Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Disseminazioni in reti peer-to-peer attraverso algoritmi di gossip a due fasi

Tesi di Laurea in Simulazione di Sistemi

Relatore:
Gabriele D'Angelo

Presentata da:
Luca Valentini

Sessione III
Anno Accademico 2014-2015

Introduzione

Il continuo aumento di dispositivi connessi alla rete, dovuto in gran parte alle evoluzioni e ai progressi fatti dalla tecnologia utilizzata dai device mobile, ha portato a valutare alternative all'approccio client-server di molte tipologie di applicazioni. Da svariati anni, infatti, il paradigma peer-to-peer si è diffuso in una moltitudine di campi.

Si trovano infatti sue applicazioni, nella condivisione di file del precursore Napster e del più moderno BitTorrent, nella messaggistica istantanea e VoIP di Skype, nei sistemi ibridi come quello che Spotify ha utilizzato fino a poco tempo fa per alleggerire il carico sui server, fino ad arrivare a sistemi di streaming video peer-to-peer.

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 due nomi, il loro comportamento ricorda la diffusione delle malattie o di un pettegolezzo (gossip) all'interno di un gruppo sociale.

In ambito informatico ci si riferisce invece alla disseminazione di messaggi, secondo determinati criteri, all'interno di una rete.

In questa tesi si andrà a proporre ed analizzare una nuova tipologia di algoritmi, definiti algoritmi di gossip a due fasi, con lo scopo di aumentare la percentuale di nodi raggiunti dai messaggi generati, cercando di minimizzare

la quantità di messaggi duplicati, considerati inutili in questo contesto.

La tesi si divide in sei capitoli:

- nel primo verrà fatta un'introduzione agli algoritmi di gossip, ai modelli teorici sui quali si basano ed agli utilizzi proposti in letteratura e in applicazioni già esistenti
- nel secondo capitolo verranno introdotti i grafi su cui gli algoritmi sono stati testati e analizzati
- nel terzo capitolo verrà illustrato l'ambiente di sviluppo e simulazione utilizzato
- nel quarto capitolo verranno elencate caratteristiche e funzionamento di tutti gli algoritmi utilizzati, sia quelli classici che quelli a due fasi proposti
- nel quinto capitolo vedremo l'analisi dei risultati ottenuti
- nel sesto e ultimo capitolo si trarranno alcune conclusioni sui risultati e verranno suggeriti alcuni spunti su possibili lavori e sviluppi futuri utili ad analizzare meglio gli algoritmi proposti

Indice

Elenco delle figure

Elenco degli algoritmi

Capitolo 1

Disseminazione attraverso algoritmi di gossip

Gli algoritmi di gossip sono l'evoluzione dell'algoritmo di flooding, il metodo più semplice per diffondere un'informazione in una rete peer-to-peer, nel quale ogni messaggio ricevuto da un nodo viene inoltrato a qualsiasi altro nodo ad esso connesso, escluso il mittente. Si tratta però di un algoritmo molto inefficiente a causa dell'enorme numero di messaggi ridondanti trasmessi nella rete, possibile causa di broadcast storm (tempesta di trasmissioni) [17]. Gli algoritmi di gossip vanno a migliorare proprio questo aspetto, attraverso criteri di inoltro dei messaggi generalmente probabilistici. Esistono anche algoritmi di gossip totalmente deterministici, come Fixed Fanout [21], che non verrà però analizzato in questa tesi.

Spesso ci si riferisce agli algoritmi di gossip con la definizione di algoritmi epidemici, termine che fa riferimento alla somiglianza tra questi algoritmi e i modelli di diffusione delle malattie. Dei modelli epidemici si può utilizzare anche la terminologia utile a comprendere i comportamenti e gli stati di un nodo [20]. Un nodo infatti può assumere tre diversi stati:

- Susceptible (S): In questo stato il nodo è "suscettibile" all'infezione, ovvero non ha ancora ricevuto il messaggio.

- Infected (I): In questo stato il nodo è "infetto", ovvero ha ricevuto il messaggio e lo sta inoltrando.

- Removed (R): In questo stato il nodo ha ricevuto il messaggio, ma non lo sta inoltrando. In un contesto epidemiologico rappresenta la morte o l'immunità alla malattia, nel contesto gossip riguarda invece lo stato nel quale la disseminazione da parte del nodo è già avvenuta e quindi, magari per l'utilizzo di uno storico o cache dei messaggi già inviati, il nodo non inoltrerà più il messaggio.

Bisogna precisare che nei modelli analizzati lo stato del nodo è relativo al singolo messaggio e non all'intero traffico del sistema. Infatti in una stessa fase della simulazione un nodo potrebbe scartare un certo messaggio (e quindi essere in uno stato R), inoltrarne un secondo (stato I) e non averne ancora ricevuto un terzo (stato S).

In base al comportamento e agli stati che possono assumere i nodi sono stati definiti alcuni modelli [23]. I due che vengono citati sotto possono essere applicati ad algoritmi di tipo push, pull e ibridi (push-pull), tre categorie in cui possiamo suddividere gli algoritmi di gossip, in base al comportamento dei nodi.

1.1 Modello SIS (Susceptible, Infected, Susceptible)

Nel modello SIS [23] sono possibili solo gli stati Susceptible e Infected. In pratica un nodo, una volta ricevuto un messaggio e averlo disseminato, ritorna allo stato Susceptible e di conseguenza inoltrerà nuovamente lo stesso messaggio nel caso lo ricevesse una seconda volta.

1.2 Modello SIR (Susceptible, Infected, Removed)

Il modello SIR [23] aggiunge lo stato Removed, in modo da dare una terminazione alla vita del messaggio all'interno della rete. In pratica un nodo in uno stato Infected può passare allo stato Removed in caso di ricezione di un messaggio già noto. Ovviamente per implementare questo modello i nodi necessitano di avere memoria dei messaggi già ricevuti.

1.3 Algoritmi Push

Negli algoritmi di gossip push sono i nodi mittenti che decidono se e a chi trasmettere il messaggio. Ad ogni generazione e ad ogni ricezione, in base ai criteri stabiliti dall'algoritmo, il nodo valuterà se inviare o meno il messaggio a un sottogruppo dei suoi vicini (Fixed Probability), a tutti o a nessuno (Conditional Broadcast e in alcuni casi Fixed Probability). Sono però necessari meccanismi per evitare che il messaggio continui a diffondersi all'infinito all'interno della rete. I due meccanismi utilizzati a tale scopo sono TTL (Time-to-live) e cache. Il TTL è un valore legato al messaggio che viene decrementato ad ogni ricezione dello stesso, in modo da poter eliminare il messaggio dal sistema dopo un certo periodo di tempo [26]. La cache serve invece a mantenere uno storico da parte del nodo dei messaggi già ricevuti, in modo da evitare ritrasmissioni ridondanti. Negli algoritmi push la probabilità di inviare il messaggio a nodi che non l'hanno ancora ricevuto cala ad ogni step della disseminazione. Infatti, nella prima fase, solo il nodo che ha generato il messaggio ne è a conoscenza, ma ad ogni ritrasmissione diminuisce il numero di messaggi "utili" rispetto a quelli doppi. Per lo stesso motivo l'overhead (sostanzialmente una misura del numero di messaggi duplicati ricevuti) non cresce linearmente con la percentuale di copertura raggiunta, ma in maniera esponenziale. Gli algoritmi analizzati in questa tesi sono tutti di tipo push.

1.4 Algoritmi Pull

Negli algoritmi pull non sono più i nodi mittenti a valutare la trasmissione del messaggio, ma sono i riceventi che periodicamente interrogano i nodi a cui sono connessi (o un loro sottoinsieme) chiedendo l'eventuale trasmissione di nuove informazioni. Al contrario degli algoritmi push, in questo caso le prime fasi della disseminazione saranno poco produttive, mentre nelle fasi successive il messaggio inizierà a diffondersi maggiormente all'interno della rete. Per questo motivo in termini di delay sono meno efficienti degli algoritmi push, anche se tendono a generare un overhead minore [26].

1.5 Algoritmi Push-pull (ibridi)

L'idea alla base degli algoritmi ibridi è quella di sfruttare i pro degli approcci push e pull minimizzandone i lati negativi. Come detto, da una parte gli algoritmi push sono molto efficienti nelle prime fasi della disseminazione, mentre gli algoritmi pull lo sono in quelle più avanzate [26]. Proprio per questo gli esempi di algoritmi ibridi sono in genere "First Push then Pull" [18, 19]. Sono algoritmi però più complessi da implementare e analizzare, sia per la gestione dello storico dei messaggi già ricevuti, sia per la scelta del momento più opportuno in cui passare dall'approccio push a quello pull.

1.6 Utilizzi del gossip

Il gossip si è dimostrato un metodo molto efficace per distribuire informazioni all'interno di una rete peer-to-peer. I campi di applicazione sono molteplici ed è possibile individuare vari esempi in letteratura, sia di proposte teoriche, sia di applicazioni vere e proprie. In particolare, troviamo proposte di utilizzo del gossip in reti di sensori [3], reti mobile ad-hoc [4], nel gaming online multigiocatore [5], in sistemi publish/subscribe [6, 7, 8], nella gestione delle risorse in sistemi cloud [9], etc. Tra gli applicativi veri e propri che implementano algoritmi di gossip troviamo invece Amazon S3 per

lo scambio di informazioni sullo stato dei server all'interno del sistema [10], Dynamo, sistema di storage di Amazon che utilizza il gossip per il rilevamento degli errori [11], Cassandra, sistema di storage di Facebook [12] e Tribler, un client BitTorrent anonimo ispirato al sistema TOR.

Capitolo 2

Tipologie di grafi analizzate

In informatica con grafo intendiamo una struttura composta da nodi e archi [25]. In una rete peer-to-peer possiamo immaginare i nodi come i diversi utenti presenti nella rete e come archi la connessione diretta tra loro, attraverso la quale si scambiano messaggi.

Le strutture utilizzate per l'analisi degli algoritmi sono infatti reti peer-to-peer in cui i collegamenti tra i nodi sono non direzionati, ovvero attraverso lo stesso arco un nodo può sia inviare che ricevere un messaggio.

Nell'ambiente simulato utilizzato ogni messaggio impiega sempre lo stesso tempo per essere trasmesso, indipendentemente da mittente e destinatario, per questo i grafi possono essere definiti "non pesati".

Per verificare se il comportamento dell'algoritmo varia in maniera notevole in base alle caratteristiche dei grafi scelti, illustrate in seguito, sono stati utilizzati vari gruppi di grafi per le diverse prove.

I grafi scelti sono di quattro tipologie diverse, tutti composti da 500 nodi, con 1000 e 2000 archi (997 e 1990 per quelli Scale-free). I risultati di ogni esperimento sono calcolati sulla media 10 grafi diversi, della stessa tipologia e con lo stesso numero di archi.

Le figure 2.1, 2.2, 2.3 e 2.4, generate con il software R [25] e la relativa libreria igraph [2] rappresentano esempi di grafi delle quattro tipologie, ma con un numero di nodi e archi inferiori per poterne distinguere meglio le

caratteristiche.

2.1 Grafi Random (Erdös-Rényi)

Nei Grafi Random il collegamento tra i nodi avviene in modalità del tutto casuale, utilizzando una probabilità fissa uguale per ogni arco e indipendente da qualsiasi altro parametro. Il modello utilizzato è Erdös-Rényi (ER) [13] ed ogni arco viene incluso nel grafo e attaccato a due nodi in base ad una probabilità p.

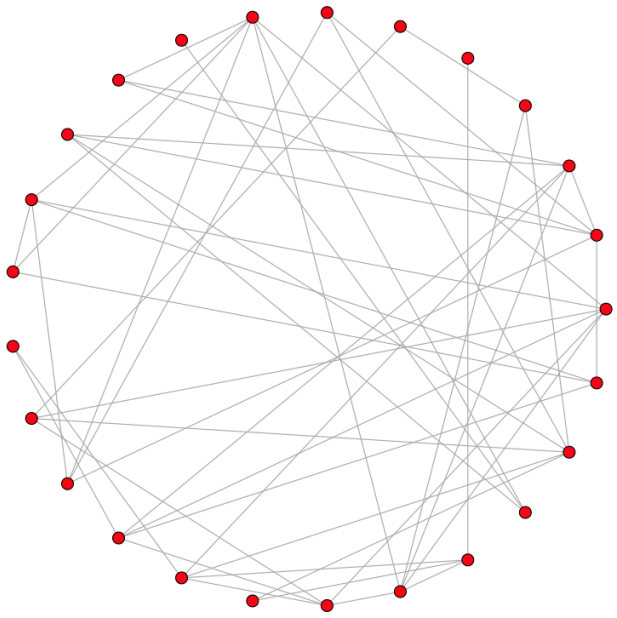


Figura 2.1: Esempio di grafo Random con 25 nodi e 50 archi

2.2 Grafi Scale-Free (Barabàsi-Albert)

I grafi di tipo Scale-Free sono caratterizzati dalla grande disparità presente nel numero di connessioni tra alcuni nodi ed altri. La distribuzione del numero di vicini dei nodi può essere approssimata da una legge di potenza.

Quindi troveremo alcuni nodi molto connessi (hub) e molti con poche connessioni. La presenza di questi hub rende però il diametro del grafo molto piccolo e di conseguenza le disseminazioni risultano più rapide. Per la costruzione dei grafi si utilizza il modello di Barabàsi-Albert (BA) [14]. Si crea un piccolo gruppo iniziale di m nodi con le loro connessioni. Successivamente viene aggiunto un nuovo nodo per volta e viene connesso ad n (con $n \leq m$) nodi presenti nel grafo. La probabilità di scelta del nodo a cui connettersi è direttamente proporzionale al numero di connessioni già presenti in quel nodo. In questo modo si crea un meccanismo per il quale più connessioni ha un nodo, più nodi si connetteranno a lui, creando i già citati hub. D'altra parte nodi con poche connessioni tenderanno a non riceverne molte altre o a rimanere con i soli archi inseriti al momento della loro creazione. Nonostante il processo di costruzione sia molto diverso, i grafi Scale-Free sono in realtà degli "ultra Small-world" con hub ancora più grandi e diametro ancora più piccolo.

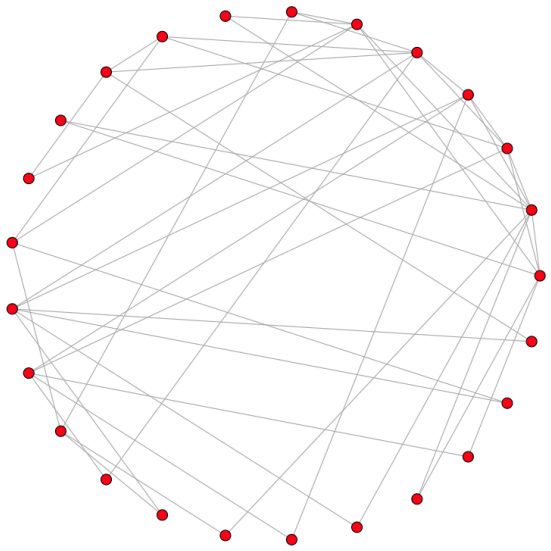


Figura 2.2: Esempio di grafo Scale-Free con 25 nodi e 47 archi

2.3 Grafi Small-world

I grafi di tipo Small-world sono caratterizzati dal fatto che quasi tutte le coppie di nodi presenti all'interno del grafo sono collegate da un percorso breve. Questo è possibile grazie alla presenza di numerosi hub, ovvero nodi con molte connessioni, dai quali passerà la maggior parte dei percorsi brevi tra due nodi. La costruzione segue il modello di Watts-Strogatz (WS) [15]. Viene costruito un reticolo di d dimensioni in cui ogni nodo viene connesso ai due più prossimi su ogni dimensione. Quindi per d = 1 ogni nodo avrà 2 connessioni, per d = 2 ne avrà 4 e così via. A quel punto, in base ad un parametro p, ogni arco potrà venire riconnesso in maniera casuale nel grafo.

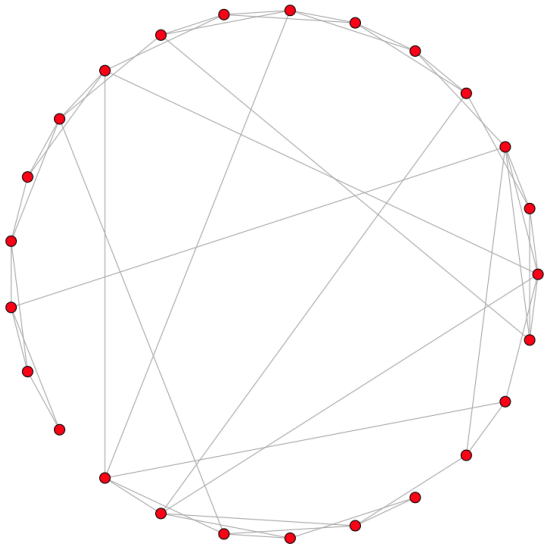


Figura 2.3: Esempio di grafo Small-world con 25 nodi e 50 archi

2.4 Grafi K-Regular

I grafi di tipo K-Regular hanno la caratteristica di avere tutti i nodi con lo stesso numero di archi [24]. Sono tipo di grafo riscontrabili in vari sistemi

Peer-to-Peer dove il numero di connessioni viene deciso (e limitato) a livello applicativo.

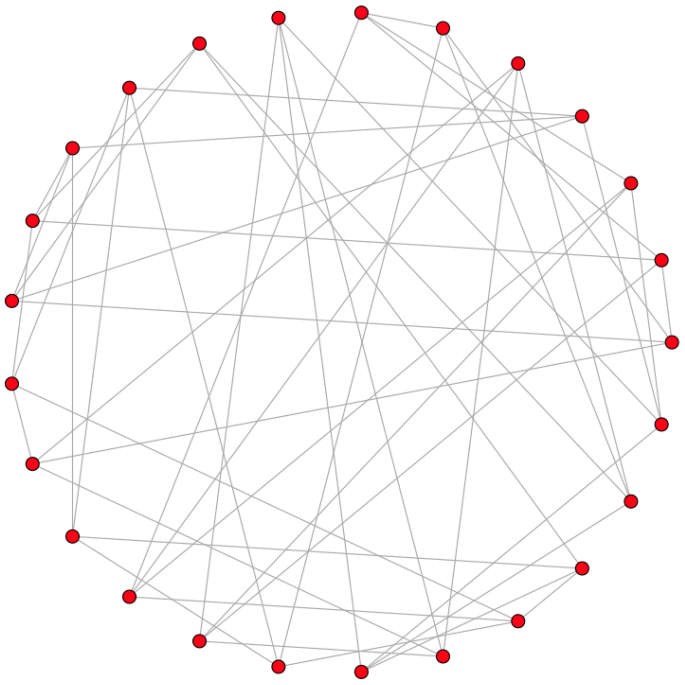


Figura 2.4: Esempio di grafo K-Regular con 25 nodi e 50 archi

Capitolo 3

Ambiente di simulazione e software utilizzati

Data la natura probabilistica degli algoritmi trattati e la complessità dei grafi su cui vengono applicati, non è possibile fare una stima a priori delle performance e dei risultati ottenibili. Il metodo migliore per valutare il comportamento di questi algoritmi è attraverso la simulazione in vari scenari differenti. Oltre all'utilizzo di diverse tipologie di grafi, ogni risultato è basato sulla media degli esperimenti effettuati su dieci grafi dello stesso tipo e con le stesse caratteristiche. Il motivo di questa scelta, è che sia la generazione dei grafi, in particolare i collegamenti tra i diversi nodi, sia gli algoritmi utilizzati sono probabilistici. Di conseguenza, l'utilizzo della media di dieci esperimenti andrà a diminuire la varianza dei risultati. Per l'analisi degli algoritmi sviluppati è stato utilizzato l'ambiente di simulazione sviluppato dal PADS, gruppo di ricerca su simulazione distribuita e parallela [1].

3.1 LUNES

LUNES (Large Unstructured NEtwork Simulator) è il tool utilizzato per l'implementazione e la simulazione degli algoritmi. È strutturato in maniera

modulare e si divide in varie componenti per svolgere le diverse fasi della simulazione:

- creazione dei grafi
- implementazione dei protocolli di simulazione
- analisi dei risultati

3.1.1 Creazione dei grafi

LUNES attualmente utilizza igraph [2], un tool per la creazione e manipolazione di grafi, per la creazione degli stessi. I file creati da igraph possono essere utilizzati direttamente da LUNES, oppure raggruppati in collezioni di più grafi con le stesse caratteristiche per poter testare gli algoritmi su più configurazioni alla volta.

3.1.2 Implementazione dei protocolli di simulazione

Il modulo di LUNES che si occupa della simulazione vera e propria, si appoggia al middleware ARTÌS e al framework GAIA. In questo modo lo sviluppo di nuovi algoritmi non necessita di programmazione di basso livello, ma è sufficiente l'utilizzo delle funzioni già presenti (invio e ricezione di messaggi, gestione della cache, etc.) per implementare facilmente nuovi protocolli.

3.1.3 Analisi dei risultati

La fase di analisi dei risultati è la parte più impegnativa, a livello di risorse utilizzate, di tutto il processo di simulazione. Una disseminazione su un grafo composto da qualche centinaio di nodi, per tenere traccia di tutti i messaggi inviati, arriva ad occupare svariati gigabyte di file temporanei su disco. Per questo motivo il modulo di LUNES incaricato del parsing delle tracce delle disseminazioni, è stato sviluppato prevalentemente in C, in appoggio ad alcuni script bash.

3.2 L'ambiente di simulazione

Per la simulazione e l'analisi dei risultati, ogni disseminazione è stata eseguita su gruppi di dieci grafi diversi, ma con la stessa configurazione. In altre parole, per ogni algoritmo vengono eseguite 30 run (con probabilità di inoltro compresa tra 1 e 100), ognuna delle quali su 10 grafi. Il risultato della disseminazione è la media calcolata sui 10 diversi grafi. Ogni run di simulazione dura 1000 unità di tempo e, a parte le prime 5 utilizzate per le connessioni tra nodi, dall'inizio alla fine ogni nodo può generare messaggi. Tra la generazione di un messaggio e il successivo, il tempo di attesa è probabilistico e segue una distribuzione esponenziale di media 10. Cache e TTL, parametri di cui si parlerà più approfonditamente nel capitolo successivo, sono stati impostati a 512 e 32.

Capitolo 4

Algoritmi di disseminazione

Per comprendere gli algoritmi di disseminazione e la loro implementazione in questo contesto simulativo, è bene introdurre alcuni dei parametri utilizzati sia dagli algoritmi classici, sia quelli aggiunti per gli algoritmi a due fasi.

4.1 Cache

La cache viene utilizzata per tenere traccia dei messaggi già ricevuti da parte dei nodi. Questo serve, sia negli algoritmi classici che in quelli a due fasi, ad evitare ritrasmissioni dello stesso messaggio nel caso venisse ricevuto più di una volta. Quindi, grazie alla cache, alla seconda ricezione (e quelle successive) il pacchetto viene scartato senza provare a ritrasmetterlo, diminuendo così il numero di messaggi ridondanti e di conseguenza l'overhead. La cache utilizzata è di tipo Least Recently Used (LRU), ovvero "meno recentemente usato". Con questo approccio, una volta riempita la cache, in caso di ricezione di un nuovo messaggio, questo sarà memorizzato al posto di quello più vecchio presente nella cache. Per questo la sua dimensione è un parametro cruciale per le performance degli algoritmi. Infatti, in caso di dimensioni troppo ridotte, dei messaggi già trasmessi potrebbero risultare "nuovi" al nodo perché sovrascritti da altri ricevuti più recentemente. Una cache molto grande può invece ridurre (fino ad eliminare) le ritrasmissioni di

pacchetti ridondanti, ma con un costo in memoria occupata. Negli algoritmi a due fasi la cache acquista ancora più importanza, dato che il parametro principale per decidere se ritrasmettere un messaggio durante la seconda fase è verificare se è stato ricevuto più di una volta.

4.2 Time To Live (TTL)

Il Time To Live è un parametro proprio di ogni messaggio e indica il numero massimo di ritrasmissioni che può avere un messaggio prima di venire scartato. Ad ogni ritrasmissione da parte di un nodo infatti il TTL viene decrementato fino a raggiungere lo 0 e venire scartato. È un parametro fondamentale perché un TTL troppo piccolo impedisce al messaggio di raggiungere nodi troppo "lontani" in un grafo dal diametro elevato. Un TTL troppo elevato invece può comportare un aumento di overhead, occupazione e saturazione della cache e messaggi che continuano a essere ritrasmessi per lunghi periodi all'interno del grafo.

4.3 Waiting Steps (WS)

È un parametro introdotto per gli algoritmi a due fasi e rappresenta il tempo (simulato) di attesa tra la prima e la seconda disseminazione di un messaggio. Dato che la seconda disseminazione inizia solamente nel caso in cui un messaggio sia stato ricevuto una sola volta, questo parametro è sostanzialmente il "timeout" entro il quale se un messaggio non viene ricevuto nuovamente viene ritrasmesso. È un parametro importante perché influenza in maniera netta la copertura raggiunta, il delay e l'impatto sulla cache.

4.4 Keep TTL (KT)

Nell'ambiente di simulazione utilizzato il TTL può avere una doppia interpretazione, ovvero il numero di hop fatti dal messaggio e il tempo (simulato)

trascorso dall'invio. Questo accade perché il tempo di invio di un messaggio è sempre e comunque pari a uno step di simulazione. Negli algoritmi a due fasi è stato quindi introdotto un parametro, chiamato "Keep TTL" per decrementare o meno il TTL dei messaggi durante gli step di simulazione di attesa tra la prima e la seconda fase. Decrementandolo lo consideriamo quindi un parametro "temporale", mentre mantenendo il TTL durante l'attesa della ritrasmissione il suo limite sarà invece spaziale, inteso come numero di hop.

4.5 Algoritmi classici

La generazione dei messaggi, comune ai due algoritmi classici, è permessa a qualsiasi nodo. Il tempo di attesa tra la generazione di un messaggio e il successivo, segue una distribuzione esponenziale di media 10. Una volta generato il messaggio viene inoltrato a tutti i vicini del nodo e inserito nella cache.

Algorithm 1 Generazione dei messaggi [22]

```
1: function GENERATE()
2:     t ← GenerationThreshold()
3:     if Random() < t then
4:         msg ← CreateMessage()
5:         Cache(msg)
6:         Broadcast(msg)
7:     end if
8: end function
```

Alla ricezione del messaggio, se non presente in cache e se il TTL è ancora positivo, verrà possibilmente inoltrato ai vicini del nodo ricevente, in base all'algoritmo utilizzato. Se presente in cache, o con TTL nullo, verrà invece scartato.

Algorithm 2 Ricezione di un messaggio [22]

```
1: function RECEIVE(msg)
2:     if (NotCached(msg) ∧ msg.ttl > 0) then
3:         Cache(msg)
4:         msg.ttl ← msg.ttl − 1
5:         Gossip(msg)
6:     end if
7: end function
```

All'inizializzazione ogni algoritmo imposterà la propria probabilità di invio, uguale per tutti i nodi e costante per tutto l'algoritmo.

Algorithm 3 Inizializzazione [22]

```
1: function INITIALIZATION()
2:     β ← GetProbability()
3: end function
```

Gli algoritmi "classici" analizzati e utilizzati come termine di paragone sono Conditional Broadcast e Fixed Probability.

4.5.1 Conditional Broadcast (CB)

Conditional Broadcast è un algoritmo di gossip probabilistico, dove l'inoltro di un messaggio dipende da una chiamata random con distribuzione uniforme tra 0 e 1, confrontata con una probabilità β definita. Con probabilità 1 è equivalente ad un algoritmo di flooding. Dopo la ricezione di un messaggio, quindi, l'algoritmo deciderà, in base al confronto tra il valore pseudocasuale generato e β, se inoltrare il messaggio a tutti i suoi vicini o a nessuno.

4.5.2 Fixed Probability (FP)

Fixed Probability è il secondo algoritmo classico preso in esame. A differenza di CB, dove il messaggio può venire inoltrato o a tutti o a nessuno,

Algorithm 4 Disseminazione con Conditional Broadcast [22]

```
function GOSSIP(msg)
    if Random() < β then
        for all n_j ∈ Π do
            Send(msg, n_j)
        end for
    end if
end function
```

in questo caso il confronto tra chiamata Random e probabilità definita viene fatto per ogni singolo vicino del nodo che ha ricevuto il messaggio. È generalmente un approccio migliore dal punto di vista della quantità di nodi che si raggiungono durante la disseminazione in rapporto all'overhead, ma il tempo necessario a farlo è leggermente maggiore rispetto al CB. Anche in questo caso con β pari a 1 diventa un flooding.

Algorithm 5 Disseminazione con Fixed Probability [22]

```
function GOSSIP(msg)
    for all n_j ∈ Π do
        if Random() < β then
            Send(msg, n_j)
        end if
    end for
end function
```

4.6 Algoritmi proposti

Gli algoritmi proposti sono stati definiti "Two Phases", perché la loro caratteristica principale è quella di provare ad inoltrare il messaggio in due fasi successive, a distanza di tempo, a meno che questo non venga ricevuto più di una volta. L'idea alla base è infatti che la ricezione multipla dello

stesso messaggio può essere un'indicazione del fatto che quel messaggio si sta diffondendo all'interno del grafo e inoltrarlo nuovamente porterebbe a maggiore overhead, senza portare grossi benefici a livello di diffusione.

La prima fase degli algoritmi è comune a quella di uno dei due classici, con l'aggiunta di alcuni meccanismi di controllo per la gestione della cache dei messaggi da inoltrare nella seconda fase.

La prima differenza la vediamo infatti nella funzione di ricezione. In questa fase, ogni messaggio inserito in cache, tranne quelli generati dal nodo stesso, viene inserito anche nel gruppo dei messaggi da inviare durante la seconda fase. Se il messaggio è invece già presente in cache, quindi ricevuto più di una volta, verrà rimosso (se presente) dalla cache per la seconda fase.

Algorithm 6 Ricezione negli algoritmi a due fasi

```
1: function RECEIVE(msg)
2:     if (NotCached(msg) ∧ msg.ttl > 0) then
3:         Cache(msg)
4:         msg.ttl ← msg.ttl − 1
5:         Gossip(msg)
6:     else
7:         RemoveFromSecondPhaseCache(msg)
8:     end if
9: end function
```

La seconda fase di disseminazione è a sua volta basata su uno dei due algoritmi classici, ma i nodi a cui è possibile inoltrare un messaggio non sarà più tutto l'insieme dei vicini, ma solo il sottoinsieme al quale il messaggio non è stato inviato nel primo caso. Ovviamente nel caso in cui la prima disseminazione sia stata fatta attraverso un Conditional Broadcast l'insieme dei nodi della seconda fase sarà o "tutti i vicini" o "nessuno".

Ad ogni step di simulazione, ogni nodo eseguirà la funzione "SecondPhase" che controllerà la cache dei messaggi da inoltrare e per ogni messaggio presente valuterà se inviarlo o meno in base a due parametri, ovvero tempo

di attesa (Waiting Steps) e TTL rimanente. Una volta applicato uno dei due algoritmi classici, il messaggio verrà rimosso dalla SecondPhaseCache.

4.6.1 Two Phases CBCB

TwoPhases CBCB è il primo algoritmo a due fasi presentato. Come suggerisce il nome, utilizza Conditional Broadcast sia nella prima che nella seconda fase di disseminazione. Data la natura di "tutti o nessuno" del Conditional Broadcast il messaggio verrà inoltrato dal nodo al massimo una volta, o durante la prima, o durante la seconda fase di disseminazione o in nessuno dei due casi. Alla ricezione infatti, il messaggio può essere inoltrato a tutti i vicini in base al confronto tra il valore pseudocasuale generato e la probabilità β fissata. Nel caso non venisse inoltrato, verrà invece salvato nella SecondPhaseCache. Nel caso non venisse inoltrato neanche durante la seconda fase, verrà comunque scartato.

Algorithm 7 Prima fase di disseminazione con Two Phases CBCB

```
1: function GOSSIP(msg)
2:     if Random() < β then
3:         for all n_j ∈ Π do
4:             Send(msg, n_j)
5:         end for
6:     else
7:         AddToSecondPhaseCache(msg)
8:     end if
9: end function
```

4.6.2 Two Phases FPFP

TwoPhases FPFP utilizza due disseminazioni basate su Fixed Probability. Rispetto agli algoritmi dove la prima fase è di tipo Conditional Broadcast, nel caso la prima disseminazione avvenga con Fixed Probability è necessario

Algorithm 8 Seconda fase di disseminazione con Two Phases CBCB

```
1: function SecondPhase
2:     for all msg_i ∈ SecondPhaseCache do
3:         if Random() < β then
4:             for all n_j ∈ Π do
5:                 Send(msg_i, n_j)
6:             end for
7:         end if
8:         RemoveFromSecondPhaseCache(msg_i)
9:     end for
10: end function
```

memorizzare, oltre al messaggio, anche l'elenco dei nodi a cui non è stato inviato. Se dopo il tempo di attesa il messaggio risulterà da inoltrare per la seconda fase, verrà inviato con Fixed Probability tra i nodi a cui non è già stato inviato. Di conseguenza, durante la seconda disseminazione, il ciclo for non avverrà tra tutti i vicini del nodo, ma solo sul sottoinsieme a cui il messaggio non è stato inviato durante la prima fase.

Algorithm 9 Prima fase di disseminazione con Two Phases FPFP

```
1: function Gossip(msg)
2:     for all n_j ∈ Π do
3:         if Random() < β then
4:             Send(msg, n_j)
5:         else
6:             AddToSecondPhaseCache(msg, n_j)
7:         end if
8:     end for
9: end function
```

Algorithm 10 Seconda fase di disseminazione con Two Phases FPFP

```
function SecondPhase
    for all msg_i] ∈ SecondPhaseCache do
        for all n_j ∈ msg_i do
            if Random() < β then
                Send(msg_i, n_j)
            end if
        end for
        RemoveFromSecondPhaseCache(msg_i)
    end for
end function
```

4.6.3 Two Phases CBFP

Il terzo algoritmo presentato si comporta come un Conditional Broadcast nella prima fase e, nel caso il messaggio non venga inviato, durante la seconda fase proverà a inoltrarlo con un Fixed Probability.

Algorithm 11 Prima fase di disseminazione con Two Phases CBFP

```
1: function Gossip(msg)
2:     if Random() < β then
3:         for all n_j ∈ Π do
4:             Send(msg, n_j)
5:         end for
6:     else
7:         AddToSecondPhaseCache(msg)
8:     end if
9: end function
```

4.6.4 Two Phases FPCB

L'ultimo algoritmo presentato si comporta come un Fixed Probability durante la prima fase, salvando i nodi ai quali non viene inviato un messaggio,

Algorithm 12 Seconda fase di disseminazione con Two Phases CBFP

```
function SecondPhase
    for all msg_i ∈ SecondPhaseCache do
        for all n_j ∈ msg_i do
            if Random() < β then
                Send(msg_i, n_j)
            end if
        end for
        RemoveFromSecondPhaseCache(msg_i)
    end for
end function
```

per poi, eventualmente, inoltrarlo durante la seconda fase come Conditional Broadcast.

Algorithm 13 Prima fase di disseminazione con Two Phases FPCB

```
1: function Gossip(msg)
2:     for all n_j ∈ Π do
3:         if Random() < β then
4:             Send(msg, n_j)
5:         else
6:             AddToSecondPhaseCache(msg, n_j)
7:         end if
8:     end for
9: end function
```

4.7 Metriche di valutazione

Per la valutazione degli algoritmi proposti sono state utilizzate tre metriche, in particolare copertura, overhead e delay, da confrontare con i risultati ottenuti con gli algoritmi classici. In genere, per l'analisi dei risultati, copertura e delay vengono confrontati in rapporto all'overhead.

Algorithm 14 Seconda fase di disseminazione con Two Phases FPCB

```
function SecondPhase
    for all msg_i ∈ SecondPhaseCache do
        if Random() < β then
            for all n_j ∈ SecondPhaseCache.msg_i do
                Send(msg_i, n_j)
            end for
        end if
        RemoveFromSecondPhaseCache(msg_i)
    end for
end function
```

4.7.1 Copertura

La copertura è la percentuale media, calcolata su ogni singolo messaggio creato, di quanti dei nodi del grafo sono stati raggiunti dalla disseminazione. L'obiettivo è ovviamente raggiungere valori elevati.

4.7.2 Overhead

Con limite minimo teorico, si intende il numero di messaggi necessari a raggiungere una copertura del 100% nel grafo preso in esame, per ogni singolo messaggio generato durante l'esperimento. Definito questo parametro, l'overhead è calcolato come rapporto tra il numero totale di messaggi inviati durante l'esperimento e il limite minimo teorico. Minore è l'overhead, in rapporto alle altre metriche, migliore è il risultato.

4.7.3 Delay

Il delay, negli altri algoritmi sviluppati e analizzati tramite LUNES, indica il numero medio di hop fatti da ogni messaggio durante la disseminazione. Però, nell'ambiente di simulazione utilizzato, l'invio di un messaggio avviene sempre in uno step di simulazione. Di conseguenza tempo simulato e numero

di hop sono equivalenti negli algoritmi classici. Negli algoritmi a due fasi, il numero di step di simulazione attesi da un nodo per ritrasmettere il messaggio vanno a influenzare il delay, che viene comunque calcolato come numero di step di simulazione intercorsi tra creazione del messaggio e consegna ad un nodo. Di conseguenza in questa tesi tratteremo il delay come una misura temporale e non come il numero di hop fatti dal messaggio. Ovviamente un delay minore è preferibile, ma in molti casi aumentare il delay è l'unico modo per migliorare il rapporto tra copertura e overhead.

Capitolo 5

Analisi dei risultati

Per una migliore leggibilità dei risultati ottenuti, l'analisi dei risultati è stata divisa in più confronti:

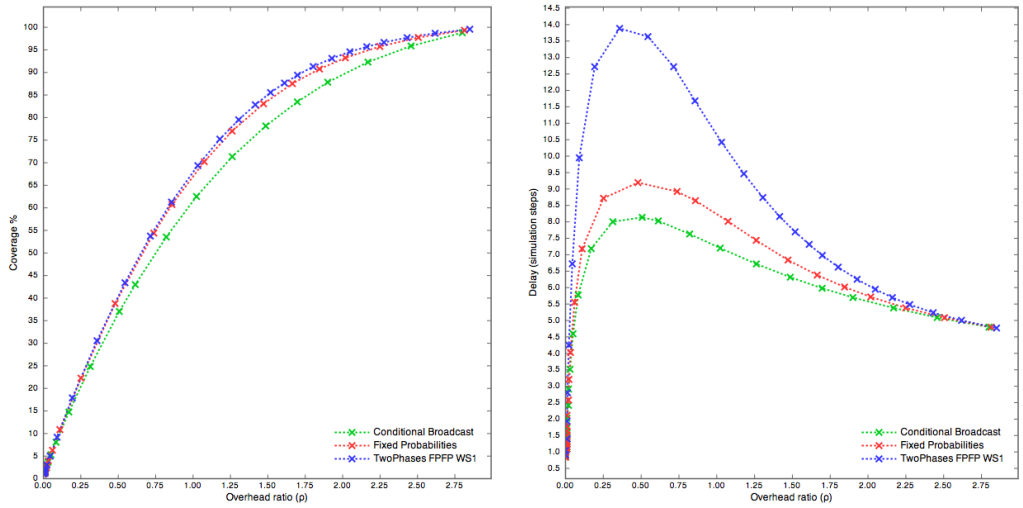
- nella prima sezione viene confrontato l'algoritmo TwoPhases FPFP con il parametro waiting steps pari a 1 con i due algoritmi classici, ovvero Conditional Broadcast e Fixed Probability
- nella seconda sezione vengono confrontati tra loro i quattro algoritmi a due fasi proposti, sempre con parametro waiting steps pari a 1
- nella terza sezione viene analizzato il parametro waiting steps dei quattro algoritmi a due fasi
- nella quarta sezione viene analizzato invece il parametro Keep TTL nell'algoritmo TwoPhases FPFP

5.1 Two Phases FPFP, Conditional Broadcast e Fixed Probability

In questa sezione vengono analizzati i risultati raggiunti dall'algoritmo Two Phases FPFP con tempo di attesa pari a 1 (WS1) con i due algoritmi canonici Conditional Broadcast e Fixed Probability.

Grafi Random con 500 nodi e 1000 archi

I risultati ottenuti dalle simulazioni fatte su grafi di tipo random con mille archi mostrano un miglior rapporto tra copertura e overhead e un ovvio, data la natura dell'algoritmo, aumento del delay. Una percentuale di copertura del 95% viene raggiunta con un overhead di 1.73 da TwoPhases FPFP contro 1.80 di Fixed Probability con un miglioramento quindi di circa il 4%. Il delay di TwoPhases FPFP è molto elevato con probabilità di disseminazioni basse, ma va ad avvicinarsi a quello riscontrato negli altri algoritmi per percentuali di copertura più elevate. Infatti, con un overhead pari a 1.73, valore per il quale la copertura raggiunge il 95%, avremo un delay di 6.9 per TwoPhases FPFP contro i 6.2 per Fixed Probability e 5.8 per Conditional Broadcast.



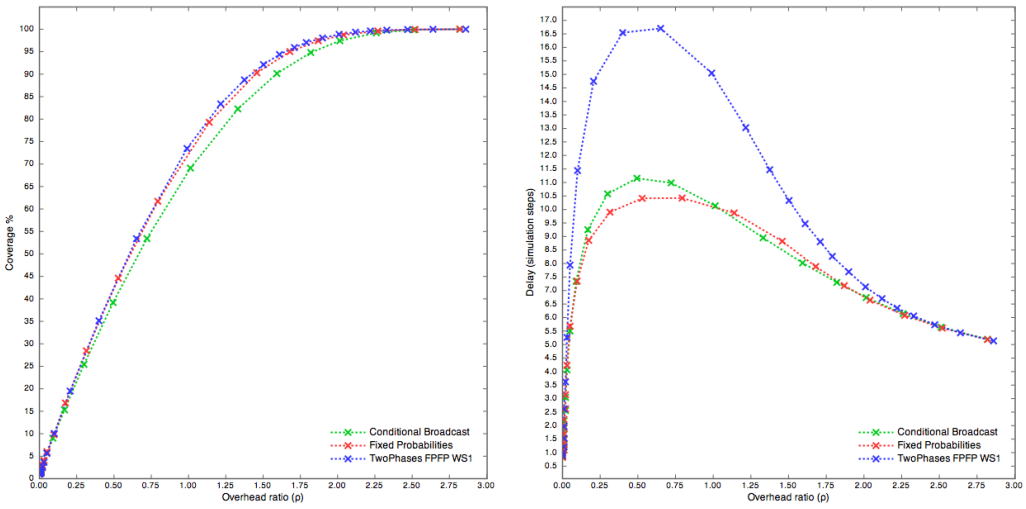
(a) Coverage (b) Delay

Figura 5.1: Grafi Random, 500 nodi, 1000 archi, diametro massimo = 12, diametro medio = 10,10

Grafi K-Regular con 500 nodi e 1000 archi

Anche nei grafi K-Regular abbiamo un miglioramento nella copertura raggiunta, ma inferiore rispetto a quanto visto nei grafi Random. Infatti per il 95% di copertura l'overhead risulta di 1.65 per il TwoPhases FPFP, di 1.68

per il Fixed Probability, con un miglioramento inferiore al 2%. Conditional Broadcast arriva ad un overhead di 1.82. Il delay è invece di 9.2 per Two-Phases FPFP contro 8 di Fixed Probability e 7.8 di Conditional Broadcast.



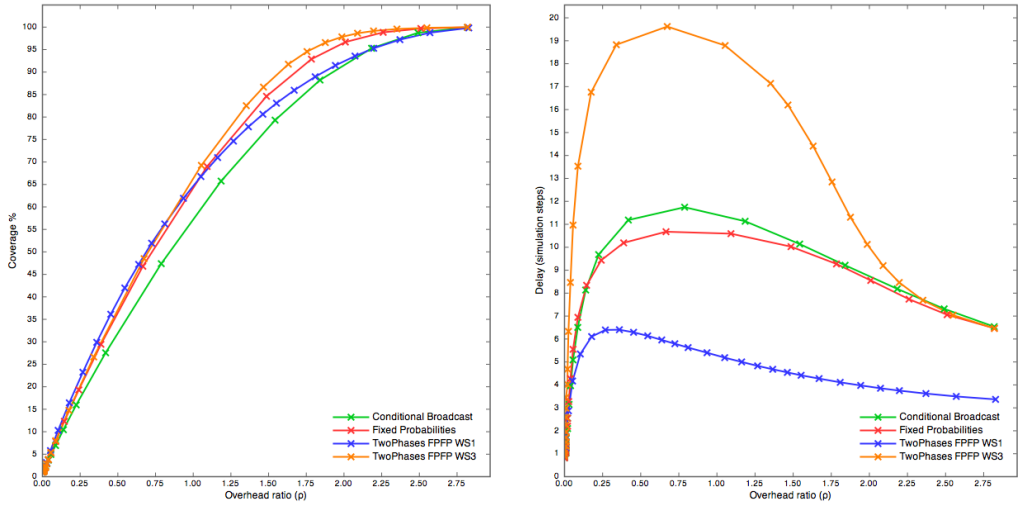
(a) Coverage (b) Delay

Figura 5.2: Grafi K-regular, 500 nodi, 1000 archi, diametro massimo = 8, diametro medio = 8

Grafi Scale-free con 500 nodi e 997 archi

Nel caso dei grafi Scale-free abbiamo invece dei risultati imprevisti. Probabilmente a causa della loro struttura particolare, con un Waiting Steps pari a 1 TwoPhases FPFP raggiunge livelli di copertura inferiori rispetto a Fixed Probability e del tutto simili a Conditional Broadcast. L'altro risultato inaspettato è il delay, che risulta nettamente inferiore ai due algoritmi di riferimento. Per la copertura al 95% avremo infatti un overhead di 1.91 per Fixed Probabilities e di 2.17 per Conditional Broadcast e TwoPhases FPFP. In compenso il delay risulta più che dimezzato, con un valore di 4 per TwoPhases FPFP contro circa 9 per gli algoritmi classici. Nel grafico è stato inserito in questa occasione anche la versione WS3 dell'algoritmo in due fasi,

a mostrare che con un tempo di attesa diverso l'anomalia sparisce e abbiamo risultati in linea con quelli visti nelle altre tipologie di grafo. Un'analisi più dettagliata del confronto tra FPFP WS1 e WS3 verrà fatta però in seguito.



(a) Coverage (b) Delay

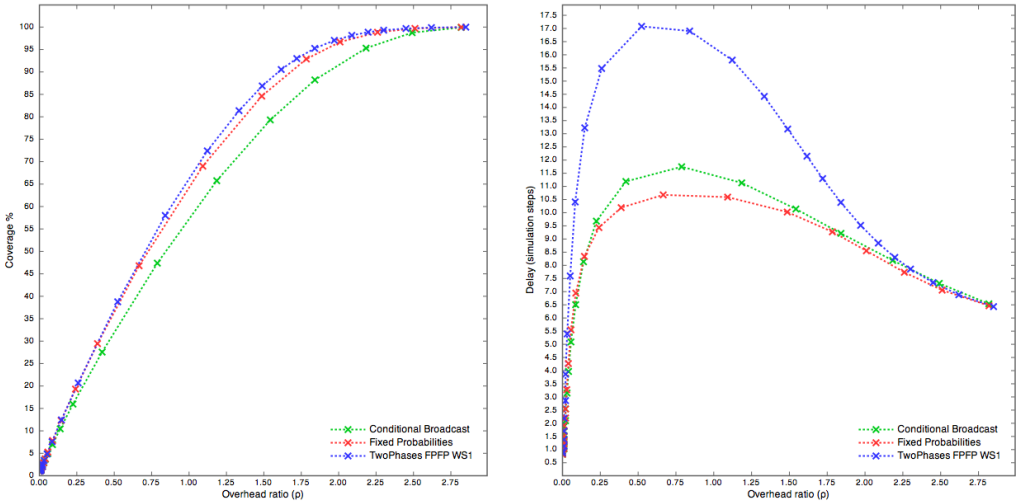
Figura 5.3: Grafi Scale-free, 500 nodi, 997 archi, diametro massimo = 7, diametro medio = 6,10

Grafi Small-world con 500 nodi e 1000 archi

Anche nei grafi Small-world TwoPhases FPFP raggiunge una percentuale di copertura maggiore. L'overhead per il 95% di copertura raggiunta è di 1.82 per TwoPhases FPFP, 1.90 per Fixed Probability e 2.17 per Conditional Broadcast. Il delay passa invece dal 9.2 degli algoritmi classici a 10.6.

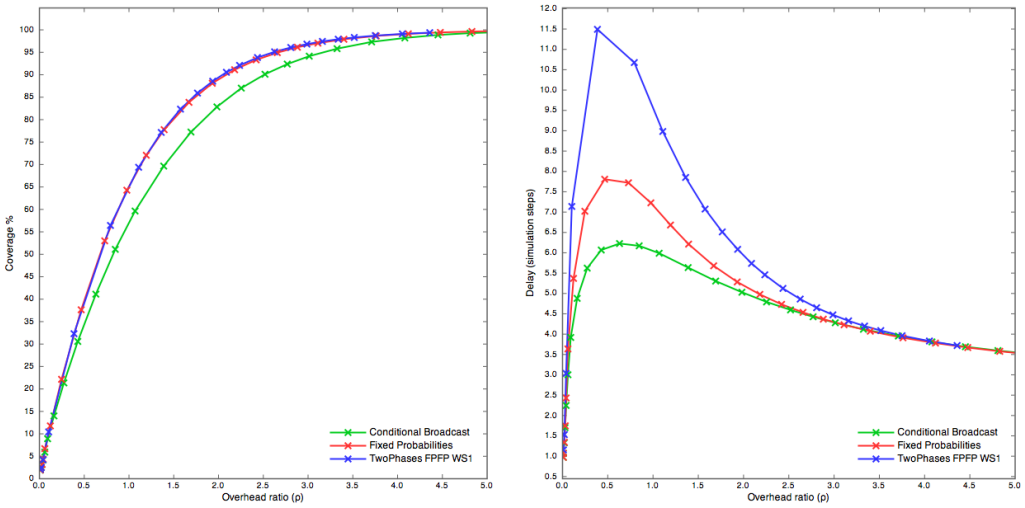
Grafi con 2000 (1990) archi

Aumentando il numero di archi la disseminazione diventa più rapida e l'algoritmo TwoPhases FPFP risulta meno efficace. Infatti il miglioramento nella copertura raggiunta è pressoché impercettibile, fino a risultare inferiore nei grafi Scale-free 1990 (dove però sparisce l'anomalia riscontrata con 997 archi). Il delay rimane ovviamente maggiore.



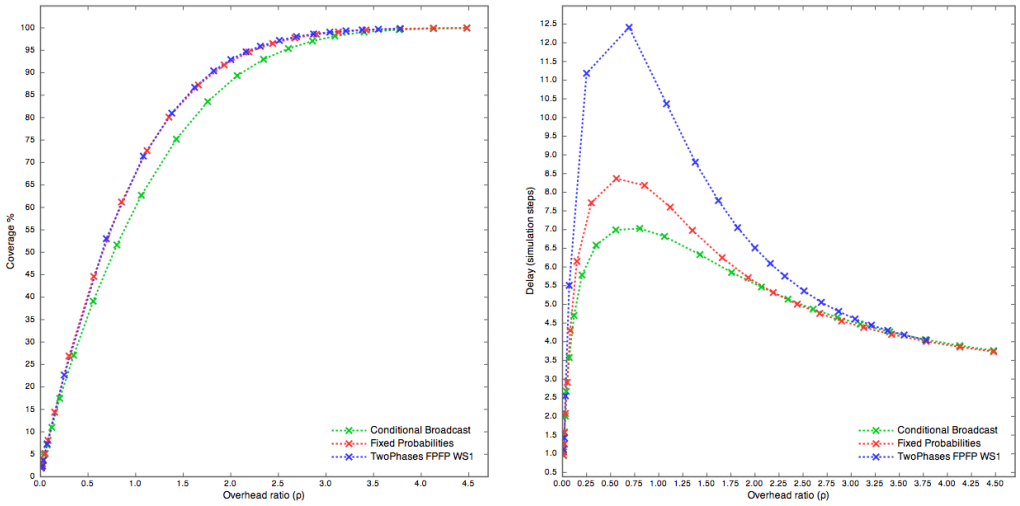
(a) Coverage (b) Delay

Figura 5.4: Grafi Small-world, 500 nodi, 1000 archi, diametro massimo = 13, diametro medio = 11,30



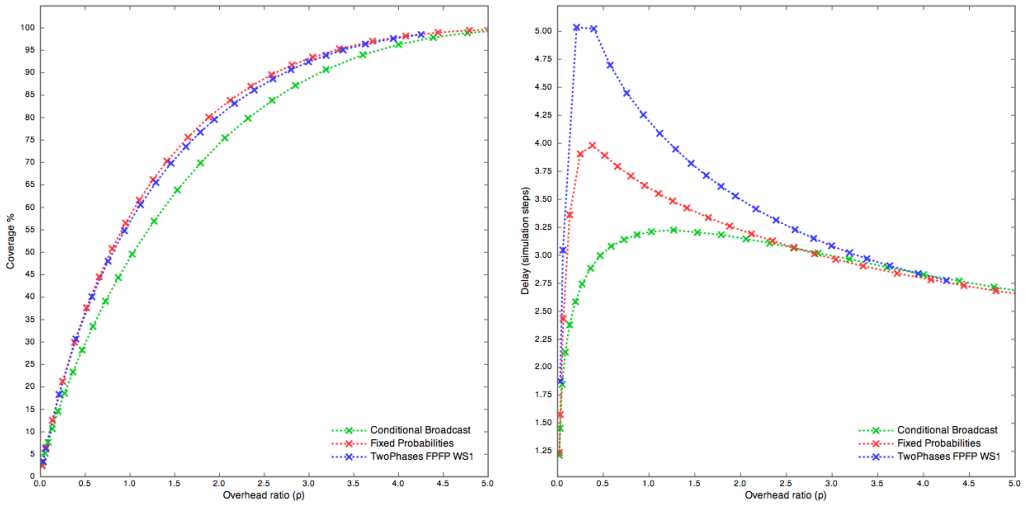
(a) Coverage (b) Delay

Figura 5.5: Grafi Random, 500 nodi, 2000 archi, diametro massimo = 6, diametro medio = 5,80



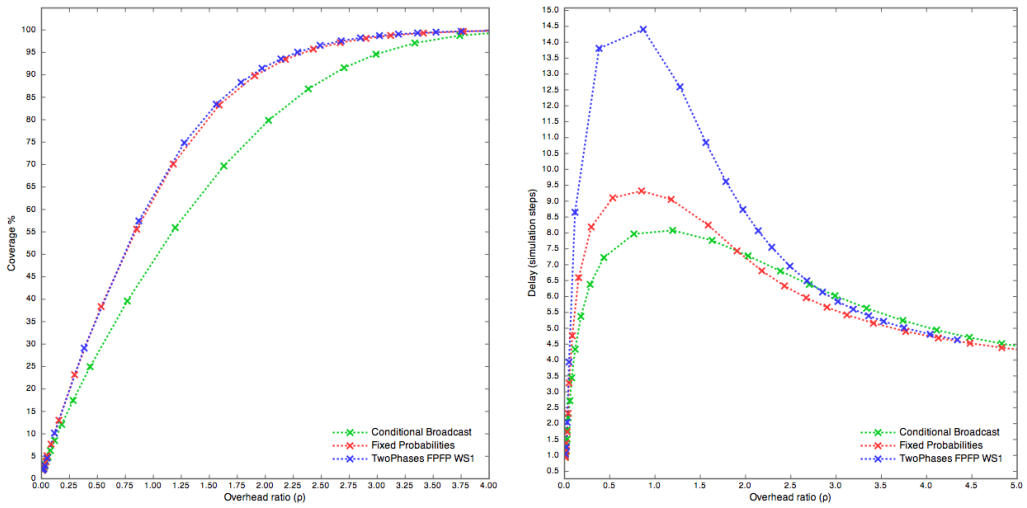
(a) Coverage (b) Delay

Figura 5.6: Grafi K-regular, 500 nodi, 2000 archi, diametro massimo = 5, diametro medio = 5



(a) Coverage (b) Delay

Figura 5.7: Grafi Scale-free, 500 nodi, 1990 archi, diametro massimo = 4, diametro medio = 4

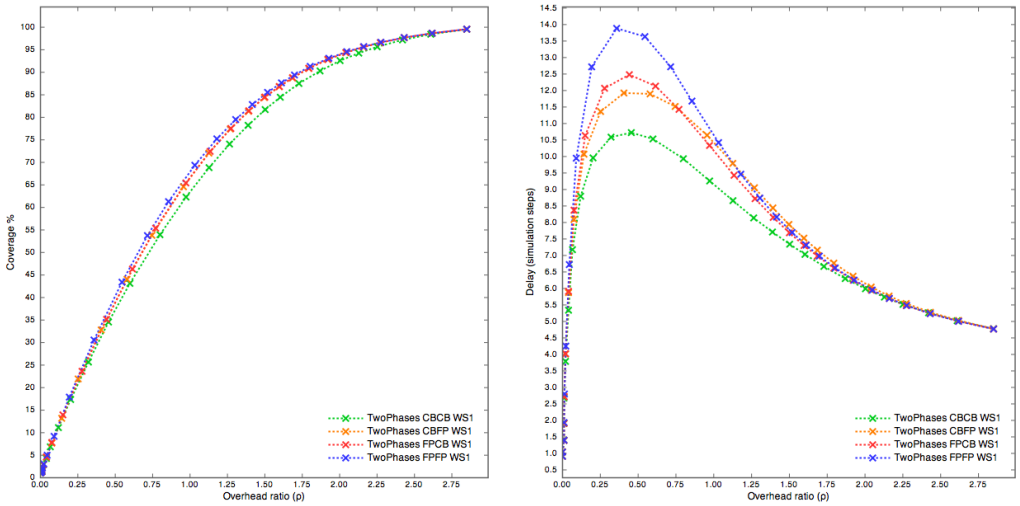


(a) Coverage (b) Delay

Figura 5.8: Grafi Small-world, 500 nodi, 2000 archi, diametro massimo = 7, diametro medio = 6,30

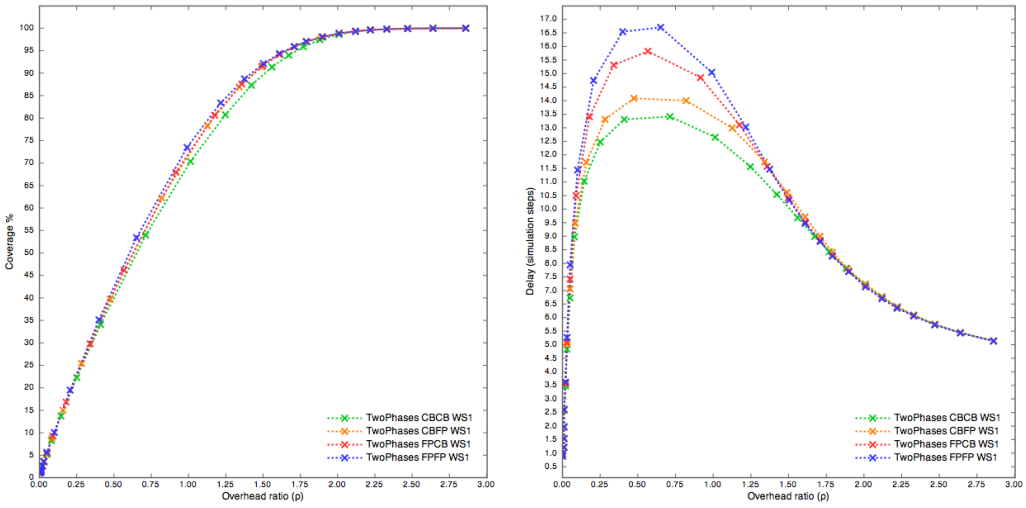
5.2 Two Phases

Passiamo ora a confrontare tra loro gli algoritmi a due fasi proposti. Possiamo notare che in tutte le tipologie di prove effettuate si ritrovano le caratteristiche degli algoritmi classici utilizzati nelle due fasi. Utilizzare due volte Fixed Probability porta risultati migliori nella copertura, così come il doppio Conditional Broadcast risulta quello peggiore. Allo stesso tempo, così come un Conditional Broadcast generalmente riesce ad effettuare la disseminazione con un delay inferiore rispetto al Fixed Probabilities, anche negli algoritmi a due fasi CBCB risulta quello con delay minore. Ovviamente i due algoritmi "misti", FBCB e CBFP vanno a posizionarsi a metà strada in entrambi i casi. Ad alte percentuali di copertura, i delay dei quattro algoritmi vanno a convergere. Solo nel caso dei grafi Scale-free si può notare come a livello di delay un Conditional Broadcast eseguito durante la seconda fase risulti invece penalizzante.



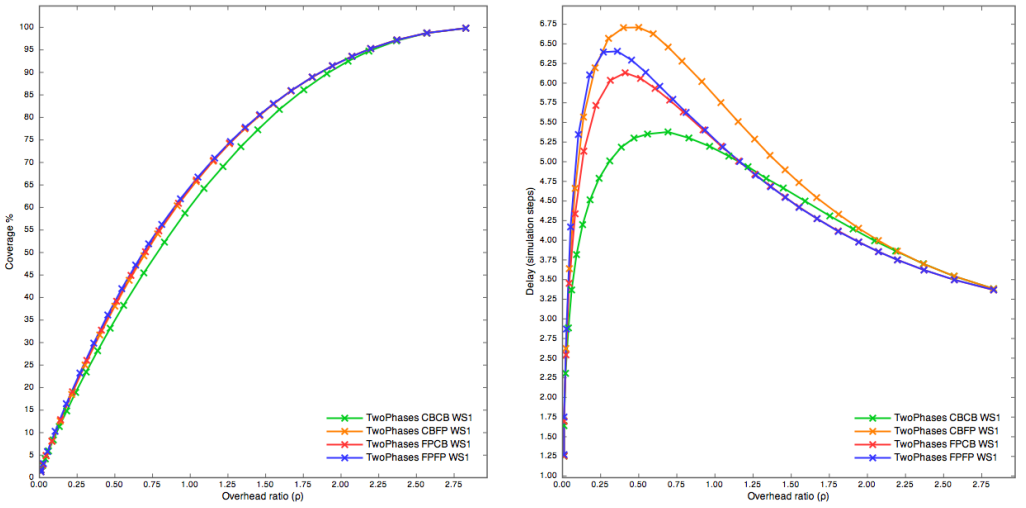
(a) Coverage (b) Delay

Figura 5.9: Grafi Random, 500 nodi, 1000 archi, diametro massimo = 12, diametro medio = 10,10



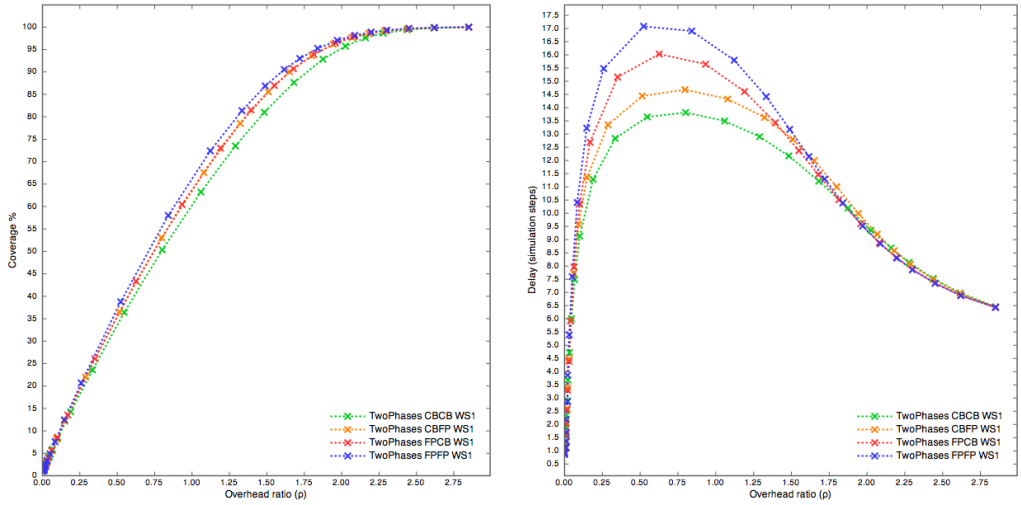
(a) Coverage (b) Delay

Figura 5.10: Grafi K-regular, 500 nodi, 1000 archi, diametro massimo = 8, diametro medio = 8



(a) Coverage (b) Delay

Figura 5.11: Grafi Scale-free, 500 nodi, 997 archi, diametro massimo = 7, diametro medio = 6,10



(a) Coverage (b) Delay

Figura 5.12: Grafi Small-world, 500 nodi, 1000 archi, diametro massimo = 13, diametro medio = 11,30

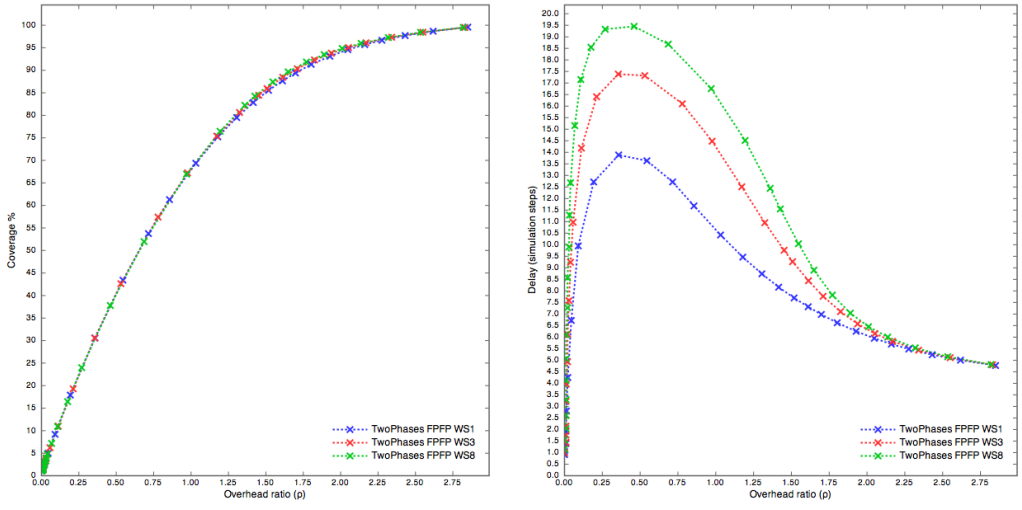
5.3 Waiting Steps

Il parametro Waiting Steps è stato analizzato ipotizzando un raggiungimento di una percentuale di copertura maggiore, a costo di un ovvio sacrificio in termini di delay. I risultati sono stati però abbastanza deludenti.

5.3.1 FPFP

Grafi Random con 500 nodi e 1000 archi

Nei grafi Random con 1000 archi notiamo un leggero miglioramento aumentando il tempo di attesa. Il 95% di copertura viene infatti raggiunto con un overhead pari a 2.09 con WS1 e 2.02 con WS8. WS3 è a 2.04. L'overhead aumenta nettamente nelle disseminazioni che raggiungono una copertura non molto elevata, per poi diventare simile tra i tre algoritmi. Infatti con 2.09 di overhead, la differenza è di circa 0.2 step di simulazione tra un algoritmo e l'altro, a partire dal 5.8 del WS1, fino al 6.2 del WS8.

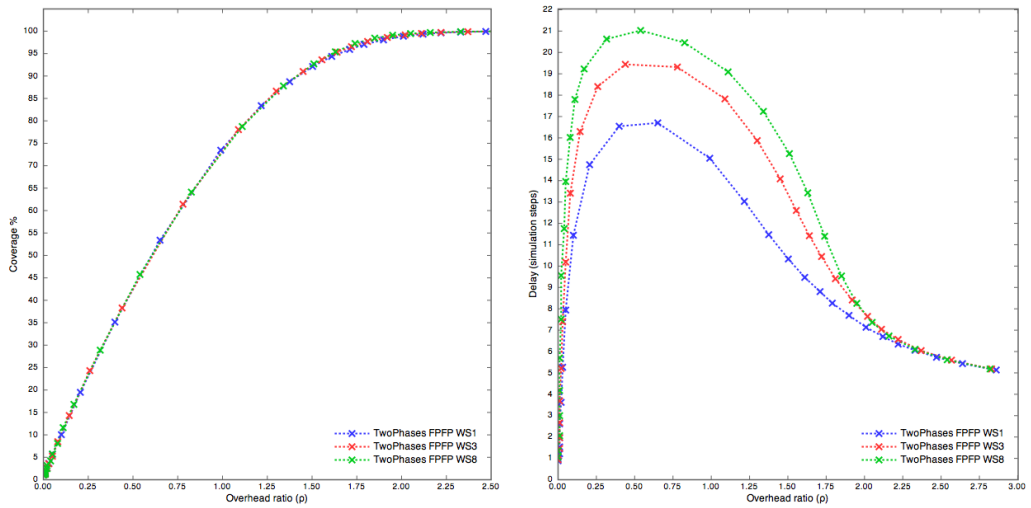


(a) Coverage (b) Delay

Figura 5.13: Grafi Random, 500 nodi, 1000 archi, diametro massimo = 12, diametro medio = 10,10

Grafi K-regular con 500 nodi e 1000 archi

Anche nei grafi K-regular vediamo un piccolo miglioramento con WS più elevato. Per raggiungere il 95% di copertura infatti avremo rispettivamente 1.61, 1.62 e 1.65 di overhead per WS8, WS3 e WS1. Il delay però è decisamente maggiore a questo livello di overhead. Tra WS1 e WS8 ci sono infatti più di 4 step di simulazione di differenza, ovvero da circa 9 a circa 13.5 di delay. WS3 ottiene invece 11.5.



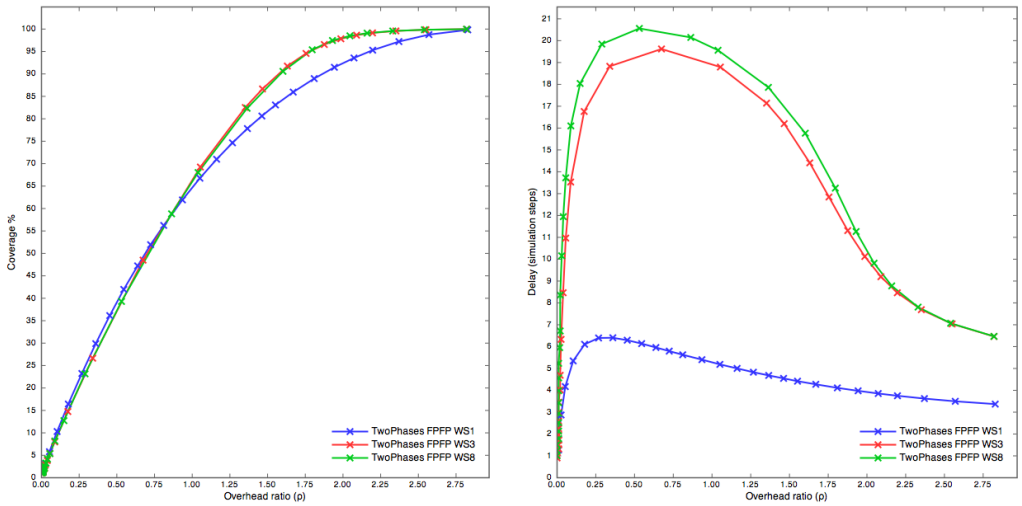
(a) Coverage (b) Delay

Figura 5.14: Grafi K-regular, 500 nodi, 1000 archi, diametro massimo = 8, diametro medio = 8

Grafi Scale-free con 500 nodi e 997 archi

Nei grafi Scale-free vediamo l'anomalia di cui avevamo accennato. Gli algoritmi a due fasi con WS1 in questo gruppo di grafi raggiunge valori di copertura decisamente inferiori, con overhead di 2.17 per una copertura del 95% contro l'1.78 di WS3 e WS8. Tra WS3 e WS8 non ci sono invece praticamente benefici. Per quanto riguarda il delay troviamo invece la disseminazione con WS1 incredibilmente "rapida", anche migliore rispetto agli

algoritmi classici. Con l'overhead a 2.17 infatti il delay di WS1 è di circa 3.77, contro il 13.5 e 12.5 di WS8 e WS3.

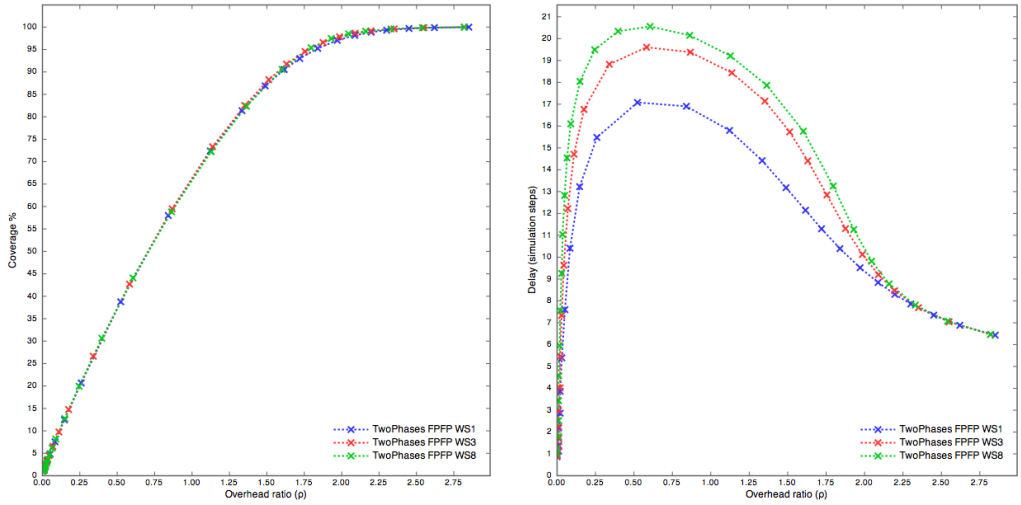


(a) Coverage (b) Delay

Figura 5.15: Grafi Scale-free, 500 nodi, 997 archi, diametro massimo = 7, diametro medio = 6,10

Grafi Small-world con 500 nodi e 1000 archi

Nei grafi Small-world il comportamento è simile a quanto avviene in quelli Random e nei K-Regular, con un miglioramento risibile della percentuale di copertura con WS8, ma un delay aumentato.

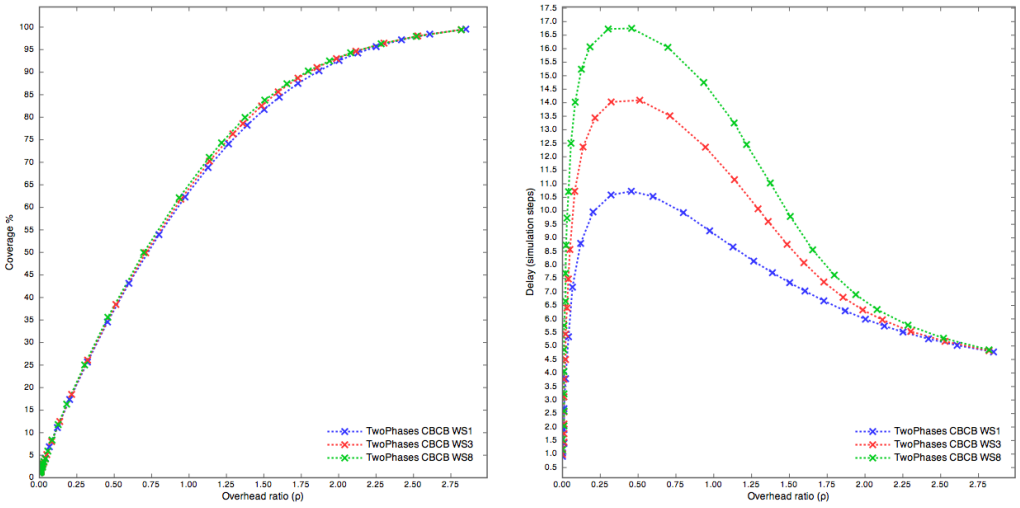


(a) Coverage (b) Delay

Figura 5.16: Grafi Small-world, 500 nodi, 1000 archi, diametro massimo = 13, diametro medio = 11,30

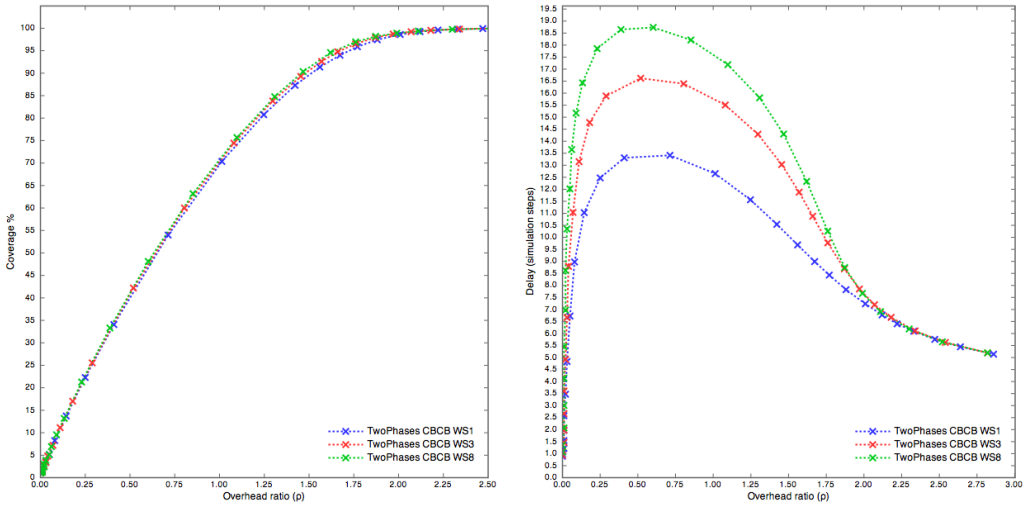
5.3.2 CBCB, CBFP, FPCB

Gli andamenti dei risultati non si discosta da quelli visti in FPFP. Anche l'anomalia dei risultati nei grafi Scale-free con WS pari a 1 si presenta nello stesso modo con tutti e tre gli altri algoritmi.



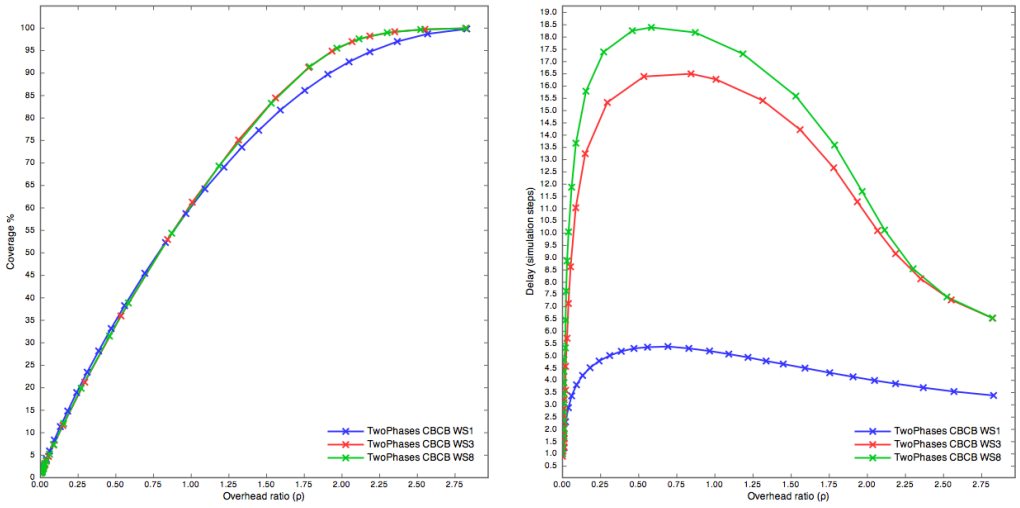
(a) Coverage (b) Delay

Figura 5.17: Grafi Random, 500 nodi, 1000 archi, diametro massimo = 12, diametro medio = 10,10



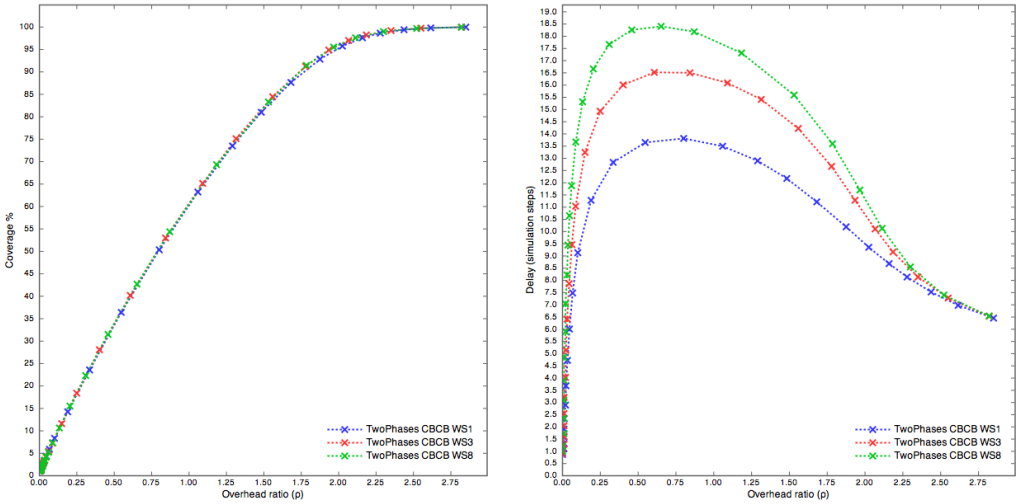
(a) Coverage (b) Delay

Figura 5.18: Grafi K-regular, 500 nodi, 1000 archi, diametro massimo = 8, diametro medio = 8



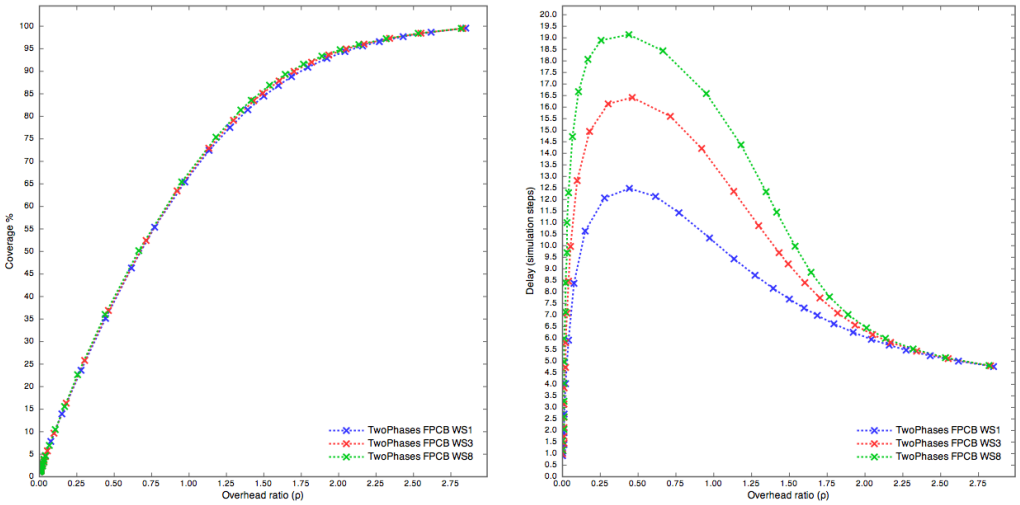
(a) Coverage (b) Delay

Figura 5.19: Grafi Scale-free, 500 nodi, 997 archi, diametro massimo = 7, diametro medio = 6,10



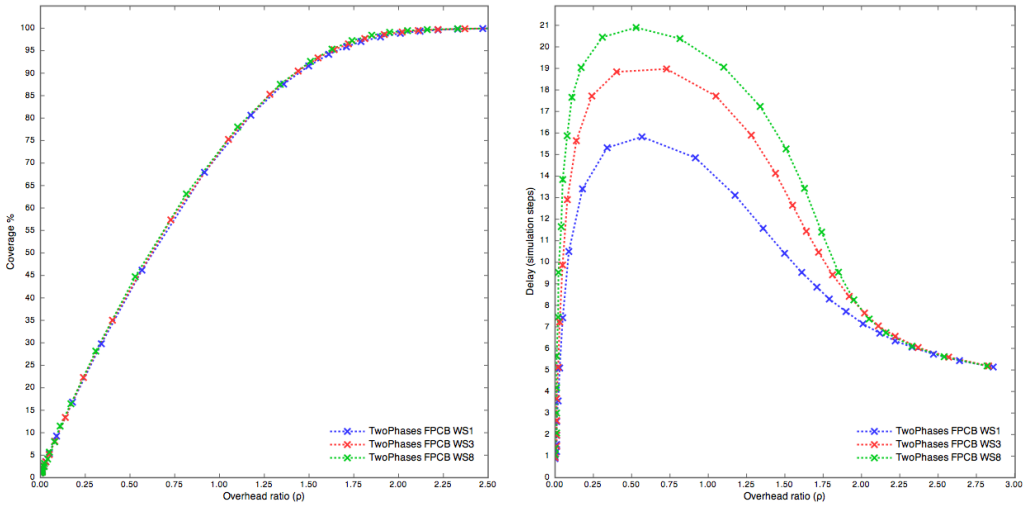
(a) Coverage (b) Delay

Figura 5.20: Grafi Small-world, 500 nodi, 1000 archi, diametro massimo = 13, diametro medio = 11,30



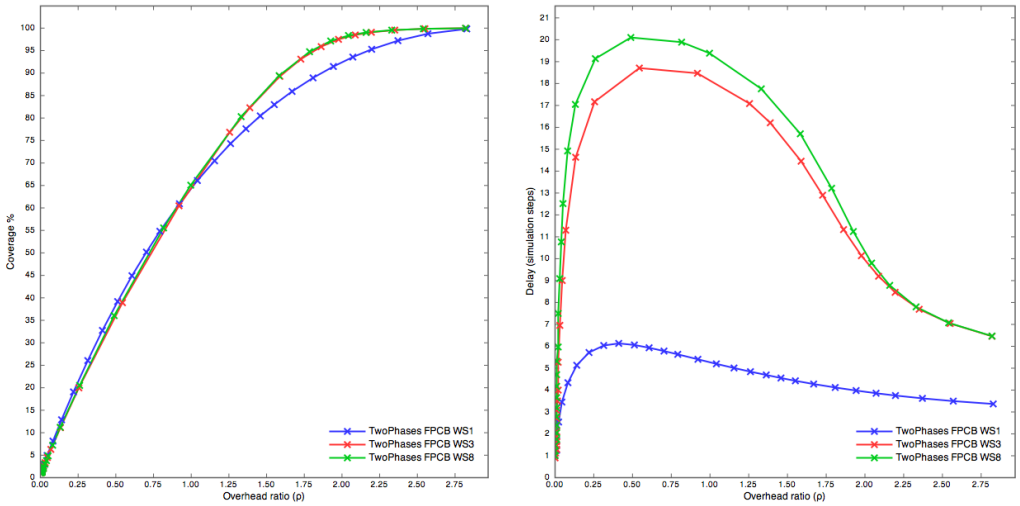
(a) Coverage (b) Delay

Figura 5.21: Grafi Random, 500 nodi, 1000 archi, diametro massimo = 12, diametro medio = 10,10



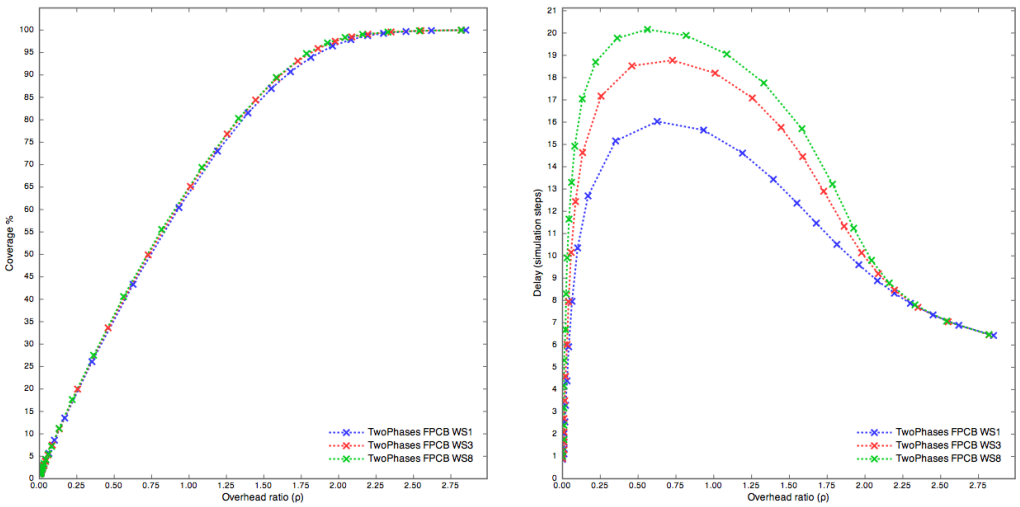
(a) Coverage (b) Delay

Figura 5.22: Grafi K-regular, 500 nodi, 1000 archi, diametro massimo = 8, diametro medio = 8



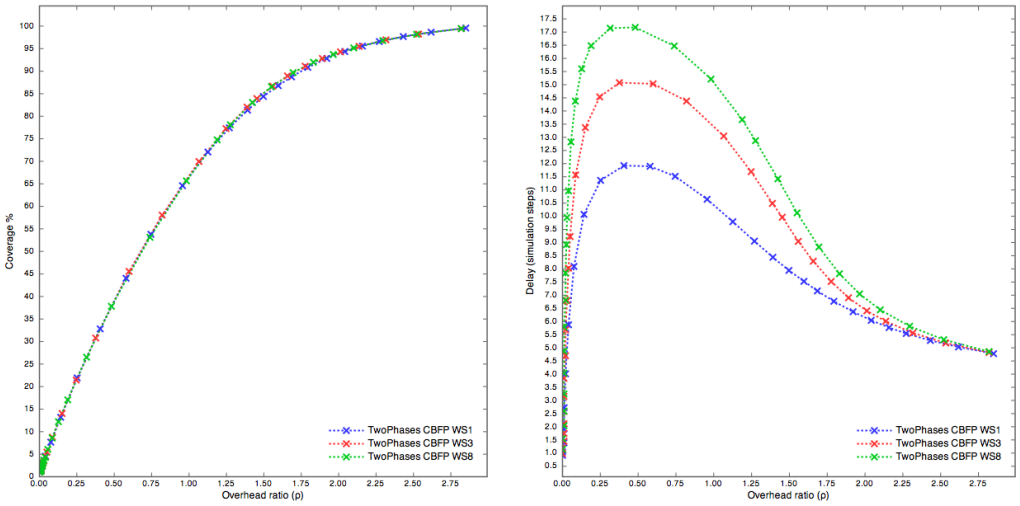
(a) Coverage (b) Delay

Figura 5.23: Grafi Scale-free, 500 nodi, 997 archi, diametro massimo = 7, diametro medio = 6,10



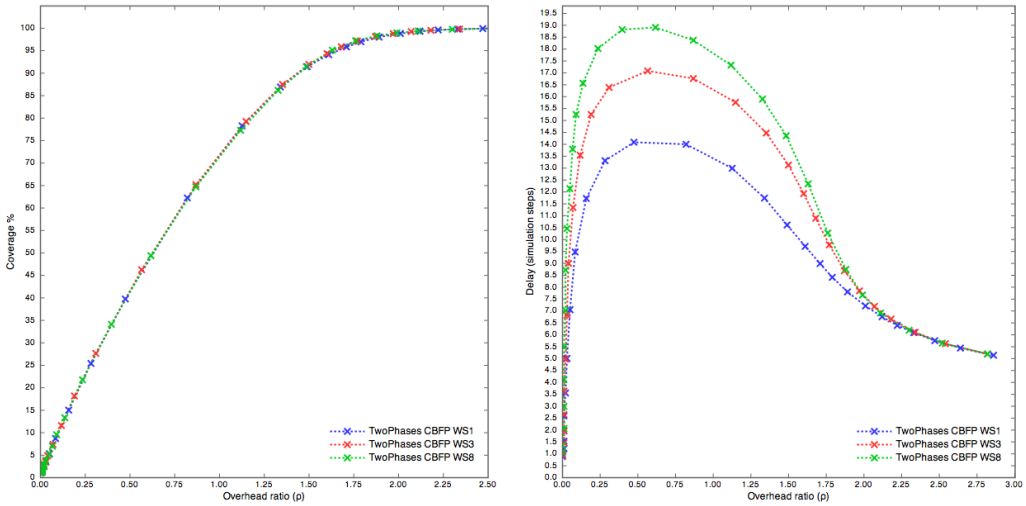
(a) Coverage (b) Delay

Figura 5.24: Grafi Small-world, 500 nodi, 1000 archi, diametro massimo = 13, diametro medio = 11,30



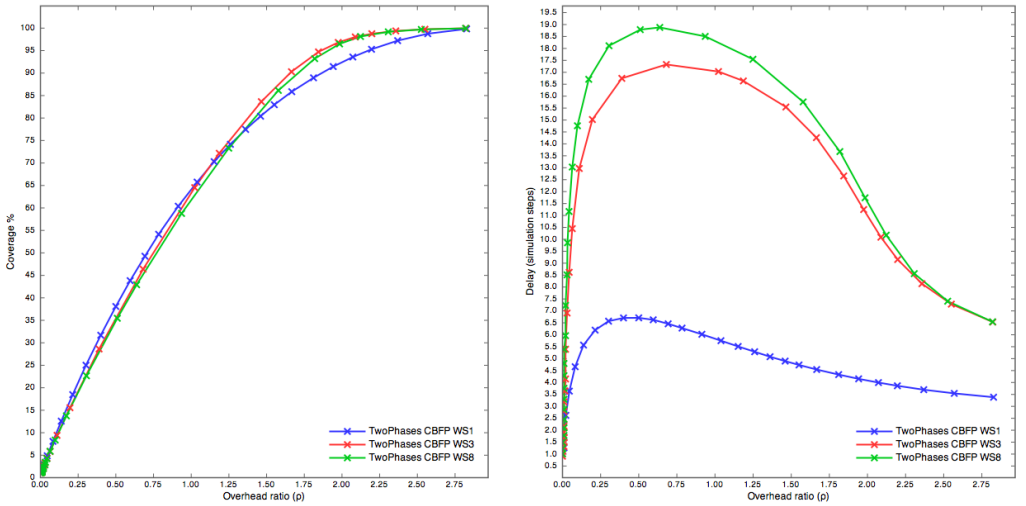
(a) Coverage (b) Delay

Figura 5.25: Grafi Random, 500 nodi, 1000 archi, diametro massimo = 12, diametro medio = 10,10



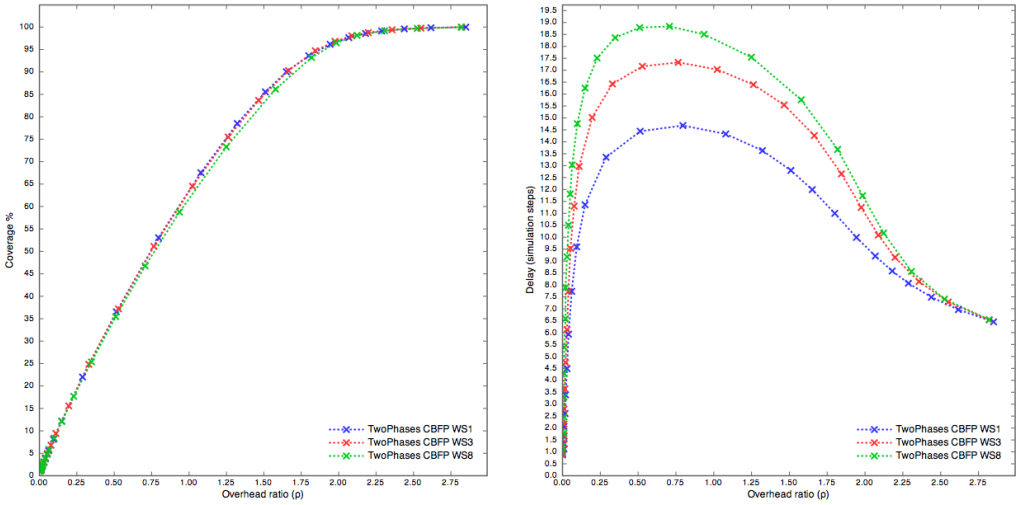
(a) Coverage (b) Delay

Figura 5.26: Grafi K-regular, 500 nodi, 1000 archi, diametro massimo = 8, diametro medio = 8



(a) Coverage (b) Delay

Figura 5.27: Grafi Scale-free, 500 nodi, 997 archi, diametro massimo = 7, diametro medio = 6,10

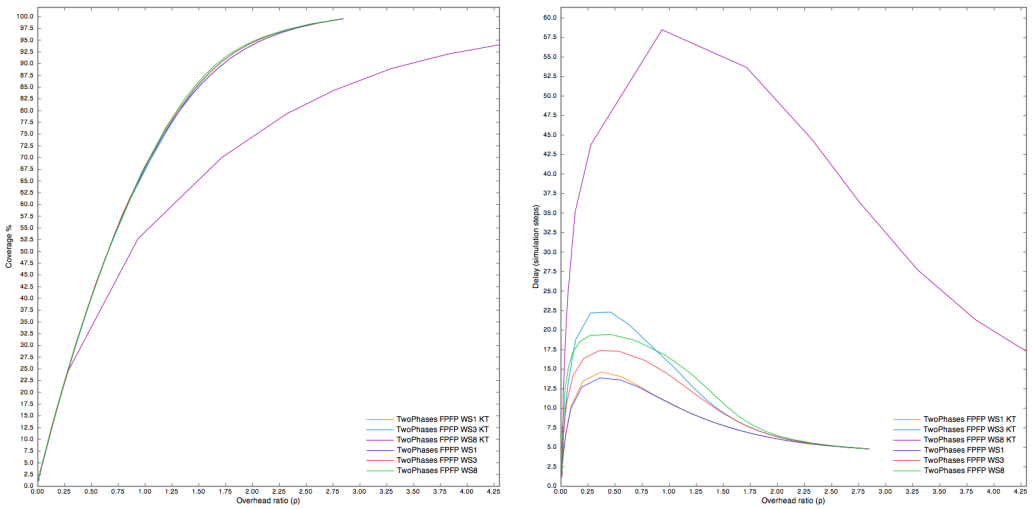


(a) Coverage (b) Delay

Figura 5.28: Grafi Small-world, 500 nodi, 1000 archi, diametro massimo = 13, diametro medio = 11,30

5.4 Il parametro Keep TTL (KT)

L'analisi del parametro KT è stata soltanto abbozzata. Infatti si può notare subito come in questa configurazione, con TTL già elevato in rapporto al diametro dei grafi, il mantenere i messaggi all'interno della rete per molto tempo porta alla saturazione della cache e alla ritrasmissione continua di messaggi "vecchi", problema evidente nella disseminazione con WS8 e KT. Con WS1 e WS3 il messaggio rimane in attesa per un tempo minore e i valori di copertura sono in linea con le disseminazioni senza Keep TTL. Il delay però risulta maggiore, senza portare benefici alla disseminazione. Si può concludere quindi che, almeno in una configurazione come questa, mantenere il TTL tra le due fasi di disseminazione non è una strategia ottimale.



(a) Coverage (b) Delay

Figura 5.29: Grafi Random, 500 nodi, 1000 archi, diametro massimo = 12, diametro medio = 10,10

Capitolo 6

Conclusioni e sviluppi futuri

Gli algoritmi a due fasi hanno dimostrato di poter raggiungere livelli di copertura spesso migliori rispetto ai classici Fixed Probability e Conditional Broadcast, pur con un delay maggiore. Sono quindi algoritmi interessanti per quei contesti dove lo scopo è diffondere l'informazione minimizzando l'overhead senza avere stringenti requisiti per quanto riguarda il tempo di attesa. L'analisi fatta si può considerare preliminare, infatti molti aspetti, per questioni di tempo e risorse computazionali disponibili non sono state analizzate nel dettaglio.

6.1 Analisi di WS e KT

Il parametro WS può essere esaminato in maniera più approfondita. Le prove sono state fatte con valori di 1, 3 e 8 per cercare un ipotetico andamento dei risultati supponendo di raggiungere maggiori livelli di copertura con tempi di attesa più lunghi. È però probabile che la dimensione limitata della cache possa "nascondere" eventuali risultati positivi. Infatti all'aumentare del tempo di attesa aumenta anche il numero di messaggi diversi presenti nello stesso momento all'interno della rete. Per questo motivo alcuni nodi potrebbero ritrasmettere messaggi vecchi ma già rimossi dalla cache. Per lo stesso motivo bisognerebbe analizzare meglio KT, parametro che può

far rimanere un messaggio nella rete per molto più tempo rispetto al TTL iniziale.

6.2 Cache e TTL

In questa tesi cache e TTL sono stati impostati a valori abbastanza elevati per minimizzare il loro impatto sui risultati. Ci sono stati però casi dove la saturazione della cache risulta evidente e altri dove potrebbe aver influito aumentando l'overhead. Un lavoro più preciso da questo punto di vista, come svolto in [22] è necessario per capire quante di quante "risorse" in più sono necessarie per una efficiente disseminazione in due fasi.

6.3 Altri algoritmi

L'approccio in due fasi può essere adattato a qualsiasi tipo di algoritmo. In questa tesi sono stati utilizzati come base i due canonici, ma nel tempo sono stati proposti approcci più sofisticati, dagli algoritmi adattivi [5], agli algoritmi "degree-dependent" [16] a svariate altre proposte. L'approccio a due fasi potrebbe essere applicato anche a questi algoritmi.

6.4 Grafi dinamici

Una delle caratteristiche delle reti peer-to-peer è l'essere altamente dinamiche. In questa tesi sono state invece utilizzate solo reti statiche, con nodi immuni da qualsiasi tipo di problematica, dal crash alla perdita di pacchetti. Può essere interessante analizzare se l'approccio in due fasi può dare buoni risultati in questo contesto, proprio grazie al suo doppio invio a distanza di tempo.

Bibliografia

[1] *Parallel And Distributed Simulation (PADS) Research Group, http://pads.cs.unibo.it*, (2015).

[2] *igraph the network analysis package, http://http://igraph.org*, (2015).

[3] C. Anagnostopoulos, O. Sekkas, S. Hadjiefthymiades, *An adaptive epidemic information dissemination model for wireless sensor networks*, Pervasive Mob. Comput. 8 (5) (2012) 751–763

[4] B. Garbinato, D. Rochat, M. Tomassini, *Impact of scale-free topologies on gossiping in ad hoc networks*, in: NCA, IEEE Computer Society, 2007, pp. 269–272.

[5] G. D'Angelo, S. Ferretti, M. Marzolla, *Adaptive event dissemination for peer-to-peer multiplayer online games*, in Proceedings of Workshop on DIstributed SImulation and Online gaming (DISIO 2011), ICST, ICST, Brussels, Belgium, Belgium, 2011.

[6] R. Baldoni, R. Beraldi, V. Quema, L. Querzoni, S. Tucci-Piergiovanni, *Tera: Topic-based event routing for peer-to-peer architectures*, in Proceedings of the 2007 Inaugural International Confer- ence on Distributed Event-based Systems, DEBS '07, ACM, New York, NY, USA, 2007, pp. 2–13

[7] S. Ferretti, *Publish-subscribe systems via gossip: A study based on complex networks*, in: Proceed- ings of the Fourth Annual Workshop on Sim-

plifying Complex Networks for Practitioners, SIMPLEX '12, ACM, New York, NY, USA, 2012, pp. 7–12.

[8] S. Voulgaris, E. Riviere, A. Kermarrec, M. van Steen, *Sub-2-sub: Self-organizing content-based publish subscribe for dynamic large scale collaborative networks*, in: 5th International workshop on Peer-To-Peer Systems, IPTPS 2006, Santa Barbara, CA, USA, February 27-28, 2006.

[9] M. Marzolla, O. Babaoglu, F. Panzieri, *Server consolidation in clouds through gossiping* in: World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2011 IEEE International Symposium on a, 2011, pp. 1–6.

[10] Amazon s3 availability event: July 20, 2008 (July 2008).

[11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubrama- nian, P. Vosshall, W. Vogels, *Dynamo: Amazon's highly available key-value store*, SIGOPS Oper. Syst. Rev. 41 (6) (2007) 205–220.

[12] A. Lakshman, P. Malik, *Cassandra: A decentralized structured storage system*, SIGOPS Oper. Syst. Rev. 44 (2) (2010) 35–40.

[13] P. Erdös, A. Rényi, *On random graphs, I*, Publicationes Mathematicae (Debrecen) 6 (1959) 290– 297.

[14] R. Albert, A.-L. Barabási, *Statistical mechanics of complex networks*, Rev. Mod. Phys. 74 (2002) 47–97.

[15] D. J. Watts, S. H. Strogatz, *Collective dynamics of 'small-world' networks*, Nature393(6684)(1998) 409–10.

[16] Giulio Cirnigliaro, *Data dissemination on overlay networks through gossip*, Master's Degree thesis, 2013

[17] Sze-Yao Ni, Yu-Chee Tseng, Yuh-Shyan Chen, and Jang-Ping Sheu, *The Broadcast Storm Problem in a Mobile Ad Hoc Network*, International Conference on Mobile Computing and Networking, 8:151– 162, 1999.

[18] A. Saidi and M. Mohtashemi, *Minimum-cost first-push-then-pull gossip algorithm*, in Wireless Communications and Networking Conference (WCNC), 2012 IEEE, pp. 2554–2559, IEEE, 2012.

[19] R. Gupta, A.C. Maali and Y. N. Singh, *Adaptive Push-Then-Pull Gossip Algorithm for Scale-free Networks*, CoRR 2013.

[20] Márk Jelasity, *Gossip. Self-organising software*, 2011. pp 139 - 162.

[21] Ruijing Hu, Julien Sopena, Luciana Arantes, Pierre Sens and Isabelle Demeure, *A fair comparison of gossip algorithms over large-scale random topologies*

[22] Gabriele D'Angelo, Stefano Ferretti, *Performance evaluation of gossip protocols on complex networks*, in CoRR 2015

[23] Herbert W. Hethcote, *The Mathematics of Infectious Diseases*, SIAM review 42.4 (2000): 599-653.

[24] R. Brooks, *The spectral geometry ofk-regular graphs*, Journal d'Analyse Mathématique, Volume 57, Issue 1, 1991, pp 120-151.

[25] *The R Project for Statistical Computing https://www.r-project.org*, 2016

[26] Pascal Felber, Anne-Marie Kermarrec, Lorenzo Leonini, Etienne Rivière, Spyros Voulgaris, *PULP: An adaptive gossip-based dissemination protocol for multi-source message streams* in Peer-to-Peer Networking and Applications, March 2012, Volume 5, Issue 1, pp 74-91.