

Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Corso di Laurea Magistrale in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria



Protocolli per la diffusione epidemica di messaggi in reti Bluetooth di emergenza

Relatore: Prof. Raffaela Mirandola
Correlatore: Ing. Lorenzo Pagliari

Tesi di laurea di:
Davide Malvestiti - Matr. 858664

Anno Accademico 2017-2018



“Ogni cosa è meglio col bluetooth”

Sheldon Cooper

Ringraziamenti



Il mondo delle telecomunicazioni e delle reti è il perno dell'odierna società.

Di comune dominio è il senso d'indispensabilità del poter comunicare.

Può tuttavia succedere che particolari eventi, sia atmosferici (grandinate, alluvioni, ecc.) che naturali (terremoti, frane, ecc.) o di altra natura (guerre, attentati, ecc.) determino un impedimento alle reti di telecomunicazione rendendo di fatto impossibile la trasmissione di notizie e/o informazioni spesso di fondamentale importanza.

Da questi assunti era già partito un lavoro svolto in passato al Politecnico di Milano per sperimentare soluzioni in grado di sopperire alle difficoltà di comunicazione dei dispositivi mobili.

Con lo studio odierno ci siamo posti lo scopo di ampliare il precedente lavoro utilizzando per la sperimentazione un programma differente che permettesse un più alto livello di realismo.

Mostreremo in dettaglio la soluzione già proposta: un sistema che cerca di diffondere messaggi in maniera epidemica sfruttando come canale di trasmissione la tecnologia Bluetooth Low Energy, quindi con particolare attenzione anche al tema del risparmio energetico.

Successivamente illustreremo in dettaglio il programma di simulazione da noi utilizzato, scelto in modo che possa permettere di riprodurre ed estendere le precedenti simulazioni.

Illustreremo inoltre i risultati delle simulazioni, che permettono di analizzare il sistema per diverse situazioni, che tengano conto sia delle densità che della dimensione della popolazione coinvolta. Questi risultati verranno confrontati con quelli del lavoro precedente.

Infine saranno analizzate nel dettaglio le varie estensioni del protocollo, in particolare le due che hanno permesso di simulare più realisticamente lo schema distributivo dei nuclei abitativi e il movimento delle persone. In ogni città, gli abitanti non sono mai distribuiti uniformemente, è stato quindi necessario migliorare il modo in cui vengono disposti i nodi nella rete.

L'ultima estensione che mostriamo permette di simulare la localizzazione di persone disperse utilizzando la tecnologia GPS. Si tratta quindi di una funzionalità molto diversa dalle altre.

Vedremo infine un'analisi delle proprietà del nostro codice.



Abstract

The world of telecommunications and networks is the pinnacle of today's society. Nowadays it is perceived as necessary to be able to communicate.

However, it may happen that particular events, such as atmospheric ones (hailstorms, floods, etc.), natural ones (earthquakes, landslides, etc.) or of other nature (wars, attacks, etc.) may cause an impediment to the telecommunications networks, making it impossible the transmission of news and / or information of often fundamental importance.

From these assumptions, a previous work has already started at Politecnico di Milano to experiment with solutions able to overcome the difficulties in communicating with mobile devices.

With this research, we aim to set the goal of expanding the previous work by using a different program for experimentation, which could allow a higher level of realism.

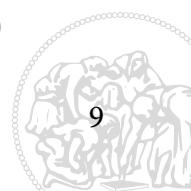
We will show in detail the solution already proposed: a system that tries to spread messages in an epidemic way, using the Bluetooth Low Energy technology as a transmission channel, paying particular attention to the topic of energy saving.

Later, we will show in detail the simulation program used, which can reproduce and extend the previous simulations. We will also show the results of the simulations, which allow analyzing the system in different situations, which take into account both the density and the size of the population involved. These results will then be compared with those of the previous work. Finally, various extensions of the protocol will be analyzed in detail, in particular the two that have made it possible to simulate more realistically the distribution pattern of housing units and people's movement.

In every city, the inhabitants are never distributed evenly, so it was necessary to improve the way the nodes are arranged in the network. The last extension that we show allows simulating the localization of lost people using GPS technology. It is a different functionality from the others. Finally, we will see an analysis of the properties of our code.

Indice

Sommario	6
Abstract	7
Elenco delle figure	11
Elenco degli algoritmi	13
Capitolo 1 - Introduzione	15
1.1 Struttura del documento	16
Capitolo 2 - Stato dell'arte	19
2.1 Sistemi Self-Adaptive	19
2.2 Stati di operatività del link layer del BLE	20
2.3 Reti Peer-to-Peer	21
2.4 Random Geometric Graph	22
2.5 Metodi di diffusione degli algoritmi di gossip	23
2.6 Fixed Fanout	25
2.7 Peersim	25
Capitolo 3 - Funzionamento e riprogettazione del protocollo	31
3.1 Introduzione	31
3.1.1 Stati del link layer	32
3.1.2 Funzionamento effettivo del protocollo	33
3.2 Progettazione del codice	35
3.2.1 Analisi della struttura del codice	35
Capitolo 4 - Simulazioni e valutazioni dei risultati	43
4.1 Valutazione qualitativa dei risultati	45
4.2 Valutazione risultati	52
Capitolo 5 - Estensioni	57
5.1 Introduzione	57
5.1.1 Estensioni: coda di priorità	60



5.1.2 Estensioni: caso multi-messaggio	64
5.1.3 Estensioni: caso senza DF	66
5.1.4 Estensioni: ambiente urbano e mobilità dei nodi	69
5.2 Estensioni: localizzazione GPS di persone disperse	80
5.2.1 Proprietà tipiche dell'ingegneria del software	80
5.2.2 Funzionamento del protocollo di localizzazione GPS	82
Capitolo 6 - Conclusioni	85
Bibliografia	87
Appendice A	89
Appendice B	97
Appendice C	99

Elenco delle figure



Figura 2.1	Schema Link Layer BLE	20
Figura 2.2	Nodi connessi e nodi isolati	22
Figura 2.3	Esempio di Random Geometric Graph	22
Figura 2.4	Metodo Push	23
Figura 2.5	Metodo Pull	24
Figura 2.6	Metodo Push & Pull	24
Figura 2.7	Componenti interni ai nodi	26
Figura 2.8	Vari livelli di Protocol	27
Figura 2.9	Cycle-based e Event-based	27
Figura 2.10	Scheduling guidato dagli eventi	29
Figura 3.1	Formula del Dynamic Fanout	31
Figura 3.2	Formula dell'Advertising Limit	32
Figura 3.3	Schema Link Layer nell'algoritmo Dynamic Fanout	32
Figura 3.4	Procedura di invio messaggio nei nodi Sender (S) e Receiver (R)	41
Figura 4.1	Evoluzione della rete durante una simulazione	46
Figura 4.2	Confronto grafici della copertura della rete	53
Figura 4.3	Confronto grafici del tempo totale di trasmissione	54
Figura 4.4	Confronto grafici del Fattore di Efficienza	55
Figura 5.1	Component Diagram	59
Figura 5.2	Confronto evoluzione della rete con e senza coda di priorità	60
Figura 5.3	Confronto grafici del tempo totale di trasmissione	63
Figura 5.4	Confronto grafici del Fattore di Efficienza	64
Figura 5.5	Confronto grafici della copertura della rete	64
Figura 5.6	Confronto grafici del tempo totale di trasmissione	65
Figura 5.7	Confronto grafici del Fattore di Efficienza	65
Figura 5.8	Confronto grafici della copertura della rete	65
Figura 5.9	Grafico tempi totali con aumento di messaggi	66
Figura 5.10	Confronto grafici del tempo totale di trasmissione	67

| Elenco delle figure

Figura 5.11	Confronto grafici del Fattore di Efficienza	67
Figura 5.12	Confronto grafici della copertura della rete	68
Figura 5.13	Evoluzione della rete in ambiente urbano	69
Figura 5.14	Evoluzione della rete con mobilità dei nodi	76
Figura 5.15	Confronto grafici del tempo totale di trasmissione	78
Figura 5.16	Confronto grafici del Fattore di Efficienza	78
Figura 5.17	Confronto grafici della copertura della rete	78
Figura 5.18	Confronto grafici del tempo totale di trasmissione	79
Figura 5.19	Confronto grafici del Fattore di Efficienza	79
Figura 5.20	Confronto grafici della copertura della rete	79
Figura 5.21	Evoluzione della rete per localizzazione GPS	83

Elenco degli algoritmi

EA

Algoritmo 1	Fixed Fanout	25
Algoritmo 2	InvioMessaggio (Pagliari L.)	36
Algoritmo 3	Invio Messaggio	37
Algoritmo 4	Timer Ricevuto	37
Algoritmo 5	Connection_Request Ricevuto	38
Algoritmo 6	Connection_Msg Ricevuto	39
Algoritmo 7	Unlock Ricevuto	39
Algoritmo 8	Do Advertising	38
Algoritmo 9	Advertising Ricevuto	38
Algoritmo 10	Azioni Periodiche	40
Algoritmo 11	AzioniPeriodiche (Pagliari L.)	40

Introduzione

L'oggetto di studio di questa tesi è sperimentare soluzioni software in grado di sopperire alle difficoltà di comunicazione dei dispositivi mobili in seguito ad eventi atmosferici catastrofici o altre situazioni che provochino indisponibilità delle reti di telecomunicazione, permettendo comunque la diffusione di informazioni [17]. Il tutto ponendo grande attenzione al risparmio energetico. Il lavoro ne amplia uno precedentemente svolto all'interno dello stesso Politecnico [13], [15].

I motivi che ci hanno portato a pensare che riproporre questo precedente lavoro potesse essere di interesse, sono derivati prevalentemente dalla possibilità di utilizzare per la sperimentazione un programma differente, che permettesse una maggiore estendibilità e di conseguenza il raggiungimento di un più alto livello di realismo.

I primi capitoli trattano in modo sintetico del precedente lavoro, di cui consigliamo tuttavia la lettura per avere una visione più chiara e completa dell'argomento in esame.

Successivamente viene mostrata in dettaglio la soluzione proposta nel lavoro precedente: un sistema che cerca di diffondere messaggi in maniera epidemica sfruttando come canale di trasmissione la tecnologia Bluetooth Low Energy. La diffusione dell'informazione è guidata dagli algoritmi di gossip ed il sistema gestisce dinamicamente il carico di lavoro in base alle condizioni esterne e interne del dispositivo. Questa soluzione era stata provata solo su uno specifico simulatore che offriva poca flessibilità nelle sperimentazioni.

Viene quindi illustrato nel dettaglio il programma di simulazione da noi utilizzato, scelto in modo che possa permettere di riprodurre e estendere le precedenti simulazioni.

Per sviluppare la soluzione proposta è stato definito un algoritmo adattativo progettato come estensione di un algoritmo di gossip, che sfrutta le caratteristiche del gossip per diffondere informazioni e grazie alla sua capacità di adattamento, cerca di trovare un compromesso nel definire il carico di lavoro del dispositivo tra autonomia ed efficienza.

Questo algoritmo viene esposto chiaramente e ne è mostrato un esempio di funzionamento.

Viene inoltre illustrato il processo di riprogettazione del precedente algoritmo, che si è visto necessario a causa delle forti differenze tra le strutture dei due simulatori in questione.

Il risultato così ottenuto è un nuovo algoritmo con il medesimo funzionamento

di quello precedente ma che è stato possibile implementare ottenendo un protocollo per il nuovo simulatore.

Nel capitolo successivo vengono mostrati i risultati delle simulazioni, che permettono di analizzare il sistema per diverse situazioni, che tengano conto sia delle densità che della dimensione della popolazione coinvolta. I risultati ottenuti sono confrontati con quelli del lavoro precedente, in modo da verificare se il funzionamento del precedente algoritmo è stato riprodotto correttamente.

Infine vengono analizzate nel dettaglio le varie estensioni del protocollo, che hanno permesso di simulare comportamenti della popolazione in modo più realistico.

In particolare due estensioni hanno permesso di simulare più realisticamente lo schema distributivo dei nuclei abitativi e il movimento delle persone. In ogni città, gli abitanti non sono mai distribuiti uniformemente su tutta l'area sotto la giurisdizione comunale, era quindi necessario migliorare il modo in cui venivano disposti i nodi nella rete. Anche l'aggiunta della mobilità dei nodi è stata importante: dispositivi contagiati che si spostano possono incrementare le prestazioni in termini di copertura della rete, proprio come nei casi delle epidemie. Per questo motivo è interessante vedere l'analisi dei risultati provenienti da simulazioni in cui sono state utilizzate queste estensioni.

Altra estensione esaminata riguarda il comportamento della rete con differenti flussi di messaggio, cioè diversi messaggi generati nello stesso momento da dispositivi differenti.

Vengono poi mostrate estensioni con lo scopo di aumentare o analizzare l'efficienza del protocollo.

L'ultima estensione che viene discussa permette di simulare la localizzazione di persone disperse utilizzando la tecnologia GPS. Questo aggiunge alla nostra simulazione funzionalità diverse da quelle per cui è stata pensata in origine. Il poter scrivere questo nuovo protocollo semplicemente estendendo il precedente mostra il buon grado di riusabilità raggiunto dal codice.

Vediamo inoltre un'analisi delle altre proprietà del nostro codice.

1.1 | Struttura del documento

In questo paragrafo presentiamo il lavoro svolto, organizzato con la seguente struttura:

- ***Capitolo 2 - Stato dell'arte:***

in questo capitolo presentiamo lo stato dell'arte concernente le tecnologie, i modelli, gli strumenti e gli studi su cui questo lavoro si basa. Analizzeremo lo strumento di diffusione scelto e il modello di rete adatto a rappresentare il nostro sistema. Insieme alla Reti peer-to-peer

parleremo degli algoritmi di gossip, e nello specifico di quello scelto come base di partenza per il nostro algoritmo. Tratteremo inoltre in maniera approfondita dello strumento di simulazione utilizzato per il lavoro.

- ***Capitolo 3 - Funzionamento e riprogettazione del protocollo:***
in questo capitolo presenteremo l'algoritmo utilizzato per la soluzione del problema, e il modo in cui è stato riprogettato. Presenteremo nel dettaglio la struttura e il funzionamento dell'algoritmo e del protocollo da esso derivato, vedendone inoltre un esempio di funzionamento. Illustreremo la fase di riprogettazione dell'algoritmo, con le relative scelte implementative, necessaria per poter derivarne un nuovo protocollo che sia funzionante sul simulatore scelto.
- ***Capitolo 4 - Simulazioni e valutazione dei risultati:***
in questo capitolo discuteremo la fase di simulazione e raccolta dati, per poi andare ad analizzare i risultati e usarli per un confronto con quelli ottenuti dal lavoro precedente. Illustreremo inoltre uno strumento da noi creato per permettere di ottenere una visualizzazione grafica dell'evoluzione della rete.
- ***Capitolo 5 - Estensioni:***
in questo capitolo discuteremo di tutte le estensioni del protocollo. Vedremo quelle aggiunte per aumentare il grado di realismo della simulazione, come la mobilità dei nodi e un diverso modo di distribuirli nella rete. Vedremo inoltre l'estensione riguardante la gestione di diversi flussi di messaggio e quelle aggiunte con lo scopo di aumentare o analizzare l'efficienza del protocollo. Mostreremo infine l'estensione che permette di simulare la localizzazione di persone disperse utilizzando la tecnologia GPS. Questa sarà usata come spunto per analizzare le proprietà del nostro codice.
- ***Capitolo 6 - Conclusioni e sviluppi futuri:***
in questo capitolo presenteremo le nostre conclusioni in merito al lavoro svolto e discuteremo delle possibili direzioni future, in particolare di studi da intraprendere su altri protocolli per la diffusione di messaggi sul simulatore da noi utilizzato.

Stato dell'arte

In questo capitolo presentiamo lo stato dell'arte, vale a dire il lavoro precedentemente svolto [13] che è stato punto di partenza di questa tesi.

Per prima cosa è utile mostrare brevemente le tecnologie e i modelli su cui questo lavoro si basa, per poter meglio comprendere anche le nostre scelte successive.

Dietro questo lavoro vi era uno studio approfondito di diversi metodi di gestione delle risorse ottimizzati per la minimizzazione del consumo energetico. Tratteremo quindi i sistemi Self-Adaptative, con un'attenzione particolare al risparmio energetico.

Successivamente parleremo del Bluetooth Low Energy, in particolare degli stati di operatività del *link layer* di fatto la caratteristica con cui questa tecnologia gestisce lo scambio di messaggi.

In seguito parleremo delle reti Peer-to-Peer, e del motivo per cui questa classe sia la più adatta a rappresentare la nostra rete. Inoltre vedremo quale tipo di algoritmo di gossip è stato scelto per modellare la diffusione dei messaggi del sistema, e quale tipo di grafo per la struttura della rete.

Infine presenteremo Peersim¹, lo strumento di simulazione utilizzato. Esporremo le dinamiche di funzionamento del motore del simulatore, che saranno utili anche per comprendere le scelte effettuate durante la fase di progettazione.

Ho scritto una guida all'utilizzo del programma, vedi [Appendice A](#), che può risultare utile anche per rendere di più facile comprensione la lettura del mio codice.

2.1 | Sistemi Self-Adaptative

In letteratura si trovano molti studi riguardanti possibili soluzioni per migliorare la gestione delle risorse e di conseguenza minimizzare lo spreco energetico.

Questi tipi di sistemi, chiamati Self-Adaptative, riescono ad autoregolare parametri interni allo scopo di consumare il minimo quantitativo di energia. Tuttavia in applicazioni reali bisogna tenere conto non solo dei vincoli energetici, ma anche della qualità del servizio (Qos). Quindi queste soluzioni self-adaptative devono trovare un compromesso tra consumo di energia e il Qos.

¹ <http://peersim.sourceforge.net/>

Questo trade-off è il motivo per cui non si sono trovate soluzioni definitive. In letteratura vi sono molti articoli riguardanti sistemi auto adattativi con vincoli di Qos, come [6], [7], [8] e [9]. L'idea di base è di poter gestire il numero di risorse per un determinato servizio, in modo da allocarne il meno possibile garantendo allo stesso tempo una qualità del servizio accettabile.

Questo tipo di adattamento è molto dinamico e dipendente dalla distribuzione delle richieste che vengono fatte per quel particolare servizio.

2.2 | Stati di operatività del link layer del BLE

La tecnologia Bluetooth Low Energy (BLE) [5] è una tecnologia wireless rilasciata a metà 2010.

Si è scelto di adottare questa tecnologia per la comunicazione tra nodi in quanto è volta al risparmio energetico ed è adatta al nostro tipo di rete, come vedremo in seguito.

Di particolare interesse per il nostro sistema è osservare gli stati che descrivono l'operatività del link layer, presi dalle specifiche ufficiali [4].

Essi rappresentano elementi dell'automa a stati finiti che modella il comportamento del link layer.

Andremo ora a descriverli singolarmente, partendo dalla definizione del bluetooth specification.

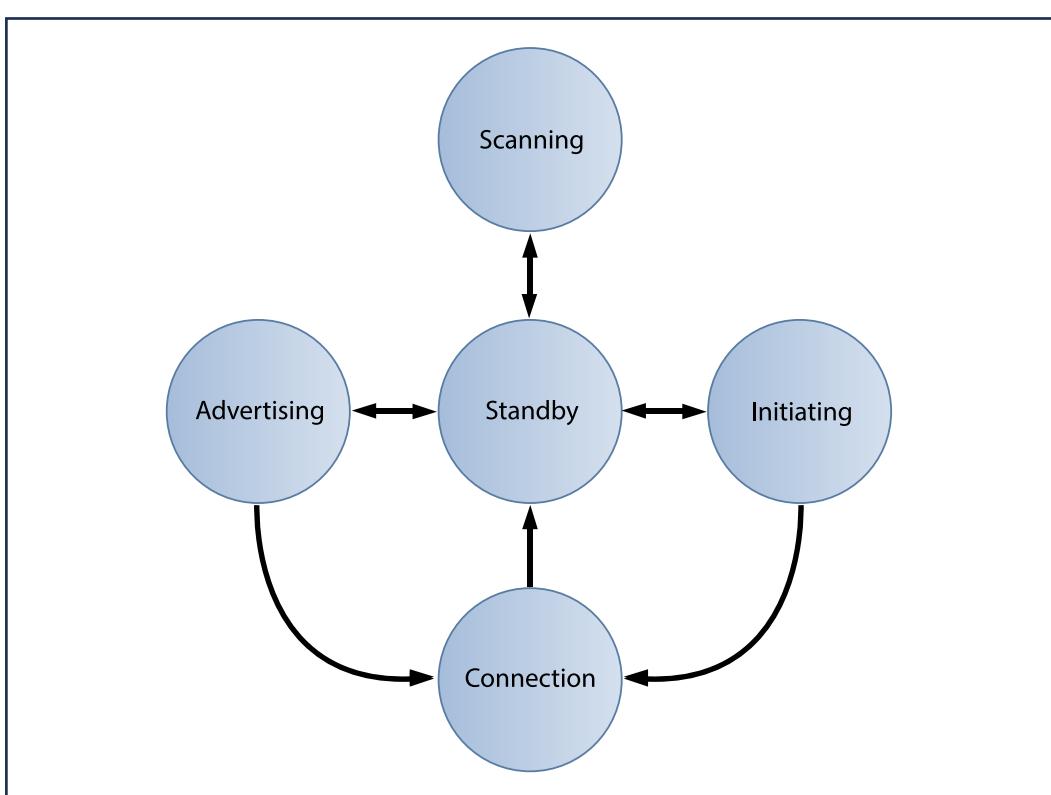


Figura 2.1 - Schema Link Layer BLE

Standby

“Devices cannot send nor receive messages”

I dispositivi che non possono eseguire nessuna trasmissione o ricezione di pacchetti.

Scanning

“Devices that receive advertising packets on advertising channels without the intention to connect to advertising device are referred to as Scanners”

I dispositivi che ascoltano la rete in attesa di pacchetti di interesse.

Initiating

“Devices that need to form a connection to another device listen for connectable packets are referred to as Initiators”

I dispositivi che hanno individuato un messaggio di loro interesse.

Advertising

“Devices that transmit advertising packets on advertising channels are referred to as Advertisers”

I dispositivi che diffondono pacchetti di advertising, con informazioni sui loro messaggi.

Connection

“Devices where the communication of the information to transmit is taking place”

I dispositivi in cui è in corso la trasmissione di un messaggio.

2.3 | Reti Peer-to-Peer

Peer-to-Peer è un modello logico di rete nel quale non vi è una struttura gerarchica e ogni nodo è considerato allo stesso livello di tutti gli altri, potendo fungere al contempo da Client o Server [3].

Senza la presenza di nodi più importanti, nessun nodo può avere una visione completa della rete.

Ogni nodo quindi ha una visione parziale e locale della struttura della rete.

Le topologie di reti di questo tipo sono rappresentate da grafi bidirezionali, in quanto la comunicazione tra due nodi vicini può avvenire in entrambe le direzioni.

Nella [Sezione 2.4](#) tratteremo una di queste, il Random Geometric Graph, quella che meglio si adatta a rappresentare il nostro sistema di cellulari collegati via Bluetooth.

Uno dei vantaggi di questo tipo di rete è il basso costo di implementazione, visto che si evita l'utilizzo di macchine Server, che rappresentano la spesa maggiore nelle altre tipologie di rete.

Altro vantaggio è dato dall'amministrazione decentralizzata, le informazioni non sono gestite da Server (quindi nodi di maggiore importanza) ma sono in

possesso dei singoli nodi, che le metteranno a disposizione della rete, quindi di qualsiasi nodo ne faccia richiesta.

Inoltre si ha spesso ridondanza dei dati, visto che ogni informazione può essere presente in un alto numero di nodi, e questo ne garantisce integrità e buona reperibilità degli stessi.

Tutte queste caratteristiche le ritroviamo anche nel nostro sistema, considerato che ogni cellulare ha lo stesso livello gerarchico nella rete e funge sia da ricevitore che da trasmettitore di informazioni.

Appare quindi evidente il perché questo è stato il modello adottato per la nostra rete.

La principale applicazione di questo modello di rete è rappresentata dal file sharing, per il quale sono nati tanti sistemi quali Napster, e-Mule, la rete Torrent etc

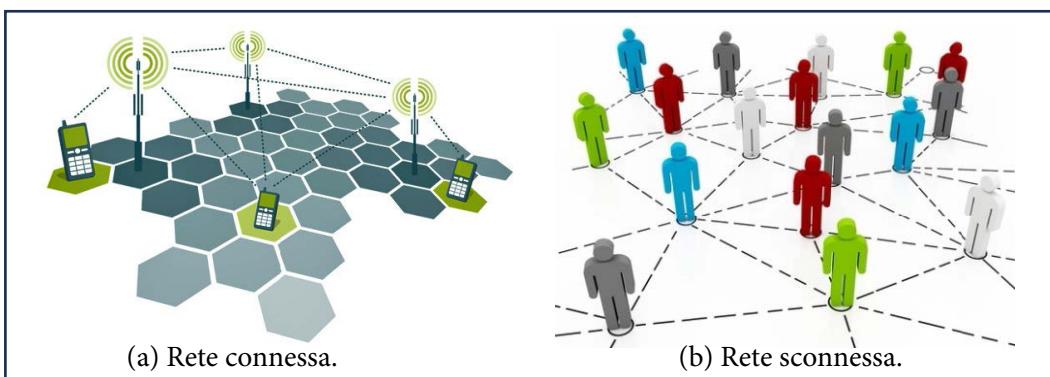


Figura 2.2 - Nodi connessi e nodi isolati

2.4 | Random Geometric Graph

E' un metodo di rappresentazione grafica bidirezionale casuale che genera un grafo i cui N nodi sono disposti in maniera casuale ed uniforme in un'area limitata. Due nodi poi possono essere connessi tra di loro se si trovano ad una distanza minore di un dato ρ . Reti di questo tipo sono estremamente adatte alla rappresentazione di reti wireless, caratterizzate dalla distanza fisica tra i nodi e da un valore soglia ρ entro il quale è possibile eseguire trasmissioni tra essi.

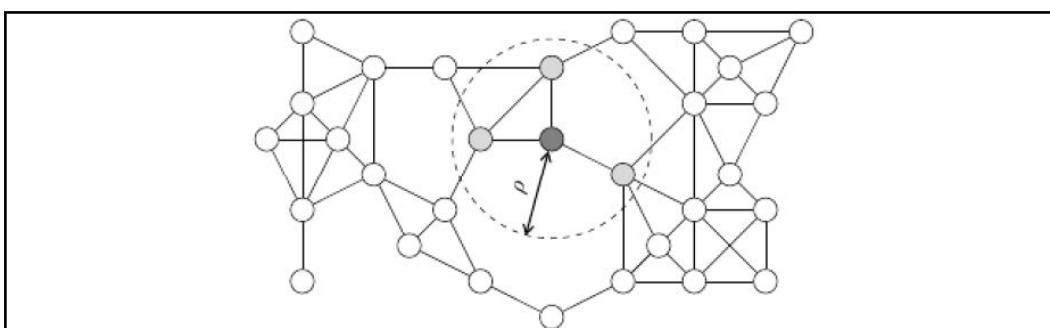


Figura 2.3 - Esempio di Random Geometric Graph [12]

2.5 | Metodi di diffusione degli algoritmi di gossip

Una rete peer-to-peer è per sua natura dinamica e disordinata, infatti i nodi possono connettersi e disconnettersi in ogni momento. Di conseguenza è importante lo studio di algoritmi in grado di massimizzare l'efficienza nella diffusione delle informazioni.

Gli algoritmi di gossip, detti anche algoritmi epidemici, si ispirano a fenomeni propri del mondo naturale o di quello sociale. Infatti, come suggeriscono i nomi, il loro comportamento ricorda il diffondersi delle malattie o di un pettegolezzo all'interno di un gruppo sociale.

In base agli stati che possono assumere i nodi sono stati definiti alcuni modelli. Di seguito illustreremo brevemente il modello scelto, che fa parte di una delle categorie in cui possiamo suddividere gli algoritmi di gossip, in base al comportamento dei nodi [18], [19].

Il modello Suscettibile - Infected - Removed [1] ci dice che ad ogni iterazione di un algoritmo tutte le unità della popolazione devono comunicare con un nodo scelto in maniera casuale. Vi si riscontrano tre metodi attraverso i quali le unità di popolazione possono scambiare informazioni:

- **Metodo Push**, prevede che i nodi contagiati prendano l'iniziativa di diffondere l'informazione autonomamente ovvero, ad ogni istante t il nodo contagiato sceglie un nodo casuale da contagiare. Questa strategia è molto efficace all'inizio della diffusione, quando vi è un alto numero di unità suscettibili e poche contagiate o rimosse.

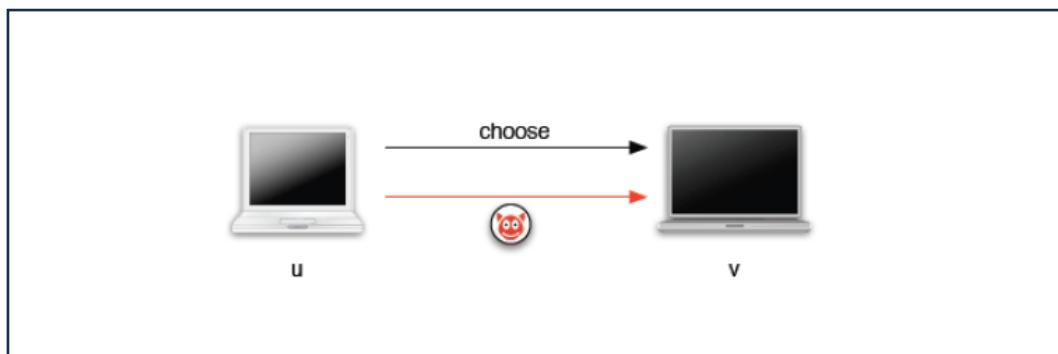


Figura 2.4 - Metodo Push [1]

- **Metodo Pull**, prevede che i nodi contagiati non si muovano attivamente nel diffondere l'informazione, ma invece che siano i nodi suscettibili a fare richiesta di nuove informazioni ai nodi contagiati; ovvero ad ogni istante t , un nodo suscettibile seleziona casualmente un altro nodo e gli chiede se ha nuove informazioni. Se il nodo contattato ne ha, allora restituisce l'informazione. Questo metodo non garantisce che il processo di diffusione abbia inizio poiché vi è una probabilità che nessun nodo suscettibile contatti il nodo contagiato, ma vi è tuttavia anche la possibilità che tutti i nodi scelgano il nodo contagiato generando così un rapido inizio di epidemia.

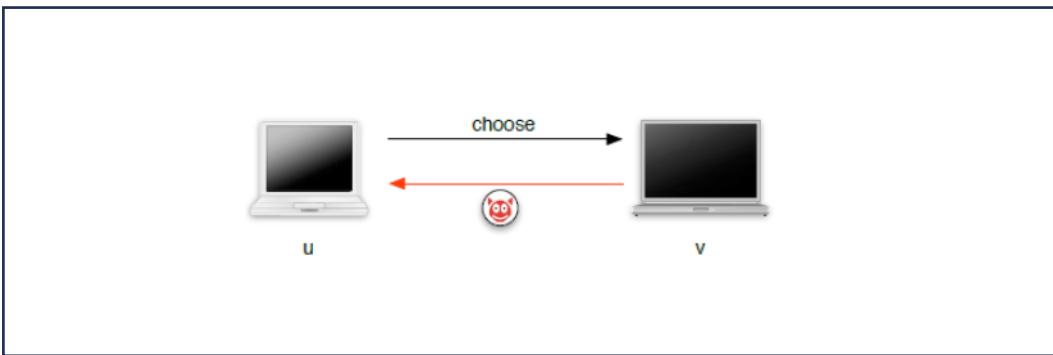


Figura 2.5 - Metodo Pull [1]

- **Metodo Push&Pull**, ha lo scopo di combinare i vantaggi dei metodi precedenti. Per questi metodi infatti, i nodi contagiati utilizzano una strategia Push, mentre i nodi suscettibili una strategia Pull.

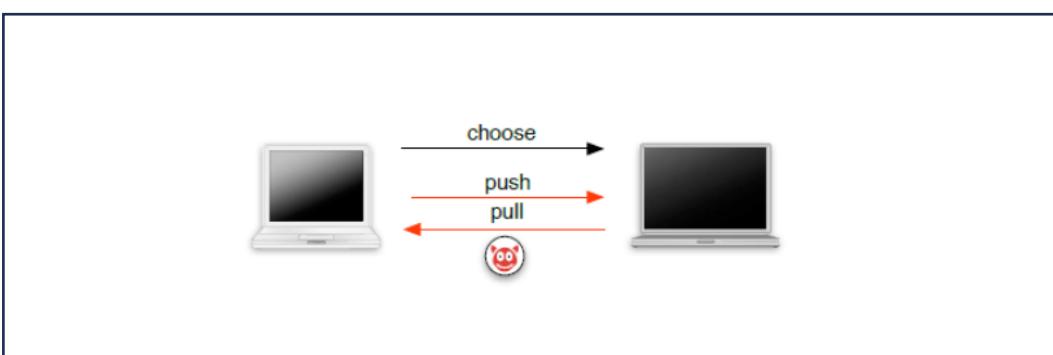


Figura 2.6 - Metodo Push&Pull [1]

2.6 | Fixed Fanout

Tra gli algoritmi di tipo Push quello scelto, il *Fixed Fanout* [12], prende il nome dal suo parametro più importante il *Fanout*. Quest'ultimo rappresenta il numero di nodi a cui diffondere il messaggio. L'algoritmo prevede che ogni nodo che ha l'informazione, o l'ha ricevuta, la diffonda ad un numero di nodi adiacenti, dato dal valore del *Fanout*, scelti a caso tra tutti i nodi vicini. Il valore del *Fanout* è definito all'inizio e resta costante per tutta l'esecuzione.

Di seguito viene mostrato l'algoritmo nel dettaglio.

Algorithm 1 Fixed Fanout

```

1: function GOSSIPPE(msg,fanout)
2:   if fanout  $\geq V_i$  then
3:     toSend  $\leftarrow \Lambda_i$ 
4:   else
5:     toSend  $\leftarrow \emptyset$ 
6:     for f = 1 to fanout do
7:       random select sj  $\in \Lambda_i / toSend$ 
8:       toSend  $\leftarrow toSend \cup s_j$ 
9:     end for
10:   end if
11:   for all sj  $\in toSend$  do
12:     Send(msg,sj)
13:   end for
14: end function

```

2.7 | Peersim

Peersim è una libreria Java sviluppata in ambiente accademico che consente di simulare reti con un'alta flessibilità sul numero di nodi, quindi il simulatore è adatto per reti di grandi dimensioni. In letteratura troviamo diverse trattazioni in merito come [2], [10] e [11].

La struttura del simulatore si basa su componenti e semplifica la rapida prototipazione di un protocollo. Ogni classe Java corrisponde ad un singolo componente o a un componente aggregato, composto a sua volta da componenti singoli. In questo modo è possibile costruire reti su più livelli, e poter fare analisi da diversi punti di vista e diverse granularità.

Non è presente alcun tipo di strumento grafico per la rappresentazione della rete o dei messaggi tra i nodi, la simulazione è solo computazione.

Durante il run del programma, per monitorare l'andamento di determinate variabili (quelle utili da osservare) alla periodicità necessaria, vanno usati appositi componenti, chiamati Observer.

Essi potranno essere usati per stampare valori sulla consolle ma vi è anche la

possibilità di usarli per salvare valori di interesse su file .dat, così da tenerne traccia. In questo modo li si avrà a disposizione anche per successive analisi. Saranno inoltre visualizzabili graficamente utilizzando un programma esterno, Gnuplot³, che permette di stampare a monitor grafici partendo da dati.

PeerSim è stato progettato per incoraggiare la programmazione modulare basata su oggetti. Ogni blocco è facilmente sostituibile da un altro componente che implementa la stessa interfaccia (vale a dire, la stessa funzionalità).

Le principali tipologie di componenti sono i *Protocol*, classi che implementano l'interfaccia *Protocol*, e i *Control*, classi che implementano l'interfaccia *Control*. La prima tipologia chiaramente tratta di protocolli, quello/i su cui si vuole sperimentare e quelli ausiliari ad esso, mentre la seconda di controllori che osservano i protocolli o/e agiscono modificando lo stato di questi ultimi.

Entrambe le tipologie di componenti hanno due caratteristiche importanti.

La prima, fondamentale, è la periodicità con cui si attivano, ad ogni componente ne può essere assegnata una differente.

La seconda riguarda con quali componenti sono collegati (quindi per i *Protocol* la scelta del livello).

Implica a quali dati e risorse di altri componenti hanno accesso, quindi cosa possono modificare e cosa possono sfruttare. Da ricordare che il collegamento è unidirezionale.

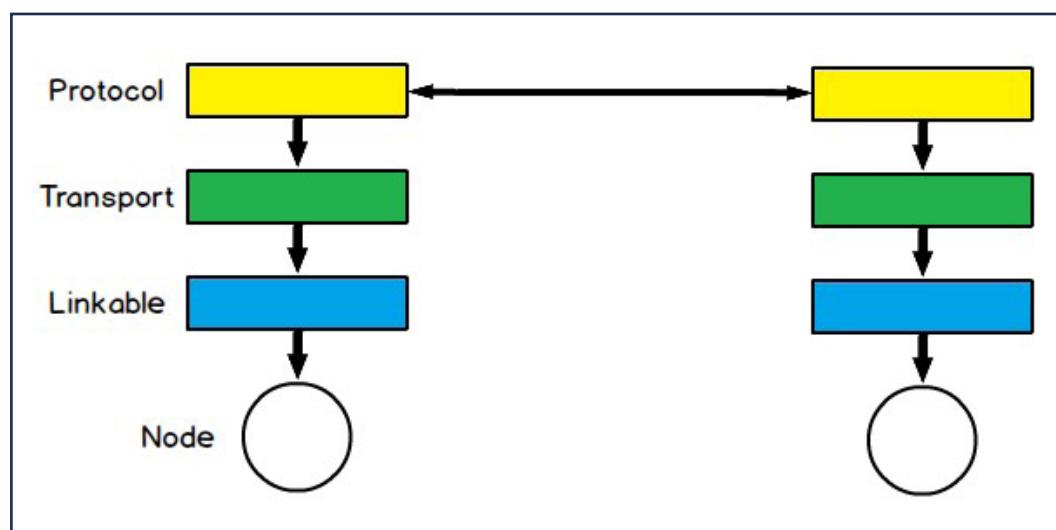


Figura 2.7 - Componenti interni ai nodi

La differenza fondamentale sta nel modo in cui essi vedono la rete; i *Control* vedono tutti i nodi e la rete nella sua totalità, quindi hanno accesso a tutte le informazioni di quest'ultima; i *Protocol* sono contenuti in ogni nodo, possono agire sul nodo stesso e sui soli nodi vicini di cui hanno visione.

I *Protocol* inoltre, dentro il singolo nodo, possono essere disposti su più livelli, formando una pila, in modo del tutto analogo a quella della rappresentazione ISO/OSI o TCP/IP per esempio.

Vi è anche la possibilità di mettere più protocolli, in parallelo, nello stesso livello.

Ogni istanza di un protocollo contenuta in un nodo avrà accesso ai dati e alle risorse delle istanze degli altri protocolli di livello minore o uguale, contenuti nel nodo stesso, e delle istanze dello stesso protocollo contenute negli altri nodi.

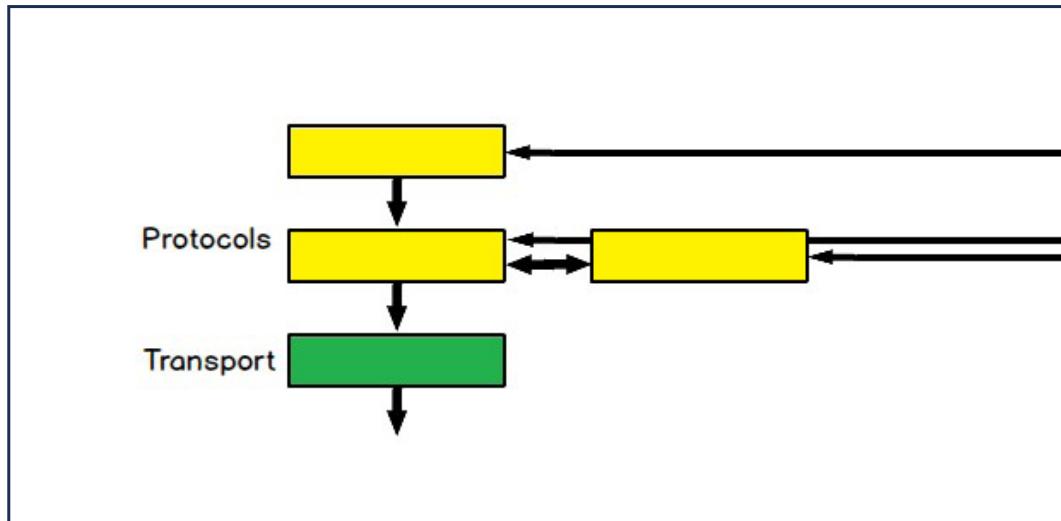


Figura 2.8 - Vari livelli di Protocol

PeerSim supporta due modelli di simulazione: il modello event-based e il modello cycle-based. Quest'ultimo modello è semplificato, il che rende possibile raggiungere un'estrema scalabilità e prestazioni, al costo di alcune perdite di realismo. Tuttavia molti protocolli semplici possono tollerare questa perdita senza problemi.

Le ipotesi semplificative del modello cycle-based sono la mancanza di simulazione del livello di trasporto e la mancanza di concorrenza. In altre parole, i nodi comunicano direttamente tra loro, e ai nodi viene dato il controllo periodicamente, in ordine sequenziale, in modo che possano eseguire azioni arbitrarie.

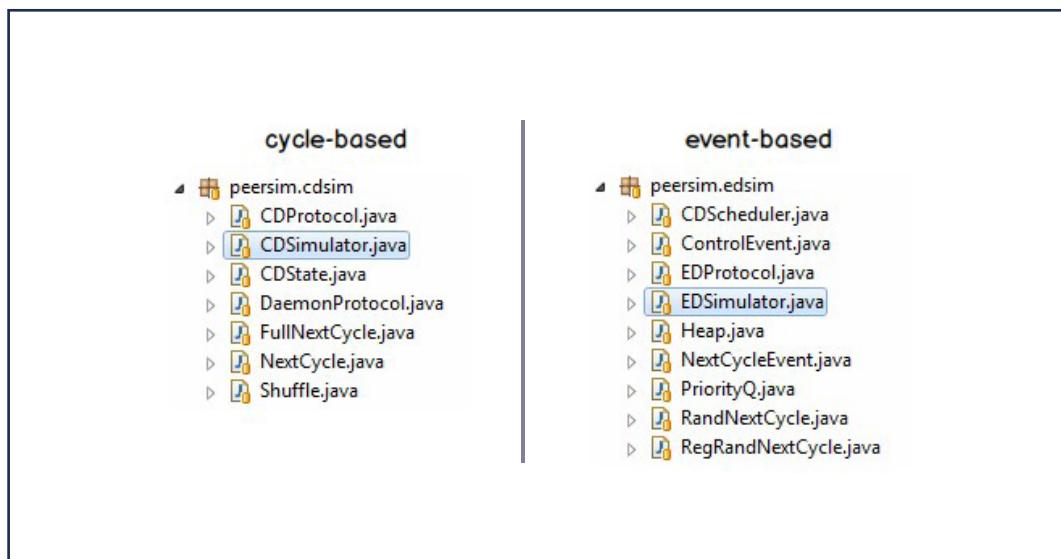


Figura 2.9 - Cycle-based e Event-based

La differenza fondamentale tra i due modelli è la diversa visione del tempo; in cycle-based viene dato in “cicli di esecuzione” mentre in event-based in valori di tempo, quantità di “unità di tempo”.

Una volta scelta l’unità di misura del tempo (quella di `simulation.endtime`) si dovrà rimanere coerenti con essa ogni volta che si avrà necessità di inserire o manipolare un valore di tempo.

L’idea generale del modello di simulazione è:

- scegliere la dimensione della rete (numero di nodi)
- scegliere uno o più protocolli da sperimentare, e inizializzarli
- scegliere uno o più oggetti *Control* per monitorare le proprietà a cui si è interessati e modificare alcuni parametri durante la simulazione (ad esempio, la dimensione della rete, lo stato interno dei protocolli, ecc.)
- eseguire la simulazione invocando la classe `Simulator` con un file di configurazione, che contiene le informazioni di cui sopra

Il ciclo di vita di una simulazione è il seguente. Il primo passo è leggere il file di configurazione, dato come parametro da riga di comando o dall’Editor. La configurazione contiene tutti i parametri di simulazione relativi a tutti gli oggetti coinvolti nell’esperimento.

Quindi il simulatore imposta la rete inizializzando i nodi nella rete e i protocolli in essi contenuti. Ogni nodo ha gli stessi tipi di protocolli; cioè le istanze di un protocollo formano un array nella rete, con un’istanza in ciascun nodo. Le istanze dei nodi e dei protocolli sono create tramite clonazione. Viene generata una sola istanza usando il costruttore dell’oggetto, che funge da prototipo, e tutti i nodi della rete sono clonati da questo prototipo. Per questo motivo, è molto importante prestare attenzione durante l’implementazione del metodo di clonazione dei protocolli.

Se nei nodi e nei protocolli sono presenti variabili di tipi non primitivi va fatto con deep clone.

A questo punto, è necessario eseguire l’inizializzazione, che imposta gli stati iniziali di ciascun protocollo. La fase di inizializzazione viene eseguita dai *Control* pianificati per l’esecuzione solo all’inizio di ogni esperimento. Nel file di configurazione i componenti di inizializzazione sono facilmente riconoscibili dal prefisso `init`. Questi oggetti `initializer` sono semplicemente *Control*, configurati per l’esecuzione nella fase di inizializzazione.

Dopo l’inizializzazione il motore richiama tutti i componenti *Protocol* e *Control* una volta in ogni ciclo, fino a un determinato numero di cicli o fino a quando un componente decide di terminare la simulazione. In Peersim tutti gli oggetti *Protocol* e *Control* sono assegnati a un oggetto *Scheduler* che definisce quando verranno eseguiti esattamente. Per impostazione predefinita, tutti gli oggetti vengono eseguiti in ogni ciclo. Tuttavia è possibile configurare un protocollo o un controllo per l’esecuzione solo in determinati cicli ed è anche possibile scegliere l’ordine di esecuzione dei componenti all’interno di ciascun ciclo.

Nel modello event-based tutto funziona esattamente nello stesso modo del modello cycle-based, eccetto la gestione del tempo e il modo in cui ai protocolli viene passato il controllo. *Protocols* che non sono eseguibili (usati solo per memorizzare dati) possono essere applicati e inizializzati esattamente nello stesso modo. È possibile utilizzare anche i *Controls* di qualsiasi package esterno al package *peersim.cdsim*. Per impostazione predefinita, i *Controls* vengono chiamati in ogni ciclo nel modello cycle-based. Nel modello event-based devono essere schedulati esplicitamente.

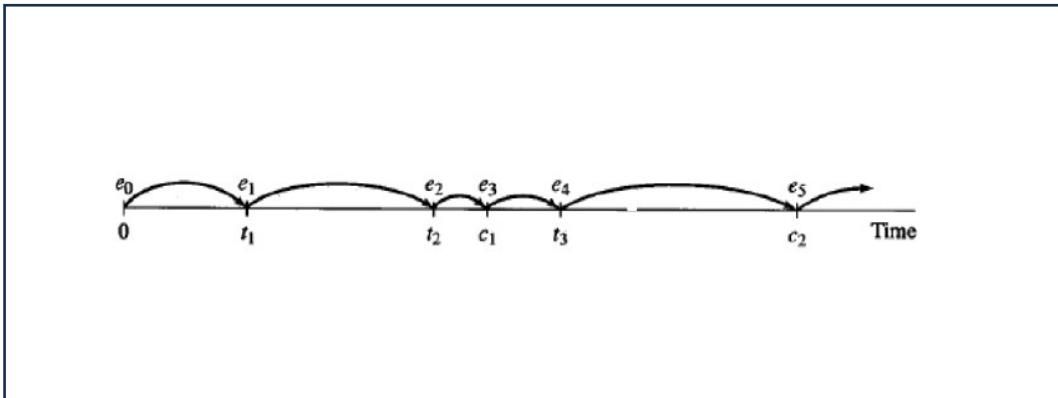


Figura 2.10 - Scheduling guidato dagli eventi

Funzionamento e riprogettazione del protocollo

3.1. | Introduzione

Per il protocollo vero e proprio è stato scelto di usare come base l'algoritmo *Fixed Fanout*, a sua volta esteso aggiungendo parametri dinamici in modo da renderlo proprio di logiche tipiche dei sistemi self-adaptative. L'obiettivo era quindi dargli capacità di adattamento volte alla minimizzazione del consumo energetico del singolo dispositivo.

In particolare vengono utilizzati dinamicamente parametri quali il livello di carica della batteria e il numero di nodi vicini percepiti dal dispositivo.

In questo modo si riesce a calibrare il carico di lavoro tra i dispositivi garantendo risparmio energetico da un lato e prestazioni accettabili dall'altro. Ovvero se il livello di batteria del dispositivo non è elevato il protocollo comincia a calibrare i parametri in modo da trovare un compromesso di carico di lavoro ed efficienza che non gravi eccessivamente sulla restante autonomia. Dato che non si tratta di un sistema dedicato, non è pensato per occupare completamente le risorse del dispositivo privando l'utente degli eventuali altri servizi che possano servirgli.

Per questo il protocollo dopo che la batteria scende sotto una certa soglia limite, pone il sistema in stato di Standby per non intaccare la restante autonomia residua.

I parametri che lo differenziano dal *Fixed Fanout* sono il *Dynamic Fanout* e l'*Advertising Limit*.

Il *Dynamic Fanout* rappresenta il limite alle trasmissioni, mentre l'*Advertising Limit* rappresenta il limite agli Advertising Event a vuoto consecutivi.

Nel momento di inviare una nuova informazione, il protocollo sceglie un nodo casuale dall'insieme dei suoi vicini e tenta di trasferirgli l'informazione. Se il trasferimento va a buon fine, incrementa un contatore di uno. Ripete quest'operazione fino a quanto il conteggio raggiunge un valore limite del *Dynamic Fanout*. Il sistema esegue esattamente DF trasferimenti e poi cambia stato.

$$DF = \begin{cases} 1 + \left(\frac{\sqrt{0,2 \cdot z - 1,9}}{10} \right) \cdot x - 0,0000004x^4 & , x < X_{\max} \\ \max \left(1 + \left(\frac{\sqrt{0,2 \cdot z - 1,9}}{10} \right) \cdot x - 0,0000004x^4; 1 + \frac{1}{2}DF_{\max} \right) & , x \geqslant X_{\max} \end{cases}$$

Figura 3.1 - Formula del Dynamic Fanout

$$\text{AL} = \ln(2x) + 1$$

Figura 3.2 - Formula dell'Advertising Limit

3.1.1 | Stati del link layer

Il protocollo prevede che ogni nodo si trovi in uno dei 5 stati che di seguito andremo a illustrare, ispirati agli stati di operatività del link layer del Bluetooth BLE precedentemente visti [4].

A seconda dello stato in cui si trova il nodo il protocollo si comporterà in maniera diversa.

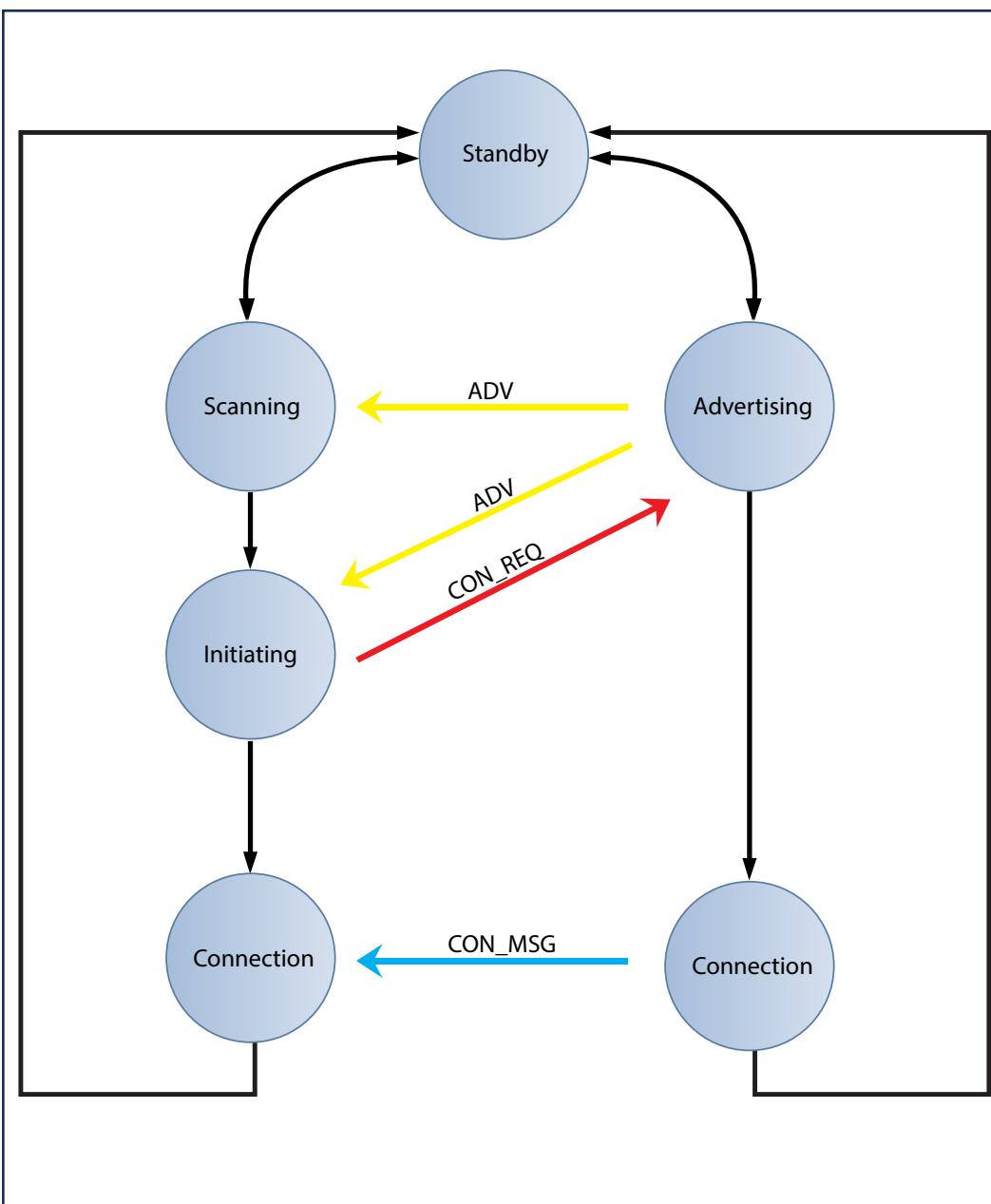


Figura 3.3 - Schema Link Layer nell'algoritmo Dynamic Fanout

0 Standby

In questo stato non è possibile nessuna trasmissione o ricezione di pacchetti.

1 Scanning

I nodi in questo stato ascoltano la rete in attesa di ricevere pacchetti di *advertising*. Si trovano in questo stato tutti i nodi che non sono in attesa di inviare un messaggio o di riceverlo.

Questo secondo caso è spiegato dal fatto che il nodo non ha ricevuto pacchetti di *advertising* di messaggi di cui non è già in possesso. Il nodo risponderà invece a quei pacchetti di *advertising* ai quali è interessato con un pacchetto di *connection_request*.

2 Initiating

I nodi passano in questo stato dopo aver mandato un pacchetto di *connection_request*.

Essi sono quindi in attesa che venga aperta una connessione dal nodo da cui devono ricevere il messaggio, ma continuano a rimanere in grado di ricevere altri pacchetti di *advertising*.

3 Advertising

I nodi sono in questo stato quando sono in attesa di inviare un messaggio. Mandano quindi a tutti i nodi vicini pacchetti di *advertising* e aspettano un pacchetto di *connection_request*. Appena ne ricevono uno apriranno la connessione col nodo corrispondente.

4 Connection

I nodi sono in questo stato durante l'effettivo invio o ricezione di un messaggio. I nodi che arrivano in questo stato da quello di Advertising sono quelli che stanno inviando un messaggio, mentre quelli che ci arrivano da quello di Initiating sono quelli che lo stanno ricevendo.

3.1.2 | Funzionamento effettivo del protocollo

La spiegazione completa ed esaustiva del funzionamento del protocollo è presente sempre nella tesi a cui faccio riferimento [13]. Ritengo tuttavia che mostrare un esempio di esecuzione del protocollo possa essere utile a dare buona comprensione al lettore.

Nella situazione di partenza tutti i nodi sono in stato di Standby per poi passare immediatamente (nel caso avessero batteria sufficiente) in stato di Scanning per aspettare nuovi messaggi.

Quando un dispositivo genera un messaggio, il corrispondente nodo che lo rappresenta passa in stato di Advertising e fa partire la procedura di *invio messaggio*. Sono quindi inizializzati i due contatori facenti riferimento al DF e al AL. Il contatore di *advertise* viene incrementato di 1 e vengono quindi mandati pacchetti di *advertising* a tutti i nodi vicini (neighbors)

indipendentemente dal loro stato. Dopo di che il nodo si mette in ascolto e viene fatto partire un timer.

Se il timer scade senza che nessuna richiesta *connection_request* è arrivata viene incrementato nuovamente il contatore di advertise con conseguente nuovo invio di pacchetti di *advertising*.

Questo fino a quando il contatore di advertise non arriva a AL, ovvero il limite di *advertising* che possono andare a vuoto consecutivamente.

Se invece arriva un pacchetto di *connection_request* da parte di un nodo, che è quindi in stato di Initiating, viene aperta una connessione. Entrambi i nodi, quello inviante e quello ricevente sono segnati occupati e messi in stato di Connection. A questo punto avviene quindi l'invio vero e proprio del messaggio.

Terminata la connessione, viene incrementato di 1 il contatore di trasmissioni effettuate e viene azzerato quello di advertise, perché in questo caso non è andato a vuoto.

Viene decrementata la batteria e controllato se il contatore di trasmissioni non è arrivato a DF.

In caso negativo viene eseguita nuovamente tutta la procedura di *advertising*. Se invece uno dei due contatori ha raggiunto il valore soglia, la procedura di invio messaggio termina, il nodo è riportato in stato di Scanning e viene eseguita la procedura *AzioniPeriodiche*.

Questa procedura di *invio messaggio* viene eseguita anche quando il nodo riceve un nuovo messaggio, che viene trattato alla stregua del caso di messaggio generato dal dispositivo stesso.

Invece quando un nodo è in stato di Scanning (o anche in stato di Initiating) e riceve un pacchetto di *advertising* che gli segnala un messaggio non ancora ricevuto, passa in stato di Initiating (nel caso non lo fosse già) e manda un pacchetto di *connection_request* al nodo corrispondente.

A questo punto, come detto in precedenza, è il nodo advertiser che deve aprire la connessione.

Nel frattempo il nodo in stato di Initiating può continuare a ricevere pacchetti di *advertising* da altri nodi e inviare a sua volta pacchetti di *connection_request*.

Parallelamente viene eseguita la procedura *AzioniPeriodiche*, sia al termine di ogni procedura invio messaggio, sia periodicamente ogni 30 secondi, a patto che il nodo non sia segnato occupato.

Vengono aggiornati i valori dei DF e del AL e controllato il livello di batteria. Se quest'ultimo è sceso sotto il valore soglia, il nodo viene messo in stato di Standby, e verrà riportato in stato di Scanning solo quando sarà rilevato un livello di batteria nuovamente sufficiente.

3.2 | Progettazione del codice

La fase di implementazione del protocollo su Peersim non è stata immediata. Prima è servita una fase preliminare, dato che i due simulatori funzionano con logiche differenti.

Avendo a disposizione [13] e il suo codice [14], ho dovuto astrarne le specifiche. Successivamente descriverò questo processo, andando anche a trattare la fase di sviluppo vero e proprio sul simulatore.

In altre parole ho dovuto per prima cosa prendere confidenza sia con le meccaniche di Peersim sia con il funzionamento del protocollo vero e proprio, entrambi descritti nei capitoli precedenti, per poi poter essere in grado di progettare il codice che poi avrei effettivamente implementato.

Oltre all'uso di due diversi linguaggi di programmazione, di fatto non un grosso problema, le principali differenze riguardano gli strumenti messi a disposizione da Omnet++² che il motore di Peersim, puramente ad eventi, non offre (ad esempio i timer e i wait per attesa di connessione).

È stato quindi necessario riprogettare, tramite un processo di re-engineering, i metodi principali del codice originale in modo da eliminarli da dipendenze legate a questi strumenti.

L'obiettivo era quindi scrivere codice che mimasse il medesimo funzionamento dell'originale nonostante ne fosse molto diverso a livello sintattico.

Per prima cosa ho sostituito i timer con eventi della stessa periodicità, ma essi non sono provvisti di un meccanismo di timer.stop, perché un evento che è schedulato non può essere cancellato.

La soluzione che ho adottato è stata quella di dotare ogni evento Timer di un seriale univoco, così da poterli invalidare singolarmente. In pratica ho provvisto il protocollo di un array che memorizzasse tutti i seriali invalidati. In questo modo ogni volta che si verifica un evento Timer è sufficiente controllare se sia "valido" per decidere se eseguire effettivamente la parte di codice che si occupa appunto della gestione di questo tipo di evento.

Per quanto riguarda l'altra differenza, è stato necessario "spezzare" tutti i metodi che al loro interno contengono costrutti che prevedono un'attesa di connessione bloccando l'esecuzione.

Questo ha portato a ottenere differenti eventi partendo da un singolo metodo.

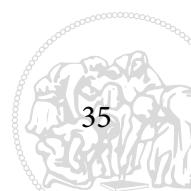
Di seguito verranno illustrati nel dettaglio, così da mostrare come si è raggiunto a livello logico lo stesso risultato del protocollo originale.

3.2.1 | Analisi della struttura del codice

Come possiamo vedere nel codice originale, vedi Algorithm 2, in unico metodo viene:

- inizializzato il contatore TX e inizializzato AE

² <https://www.omnetpp.org/>



- incrementato AE
- inviato pacchetto di *Advertising* a tutti i nodi vicini
- fatto partire un timer e bloccata l'esecuzione
- testato se AE ha raggiunto il limite massimo, ovvero il valore dell'*Advertising Limit*

Algorithm 2 Invio Messaggio

```

1: function INVIOMESSAGGIO(btState, msg)
2:   if btState ≠ STANDBY then
3:     btState ← STANDBY
4:   end if
5:   Tx counter ← 0      ▷ Contatore trasmissioni effettuate.
   START:
6:   AE counter ← 0      ▷ Contatore degli Advertising Event.
7:   btState ← ADVERTISING
8:   AEcounter ← +1
9:   begin
10:    ADVERTISING EVENT
11:   end
12:   if èTimeoutScaduto() then
13:     if AE counter < AL then ▷ AL = Advertising Limit.
14:       go to START
15:     else
16:       btState ← STANDBY
17:       AzioniPeriodiche(btState)
18:     end if
19:   end if
20:   busy ← VERO
21:   btState ← CONNECTION_SLAVE
22:   begin
23:    CONNECTION EVENT
24:   end
25:   Tx counter ← +1
26:   decrementaBatteriaTx()
27:   busy ← FALSO
28:   btState ← STANDBY
29:   if Tx counter < DF then
30:     go to START
31:   else
32:     AzioniPeriodiche(btState)
33:   end if
34: end function

```

se si riceve un *connection_request* viene:

- fatto il lock del nodo (segnato come occupato)
- inviato effettivamente il messaggio
- incrementato TX



- decrementata la batteria
- fatto l'unlock del nodo (segnato come libero)
- testato se TX ha raggiunto il limite massimo, ovvero il valore del *Dynamic Fanout*

Partendo da queste specifiche ho dovuto riorganizzare la struttura del mio codice, per le ragioni spiegate in precedenza, in modo che ogni azione fosse eseguita in ricezione di un evento.

L'avvio della procedura di *invio messaggio* è identico per entrambi gli algoritmi, con la differenza che nel mio Algorithm 3 vengono inizializzati i contatori TX e AE senza eseguire altre azioni, tranne quella di far partire un evento che verrà immediatamente ricevuto.

Algorithm 3 Invio Messaggio

```

1: procedure SENDMESSAGE
2:   new cookie(readDF(), readAL(), 0, 0)
3:   new timerIdMsg(cookie)
4:   timerValid ← 0
5: end procedure

```

Tale evento è quello che rappresenta lo scadere del Timer, quindi ogni volta che esso viene ricevuto verrà gestito seguendo la procedura Algorithm 4, cioè come fosse un Timer scaduto.

Algorithm 4 Timer Ricevuto

```

1: procedure EVENT_TIMERIDMSG
2:   if timerIdMsg.valid == timerValid AND btstate != 0 then
3:     if cookie.ae < cookie.AL AND cookie.tx < cookie.DF then
4:       new timerIdMsg(cookie.DF, cookie.AL, cookie.tx, cookie.ae++)
5:       btstate ← 3
6:       doAdvertising(cookie)
7:     else
8:       timerValid ← 0
9:       btstate ← 0
10:      periodicActions()
11:    end if
12:   end if
13: end procedure

```

Normalmente ogni Timer dura 5 secondi, ma il primo che viene fatto partire, come nel caso appena descritto, viene impostato con scadenza 0 secondi, quindi immediatamente gestito.

Ricevuto l'evento Timer scaduto viene:

- testato se il Timer è ancora valido, cioè se non è stato cancellato
- testato se AE ha raggiunto il limite massimo, ovvero il valore dell'*Advertising Limit*

- testato se TX ha raggiunto il limite massimo, ovvero il valore del *Dynamic Fanout*
- incrementato AE
- inviato pacchetto di *Advertising* a tutti i nodi vicini
- generato un nuovo evento Timer, Algorithm 4, con periodicità 5 secondi

l'invio dei pacchetti di *Advertising* è gestito dalla procedura Algorithm 8. Quando un nodo riceve un evento *advertising*, viene gestito seguendo la

Algorithm 8 Do Advertising

```

1: procedure DOADVERTISING
2:   for  $i = 0; i < \text{linkable.degree}(); i++$  do
3:     new adv(linkable.neighbour( $i$ ), cookie)
4:   end for
5: end procedure

```

procedura Algorithm 9, che consiste nel controllare se il nodo è nello stato adatto ed è interessato al messaggio. Nel caso entrambe queste condizioni siano verificate il nodo invia un evento *connection_request* al mittente.

Algorithm 9 Advertising Ricevuto

```

1: procedure EVENT ADV
2:   if !messages.contains(msg) AND (  $btstate == 1$  OR  $btstate == 2$  )
   then
3:      $btstate \leftarrow 2$ 
4:     new con_req(cookie)
5:   end if
6: end procedure

```

Quando un nodo, dopo la procedura Algorithm 8, riceve un evento *connection_request* in tempo utile (Timer non ancora scaduto) questo è gestito con la procedura Algorithm 5.

In questa procedura viene:

Algorithm 5 Connection_Request Ricevuto

```

1: procedure EVENT CON_REQ
2:   if !busy AND !rq.busy AND cookie.tx < cookie.DF then
3:     timerValid++
4:     busy  $\leftarrow$  true
5:     rq.busy  $\leftarrow$  true
6:     new con_msg()
7:     new unlock(cookie)
8:   end if
9: end procedure

```

- testato se TX ha raggiunto il limite massimo, ovvero il valore del *Dynamic Fanout*



- testato se uno dei due nodi, mittente o destinatario, è occupato
- invalidato il/i Timer creato/i in precedenza
- fatto il lock dei due nodi
- inviato il messaggio al nodo destinatario, come evento *connection_Msg*
- generato un nuovo evento *Unlock* con periodicità uguale al tempo di invio messaggio

A questo punto il nodo destinatario riceve l'evento *connection_Msg* che gestisce seguendo la procedura Algorithm 6, in cui viene:

- testato se il nodo è occupato
- salvato il messaggio appena ricevuto
- fatto l'unlock del nodo
- decrementata la batteria
- fatta partire a sua volta la procedura di *invio messaggio*

Algorithm 6 Connection_Msg Ricevuto

```

1: procedure EVENT CON_MSG
2:   if !busy then
3:     messages.put(msg)
4:     busy ← false
5:     battery -= 3
6:     btstate ← 3
7:     sendMessage()
8:   end if
9: end procedure

```

Il nodo mittente, rimasto in stato di lock, trascorso il tempo necessario per l'invio messaggio, riceve l'evento *Unlock* in contemporanea con la ricezione dell'evento *connection_Msg* del destinatario.

Questo evento è gestito con la procedura Algorithm 7, in cui viene:

- fatto l'unlock del nodo
- decrementata la batteria
- incrementato TX e azzerato AE
- generato un nuovo evento Timer, Algorithm 4, con periodicità 5 secondi

Algorithm 7 Unlock Ricevuto

```

1: procedure EVENT UNLOCK
2:   busy ← false
3:   battery -= 3
4:   new timerIdMsg(cookie.DF, cookie.AL, cookie.tx++, 0)
5: end procedure

```

Tutta la procedura di *invio messaggio* appena descritta è ben mostrata dal diagramma [Figura 3.4](#).

Infine il metodo Algorithm 11 del codice originale è stato leggermente riadattato in Algorithm 10.

Algorithm 10 Azioni Periodiche

```

1: procedure PERIODIC ACTIONS
2:   battery --
3:   if btstate != 4 AND !busy then
4:     if battery > LOWLIMIT then
5:       parametersUpdate()
6:       if btstate == 0 then
7:         btstate ← 1
8:       end if
9:     else
10:    btstate ← 0
11:   end if
12: end if
13: end procedure
```

Algorithm 11 Azioni Periodiche

```

1: function AZIONI PERIODICHE(btState)
2:   decrementaBatteriaIdle()
3:   batteria ← livelloBatteria()
4:   if (busy || batteria ≥ 0) then
5:     AggiornaParametri(batteria)
6:     if btState = STANDBY then
7:       btState ← INITIATING
8:     end if
9:   else
10:    btState ← STANDBY
11:   end if
12: end function
```

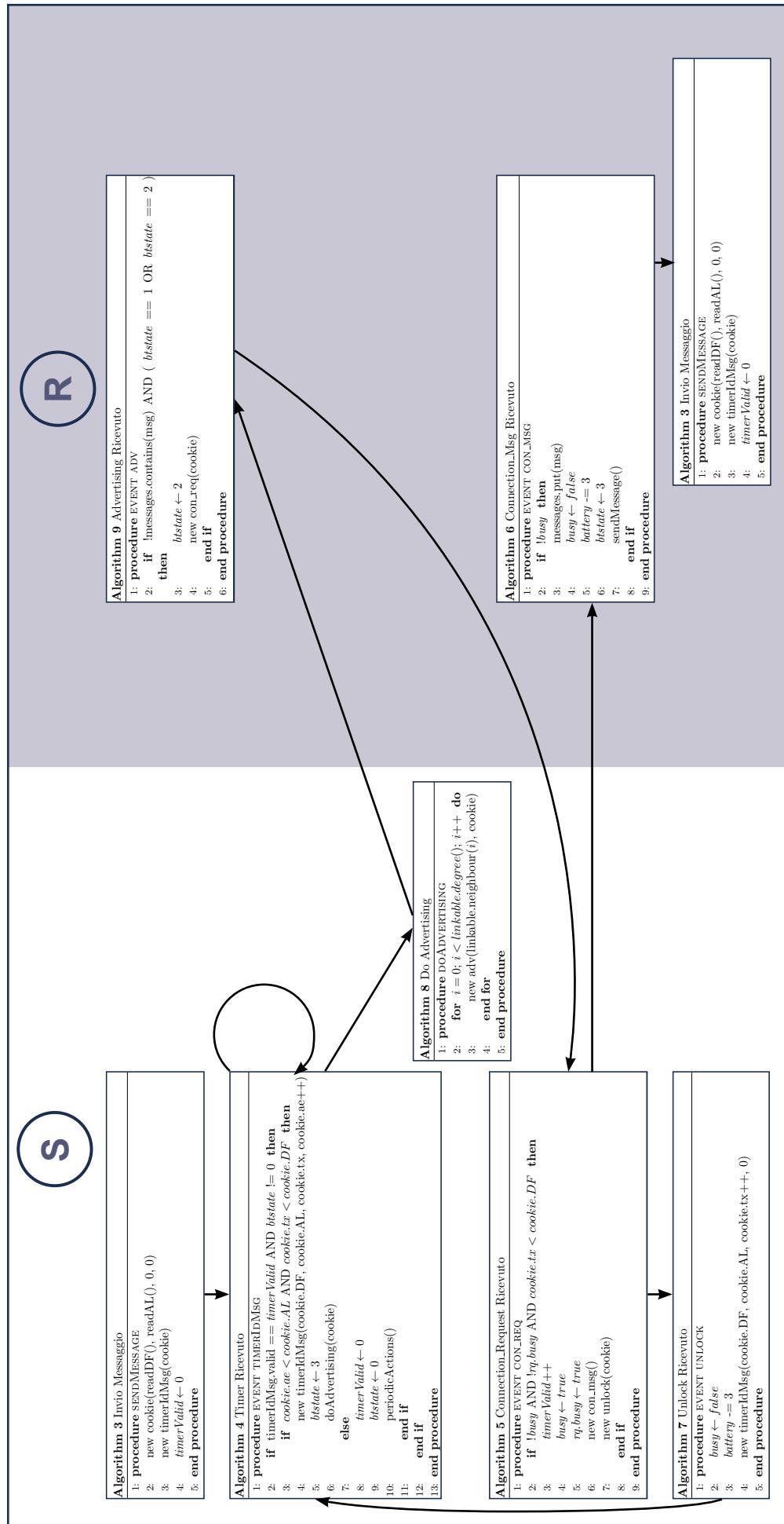


Figura 3.4 - Procedura di invio messaggio nei nodi Sender (S) e Receiver (R)

Simulazioni e valutazioni dei risultati

Come risultato della fase di implementazione del codice abbiamo ottenuto un protocollo che condivide le stesse specifiche e lo stesso funzionamento di quello originale [15], ma è utilizzabile su Peersim, vedi [Sezione 3.2](#), il simulatore da noi utilizzato.

A questo punto si è potuto procedere alla parte di realizzazione sperimentale del nostro lavoro.

Questa fase prevede di effettuare un alto numero di simulazioni al variare dei 3 parametri fondamentali che caratterizzano il nostro sistema.

Appena parte la simulazione sono letti dal file di configurazione i parametri della stessa, i 3 principali sono il numero di nodi, la densità degli stessi e il raggio del BLE.

In questo modo si riesce a verificare il comportamento del nostro protocollo, ma più importante con questa mole di dati raccolti si può effettuare un'analisi più oggettiva delle sue prestazioni.

Per avere una sufficiente base statistica, abbiamo impostato 20 simulazioni per ogni terna di valori differenti dei 3 parametri principali: numero di nodi, densità e raggio.

Queste 20 simulazioni quindi differivano soltanto dal seed e dalla posizione del nodo che generava il messaggio, ma condividevano lo stesso numero di nodi, densità e raggio.

I valori esaminati del numero di nodi n sono:

- n = 2
- n = 5
- n = 10
- n = 30
- n= 50
- n= 80
- n= 100
- n= 200
- n = 500
- n = 1000

Le densità d studiate sono:

- $d = 0.02 \frac{\text{nodi}}{\text{m}^2}$.
- $d = 0.01 \frac{\text{nodi}}{\text{m}^2}$.
- $d = 0.008 \frac{\text{nodi}}{\text{m}^2}$.
- $d = 0.001 \frac{\text{nodi}}{\text{m}^2}$.
- $d = 0.0005 \frac{\text{nodi}}{\text{m}^2}$.
- $d = 0.0001 \frac{\text{nodi}}{\text{m}^2}$.

I valori del raggio ρ sono:

- $\rho = 10\text{m}$.
- $\rho = 15\text{m}$.
- $\rho = 50\text{m}$.

Le densità da $d = 0.02 \text{ nodi/m}^2$ a $d = 0.001 \text{ nodi/m}^2$ sono state pensate per simulare ambienti urbani densamente popolati. Riusciamo così a modellare sia medie-grandi città che, per le densità più elevate, situazioni di forte concentrazione di persone in un'area ristretta. Quest'ultimo caso può rappresentare una locale alta concentrazione abitativa come una serie di palazzi vicini tra loro, ma anche eventi che concentrano molte persone in uno stesso luogo.

Le densità più piccole, $d = 0.0005 \text{ nodi/m}^2$ e $d = 0.0001 \text{ nodi/m}^2$ sono state pensate per studiare la scalabilità del sistema a fronte di dispersione dei nodi, ma anche per simulare situazioni tipiche di piccoli comuni con un basso numero di abitanti, ad esempio i paesini di campagna.

Peersim non ha già pronti componenti per la raccolta e manipolazione dei dati, quindi è stato necessario scrivere un *Control* (che avesse la caratteristica di observer) specifico per il nostro protocollo che individuasse i dati di interesse, alla fine di ogni simulazione.

Questo componente, BLEReport, vedi [Appendice B](#), non serve per scrivere i valori su consolle ma per raccoglierli e salvarli su un file .dat, così che possano essere successivamente analizzati.

Il primo valore che abbiamo voluto osservare è stato la percentuale di diffusione dell'informazione, per studiare quali fossero i limiti di applicabilità in termini di efficienza del sistema e per avere un insieme di scenari in cui il nostro protocollo avesse un buon livello di prestazione.

Il secondo risultato raccolto è stato il tempo totale di trasmissione, inteso



come l'istante di tempo in cui l'ultima trasmissione si è conclusa. Questo perché si è voluto dare un corrispettivo valore temporale alla percentuale di nodi che il protocollo riesce a coprire.

Infatti a seconda della dispersione spaziale della rete non è garantito che tutti i nodi siano connessi ad uno stesso grafo. Quindi il tempo totale di trasmissione deve essere analizzato in coppia con il valore percentuale di rete coperta durante quella specifica simulazione, così da dare un'idea di come il tempo di propagazione evolva al variare della copertura e della densità della rete.

Per valutare meglio l'efficacia della propagazione, abbiamo anche analizzato un *fattore di efficienza*: il rapporto tra la copertura raggiunta, in termini di numero di nodi, e il tempo impiegato.

4.1. | Valutazione qualitativa dei risultati

I risultati, raccolti con gli strumenti illustrati precedentemente, possono a questo punto essere analizzati e confrontati con quelli del precedente lavoro [13], [15] così da trarne alcune considerazioni.

Tuttavia ancor prima che da un'analisi dei risultati, il comportamento del protocollo può essere esaminato osservando l'evoluzione della rete durante una simulazione.

D'altronde Peersim non dispone di alcun tipo di strumento grafico per la rappresentazione della rete o dei messaggi tra i nodi, come già sottolineato nella [Sezione 2.7](#).

Ho pensato potesse essere di interesse implementare uno strumento per porre rimedio a questa mancanza, che fornisca un output grafico dell'esecuzione della simulazione.

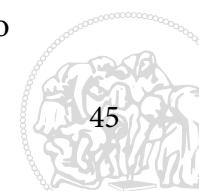
Ottenerne tutte le funzionalità tipiche di un'interfaccia grafica vera e propria (input) presentava un alto grado di complessità, tuttavia ho ritenuto potesse essere di utilità raggiungere un livello base.

In pratica uno strumento che possa fornire un output grafico dell'evoluzione della rete, costituito dal mostrare a livello visivo la situazione della rete, comprensiva dello stato in cui si trova ogni nodo, ad intervalli regolari. Quindi con una determinata periodicità visualizzare le varie situazioni della rete, in dati istanti di tempo, in sequenza; in modo da poter di fatto osservare l'evoluzione della rete.

La situazione della rete in un dato istante è costituita dalla disposizione dei nodi, dal modo in cui essi sono collegati tra loro e dal loro stato link layer, vedi [Sezione 3.1.1](#). Ogni stato è associato ad un colore, il nodo è mostrato del colore dello stato in cui si trova in quell'istante, inoltre sono evidenziati tutti i collegamenti in cui sta avvenendo la trasmissione di un messaggio.

Per dare un esempio di funzionamento prendiamo come periodicità 5 secondi, il mio strumento mostrerà la rete e lo stato dei nodi al tempo 0, poi al tempo 5, poi al tempo 10 e così via.

A livello pratico è stato necessario scrivere un *Control*, BLEStamp, vedi [Appendice C](#), per Peersim che servisse per raccogliere le informazioni sullo



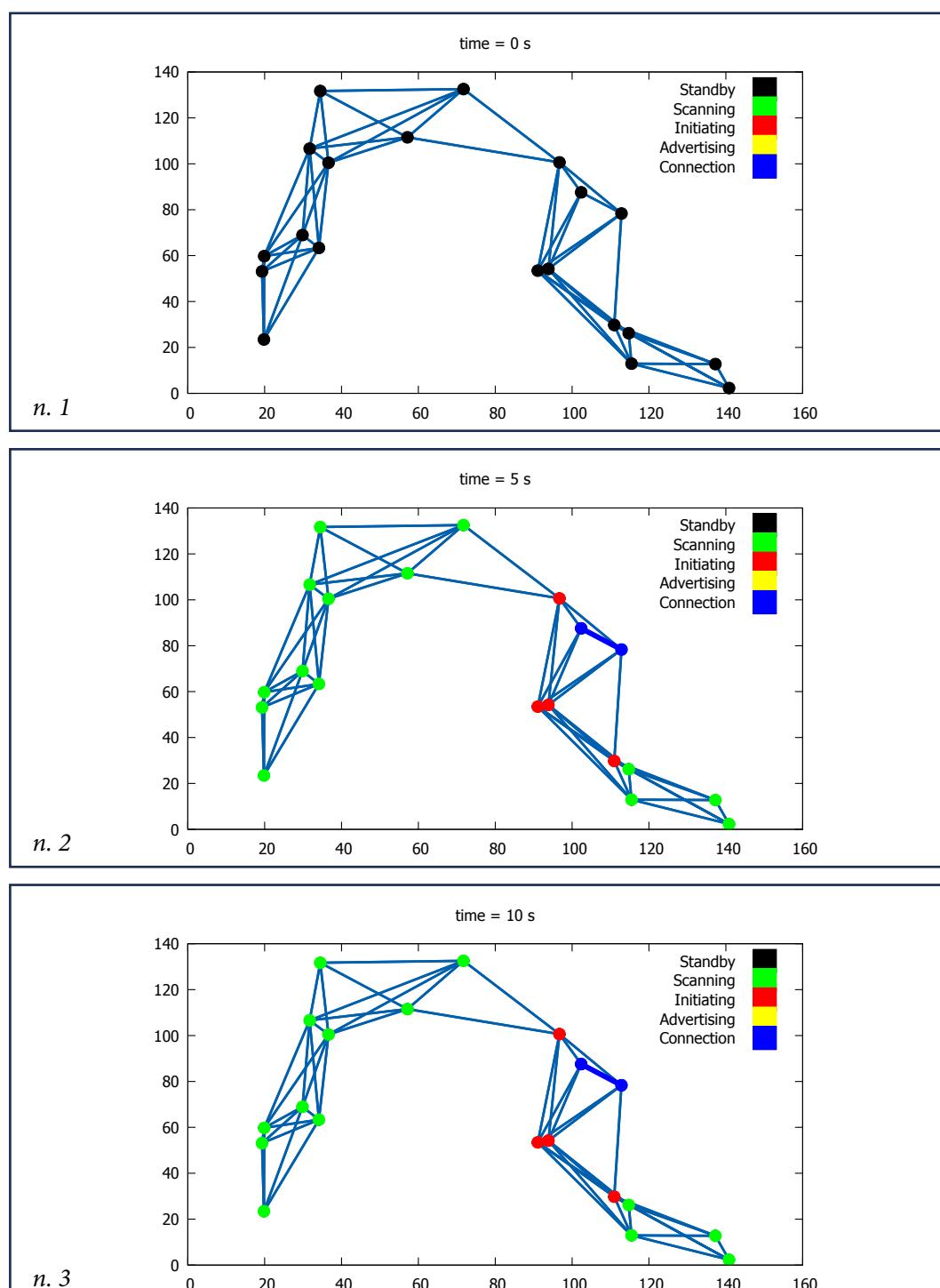
stato dei nodi durante la simulazione.

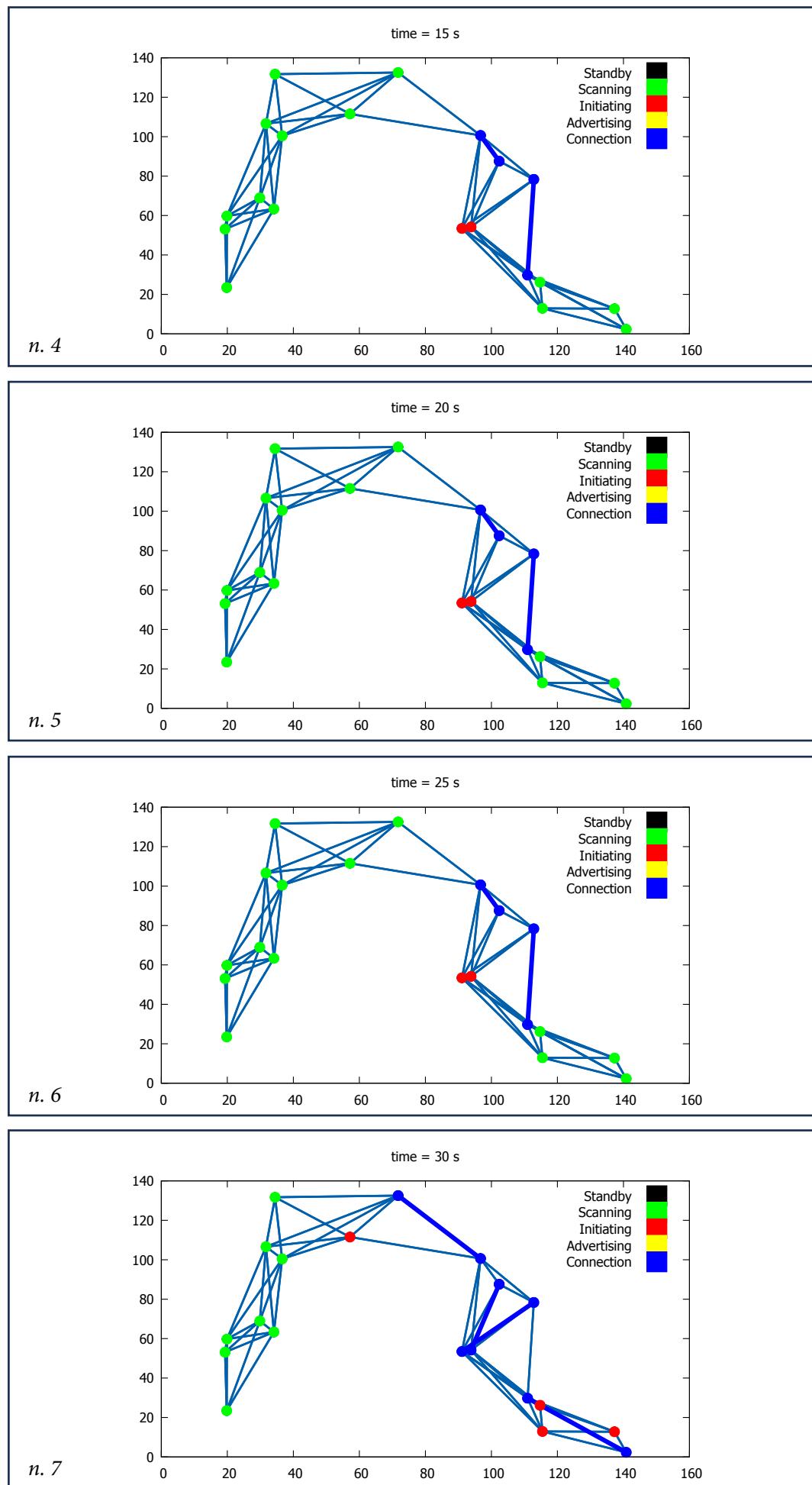
Questo componente, che funge da observer, raccoglie i dati sopracitati con periodicità da noi scelta e li salva in vari file .dat, in formato compatibile con Gnuplot.

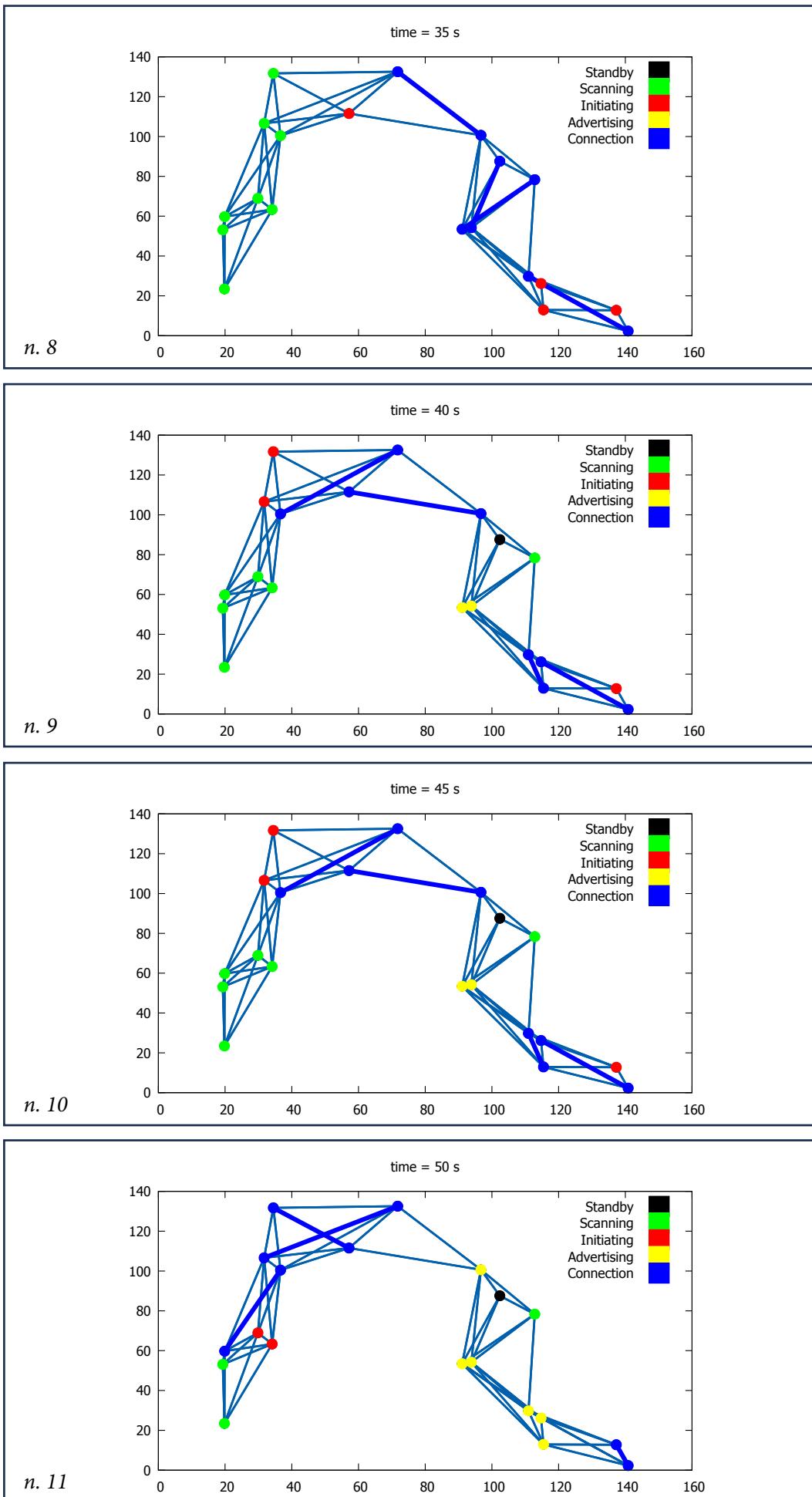
Usando quindi uno script scritto per Gnuplot, ricordiamo programma che permette di visualizzare graficamente file di dati, riusciamo a stampare a monitor i file raccolti.

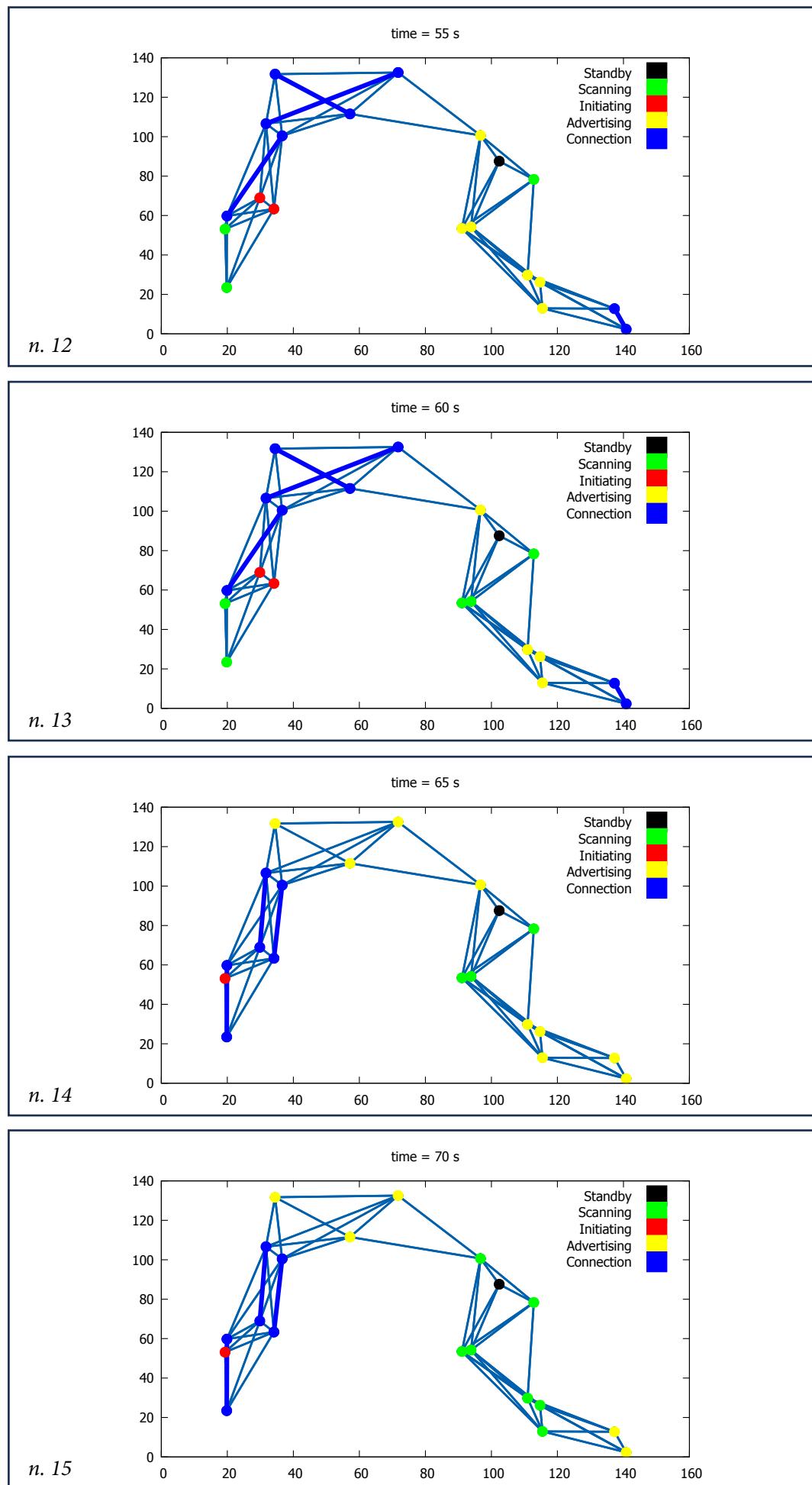
In questo modo vengono mostrate le situazioni della rete in sequenza in ordine di tempo, l'effetto ottenuto è quindi il poter osservare l'evoluzione della rete graficamente.

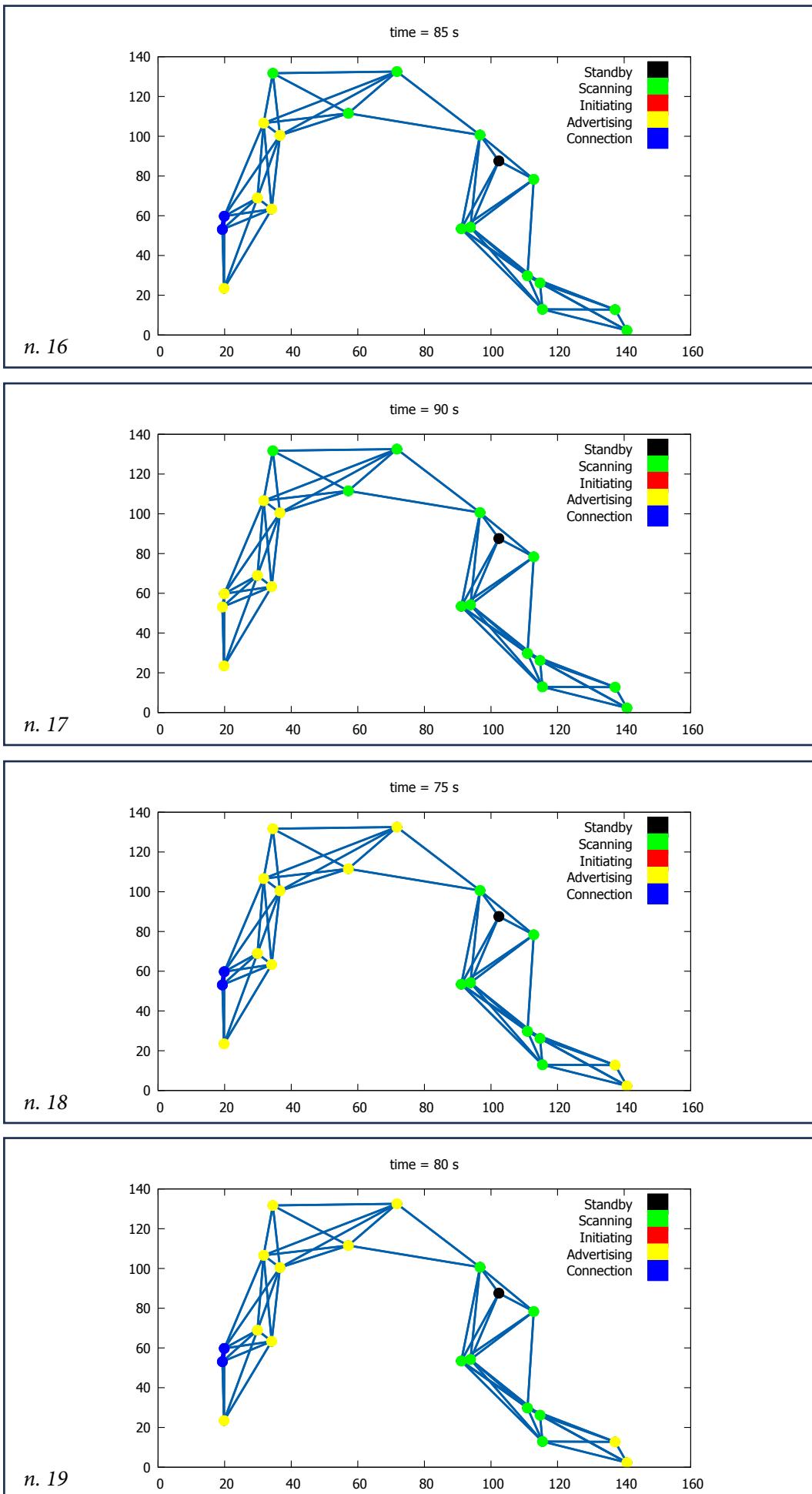
Mostriamo di seguito un'esecuzione dello strumento appena discusso.

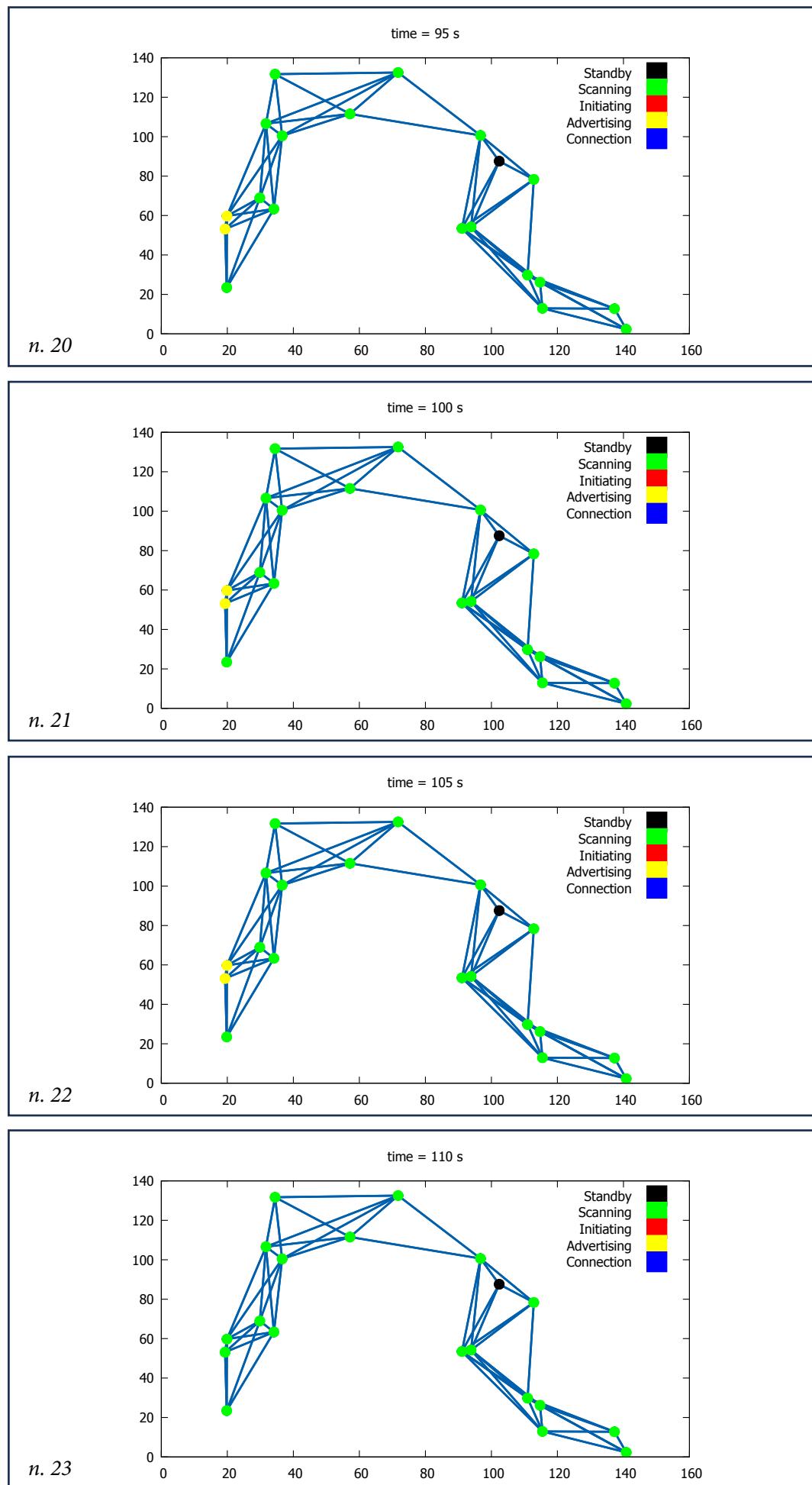












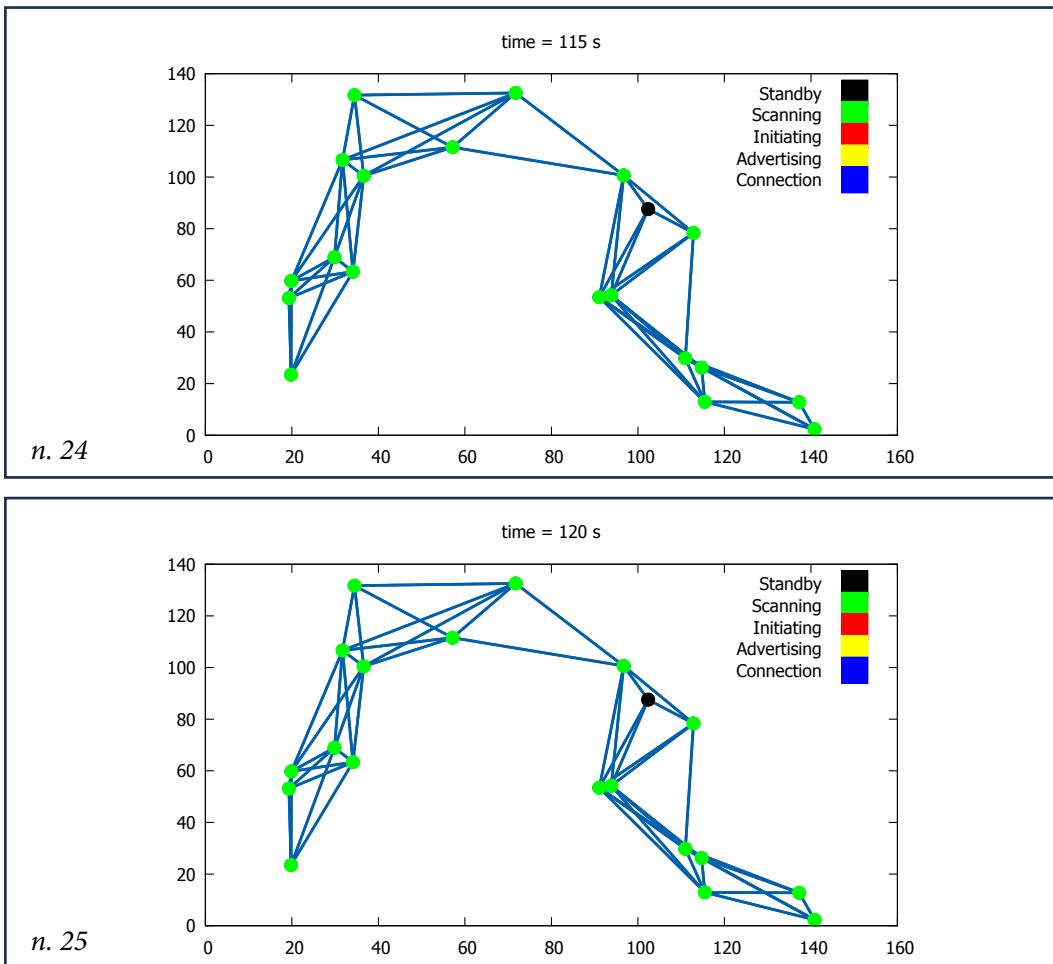


Figura 4.1 (numerate da 1 a 25) - Evoluzione della rete durante una simulazione

4.2 | Valutazione risultati

Dopo aver terminato le simulazioni, abbiamo raccolto i dati con il procedimento descritto ad inizio [Sezione 4](#), e li abbiamo elaborati e successivamente analizzati. I risultati che abbiamo monitorato sono stati:

- Copertura: percentuale di contagio che il messaggio ha raggiunto.
- Tempo totale di trasmissione: tempo totale impiegato per raggiungere tale copertura.

Passaggio fondamentale è stato portare i risultati nella stessa forma di quelli derivanti dal lavoro precedente [13], [15] per poterli in questo modo confrontare con questi ultimi.

A questo scopo è stato necessario calcolare valore medio, varianza, deviazione standard e intervallo di confidenza al 95% così da poter portare tutti i risultati in forma di grafico.

Prima di parlare dei singoli risultati ripetiamo alcuni aspetti che hanno caratterizzato la simulazione.

Per avere una maggior variabilità durante le simulazioni abbiamo assegnato molti valori tramite l'uso di generatori casuali, come ad esempio la disposizione



dei nodi, il livello di batteria iniziale di ogni dispositivo e la scelta del nodo che possiede il messaggio da diffondere.

Per ogni densità, sono stati simulati i tre raggi $\rho = 10 \text{ m}$, $\rho = 15 \text{ m}$, $\rho = 50 \text{ m}$ scelti e poi è stata fatta la media.

Confrontare i nostri grafici con quelli del lavoro precedente serve per capire se le considerazioni sui risultati del protocollo originale restano valide anche per quelli ottenuti da noi, cosa ovviamente auspicabile e che per certi versi permette di valutare la qualità di questo lavoro.

In condizioni ideali, tutti i nodi connessi in un unico grafo e batteria infinita per tutti i dispositivi, gli algoritmi di gossip con strategia di diffusione Push&Pull hanno queste caratteristiche:

- L'informazione viene diffusa a tutti dopo $O(\ln n)$ cicli.
- La copertura completa richiede almeno $O(n \log \log n)$ trasmissioni di messaggi.

con n il numero di nodi della rete.

Nel nostro caso invece è molto difficile poter dire con quale legge si comporta il nostro algoritmo per via dello scaricarsi dei dispositivi e della formazione di sotto-reti isolate nei casi di bassa densità.

Quest'ultimo problema può essere, almeno in parte, superato andando a considerare la mobilità dei nodi, come vedremo nella [Sezione 5.1.4](#).

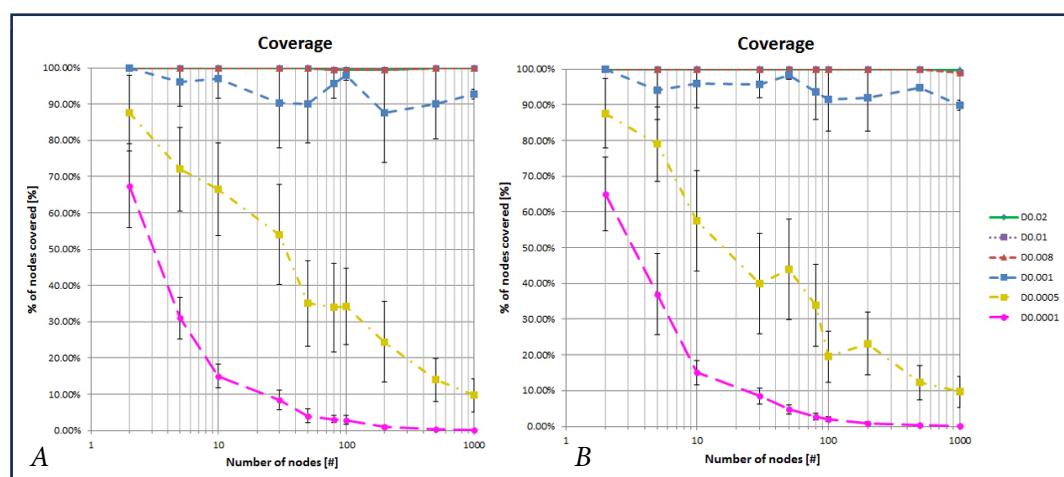


Figura 4.2 - Confronto grafici della copertura della rete (A: Pagliari L. - B: Malvestiti D.)

Il primo parametro che presentiamo è quello della percentuale di rete contagiata dal messaggio.

Confrontando i due grafici relativi alla copertura possiamo notare che sono quasi identici, questo ci dice che il funzionamento del protocollo è stato riprodotto correttamente.

Quindi le considerazioni seguenti varranno per entrambi i casi.

Per le densità $d = 0.02 \text{ nodi/m}^2$, $d = 0.01 \text{ nodi/m}^2$ e $d = 0.008 \text{ nodi/m}^2$ notiamo

copertura del 100% pressoché costante anche all'aumentare dei nodi.

Questo è indice di un'ottima efficienza prestazionale, per tutti e tre i raggi ρ considerati.

Per il caso $d = 0.001$ nodi/m² vengono evidenziati i limiti di operabilità del nostro protocollo.

I valori della copertura rimangono ottimi solo per il caso $\rho = 50$ m mentre scendono fortemente per gli altri due, lo si nota dall'intervallo di confidenza che diventa molto ampio.

In generale come media siamo attorno al 90%, quindi ancora valori accettabili.

Per $d = 0.0005$ nodi/m² abbiamo un degrado prestazionale linearmente dipendente dalla grandezza della rete. Questo ci porta a considerare questa densità come un limite inferiore di operatività.

Anche per questo caso abbiamo una crescita dell'intervallo di confidenza.

Per l'ultima densità abbiamo un degrado prestazionale forte, sistema quindi inutilizzabile.

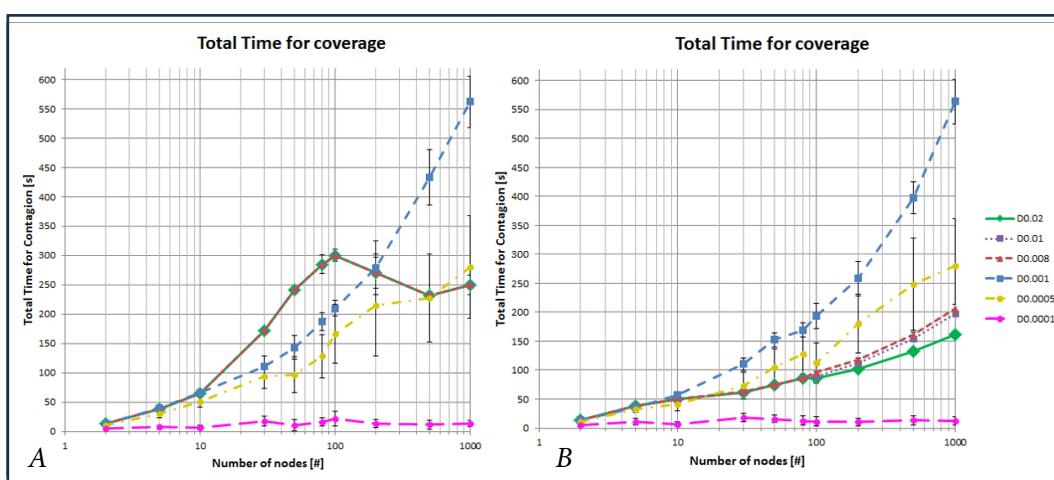


Figura 4.3 - Confronto grafici del tempo totale di trasmissione (A: Pagliari L. - B: Malvestiti D.)

Altro parametro che abbiamo esaminato è il tempo totale di trasmissione. Questo indica il tempo che è stato necessario per raggiungere la massima copertura in ogni circostanza.

Per le prime tre densità più alte, $d = 0.02$ nodi/m², $d = 0.01$ nodi/m² e $d = 0.008$ nodi/m², abbiamo tempi leggermente differenti per i due grafici, ma ugualmente buoni.

L'alto grado di vicinanza tra i nodi ha permesso di contenere i tempi totali. Questo ovviamente è dovuto anche al fatto che l'alta ridondanza dei collegamenti impedisce la formazione di bottleneck.

Per il caso $d = 0.001$ nodi/m² la curva dei tempi ha crescita quadratica, tuttavia viene mantenuta una buona copertura, come detto in precedenza.

Ragionamento analogo per quanto riguarda l'andamento dei tempi si può fare per $d = 0.0005$ nodi/m², ma in questo caso la copertura è peggiore.

Per l'ultima densità considerata abbiamo tempi bassi, ma totalmente dovuti al basso livello di operatività del nostro protocollo per densità uguali o minori di questa.

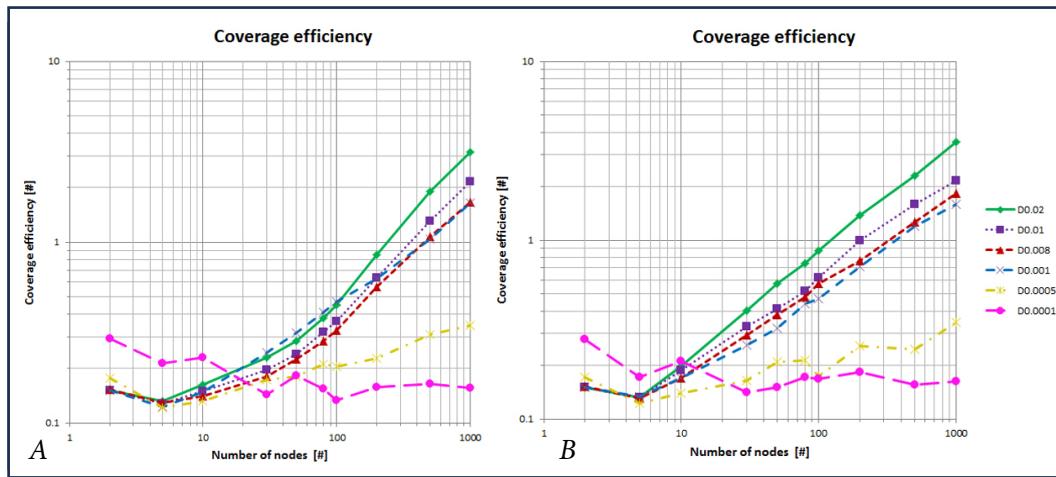


Figura 4.4 - Confronto grafici del Fattore di Efficienza (A: Pagliari L. - B: Malvestiti D.)

Passiamo quindi ad analizzare il *fattore di efficienza*, il rapporto tra la copertura raggiunta, in termini di numero di nodi, e il tempo impiegato per raggiungere tale copertura.

Le prime quattro densità più alte hanno efficienza molto simile. L'estrema vicinanza tra i nodi fa sì che le curve abbiano pressoché lo stesso andamento. Per il caso $d = 0.0005$ nodi/ m^2 l'efficienza fatica ad aumentare al crescere dei nodi. Invece per l'ultima densità considerata, la curva dell'efficienza tende quasi a decrescere.

Estensioni

5.1 | Introduzione

Conclusa la parte di raccolta dati dal simulatore, ci siamo dedicati ad una fase di ricerca che permetesse di individuare in che modo migliorare il nostro protocollo al fine di raggiungere un più alto grado di realismo e aggiungerne ulteriori funzionalità.

Abbiamo così analizzato diverse possibilità, non tutte si sono rivelate fattibili e al termine del processo solo alcune si sono tradotte in estensioni per il simulatore.

Inizialmente abbiamo tentato di aumentare l'efficienza del protocollo aggiungendo una coda di priorità che gestisse l'ordine di invio dei messaggi del singolo nodo.

In secondo luogo, essendo il protocollo basato su un algoritmo di tipo Push & Pull, abbiamo implementato un componente in grado di gestire efficacemente la fase di Pull vera e propria.

Un'altra estensione ha riguardato il disabilitare le funzionalità principali del protocollo, in quanto si è pensato potesse essere d'interesse fare un confronto di risultati con il caso base, cioè con DF = 1.

Successivamente ci siamo dedicati a rendere in modo più realistico l'ambiente cittadino.

Per sviluppare tale casistica mi sono avvalso della collaborazione di un collega ingegnere edile - architetto che mi ha fornito le nozioni base in campo urbanistico per una schematizzazione approssimativa del sistema città e del comportamento degli abitanti della suddetta. Ho quindi voluto approssimare la città ad una circonferenza, in quanto lo schema distributivo dei nuclei abitativi storici e moderni si sviluppa sempre attorno ad un centro e da lì si espande grazie a quelle che sono le dinamiche del "principio di agglomerazione" che portano i cittadini, come le imprese produttrici di beni e servizi, a localizzarsi gli uni presso gli altri in modo da diminuire "i costi di trasporto" e di conseguenza le spese. Preso quindi un UFI [20] (Urban Fragmentation Index - l'indice che indica il grado di frastagliatura dei confini di una città) pari ad 1, la città è quindi schematizzata attraverso una circonferenza perfetta. Ho poi pensato di simulare l'atteggiamento della popolazione in caso di un evento catastrofico. I movimenti consistono perlopiù nello spostarsi dal nucleo centrale della città verso i margini più esterni, in quanto si è appurato, con l'esperienza, che in caso di panico la popolazione sviluppa questa tendenza ad allontanarsi dalle zone più frequentate per raggiungere ambienti circostanti più isolati.

Nella fase finale del mio lavoro di ricerca, abbiamo voluto concentrarci sui possibili altri campi d'applicazione del software nella vita quotidiana evidenziandone vantaggi e criticità.

E' dato ormai consapevole che, al giorno d'oggi, quasi tutti gli individui di una data popolazione posseggano uno smartphone e con esso un dispositivo

di localizzazione GPS. Abbiamo quindi pensato che il software possa essere impiegato dagli enti preposti per la tutela e la salvaguardia della popolazione sul territorio al fine di localizzare gli individui da salvare nel caso in cui vi si presentino delle condizioni ambientali catastrofiche.

L'apporto che quindi può dare il software è quello della localizzazione dei cittadini che, presi dal panico, possano essere dispersi o possano essere tra le macerie lasciate da un sisma, da un'alluvione o da qualsiasi altro tipo di catastrofe.



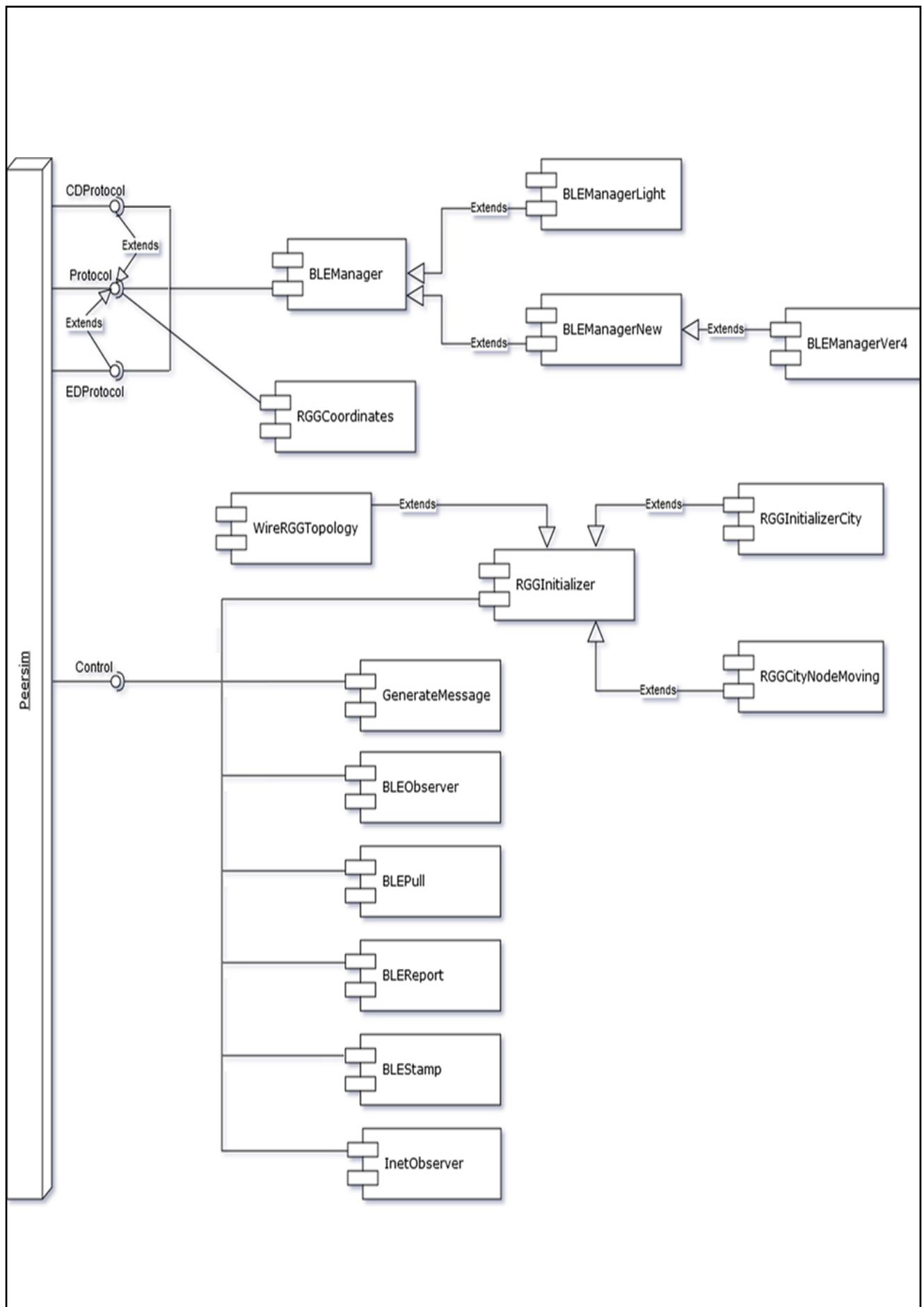


Figura 5.1 - Component Diagram

5.1.1 | Estensioni: coda di priorità

Dopo aver osservato il funzionamento del protocollo all'atto pratico durante le prime esecuzioni, è stato subito chiaro che il primo nodo a ricevere il pacchetto di advertising era nella larga maggioranza dei casi quello selezionato per ricevere il messaggio.

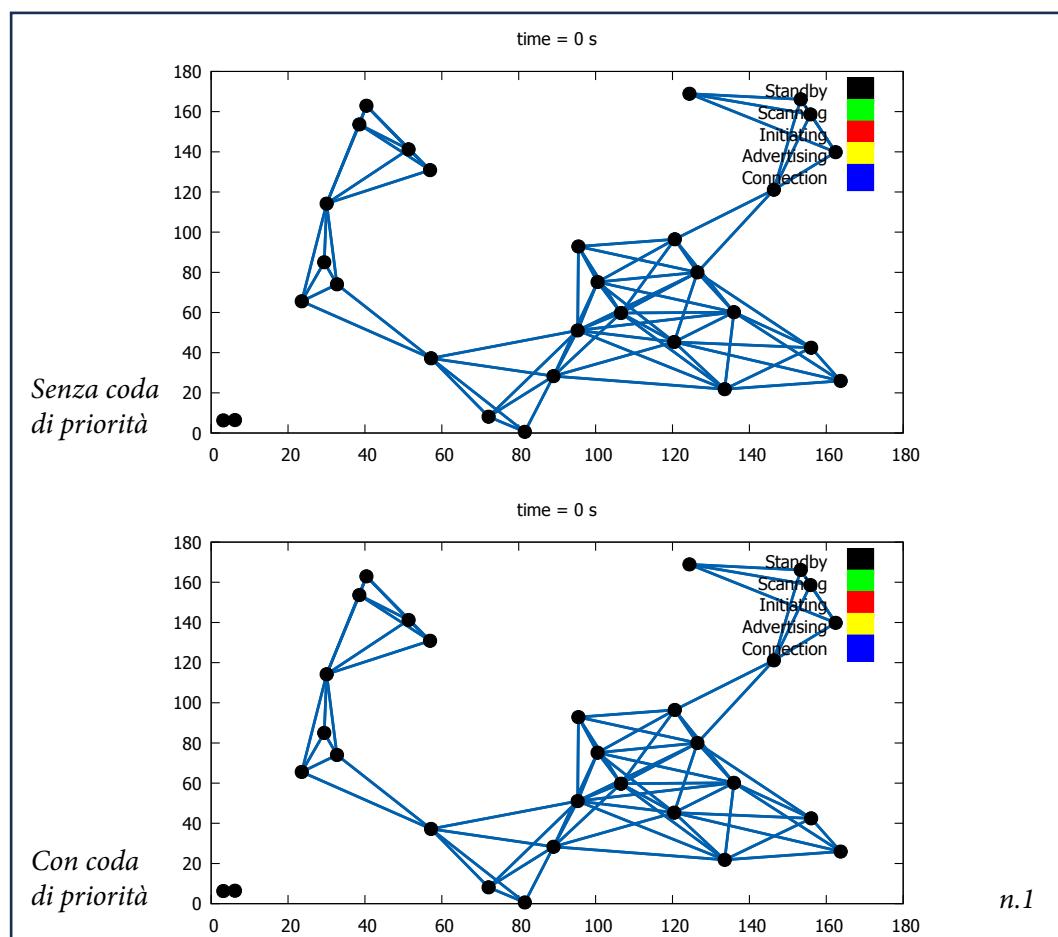
Visto che l'ordine dei neighbors con cui mandare i pacchetti di advertising era in precedenza scelto randomicamente, ho pensato di creare una coda di priorità con cui gestire questo aspetto.

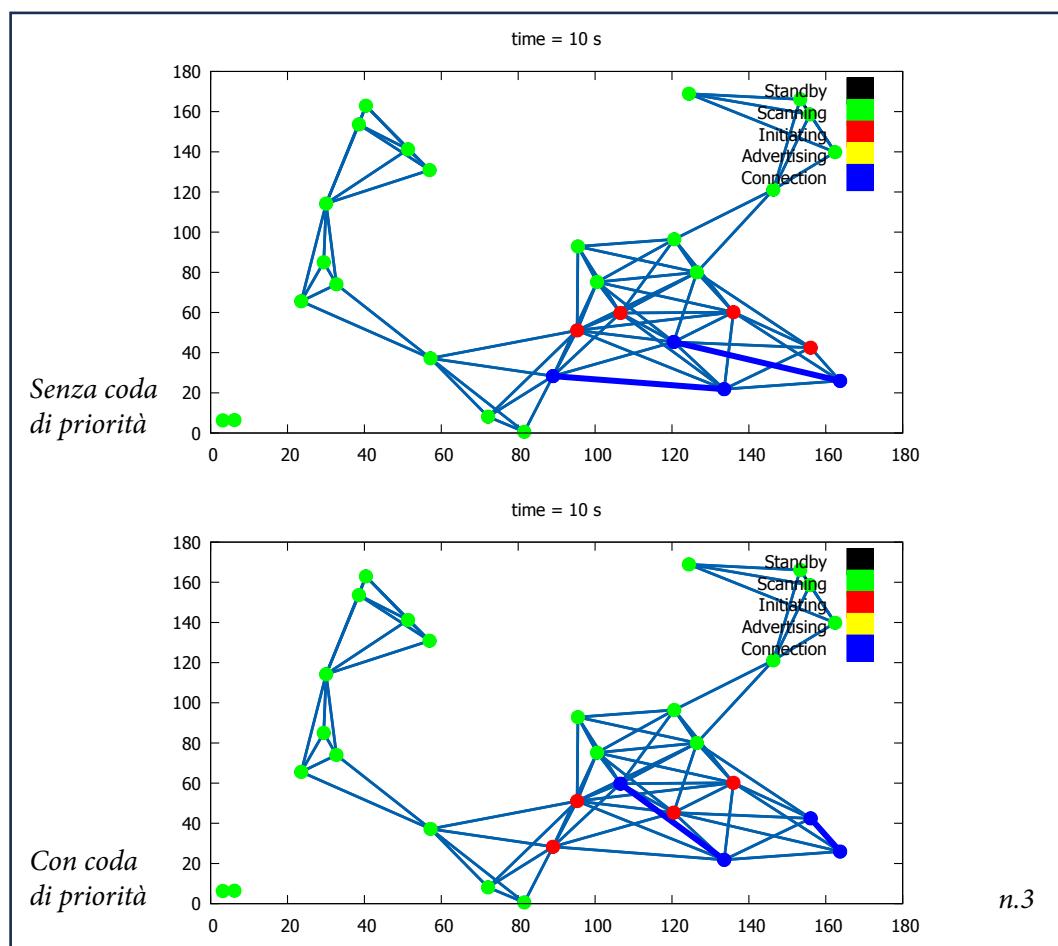
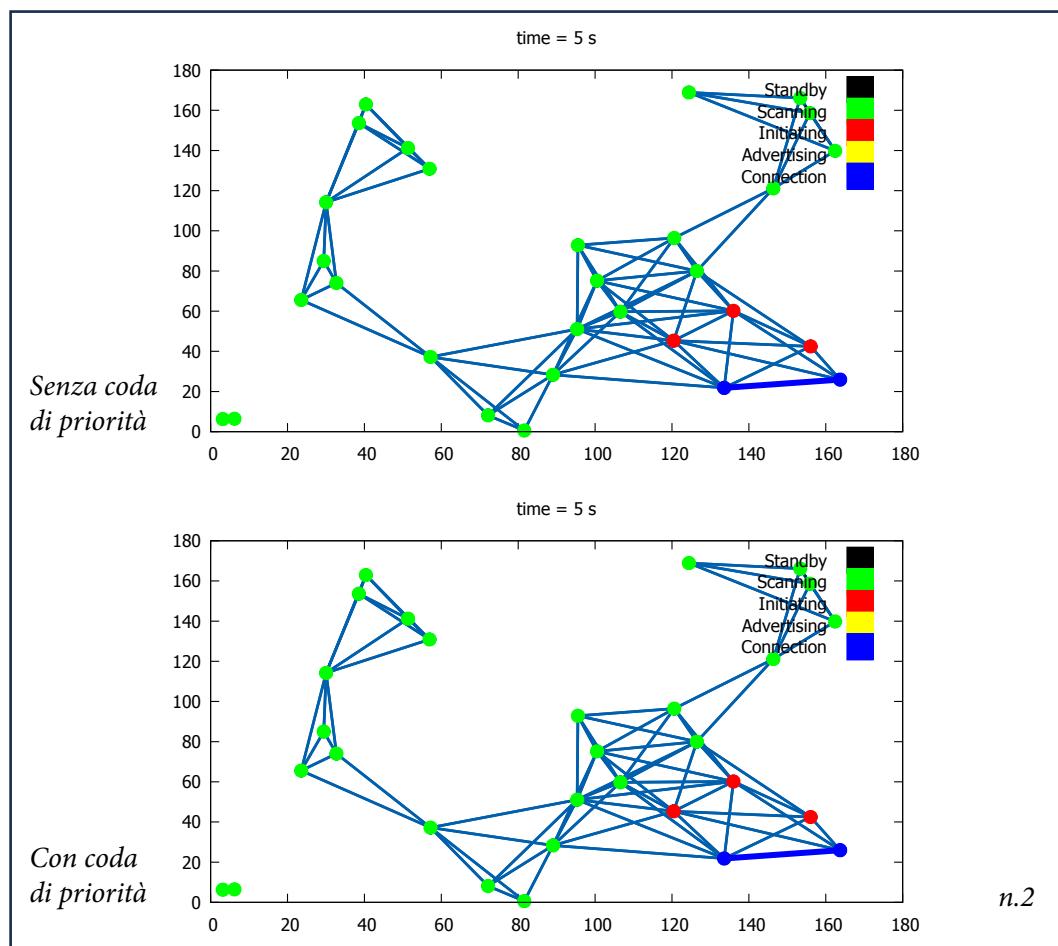
Questo per fare in modo che ogni nodo abbia memorizzato internamente non solo i riferimenti ai nodi vicini ma anche l'ordine con cui mandare i pacchetti di advertising.

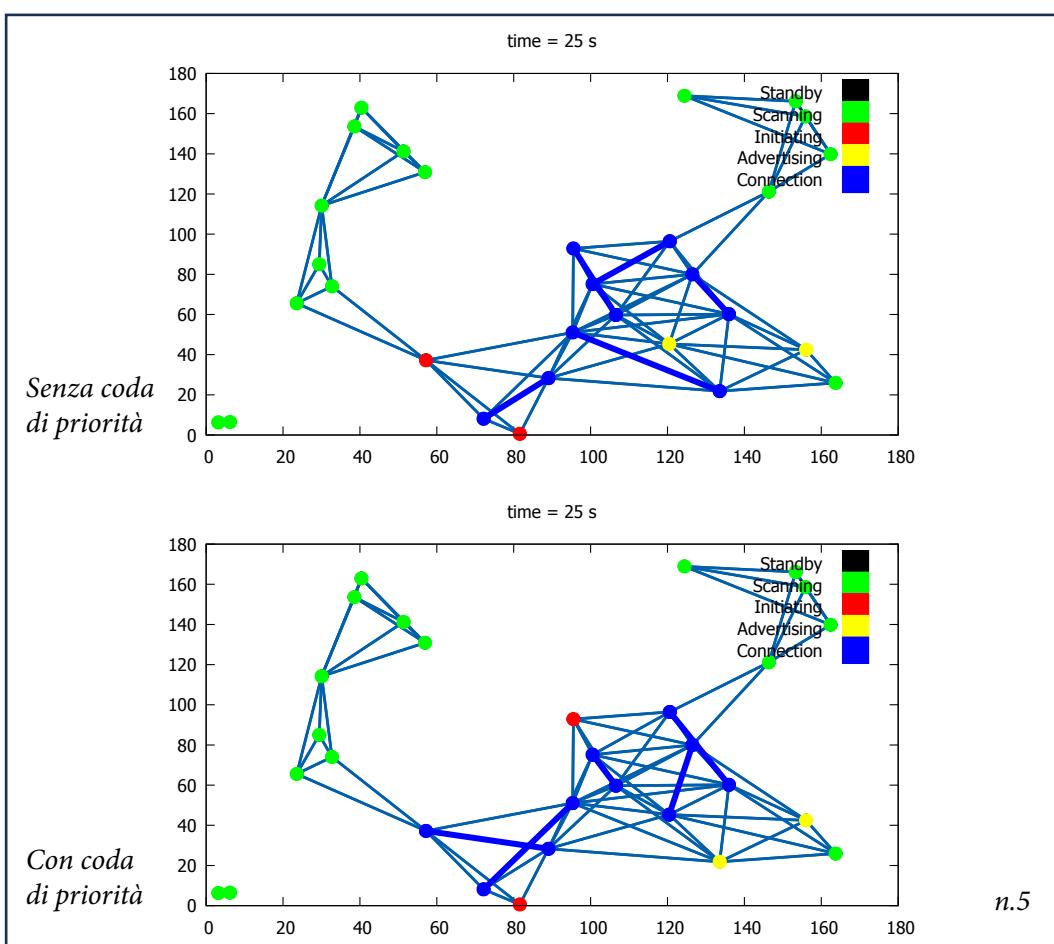
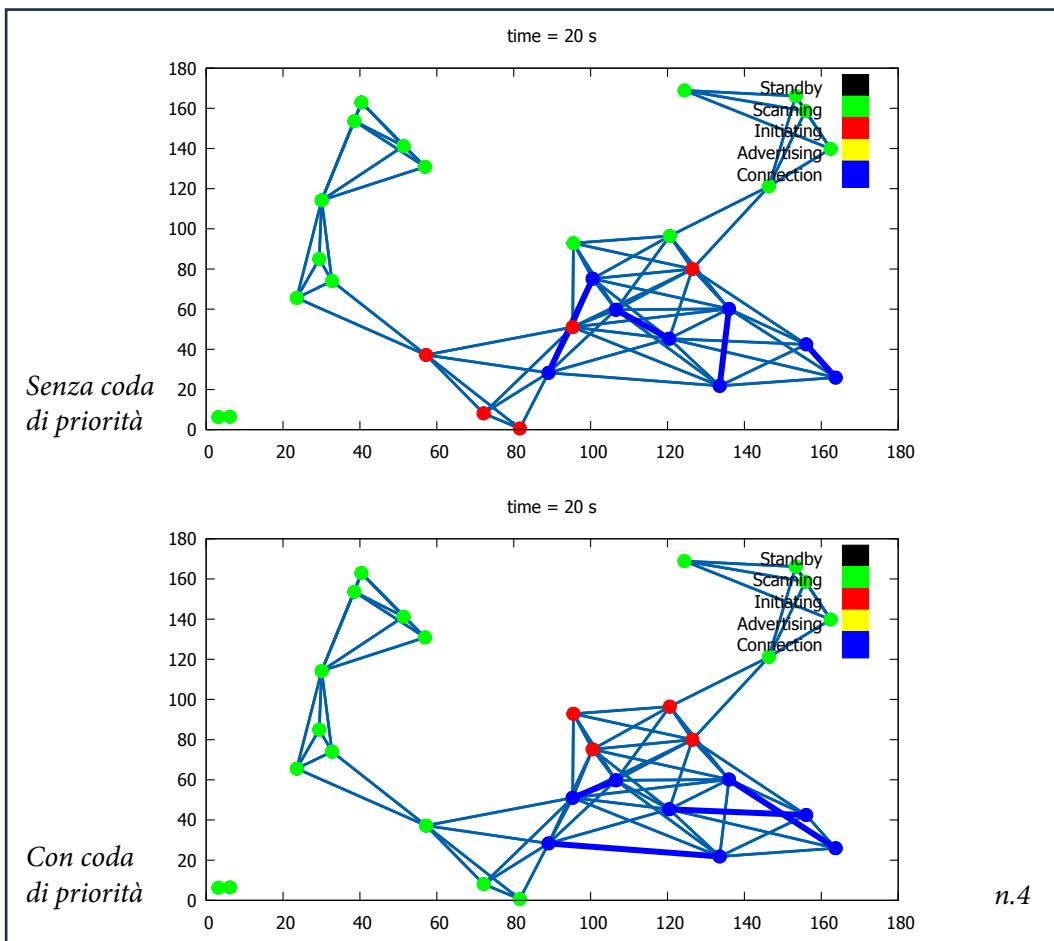
Il criterio che ho scelto per ordinare i nodi era in linea con le idee alla base della tesi precedente, ovvero puntare sul risparmio energetico mantenendo buone performance.

Per ottenere questo risultato il modo migliore è privilegiare nodi con maggiore "capacità di diffusione" ovvero quelli che una volta ricevuto il messaggio lo diffonderanno maggiormente.

Si potrebbe pensare quindi che ordinare i nodi per livello di batteria sia la scelta migliore, ma in realtà il *Dynamic Fanout* rappresenta molto meglio la capacità di diffusione. Il parametro DF racchiude in sé sia l'autonomia rimanente del dispositivo sia il numero di nodi a cui esso può arrivare, quindi a quanti nodi può diffondere il messaggio.







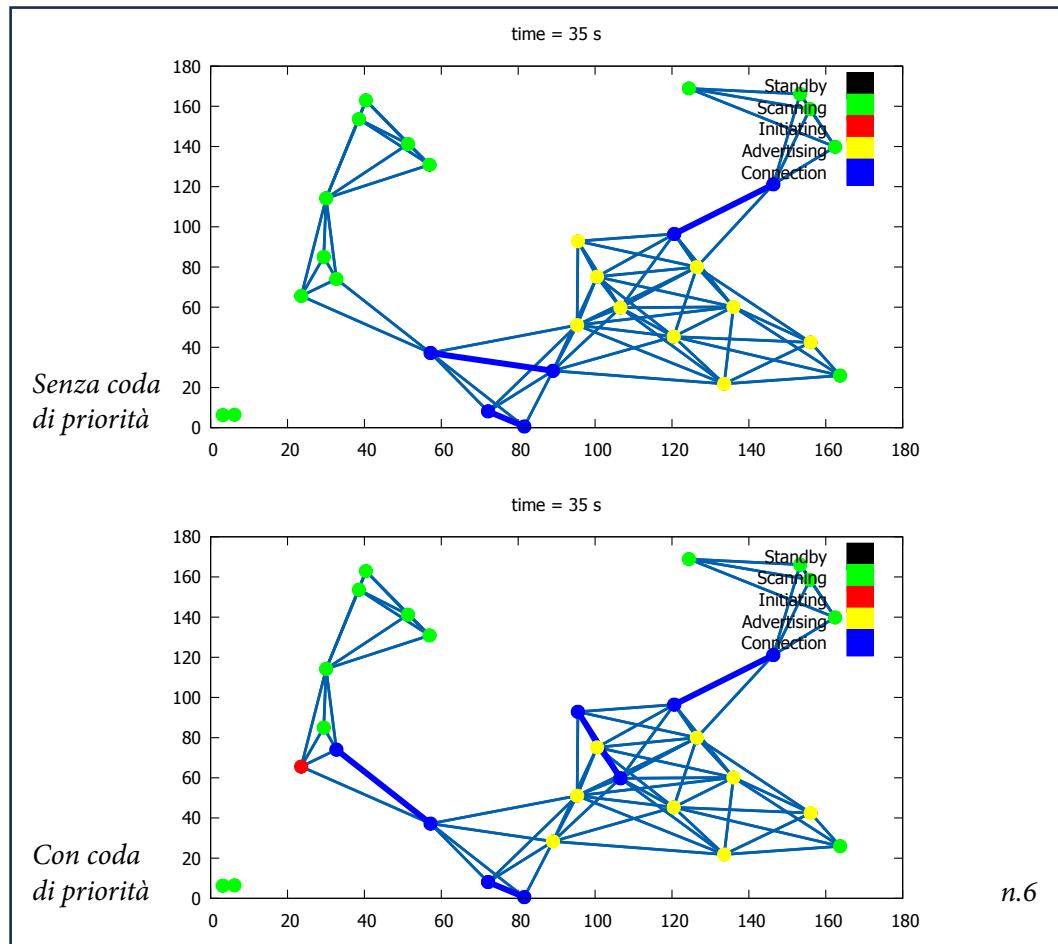


Figura 5.2 (numerate da 1 a 6) - Confronto evoluzione della rete con e senza coda di priorità

Si sarebbe potuta ottenere una soluzione migliore privilegiando i nodi che sono bottleneck per la rete, tuttavia la loro individuazione presenta un livello di complessità molto alto. Essendo il parametro DF in costante aggiornamento a runtime, si è visto necessario far in modo che ogni qualvolta un nodo eseguisse la procedura *AzioniPeriodiche* il valore aggiornato venisse comunicato a tutti i nodi vicini, per poter riordinare la coda di priorità.

Le performance ottenute sono evidenziabili sia tramite immagini della simulazione durante l'esecuzione sia attraverso l'analisi dei risultati visualizzati graficamente.

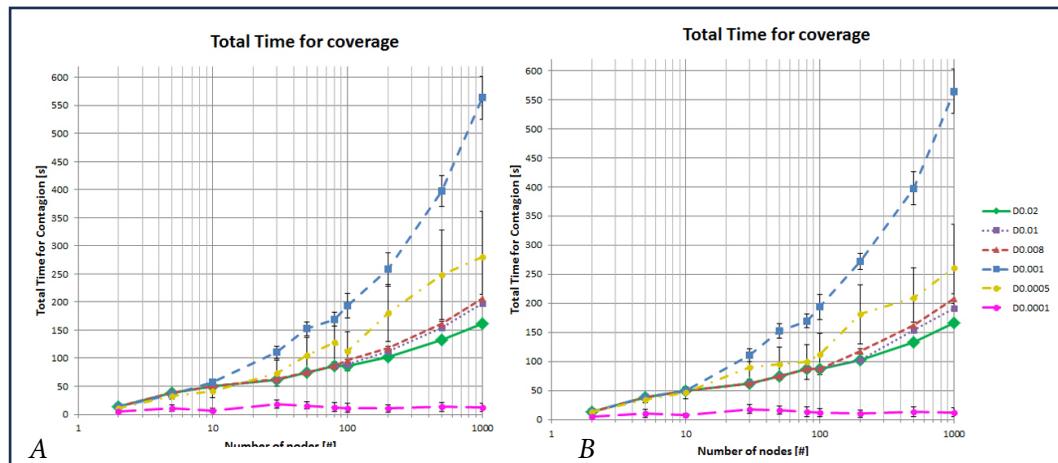


Figura 5.3 - Confronto grafici del tempo totale di trasmissione (A: Protocollo normale. - B: Protocollo con estensione coda di priorità)

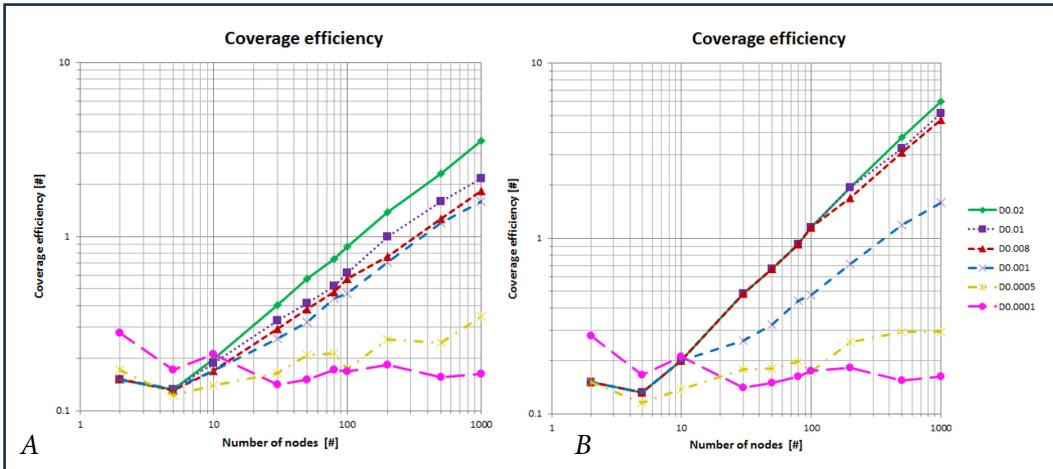


Figura 5.4 - Confronto grafici del Fattore di Efficienza (A: Protocollo normale. - B: Protocollo con estensione coda di priorità)

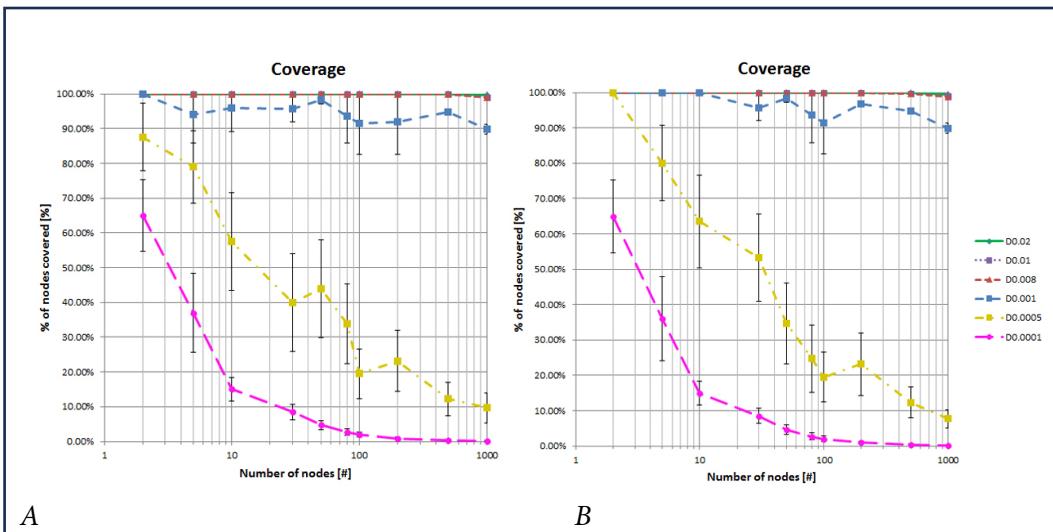


Figura 5.5 - Confronto grafici della copertura della rete (A: Protocollo normale. - B: Protocollo con estensione coda di priorità)

Dal confronto dei grafici dei risultati i miglioramenti di performance non sono così evidenti.

Si notano maggiormente solo nei casi di densità intermedie.

Ne deduciamo che potrebbero non essere tali da giustificare l'introduzione di questa miglioria.

5.1.2 | Estensioni: caso multi-messaggio

L'implementazione del lavoro precedente era pensata per la gestione di un solo messaggio. In caso di partenza di due o più messaggi da nodi diversi il protocollo funzionava sì correttamente, ma i "flussi" dei messaggi si arrestavano appena entravano in contatto tra loro.

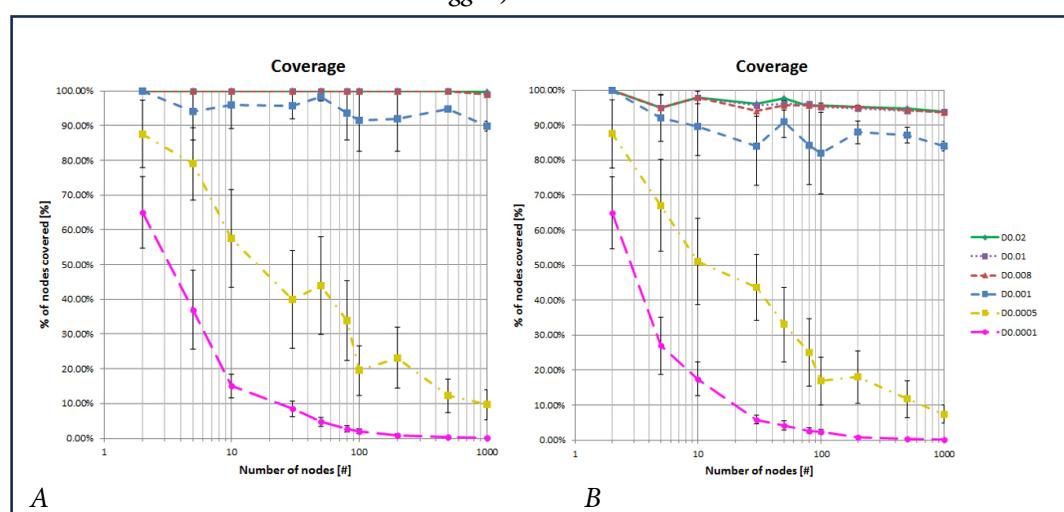
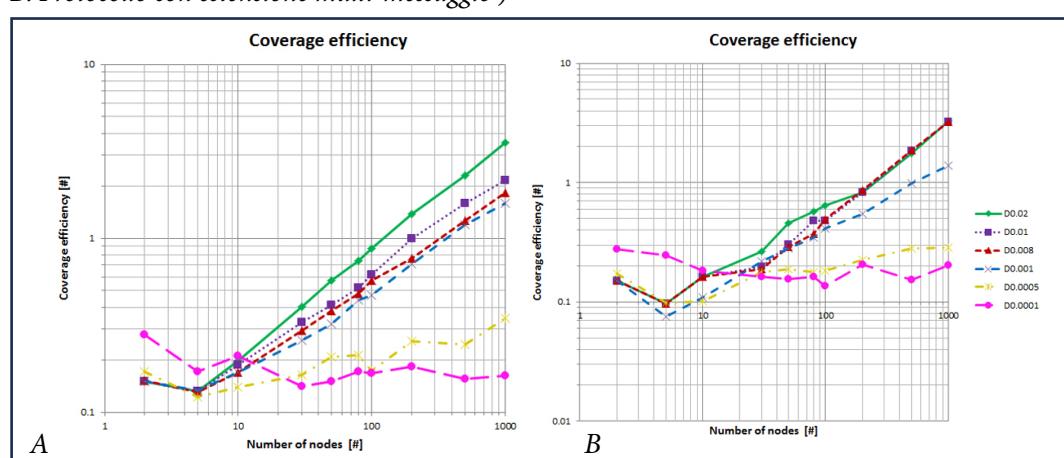
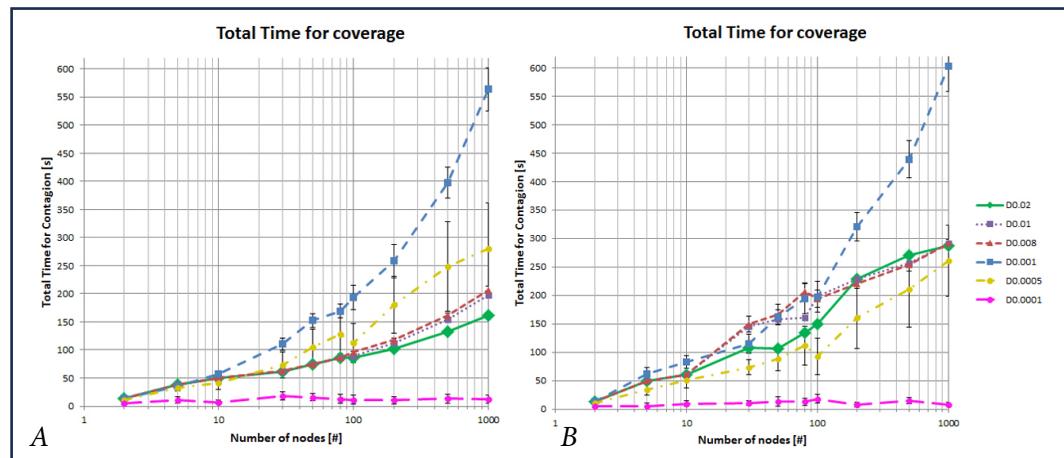
La soluzione scelta per risolvere questo problema mi è stata ispirata dai metodi di diffusione degli algoritmi di gossip, in particolare il metodo Pull. Questo metodo prevede che siano i nodi suscettibili a far richiesta di nuove informazioni ai nodi contagiatati.

Per ottenere questo meccanismo, a livello pratico, ho dovuto aggiungere un



componente che si occupasse della gestione di questo aspetto. In questo modo tutto il protocollo acquisiva pienamente le caratteristiche del metodo Push & Pull.

Il nuovo componente da me implementato ha il compito di attivare tutti quei nodi che sono rimasti inutilizzati per un lungo periodo. Una volta attivati essi chiederanno ai nodi vicini tutti i messaggi di cui sono a conoscenza ma che non hanno ancora effettivamente ricevuto.



In questo modo è garantita la diffusione di più messaggi anche in contemporanea ma, per ovvie ragioni, in misura più rallentata rispetto al caso con singolo messaggio.

I risultati ottenuti con l'estensione multi-messaggio possono essere confrontati con quelli ottenibili con l'utilizzo del normale protocollo.

Da questo confronto emerge che analizzando un caso più realistico, quello con almeno due flussi di messaggi, i risultati risultano peggiori rispetto a quelli del normale protocollo.

Questo peggioramento è in linea con quelle che erano le nostre aspettative.

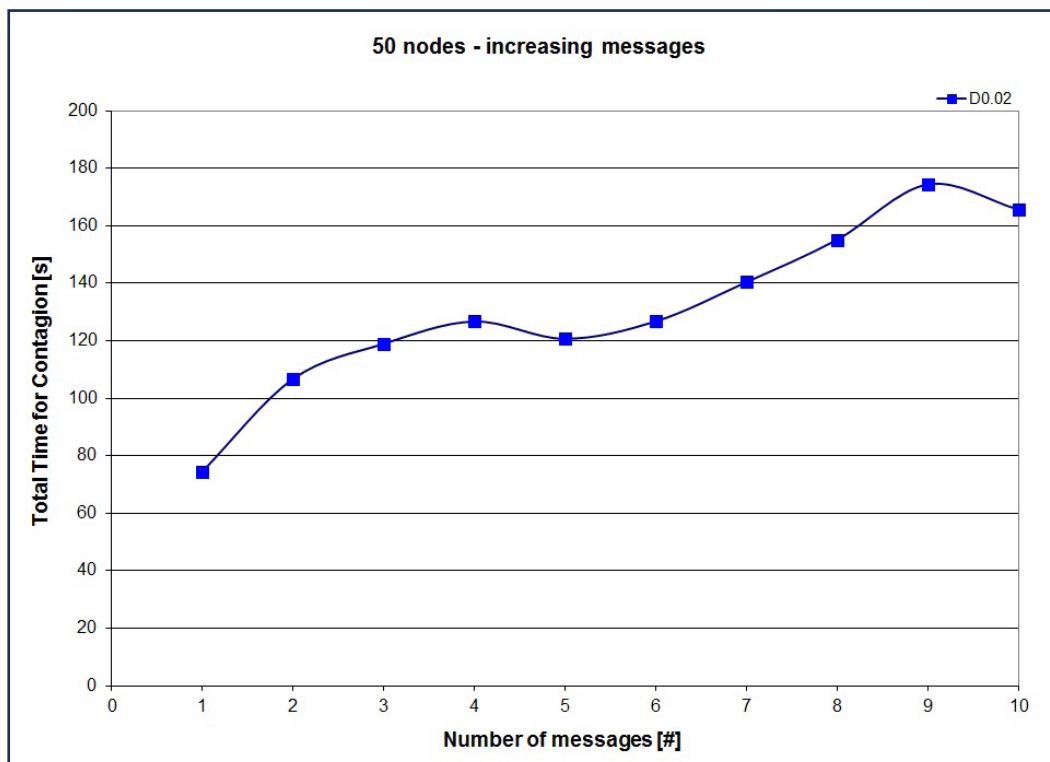


Figura 5.9 - Grafico tempi totali con aumento di messaggi

Da questo grafico possiamo meglio osservare come si comportano i tempi di trasferimento all'aumentare del numero di messaggi contemporanei nella rete.

Risulta evidente un incremento del tempo totale passando da uno a due messaggi, come era emerso anche dal confronto dei tempi totali, [Figura 5.6](#). Oltre i due messaggi l'incremento del tempo totale è presente in misura minore, non linearmente proporzionale.

5.1.3 | Estensioni: caso senza DF

Un certo interesse poteva avere anche il confrontare i risultati del normale protocollo con quelli della sua versione base, cioè senza le funzioni caratteristiche.

Una versione del protocollo che avesse quindi la procedura AzioniPeriodiche



assente e la procedura di invio messaggio di molto semplificata.

Quest'ultima va a perdere del tutto la componente legata agli algoritmi di diffusione, quindi l'uso dei parametri DF e AL comprensivo di timer per la gestione degli advertising a vuoto.

Rimane, nella versione base, solo la logica legata all'uso del Bluetooth e alla gestione batteria.

Il funzionamento degli stati rimane quindi immutato, per l'invio di un messaggio sarà sempre necessaria la fase di advertise e di connection_request prima della connessione vera a propria.

Come risultato di questa semplificazione ogni nodo, dopo aver mandato pacchetti di advertising a tutti i nodi vicini come normale fase di advertise, invierà una sola volta ogni messaggio.

Di fatto come se il DF di ogni nodo fosse sempre a 1. Ovviamente questo porta ad una diffusione del messaggio nella rete molto più lenta.

A livello implementativo mi è bastato estendere la classe principale del protocollo e disattivare tutti i metodi riguardanti le funzionalità sopra citate.

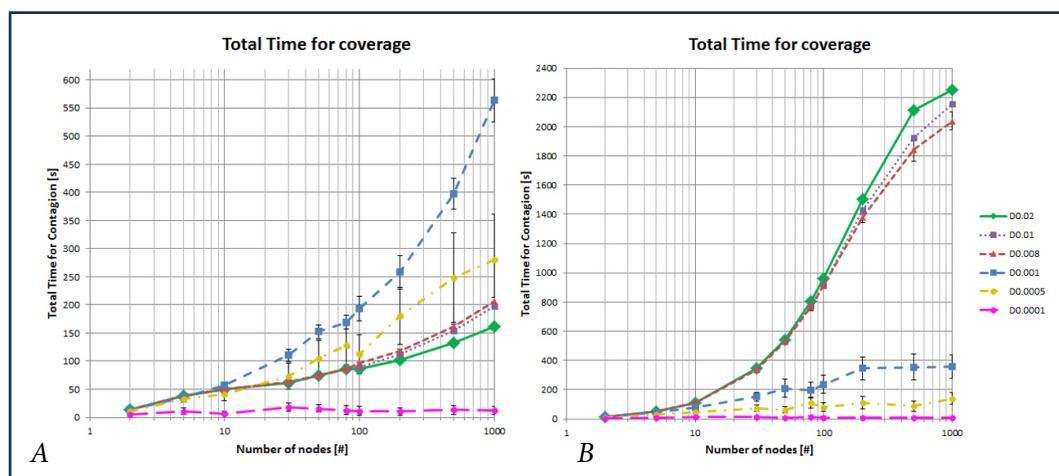


Figura 5.10 - Confronto grafici del tempo totale di trasmissione (A: Protocollo normale. - B: Protocollo senza DF)

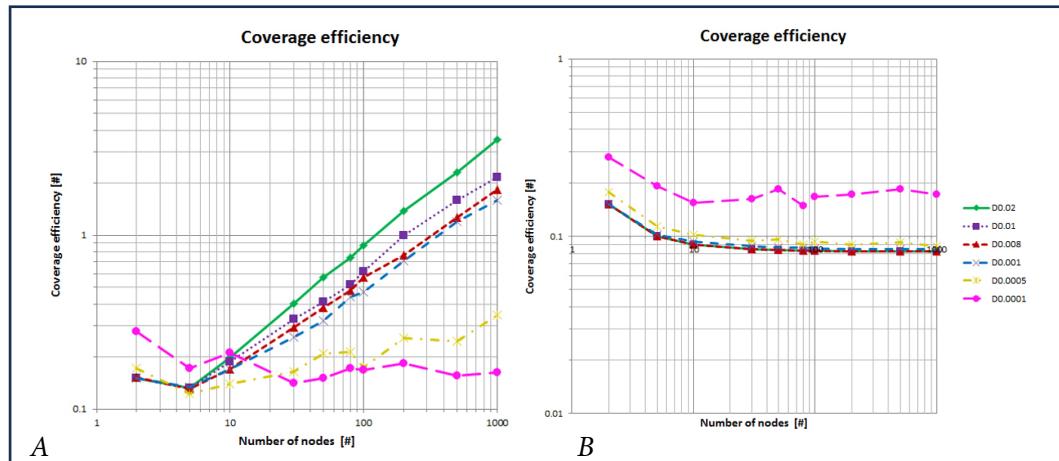


Figura 5.11 - Confronto grafici del Fattore di Efficienza (A: Protocollo normale. - B: Protocollo senza DF)

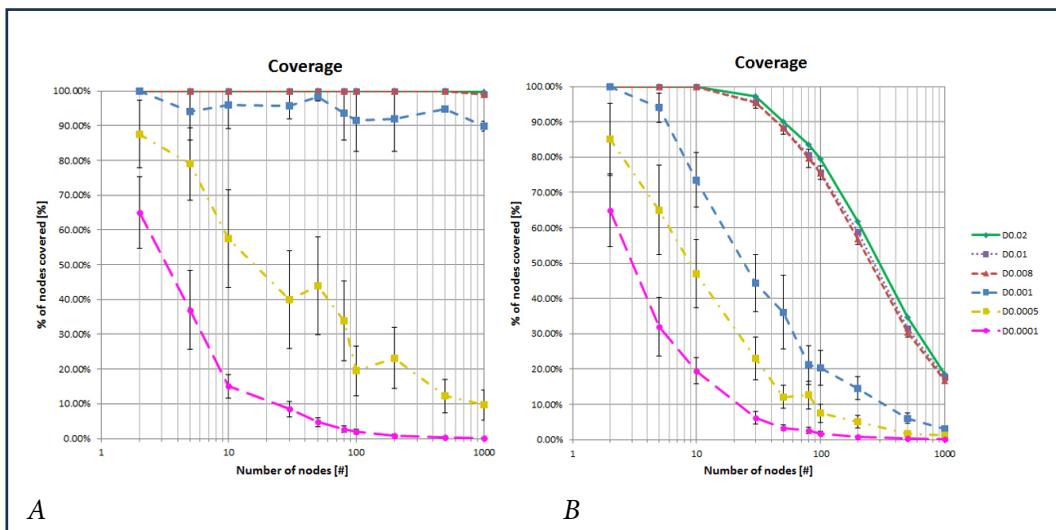


Figura 5.12 - Confronto grafici della copertura della rete (A: Protocollo normale. - B: Protocollo senza DF)

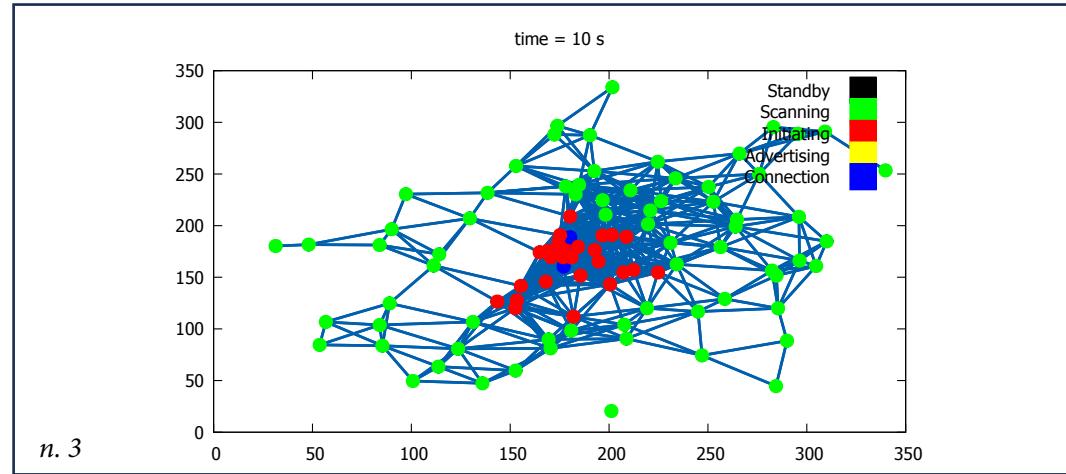
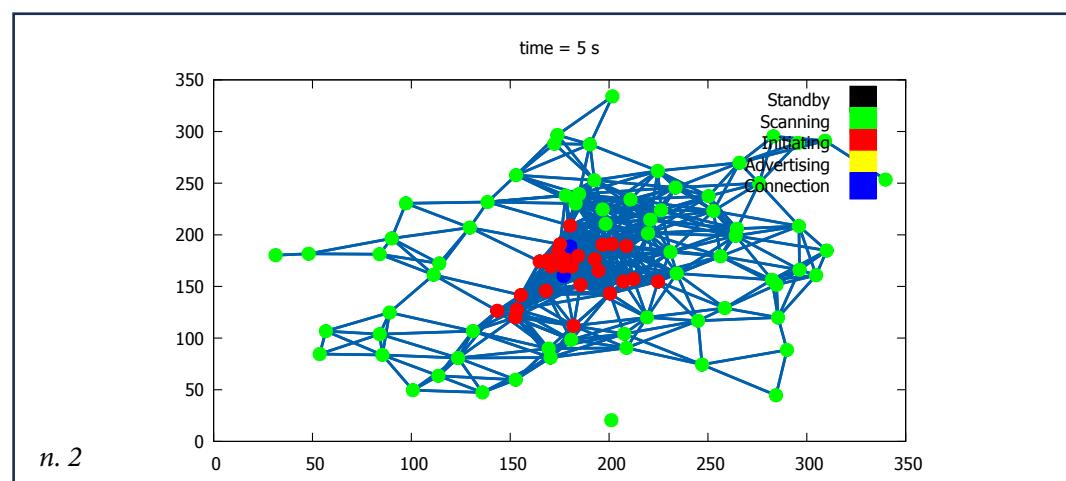
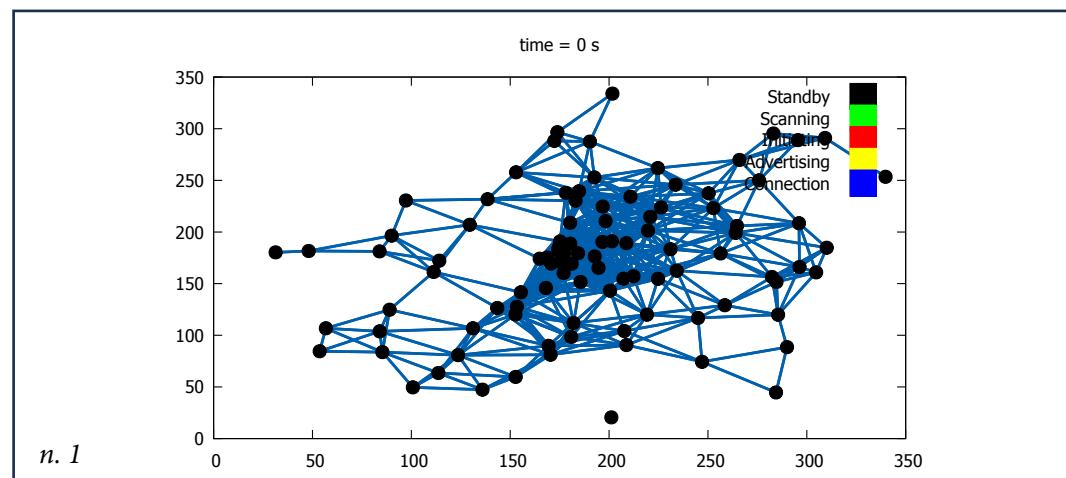
Come ci si poteva attendere le performance sono nettamente peggiori per questo caso.

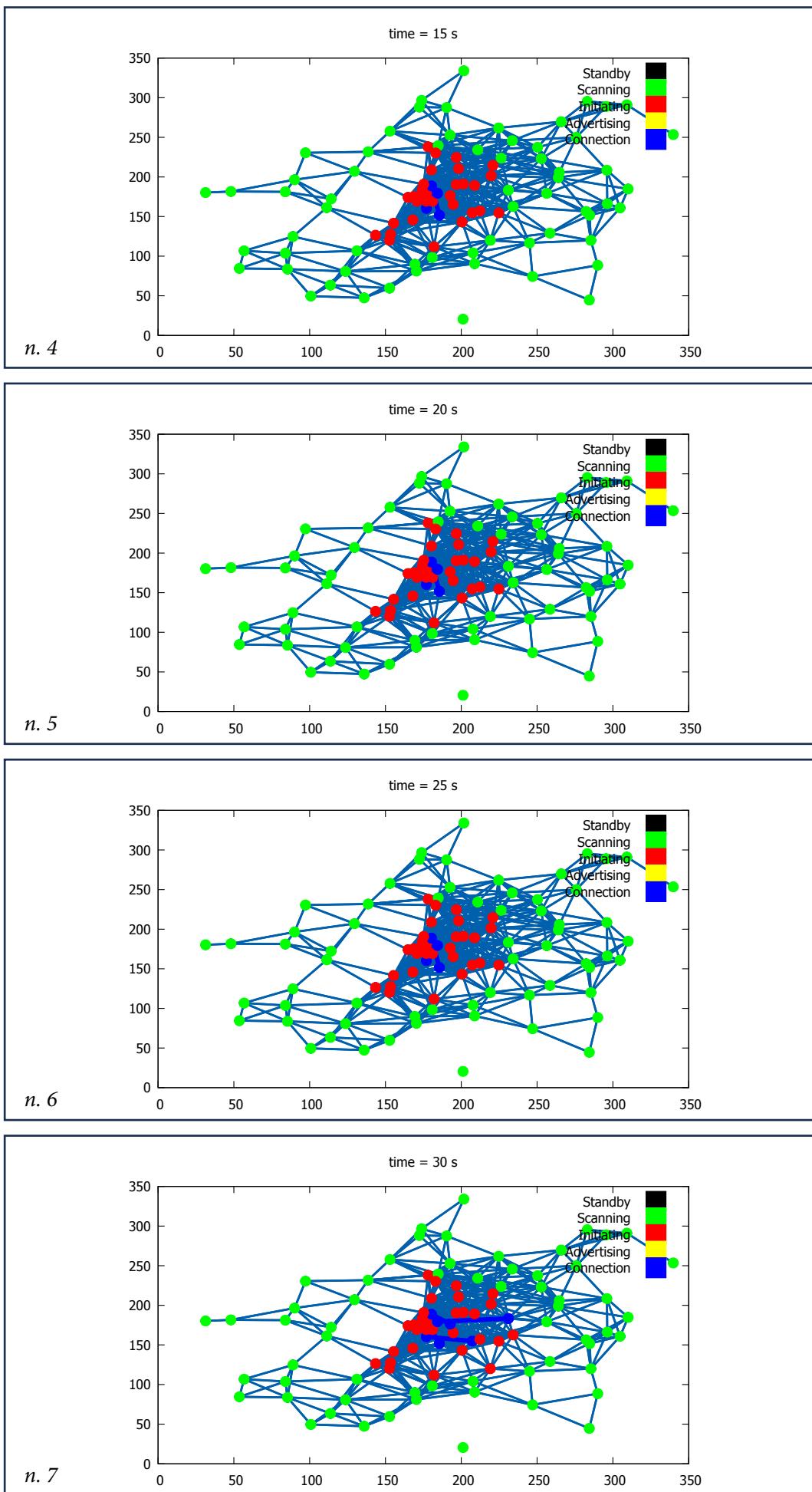
Per le tre densità più alte lo si deduce osservando i grafici dei tempi di trasferimento. Da notare inoltre che per l'asse delle ordinate, proprio per tale motivo, sono state usate dimensioni diverse.

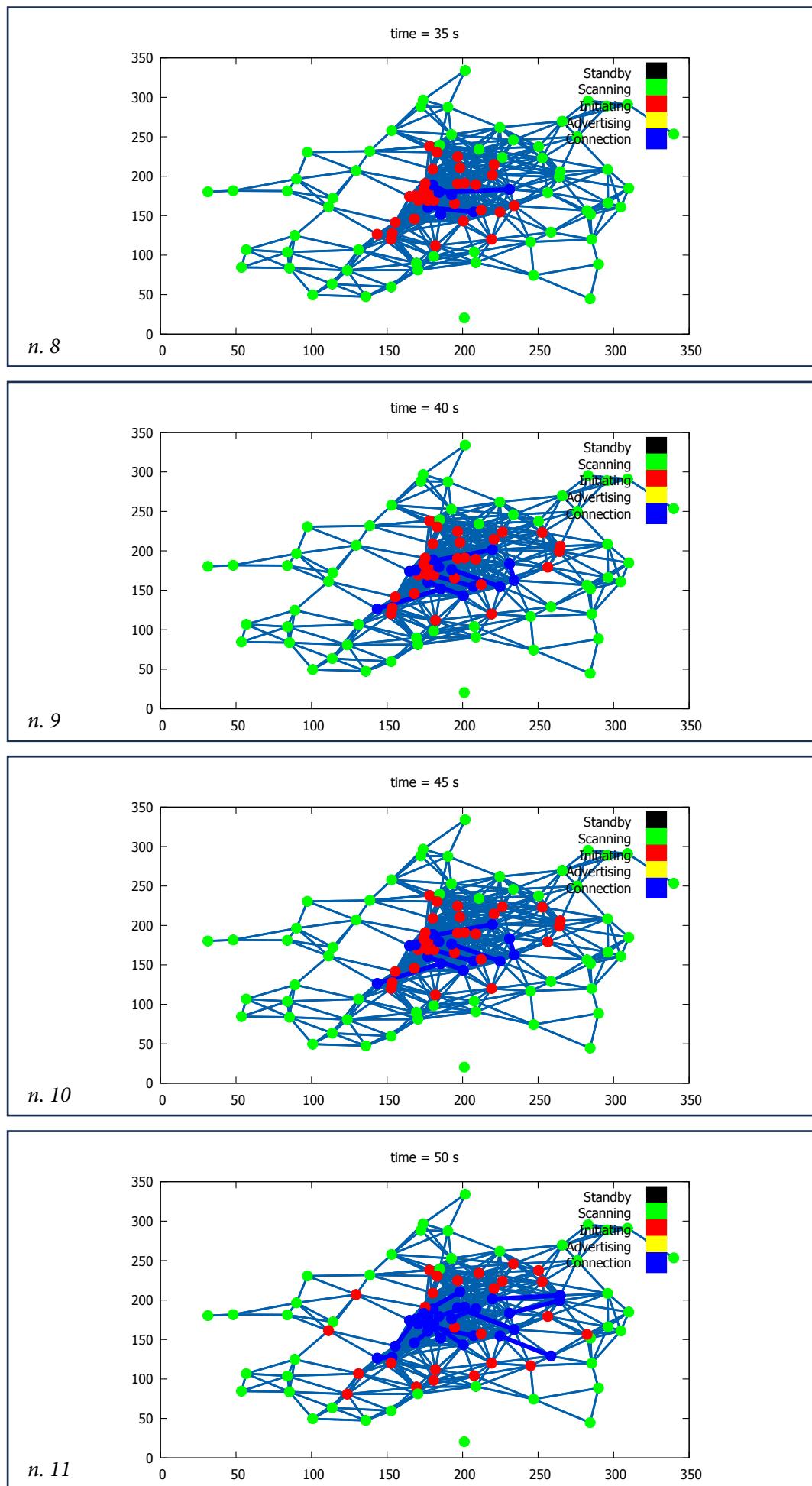
I tempi totali apparentemente sembrerebbero migliorare per le altre tre densità più basse.

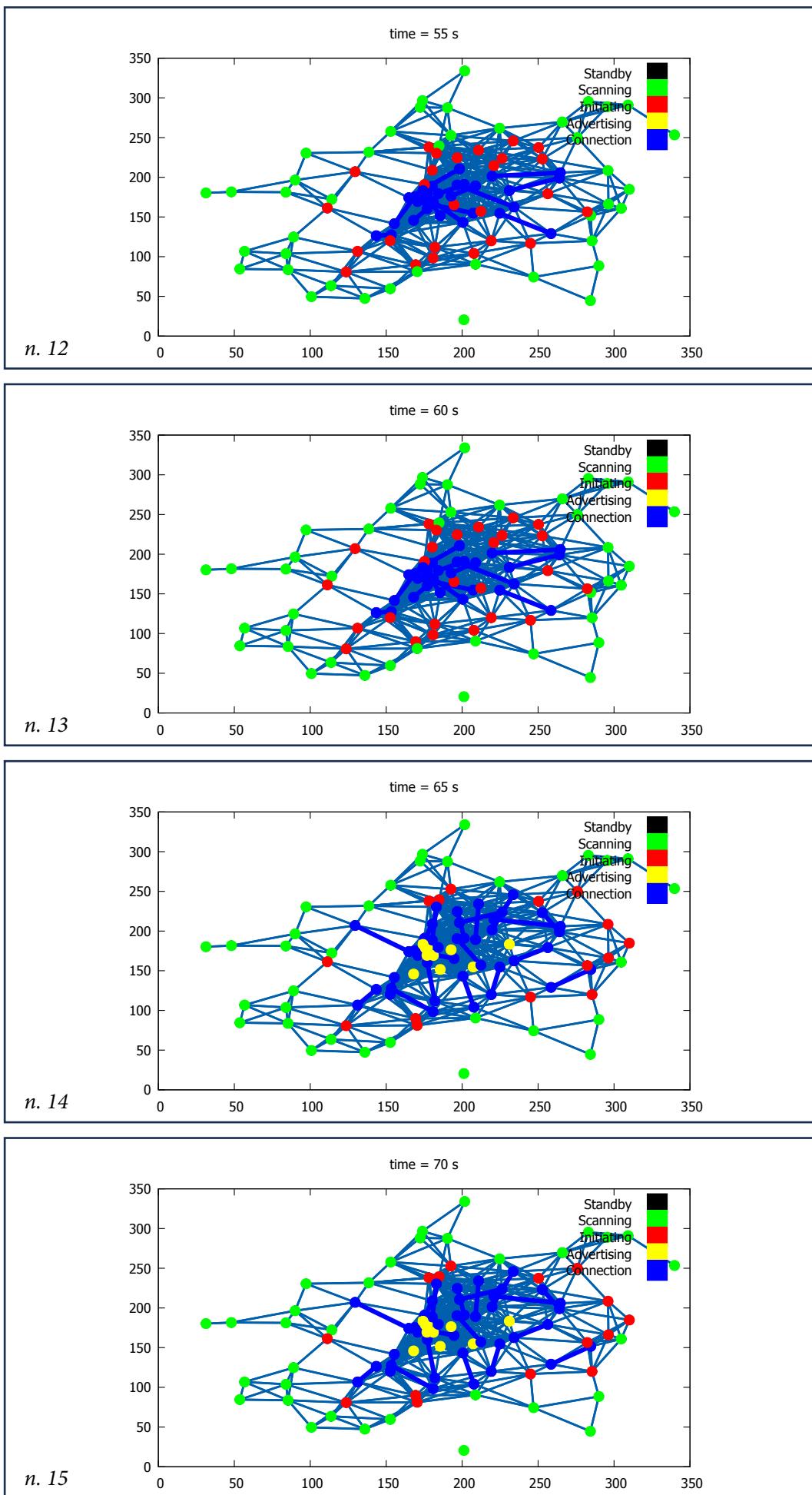
In realtà, confrontando i grafici della percentuale di contagio, appare subito evidente che la copertura crolla vertiginosamente, in particolare per queste tre densità più basse.

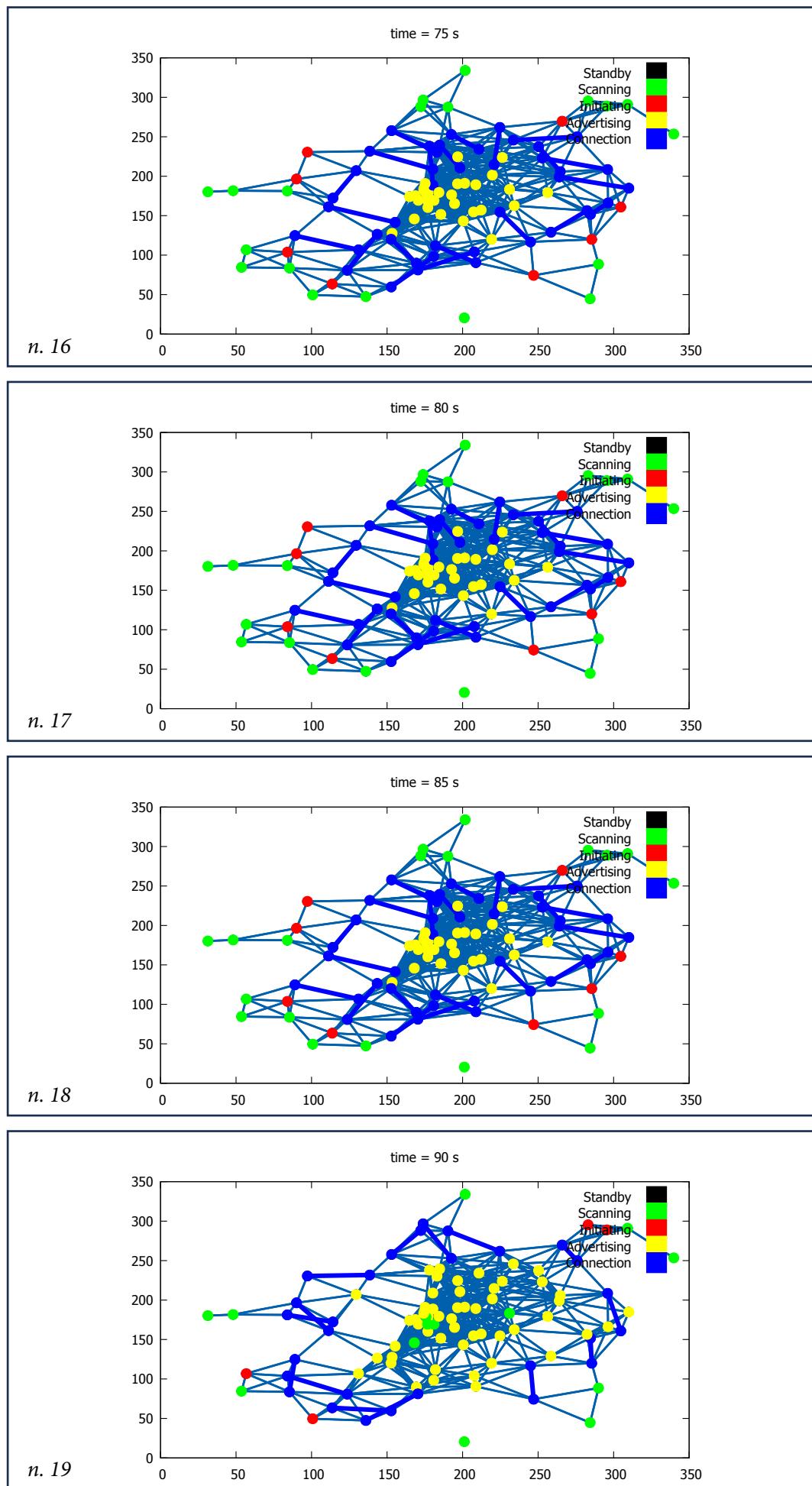
5.1.4 | Estensioni: ambiente urbano e mobilità dei nodi

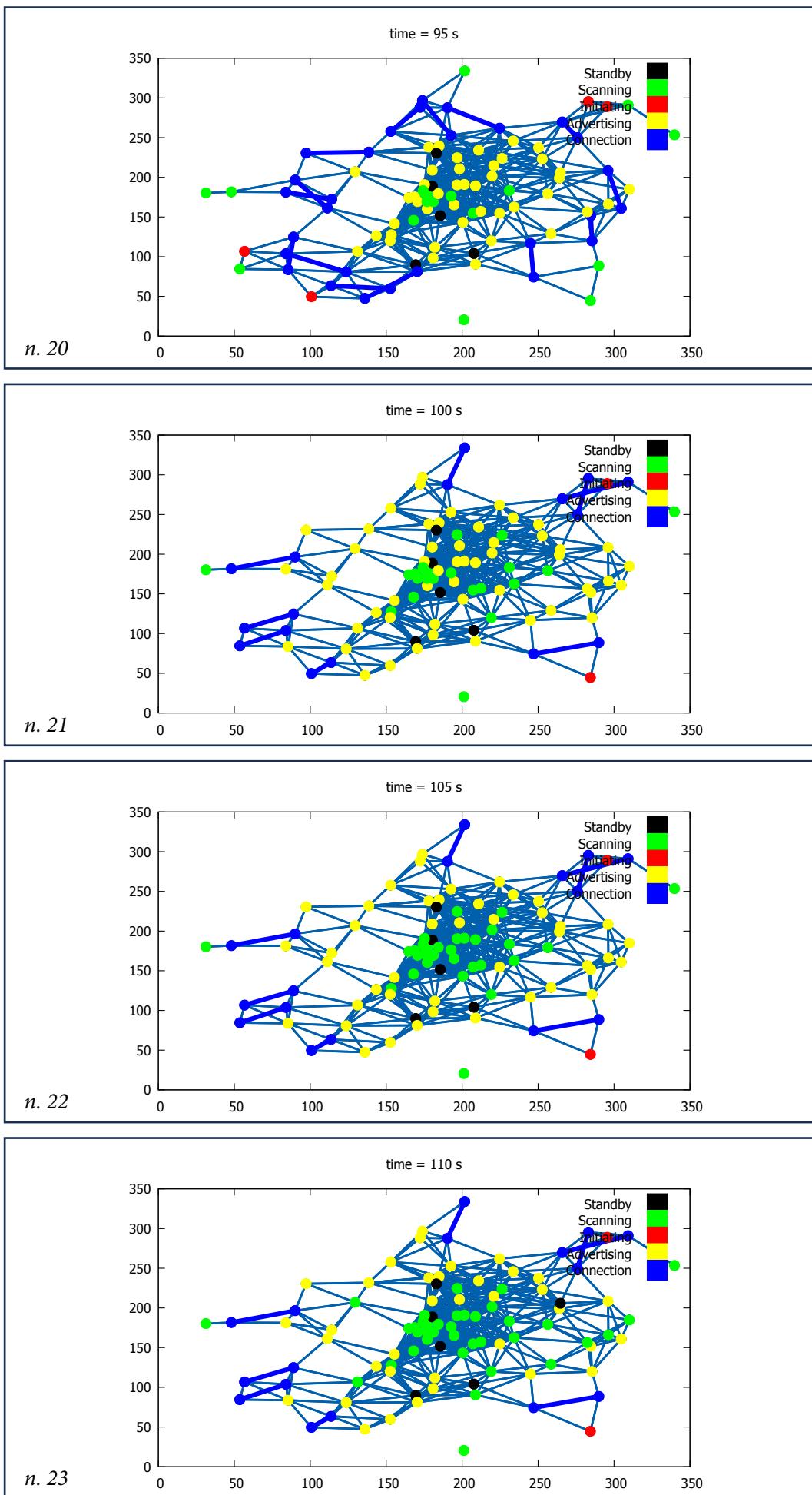












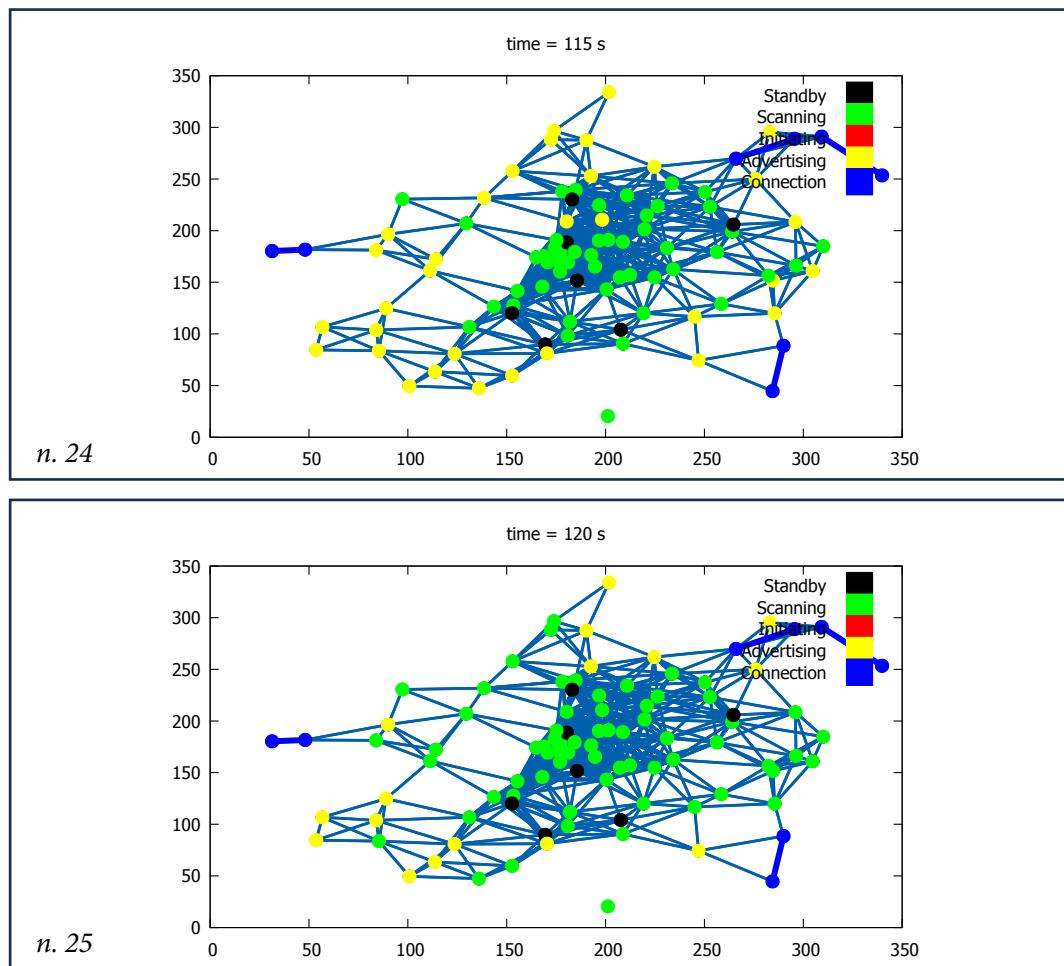


Figura 5.13 (numerate da 1 a 25) - Evoluzione della rete in ambiente urbano

La struttura della rete su cui appoggia la prima versione del protocollo era abbastanza semplificata: veniva utilizzato un Random Geometric Graph inserito in un'area quadrata.

A questo punto è utile dare una breve spiegazione su come venisse, a livello pratico, costruita dal simulatore la struttura della rete all'inizio di ogni esecuzione.

Appena parte la simulazione sono letti dal file di configurazione i parametri della stessa, tra cui molto importanti sono il numero di nodi, la densità degli stessi e il raggio del BLE.

In base alla densità e al numero dei nodi è calcolata l'area dentro la quale essi verranno disposti.

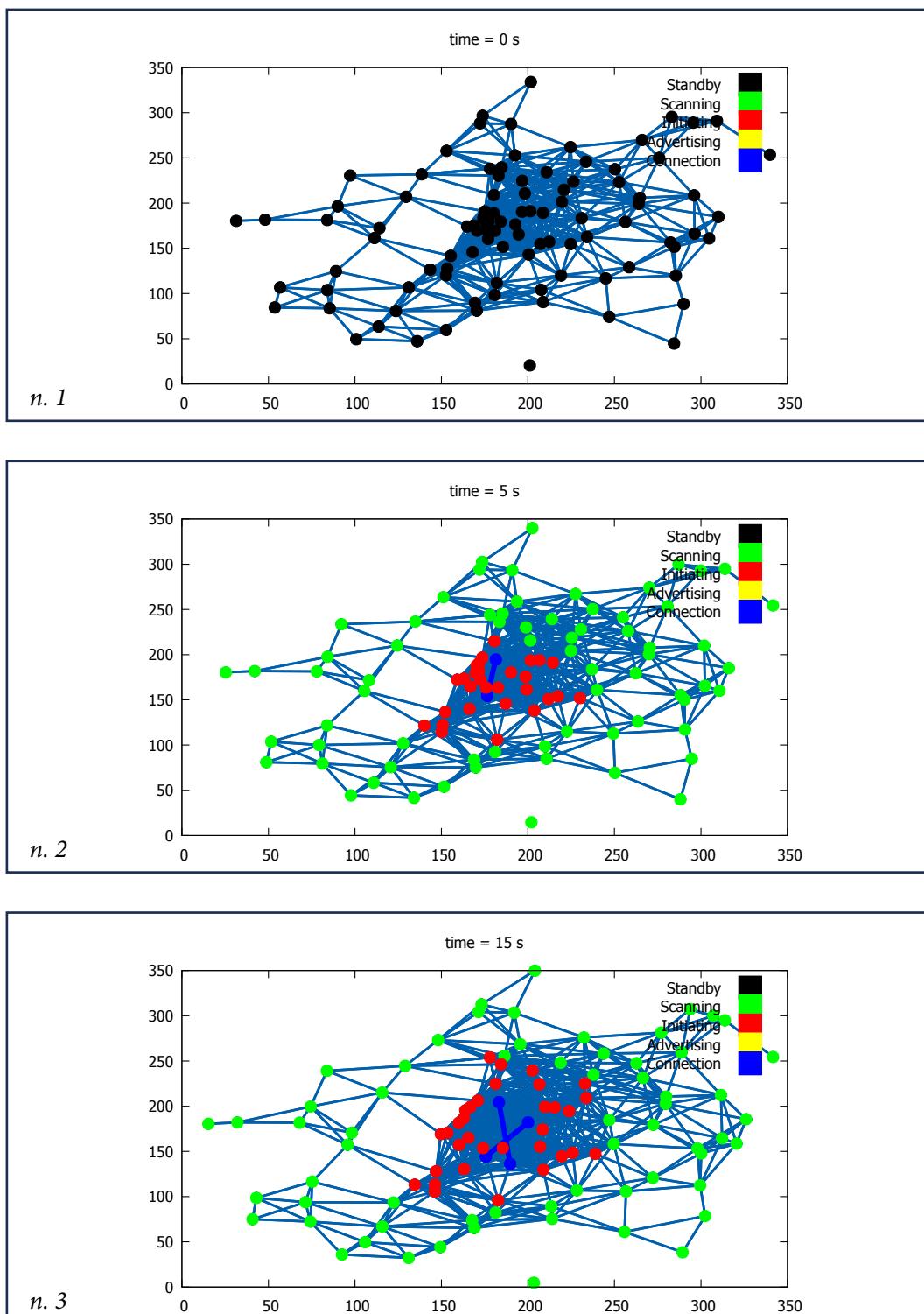
Il simulatore provvede quindi a posizionare ogni singolo nodo, assegnando coordinate randomiche che siano all'interno dell'area, e infine a collegare tra loro i nodi distanti entro il raggio.

Per aumentare il grado di realismo, obiettivo importante per questo lavoro, ho pensato di modellare diversamente l'area del grafo e il modo in cui venissero distribuiti i nodi.

Dopo una consultazione con un collega ingegnere, come anticipato nella [Sezione 5.1](#), sono arrivato alla conclusione che un buon modo per approssimare un nucleo cittadino fosse quello di usare una circonferenza. Per seguire meglio lo schema distributivo dei reali nuclei abitativi ho voluto aggregare un maggior

numero di nodi nel centro della circonferenza, dentro una circonferenza più piccola con raggio pari a $2/3$ di quello base. La circonferenza centrale concentra il 75% dei nodi totali, il restante 25% vanno nell'anello tra le due. Questi valori li ho trovati empiricamente, dopo vari esperimenti che mi hanno permesso di capire per quali valori soglia il funzionamento della rete cominciasse a degenerare.

In questo modo sono riuscito a progettare una struttura della rete che rappresentasse in modo più realistico un centro urbano con relativa periferia attorno.



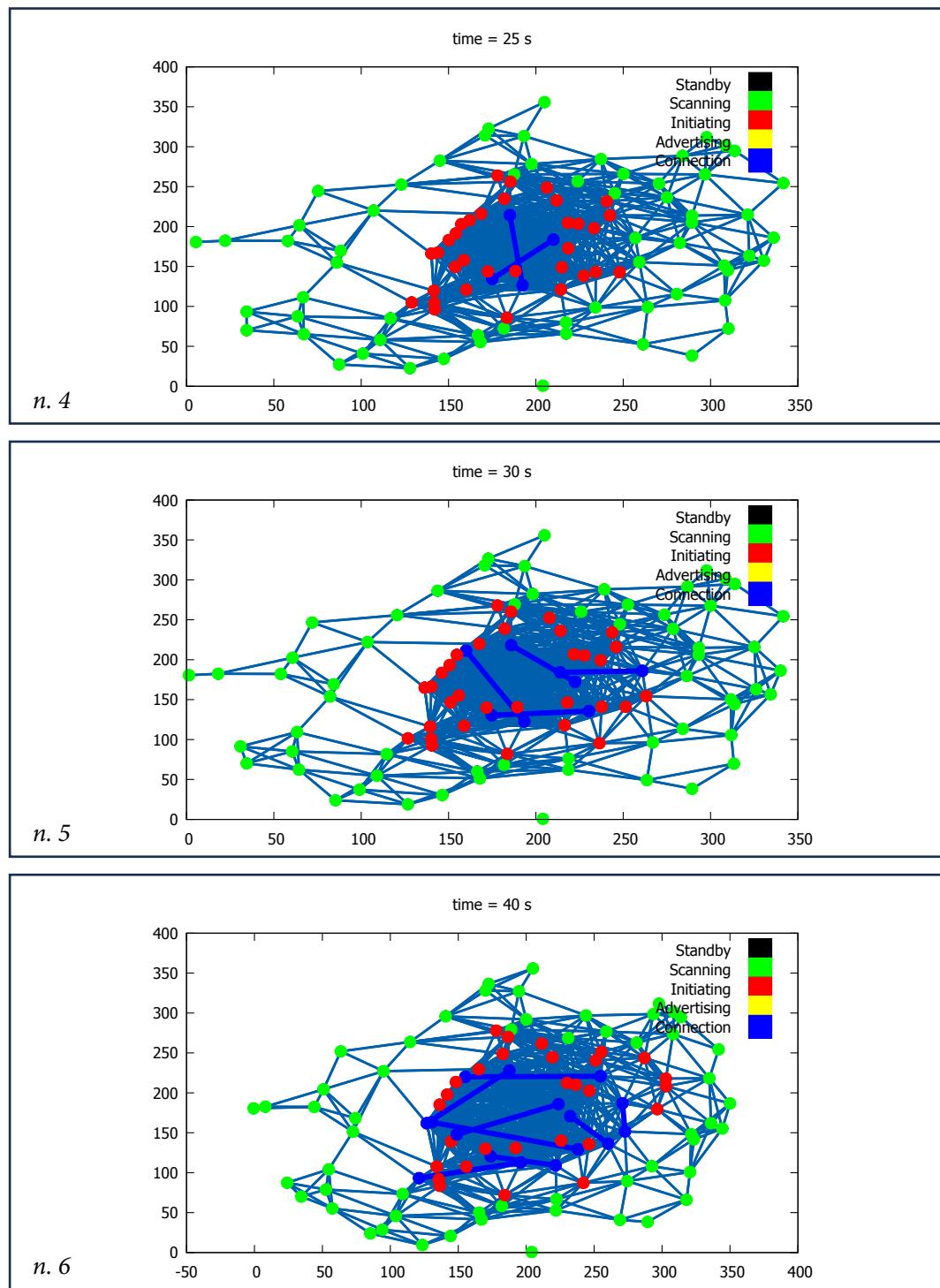


Figura 5.14 (numerate da 1 a 6) - Evoluzione della rete con mobilità dei nodi

Infine un’ulteriore estensione è servita per ottenere la mobilità dei nodi. Dispositivi contagiatriche si spostano possono incrementare le prestazioni in termini di copertura della rete, proprio come nei casi delle epidemie. Per questo motivo diventa interessante un’analisi dei risultati provenienti da simulazioni dove i nodi hanno la possibilità di spostarsi.

Il movimento, come spiegato nella [Sezione 5.1](#), consiste in nodi che si spostano lentamente dal centro verso la periferia. In questo modo viene approssimata la tendenza delle persone in preda al panico ad allontanarsi dalle zone più frequentate per raggiungere ambienti circostanti più isolati.

I risultati raccolti con queste due ulteriori estensioni possono essere confrontati con quelli ottenibili con l'utilizzo del normale protocollo.

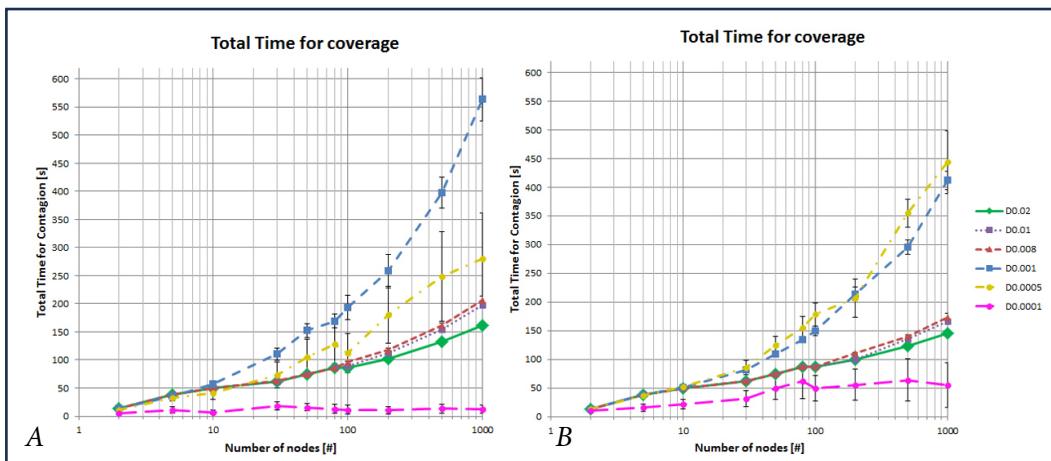


Figura 5.15 - Confronto grafici del tempo totale di trasmissione (A: Protocollo normale. - B: Protocollo con estensione ambiente urbano)

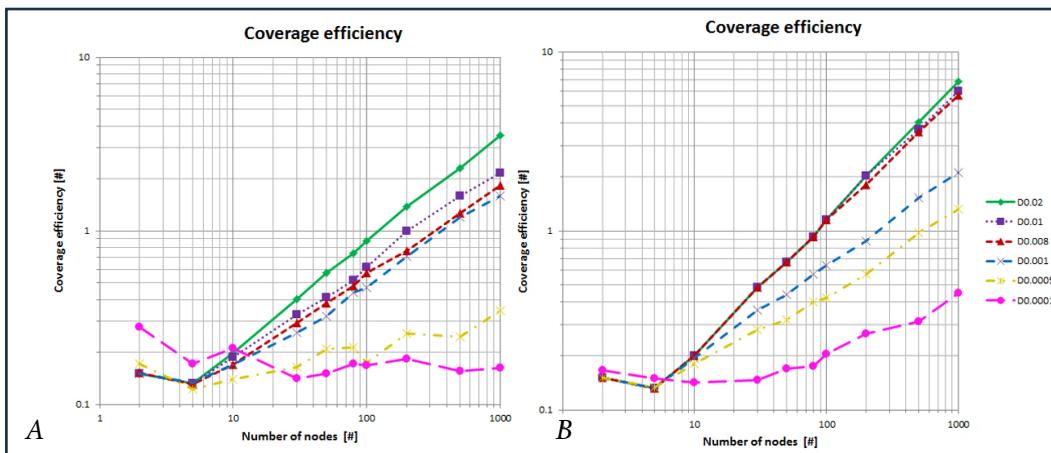


Figura 5.16 - Confronto grafici del Fattore di Efficienza (A: Protocollo normale. - B: Protocollo con estensione ambiente urbano)

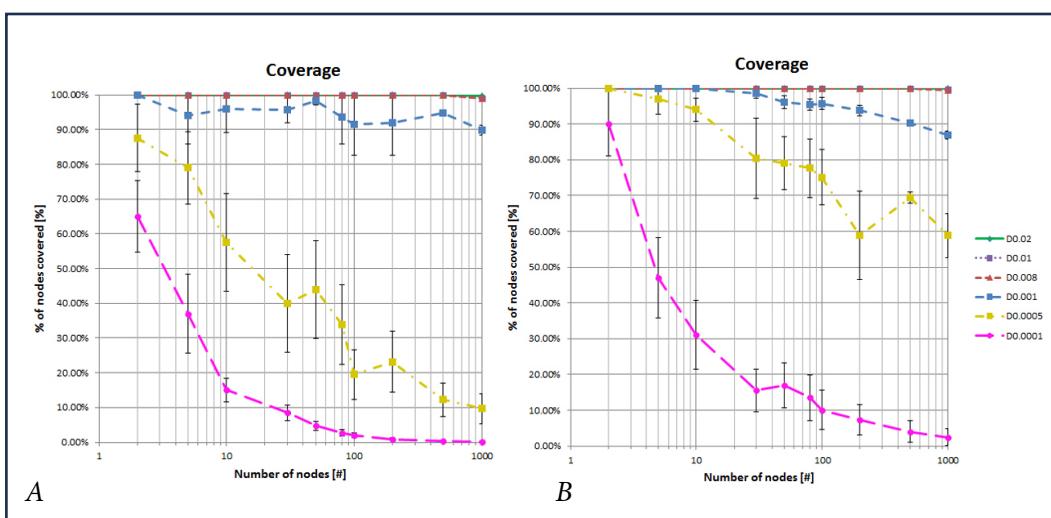


Figura 5.17 - Confronto grafici della copertura della rete (A: Protocollo normale. - B: Protocollo con estensione ambiente urbano)

Va evidenziato che le due estensioni appena trattate hanno in comune con quella multi-messaggio, vedi [Sezione 5.1.2](#), lo stesso scopo: aumentare il livello di realismo della simulazione.

A differenza di quanto accadeva per quella multi-messaggio, nei due casi che stiamo analizzando abbiamo un evidente miglioramento delle performance. In particolare l'estensione relativa all'ambiente urbano denota principalmente un miglioramento dei tempi totali, dovuto alla maggiore concentrazione dei nodi nel “centro” città.

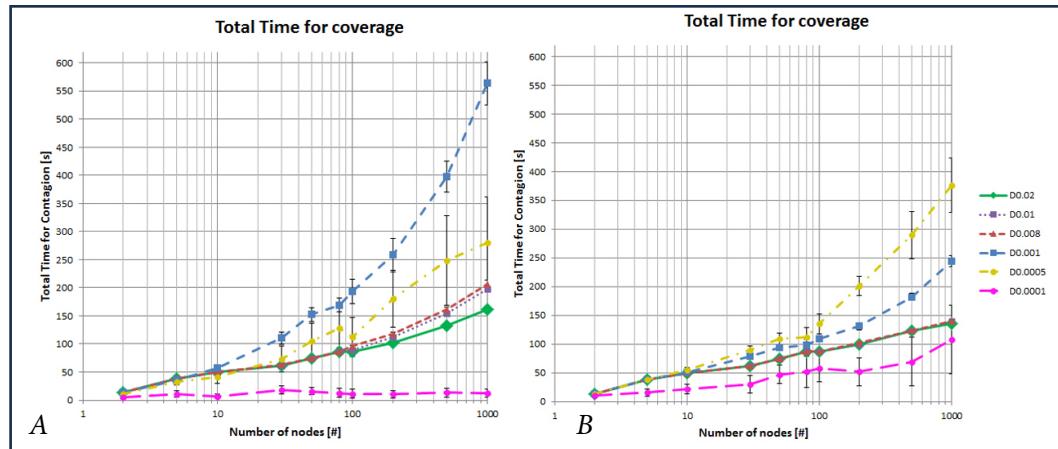


Figura 5.18 - Confronto grafici del tempo totale di trasmissione (A: Protocollo normale. - B: Protocollo con estensione mobilità dei nodi)

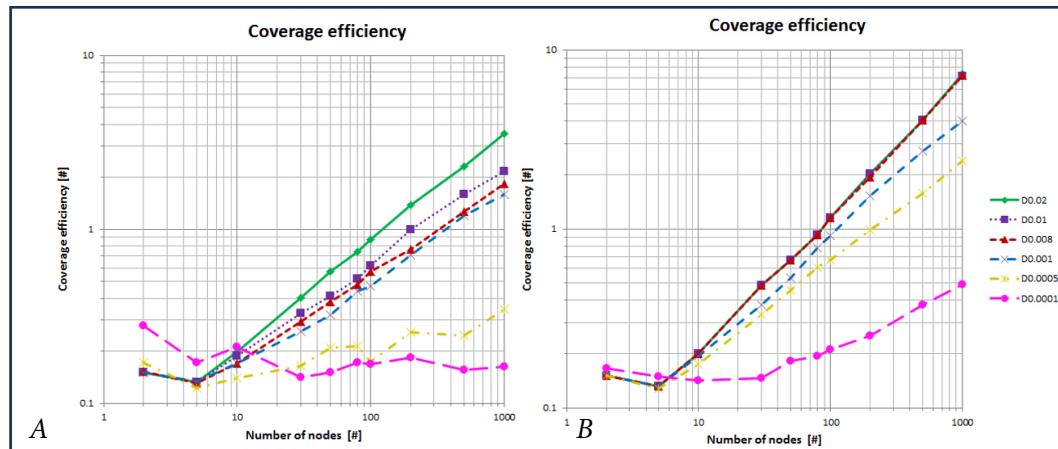


Figura 5.19 - Confronto grafici del Fattore di Efficienza (A: Protocollo normale. - B: Protocollo con estensione mobilità dei nodi)

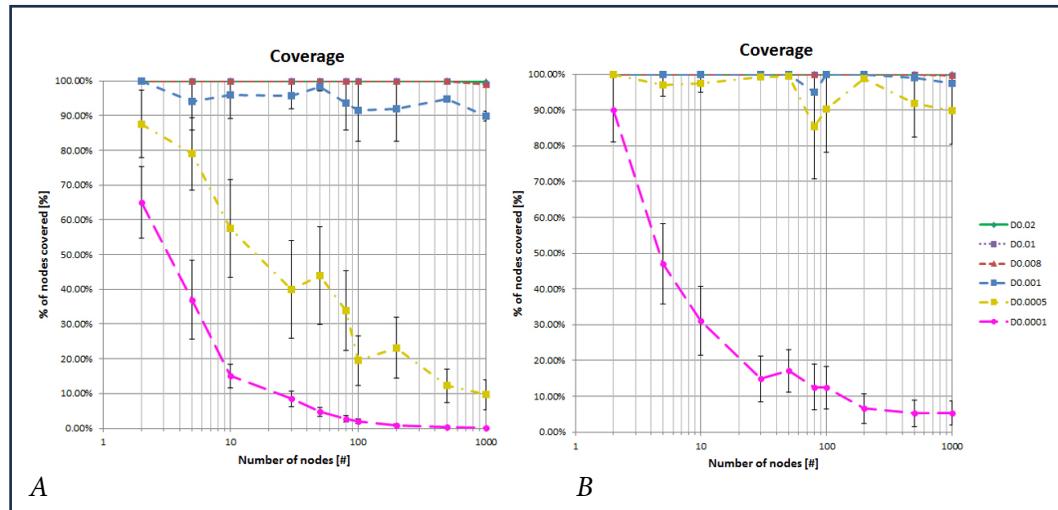


Figura 5.20 - Confronto grafici della copertura della rete (A: Protocollo normale. - B: Protocollo con estensione mobilità dei nodi)

Per quanto riguarda invece l'estensione relativa alla mobilità dei nodi emerge,

come ci aspettavamo, un notevole aumento della copertura. Nodi che si spostano possono andare ad allargare il contagio anche in altre aree. Tale miglioramento è però ovviamente legato al “grado” di movimento dei nodi.

5.2 | Estensioni: localizzazione GPS di persone disperse

Nel finale della [Sezione 5.1](#) si è discussa di un'altra possibile estensione del nostro protocollo, che permettesse di simulare la localizzazione di nodi della rete, rappresentanti persone disperse, utilizzando la tecnologia GPS. Questo aggiungerebbe al nostro programma una funzionalità diversa da quelle per cui era stato progettato originariamente.

Questa differenza emerge anche a livello implementativo in quanto le dinamiche della localizzazione di nodi sono molto diverse da quelle di funzionamento del nostro protocollo.

L'unico aspetto in comune è rappresentato dallo scambio di informazioni tra nodi usando dei messaggi, per il resto la simulazione ha funzionamento e obiettivi molto differenti. Questo porterebbe a pensare di dover progettare da zero un nuovo protocollo, invece è stato possibile implementare la classe del protocollo GPS estendendo quella del nostro protocollo.

Ho potuto fare in questo modo perché durante la fase di scrittura del codice ho voluto dargli un buon grado di riusabilità, considerando anche la struttura di Peersim.

Aspetto importante tenuto in considerazione durante l'implementazione è stato il seguire metodologie tipiche dell'ingegneria del software, in modo da dare al codice determinate proprietà volte a garantire longevità e riutilizzo per scopi futuri, anche differenti dai miei.

5.2.1 | Proprietà tipiche dell'ingegneria del software

Prima di analizzare singolarmente le varie proprietà [\[16\]](#) procediamo ad una loro breve descrizione.

Correttezza

Un software si dice corretto se si comporta in accordo a quanto previsto dalla sua specifica dei requisiti.

Affidabilità

Un sistema è tanto più affidabile quanto più raramente, durante l'uso del sistema, si manifestano malfunzionamenti.

Usabilità

Un sistema è facile da usare se un essere umano lo reputa tale.

Scalabilità

Un sistema è scalabile se può essere adattato a diversi contesti con forti

differenze di complessità (per esempio database molto piccoli o molto grandi) senza che questo richieda la riprogettazione dello stesso sistema.

Efficienza

Un sistema è efficiente se usa le risorse HW/SW in modo proporzionato ai servizi che svolge.

Riparabilità

Un sistema è riparabile se la correzione degli errori è poco faticosa. La riparabilità si persegue attraverso la modularizzazione. Non conta tanto il numero, ma piuttosto il come sono organizzati tra loro e al loro interno.

Riusabilità

Facilità con cui è possibile riusare parti di sistema per realizzare un prodotto diverso.

Manutenibilità

Facilità di apportare modifiche a sistema realizzato.

La presenza della prima proprietà è stata provata grazie al confronto dei risultati, come abbiamo mostrato nella [Sezione 4.2](#). Altre sono immediatamente derivate dall'utilizzo stesso del simulatore, infatti la struttura di Peersim garantisce affidabilità, usabilità e efficienza.

Realizzare le estensioni mi ha richiesto poca aggiunta di codice, questo prova una buona scalabilità.

Le stesse estensioni sono prova di aver ottenuto un alto livello di modularità, in quanto tutti i moduli di una tipologia sono tra loro intercambiabili, seguendo il Liskov substitution principle [\[22\]](#).

I moduli di creazione della rete, quelli protocollo e quelli per la raccolta dati possono essere usati in qualsiasi combinazione tra essi. Ad esempio si può far partire una simulazione con il protocollo normale oppure con l'estensione della coda di priorità e scegliere indifferentemente come struttura della rete quella a quadrato classica oppure quella con logica da centro abitativo, [Sezione 5.1.4](#), e inoltre decidere di raccogliere i dati utilizzando l'observer per output grafico, vedi [Sezione 4.1](#), oppure quello per salvare i risultati (i valori di copertura e tempo totale di trasmissione).

Questo perché le classi che implementano questi moduli sono ben disaccoppiate tra loro.

Laver mantenuto un'alta modularità mi ha permesso di ottenere buona riparabilità e manutenibilità. Ho avuto modo di verificarlo in più di un'occasione, avendo dovuto più volte correggere errori e in certi casi anche modificare il funzionamento di interi moduli per allinearli meglio alle specifiche.

Infine con la [Sezione 5.2.2](#) voglio portare un esempio di protocollo ottenuto estendendo la mia classe di partenza, così da mostrare il buon grado di riusabilità del mio codice.

5.2.2 | Funzionamento del protocollo di localizzazione GPS

La simulazione con il protocollo sopra citato ha l'obiettivo di stimare quante persone disperse si potrebbero localizzare e quanto tempo è necessario per farlo.

La struttura della rete rimane la stessa di tutti gli altri tipi di simulazione, in quanto si presuppone che la funzione di localizzazione sia utilizzata in contemporanea a quella di diffusione messaggi.

All'avvio del simulatore va definita, tramite il file di configurazione, la percentuale di nodi che sono segnati come dispersi sul totale, per poi verificare quanti di questi saranno trovati.

Quindi un certo numero di nodi rappresentano persone disperse mentre i restanti sono tutti quei dispositivi che, oltre ad essere collegati nella rete, sono riusciti a localizzarsi tramite GPS.

Ogni dispositivo che, per vari motivi, non riesce a utilizzare il GPS chiede a tutti i dispositivi vicini di inviargli le loro coordinate GPS così da utilizzarle per localizzarsi. Quando vengono ricevute almeno tre di esse, il protocollo sfrutta queste informazioni e le combina con la distanza dai rispettivi dispositivi per poter determinare la propria posizione.

Il calcolo della posizione è fatto intersecando tre circonferenze, aventi centro nei punti dati dalle coordinate GPS ricevute e raggio che soddisfa la misura della loro distanza dal dispositivo. Questo procedimento si basa sul metodo della triangolazione [21].

Mostreremo di seguito un esempio di esecuzione del protocollo appena discusso.

Ogni nodo disperso manda ai nodi vicini un pacchetto di connection_request per chiedere l'invio della loro posizione GPS. Un nodo che riceve tale pacchetto, se non ha scarica la batteria o non è a sua volta disperso, risponderà inviando un messaggio contenente le proprie coordinate GPS.

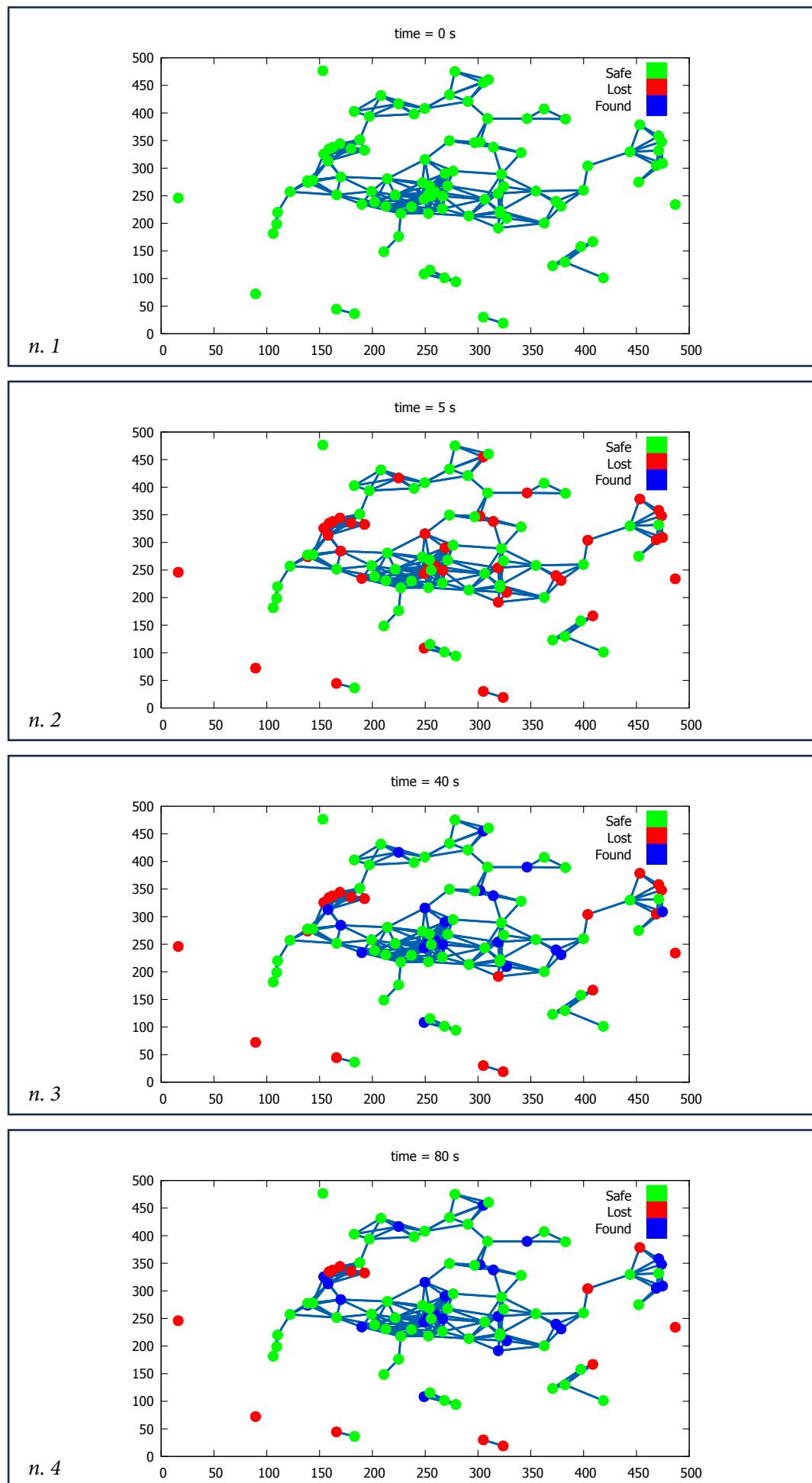
Appena un nodo disperso riceve una posizione, controlla la relativa distanza tra lui e il nodo mittente e verifica se possiede un numero sufficiente di coordinate per il processo di localizzazione.

Se ne possiede almeno tre è in grado già di calcolare la propria posizione, se ne possiede in numero maggiore riesce a localizzarsi con miglior accuratezza. A fine simulazione un apposito observer comunica il numero di nodi che sono stati ritrovati.

In questo modo riusciamo a capire quanti dei nodi dispersi sono stati ritrovati e di conseguenza qual è la percentuale di nodi dispersi per ogni densità di questi che il sistema riesce a tollerare.

Possiamo così sapere per quali casi il sistema localizzerà tutti i nodi dispersi. Vediamo di seguito un'esecuzione di una simulazione con il protocollo appena discusso.





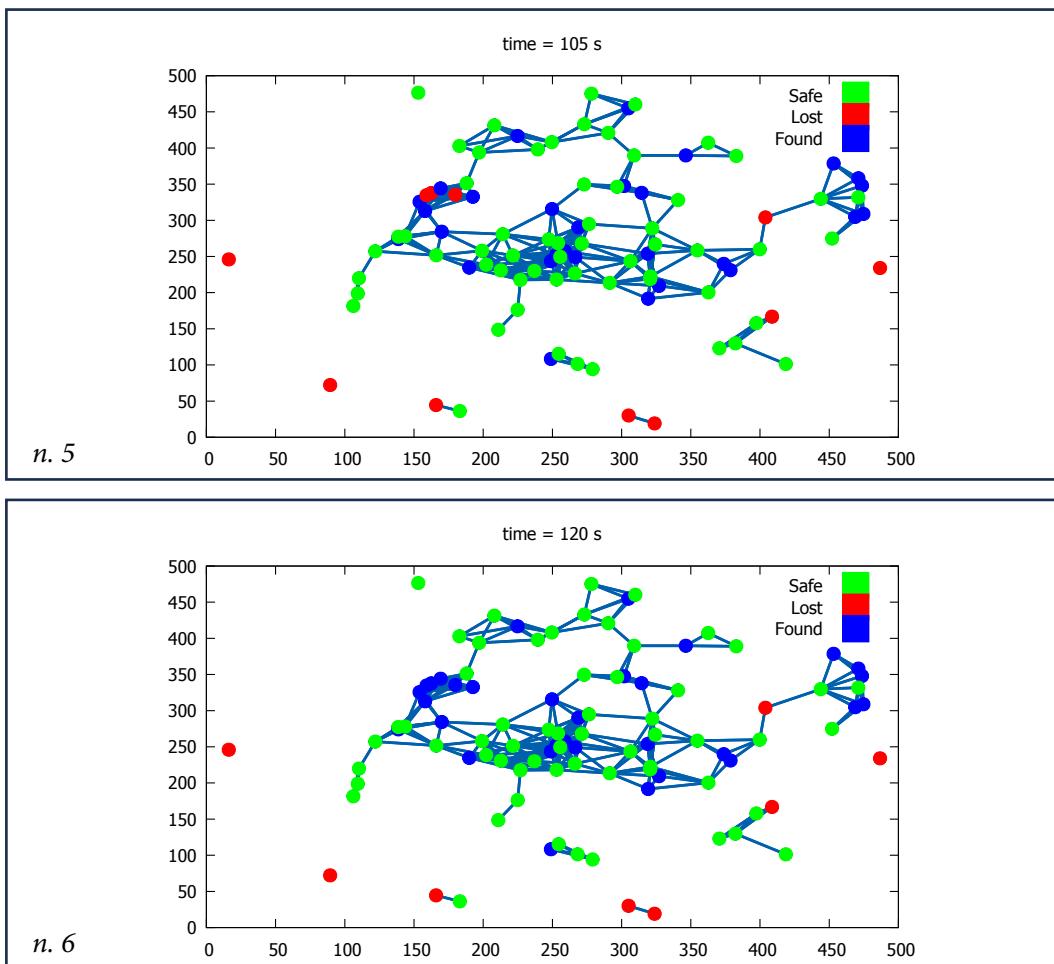


Figura 5.21 (numerate da 1 a 6) - Evoluzione della rete per localizzazione GPS

Conclusioni

In questo lavoro siamo partiti da uno studio effettuato da un ingegnere del Politecnico per sperimentare soluzioni software in grado di sopperire alle difficoltà di comunicazione dei dispositivi mobili in seguito a vari eventi catastrofici, nella convinzione che potesse essere di interesse estendere il protocollo proposto e raggiungere un più alto livello di realismo.

Abbiamo mostrato la soluzione proposta nel lavoro precedente e illustrato nel dettaglio il programma di simulazione da noi utilizzato, scelto in modo opportuno da permettere di riprodurre e estendere le precedenti simulazioni. La soluzione proposta riguardava un algoritmo adattativo, progettato come estensione di un algoritmo di gossip, che sfrutta le caratteristiche del gossip per diffondere informazioni e grazie alla sua capacità di adattamento, cerca di trovare un compromesso nel definire il carico di lavoro del dispositivo tra autonomia ed efficienza. Abbiamo esposto tale algoritmo utilizzando un esempio di funzionamento e successivamente illustrato il processo di riprogettazione dello stesso, resosi necessario a causa delle forti differenze tra le strutture dei due simulatori in questione.

Abbiamo mostrato i risultati raccolti confrontandoli con quelli del lavoro svolto dal collega.

Gran parte del lavoro è stata dedicata a realizzare le varie estensioni per il protocollo, che hanno permesso di simulare comportamenti della popolazione in modo più realistico.

In particolare i risultati più interessanti li abbiamo ottenuti con le estensioni riguardo l'ambiente urbano e la mobilità dei nodi, che hanno mostrato miglioramenti di performance.

Infine per mostrare il buon grado di riusabilità del codice abbiamo realizzato una diversa simulazione, riguardante la localizzazione GPS di persone disperse, estendendo il nostro protocollo.

In conclusione ritengo soddisfacenti i risultati raggiunti ma difficili ulteriori sviluppi di questo lavoro. Altra valutazione invece può essere fatta se consideriamo in generale solo il simulatore ed i protocolli di gestione e diffusione dei messaggi.

In tale ambito ci sono certamente innumerevoli possibilità di sperimentazione, date dalla grande versatilità di Peersim. Tali possibili studi futuri potrebbero trovare benefici da questo lavoro.

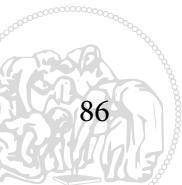
La guida, da me scritta, all'utilizzo di Peersim, presente come [Appendice A](#) di questo lavoro, può semplificare il primo approccio da parte di chiunque voglia utilizzare questo simulatore.

Il codice del protocollo presentato può rappresentare una buona base di partenza per nuovi lavori riguardanti sempre l'ambito di sistemi per la diffusione di messaggi.

Infine lo strumento, da me sviluppato, di visualizzazione grafica dell'evoluzione della rete e dello stato dei nodi, [Sezione 4.1](#), può essere, con poche modifiche,

utilizzato per qualsiasi altro tipo di simulazione che coinvolga reti e nodi con differenti stati, come abbiamo visto per il caso GPS.

Una possibile direzione futura, che un lavoro su protocolli di diffusione di messaggi potrebbe prendere, è rappresentata dal migliorare la gestione dei messaggi con eventuali pattern o modelli più accurati per l'incremento dell'efficienza. Ad esempio con l'inserimento di un Time To Live e con politiche di rinvio di messaggi al rilevamento di nuovi dispositivi nell'area circostante, analizzando la possibilità di usare modelli più elaborati rispetto agli algoritmi di tipo SIR.



Bibliografia

- [1] Christian Schindelhauer. «Epidemic Algorithms». Topic. Paderborn University. 2004. URL: <https://www2.cs.uni-paderborn.de/cs/ag-madh/WWW/Teaching/2004SS/AlgInternet/Submissions/09-Epidemic-Algorithms.pdf>
- [2] Montresor Alberto e Jelasity Márk. «Peersim: A scalable p2p simulator». In: Peer-to-Peer Computing, 2009. P2P'09. IEEE Ninth International Conference (2009). pp. 99–100
- [3] Oramhor Andy. Peer-to-Peer. Harnessing the Power of Disruptive Technologies. A cura di Oramhor Andy. p. 448
- [4] Bluetooth Core Specification 4.2 . 2 dic 2014. URL: <https://www.bluetooth.com/specifications/bluetooth-core-specification/archived-specifications>. pp. 171, 172, 2573, 2574
- [5] Gomez Carles, Oller Joaquim e Paradells Josep. «Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology». In: Sensors 2012 (9 ago. 2012). pp. 1587–1611
- [6] Perez-Palacin Diego, Mirandola Raffaela e Merseguer José. «Accurate Modeling and Efficient QoS Analysis of Adaptive Systems under BurstyWorkload». In: JOURNAL OF LATEX CLASS FILES 6.1 (gen. 2007)
- [7] Perez-Palacin Diego, Mirandola Raffaela e Merseguer José. «On the relationships between QoS and software adaptability at the architectural level». In: Journal of Systems and Software 87 (gen. 2013). pp. 1–17
- [8] Perez-Palacin Diego, Mirandola Raffaela e Merseguer José. «QoS and energy management with Petri nets: A selfadaptive framework». In: Journal of Systems and Software 85 (12 dic. 2012). pp. 2796–2811
- [9] Marzolla Moreno e Mirandola Raffaela. «Dynamic power management for QoS-aware applications». In: Sustainable Computing: Informatics and Systems 3 (4 feb. 2013). pp. 231–248
- [10] Jelasity Márk et al. PeerSim: A peer-to-peer simulator. Lesson. 2009. URL: <http://peersim.sourceforge.net>

- [11] Jelasity Márk et al. PeerSim: A peer-to-peer simulator. Lesson. 2010.
URL: <http://peersim.sourceforge.net>
- [12] Hu Ruijing et al. «A fair comparison of gossip algorithms over large-scale random topologies». In: Reliable Distributed Systems (SRDS) (nov. 2012). pp. 331–340
- [13] Pagliari Lorenzo. «Algoritmi adattativi per il risparmio energetico di sistemi broadcast via Bluetooth». 2015. URL: <https://github.com/lorenzo-pagliari/Tesi>
- [14] Pagliari Lorenzo. Codice per Omnet++. «Dynamic Fanout». URL: <https://github.com/lorenzo-pagliari/Tesi/tree/master/ProjectSimulationFolder/prova1>
- [15] Pagliari Lorenzo, Mirandola Raffaela, Perez-Palacin Diego e Trubiani Catia. «Energy-aware adaptive techniques for information diffusion in ungoverned peer-to peer networks». In: 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA) (apr. 2016)
- [16] Peron Adriano. Università degli Studi di Napoli Federico II. Corso di ingeneria del software 1. «Qualità esterne ed interne del SW». pp. 42–45
- [17] Wu Xian, Mazurowski Maciej, Chen Zhen e Meratnia Nirvana. «Emergency message dissemination system for smartphones during natural disasters». In: International Conference on Telecommunications (ITST). 2011. pp. 258–263
- [18] Jelasity Márk, Voulgaris Spyros, Guerraoui Rachid, Kermarrec Anne-Marie e van Steen Maarten. «Gossip-based peer sampling». ACM Trans. Comput. Syst. vol 25. 2007.
- [19] Jelasity Márk, Montresor Alberto e Babaoglu Ozalp. «Gossip-based aggregation in large dynamic networks». ACM Trans. Comput. Syst. vol 23. no 3. 2005. pp. 219–252
- [20] Romano Bernardo. «Evaluation of urban fragmentation in the ecosystems». L'Aquila. ott 2002. pp. 4–5
- [21] Bianconi Marco. «Sviluppo di algoritmi per la triangolazione e stima della posizione di nodi basata su dati GPS». Masters thesis, University of Trento. 2005.
- [22] Briand Lionel, Dzidek James et al. «Instrumenting contracts with aspect-oriented programming to increase observability and support debugging». In: 21st IEEE International Conference on Software Maintenance (ICSM'05) (set. 2005)

Appendice A

Peersim - How to use it

I sistemi peer-to-peer (P2P) possono essere estremamente grandi (milioni di nodi). I nodi si uniscono e lasciano la rete continuamente. Sperimentare con un protocollo in un tale contesto non è affatto un compito facile.

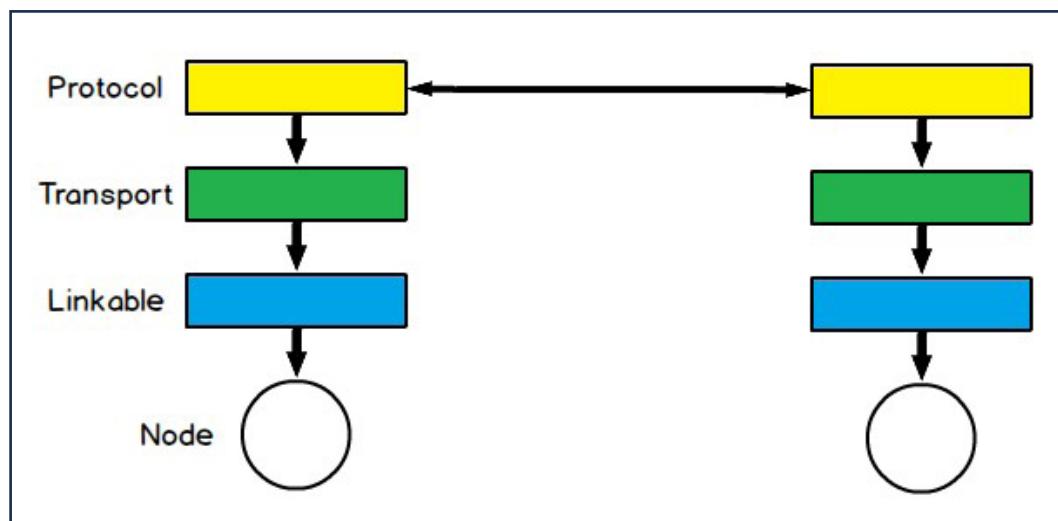
PeerSim è stato sviluppato per far fronte a queste problematiche e quindi raggiungere un'estrema scalabilità e supportare il dinamismo. Inoltre, la struttura del simulatore si basa su componenti e semplifica la rapida prototipazione di un protocollo, combinando diversi blocchi predefiniti, che sono appunto oggetti Java.

L'ovvio vantaggio dell'utilizzo di Peersim è rappresentato dal dover imparare a programmare due sole tipologie di componenti (di classi Java) e per tutto il resto potersi appoggiare su un sistema già funzionante, di cui serve capire le meccaniche di base, piuttosto che dover scrivere tutto da zero.

Questo tutorial si pone lo scopo di fornire informazioni utili per progettare e realizzare simulazioni personalizzate, in base alle esigenze dell'utente, risparmiando così la lettura del codice sorgente del programma altrimenti necessaria.

In particolare sarà necessario capire come scrivere le classi utili per la creazione di un proprio protocollo personalizzato, senza avere profonda comprensione di tutti i componenti.

Verrà usato un approccio descrittivo senza mostrare esempi specifici, ma piuttosto illustrando in generale le dinamiche di funzionamento del simulatore nel suo complesso.

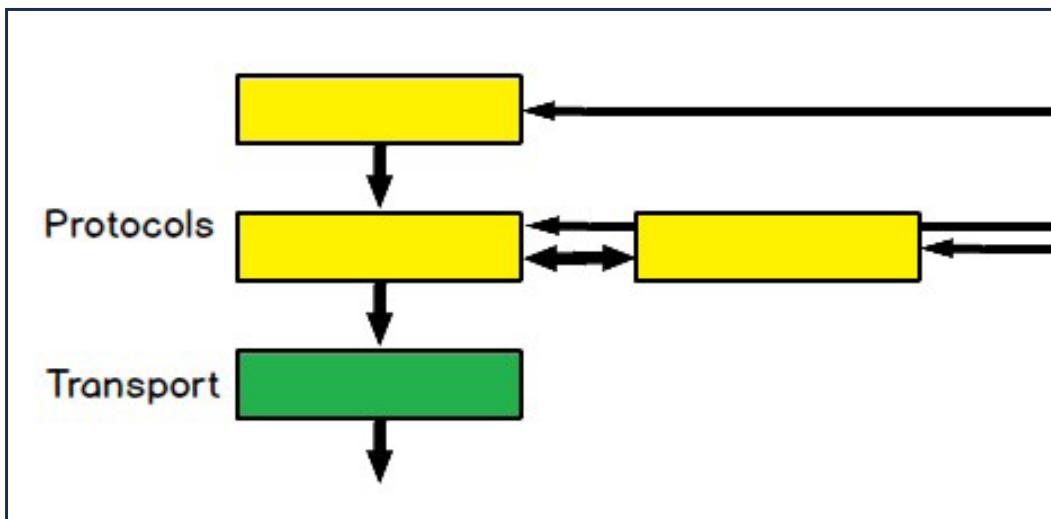


Componenti interni ai nodi

Per la quasi totalità delle simulazioni personalizzate sarà sufficiente imparare a scrivere due sole tipologie di componenti: i *Protocol*, classi che implementano l'interfaccia *Protocol*, e i *Control*, classi che implementano l'interfaccia *Control*.

La prima tipologia chiaramente tratta di protocolli, quello/i su cui si vuole sperimentare e quelli ausiliari ad esso, mentre la seconda di controllori che osservano i protocolli o/e agiscono modificando lo stato di questi ultimi.

La differenza fondamentale sta nel modo in cui essi vedono la rete; i *Control* vedono tutti i nodi e la rete nella sua totalità, quindi hanno accesso a tutte le informazioni di quest'ultima; i *Protocol* sono contenuti in ogni nodo, possono agire sul nodo stesso e sui soli nodi vicini di cui hanno visione.



Vari livelli di Protocol

I *Protocol* inoltre, dentro il singolo nodo, possono essere disposti su più livelli, formando una pila, in modo del tutto analogo a quella della rappresentazione ISO/OSI o TCP/IP per esempio.

Vi è anche la possibilità di mettere più protocolli, in parallelo, nello stesso livello. Ogni istanza di un protocollo contenuta in un nodo avrà accesso ai dati e alle risorse delle istanze degli altri protocolli di livello minore o uguale, contenuti nel nodo stesso, e delle istanze dello stesso protocollo contenute negli altri nodi.

Molto utile è dare una lettura ai *Protocol* e ai *Control* già esistenti, da un lato per poter osservare le best practice usate nella scrittura di questi ultimi, dall'altro per evitare di dover scrivere da zero classi che in alternativa si potrebbero ottenere estendendo o prendendo spunto da quelle esistenti.

Entrambe le tipologie di componenti hanno due caratteristiche importanti.

La prima, fondamentale, è la periodicità con cui si attivano, ad ogni componente ne può essere assegnata una differente.

La seconda riguarda con quali componenti sono collegati (quindi per i *Protocol* la scelta del livello).

Implica a quali dati e risorse di altri componenti hanno accesso, quindi cosa possono modificare e cosa possono sfruttare. Da ricordare che il collegamento è unidirezionale.

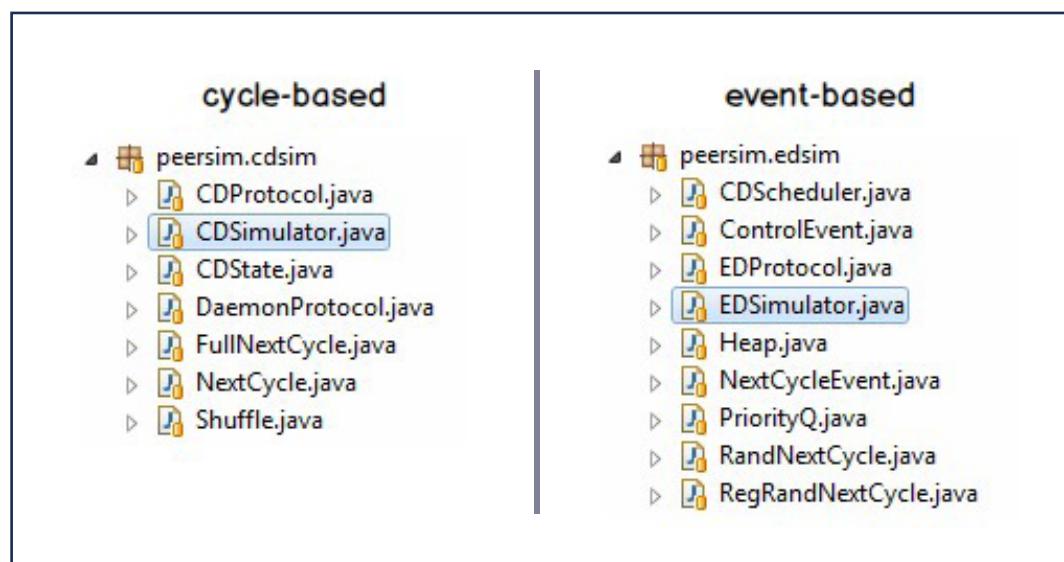
Il poter gestire velocemente e semplicemente queste due caratteristiche, a livello di codice, è uno dei punti di forza di Peersim, che ne evidenzia la sua versatilità.

Altra cosa importante da sapere è che le istanze di componenti non possono essere generate dall'utente nel suo codice altrimenti il simulatore non ne gestirà il funzionamento.

Per osservare l'andamento della simulazione si utilizzano *Control* specifici per questo scopo, che hanno quindi la caratteristica di observer. Essi possono essere usati per stampare dati anche su file.

PeerSim supporta due modelli di simulazione: il modello event-based e il modello cycle-based. Quest'ultimo modello è semplificato, il che rende possibile raggiungere un'estrema scalabilità e prestazioni, al costo di alcune perdite di realismo. Tuttavia molti protocolli semplici possono tollerare questa perdita senza problemi.

Le ipotesi semplificative del modello cycle-based sono la mancanza di simulazione del livello di trasporto e la mancanza di concorrenza. In altre parole, i nodi comunicano direttamente tra loro, e ai nodi viene dato il controllo periodicamente, in ordine sequenziale, in modo che possano eseguire azioni arbitrarie. Da notare che è relativamente facile migrare qualsiasi simulazione cycle-based al motore event-based.



Cycle-based e Event-based

La differenza fondamentale tra i due modelli è la diversa visione del tempo; in cycle-based viene dato in “cicli di esecuzione” mentre in event-based in valori di tempo, quantità di “unità di tempo”.

Una volta scelta l’unità di misura del tempo (quella di *simulation.endtime*) si dovrà rimanere coerenti con essa ogni volta che si avrà necessità di inserire o manipolare un valore di tempo.

PeerSim è stato progettato per incoraggiare la programmazione modulare basata su oggetti. Ogni blocco è facilmente sostituibile da un altro componente che implementa la stessa interfaccia (vale a dire, la stessa funzionalità). L’idea generale del modello di simulazione è:

- scegliere la dimensione della rete (numero di nodi)
- scegliere uno o più protocolli da sperimentare, e inizializzarli
- scegliere uno o più oggetti *Control* per monitorare le proprietà a cui si è interessati e modificare alcuni parametri durante la simulazione (ad esempio, la dimensione della rete, lo stato interno dei protocolli, ecc.)

- eseguire la simulazione invocando la classe Simulator con un file di configurazione, che contiene le informazioni di cui sopra

Il ciclo di vita di una simulazione è il seguente. Il primo passo è leggere il file di configurazione, dato come parametro da riga di comando o dall'Editor. La configurazione contiene tutti i parametri di simulazione relativi a tutti gli oggetti coinvolti nell'esperimento.

Quindi il simulatore imposta la rete inizializzando i nodi nella rete e i protocolli in essi contenuti. Ogni nodo ha gli stessi tipi di protocolli; cioè le istanze di un protocollo formano un array nella rete, con un'istanza in ciascun nodo. Le istanze dei nodi e dei protocolli sono create tramite clonazione. Viene generata una sola istanza usando il costruttore dell'oggetto, che funge da prototipo, e tutti i nodi della rete sono clonati da questo prototipo. Per questo motivo, è molto importante prestare attenzione durante l'implementazione del metodo di clonazione dei protocolli.

Se nei nodi e nei protocolli sono presenti variabili di tipi non primitivi va fatto con deep clone.

A questo punto, è necessario eseguire l'inizializzazione, che imposta gli stati iniziali di ciascun protocollo. La fase di inizializzazione viene eseguita dai *Control* pianificati per l'esecuzione solo all'inizio di ogni esperimento. Nel file di configurazione i componenti di inizializzazione sono facilmente riconoscibili dal prefisso init. Questi oggetti initializer sono semplicemente *Control*, configurati per l'esecuzione nella fase di inizializzazione.

Dopo l'inizializzazione il motore richiama tutti i componenti *Protocol* e *Control* una volta in ogni ciclo, fino a un determinato numero di cicli o fino a quando un componente decide di terminare la simulazione. In Peersim tutti gli oggetti *Protocol* e *Control* sono assegnati a un oggetto Scheduler che definisce quando verranno eseguiti esattamente. Per impostazione predefinita, tutti gli oggetti vengono eseguiti in ogni ciclo. Tuttavia è possibile configurare un protocollo o un controllo per l'esecuzione solo in determinati cicli ed è anche possibile scegliere l'ordine di esecuzione dei componenti all'interno di ciascun ciclo.

Nel modello event-based tutto funziona esattamente nello stesso modo del modello cycle-based, eccetto la gestione del tempo e il modo in cui ai protocolli viene passato il controllo. *Protocols* che non sono eseguibili (usati solo per memorizzare dati) possono essere applicati e inizializzati esattamente nello stesso modo. È possibile utilizzare anche i *Controls* di qualsiasi package esterno al package *peersim.cdsim*. Per impostazione predefinita, i *Controls* vengono chiamati in ogni ciclo nel modello cycle-based. Nel modello event-based devono essere schedulati esplicitamente, poiché non ci sono cicli.

Scelta la visione del tempo che è più congeniale, si deve stabilire quanti componenti servono per gestire il sistema che vogliamo realizzare.

Per ogni componente va decisa la periodicità e i collegamenti con gli altri componenti.

La periodicità, come detto in precedenza, nel modello cycle-based ha la granularità del ciclo, mentre nel modello event-based è espressa in unità di tempo.

Questo secondo modello quindi ci permette di ottenere maggior realismo da questo punto di vista.

Per questo si consiglia allo sviluppatore di scrivere i tempi utilizzando una convenzione che renda palese l'unità di misura temporale scelta anche per gli altri lettori del codice. Per esempio specificando per la durata della simulazione `60*60*1000`, allora l'interpretazione naturale è che l'unità di misura sia in millisecondi. Si ottiene così una simulazione di durata di 1 ora, anche se si avrebbe lo stesso risultato scrivendo 1 e tenendo tutti gli altri tempi in ore. Sarebbe del tutto analogo, ma l'unità di misura risulterebbe sicuramente molto meno "visibile".

Il numero di unità di tempo indicanti la durata totale della simulazione va specificato tramite il file di configurazione, in `simulation.endtime`

Anche la periodicità di ogni componente è specificata nel file di configurazione per i *Protocols* in `protocol.NOME.step` e per i *Controls* in `control.NOME.step` e indica ogni quante unità di tempo il componente si attiverà.

Inoltre per ogni componente può essere indicato un tempo di inizio, differente dal tempo 0, e un tempo di fine esecuzione, differente da `simulation.endtime` indicato in `protocol(control).NOME.from` e in `protocol(control).NOME.until`

In alternativa il componente può essere impostato per una sola esecuzione singola al tempo indicato in `protocol(control).NOME.at`

Da notare inoltre che è possibile indicare un valore di tempo anche per il delay di un evento (ad esempio un messaggio tra un nodo e un altro).

Per quanto riguarda il collegamento tra i vari componenti, è necessario valutare per ogni componente se questo avrà bisogno di accedere a dati o metodi di altre istanze di componenti.

Esempio classico se un *Protocol* avrà bisogno di modificare delle sue variabili interne o attivare dei suoi metodi con periodicità (`protocol.NOME.step`) diversa dalla propria sarà necessario creare un *Control* con la periodicità (`control.NOME.step`) che serve e collegarlo ad ogni istanza del *Protocol*. Va ricordato che per ogni *Control* verrà creata una sola istanza (che vede la rete nella sua totalità e ha quindi accesso a tutti i nodi) mentre per ogni *Protocol* sarà creata una sua istanza per ogni nodo.

Per collegare un componente si utilizza questa convenzione:

```

7 public class Observer implements Control {
8     /**
9      * Parameters
10     */
11
12     /**
13      * The protocol to look at.
14      *
15      * @config
16      */
17     private static final String PAR_PROT = "protocol";
18
19     /**
20      * Fields
21      */
22
23     /**
24      * Value obtained from config property
25      * {@link #PAR_PROT}.
26      */
27     private final int pid;
28
29     /**
30      * Constructor
31      */
32     /**
33      * Standard constructor that reads the configuration parameters. Invoked by
34      * the simulation engine.
35      *
36      * @param prefix
37      *          the configuration prefix for this class.
38      */
39     public BLEObserver(String prefix) {
40         pid = Configuration.getPid(prefix + "." + PAR_PROT);
41     }
42 }
```

| Appendice A

In questo modo si ha a disposizione la variabile pid (l'id del componente che si vuole collegare).

Quello che si vede nell'ultima riga di codice è il modo con cui vengono lette le informazioni dal file di configurazione, è da quest'ultimo che dovrà essere indicato il nome del componente da collegare.

Analogamente dal file di configurazione è possibile passare anche valori numerici o testuali, basterà chiamare il metodo adeguato (`Configuration.getString` per una stringa ad esempio).

Nei *Controls* si può quindi accedere alle istanze di tutti i nodi dei protocolli collegati.

```
// Control interface method.  
public boolean execute() {  
  
    for (int i = 0; i < Network.size(); i++){  
  
        Myprotocol myprotocol = (Myprotocol) Network.get(i).getProtocol(pid);  
        ...  
    }  
  
    return false;  
}
```

Mentre nei *Protocols* si può accedere alle istanze dei componenti collegati (sempre tramite la variabile pid) ma dei soli nodi di cui si ha visione, cioè di cui si conosce l'id

```
Linkable linkable = (Linkable) node.getProtocol( FastConfig.getLinkable(pid) );  
Transport transport = (Transport) node.getProtocol( FastConfig.getTransport(pid) );  
  
for (int i = 0; i < linkable.degree(); i++) {  
  
    Node peern = linkable.getNeighbor(i);  
  
    Myprotocol myprotocol = (Myprotocol) peern.getProtocol(pid);
```

Il pid di un *Protocol* sarà sempre lo stesso per tutti i nodi, questo deriva dal fatto che tutte le istanze di uno stesso *Protocol* sono collegate fra loro di default, come detto in precedenza.

Per poter scambiare messaggi tra istanze di nodi diversi di uno stesso protocollo si

```
Linkable linkable = (Linkable) node.getProtocol( FastConfig.getLinkable(pid) );  
Transport transport = (Transport) node.getProtocol( FastConfig.getTransport(pid) );  
  
for (int i = 0; i < linkable.degree(); i++) {  
  
    Node peern = linkable.getNeighbor(i);  
  
    transport.send(node, peern, msg, pid);
```

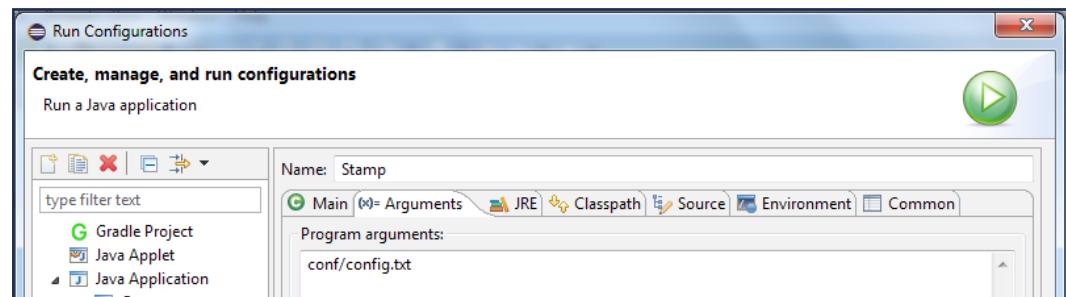
può usare il livello di trasporto oppure direttamente `EDSimulator.add`

```
EDSimulator.add(  
    delay,  
    msg,  
    peern,  
    pid);
```

mentre per accedere ai nodi di cui si ha visione va utilizzato il livello linkabile, come mostrato.

Una volta presa confidenza con la gestione di queste due caratteristiche fondamentali si hanno tutti gli strumenti per progettare una propria simulazione e successivamente implementare un proprio protocollo, su cui si vuole sperimentare, e tutte le classi degli altri componenti necessari.

Importante infine ricordare che anche nel caso in cui si voglia avviare la simulazione da un Editor è necessario passare il file di configurazione come parametro:



Durante il run del programma, per monitorare l'andamento di determinate variabili (quelle utili da osservare) alla periodicità necessaria, vanno usati appositi *Control*, chiamati *Observer*.

Si consiglia tuttavia di usarli anche per salvare valori di interesse su file .dat, così da tenerne traccia.

In questo modo li si avrà a disposizione anche per successive analisi. Saranno anche utilizzabili senza problemi su Gnuplot, oppure facilmente convertibili in formato Excel.

Appendice B

```

package controls;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;

import managers.BLE;
import peersim.config.*;
import peersim.core.*;

public class BLEReport implements Control {
    // -----
    // Parameters
    // -----

    /**
     * The alpha parameter. It affects the distance relevance in the wiring
     * process.
     *
     * @config
     */
    private static final String PAR_DENSITY = "density";

    private static final String PAR_RADIUS = "radius";

    /**
     * The protocol to look at.
     *
     * @config
     */
    private static final String PAR_PROT = "protocol";

    // -----
    // Fields
    // -----

    /**
     * Value obtained from config property
     * {@link #PAR_DENSITY}.
     */
    private final double density;

    /**
     * Value obtained from config property
     * {@link #PAR_RADIUS}.
     */
    private final double radius;

    /**
     * Value obtained from config property
     * {@link #PAR_PROT}.
     */
    private final int pid;

    // -----
    // Constructor
    // -----

    /**
     * Standard constructor that reads the configuration parameters. Invoked by
     * the simulation engine.
     *
     * @param prefix
     *          the configuration prefix for this class.
     */
    public BLEReport(String prefix) {
        pid = Configuration.getPid(prefix + "." + PAR_PROT);
    }
}

```



| Appendix B

```
density = Configuration.getDouble(prefix + “.” + PAR_DENSITY);
radius = Configuration.getDouble(prefix + “.” + PAR_RADIUS);
}

// Control interface method.
public boolean execute() {

    long max = -1;
    int nomsg = 0;
    for (int j = 0; j < Network.size(); j++){

        BLE blem = (BLE) Network.get(j).getProtocol(pid);

        if ( blem.getTimemsg() == -1 ) {
            nomsg++;
        } else if ( max < blem.getTimemsg() ) {
            max = blem.getTimemsg();
        }
    }

    //
    String den = Double.toString(density);
    if (density == 0.0005) den = “0.0005”;
    else if (density == 0.0001) den = “0.0001”;
    //

    try {
        String fname = “n” + Network.size() + “r” + (int)radius + “d” +
den.replace(“.” , “,”) + “.dat”;
        File file = new File(fname);
        if (!file.exists()) {
            file.createNewFile();
        }

        FileOutputStream fos = new FileOutputStream(file.
getAbsolutePath(), true);
        System.out.println(“Writing to file “ + fname);
        PrintStream pstr = new PrintStream(fos);

        pstr.println( ((double)max/1000) + “\t” + nomsg);

        fos.close();

    } catch (IOException e) {
        throw new RuntimeException(e);
    }

    return false;
}
}
```

Appendice C

```

package controls;

import java.io.FileOutputStream;
import java.io.IOException;
import java.io.PrintStream;

import managers.BLE;
import peersim.config.Configuration;
import peersim.core.Node;
import peersim.graph.Graph;
import peersim.reports.GraphObserver;
import rgg.RGGCoordinates;

public class BLEStamp extends GraphObserver {
    // -----
    // Parameters
    // -----

    /**
     * The filename base to print out the topology relations.
     *
     * @config
     */
    private static final String PAR_FILENAME_BASE = "file_base";

    /**
     * The coordinate protocol to look at.
     *
     * @config
     */
    private static final String PAR_COORDINATES_PROT = "coord_protocol";

    /**
     * The protocol to look at.
     *
     * @config
     */
    private static final String PAR_PROT = "pid";

    // -----
    // Fields
    // -----

    /**
     * Topology filename. Obtained from config property
     * {@link #PAR_FILENAME_BASE}.
     */
    private final String graph_filename;

    /**
     * The number of filenames already returned.
     */
    private long counter = 0;

    /**
     * Coordinate protocol identifier. Obtained from config property
     * {@link #PAR_COORDINATES_PROT}.
     */
    private final int coordPid;

    /**
     * Value obtained from config property
     * {@link #PAR_PROT}.
     */
    private final int pid;

    // -----

```

| Appendix C

```
// Constructor
// -----
/** 
 * Standard constructor that reads the configuration parameters. Invoked by
 * the simulation engine.
 *
 * @param prefix
 *          the configuration prefix for this class.
 */
public BLEStamp(String prefix) {
    super(prefix);
    coordPid = Configuration.getPid(prefix + "." + PAR_COORDINATES_PROT);
    pid = Configuration.getPid(prefix + "." + PAR_PROT);
    graph_filename = Configuration.getString(prefix + "."
        + PAR_FILENAME_BASE, "bleStates");
}

// Control interface method.
public boolean execute() {
    try {
        updateGraph();

        System.out.print(name + ": ");

        // initialize output streams
        String fname1 = graph_filename + counter + ".dat";
        String fname2 = "receivers" + counter + ".dat";
        counter++;

        FileOutputStream fos = new FileOutputStream(fname1);
        System.out.println("Writing to file " + fname1);
        PrintStream pstr = new PrintStream(fos);

        // dump topology:
        statesToFile(g, pstr, coordPid, pid);

        fos.close();

        FileOutputStream flos = new FileOutputStream(fname2);
        System.out.println("Writing to file " + fname2);
        PrintStream ps = new PrintStream(flos);

        // dump topology:
        graphToFile(g, ps, coordPid, pid);

        flos.close();
    } catch (IOException e) {
        throw new RuntimeException(e);
    }

    return false;
}

private static void statesToFile(Graph g, PrintStream ps, int coordPid, int pid) {
    for (int i = 0; i < g.size(); i++) {

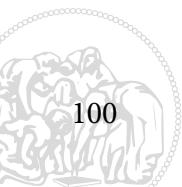
        Node current = (Node) g.getNode(i);

        double x_to = ((RGGCoordinates) current
            .getProtocol(coordPid)).getX();
        double y_to = ((RGGCoordinates) current
            .getProtocol(coordPid)).getY();

        int bleState = ((BLE) current
            .getProtocol(pid)).getbleState();

        ps.println(x_to + " " + y_to + " " + bleState);
        ps.println();
    }
}

private static void graphToFile(Graph g, PrintStream ps, int coordPid, int pid) {
    for (int i = 0; i < g.size(); i++) {
```



```
Node current = (Node) g.getNode(i);

double x_to = ((RGGCoordinates) current
    .getProtocol(coordPid)).getX();
double y_to = ((RGGCoordinates) current
    .getProtocol(coordPid)).getY();

Node receiver = ((BLE) current
    .getProtocol(pid)).getReceiver();

if (receiver != null){

    double x_from = ((RGGCoordinates) receiver
        .getProtocol(coordPid)).getX();
    double y_from = ((RGGCoordinates) receiver
        .getProtocol(coordPid)).getY();

    ps.println(x_from + " " + y_from);
    ps.println(x_to + " " + y_to);
    ps.println();
}
}

}

}
```

