Figura 4.19. Come possiamo osservare, questa è una situazione molto rilevante e che rappresenta la transizione tra ambiente con agenti con poche connessioni e agenti con molte connessioni.

4 agenti: Figura 4.20.

3 agenti: Figura 4.21.

2 agenti: Figura 4.22. In questa configurazione entrambi gli agenti hanno una sola connessione con l'altro e quindi nessuna ridondanza che aiuti a ovviare alle perdite di messaggi.

Nella tabella sono riassunti i valori di n_{min} al variare di p.

	0.00	0.05	0.10	0.15	0.30	0.50
9	2	2	2	2	2	3
8	2	2	2	2	2	3
7	2	2	2	2	2	3
6	2	2	2	2	3	3
5	2	2	2	2	3	3
4	2	2	2	2	3	4
3	2	2	2	2	3	4
2	1	2	2	2	3	5

Tabella 4.4: Tabella riassuntiva

In conclusione, osservando i precedenti grafici, si nota che togliendo connessioni tra gli agenti, la ridondanza diminuisce e di conseguenza il numero minimo di messaggi necessario per ottenere il consenso tende ad aumentare aumentare.

Capitolo 5

Conclusioni

In seguito alla parziale reimplementazione, in Omnet++, di un simulatore non in grado di simulare in modo completo un sistema reale, siamo stati in grado di dimostrare che il protocollo di consenso continua a funzionare anche in situazioni reali, ovvero, in cui il canale di comunicazione non è molto affidabile e porta ad avere una notevole perdita di messaggi.

Abbiamo inoltre dimostrato che il numero minimo di messaggi da trasmettere all'interno di ogni intervallo T, è dipendente dalla probabilità di perdita di pacchetti del canale e dalla particolare topologia in cui ci troviamo. Infatti, maggiore è il numero di collegamenti che ogni agente ha con gli altri, maggiore è la quantità di informazioni che esso riesce a scambiare.

Appendice A

Sorgenti

A.1 Reputation Manager

A.1.1 ReputationManager.h

```
class ReputationManager {
    protected:
        /*
         * Coppie idTarget, Monitor associato
         */
        std::map<int, Monitor> agent;
        /*
         * Utilizzato per gestire in modo corretto
         * il primo passo dell'algoritmo dove il Monitor
         * non ha ancora potuto calcolare le posizioni ammesse
         * perche' non ha dati
         */
        bool firstStep;
        RMobility * mobility;
        double tSim;
        /*
         * Calcola le posizioni finali ammesse
         */
        void stepInit(std::vector<CoordId> qList,
                      std::vector<BlockCoord> hList);
```

```
        /*
         * Verifica se la posizione reale appartiene
         * all'insieme di quelle calcolate
         */
        std::map<int,int> stepCalculate(std::vector<CoordId> qList,
                                        std::vector<BlockCoord> hList);

    public:
        ReputationManager(RMobility *mobility, double _tSim);
        std::map<int,int> step(std::vector<CoordId> qList,
                               std::vector<BlockCoord> hList);

};
```

A.1.2 ReputationManager.cc

```
ReputationManager::ReputationManager(RMobility * _mobility,
                                     double _tSim)
{
    mobility = _mobility;
    tSim = _tSim;
    firstStep = true;
}

void ReputationManager::stepInit(std::vector<CoordId> qList,
                                 std::vector<BlockCoord> hList)
{
    // scorro ogni elemento della qList
    for(unsigned int i=0; i<qList.size(); i++) {
        std::map<int,Monitor>::iterator it = agent.find(qList[i].id);
        // se non c'e' un Monitor per quell'elemento lo creo
        if( it == agent.end() ) {
            agent[qList[i].id] = Monitor(mobility, qList[i].id, tSim);
        }
    }
```

```
    // scorro i Monitor, per ciascuno chiamo la step
    for( std::map<int,Monitor>::iterator it=agent.begin();
                                        it != agent.end(); it++ ) {
        (*it).second.step(qList, hList);
    }
    return;
}

std::map<int,int> ReputationManager::stepCalculate(
                                std::vector<CoordId> qList,
                                std::vector<BlockCoord> hList)
{
    std::map<int,int> ret;
    /*
     * scorro la qList, per ogni agente richiamo la isCooperative()
     * se esso e' presente nella mappa
     */
    for(unsigned int i=0; i<qList.size(); i++) {
        std::map<int,Monitor>::iterator it = agent.find(qList[i].id);
        // se l'agente c'e' allora verifico che sia cooperativo
        if( it != agent.end() ) {
            ret[qList[i].id] = agent[qList[i].id].isCooperative(
                                    Coord(qList[i].x, qList[i].y) );
        }
    }
    return ret;
}

std::map<int,int> ReputationManager::step(
                                std::vector<CoordId> qList,
                                std::vector<BlockCoord> hList)
{
    std::map<int,int> ret;
    if(firstStep == false) {
```

```
        ret = stepCalculate(qList, hList);
    } else {
        firstStep = false;
    }
    stepInit(qList, hList);
    return ret;
}
```

A.2 Monitor

A.2.1 Monitor.h

```
class Monitor {
    protected:
        /*
         * puntatore al modulo di movimento per richiamare
         * le funzioni di simulazione
         */
        RMobility *mobility;
        int idTarget;
        /*
         * durata della simulazione per la previsione
         */
        double tSim;
        /*
         * contiene le posizioni che vengono predette dal monitor
         * se la nuova posizione appartiene a cooperative,
         * allora l'agente e' cooperativo
         * altrimenti e' non cooperativo
         */
        std::map<int,Position> cooperative;

    public:
        Monitor();
        Monitor(RMobility * _mobility, int id, double _tSim);
```

```
        void init(RMobility * _mobility, int id, double _tSim);
        void step(std::vector<CoordId> qList,
                  std::vector<BlockCoord> hList);
        int isCooperative(Coord pos);
};
```

A.2.2 Monitor.cc

```
Monitor::Monitor()
{
    init(NULL, -1, 0.0);
}

Monitor::Monitor(RMobility * _mobility, int id, double _tSim)
{
    init(_mobility, id, _tSim);
}

void Monitor::init(RMobility * _mobility, int id, double _tSim)
{
    mobility = _mobility;
    idTarget = id;
    tSim = _tSim;
}

void Monitor::step(std::vector<CoordId> qList,
                   std::vector<BlockCoord> hList)
{
    Position tmpPos;
    cooperative.clear();
    // simulazioni sfruttando le ipotesi
    if( mobility->simulateRight(idTarget,
                                qList, hList, tSim, tmpPos) ) {
        cooperative[RMobility::RIGHT] = tmpPos;
    } else
```

```
    if( mobility->simulateLeft(idTarget,
                               qList, hList, tSim, tmpPos) ) {
        cooperative[RMobility::LEFT] = tmpPos;
    } else
    if( mobility->simulateBrake(idTarget,
                                qList, hList, tSim, tmpPos) ) {
        cooperative[RMobility::BRAKE] = tmpPos;
    } else
    {
        mobility->simulateFast(idTarget,
                               qList, hList, tSim, tmpPos);
        cooperative[RMobility::FAST] = tmpPos;
    }

    // simulazioni con permutazione delle celle unknown
    if( mobility->simulateRight(idTarget, qList, tSim, tmpPos) ) {
        cooperative[RMobility::RIGHT] = tmpPos;
    } else
    if( mobility->simulateLeft(idTarget, qList, tSim, tmpPos) ) {
        cooperative[RMobility::LEFT] = tmpPos;
    } else
    if( mobility->simulateBrake(idTarget, qList, tSim, tmpPos) ) {
        cooperative[RMobility::BRAKE] = tmpPos;
    } else
    {
        mobility->simulateFast(idTarget, qList, tSim, tmpPos);
        cooperative[RMobility::FAST] = tmpPos;
    }
    return;
}

int Monitor::isCooperative(Coord pos)
{
    for( std::map<int,Position>::iterator it=cooperative.begin();
```

```
                                it != cooperative.end(); it++ ) {
        if( (*it).second.contains(pos.x, pos.y) ) {
            if( cooperative.size() == 2) {
                // incerto
                if( (*it).first == RMobility::RIGHT )
                    std::cout << idTarget << " is INCERTO\n";
                if( (*it).first == RMobility::LEFT )
                    std::cout << idTarget << " is INCERTO\n";
                if( (*it).first == RMobility::BRAKE )
                    std::cout << idTarget << " is INCERTO\n";
                if( (*it).first == RMobility::FAST )
                    std::cout << idTarget << " is INCERTO\n";
                return 1;
            } else {
                // certo e cooperativo
                if( (*it).first == RMobility::RIGHT )
                    std::cout << idTarget
                              << " is cooperative: RIGHT\n";
                if( (*it).first == RMobility::LEFT )
                    std::cout << idTarget
                              << " is cooperative: LEFT\n";
                if( (*it).first == RMobility::BRAKE )
                    std::cout << idTarget
                              << " is cooperative: BRAKE\n";
                if( (*it).first == RMobility::FAST )
                    std::cout << idTarget
                              << " is cooperative: FAST\n";
                return 2;
            }
        }
    }
    std::cout << idTarget << " is NOT cooperative\n";
    return 0;
}
```

A.3 Mobility Module

A.3.1 RMobility.h

```
class RMobility : public BasicMobility
{
  public:
    enum ManeuverType {BRAKE, RIGHT, LEFT, FAST};

  protected:
    enum stato {UNKNOWN, LIBERO, OCCUPATO};
    ManeuverType maneuver;
    typedef std::vector<stato> Rowstato;
    typedef std::vector<Rowstato> Matrixstato;

    // Valore max dell'angolo quando si gira
    double angleMax;
    // Angolo
    double angle;
    // Accelerazione
    double acceleration;
    /*
     * Mantiene caso per caso la condizione di fine
     * manovra da controllare (attualmente si usa solo la y)
     */
    Coord maneuverFinal;
    double topSpeed;

    Matrixstato hypMatrix;
    double playgroundSizeX;
    double playgroundSizeY;
    double xBlockSize;
    double yBlockSize;

    // Disabilita il modulo di mobilità
```

```
    bool stop_mobility;
    /*
     * intervallo di update del modulo di Mobility
     * si usa per simulare le manovre
     */
    double updateInterval;

    /*
     * Mantiene l'ultimo step per l'update
     * della visualizzazione
     */
    Coord stepTarget;

  public:
    Module_Class_Members( RMobility, BasicMobility, 0 );

    // Inizializza i parametri del modulo
    virtual void initialize(int);

    std::vector<CoordId> getSensingList();
    double getXBlockSize(){ return xBlockSize; }
    double getYBlockSize(){ return yBlockSize; }

    /*
     * Le seguenti funzioni fungono da interfaccia per le
     * relative funzioni Block e restituiscono
     * true o false in base alla possibilita' di eseguire
     * la specifica manovra
     */
    bool testFast(Coord& myPos, Matrixstato& matrix);
    bool testBrake(Coord& myPos, Matrixstato& matrix);
    bool testLeft(Coord& myPos, Matrixstato& matrix);
    bool testRight(Coord& myPos, Matrixstato& matrix);
```

```
  /*
   * Le seguenti funzioni fungono da simulatori per le
   * relative manovre e restituiscono true o false
   * in base alla possibilita' di eseguire la specifica
   * manovra
   */
  bool simulateBrake(int idTarget,
                     std::vector<CoordId> qList,
                     double tSim, Position& posArea);
  bool simulateRight(int idTarget,
                     std::vector<CoordId> qList,
                     double tSim, Position& posArea);
  bool simulateLeft(int idTarget,
                    std::vector<CoordId> qList,
                    double tSim, Position& posArea);
  bool simulateFast(int idTarget,
                    std::vector<CoordId> qList,
                    double tSim, Position& posArea);

protected:
  // funzione di movimento dell'host
  virtual void makeMove();

  virtual void fixIfHostGetsOutside();

  void buildHypMatrix(std::vector<BlockCoord>);
  void secondPassBuildHypMatrix();
  void contornoElemHypMatrix(int i, int j,
                             bool doppio = false);

  /*
   * Le seguenti funzioni restituiscono true o false
   * in base alla possibilita' di eseguire
   * la specifica manovra
```

```
     */
    bool testBlockFast(BlockCoord& myPos, Matrixstato& matrix);
    bool testBlockBrake(BlockCoord& myPos, Matrixstato& matrix);
    bool testBlockLeft(BlockCoord& myPos, Matrixstato& matrix);
    bool testBlockRight(BlockCoord& myPos, Matrixstato& matrix);
};
```

A.4 Agent Host

A.4.1 AgentHost.ned

```
import
    "AgentApplLayer",
    "SimpleArp",
    "SimpleNetwLayer",
    "NicCsma",
    "RMobility",
    "Blackboard";

module AgentHost
    parameters:
        xBlockSize : numeric const,
        yBlockSize : numeric const;

    gates:
        in: radioIn; // gate per sendDirect

    submodules:
        blackboard: Blackboard;
            display: "p=200,75;b=30,25";

        mobility: RMobility;
            parameters:
            display: "p=200,150;b=30,25";
```

```
        arp: SimpleArp;
            display: "p=200,225;b=30,25";

        appl: AgentApplLayer;
            parameters:
            display: "p=90,50;b=100,20,rect";

        net: SimpleNetwLayer;
            display: "p=90,100;b=100,20,rect";

        nic: NicCsma;
            display: "p=90,150;b=100,20,rect";

    connections:
        nic.uppergateOut --> net.lowergateIn;
        nic.uppergateIn <-- net.lowergateOut;
        nic.upperControlOut --> net.lowerControlIn;

        net.uppergateOut --> appl.lowergateIn;
        net.uppergateIn <-- appl.lowergateOut;
        net.upperControlOut --> appl.lowerControlIn;

        radioIn --> nic.radioIn;

    display: "p=10,10;b=250,250,rect;o=white";
endmodule
```

A.5 Network

A.5.1 Network.ned

```
import
    "AgentHost",
    "AgentChannelControl";
```

```
module Sim
    parameters:
        // parametri per il framework
        playgroundSizeX : numeric const,
        playgroundSizeY : numeric const,
        xBlockSize : numeric const,
        yBlockSize : numeric const,
        numHosts : numeric const;

    submodules:
        channelcontrol: AgentChannelControl;
            parameters:
                playgroundSizeX = playgroundSizeX,
                playgroundSizeY = playgroundSizeY;
            display: "p=50,50;i=igor";
        host: AgentHost[numHosts];
            parameters:
                xBlockSize = xBlockSize,
                yBlockSize = yBlockSize;
            display: "p=50,30;i=agent";

    connections nocheck:
    // tutte le connessioni e i gates verranno
    // generate automaticamente

    display: "p=0,0;b=$playgroundSizeX,$playgroundSizeY,rect;o=white";
endmodule

network sim : Sim
    parameters:
        playgroundSizeX = input(40,"playgroundSizeX"),
        playgroundSizeY = input(40,"playgroundSizeY"),
        xBlockSize = input(50,"xBlockSize"),
        yBlockSize = input(100,"yBlockSize"),
```

```
        numHosts =input(2,"Number of hosts:");
endnetwork
```

Appendice B

Installazione Omnet++ e Mobility Framework

B.1 GNU/Linux Debian unstable 368 (architettura 32 bit)

Inseriamo in `/etc/apt/sources.list` i seguenti repository.

```
deb http://tevp.net/debian stable/
deb http://tevp.net/debian testing/
deb http://tevp.net/debian unstable/
deb-src http://tevp.net/debian source/
```

A questo punto è necessario recuperare la lista dei nuovi pacchetti disponibili ed installare Omnet++.

```
apt-get update
apt-get upgrade
apt-get install omnetpp
```

B.2 Installazione Mobility Framework (architettura 32 bit)

Scarichiamo il framework dal sito *http://sourceforge.net/projects/mobility-fw/* e salviamolo nella nostra home directory che da ora in avanti indicherò con `/_home_dir_/`. Scompattiamolo nella cartella corrente con il comando

```
tar xzvf mobile-fw2.0p3.tgz
```

Settiamo la variabile di ambiente `LD_LIBRARY_PATH`

```
export LD_LIBRARY_PATH=
${LD_LIBRARY_PATH}":/_home_dir_/mobility-fw2.0p3/core/lib
:/_home_dir_/mobility-fw2.0p3/contrib/lib"
```

Entriamo nella cartella appena creata di nome mobile-fw2.0p3 ed eseguiamo il comando `./mkmk`. Prima di compilare il framework è necessario patchare uno dei sorgenti di omnetpp. Da utente root eseguiamo il comando

```
vim /usr/include/omnetpp/cwatch.h
```

ed inseriamo subito sotto gli include le seguenti righe

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
```

Adesso possiamo compilare il framework. Torniamo ad essere utenti non privilegiati e nella cartella

```
/_home_dir_/mobility-fw2.0p3/
```

eseguiamo il comando `make all` che provvederà a compilare contrib, core e networks.

Alla fine della compilazione eseguiamo il comando `make testSuite` per compilare i file di test con cui verificare il funzionamento combinato di Omnet++ e Mobility framework. A questo scopo dobbiamo eseguire i tre programmi presenti nelle sottodirectory di `./testSuite`.

Appendice C

Struttura della directory

Dopo aver installato e testato il kernel di Omnet++ ed il Mobility Framework come descritto in Appendice B, siamo in grado di creare un ambiente di lavoro in qualunque directory. Scegliamo ad esempio `/_home_dir_/esempio/`.

All'interno di questa directory andremo ad inserire tutti i files che serviranno al nostro progetto. Solitamente i files riportati nella seguente lista sono sufficienti a personalizzare una network.

```
/MyApplLayer.ned
/MyApplLayer.h
/MyApplLayer.cc
/MyMobility.ned
/MyMobility.h
/MyMobility.cc
/MyHost.ned
/Network.ned
/omnetpp.ini
```

Ogni coppia di file è necessaria per implementare uno modulo semplice, mentre il file .ned determina un modulo complesso. Possiamo ridefinire anche la struttura del singolo messaggio scambiato personalizzando i seguenti files:

```
/MyPkt.msg
/MyPkt.h
/MyPkt.cc
```

Diamo una breve occhiata alle estensioni per capire cosa dovranno contenere:

.ned: contiene la definizione di un singolo modulo complesso in linguaggio ned. Questa definizione sarà processata prima della compilazione e saranno generati due files `MyNed_m.cc` e `MyNed_m.h` supponendo che il nome del file originario fosse MyNed.ned.

.h: contiene la dichiarazione della classe che implementa il modulo.

.cc: contiene le definizioni delle funzioni membro della classe dichiarata nel relativo .h. Inoltre, contiene una o più direttive necessarie ad omnet per riconoscere e quindi gestire correttamente il modulo.

Nella directory sarà inoltre presente un file `Makefile.gen` necessario per creare il nuovo makefile ogni volta che andremo a modificare i files .ned e .msg.

In Figura C.1 si può vedere lo schema di compilazione, link tra files ed esecuzione dell'applicazione.

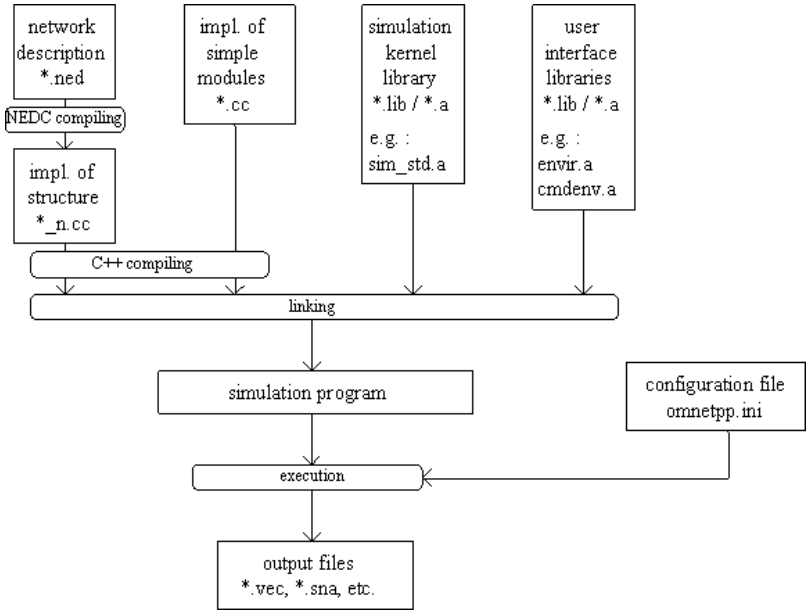


Figura C.1: Schema di compilazione, link tra files e esecuzione

Una volta definiti i moduli singoli e composti, i messaggi e tutto ciò che ci serve, avremo bisogno di compilare i files sorgenti, configurare l'ambiente

di simulazione mediante il file `omnetpp.ini` ed infine eseguire l'applicazione. Dopo aver eseguito la simulazione, se previsto all'interno del codice, saranno generati due files

omnetpp.sca: questo file contiene delle quantità scalari, solitamente medie o altri valori calcolati sull'intera simulazione

omnetpp.vec: questo file contiene dei vettori, solitamente valori campionati periodicamente

I comandi da eseguire sono i seguenti:

`make -f Makefile.gen`: compila i files .ned ed .msg e crea il file `Makefile`. Bisogna notare che il nome dell'eseguibile generato sarà uguale al nome della directory

`make`: compila l'applicazione

Adesso, prima di avviare l'applicazione, rimane da editare il file `omnetpp.ini` con un qualsiasi editor di testo in modo da configurare la nostra simulazione come meglio desideriamo. Per avviare l'applicazione sarà sufficiente digitare

```
./nome_appl
```

Una volta eseguita la simulazione, è possibile graficare i valori contenuti all'interno di `omnetpp.sca` e `omnetpp.vec` eseguendo i comandi

```
scalar omnetpp.sca
plove omnetpp.vec
```