UNIVERSITÀ DEGLI STUDI DI ROMA
"TOR VERGATA"

FACOLTÀ DI INGEGNERIA
DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

TESI DI LAUREA IN INGEGNERIA INFORMATICA

SVILUPPO DI UN PROTOCOLLO EPIDEMICO PER UN SISTEMA P2P DECENTRALIZZATO PURO

Relatore
Ing. Valeria Cardellini

Laureando:
Antonio Papa

Anno Accademico 2007/2008

Ringraziamenti

Un particolare ringraziamento ai miei genitori e a mia sorella, che mi sono stati sempre vicino nei momenti di sconforto e mai si sono opposti alle mie scelte universitarie.

Un ringraziamento a Valeria Cardellini, per la sua disponibilità e cortesia dimostrata durante lo sviluppo della tesi. Grazie a lei ho affrontato un percorso che mi ha dato molte soddisfazioni e fatto conoscere lati del mondo dell'informatica che non avevo ancora approfondito.

Un ringraziamento alla nonna Giuseppa che con i suoi sformati di zucchine ha reso meno problematica la mia situazione di studente fuori sede.

Un ringraziamento ad Alessandro Pacca che ha reso le giornate universitarie meno tetre e mi è stato vicino durante i periodi di stallo con preziosi consigli. Se la mia visione del mondo universitario è mutata lo devo in particolar modo a questa persona. Inoltre senza il suo calcolatore non avrei potuto eseguire i primi test sul simulatore.

Infine un ringraziamento alle mie fonti di ispirazione: Golden Virginia, Sofje ed i Dream Theater.

Indice

Introduzione

Nell'ultimo decennio i sistemi P2P hanno riscontrato un notevole successo, sia presso il grande pubblico, sia negli ambienti scientifici e accademici. A differenza dell'architettura client/server, il modello P2P non possiede nodi gerarchizzati, ma un numero di *nodi equivalenti* (in inglese *peer*) che fungono sia da *client* che da *server* verso altri nodi della rete. I sistemi P2P completamente decentralizzati che organizzano la rete in modo non strutturato, rivestono un particolare interesse per la loro scalabilità e tolleranza alla faliure. La completa decentralizzazione è ottenuta grazie all'indipendenza di ogni peer della rete nell'effettuare operazioni di *lookup* e di scambio di risorse.
Gnutella è stato uno dei primi sistemi P2P ad avere un'architettura decentralizzata pura, il suo protocollo (la versione 0.4), definisce un insieme di messaggi (scambiati attraverso protocollo TCP) e di regole di *routing* che permettono ai diversi peer di comunicare tra loro. Gnutella pur godendo di notevoli vantaggi propri dei sistemi P2P completamente decentralizzati, generava un eccessivo traffico di rete dovuto all'eccessivo *flooding* dei messaggi di tipo multicast (Ping e Query).
Gli algoritmi epidemici offrono una soluzione interessante al problema del flooding nella rete, attraverso una rapida propagazione delle informazioni in un grande insieme di nodi. In questa classe di algoritmi tutti i nodi del sistema sono potenzialmente coinvolti nella diffusione delle informazioni. Fondamentale ogni nodo registra il messaggio che riceve in un buffer di capacità b e lo inoltra un numero di volte t. I nodi inoltrano i messaggi ogni volta ad un insieme casuale di loro vicini di cardinalità f (fan-out della disseminazione). Il modello proposto dagli algoritmi epidemici permette di diffondere le informazioni di *membership* senza generare eccessivo flooding nella rete.
Lo scopo della mia trattazione è presentare un protocollo epidemico che permetta di ridurre il *flooding* generato sia dalle informazioni di *membership* che dalle richieste di localizzazione di risorse.
Il protocollo 2Rounds che ho sviluppato è così composto:

- **Protocollo di rete 2Rounds:** definisce un insieme di messaggi (scambiati attraverso protocollo TCP) e di regole di *routing* per permettere ai peer di comunicare tra loro. Il protocollo di rete consta di due versioni:
  - 2Rounds 1.0: Le informazioni di membership sono diffuse in modo epidemico.
  - 2Rounds 2.0: Le informazioni di membership e di localizzazione delle risorse sono diffuse in modo epidemico.
- **Algoritmo 2Rounds:** definisce l'insieme di strutture e processi, che permettono la memorizzazione la ricezione, l' elaborazione e l'invio di messaggi nella rete.

Inoltre ho deciso di sviluppare un'applicazione di rete basata sul protocollo 2Rounds, realizzata in lin-

guaggio C, per un SO di tipo UNIX.
Nell'implementazione dell'applicazione, mi sono servito dell'API del socket di Berkley per lo sviluppo della parte relativa alla programmazione di rete, mentre ho usato la programmazione multi-threading ed i semafori per lo sviluppo della parte concorrente. Il test funzionale sull'applicazione ha richiesto l'implementazione di un *server di bootstrap*, cosi da permettere ai vari peer di sistema di comunicare tra loro sia in rete che in locale.
Infine per mostrare i vantaggi del protocollo *2Rounds* è stato creato un simulatore che mostra il grado di scalabilità del sistema attraverso l'analisi della *view* dei nodi.

Organizzazione

La tesi è organizzata come segue:

- Nel Capitolo 1 si introducono i principali concetti alla base degli ambienti peer-to-peer, le possibili rete P2P e la loro architettura di sistema.
- Nel Capitolo 2 viene descritto il protocollo *Gnutella 0.4*, attraverso la descrizione dei messaggi, del meccanismo di *routing* e della gestione delle connessioni. Inoltre viene presentata la sua evoluzione, la versione 0.6.
- Nel Capitolo 3 si analizzano gli *algoritmi epidemici*, in particolare quelli basati sul modello di *anti-entropia* e di *diffusione del rumore* (*gossiping*). Viene presentato il problema dell'eliminazione dati ed una delle sue soluzioni: i *certificati di morte*. Infine viene descritto un algoritmo epidemico basato sul *gossiping* il *lightweight probabilistic broadcast* (*lpbcast*).
- Nel Capitolo 4 è presentato il protocollo di rete il *2Rounds 1.0*, attraverso la descrizione dei messaggi del meccanismo di *routing* e della gestione delle connessioni. Inoltre viene presentata la versione successiva la 2.0 ed i suoi vantaggi rispetto alla prima.
- Nel Capitolo 5 è presentato l'algoritmo *2Round* che collabora con il suo protocollo di rete (la versione 2.0) al fine di creare un sistema P2P decentralizzato puro. In particolare dell'algoritmo saranno analizzate le strutture dati e di sincronizzazioni e le famiglie di processi.
- Nel Capitolo 6 viene mostrata l'applicazione realizzata attraverso le specifiche del protocollo *2Rounds*. Inoltre viene presentato un *server di bootstrap,* realizzato per testare l'applicazione sia in locale che in rete.

1. Sistemi peer-to-peer

1.1 Che cos'è una sistema peer-to-peer

Definizione 1.1(sistema peer-to-peer):

Un sistema peer-to-peer è un insieme di entità autonome (peers), capaci di auto-organizzarsi, che condividono un insieme di risorse distribuite presenti all'interno di una rete di calcolatori. Il sistema utilizza tali risorse per fornire una determinata funzionalità in modo completamente o parzialmente decentralizzato.

Generalmente per peer-to-peer (o P2P), cioè *rete/sistema paritaria/o*, si intende una classe di reti/sistemi ed applicazioni che utilizzano risorse distribuite e non possiedono nodi gerarchizzati come client o server fissi (clienti e serventi), ma un numero di *nodi equivalenti* (in inglese *peer*) che fungono sia da cliente che da servente verso altri nodi del sistema. Questo modello di rete/sistema, come mostrato in *figura* 1.1, è l'antitesi dell'architettura client-server.

Mediante questa configurazione, qualsiasi nodo è in grado di avviare o completare una transazione. I nodi equivalenti possono differire: nella configurazione locale, nella velocità di elaborazione, nell'ampiezza di banda e nella quantità di dati memorizzati. L'esempio classico di P2P è la/il rete/sistema per la condivisione di file (File sharing).

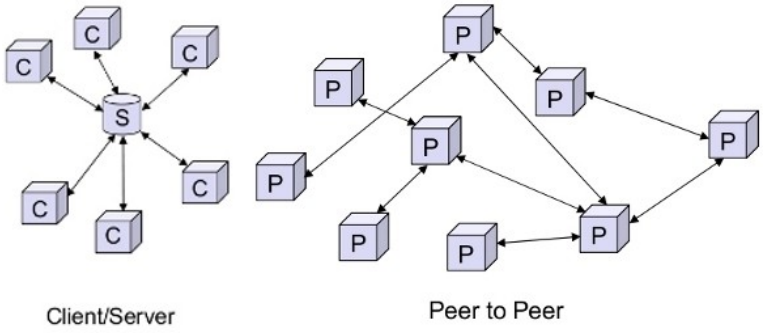


Figura 1.1:Architettura client/server e peer-to-peer a confronto

I nodi (peer) del sistema hanno identiche capacità e responsabilità (almeno in linea di principio) , infatti sono indipendenti (autonomi) e localizzati ai bordi (*edge*) di Internet, da ciò si evince che non è necessario nessun controllo centralizzato per il funzionamento del sistema. Ogni peer, a seguito della sua funzionalità simmetrica di client e server è anche definito *servent* = server + client. Le sue operazioni di base sono:

- **Boot:** Fase di ingresso nella infrastruttura P2P, può servirsi di *file di cache preesistenti*, *cache server* o *nodi centralizzati di bootstrap*.
- **Lookup:** Fase in cui il peer tenta di trovare il gestore/responsabile di una data risorsa.
- **Scambio di risorse:** Fase in cui avviene lo scambio di risorse tra due o più peer.

I sistemi P2P presentano un insieme di vantaggi, infatti basandosi su protocolli come *http* e *ftp*, prevedono un libero scambio di informazioni attraverso i *router* ed i *firewall*. Non richiedono operazioni di autenticazione, questo garantisce l'anonimato degli utenti del sistema. Inoltre non necessitano di amministratori e sono particolarmente semplici da utilizzare, in particolare il costo dell'aggiornamento delle informazioni che circolano in rete è praticamente nullo. I sistemi P2P a grazie alla loro natura decentralizzata sono estremamente tolleranti ai guasti poiché ,in linea di massima, non dipendono da una struttura centrale, quindi la perdita di un nodo non porta ad un isolamento totale dalle informazioni a cui si desidera accedere.

A fronte di tali vantaggi, notiamo che non è sempre facile riuscire a reperire i dati cercati, questo è dovuto essenzialmente alla struttura della rete. Inoltre la larghezza di banda del peer impone delle restrizioni, infatti ciascun nodo ha la libertà di stabilire quanti nodi possono collegarsi ad esso: un numero troppo elevato di collegamenti porta ad un consumo eccessivo della sua larghezza di banda, mentre un numero esiguo di collegamenti porta ad una vera e propria restrizione sulla possibilità di accesso alla rete, da parte di nuovi nodi. I sistemi P2P non sono in grado in molti casi di fornire una garanzia sulla qualità e l'affidabilità dei dati che vengono forniti da un nodo. Alcune risorse infatti possono essere fittizie e sono dovute ad attività mirate alla distruzione del sistema stesso. Infine l'elevato tasso con cui i nodi entrano/escono dal sistema P2P può portare ad una instabilità del sistema che ne influenza negativamente le prestazioni .

1.2 I vari tipi di reti peer-to-peer

Definizione 1.2(Overlay network):

Un *overlay network* è una rete di computer che si basa su un'altra rete. I nodi dell'*overlay* sono collegati tra loro da legami logici o virtuali, ciascuno dei quali corrisponde a un percorso, anche attraverso numerosi collegamenti fisici della rete sottostante.

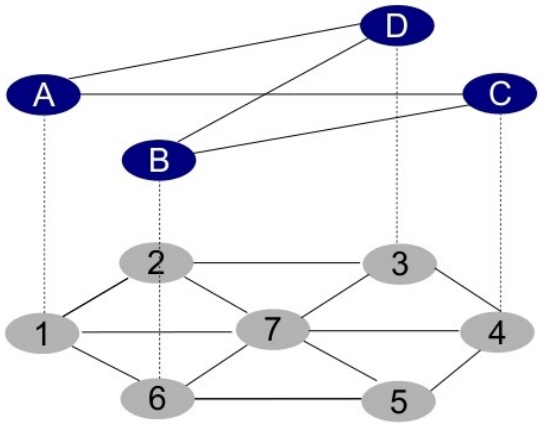


Figura 1.2: La connessione logica TCP (es. A-D) tra due nodi si risolve in un path fisico (presumibilmente) multi-hop (1-2-3)

L'*overlay network* è la rete virtuale che interconnette i peer, i collegamenti della rete rappresentano i vari canali di comunicazione a disposizione del peer. Questa rete virtuale, come è possibile notare dalla *figura* 1.2, ha il compito di realizzare dei meccanismi che permettano ai nodi di comunicare tra loro. Il problema dell'organizzazione risulta particolarmente arduo da risolvere, in quanto si deve tenere conto della totale dinamicità della rete, dovuta al fatto che generalmente ogni nodo si può inserire o rimuovere in qualunque momento.

L'*overlay network* è completamente indipendente dalla rete fisica, grazie al livello di astrazione TCP/IP può essere strutturata in modo completamente decentralizzato (*Gnutella 0.4*), gerarchico (*Gnutella Super-peers 0.6*, *JXTA rendez- vouz peers*) o può includere un server centralizzato (*look up server* di *Napster*). I peer partecipano attivamente all'overlay network :

- Fornendo informazione
- Ricercando informazione
- Svolgendo funzioni di routing

In base all'interconnessione tra i nodi dell'*overlay network* possiamo suddividere le reti P2P in *non strutturate* e *strutturate*.

Reti non strutturate

In questa tipologia i nodi sono organizzati come un grafo random e l'organizzazione della rete segue principi molto semplici. Non vi sono vincoli sul posizionamento delle risorse rispetto alla topologia del grafo e per questo motivo, la localizzazione delle risorse è resa difficoltosa dalla mancanza di organizzazione della rete. Di contro l'aggiunta o la rimozione di nodi è un'operazione semplice e poco costosa. L'obiettivo principale di questo tipo di reti è quello di gestire nodi con comportamento fortemente transiente. Alcuni esempi di software che utilizzano questo tipo di rete sono: *Gnutella*, *FastTrack*, e *Donkey*.

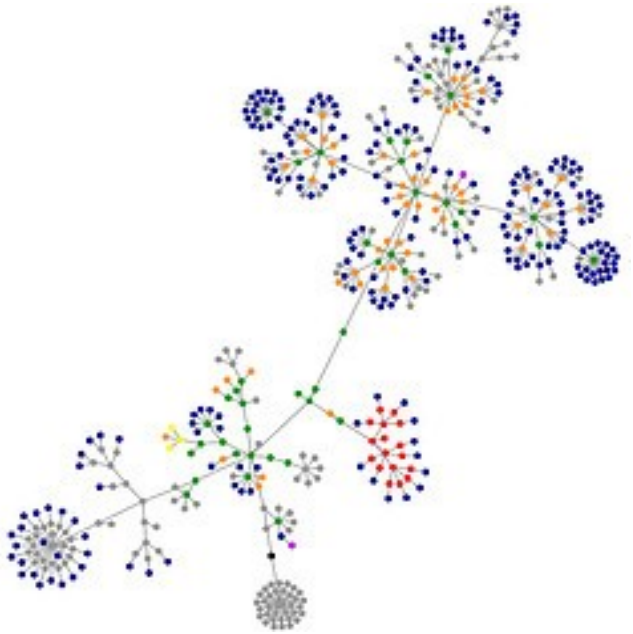


Figura 1.3: La topologia della rete Gnutella 0.6

Reti strutturate

Nelle reti strutturate ci sono dei vincoli sul grafo e sul posizionamento delle risorse sui nodi del grafo. L'organizzazione della rete segue principi rigidi e l'aggiunta o la rimozione di nodi è un'operazione costosa. L'obiettivo di questa tipologia di rete è quello di migliorare la localizzazione delle risorse. Esempi di reti strutturate li troviamo in *Chord*, *Pastry*, *CAN*.

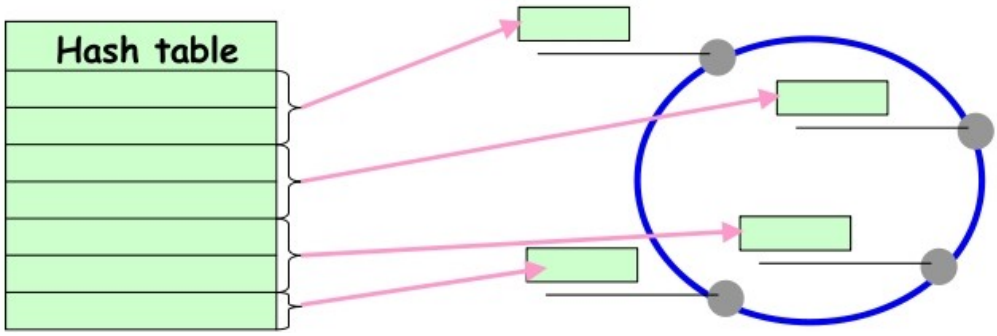


Figura 1.4: Le DHT nelle infrastrutture peer-to-peer

Le reti peer-to-peer prevedono lo scambio di diversi tipi di dati e risorse. Tipicamente le applicazioni di tali reti prevedono :

1. Scambio di informazioni.
2. Condivisione di potere computazionale.
3. Scambio istantaneo di messaggi.

1.2.1 Reti P2P per lo scambio di informazioni

Alla base di queste reti ci sono spesso motivazioni non solo meramente utilitaristiche ma anche ideologiche: la libertà di parola, la libertà di comunicazione, lo scambio di idee ed opinioni, il libero flusso di informazione. Alcuni esempi di sistemi distribuiti di scambio di dati più popolari:

1) **Napster [**1]**:** Un sistema che permette agli utenti di condividere musica in digitale. Un database centrale memorizza la posizione di tutti i file, ma i dati vengono distribuiti direttamente da un utente all'altro.

2) **Gnutella** [2]**:** Una rete completamente *serverless*, distribuita. I nodi sono trattati in modo paritario, indipendentemente dalla banda e dal numero di file condivisi. Ogni nodo si occupa sia di fornire i file che di inviare e rispondere alle richieste di *routing* degli altri nodi, compito riservato ai server in una rete centralizzata.

3) **Fast Track** [3]**:** Un sistema che utilizza il concetto di nodi e supernodi, al fine di migliorare la scalabilità della rete. A differenza di *Gnutella* non è open source. E' alla base di famose applicazioni come: *KaZaA* e *MusicCity.*

1.2.2 Reti P2P per il calcolo computazionale

In questo tipo di reti le risorse condivise a differenza delle precedenti sono risorse hardware. Le principali reti che si occupano del calcolo computazionale:

1) **SETI@home** [4]**:** E' un acronimo per *Search for Extraterrestrial Intelligence* (*Ricerca di Intelligenza Extraterrestre*). Lo scopo di SETI@home è quello di analizzare i dati provenienti dal Radiotelescopio di Arecibo, alla ricerca di eventuali prove di trasmissioni radio provenienti da intelligenza extraterrestre. Con oltre 5 milioni di utenti in tutto il mondo, il progetto è attualmente l'esempio di maggior successo di elaborazione distribuita.
2) **GIMPS** [5]**:** E' l'acronimo di *Great Internet Mersenne Prime Search*, ed è un progetto per la ricerca dei numeri primi di Mersenne, ovvero numeri primi nella forma $2^p - 1$, dove p è a sua volta un numero primo.

1.2.3 Reti P2P per lo scambio di messaggi

L'utilizzo del cosiddetto *instant messaging* ha avuto un crescente successo negli ultimi anni. Fra gli svantaggi maggiori che contraddistinguono queste reti troviamo la scarsa sicurezza (diffusione facilitata di virus e worm) e l'impossibilità di garantire la privacy. A favore la connessione fra due utenti risulta molto più robusta, scalabile e veloce, in quanto i messaggi non necessitano di un passaggio attraverso un server centrale.

1) **AIM** [6]**:** E' un programma di *instant messaging* prodotto da *AOL* che sfrutta i protocolli di messaggistica *OSCAR* e *TOC* per consentire agli utenti di comunicare in tempo reale. È stato rilasciato per la prima volta nel mese di maggio del 1997.
2) **ICQ** [7]**:** Permette di: mandare messaggi istantanei (anche a utenti offline), *SMS*, *URL*, cartoline; avviare chat multi-user e giochi online, trasferire file, ricevere email tramite *POP3* e condividere una cartella remota.

1.3 Sistemi peer-to-peer: architettura

I sistemi P2P possono essere principalmente di due tipologie *ibridi* o *puri* [8,9]. I modelli ibridi dispongono di un un server centrale che conserva informazioni sui peer e risponde a richieste su quelle informazioni, mentre nei modelli puri tutti i nodi sono peer, senza il bisogno di un coordinatore centralizzato. A queste due tipologie di sistemi P2P se ne affianca un'altra definita "parzialmente centralizzata" in cui alcuni nodi si promuovono "*Ultrapeer*" facilitando la connessione tra gli altri peer della rete.

1.3.1 Decentralizzate ibride

Questa categoria di sistemi P2P, cui appartiene anche *Napster*, utilizza un sistema centrale che dirige il traffico fra i singoli peer.
I server centrali mantengono delle directory in cui sono presenti le indicazioni sui file condivisi dagli utenti. Tali server consentono ai peer di accedere ai propri database, e forniscono le indicazioni di localizzazione delle risorse.

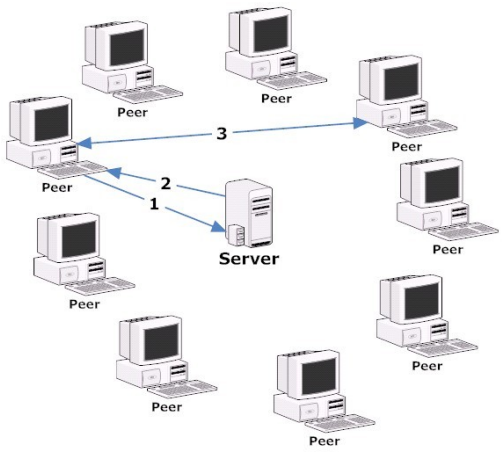


Figura 1.5: Architettura P2P Ibrida

Un esempio di come un peer riesce ad acquisire una risorsa è mostrato in *figura* 1.5: il peer contatta il *server di lookup* (1), ottiene la lista di peer che condividono la risorsa da lui cercata (2) e procede a contattare uno di questi per ottenere la risorsa (3).
I server non contengono comunque dei dati di alcun tipo, servono solamente a far comunicare i vari nodi della rete l'uno con l'altro (come era il caso di *Napster*) .
E' bene sottolineare che l'acquisizione della risorsa è effettivamente P2P, infatti come detto sopra il server contiene solo meta-informazioni.

1.3.2 Decentralizzate pure

Nei sistemi P2P puri non esiste una gerarchia che coordini le entità della rete, infatti, tutti i nodi sono dei semplici peer, con le stesse capacità di elaborazione e di *routing*.
Gnutella è il più famoso sistema P2P puro, implementa un modello di comunicazione decentralizzato, che garantisce agli utenti del sistema un elevato livello di autonomia nell'utilizzo della rete.
Come è possibile notare anche in *figura* 1.6, in questo tipo di sistemi, ogni peer è un"servent", con accesso diretto alle risorse e senza il supporto di peer intermediari.

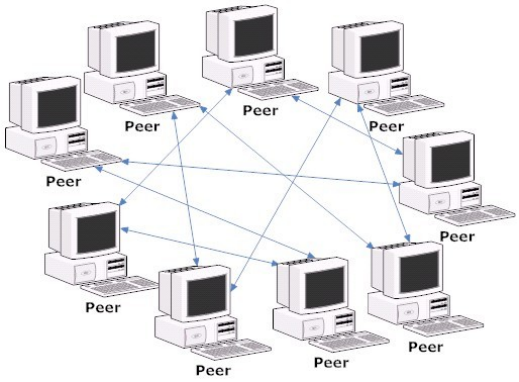


Figura 1.6: Architettura P2P Pura

A differenza dei modelli ibridi, ogni peer ha la possibilità di aprire una connessione diretta con gli altri peer, formando una *overlay network* con connessioni point-to-point fra le entità della rete. I peer che appartengono all'*overlay network*, inviano una richiesta a tutti i peer con i quali sono direttamente connessi. Queste ultimi hanno il compito di risolvere localmente le richieste. Nel caso in cui non dispongano delle informazioni necessarie per risolverla, la propagano ai propri peer limitrofi.

1.3.3 Parzialmente centralizzate

Negli ultimi tempi si sono sviluppati dei sistemi P2P parzialmente centralizzati, che colgono entrambi gli aspetti dei sistemi decentralizzati ibridi e puri. In sostanza si tratta di sistemi che hanno una struttura decentralizzata di base, all'interno del quale sono annidati svariati sistemi centralizzati.
Uno dei sistemi di file sharing che usa questo tipo di struttura è *FastTrack* utilizzato per *KaZaA*. La maggior parte dei peer sono collegati a dei supernodi cui inoltrano tutte le richieste di risorse di qualunque tipo. Ogni supernodo detto anche *Ultrapeer* dispone delle seguenti strutture dati:

- una tabella nella quale sono memorizzate le caratteristiche fisiche dei peer (esempio: indirizzo IP, velocità di connessione, ecc...) e

- una lista degli risorse condivise.

Come è possibile notare anche in *figura* 1.7, nella fase di valutazione delle richieste, e successivamente di recupero delle informazioni, ogni peer inoltra le richieste all'*Ultrapeer* (1). Dopo aver ricevuto la richiesta, l'*Ultrapeer* la confronta con le informazioni contenute nelle strutture dati d'appoggio, allo scopo di individuare la lista delle "best-peer" che soddisfano la richiesta, inoltrandola anche ad altri supernodi conosciuti (2-3). Ricevuta la lista delle "best-peer" (4), il peer sorgente inizia il processo di recupero della risorsa aprendo una connessione diretta con un altro peer ed effettuando il download di questa (5).

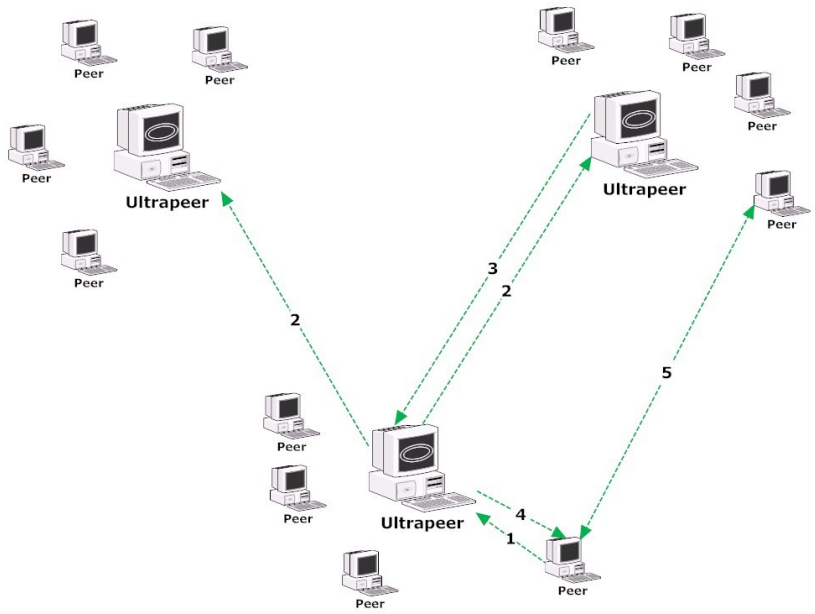


Figura 1.7: Architettura P2P Parzialmente Centralizzata

1.4 Nascita ed evoluzione dei sistemi peer-to-peer

Sistemi P2P di prima generazione.

All'inizio del 1999, Shawn Fanning, uno studente universitario della North-eastern University di Boston, diede vita ad un fenomeno chiamato *Napster.*

Napster fu il primo sistema di peer-to-peer di massa che consentiva la condivisione di file mp3. Tuttavia non era un peer-to-peer puro in quanto utilizzava un sistema di server centrali che mantenevano la lista dei sistemi connessi e dei file condivisi, mentre le transazioni vere e proprie avvenivano direttamente tra i vari utenti. La rete *Napster* vide passare 30 milioni di utenti, con oltre 800 mila nodi che effettuavano un accesso simultaneo. L'unico limite superiore sul numero di utenti di *Napster* era imposto dalla banda passante del computer centrale: da questa infatti scaturivano delle restrizioni sul numero possibile di utenti che potevano accedere al server *Napster* ed utilizzare i servizi da esso forniti.

A seguito degli attacchi legali condotti dalla *RIA* (**R**ecording **I**ndustry **A**ssociation of **A**merica, *Associazione americana dei produttori discografici*) e dalle maggiori case discografiche nel luglio 2001 *Napster Inc* fu costretta a interrompere l'attività del server centralizzato.

La rete virtuale costituita dagli utenti di *Napster* cessò di esistere, ma erano già disponibili un buon numero di nuove applicazioni con funzionalità simili a quelle di *Napster* (G*nutella*, *Kazaa/FastTrack, Freenet*). Le nuove applicazioni furono progettate in modo da evitare l'utilizzo di un server centralizzato per evitare di incorrere in nuove sanzioni legali.

Nel Marzo 2000 *NullSoft* (*Gene Khan*) presentò *Gnutella* progetto opensource.

Gnutella si dimostrò a pieno titolo un sistema scalabile, infatti la cardinalità della rete passò in soli 7 mesi da 2K a 34K nodi.

Sistemi P2P di seconda generazione.

Nell'Ottobre 2000 *Gnutella* adottò il routing gerarchico, introducendo il concetto di SuperNodo, che aumentò notevolmente la scalabilità di sistema.

Sistemi P2P di terza generazione.

Nel 2001 ci fu l'avvento delle reti strutturate, grazie all'uso di algoritmi di *routing* basati su *Distributed Hash Tables* (*DHTs*).

2 La Rete Gnutella

2.1 Il protocollo Gnutella 0.4

2.1.1 Definizione del protocollo

Il protocollo *Gnutella* [11,12] definisce un insieme di messaggi (scambiati attraverso protocollo TCP) e di regole di *routing* che permettono ai diversi *servent* di comunicare tra loro.

Sono supportati 5 tipi di messaggi:

Tipo messaggio	**Descrizione**
Ping	Messaggio per il *discovery* dei nodi vicini
Pong	Messaggio di risposta ad un Ping: un nodo che riceve un Ping risponde con uno o più Pong, includendo il proprio indirizzo
Query	Messaggio di richiesta per la localizzazione della risorsa.
QueryHits	Messaggio di risposta ad una Query: contiene le informazioni per reperire la risorsa.
Push	Messaggio che permette a servent protetti da *firewall* di contribuire ugualmente allo scambio di risorse.

Un *servent Gnutella* si collega alla rete stabilendo una connessione con un altro *servent* già connesso alla rete. Ci sono diversi modi per acquisire l'indirizzo del primo *servent* a cui connettersi, ad esempio attraverso la connessione ad un *host cache service* contenenti gli indirizzi di svariati *servent* attualmente connessi. Un'altra soluzione è quella di connettersi agli indirizzi reperiti nella lista di centinaia di *servent* comunemente inclusa in molti *client gnutella* : la probabilità che almeno uno di questo sia on-line è molto elevata.
Una volta connessi la comunicazione e lo scambio di informazioni avviene tramite messaggi che possono essere di tre tipi:

Multicast
Questi messaggi devono essere propagati attraverso l'infrastruttura di rete e vengono letti e interpretati da tutti i *servent*. Tali messaggi servono sia per effettuare delle ricerche (messaggi Query), sia per segnalare la propria presenza nella rete cercando di ottenere nel contempo una stima della sua estensione (mes-

saggi Ping). Ogni nodo è connesso ad un certo numero di altri nodi e, in presenza di questo genere di messaggi, ha il compito di propagarli verso tutti i nodi (ad eccezione del nodo da cui è arrivato il messaggio).

Unicast

I messaggi *unicast* utilizzano la rete *Gnutella* e servono per mettere in comunicazione due nodi. Tipicamente questi messaggi sono risposte a messaggi *multicast*, come ad esempio messaggi QueryHit (in risposta a Query) e messaggi Pong (in risposta a Ping). Un altro tipo di messaggio *unicast* è Push, utilizzato per mettere in comunicazione diretta nodi protetti da f*irewall*.

Diretto

I messaggi diretti tra nodi avvengono senza l'utilizzo dell'infrastruttura *Gnutella* e servono a mettere in comunicazione diretta due nodi che tipicamente devono scambiarsi dei file. Questi messaggi servono essenzialmente per la gestione e il trasferimento delle risorse.

2.1.2 Hand-shaking

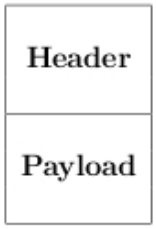
Una volta reperito un indirizzo di un *servent* è possibile stabilire una connessione TCP attraverso l'invio della seguente stringa:

GNUTELLA CONNECT/ <protocol version string>\n\n

dove <protocol version string> è la seguente stringa ASCII: "0.4".
Se il *servent* destinatario del messaggio accetta la richiesta di connessione risponde con :

GNUTELLA OK\n\n

Ogni messaggio diverso da quello sopra riportato sarà interpretato come un rifiuto della connessione. Se la fase di connessione ha dato esito positivo il *servent* può comunicare con gli altri nodi della rete attraverso l'invio/ricezione di messaggi del protocollo.
Questi messaggi sono formati da un *header*, comune a tutti i tipi di messaggi, di dimensione fissa 22 byte e dal messaggio vero e proprio chiamato *payload*.



2.1.3 Header dei messaggi

L'*header* è composto dai seguenti campi:

Descriptor ID	Payload Descriptor	TTL	Hops	Payload Length
16 byte	*1 byte*	*1 byte*	*1 byte*	*4 byte*

Descriptor ID: Una stringa che identifica il messaggio nella rete, serve ad eliminare il traffico ridondante.

Payload Descriptor: identifica la tipologia del messaggio. Attualmente sono utilizzati i seguenti valori: 0*x*00 (Ping), 0*x*01 (Pong), 0*x*40 (Push), 0*x*80 (Query), 0*x*81(QueryHit).

TTL: Time To Live. Ogni servent decrementa questo campo di 1 prima di passarlo ad un altro nodo della rete. Quando il TTL raggiunge 0 il messaggio non viene più inoltrato. Il campo TTL rappresenta l'unico meccanismo in grado di far scomparire i messaggi dalla rete, eventuali incrementi sconsiderati possono portare ad un aumento del traffico e una ridotta efficienza della rete.

Hops: Indica il numero di volte in cui è stato inoltrato il messaggio. Ogni servent aumenta di 1 l'Hops. La relazione che deve essere sempre valida fra il TTL e l'Hops è la seguente:

$$\mathbf{TTL(0) = TTL(i) + Hops(i)}$$

In cui TTL(i) e Hops(i) indicano i rispettivi valori dei due campi all'i-esimo passaggio del messaggio. Questa relazione viene rispettata poiché ad ogni passaggio del messaggio i due campi vengono aggiornati dal nodo che riceve il messaggio stesso.
PayloadLength: Rappresenta la lunghezza del messaggio nel pacchetto. La dimensione massima in teoria è di 2^{32} byte (4GB), ma generalmente viene fissato a 64 Kb.

2.1.4 I messaggi

2.1.4.1 Ping

I messaggi Ping non hanno associato alcun carico e hanno lunghezza 0. Un Ping è quindi rappresentato soltanto da un *Desciptor Header* dove sia il campo *Payload Descriptor* che il campo *Payload Length*

sono fissati a 0.

Un *servent* che riceve un messaggio Ping potrebbe rispondere con un messaggio Pong, indicando la sua presenza e la quantità di risorse condivise.

Non c'è alcuna specifica atta a descrivere le modalità e i tempi di utilizzo di questa tipologia di messaggi, ma ogni nodo non dovrebbe eccedere nell'invio di questi, in quanto la maggior parte dei pacchetti che transitano su una rete *Gnutella* sono di questa tipologia

2.1.4.2 Pong

Port	Ip Address	File Shared	Num Of KB Shared
2 byte	*4 byte*	*4 byte*	*4 byte*

Viene inviato solo in risposta ad un messaggio Ping, ed è caratterizzato dai seguenti campi:

- **Port:** La porta utilizzata dal *servent* per rispondere alle domande di connessione in entrata.
- **Ip Address:** L'indirizzo ip dell'host che risponde, rappresentato in formato *big endian*.
- **File Shared:** Numero di file condivisi dal *servent*.
- **Num Of KB Shared**: Dimensione totale dei file espressi in KB.

2.1.4.3 Query

Minimum Speed	Search Criteria	Null Terminator
1 byte	*N byte*	*1 byte*

I messaggi Query servono ad effettuare delle ricerche nella rete. Sono messaggi *multicast* ed ogni nodo che li riceve è tenuto a confrontare i criteri di ricerca con il proprio insieme di risorse condivise per verificare la presenza di file che risponderebbero alle esigenze descritte. Così come accadeva per il Ping, il messaggio Query viene inoltrato su tutte le connessioni eccetto quella di provenienza.

Questi messaggi sono caratterizzati dai seguenti campi:

- **Minimun Speed**: La velocità minima, espressa in Kb/s, definisce il limite inferiore di velocità richiesta ai servent che rispondono.
- **Search Criteria**: Stringa ASCII-compatibile che contiene le keywords di ricerca separate dal carattere di spazio.
- **Null terminator**: terminatore espresso dal carattere *Null*.

2.1.4.4 QueryHits

Num Of Hits	Port	Ip	Speed	Result Set	Server ID
1 byte	*2 byte*	*4 byte*	*4 byte*	*N byte*	*16 byte*

Viene inviato in risposta ad un Query se il *servent* trova un match fra l'insieme dei suoi dati condivisi in rete e la stringa di richiesta contenuta nel Query analizzato.
Alla ricezione del messaggio l'host può avviare il download tramite una richiesta GET del protocollo HTTP.
Un messaggio QueryHits dovrebbe avere campo Hops uguale a 0 e campo TTL uguale al Hops del messaggio Query corrispondente.
Questa tipologia di messaggi è caratterizzata dai seguenti campi:

- **Num Of Hits**:Numero di record nel campo risultati.
- **Port**: Porta su cui il *servent* accetta connessioni in entrata.
- **Ip**: Indirizzo Ip del *servent* in risposta.
- **Speed**: Velocità espressa in Kb/s.
- **Result Set**: Contiene gli identificatori delle risorse che soddisfano il messaggio di Query.
- **Server ID**: Identifica univocamente il *servent* della rete.

2.1.4.5 Push

Servent ID	File Index	Ip Address	Port
16 byte	*4 byte*	*4 byte*	*2 byte*

Il messaggio Push viene inviato se il *servent* riceve un QueryHits da un nodo che non supporta connessioni, tipico esempio di un *servent* dietro un *firewall*. Viene quindi invitato il mittente ad avviare la connessione e inviare la risorsa tramite protocollo HTTP.
I messaggi di Push sono così composti:

- **Servent ID**: Specifica il servent destinatario del messaggio.
- **File Index**: L'indice della risorsa che l'host remoto deve inviare.
- **Ip Address**: Ip dell'host mittente del messaggio su cui avviare la connessione.
- **Port:** Porta dell'host mittente del messaggio su cui avviare la connessione.

2.1.5 Meccanismo di routing

Il meccanismo di *routing* di *Gnutella* si basa su poche e semplici regole:

1. I messaggi Pong devono essere smistati solo attraverso lo stesso percorso del messaggio Ping corrispondente.
2. I messaggi QueryHits devono essere smistati solo attraverso lo stesso percorso del messaggio Query corrispondente.
3. I messaggi Push devono essere smistati solo attraverso lo stesso percorso del messaggio QueryHits corrispondente.
4. Un servent dovrebbe rinviare una richiesta di Ping o di Query a tutti i nodi a cui è collegato, tranne quello mittente.
5. Prima di ritrasmettere un messaggio pervenuto il *servent* deve modificare i campi TTL e Hops. Se TTL risulta essere 0 il messaggio non deve essere più ritrasmesso.
6. Un servent a cui perviene un messaggio con un ID incontrato precedentemente, dovrebbe scartare il messaggio in modo da non generare traffico ridondante nella rete.

2.1.6 Download dei file

Alla ricezione di un QueryHits un *servent* può decidere di scaricare la risorsa stabilendo una connessione diretta al *servent* relativo. Il trasferimento dei file avviene tramite il protocollo HTTP 1.0.

GET /get/<indice del file>/<nome del file>/ HTTP/1.0
Connection: Keep-Alive\r\n
Range: bytes=0-\n\r
User-Agent: Gnutella\r\n
\r\n

La risposta da parte del *servent* nel caso in cui disponga della risorsa è la seguente:

HTTP 200 OK\r\n
Server: Gnutella\r\n
Content-type: application/binary\r\n
Content-length: 5123

2.2 Vantaggi e svantaggi del protocollo Gnutella 0.4

La rete *Gnutella* è una rete P2P completamente decentralizzata, in quanto tale gode (in linea di massima) degli stessi vantaggi/svantaggi propri a questa tipologia di rete.

Vantaggi

Essendo una rete decentralizzata non è organizzata da nessun server centrale pertanto non c'è la possibilità di avere eventuali colli di bottiglia (almeno in linea teorica) e il sistema gode di un'alta *fault-tolerance* grazie all'esplorazione di molti percorsi. Inoltre la decentralizzazione dell'organizzazione della rete elimina la dipendenza da un server cache per l'accesso alla rete e l'impossibilità di essere suscettibile a possibili attacchi di tipo *DoS*.
Un ulteriore aspetto positivo dal punto di vista legale è la non perseguibilità penale sia per l'assenza di un server centrale, sia per l'aspetto dei file scambiati che possono essere confusi con qualsiasi altro file scambiato nel web.
Inoltre non è limitata allo scambio una sola tipologia di risorsa (vedi *Napster* che supportava solo lo scambio di file *.mp3*) e permette la connessione ai *servent* che sono dietro un *firewall*.

Svantaggi

I principali difetti che si possono evincere dalla struttura di *Gnutella* riguardano particolarmente il *flooding* generato nella rete. Infatti l'invio in di alcune tipologie di messaggi comporta un carico eccessivo per la rete, si pensi che per un TTL = 7 e C = 4 (C numero di vicini) ogni messaggio Query/Ping genera circa 22K messaggi. Nonostante il numero di messaggi generato ogni peer ha una "vista limitata": il TTL non permette ai messaggi di raggiungere i nodi della rete che sono localizzati al passo successivo, oltre il quale il TTL scade. Inoltre la topologia è incontrollabile e imprevedibile in quanto è arduo stimare il tempo di esecuzione del messaggio Query o la probabilità di successo di questo. Un sistema che integra funzionalità basate sul riconoscimento utente (crediti utente) non è realizzabile sul protocollo *Gnutella*, causa l'impossibilità del riconoscimento dei *servent* (il proprio identificativo può variare da una sessione all'altra di connessione). L'uguaglianza gerarchica dei *servent*, genera degli sprechi di risorse elaborative e di rete, infatti ogni nodo della rete ha proprie caratteristiche prestazioni che in questo modello non vengono sfruttate in modo soddisfacente. E' possibile riscontrare nel modello *Gnutella* anche dei problemi di natura "sociale", uno di questi è il fenomeno Free-Riders, dove il 70% dei peer non condivide risorse e il 50% delle risposte/download è fornito dall'1% dei peer, così da avere un concentrazione del carico di lavoro su pochi peer con un peggioramento delle prestazioni del sistema ed un aumento della vulnerabilità (eventuali crash dei peer "importanti").

2.3 La versione 0.6

Nella versione di *Gnutella 0.4* non c'erano gerarchie tra i peer della rete, tuttavia questo approccio limitava la scalabilità del sistema come è stato evidenziato dalle problematiche trattate nel paragrafo precedente.
La versione *Gnutella 0.6* [13] si fa carico di queste problematiche cercando di risolverle attraverso la strutturazione della rete in due 2 livelli gerarchici.
L'idea di base consiste nel dividere in due grandi tipologie i nodi che fanno parte della rete: *foglie* e *Ultrapeers*. I nodi con buone prestazioni di rete e computazionali si occupano della maggior parte del carico (*Ultrapeer*) rispetto ai nodi con prestazioni più modeste (*foglie*).
Questa suddivisione avviene all'interno della rete in modo completamente dinamico e automatico.
Le *foglie* mantengono solo le connessioni agli *Ultrapeer*, mentre questi oltre alle connessioni instaurate con i propri *nodi foglia* devono mantenere alcune connessioni con altri *Ultrapeer*.
Il *flooding* nella rete è generato esclusivamente dai Supernodi, in modo da abbassare il carico di rete generato dai messaggi di tipo Query/Ping.
I nodi foglia inviano periodicamente le loro *tavole hash* agli Ultrapeer, i quali a loro volta, inoltreranno le Query verso quei nodi la cui *tavola hash* ha un campo corrispondente alla richiesta ricevuta.
Una foglia può eleggersi Ultrapeer solo se soddisfa una serie di requisiti minimi (banda passate, non devono risiedere dietro un *firewall* etc..) alcuni inerenti anche al sistema operativo usato.

3 Algoritmi epidemici

Gli algoritmi epidemici hanno recentemente guadagnato popolarità, poiché sono una valida soluzione per la diffusione di informazioni nei sistemi distribuiti di grandi dimensioni.
In aggiunta alla loro intrinseca scalabilità, sono facili da implementare, robusti e resistenti alla *faliure*.
Il comportamento degli algoritmi epidemici è simile alla diffusione di una malattia contagiosa. Allo stesso modo in cui un individuo infetto trasmette un virus ad un insieme di persone con cui è entrato in contatto, così ogni nodo del sistema invia nuove informazioni ricevute ad un numero casuale di coetanei.

3.1 Caratteristiche generali dei protocolli epidemici

L'obiettivo principale dei protocolli epidemici è la rapida propagazione delle informazioni in un grande insieme di nodi, usando solo informazioni locali, senza l'intervento di un'entità di coordinazione.
In un algoritmo epidemico [14] tutti i nodi del sistema sono potenzialmente coinvolti nella diffusione delle informazioni. Fondamentale ogni nodo registra il messaggio che riceve in un buffer di capacità b e lo inoltra un numero di volte t. I nodi inoltrano i messaggi ogni volta ad un insieme casuale di loro vicini di cardinalità f (*fan-out* della disseminazione). Ci sono molte varietà di algoritmi epidemici, si distinguono per il diverso valore dei parametri b, t e f, che possono essere fissati indipendentemente dal numero di nodi del sistema. A differenza di altre tipologie di algoritmi, in cui i nodi in presenza di errori di trasmissione sono costretti alla ritrasmissione del messaggio, negli algoritmi epidemici ciò non è necessario non c'è nessun meccanismo per rilevare faliure.
Nello sviluppo di un algoritmo epidemico ci si sofferma in particolar modo sui seguenti aspetti:

- **Membership**: Come un nodo del sistema acquisisce informazioni sull'esistenza di altri nodi (i sui vicini di *overlay*), e quanti nodi ha bisogno di "conoscere".
- **Network Awareness**: Come la connettività tra i vari nodi riflette l'attuale topologia di rete, allo scopo di garantire performance accettabili.
- **Buffer Manager**: Come vengono gestite le informazioni destinate ad un nodo quando il suo buffer è pieno.
- **Message Filterring**: Come tener conto del reale interesse di ogni nodo, decrementando la probabilità di ricezione e memorizzazione di informazioni di interesse nullo per il nodo in questione.

3.1.1 Membership

La *membership* è una questione fondamentale per l'implementazione di algoritmi epidemici, infatti in una disseminazione epidemica di informazioni ogni nodo *p* che riceve un messaggio può inoltrarlo solo ad altri nodi che conosce. Come avviene l'acquisizione delle sue informazioni di *membership* ha un alto impatto sulle performance delle successive disseminazioni e sulla scalabilità del sistema.
Per esempio nei primi algoritmi epidemici *broadcast* si assumeva che ogni nodo conoscesse tutti gli altri nodi del sistema e potesse comunicare direttamente con loro. Questa assunzione è realistica se consideriamo un sistema con un ristretto numero di nodi, ma in presenza di sistemi con un elevato numero di nodi questa soluzione diventa impraticabile poiché lo spazio richiesto per la memorizzazione di informazioni di *membership* cresce linearmente con la grandezza del sistema. Infatti per garantire la consistenza delle viste di ogni nodo bisogna generare un carico extra nella rete, in particolar modo in ambiente molto dinamici.
Per garantire la scalabilità del sistema, bisogna restringere la vista di ogni nodo migrando ad una vista parziale, ossia un sottoinsieme di tutti i nodi appartenenti al sistema. Un algoritmo epidemico deve quindi trovare il giusto equilibrio tra scalabilità ed affidabilità, ossia tra vista parziale e vista totale del sistema.
Una possibile soluzione a questo spinoso problema consiste nell'integrare la *membership* con la diffusione epidemica, ossia quando un nodo trasmette un messaggio ingloba in questo un insieme di nodi che fanno parte della propria vista. Così il nodo destinatario del suddetto messaggio è in grado di aggiornare la sua lista *membership*. Utilizzando questo approccio non si ha un aumento significativo della dimensione dei messaggi in quanto le informazioni di *membership* sono una semplice lista di identificatori di nodi del sistema.
Un approccio simile si basa sullo scambio periodico di messaggi, contenenti identificatori di nodi e *time-stamp*, tra i nodi vicini; così da poter inserire nella propria lista *membership* solo gli identificatori meno datati.

3.1.2 Network Awareness

Gli algoritmi di *membership* non considerano la topologia di rete, considerano tutti i nodi del sistema ugualmente raggiungibile, questa assunzione può generare un elevato carico sulla rete.
La maggior parte delle soluzioni proposte per affrontare questo problema si basano sull'organizzazione gerarchica dei nodi, in modo da poter rispecchiare in parte la topologia fisica della rete. L'organizzazione gerarchica favorisce lo scambio di messaggi tra i nodi dello stesso ramo della gerarchia, limitando cosi' il carico sul nucleo della rete.

3.1.3 Buffer Management

Prendendo in esame un semplice *algoritmo epidemico broadcast*, è possibile notare che a seconda del tasso di trasmissione, il buffer di un nodo può essere di dimensioni insufficiente per memorizzare ogni messaggio a lui pervenuto.
Ci sono principalmente due approcci che tentano di risolvere anche parzialmente il problema: il primo ottimizzando l'uso della memoria, il secondo riducendo il flusso di informazioni.

Ottimizzazione dell'uso della memoria: per favorire un'oculata gestione della memoria, è possibile associare ad ogni messaggio una priorità. Le priorità associate ai messaggi possono essere definite in molteplici modi, il più usato l'*Age-based* si basa sull'idea di "marchiare" un messaggio con la propria età prima dell'inoltro. L'età di un messaggio è spesso equivalente al suo numero di inoltri (Hops).
Se il buffer dei messaggi di un nodo risulta pieno, si procederà alla cancellazione del messaggio con età più elevata. In determinate condizioni questa tecnica riduce l'utilizzo delle risorse pur preservando l'affidabilità.

Riduzione del flusso di informazioni: un altro metodo per garantire la scalabilità delle risorse mantenendo un alto grado di affidabilità è ridurre il flusso di informazioni che il nodo genera.
Una soluzione richiede che ogni nodo calcoli la dimensione media del buffer dei nodi che comunicano con lui, e la trasmetta a questi. Quando la frequenza di invio dei messaggi aumenta rispetto a quella media, il nodo riduce il tasso di trasferimento localmente. Lo svantaggio di questo metodo consiste nella variazione della frequenza di invio dovuta ai nodi che hanno un buffer di ridotte dimensioni.

3.1.4 Message Filterring

Garantire che ogni messaggio raggiunga ogni nodo del sistema è vantaggioso, quando tutti i nodi sono ugualmente interessati a ricevere ogni messaggio. Tuttavia, diversi gruppi di nodi possono avere interessi distinti. In questo scenario, potrebbe essere più accattivante dividere i nodi in gruppi accomunati dagli stessi interessi, così da diffondere i messaggi all'interno di ciascun gruppo. Un approccio alternativo consiste nel permettere ai nodi di scegliere all'interno del sistema i loro specifici interessi e aumentare la probabilità di ricezione per quelle determinate tipologie di messaggi. Tuttavia l'implementazione di un adeguato meccanismo di *message filtering* presenta diverse problematiche, infatti è difficile implementare un modello che permetta ad un nodo di sapere l'insieme di messaggi che interessano ai propri vicini. Fornire questa informazione in un sistema decentralizzato non è una cosa banale, considerando che ancora non è chiara la modalità di integrazione di tali informazioni attraverso la diffusione epidemica. Inoltre anche quando un nodo p sa che un determinato messaggio non è di interesse per un altro nodo q, la decisione di

non fornirglielo potrebbe essere errata. In quanto il percorso passante per *q* potrebbe essere cruciale per il raggiungimento di altri nodi interessati a quella tipologia di messaggi.

3.2 Modelli di propagazione delle informazioni

L'idea base su cui si fondano la maggior parte dei modelli epidemici di propagazione delle informazioni si basa sulla diffusione di informazioni su nodi (detti *infetti*) che provvedono a loro volta a diffondere le informazioni su altri nodi cui sono collegati. I nodi che ancora non hanno ricevuto i dati sono detti *suscettibili*, mentre quelli che hanno già ricevuto dati o non sono in grado di diffonderli sono detti *rimossi*.
I due principali modelli di propagazione [15, 16] delle informazioni sono:

- **Modello anti-entropia:** Ciascuna nodo sceglie a caso un altro nodo, inizia la fase di scambio delle informazioni detta "aggiornamento", al termine della quale si ha uno stato identico su entrambi i nodi.
- **Modello di diffusione del rumore** o semplicemente **gossiping**: Un nodo che è stato appena aggiornato (ossia infettato) contatta un certo numero di nodi scelti casualmente inviandogli il proprio aggiornamento (infettandoli a sua volta).

3.2.1 Modello anti-entropia

In questo modello un nodo p prende a caso un nodo *q* e successivamente scambia gli aggiornamenti con questo. Per realizzare ciò si sono tre approcci:

1. *p* manda (*push)* soltanto i suoi aggiornamenti a *q*.
2. *p* prende (*pull*) soltanto i nuovi aggiornamenti da *q*.
3. *p* e *q* si mandano reciprocamente gli aggiornamenti (*push* - *pull*).

Quando è necessario diffondere rapidamente gli aggiornamenti, solo l'approccio *pull* non risulta una buona scelta. Si può notare che un approccio *pull* puro propaga gli aggiornamenti solo ai nodi infetti, ma la probabilità che uno di questi sia un nodo suscettibile è relativamente bassa. Di conseguenza c'è la possibilità, che determinati nodi rimangano suscettibili a lungo solo perché non contattati da un nodo infetto.
L'approccio *push* diversamente dal *pull* lavora in modo soddisfacente in presenza di molti nodi infetti, poiché la diffusione degli aggiornamenti è essenzialmente scatenata dai nodi sensibili. La probabilità che i nodi suscettibili contattino un nodo infetto per prendere gli aggiornamenti e infettarsi a loro volta è molto alta.
Si può notare che in presenza di un solo nodo infetto, gli aggiornamenti si propaghino rapidamente su tut-

ti i nodi usando entrambi gli approcci di *anti-entropia*, anche se la strategia *push–pull* resta la migliore. Nell'approccio *push–pull* si definisce un *round,* come intervallo di tempo in cui il nodo si è scambiato almeno una volta i propri aggiornamenti con un altro nodo scelto a caso. E' possibile dimostrare che il numero di round necessario per propagare un singolo aggiornamento a tutti i nodi, è pari a O(log(N)) dove N è il numero di nodi del sistema. Dopo questa considerazione è facile notare che la propagazione degli aggiornamenti è veloce e soprattutto scalabile.

3.2.2 Modello di diffusione del rumore

Per spiegare il funzionamento del *gossiping* consideriamo un nodo *p* appena aggiornato, che contatta un certo numero di nodi scelti a caso. Se un nodo con cui entra in contatto ha già ricevuto l'aggiornamento (ossia è già infetto), allora *p* smette con probabilità 1/k di contattare altri nodi. Il *gossiping* si rivela un ottimo metodo per la rapida diffusione delle notizie, ma non può garantire che tutti i nodi vengano effettivamente raggiunti. Si può dimostrare che quando il numero di nodi è elevato, la frazione di nodi *s* che rimangono suscettibili soddisfa l'equazione:

$$s = e^{-(k+1)(1-s)}$$

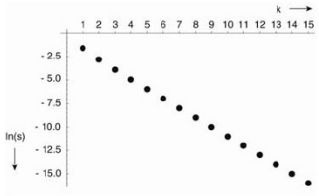


Figura 3.1

La *figura* 3.1 mostra *ln(s)* come funzione di *k*. Ad esempio se *k* = 4 abbiamo *ln(s)* = -4,97, con *s* inferiore a 0,007, il che significa che meno dello 0,7% dei nodi rimane suscettibile. Per garantire una copertura totale sono necessarie misure speciali, ad esempio combinando il *gossipping* con il modello di *anti-entropia*.

Un evoluzione del *gossiping* è rappresentato dal *gossiping direzionale*. Per i sistemi globali, Lin e Marzullo (1999) hanno dimostrato che per ottenere risultati migliori è necessario lavorare sulla topologia di rete. Nel loro approccio, i nodi, che sono connessi ad un basso numero di loro pari, hanno una probabilità relativamente alta di essere contattati. La supposizione su cui si basa tale approccio, definisce questi nodi come un "ponte" verso altre parti remote della rete, quindi dovrebbero essere contattati il più presto possibile.

3.3 Eliminazione dei dati

Gli algoritmi epidemici sono estremamente efficaci per la diffusione degli aggiornamenti. Tuttavia, hanno come effetto collaterale la difficoltà nel diffondere l'eliminazione di un dato. In sostanza questo problema è dovuto al fatto che l'eliminazione di un dato cancella tutte le informazioni su quell'elemento. Di conseguenza quando un dato viene cancellato da un nodo, quest'ultimo potrebbe ricevere delle copie vecchie di quel dato e interpretarle come aggiornamenti. Una soluzione è memorizzare la cancellazione del dato come un aggiornamento. In questo modo le vecchie copie non verranno interpretate come un aggiornamento, ma saranno semplicemente scartate. La memorizzazione dell'eliminazione avviene attraverso la diffusione di *certificati di morte*.

Per evitare che ogni nodo si costruisca gradatamente un'ampia base locale di dati contenenti informazioni storiche sui dati cancellati, è necessario cancellare dopo un certo tempo anche i *certificati di morte*. Una soluzione a questo nuovo problema è rappresentata dai *certificati di morte inattivi*. Al momento della creazione ogni certificato di morte viene contrassegnato con un *timestamp*. Si valuta il tempo necessario a propagare un aggiornamento, trascorso questo tempo i *certificati di morte* vengono eliminati. Tuttavia per garantire che le cancellazioni siano propagate a tutti i nodi, soltanto alcuni nodi mantengono *certificati di morte inattivi* che non vengono mai eliminati.

3.4 Lightweight Probabilistic Broadcast

Il *lightweight probabilistic broadcast* [17] è un protocollo completamente decentralizzato, in quanto non viene mai richiesto ad ogni nodo che lo esegue una conoscenza completa del sistema. *Lpbcast* ricorre ad un approccio probabilistico nel trattare la *membership*: ogni nodo ha infatti una vista parziale del sistema, scelta in modo casuale e di lunghezza *l*, così come il *fan-out F*. Entrambi i valori sono fissati per ogni nodo del sistema in particolare si ha che: $F \leq l$, ossia l'ammontare della conoscenza *l* che ogni nodo acquisisce, non impatta nelle performance del protocollo, che risultano influenzate solo dal *fan-out F*. Ciò non significa che il valore di *l* non abbia alcun impatto per quello che riguarda le prestazioni dell'algoritmo, infatti al decrescere di *l* aumentano le probabilità di riscontrare un partizionamento del sistema. Il protocollo *lpbcast* risulta *lightweight*, poiché utilizza una quantità limitata di risorse di memoria e non richiede messaggi dedicati per la gestione della *membership*, infatti i messaggi di *gossip* vengono utilizzati non solo per diffondere le notifiche degli eventi, ma anche per propagare informazioni relative al sistema stesso e ai suoi membri.

3.4.1 Messaggi di gossip e strutture dati

L'algoritmo *lpbcast* è basato sull'invio periodico di messaggi di *gossip*, attraverso periodi non sincronizzati. Un messaggio di *gossip* contiene diversi tipi di informazioni e viene utilizzato per vari scopi:

- **Event Notification.** un messaggio di gossip trasporta le informazioni relative a tutti gli eventi ricevuti per la prima volta dall'ultimo messaggio inviato. Ogni nodo conserva le notifiche degli eventi ricevuti nel buffer di memoria chiamato *events*. Tutti gli eventi vengono inviati in *gossip* almeno una volta ed inoltre le informazioni relative ad eventi più vecchi vengono conservate in un buffer separato, poiché verranno rinviate solo in caso di richieste di ritrasmissione.
- **Event notification identifiers.** Ogni messaggio trasporta inoltre un *digest*, una storia, di tutte le notifiche di eventi che il nodo sorgente ha ricevuto. Alla fine ogni processo immagazzina e conserva gli identificatori di tutte le notifiche ricevute in un buffer, chiamato *eventIds*.
- **Subscription.** Ogni messaggio contiene un insieme di informazioni legate ai nodi che hanno effettuato sottoscrizioni. Queste informazioni sono memorizzate in un buffer chiamato *subs* e vengono utilizzate per aggiornare la propria vista, conservata in un altro buffer *view*.
- **Unsubscription.** I messaggi di *gossip* inoltre trasportano un insieme di informazioni sui nodi che hanno abbandonato il sistema, queste informazioni vengono utilizzate per la rimozione graduale di tali nodi dalle viste. Le informazioni di *Unsubscription* che possono essere inviate con il pros-

simo messaggio di *gossip* sono memorizzate nel buffer *unSub*.

In conclusione il protocollo lpbcast utilizza le seguenti strutture dati: *events, eventsIds, unSubs, subs, view.* Dove *events*, *eventIds*, *unSubs* e *subs* vengono inserite nel messaggio di *gossip* inviato dal nodo, mentre *view* rimane una struttura locale del nodo stesso. Nessuna di queste strutture contiene duplicati e il tentativo di aggiungere elementi già presenti le lascia inalterate. Ogni struttura/lista ha inoltre una lunghezza massima prefissata indicata con $|L|_m$, unica eccezione la lista *view* che ha lunghezza massima indicata con *l*.

3.4.2 Procedure

L'algoritmo consta principalmente di tre procedure: la prima è eseguita alla ricezione di un messaggio di *gossip*; la seconda è ripetuta periodicamente nel tentativo di propagare informazioni agli altri nodi nodi; la terza è utilizzata per la richiesta di *event notification* che sono andate perdute o non sono state ricevute da lungo tempo.

3.4.2.1 Ricezione del messaggio di gossip

Ricevuto un messaggio di *gossip* si procede con il trattamento dei dati in esso contenuti, questa procedura è divisa in diverse fasi:

1. La **prima fase**, come è possibile notare in *figura* 3.2 (*a*), consiste nel trattare i dati relativi alle *Unsubscription*, ogni nodo presente in tale lista viene rimosso dalla vista individuale e aggiunto ad *Unsub*.
2. La **seconda fase**, come è possibile notare in *figura* 3.2 (*b*), consiste nel tentare di aggiungere alla vista locale le sottoscrizioni non ancora contenute in essa. Vengono aggiornati, in questa fase i due buffer *subs* e *view* con il contenuto della *subs* presente nel messaggio di *gossip*. Inoltre il nodo che invia il messaggio di *gossip* aggiunge anche il proprio *Id* alla lista s*ubs* del messaggio.
3. La **terza fase** si occupa dell'elaborazione degli *event notifications* che sono stati ricevuti per la prima volta con l'ultimo messaggio di *gossip*. Sono impediti invi multipli dello stesso evento in un messaggio, grazie alla memorizzazione in *eventIds* di tutti gli identificatori di eventi. Alla ricezione di un messaggio di *gossip*, contenente un *Id* di un evento non consegnato: un elemento composto dall'*Id* dell'evento, seguito dal numero del round e dall'identificatore del nodo che ha generato l'evento stesso viene inserito in un ulteriore buffer chiamato *retrieveBuf*, utilizzato per poter ottenere più tardi l'evento andato perduto.

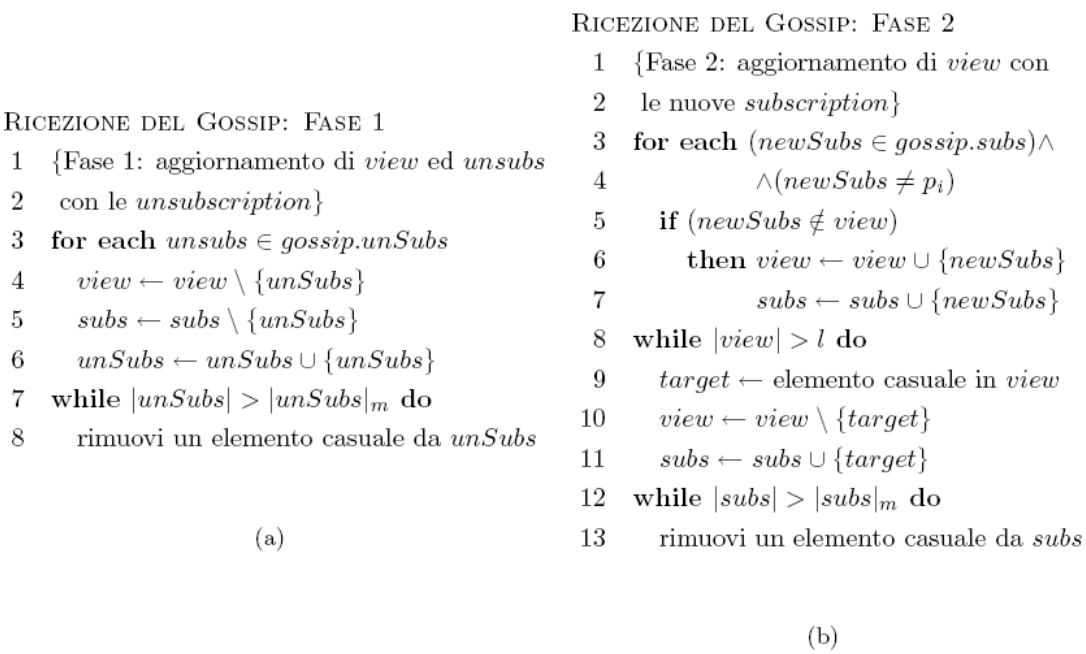


Figura 3.2: Fasi relative al trattamento dei dati dopo la ricezione di un messaggio di gossip

3.4.2.2 Invio del messaggio di gossip

Ogni nodo periodicamente, ogni *Tms*, genera un messaggio di gossip, che invia a *F* altri nodi scelti casualmente all'interno della sua vista locale *view*. Se il nodo non ha ricevuto nessuna nuova notifica dall'ultimo messaggio inviato, genererà comunque il nuovo gossip che verrà utilizzato soltanto per scambiarsi digest e mantenere le viste uniformemente distribuite.

3.4.2.3 Recupero di event notification

Il *retriveBuf* è processato per recuperare le informazioni andate perdute. Per ogni elemento del *retrieveBuf* viene eseguito un test per determinare se il nodo n_i ha atteso abbastanza (*k rounds*) prima di iniziare la procedura per il recupero dell'evento non consegnato. Inoltre viene eseguita un ulteriore test tramite *eventIds* per determinare se l'evento non sia stato consegnato con un messaggio di *gossip* successivo e ricevuto durante l'attesa. Se entrambi i test buffer falliscono allora il nodo n_i inizierà il recupero contattando il nodo che gli ha fornito la notizia dell'evento stesso. Se non si ottiene l'*event notification* in tal modo allora si sceglierà un processo in *view* e si invierà la richiesta a quest'ultimo. In caso di ulteriore fallimento si invierà la richiesta dell'evento direttamente alla sorgente che lo ha generato. La fase di recupero è basata sull'assunzione che alla ricezione di messaggio i dati contenuti in esso vengano mantenuti dal nodo che lo ha ricevuto, per un tempo limitato ma sufficiente lungo da poter attivare con successo tutta la procedura.

4 Il protocollo di rete 2Rounds

Inizialmente non era preventivato lo sviluppo di un protocollo di rete che supportasse l'algoritmo epidemico *2Rounds*. L'idea originale era di usare il protocollo *Gnutella* 0.4 con le dovute parametrizzazioni, ma durante il suo studio ho constatato che non mi avrebbe permesso di sfruttare le caratteristiche altamente dinamiche e scalabili degli algoritmi epidemici. Da queste constatazione è nata l'idea di creare un protocollo di rete che non limitasse le potenzialità dell'algoritmo *2Rounds*. Non posso affermare che il protocollo di rete da me creato sia completo, difetta di campi nei messaggi Query e QueryHits necessari alle opzioni di ricerca e non definisce una procedura per lo scambio di risorse. Anche se in minima parte non completo, il protocollo si presta ad esaltare e supportare tutte le funzionalità dell'algoritmo *2Rounds* che saranno presentate nel seguito della mia trattazione.

4.1 Il protocollo di rete 2Rounds 0.1

Procederò alla presentazione del protocollo di rete *2Rounds* 1.0 per poi illustrarne la versione 2.0 tramite il confronto con la prima.

4.1.1 Definizione del protocollo

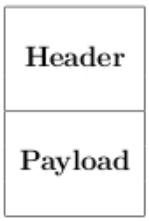
Il protocollo *2Rounds* definisce un insieme di messaggi (scambiati attraverso protocollo TCP) e di regole di *routin*g che permettono ai diversi peer di comunicare tra loro.

Sono supportati i seguenti tipi di messaggi:

Tipo messaggio	**Descrizione**
BootstrapPing1	Messaggio per il *discovery* dei nodi nella fase di *boostrap*.
BootstrapPong1	Messaggio di risposta ad un BootstrapPing1: un nodo che riceve un BootstrapPing1 può risponde con uno BootstrapPong1, includendo il proprio indirizzo.
Bye	Messaggio per comunicare l'uscita dal sistema del nodo.
Informations1	Messaggio con informazioni di *membership*.
Query	Messaggio di richiesta per la localizzazione della risorsa.
QueryHits	Messaggio di risposta ad una Query: contiene le informazioni per reperire la risorsa.

Come è possibile notare, alcuni messaggi sono simili a quelli del protocollo *Gnutella*. In particolare i messaggi di Query e QueryHits, sono utilizzati per il *discovery* delle risorse come lo erano appunto i loro antenati. A differenza dei messaggi di Ping e Pong di *Gnutella* il BootstrapPing1 e il BoootstrapPong1 sono usati solo in fase di *boostrap* per prevenire l'eccessivo *flooding* nella rete. Il messaggio Bye è usato come "certificato di morte del nodo", mentre l' Informations1 per disseminare le informazioni di *membership* nella rete.

I messaggi sono formati da un *header*, comune a tutti i tipi di messaggi, di dimensione fissa 22byte e dal messaggio vero e proprio chiamato *payload*.



4.1.2 Header dei messaggi

L'header è composto dai seguenti campi:

Descriptor ID	Payload Descriptor	Age	Payload Length
16 byte	*1 byte*	*1 byte*	*4 byte*

Descriptor ID: Una stringa che identifica il messaggio nella rete, serve ad eliminare il traffico ridondante. Viene generato nel seguente modo:

- 2 byte: per la porta del nodo (la prima usata dal protocollo).
- 4 byte : per l'ip del nodo
- 6 byte : per un Id generato casualmente
- 4 byte : per il controllo di flusso (evita l'utilizzo dello stesso *Descriptor Id* per un nuovo messaggio, quando ancora il vecchio circola nella rete).

Payload Descriptor: identifica la tipologia del messaggio. Attualmente sono utilizzati i seguenti valori: 0*x*00 (BoostrapPing1), 0*x*01 (BoostrapPong1), 0*x*02 (Bye), 0*x*78 (Informations1), 0*x*80 (Query), 0*x*81(QueryHit).

Age: Ogni peer incrementa questo campo di 1 prima di passarlo ad un altro nodo della rete. Quando l'*Age* raggiunge il valore massimo, il messaggio non viene più inoltrato. Il campo *Age* rappresenta l'unico meccanismo in grado di far scomparire i messaggi dalla rete, eventuali modifiche sconsiderati del valore massimo di *Age* possono portare ad un aumento del traffico e una ridotta efficienza del sistema.

PayloadLength: Rappresenta la lunghezza del messaggio nel pacchetto.

4.1.3 I Messaggi

Analizziamo adesso le tipologie di messaggi in dettaglio.

4.1.3.1 Messaggi di Bootstrap

I messaggi che fanno parte di questa tipologia sono:

0*x*00 BootstrapPing1

Porta	IPv4
2 byte	*4 byte*

Usato per la ricerca di nodi attivi nella rete. Un nodo che riceve questo messaggio, con una probabilità P_i (in relazione al campo *Age*) inserisce il nodo sorgente nella sua vista e risponde con un B*ootstrapPong1*. Inoltre il nodo è tenuto ad inviare il messaggio ai peer del suo *fan-out,* presupposto che l'*Age* non superi il valore stabilito. E' caratterizzato dai seguenti campi di *payload*:

- **Porta:** La prima porta utilizzata dal nodo che genera il BootstrapPing1, rappresentata in formato *network order.*
- **IPv4:** L'indirizzo ip del nodo sorgente che genera il BootstrapPing1, rappresentato in formato *network order.*

0*x*01 BootstrapPong1

Porta	IPv4
2 byte	*4 byte*

Ogni nodo che riceve un B*ootstrapPing1* ha una probabilità P_i (in relazione al campo *Age*) di rispondere con un B*ootstrapPong1*. Il BootstrapPong1 rispetto al Pong del protocollo *Gnutella* 0.4, non deve essere

inoltrato sul percorso seguito dal relativo BootstrapPong1, ma è inviato dal mittente del messaggio verso il richiedente, non ci sono percorsi intermedi.
Il campo *Age* dell'*header* è posto uguale a 0. I campi *payload* del messaggio sono:

- **Porta:** La prima porta utilizzata dal nodo che genera il BootstrapPong1, rappresentato in formato *network order.*
- **IPv4:** L'indirizzo ip del nodo che genera il BootstrapPong1, rappresentato in formato *network order.*

4.1.3.2 Messaggi di Richiesta

I messaggi che fanno parte di questa tipologia sono:

0*x*80 Query

Porta	IPv4	HashFile
2 byte	*4 byte*	*16 byte*

I messaggi Query sono utilizzati per effettuare delle ricerche nella rete. Sono messaggi *multicast* ed ogni nodo che li riceve è tenuto a confrontare la risorsa richiesta con il proprio insieme di risorse condivise. Così come accadeva per il BootstrapPing1, il messaggio Query viene inoltrato a tutti gli *F* nodi del *fan-out* a patto che l'*Age* non superi il valore stabilito. E' caratterizzato dai seguenti campi di *payload*:

- **Porta:** La prima porta utilizzata dal nodo che genera il Query, rappresentato in formato *network order.*
- **IPv4:** L'indirizzo ip del nodo che genera il Query, rappresentato in formato n*etwork order.*
- **HashFile:** un codice *hash* che identifica la risorsa condivisa nella rete.

0*x*80 QueryHits

Porta	IPv4	Sorge ID
2 byte	*4 byte*	*16 byte*

Viene inviato in risposta ad una Query se il peer trova un match fra l'insieme dei suoi dati condivisi in rete e la stringa *hash* contenuta nella Query analizzata. Il QueryHits rispetto al protocollo *Gnutella* 0.4 non deve essere inoltrato sul percorso seguito dal relativo Query, ma è inviato dal fruitore della risorsa verso

il richiedente, non ci sono percorsi intermedi.
Il campo *Age* dell'*header* è posto uguale a 0. Il messaggio è caratterizzato dai seguenti campi:

- **Porta:** La prima porta utilizzata dal nodo che genera il QueryHits, rappresentato in formato *network order.*
- **IPv4:** L'indirizzo ip del nodo che genera il QueryHits, rappresentato in formato *network order.*
- **SorgeID:** Il *Descriptor Id* del Query ricevuto dal nodo.

4.1.3.3 Il messaggio di Bye

0*x*02 Bye

Porta	IPv4
2 byte	*4 byte*

Il messaggio ha validità solo fino ad una A*ge* prestabilita, superata questa sarà scartato. L'uso di questo tipo di messaggio non è indispensabile, ma accelera l'estinzione delle informazioni inconsistenti. E' caratterizzato dai seguenti campi di *payload*:

- **Porta:** La prima porta utilizzata dal nodo che genera il Bye, rappresentato in formato *network order.*
- **IPv4:** L'indirizzo ip del nodo che genera il Bye, rappresentato in formato *network order.*

4.1.3.4 Il messaggio di Informations1

0*x*78 Informations1

Neighbour 1			.....	Neighbour n			Sorgente	
Porta	IPv4	Age	.....	Porta	IPv4	Age	Porta	IPv4
2 byte	*4 byte*	*1byte*	.....	*2 byte*	*4 byte*	*1byte*	*2 byte*	*4 byte*

I messaggi di Informations1 sono utilizzati per la diffusione di informazioni nella rete. Sono messaggi di raggio 1 (non vengono inoltrati dal ricevente ad altri nodi) ed ogni nodo che li riceve è tenuto a confrontare le informazioni ricevute con le proprie, per effettuare eventuali aggiornamenti. Il campo *Age* dell'*header* è posto uguale a 0 ed i byte del controllo di flusso del campo *DescriptorId* sono posti a 0. Il messag-

gio è caratterizzato dai seguenti campi:

- **Porta***(neigbour)***:** La prima porta utilizzata dal *neighbour*, rappresentato in formato *network order*.
- **IPv4***(neigbour)***:** L'indirizzo ip del *neighbour*, rappresentato in formato *network order*.
- **Age***(neigbour)*: L'*Age* dell'informazione di *membership* relativa al *neighbour*.
- **Porta***(sorgente)***:** La prima porta utilizzata dal nodo che genera l'Informations1, rappresentato in formato *network order*.
- **IPv4***(sorgente)***:** L'indirizzo ip del nodo che genera l'Informations1, rappresentato in formato *network order*.

4.1.4 Meccanismo di routing

Il meccanismo di *routing* del protocollo 2Rounds si basa su poche e semplici regole:

- Un peer dovrebbe inoltrare un messaggio di BoostrapPing1, Query e Bye *a F* nodi cui è collegato, escluso quello mittente, a patto che l'*Age* non superi il valore stabilito.
- Il messaggio QueryHits deve essere inviato solo al nodo che ha trasmesso il corrispondente Query.
- I messaggi BootstrapPong1 devono essere trasmessi con probabilità P_i solo al nodo che ha trasmesso il corrispondente BootstrapPing1.
- I messaggi di Informations1 sono trasmessi dal peer a *F* nodi vicini, non vengono inoltrati dai nodi riceventi.
- Prima di rinviare un messaggio il peer deve modificare il campo *Age*. Se il campo *Age* risulta uguale al massimo valore stabilito, il messaggio non deve più essere ritrasmesso.
- Un peer a cui perviene uno messaggio con un *Descriptor Id* incontrato precedentemente, dovrebbe scartare il messaggio in modo da non generare traffico ridondante nella rete.

4.1.5 Gestione delle connessioni tra i peer

Una volta reperito l'indirizzo di un peer è possibile stabilire una connessione TCP a questo. Se la fase di connessione ha dato esito positivo il peer può comunicare con gli altri nodi della rete attraverso l'invio/ricezione di messaggi del protocollo. Ogni peer è implementato in modo da avere 2 *dispatcher* il primo dedicato ai messaggi di Informations1 e il secondo per i rimanenti messaggi. Approfondimenti sull'imple-

mentazione dei *dispatcher* saranno forniti nei capitoli successivi.

4.1.6 Vantaggi e svantaggi del protocollo 2Rounds 1.0

Il protocollo *2Rounds* 1.0 è stato sviluppato per reti P2P completamente decentralizzate, in quanto tale gode (in linea di massima) degli stessi vantaggi svantaggi propri a questa tipologia di reti.
I principali aspetti positivi del protocollo 1.0 sono i seguenti:

- La rete essendo decentralizzata non è organizzata da nessun server centrale pertanto:
  1. Non c'è la possibilità di eventuali colli di bottiglia (almeno in linea teorica).
  2. Si ha un alta *fault-tolerance* grazie all'esplorazione di molti percorsi.
  3. Non è suscettibile a possibili attacchi di tipo DoS
- Non è limitata allo scambio una sola tipologia di risorsa (vedi *Napster* che supportava solo lo scambio di file .mp3).
- Il BootstrapPing1 non genera eccessivo traffico di rete, poiché è usato solo in fase di *bootstrap*.
- Il QueryHits non viene inoltrato verso percorsi intermedi tra il fruitore della risorsa e il richiedente. Complessivamente c'è un risparmio di banda nella rete (vedi il QueryHits del protocollo *Gnutella* 0.4).
- Il BoostrapPong1 non viene inoltrato verso percorsi intermedi tra il mittente del messaggio e il richiedente. Complessivamente c'è un risparmio di banda nella rete (vedi il Pong del protocollo *Gnutella* 0.4).
- Permette una buona scalabilità al sistema, grazie alle informazioni di *membership* del messaggio Informations1.

I principali difetti che si possono evincere dalla struttura di *2Rounds* 1.0 sono i seguenti:

- L'invio del messaggio Query comporta un carico eccessivo per la rete infatti per una *Age* = 7 e *F* = 4 ogni messaggio Query genera circa 16K messaggi.
- Ogni peer ha una “vista limitata”: l'*Age* non permette al messaggio Query di raggiungere i nodi della rete che sono localizzati al passo successivo, oltre il quale L'*Age* supera il valore stabilito.
- La topologia è incontrollabile e imprevedibile in quanto è arduo stimare il tempo di esecuzione del messaggio Query o la probabilità di successo di questo.

4.2 Il protocollo di rete 2Rounds 0.2

Come si è potuto facilmente notare, il problema che affligge il protocollo di rete *2Rounds* 1.0 è l'eccessivo *flooding* nella rete del messaggio Query, che tuttavia non riesce a garantire una buona localizzazione delle risorse. Anche se rispetto al protocollo *Gnutella* si è diminuito lo spreco di banda per i messaggi BootstrapPing, BootstrapPong e QueryHits non possiamo essere ancora soddisfatti, bisogna limitare l'utilizzo di banda del Query. La soluzione da me proposta nel protocollo 2Rounds 2.0 riduce il carico di rete e migliora il problema della "vista limitata". Per il problema della topologia incontrollabile e imprevedibile non ci sono soluzioni, in quanto come detto in precedenza una delle caratteristiche degli algoritmi epidemici è la generazione di una topologia di rete casuale che tende a mutare molto velocemente.
Per evitare inutili ripetizioni tratterò solo le modifiche della versione 2.0 rispetto alla 1.0 .

4.2.1 Definizione del protocollo

Il protocollo di rete 2Rounds 2.0 supporta i seguenti messaggi:

Tipo messaggio	**Descrizione**
BootstrapPing2	Messaggio per il *discovery* di nodi nella fase di *boostrap*, oltre alla porta e l'indirizzo contiene alcuni descrittori di risorsa propri del nodo che lo ha generato.
BootstrapPong2	Messaggio di risposta ad un BootstrapPing2: un nodo che riceve un BootstrapPing2 può risponde con uno BootstrapPong2, includendo il proprio indirizzo e alcuni descrittori di risorse condivisi.
Bye	Identico alla versione 1.0
Informations2	Messaggio con informazioni di *membership* e di localizzazione di risorse.
Query	Identico alla versione 1.0
QueryHits	Identico alla versione 1.0

Come è possibile notare Bye, Query, QueryHits non hanno subito modifiche con la migrazione dalla 1.0 alla 2.0 . Può sembrare paradossale il fatto che si voglia migliorare l'efficienza del messaggio Query e non se ne modifichi la struttura, ma così non è, infatti nel seguito vedremo in dettaglio come le informazioni di localizzazione delle risorse sono disseminate nella stessa maniera di quelle di membership.
I messaggi nella versione 2.0 continuano ad essere composti da due macro strutture l'*header* (comune a tutti i tipi di messaggi), e il *payload*. La struttura dell'*header* non ha subito modifiche nella 2.0, l'unico aggiornamento è riscontrabile nell'introduzione di 3 nuovi *payload Descriptor*: 0*x*79 *Informations2,* 0*x*03

BootstrapPing2, 0*x*04 *BootstrapPong2.*

4.2.2 I messaggi

0*x*03 BootstrapPing2

Porta	IPv4	HashFile 1	....	HashFile n
2 byte	*4 byte*	*16 byte*	....	*16 byte*

Usato per la ricerca di nodi attivi nella rete. Un nodo che riceve questo messaggio, con una probabilità P_i (in relazione al campo *Age*) inserisce il nodo sorgente nella sua vista e risponde con un B*ootstrapPong2*. Inoltre il nodo è tenuto a inoltrare il messaggio agli *F* nodi del suo *fan-out* a patto che l'*Age* non superi il valore stabilito. L'invio di alcuni *HashFile* posseduti dalla sorgente porta ad un miglioramento della localizzazione delle risorse. Il messaggio è caratterizzato dai seguenti campi di *payload*:

- **Porta:** La prima porta utilizzata dal nodo che genera il BootstrapPing2, rappresentata in formato *network order.*
- **IPv4:** L'indirizzo ip del nodo sorgente che genera il BootstrapPing2, rappresentato in formato *network order.*
- **HashFile** *(1..n)***:** identificatore di risorsa condivisa del nodo che genera il BootstrapPing2.

0*x*04 BootstrapPong2

Porta	IPv4	HashFile 1	....	HashFile n
2 byte	*4 byte*	*16 byte*	....	*16 byte*

Ogni nodo che riceve un B*ootstrapPing2* ha una probabilità P_i (in relazione al campo *age*) di rispondere con un B*ootstrapPong2*. Il BootstrapPong2 rispetto al Pong del protocollo *Gnutella* 0.4 non deve essere inoltrato sul percorso seguito dal relativo BootstrapPong2, ma è inviato dal mittente del messaggio verso il richiedente, non ci sono percorsi intermedi.
Il campo *Age* dell'*header* è posto uguale a 0. I campi *payload* del messaggio sono:

- **Porta**: La prima porta utilizzata dal nodo che genera il BootstrapPong2, rappresentato in formato *network order.*
- **IPv4:** L'indirizzo ip del nodo che genera il BootstrapPong2, rappresentato in formato *network order.*
- **HashFile** *(1..n)***:** identificatore di risorsa condivisa del nodo che genera il BootstrapPong2.

0*x*79 Informations2

Neighbour 1						...	Neighbour F						Sorgente				
Porta	IPv4	Age	HashFile 1	...	HashFile n	...	Porta	IPv4	Age	HashFile 1	...	HashFile n	Porta	IPv4	HashFile 1	...	HashFile n
2 byte	*4 byte*	*1byte*	*16 byte*	...	*16 byte*	...	*2 byte*	*4 byte*	*1byte*	*16 byte*	...	*16 byte*	*2 byte*	*4 byte*	*16 byte*	...	*16 byte*

I messaggi di Informations2 sono utilizzati per la disseminazione di informazioni di *membership* e di localizzazione delle risorse condivise nella rete. Sono messaggi di raggio 1 (non vengono inoltrati dal ricevente ad altri nodi) ed ogni nodo che li riceve è tenuto a confrontare le informazioni ricevute con le proprie, per effettuare eventuali aggiornamenti. A differenza dell'Informations1 un nodo che intende mandare un messaggio di Informations2 a *F* nodi (*fan-out*) per disseminare delle informazioni genererà *F* Informations2 diversi. Questo comportamento è necessario per rendere la disseminazione più variegata possibile. Il campo *Age* dell'*header* è posto uguale a 0 ed i byte del controllo di flusso per il campo *Descriptor Id* dell'*header* sono posti a zero. Il messaggio è caratterizzato dai seguenti campi:

- **Porta** *(neigbour)*: La prima porta utilizzata dal *neighbour*, rappresentato in formato *network order*.
- **Ipv4** *(neigbour)***:** L'indirizzo ip del *neighbour*, rappresentato in formato *network order*.
- **Age** *(neigbour)*: L'*Age* delle informazioni relativa al *neighbour*.
- **HashFile** *(1..n) (neigbour)***:** Identificatore di risorsa condivisa del *neighbour*.
- **Porta** *(sorgente)***:** La prima porta utilizzata dal nodo che genera l'Informations2, rappresentato in formato *network order*.
- **Ipv4** *(sorgente)***:** L'indirizzo ip del nodo che genera l'Informations2, rappresentato in formato *network order*.
- **HashFile** *(1..n) (sorgente)***:** Identificatore di risorsa condivisa del nodo che genera l'informations2.

Per quanto concerne il *routing* dei messaggi, nel protocollo *2Rounds* 2.0 abbiamo che i messaggi di Query, QueryHits e Bye non variano il loro comportamento rispetto alla versione 1.0. I messaggi di BootstrapPing2 e BootstrapPong2 hanno lo stesso comportamento dei loro antenati: BootstrapPing1 e BooootstrapPong1. L'Informations2 rispetto all'Informations1 non viene inoltrato a *F* nodi, ma ad unico nodo; in sintesi invece di un messaggio di informations1 inoltrato a *F* nodi abbiamo *F* messaggi distinti di Informations2 inoltrati ciascuno ad uno degli *F* nodi del *fan-out*.
Le connessioni vengono gestite come nella versione 1.0.

4.2.3 Vantaggi del 2Rounds 2.0

Come accennato precedentemente i vantaggi del *2Rounds 2.0* riguardano la riduzione del carico di rete e la risoluzione parziale del problema legato alla *vista limitata*, entrambi relativi al messaggio Query. Analizzando i due problemi si nota una loro relazione, infatti per avere una vista più profonda è possibile aumentare l'*Age* massima del messaggio. Questa soluzione risolve in parte il problema della *vista limitata*, ma aumenta il carico di rete, in quanto ad un *Age* massima più elevata corrisponde un incremento esponenziale del numero dei messaggi generati dall'invio di un Query. Per diminuire il carico di rete invece sarebbe possibile diminuire il valore massimo del *Age* del messaggio Query, questa modifica ridurrebbe il numero di messaggi generati dal suo invio, ma diminuirebbe anche la copertura di rete del messaggio inviato, ossia aumenterebbe il problema della *vista limitata*.
Come è possibile notare entrambi i problemi non sono risolvibili mediante la modifica strutturale o parametrica del messaggio Query, infatti nella versione 2.0 non si è effettuata nessuna modifica sul suddetto messaggio. Una delle possibile soluzione a questo duplice problema consiste nella diffusione di informazioni di localizzazione delle risorse, infatti attraverso l'Informations2 vengono disseminate non solo informazioni di *membership*, ma anche descrittori di risorse condivise. Cosi' i nodi oltre ad avere una *view* sui vicini la hanno anche sulle risorse dei suddetti. Questa soluzione aumenta la profondità della vista sulle risorse condivise diminuendo il numero di messaggi Query inoltrati nella rete. Naturalmente c'è un prezzo da pagare, rappresentato dall'aumento delle dimensioni del messaggio di Informations2 rispetto all'Informations1. L'aumento di dimensione dell'Informations2 è direttamente proporzionale al numero di descrittori di risorsa associati ad ogni *neigbour* della lista del messaggio. Quindi associare un numero elevato di descrittori ad ogni *neighbour* potrebbe apportare molti problemi in fase di trasmissione e peggiorare le prestazioni del sistema, in sintesi è sconsigliato un aumento sconsiderato dei suddetti.

5 L'algoritmo 2Rounds 2.0

2Rounds 2.0 è un algoritmo completamente decentralizzato in quanto non viene mai richiesto ad ogni nodo che lo esegue, in nessuna sua fase, la conoscenza completa del sistema. Nel seguito della trattazione presenteremo solo la versione 2.0 dell'algoritmo *2Rounds* che lavora con il corrispettivo protocollo di rete 2.0.

Come si evince dal nome una delle sue caratteristiche fondamentali è l'utilizzo di 2 round: il primo detto *roundGossip* è usato dal nodo per disseminare informazioni di membership e di localizzazione delle risorse; il secondo il *roundSpeed* è usato dal nodo per l'inoltro di messaggi che necessitano di una certa priorità temporale. I due round sono a tutti gli effetti due processi, come vedremo nel seguito, il round principale (*roundGossip*) scandisce il tempo per l'inoltro di quasi tutti i messaggi del nodo, infatti oltre a sincronizzare i suoi vari sotto-processi si preoccupa della sincronizzazione del *roundSpeed*. Entrambi hanno un tempo di vita detto: *timeGossip* per il *roundGossip* e *timeSpeed* per il *roundSpeed*.

Altri due processi fondamentali sono quelli di *dispatching*: il *dispatcherInformations* che si occupa della ricezione di messaggi del tipo *Informations2*; e il *dispatcherMessage* preposto alla ricezione di tutti i messaggi esclusi quelli di *Informations2*.

L'algoritmo, come mostrato in *figura* 5.1 consta di due fasi:

- La *fase boostrap* utilizzata: per l'inizializzazione di tutte le strutture usate dall'algoritmo, per il *discovery* dei primi vicini del nodo e per la creazione dei vari processi.
- La *fase round work*, in cui il peer può interagire con il sistema nel pieno delle sue funzionalità.

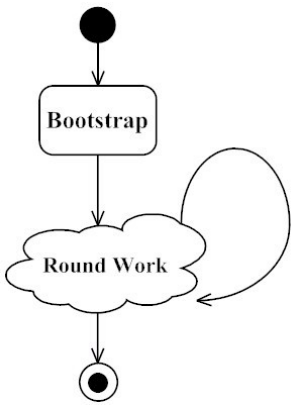


Figura 5.1: Le fasi di 2Rounds

Fase di bootstrap

La *fase di boostrap* fornisce al peer (come mostrato in *figura* 5.2) le strutture, le informazioni e i processi necessari per interagire con il sistema.

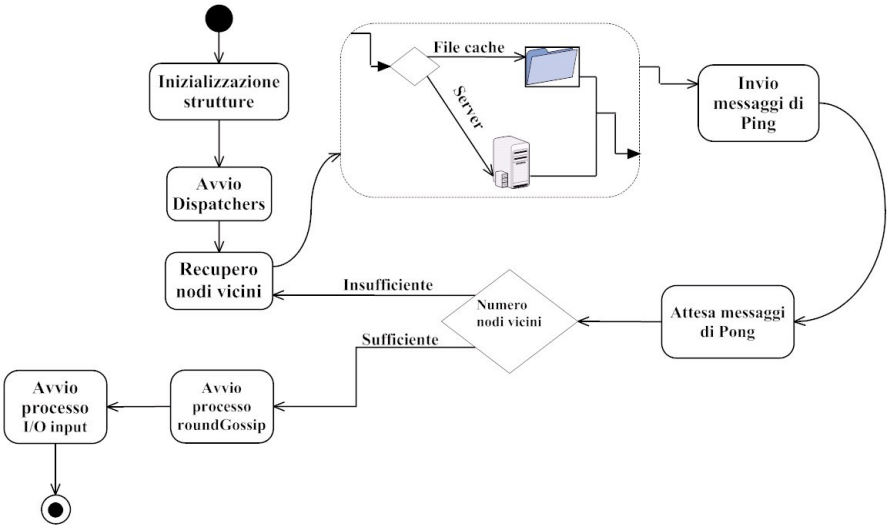


Figura 5.2: La fase di boostrap

La prima operazione eseguita nella fase di *bootstrap* è l'inizializzazione delle strutture, seguita poi dalla creazione dei *dispatcherInformations* e *dispatcherMessage*. L'operazione più delicata è quella di recupero nodi, in cui vengono reperite delle informazioni di *membership* su alcuni peer, attraverso un *file cache* oppure contattando un server di *bootstrap*. Si invia un BootstrapPing2 ad ognuno di questi peer e si attendono gli eventuali BootstrapPong2, generati dai peer contattati direttamente e da quelli a cui e stato inoltrato il messaggio di BootstrapPing2. Se i messaggi di BoostrapPong2 ricevuti sono stati sufficienti a creare una *view* iniziale, si procede con l'operazione successiva, altrimenti si ripete l'operazione di recupero nodi fino a quando non si è in grado di generare una *view.* Se il recupero nodi ha avuto successo si procede con la creazione e avvio dei processi di *roundGossip* e di *I/O input*.

Fase round work

Dopo che la *fase bootstrap* è terminata si entra nella fase di *round work*, dove abbiamo quattro fornitori di carico computazionale (detti *processi primari*):

- Il *dispatcherInformations,* che gestisce i sotto-processi adibiti alla ricezione ed elaborazione di messaggi di Informations2.
- Il *dispatcherMessage,* che gestisce i sotto-processi adibiti alla ricezione ed elaborazione di tutti i messaggi esclusi quelli di Informations2.

- Il *roundGossip* che si gestisce i sotto-processi per invio dell'Informations2 ai peer del *fan-out* e sincronizza temporalmente i cicli del *roundSpeed*.
- L' *I/O input,* che si occupa di gestire le richieste di risorse dell'utente avviando il processo *I/O output* per la visualizzazione dei risultati.

L'insieme costituito dal *processo primario* e dai suoi sotto-processi è detta *famiglia di processi*.
Prima di illustrare in dettaglio le varie famiglie di processi del *2Rounds*, conviene prima analizzare le strutture che vengono utilizzate da questi.

5.1 Le strutture

L'algoritmo 2Rounds utilizza le seguenti strutture:

- **BufferMyFile:** Lista delle risorse condivise dal nodo.
- **BufferNeighbour:** Lista, ordinata per *Age*, delle informazioni di *membership* con i rispettivi descrittori di risorsa.
- **VarX:** Variabile aleatoria utilizzata insieme al campo *Age* per generare una probabilità.
- **BufferFanout:** Lista di nodi che costituiscono il *fan-out* del nodo per quel *roundGossip*.
- **BufferMessage:** Lista dei *Descriptor ID* di messaggi ricevuti e inviati.
- **BufferByePeer:** Lista di nodi che hanno abbandonato il sistema.
- **BufferEvent:** Lista risultati generati dall'invio di una o più Query.
- **BufferRoundSpeed** (detto anche **BufferRS**)**:** Struttura composta da due code, utilizzate per l'invio e la ricezione di messaggi, esclusi quelli di Informations2.
- **BufferRoundGossip (**detto anche **BufferRG**)**:** Lista per la memorizzazione di Informations2 generati dal *roundGossip*.
- **Strutture di sincronizzazione***:* strutture utilizzate per la sincronizzazione dei processi dell'algoritmo.

5.1.1 La struttura BufferMyFile

La struttura BufferMyFile è utilizzata per la memorizzazione di meta-dati relativi alle risorse condivise dal nodo.

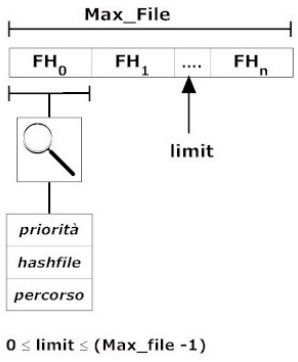


Figura 5.3: La struttura BufferMyFile

Come è possibile notare dalla figura 5.3 BufferMyfile è una lista di descrittori *FH*, con dimensioni finite. Il parametro *Max_File* indica il massimo numero di elementi che può contenere la struttura, mentre *limit* punta all'ultima posizione occupata da un elemento *FH* nella suddetta.
Un descrittore di risorsa *FH* è costituito dai seguenti campi:

- **priorità:** permette di aumentare/diminuire la probabilità di disseminazione del descrittore.
- **hashfile:** un codice hash di 16 byte che identifica la risorsa nella rete.
- **percorso:** contiene la posizione della risorsa nel disco.

Questa struttura risulta fondamentale quando si riceve un messaggio di Query, infatti una delle prime operazioni che viene eseguita è il confronto del campo *hashfile* del messaggio con i vari *hashfile* contenuti negli elementi *FH*.

5.1.2 La struttura BufferNeighbour

La struttura BufferNeighbour è utilizzata per la memorizzazione di meta-dati relativi ai nodi vicini, costituisce la *view* del nodo.

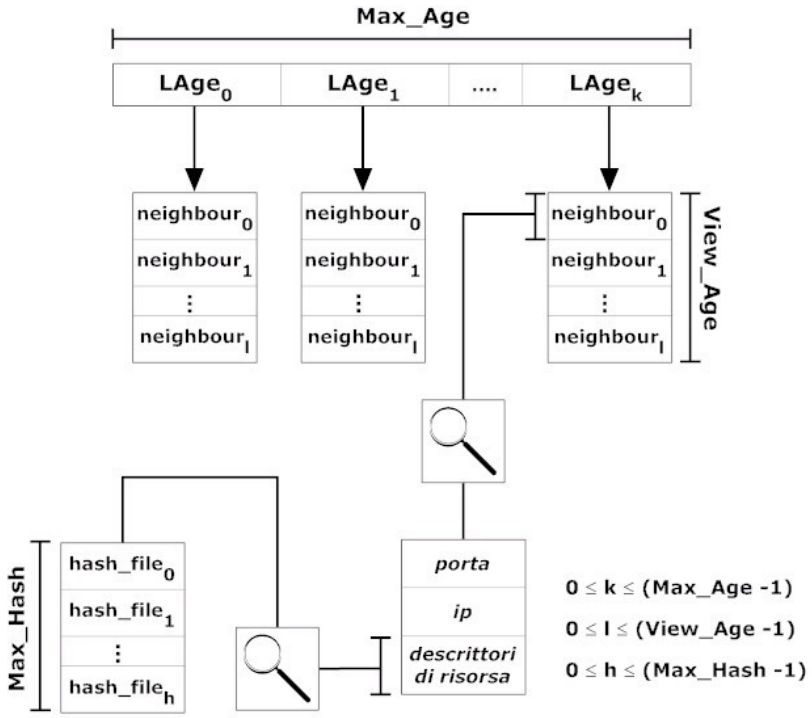


Figura 5.4: La struttura BufferNeighbour

Come mostrato nella figura *5.4* il BufferNeighbour è un insieme di *Max_Age* liste (*Lage)* che contengo-

no dei descrittori *neighbour*. Ogni lista contiene i *neighbour* con una data *Age* e può memorizzare massimo *View_Age* descrittori.

Un descrittore neighbour contiene meta-dati su un peer vicino ed è composto dai seguenti campi:

- **porta:** La porta usata dal peer vicino.
- **ip:** L'ip del peer vicino.
- **descrittori di risorsa:** Un insieme di *hashfile* (vedi *BufferMyFile*), massimo *Max_Hash*, condivisi dal peer vicino. In generale le risorse, identificate dagli *hashfile* contenuti nel campo *descrittori di risorsa*, costituiscono un sottoinsieme di tutte quelle condivise dal peer vicino.

Il BufferNeighbour può memorizzare massimo $Max\_Age \cdot View\_Age$ descrittori *neighbour* tutti distinti tra loro. Per evitare di saturare lo spazio della struttura sarebbe opportuno cancellare i descrittori più vecchi, la procedura adottata dall'algoritmo *2Rounds* per "svecchiare" la struttura è mostrata *figura* 5.5.

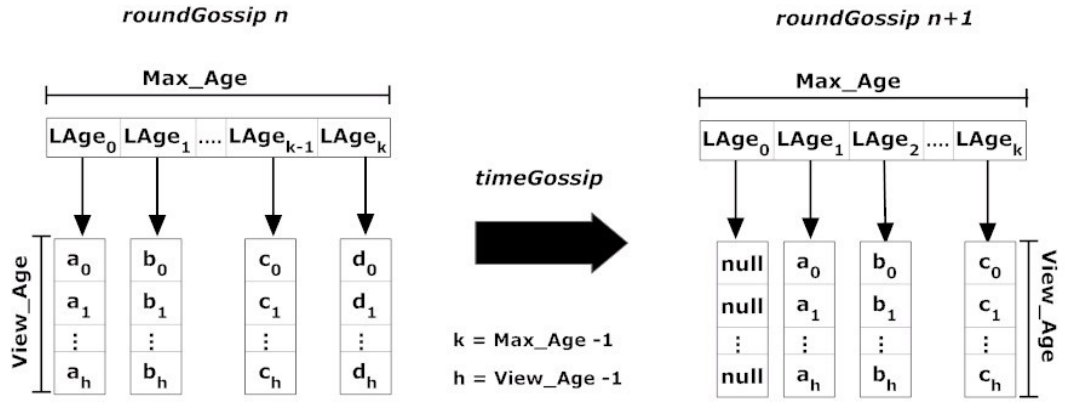


Figura 5.5: Gestione dell'Age nel BufferNeighbour

Ogni *timeGossip* viene incrementata l'*Age* di tutta la struttura BufferNeighbour, ossia ogni lista *LAge* avrà i descrittori con l'*Age* incrementata di 1. La lista *LAge* che dopo l'incremento avrà l'*Age* dei descrittori uguale a *Max_Age,* sarà resettata e messa a disposizione per la memorizzazione dei nuovi meta-dati in arrivo. I meta-dati memorizzati nel BufferNeighbour provengono dalle informazioni contenute nei messaggi di: Informations2, BootstrapPong2, BootstrapPing2 e Bye.

5.1.3 La struttura VarX

La struttura VarX è utilizzata per definire il comportamento di una variabile aleatoria discreta, utilizzata per generare le probabilità associate al variare dell'*Age*.

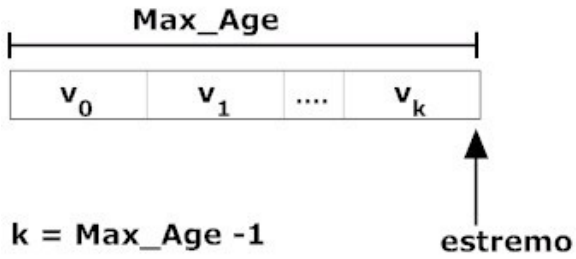


Figura 5.6 La struttura VarX

Come mostrato in figura *5.6 VarX* è composta da un insieme di *Max_Age* elementi di tipo *v* che definiscono, una partizione dello spazio campionario discreto Ω. L'elemento *estremo* identifica l'elemento di valore massimo in Ω. Per far comprendere al lettore come la struttura VarX riesca a descrivere una variabile aleatoria è necessario introdurre alcuni concetti di *teoria dei fenomeni aleatori*.

Definiamo il nostro spazio di probabilità (Ω,A,P) dove:

$\Omega = \{1 , 2 , ... , estremo\}$

eventi elementari $w_i = \{$ "*genero un numero random* ***i*** "$\}$ con $i = 1, 2, ... , estremo$

sigma-algebra $A = \mathcal{P}(\Omega)$

misura di probabilità $P: A \rightarrow [0 ,1]$:

$P(w_i) = (1 / card(\Omega))$ *con* $i = 1, 2, ... , estremo$
$P(A_i) = (card(A_i) / card(\Omega))$

Definiamo una variabile aleatoria discreta *X*:

$X: \Omega \rightarrow [0 , MAX\_AGE -1]$
$X: \{x_0 , x_1 , .. , x_k\}$ con k da 0 a MAX_AGE-1

Dati i seguenti parametri:

- *NUM*: fattore moltiplicativo dato (dipendente dal sistema).
- P_0 , P_1 , P_2: probabilità date (dipendente dal sistema).

Definiamo la variabile aleatoria X in base alle variazioni di *MAX_AGE:*

- per MAX_AGE = 1
  - $X\ (\ 0\ \le\ w \le (\ \mathrm{NUM}\ \cdot \mathrm{P}_0\ )\ ) =\ 0$
- per MAX_AGE = 2
  - $X\ (\ 0\ \le\ w \le (\ \mathrm{NUM}\ \cdot \mathrm{P}_0\ )\ ) =\ 0$
  - $X(\ (\ \mathrm{NUM}\ \cdot\ \mathrm{P}_0\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1\ )\ )\ ) = 1$
- per MAX_AGE = 3
  - $X\ (\ 0\ \le\ w \le (\ \mathrm{NUM}\ \cdot \mathrm{P}_0\ )\ ) =\ 0$
  - $X(\ (\ \mathrm{NUM}\ \cdot\ \mathrm{P}_0\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1\ )\ )\ ) = 1$
  - $X(\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1\ )\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2)\ )\ ) = 2$
- per MAX_AGE = 4
  - $X\ (\ 0\ \le\ w \le (\ \mathrm{NUM}\ \cdot \mathrm{P}_0\ )\ ) =\ 0$
  - $X(\ (\ \mathrm{NUM}\ \cdot\ \mathrm{P}_0\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1\ )\ )\ ) = 1$
  - $X(\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1\ )\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2)\ )\ ) = 2$
  - $X(\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2)\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2 + \sum_{j=4}^{4} \frac{1}{2\mathrm{j}}\ ))\ ) = 3$
- per MAX_AGE = 5
  - $X\ (\ 0\ \le\ w \le (\ \mathrm{NUM}\ \cdot \mathrm{P}_0\ )\ ) =\ 0$
  - $X(\ (\ \mathrm{NUM}\ \cdot\ \mathrm{P}_0\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1\ )\ )\ ) = 1$
  - $X(\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1\ )\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2)\ )\ ) = 2$
  - $X(\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2)\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2 + \sum_{j=4}^{4} \frac{1}{2\mathrm{j}}\ ))\ ) = 3$
  - $X(\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2 + \sum_{j=4}^{4} \frac{1}{2\mathrm{j}}\ ))\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2 + \sum_{j=4}^{5} \frac{1}{2\mathrm{j}}\ )) = 4$
- per MAX_AGE = n
  - $X\ (\ 0\ \le\ w \le (\ \mathrm{NUM}\ \cdot \mathrm{P}_0\ )\ ) =\ 0$
  - $X(\ (\ \mathrm{NUM}\ \cdot\ \mathrm{P}_0\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1\ )\ )\ ) = 1$
  - $X(\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1\ )\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2)\ )\ ) = 2$
  - $X(\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2)\ ) < w \le\ (\ NUM \cdot (\ \mathrm{P}_0 + \mathrm{P}_1 + \mathrm{P}_2 + \sum_{j=4}^{4} \frac{1}{2\mathrm{j}}\ ))\ ) = 3$

- $X(\,(\,NUM\cdot(\,P_0+P_1+P_2+\sum_{j=4}^{4}\frac{1}{2^j}\,))<w\leq(\,NUM\cdot(\,P_0+P_1+P_2+\sum_{j=4}^{5}\frac{1}{2^j}\,))=4$

  .
  .
  .

- $X(\,(\,NUM\cdot(\,P_0+P_1+P_2+\sum_{j=4}^{n-1}\frac{1}{2^j}\,))<w\leq(\,NUM\cdot(\,P_0+P_1+P_2+\sum_{j=4}^{n}\frac{1}{2^j}\,))=n$

Definiamo la funzione di probabilità:

$p_x(x_k) = p_k$

Definiamo la variabile la funzione di probabilità p_x in base alle variazioni di _MAX_AGE:

- per MAX_AGE = 1 e X: $\{x_0\}$
  - $p_x(x_0) = 1$
- per MAX_AGE = 2 e X: $\{x_0, x_1\}$
  - $p_x(x_0) = \frac{NUM\cdot P_0}{card(\Omega)}$
  - $p_x(x_1) = \frac{(NUM\cdot(P_0+P_1))-(NUM\cdot P_0)}{card(\Omega)}$
- per MAX_AGE = 3 e X: $\{x_0, x_1, x_2\}$
  - $p_x(x_0) = \frac{NUM\cdot P_0}{card(\Omega)}$
  - $p_x(x_1) = \frac{(NUM\cdot(P_0+P_1))-(NUM\cdot P_0)}{card(\Omega)}$
  - $p_x(x_2) = \frac{(NUM\cdot(P_0+P_1+P_2))-(NUM\cdot(P_0+P_1))}{card(\Omega)}$
- per MAX_AGE = 4 e X: $\{x_0, x_1, x_2, x_3\}$
  - $p_x(x_0) = \frac{NUM\cdot P_0}{card(\Omega)}$
  - $p_x(x_1) = \frac{(NUM\cdot(P_0+P_1))-(NUM\cdot P_0)}{card(\Omega)}$
  - $p_x(x_2) = \frac{(NUM\cdot(P_0+P_1+P_2))-(NUM\cdot(P_0+P_1))}{card(\Omega)}$

- $p_x(x_3) = \dfrac{\left(NUM \cdot \left(P_0 + P_1 + P_2 + \sum_{j=4}^{4} \frac{1}{2j}\right)\right) - \left(NUM \cdot \left(P_0 + P_1 + P_2\right)\right)}{card\,(\Omega)}$

- per MAX_AGE = 5 e X: $\{x_0, x_1, x_2, x_3, x_4\}$
  - $p_x(x_0) = \dfrac{NUM \cdot P_0}{card\,(\Omega)}$
  - $p_x(x_1) = \dfrac{\left(NUM \cdot \left(P_0 + P_1\right)\right) - \left(NUM \cdot P_0\right)}{card\,(\Omega)}$
  - $p_x(x_2) = \dfrac{\left(NUM \cdot \left(P_0 + P_1 + P_2\right)\right) - \left(NUM \cdot \left(P_0 + P_1\right)\right)}{card\,(\Omega)}$
  - $p_x(x_3) = \dfrac{\left(NUM \cdot \left(P_0 + P_1 + P_2 + \sum_{j=4}^{4} \frac{1}{2j}\right)\right) - \left(NUM \cdot \left(P_0 + P_1 + P_2\right)\right)}{card\,(\Omega)}$
  - $p_x(x_4) = \dfrac{\left(NUM \cdot \left(P_0 + P_1 + P_2 + \sum_{j=4}^{5} \frac{1}{2j}\right)\right) - \left(NUM \cdot \left(P_0 + P_1 + P_2 + \sum_{j=4}^{4} \frac{1}{2j}\right)\right)}{card\,(\Omega)}$

- per MAX_AGE = n e X: $\{x_0, x_1, x_2, x_3, x_4, \ldots, x_{n-1}\}$
  - $p_x(x_0) = \dfrac{NUM \cdot P_0}{card\,(\Omega)}$
  - $p_x(x_1) = \dfrac{\left(NUM \cdot \left(P_0 + P_1\right)\right) - \left(NUM \cdot P_0\right)}{card\,(\Omega)}$
  - $p_x(x_2) = \dfrac{\left(NUM \cdot \left(P_0 + P_1 + P_2\right)\right) - \left(NUM \cdot \left(P_0 + P_1\right)\right)}{card\,(\Omega)}$
  - $p_x(x_3) = \dfrac{\left(NUM \cdot \left(P_0 + P_1 + P_2 + \sum_{j=4}^{4} \frac{1}{2j}\right)\right) - \left(NUM \cdot \left(P_0 + P_1 + P_2\right)\right)}{card\,(\Omega)}$
  - $p_x(x_4) = \dfrac{\left(NUM \cdot \left(P_0 + P_1 + P_2 + \sum_{j=4}^{5} \frac{1}{2j}\right)\right) - \left(NUM \cdot \left(P_0 + P_1 + P_2 + \sum_{j=4}^{4} \frac{1}{2j}\right)\right)}{card\,(\Omega)}$
  - .
  - .
  - .
  - .
  - .
  - $p_x(x_{n-1}) = \dfrac{\left(NUM \cdot \left(P_0 + P_1 + P_2 + \sum_{j=4}^{n} \frac{1}{2j}\right)\right) - \left(NUM \cdot \left(P_0 + P_1 + P_2 + \sum_{j=4}^{n-1} \frac{1}{2j}\right)\right)}{card\,(\Omega)}$

Si può notare che: i valori *x* della variabile aleatoria X, sono i valori di *Age* dei descrittori *neighbour;* e ad ogni valore *x*, corrisponde un intervallo di Ω. Infatti generando un numero random *i* otteniamo il valore di *Age X(i)*. Per aumentare/diminuire la probabilità di uscita di un determinato valore di *Age* (ossia *x*), è sufficiente aumentare/diminuire l'intervallo di eventi elementari w_i associati al suddetto valore di *Age*. In questo modo è possibile creare una relazione inversamente proporzionale tra il valore di *Age* e la cardinalità dell'intervallo discreto di Ω. Quindi ai meta-dati che hanno un basso valore di *Age* è possibile far corrispondere un alta probabilità di invio p_k, mentre ai meta-dati più datati, ossia con valore di *Age* abbastanza elevato si può far corrispondere una bassa probabilità di invio p_k.

Ritornando alla descrizione della struttura VarX, si può facilmente comprendere che gli elementi *v* determinano i vari intervalli dello spazio campionario discreto Ω a cui sono associati i diversi valori di Age. L'utilizzo della struttura VarX, permette anche una definizione dinamica del numero di valori ad essa associata, ossia è possibile anche variare la cardinalità degli elementi *v* (*Max_Age*). Questa sua proprietà chiarisce lo scopo della suo utilizzo, infatti si sarebbe potuto associare ad ogni valore di *Age* una determinata probabilità, ma un aumento della cardinalità dei valori di *Age* (*Max_Age*) avrebbe richiesto una nuova ridefinizione statica delle probabilità ad essi associati, cosa che non succede con l'utilizzo della struttura VarX.

5.1.4 BufferFanout

La struttura BufferFanout contiene le informazioni di *membership* sui *neighbour* designati a costituire il *fan-out* del peer, per l'intero ciclo di *roundGossip*.

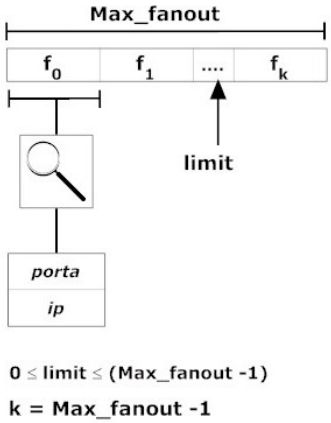


Figura 5.7: La struttura BufferFanout

Come è possibile notare dalla *figura* 5.7 BufferFanout è una lista di descrittori *f*. Il parametro *Max_fanout* indica il massimo numero di elementi che può contenere la struttura, mentre *limit* punta all'ultima posizio-

ne occupata da un elemento *f* nella suddetta.
Un descrittore di *membership f* è costituito dai seguenti campi:

- **porta:** La porta usata dal *neigbour*.
- **ip:** L'ip del *neighbour*.

Per tutto il *roundGossip* i messaggi di BootstrapPing2 , Bye e Query saranno inoltrati ai nodi identificati dagli elementi *f* della struttura. Inoltre ai nodi del *fan-out* sarà inoltrato un messaggio di Informations2 ogni *timeGossip*.
Ad ogni *roundGossip* vengono scelti i descrittori *f* (*massimo Max_fanout*) dal BufferNeighbour, in modo parzialmente casuale attraverso l'uso della struttura VarX. Mentre i vecchi descrittori *f* del precedente ciclo di *roundGossip* sono eliminati.

Algoritmo elezione descrittori *f*

```
Se card(neighbour) ≤ Max_fanout {
      limit = card(neighbour)
      copia tutti i neighbour del BufferNeighbour nel BufferFanout }

Se card(neighbour) > Max_fanout {
      Da  j = 0  fino a che a  j < Max_fanout {
            si genera in modo random un numero i di intorno [ 0 , maxAge ) ;
            k = X (i);
            si sceglie in modo casuale un elemento n di LAge_k ;
            Se n non è un elemento di BufferFanout {
                  n viene inserito nel BufferFanout;
                  j = j + 1 ; }
      }
}
```

L'algoritmo di elezione ha due comportamenti diversi a seconda del numero di *neighbour* contenuti nel BufferNeighbour. Infatti se si devono inserire nel BufferFanout *Max_fanout* descrittori, ma il BufferNeighbour ne contiene un numero inferiore, è inutile sceglierli in maniera casuale.
Quando la cardinalità dei *neighbour* è maggiore di *Max_fanout*, è possibile utilizzare una politica di elezione probabilistica, dove il valore *Age* è generato usando la struttura VarX.

5.1.5 La struttura BufferMessage

La struttura BufferMessage memorizza il campo *Descriptor ID* di alcune tipologie di messaggi inviati e ricevuti dal peer. Nella struttura sono memorizzati i messaggi di BoostrapPing2, Query, sia ricevuti che inviati e di Bye ricevuti. Per evitare l'invio di un messaggio, che è stato già precedentemente trasmesso c'è

bisogno di una *history* degli ultimi messaggi ricevuti ed inviati, questo è appunto il compito del Buffer-Message, la cui struttura è mostrata in *figura* 5.6.

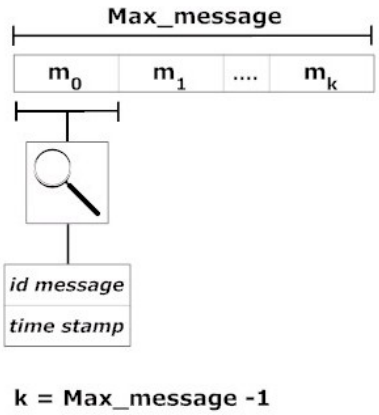


Figura 5.8: La struttura BufferMessage

I descrittori *m* sono così composti :

- **id message:** *Descriptor ID* del messaggio memorizzato
- **time stamp:** quando è stato memorizzato nel BufferMessage

Nella *figura* 5.8 ci viene mostrata anche la dimensione massima della struttura, pari a *Max_message*, da ciò è facilmente deducibile, che il BufferMessage senza una politica di eliminazione dei vecchi descrittori *m* saturerebbe lo spazio di memorizzazione. Definito un massimo tempo di vita *timeMsg*, comune a tutti i descrittori, è possibile calcolare per ogni *m* l' intervallo di vita *T* grazie al campo *time stamp,* cosi' da poter eliminare i descrittori il cui *T* è maggiore di *timeMsg*.

5.1.6 La struttura BufferByePeer

La struttura BufferByePeer è utilizzata per la memorizzazione dei peer che abbandonano il sistema. Alla ricezione di un Bye, il descrittore del peer che ha generato il messaggio è inserito nella struttura mostrata in *figura* 5.9.

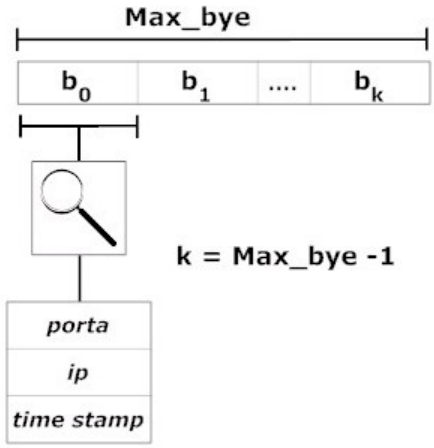


Figura 5.9: La struttura BufferByePeer

BufferByeePeer è un insieme di descrittori *b*, così composti:

- **porta:** porta del peer che abbandona il sistema
- **ip:** ip del peer che abbandona
- **time stamp:** quando è stato memorizzato nel BufferByePeer

Nella *figura* 5.9 ci viene mostrata anche la dimensione massima della struttura pari a *Max_bye*. Per evitare di saturare lo spazio di memorizzazione nel BufferByePeer viene adottata la stessa politica di eliminazione del BufferMessage.

5.1.7 La struttura BufferEvent

Il BufferEvent è preposto alla memorizzazione e gestione degli eventi generati dall'invio del Query. La struttura è preposta, come mostrato nella *figura* 5.10, alla memorizzazione di un insieme di descrittori di evento *e* (non più di *Max_event* descrittori). La procedura di eliminazione dei descrittori *e* che hanno esaurito la loro utilità è svolta dai *processi I/O* a loro associati, vedremo questo aspetto in dettaglio nel seguito del capitolo.

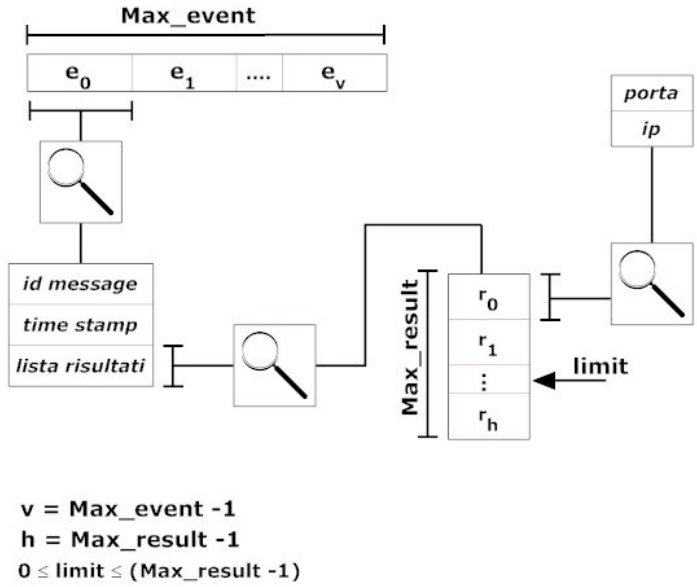


Figura 5.10: La struttura BufferEvent

Ogni descrittore *e* è così composto:

- **id message:** *Descriptor ID* del messaggio di Query che ha generato l'evento
- **time stamp:** quando è stato memorizzato nella struttura il descrittore di evento
- **lista risultati:** un insieme di descrittori *r* (massimo *Max_result*) che identificano i messaggi di QueryHits ricevuti. Il descrittore *limit* identifica l'ultima posizione della *lista risultati* che contiene un descrittore *r*.

Ad ogni QueryHits viene associato un descrittore *r* così composto:

- **porta:** porta del peer che ha inviato il QueryHits
- **ip:** ip del peer che ha inviato il QueryHits

5.1.8 La struttura BufferRoundSpeed

In questa struttura sono memorizzati tutti i messaggi ricevuti dal peer, che devono essere inoltrati ai suoi vicini (i peer del *fan-out*). Non vengono memorizzati nel BufferRS: i messaggi con *Age* maggiore o uguale a quella prestabilita; i messaggi di Information2, BootstrapPong2 e QueryHits che non necessitano di inoltro dopo la ricezione.

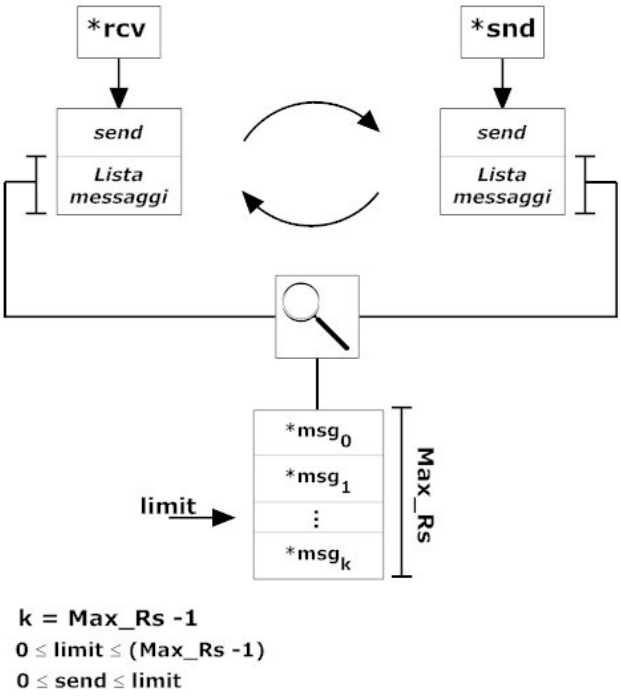


Figura 5.11: La struttura BufferRoundSpeed

La *figura* 5.11 descrive la struttura BufferRS, questa consta di due code di descrittori *msg: rcv* che contiene i descrittori di messaggi ricevuti nel *roundSpeed* corrente e *snd* che contiene i descrittori di messaggi, ricevuti nel *roundSpeed* precedente, che devono essere inviati. Entrambe le code hanno una cardinalità massima pari a *Max_Rs.*

All'inizio di ogni *roundSpeed* vengono eseguite le seguenti operazioni:

- I descrittori *msg* nella coda *snd* sono eliminati insieme ai messaggi che identificano,
- La coda *rcv* si svuota passando tutti i suoi descrittori *msg* a *snd* .

Il BufferRS è strutturato in modo da permettere ai processi, preposti alla ricezione e all'invio di messaggi, di lavorare contemporaneamente, cosa impossibile se si fosse utilizzata una solo coda. Il descrittore *send* è utilizzato per tenere traccia dei descrittori *msg* il cui messaggio è stato inviato, mentre *limit* identifica l'ultima posizione della coda occupata da un descrittore *msg*.

5.1.9 La struttura BufferRoundGossip

La struttura BufferRG è utilizzata dal processo roundGossip per memorizzare i messaggi di Informations2 da inviare ai peer del *fan-out*. Come è possibile notare dalla *figura* 5.12 la cardinalità massima della strut-

tura è *Max_fanout,* la stessa del BufferFanout. Il descrittore *limit* è usato per la localizzazione dell'ultima posizione occupata da un messaggio nella struttura.

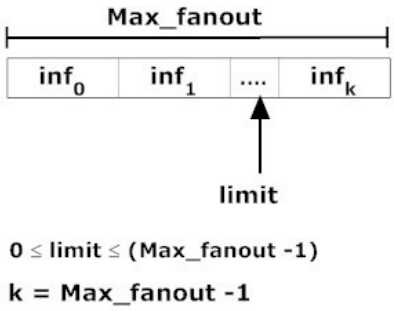


Figura 5.12: La struttura BufferRoundGossip

5.1.10 Le strutture di sincronizzazione

Le strutture dati che sono state presentate non sono in grado di gestire la coerenza dei dati in un ambiente in cui più processi lavorano simultaneamente. Per preservare la coerenza dei dati nelle strutture è necessario introdurre delle strutture ausiliare atte alla sincronizzazione dei processi. L'algoritmo *2Rounds* oltre all'uso dei semafori binari per sincronizzare i vari processi usa due strutture di sincronizzazione [18].

La struttura Two Rooms

Questa struttura di sincronizzazione è usata per eliminare il problema della s*tarvation* tra i processi che attendono per accedere a una data risorsa. L'idea è utilizzare due *turnstiles* per creare due stanze di attesa, dette anche *waiting rooms*, prima della sezione critica (ossia prima dell'accesso alla risorsa). La struttura di sincronizzazione lavora in due fasi. Durante la prima fase il primo *turnstile* è aperto ed il secondo chiuso, per far accumulare i processi nella seconda stanza. Durante la seconda fase, il primo *turnstile* è chiuso, in modo che nessun nuovo processo possa accedere alla seconda stanza, ed il secondo è aperta in modo da far accedere i processi della seconda stanza alla sezione critica. Anche se i processi che attendono nella seconda stanza possono essere di numero arbitrario, ad ognuno di essi è garantito l'accesso alla sezione critica prima dei processi presenti nella prima stanza.

La struttura Readers-Writers (writer-priority)

Questa struttura di sincronizzazione permette a più processi, che devono eseguire un operazione di lettura *processi lettori*, l'accesso alla sezione critica. Se un processo deve eseguire un operazione di scrittura, *processo scrittore*, questi avrà priorità rispetto ai *processi lettori*. Sia la *starvations* tra i processi della stessa tipologia lettori/scrittori che il *deadlock* sono scongiurati. L'idea è di definire due *token* uno in

scrittura T_w ed uno in lettura T_r che saranno usati per sincronizzare le due diverse tipologie di processi. Un processo lettore per entrare nella sezione critica controllerà che il T_w non sia stato preso, e se lo troverà prenderà il T_r. Il T_r viene rilasciato dall'ultimo processo lettore che lascia la sezione critica. Un processo scrittore per entrare nella sezione critica prenderà il T_w e T_r.

5.2 Famiglie di processi

Illustriamo le quattro famiglie di processi introdotte all'inizio del capitolo.

5.2.1 La famiglia di processi Informations

La famiglia Informations è costituita dal processo primario *dispatcherInformations* e dall'insieme dei suoi sotto-processi detti *helperInformations*.

5.2.1.1 DispatcherInformations

Nella fase di boostrap si preoccupa una volta avviato di creare i suoi sotto-processi, che terrà in vita per tutta la durata dell'algoritmo 2Rounds.
Il processo *dispatcherInformations* resta in ascolto, sulla sua porta, delle richieste di connessione, che provengono dai peer vicini. Per la gestione delle connessioni entranti, dovranno competere tra loro i sotto-processi *helperInformations*. La porta gestita in ascolto dal *dispatcherInformations* è usata solo per la ricezione dei messaggi di Informations2. Maggiori dettagli sul processo primario saranno forniti nel seguito della trattazione, dove vedremo che questa famiglia di processi è implementata attraverso lo schema *leader-follower* con *prethreading* e *locking* per l'*accept*.

5.2.1.2 HelperInformations

Gli *helperInformations* si occupano della ricezione ed elaborazione dei messaggi di Informations2, il loro comportamento è mostrato in *figura* 5.13

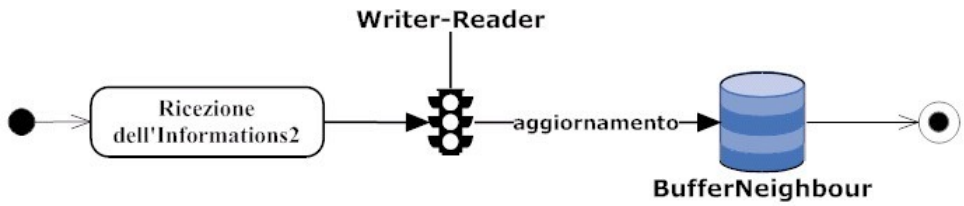


Figura 5.13: Il sotto-processo helperInformations

Una volta ricevuto il messaggio di Informations2, il sotto-processo richiede il *lock* in scrittura sulla struttura di sincronizzazione del BufferNeighbour. Ottenuto il *lock* aggiorna la struttura rilascia il *lock* e conclude. Finito il lavoro relativo al suddetto messaggio, il sotto-processo ritorna a competere per la gestione

di una nuova connessione.

5.2.2 La famiglia di processi Message

La famiglia Message è costituita dal processo primario *dispatcherMessage* e dall'insieme dei suoi sotto-processi detti *helperMessage*.

5.2.2.1 DispatcherMessage

Nella fase di boostrap si preoccupa una volta avviato di creare i suoi sotto-processi, che terrà in vita per tutta la durata dell'algoritmo I.
Il processo *dispatcherMessage* resta in ascolto, sulla sua porta, delle richieste di connessione, che provengono dai peer vicini. Per la gestione delle connessioni entranti dovranno competere tra loro i sotto-processi *helperMessage*. La porta gestita in ascolto dal *dispatcherMessage* è usata per la ricezione di tutti i messaggi tranne quelli di Informations2. Maggiori dettagli sul processo primario saranno forniti nel seguito della trattazione, dove vedremo che questa famiglia di processi è implementata attraverso lo schema *leader-follower* con *prethreading* e *locking* per l'*accept*.

5.2.2.2 HelperMessage

Gli *helperMessage* si occupano della ricezione ed elaborazione di tutti messaggi tranne quelli di Informations2, il loro comportamento è dinamico dipende dalla tipologia di messaggio che ricevono.

Ricezione di un BoostrapPing2

Nella *figura* 5.14 viene mostrato il comportamento dell'*helperMessage* dopo la ricezione di un messaggio di BootstrapPing2.

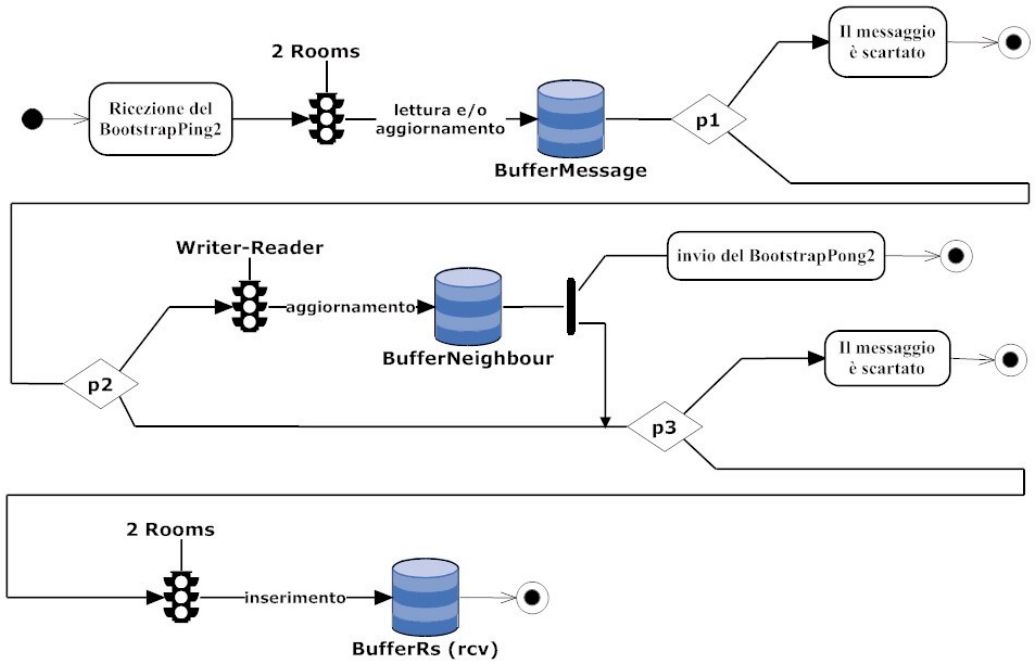


Figura 5.14: L'helperMessage dopo la ricezione del BootstrapPing2

Una volta ricevuto il messaggio di BootstrapPing2, il sotto-processo richiede il *lock* sulla struttura di sincronizzazione del BufferMessage. L'*helperMessage* ottenuto il *lock* controlla l'*ID Descriptor* del suo messaggio, se questo non risulta presente nella struttura aggiorna la struttura e rilascia il *lock* .
Se il messaggio risultava già ricevuto, a seguito del controllo sull'*ID Descriptor*, allora il sotto-processo scarta il messaggio e ritorna a competere per la gestione di una nuova connessione.
Nel caso in cui il messaggio non sia stato ricevuto precedentemente, con probabilità P_i (dipendente dall'*Age*) viene aggiornata la struttura BufferNeighbour (ottenendo il *lock in scrittura* dalla struttura di sincronizzazione relativa al BufferNeighbour ad operazione conclusa rilasciandolo) e si invia un BootstrapPong2 (sempre con probabilità P_i). Poi si procede al controllo dell'*Age* del messaggio, il quale viene scartato se risulta avere *Age* uguale o maggiore al valore massimo consentito. Se il messaggio supera con successo anche il controllo dell'*Age*, viene inserito nel BufferRS (coda *rcv*), ottenendo sempre prima il *lock* dalla struttura di sincronizzazione relativa al BufferRS e rilasciandolo ad operazione conclusa conclusione. Finito il lavoro relativo al suddetto messaggio, l'*helperMessage* ritorna a competere per la gestione di una nuova connessione.

Ricezione di un BoostrapPong2

Nella *figura* 5.15 viene mostrato il comportamento dell'*helperMessage* dopo la ricezione di un messaggio di BootstrapPong2.

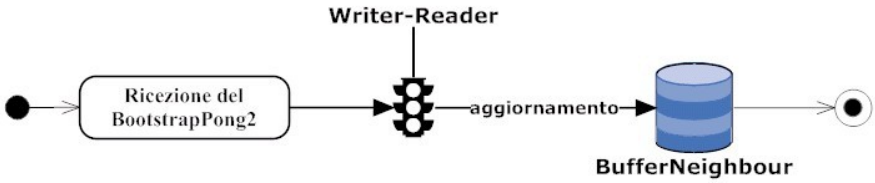


Figura 5.15: L'helperMessage dopo la ricezione di un BootstrapPong2

Una volta ricevuto il messaggio di BootstrapPong2, il sotto-processo richiede il *lock in scrittura* sulla struttura di sincronizzazione del BufferNeighbour. Ottenuto il *lock* aggiorna il BufferNeighbour, rilascia il *lock* e conclude. Finito il lavoro relativo al suddetto messaggio, il sotto-processo ritorna a competere per la gestione di una nuova connessione.

Ricezione di un Bye

Nella *figura* 5.16 viene mostrato il comportamento dell'*helperMessage* dopo la ricezione di un messaggio di Bye.

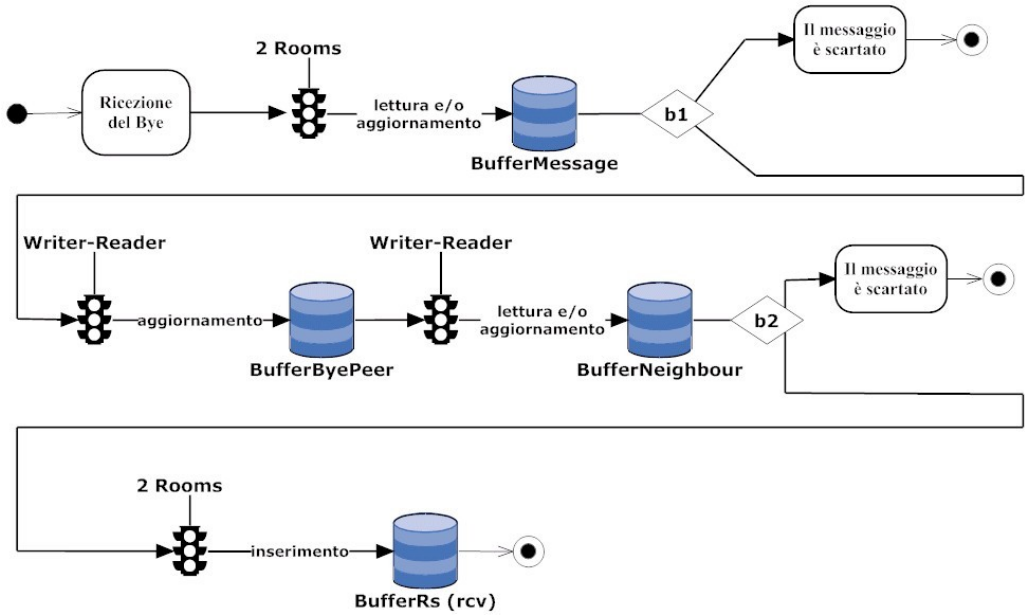


Figura 5.16: L'helperMessage dopo la ricezione di un Bye

Una volta ricevuto il messaggio di Bye, il sotto-processo richiede il *lock* sulla struttura di sincronizzazione del BufferMessage. L'*helperMessage* ottenuto il *lock* controlla l'*ID Descriptor* del suo messaggio se questo non risulta presente nella struttura BufferMessage lo inserisce e rilascia il *lock* .
Se il messaggio risulta già ricevuto, a seguito del controllo sull'*ID Descriptor*, allora il sotto-processo scarta il messaggio e ritorna a competere per la gestione di una nuova connessione.
Nel caso in cui il messaggio non sia stato ricevuto precedentemente, si richiede il *lock in scrittura* sulla

struttura di sincronizzazione del BufferByePeer. L'*helperMessage* ottenuto il *lock* inserisce il descrittore contenuto nel proprio messaggio nel BufferByPeer e rilascia il *lock* . Si procede con l'eventuale eliminazione del descrittore di *membership* del Bye dalla struttura BufferNeighbour, ottenendo sempre prima il *lock* in scrittura sulla struttura di sincronizzazione e rilasciandolo a conclusione. Viene controllato l'*Age* del messaggio, il quale viene scartato se risulta avere *Age* uguale o maggiore al valore massimo consentito. Se il messaggio supera con successo il controllo dell'*Age* viene inserito nel BufferRS (coda *rcv*), ottenendo sempre prima il *lock* sulla struttura di sincronizzazione relativa al BufferRS e rilasciandolo a conclusione. Finito il lavoro relativo al suddetto messaggio, l'*helperMessage* ritorna a competere per la gestione di una nuova connessione.

Ricezione di un Query

Nella *figura* 5.17 viene mostrato il comportamento dell'*helperMessage* dopo la ricezione di un messaggio di Query.

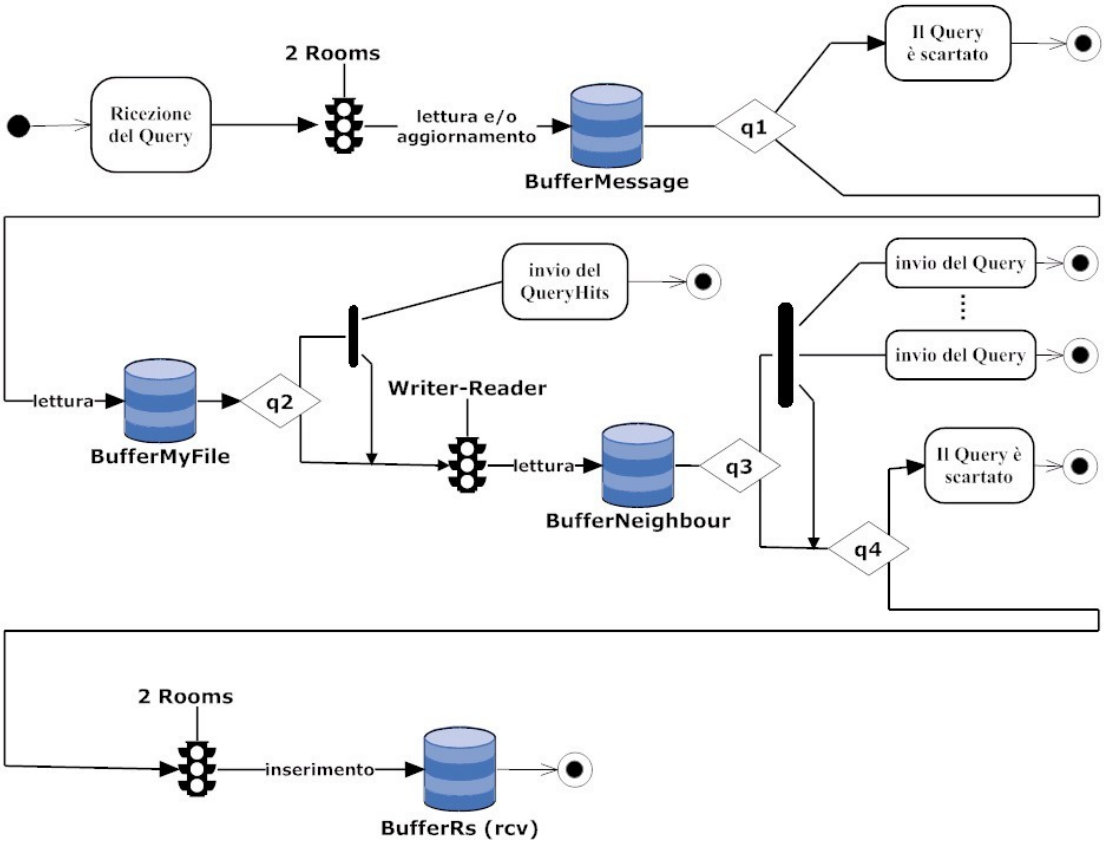


Figura 5.17: L'helperMessage dopo la ricezione di un Query

Una volta ricevuto il messaggio di Query, il sotto-processo richiede il *lock* sulla struttura di sincronizzazione del BufferMessage. L'*helperMessage* ottenuto il *lock* controlla l'*ID Descriptor* del suo messaggio se questo non risulta presente nella struttura aggiorna la aggiorna e rilascia il *lock* .

Se il messaggio risulta già ricevuto, a seguito del controllo sull'*ID Descriptor*, allora il sotto-processo scarta il messaggio e ritorna a competere per la gestione di una nuova connessione.
Nel caso in cui il messaggio non sia stato ricevuto precedentemente, si controlla nel BufferMyFile se si condivide la risorsa, richiesta nel Query. Se la ricerca ha dato esito positivo, si invia un QueryHits al peer identificato dai campi del messaggio Query.
Si effettua la ricerca della risorsa anche nel BufferNeighbour, ottenendo prima il *lock* in lettura dalla struttura di sincronizzazione e rilasciandolo a conclusione dell'operazione. Se la ricerca ha dato esito positivo si invia un Query ad ogni *neighbour* del BufferNeighbour che possiede la risorsa, il messaggio di Query che viene inviato ha i campi ip e porta uguali a quelli della Query ricevuta dall'*helperMessage*.
Viene controllato l'*Age* del messaggio ricevuto dall'*helperMessage*, il quale viene scartato se risulta avere *Age* uguale o maggiore al valore massimo consentito. Se il messaggio supera con successo il controllo dell'*Age* viene inserito nel BufferRS (coda *rcv*), ottenendo sempre prima il *lock* dalla struttura di sincronizzazione e rilasciandolo a conclusione. Finito il lavoro relativo al suddetto messaggio, l'*helperMessage* ritorna a competere per la gestione di una nuova connessione.

Ricezione di un QueryHits

Nella *figura* 5.18 viene mostrato il comportamento dell'*helperMessage* dopo la ricezione di un messaggio di QueryHits.

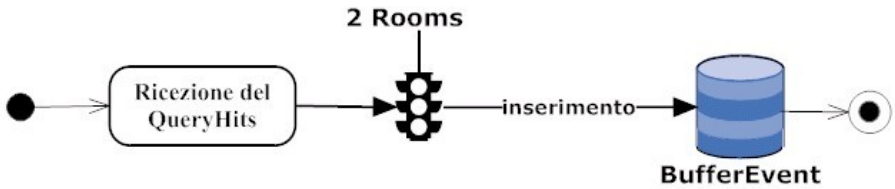


Figura 5.18: L'helperMessage dopo la ricezione di un QueryHits

Una volta ricevuto il messaggio di BootstrapPing2, il sotto-processo richiede il *lock* sulla struttura di sincronizzazione del BufferEvent. Ottenuto il *lock* inserisce nella struttura le informazioni contenute nel messaggio e rilascia il *lock* e conclude. Finito il lavoro relativo al suddetto messaggio, il sotto-processo ritorna a competere per la gestione di una nuova connessione.

5.2.3 La famiglia di processi Round

La famiglia di processi Round è costituita principalmente dal processo primario *roundGossip*, dal processo secondario *roundSpeed* e dagli *helperRS* (i sotto-processi del roundSpeed).

5.2.3.1 RoundGossip

Il processo *roundGossip* gestisce la diffusione delle informazioni tramite l'invio di messaggi di Informations2, che crea personalmente; inoltre si preoccupa dello "svecchiamento" del BufferMessage e del BufferByePeer. Un'altra funzionalità chiave del processo primario è la sincronizzazione temporanea del processo-secondario, infatti in un ciclo di *roundGossip* vengono eseguiti vari cicli del processo *roundSpeed*. I dettagli sul funzionamento del *roundGossip* sono mostrati in *figura* 5.19.

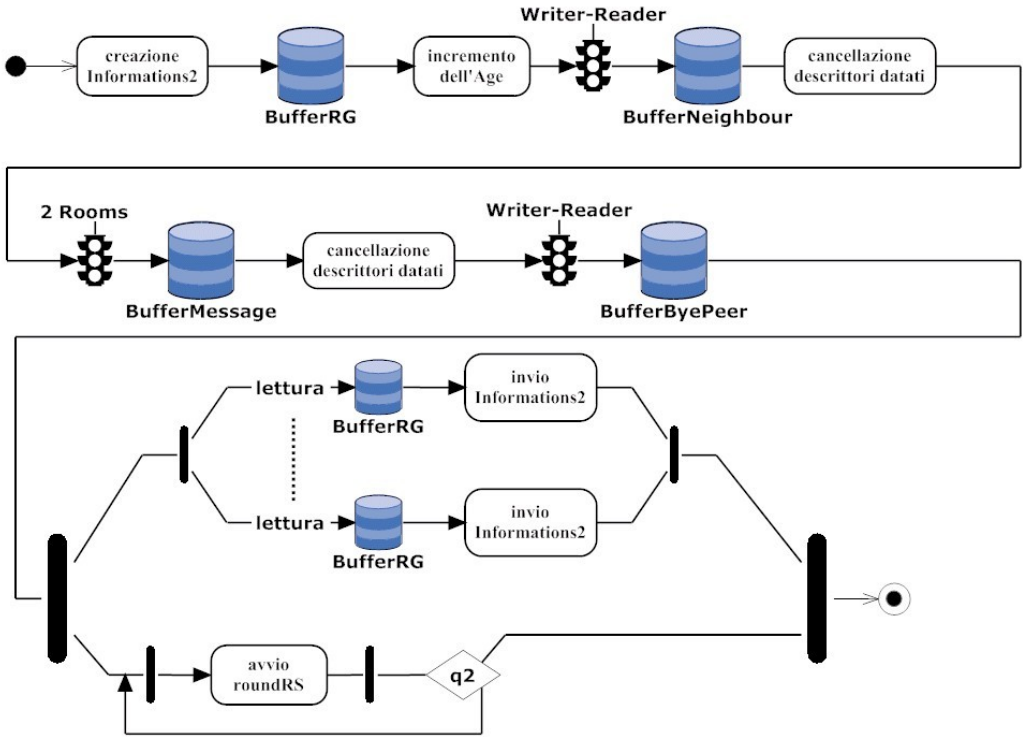


Figura 5.19: Dettagli sulle funzionalità del processo roundGossip

All'inizio di ogni ciclo il *roundGossip* crea un numero di messaggi di Informations2 (pari al numero di peer presenti nel BufferFanout) che vengono memorizzati nel BufferRG. Procede con l'incremento dell'*Age* nel BufferNeighbour, per effettuare questa operazione necessita del *lock in scrittura* sulla struttura di sincronizzazione del BufferNeighbour che viene rilasciato alla conclusione dell'operazione. Elimina i descrittori datati dal BufferMessage e dal BufferByePeer ottenendo i rispettivi *lock* dalle relative strutture di sincronizzazione e rilasciandoli a conclusione. Crea e avvia i sui sotto-processi gli *helperRg* a cui compe-

te l'invio dei messaggi di Informations2. Procede con la creazione e l'attesa di conclusione del processo *roundSpeed*, questa procedura viene eseguita un numero *n* di volte. Infine aspetta che i sotto-processi *helperRg* abbiano terminato e conclude il suo ciclo.

5.2.3.2 RoundSpeed

Il processo secondario *roundSpeed* è utilizzato per l'invio dei messaggi i cui descrittori *msg* sono residenti nel BufferRS (coda *snd*). Ad ogni suo ciclo scambia le code dei descrittori *rcv* e *snd* (sono puntatori), crea un insieme di sotto-processi che entreranno in competizione per l'invio dei messaggi nella coda *snd (*ex *rcv).* Ricordo inoltre che prima della scambio delle code, quella associata a *snd* viene resettata, poiché i messaggi da questa referenziati sonno stati inviati nel ciclo precedente. I dettagli sul funzionamento del *roundSpeed* sono mostrati in *figura* 5.20.

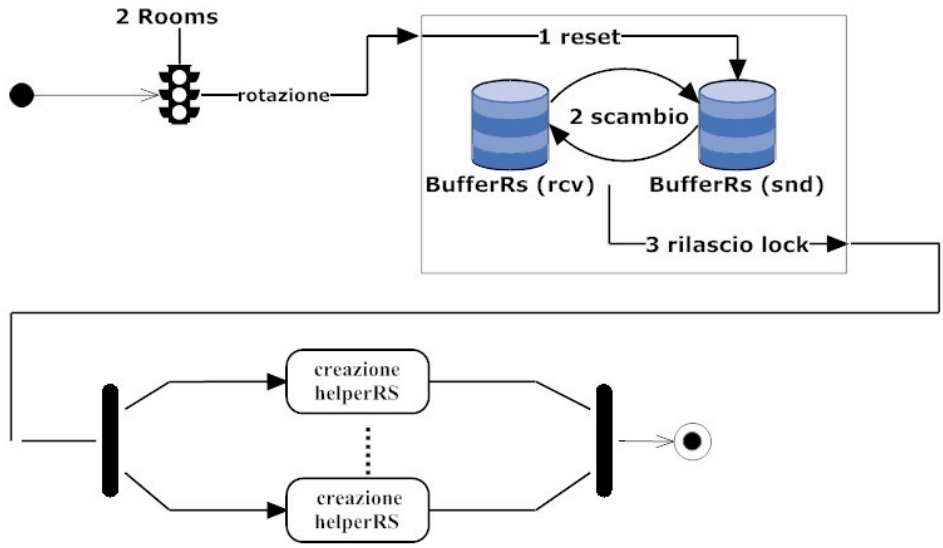


Figura 5.20: Dettagli sulle funzionalità del processo roundSpeed

All'inizio di ogni ciclo il *roundSpeed* richiede il *lock* sulla struttura di sincronizzazione associata al BufferRS. Ottenuto il *lock* procede con l'operazione di rotazione già descritta nella trattazione, finita questa rilascia il *lock* e procede alla creazione e all'avvio dei suoi sotto-processi *helperRS.* Infine il *roundSpeed* attende la terminazione dei suoi sotto-processi e conclude il ciclo.

5.2.3.3 HelperRS

I sotto-processi *helperRS* come detto precedentemente sono utilizzati dal *roundSpeed* per l'invio dei messaggi, si contendono l'accesso alla coda *snd*, per prelevare da questa i descrittori *msg*, cosi da ottenere il messaggio da inviare. Infine creano dei processi figli, di cardinalità uguale a quella del *fan-out,* necessari all'invio del messaggio ad ognuno dei peer identificati nel BufferFanout. Il funzionamento dei sotto-pro-

cessi *helperRS* è mostrato in *figura* 5.21.

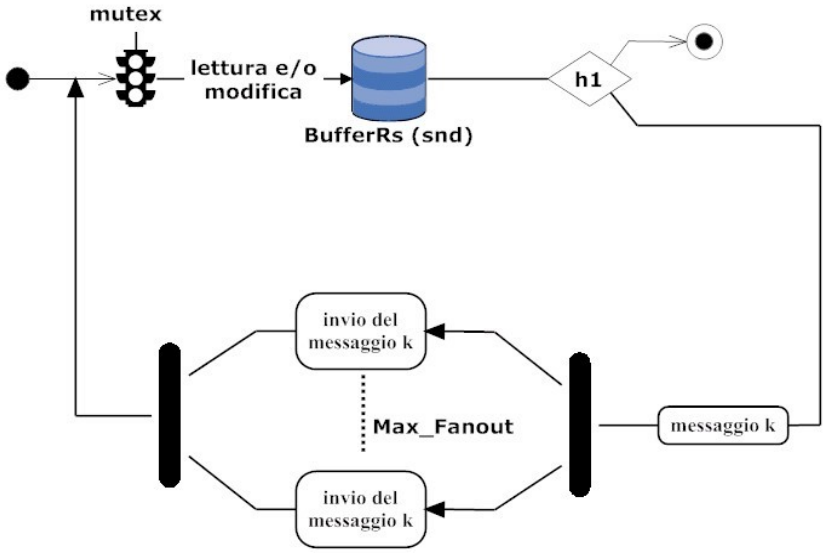


Figura 5.21: Dettagli sulle funzionalità del sotto-processo helperRS

Dalla *figura* 5.21 è possibile notare che il ciclo di vita del sotto-processo termina quando non ci sono più descrittori, di messaggi non inviati, nel BufferRS (coda *snd*).

5.2.4 La famiglia di processi *I/O*

La famiglia di processi I/O è costituita dal processo primario *I/O input* e dal processo secondario *I/O output*, il loro compito consiste nell'integrare le richieste utente con le funzionalità dell'algoritmo, quindi sono l'interfaccia utente nella fase di *round work*.

5.2.4.1 I/O input

Il processo primario *I/O input* permette all'utente di sottoscrivere le proprie richieste sulle risorse che desidera ottenere. Un'altra sua funzionalità importante è rappresentata dalla creazione del processo *I/O output*. Il funzionamento del processo *I/O input* è mostrato in *figura* 5.22.

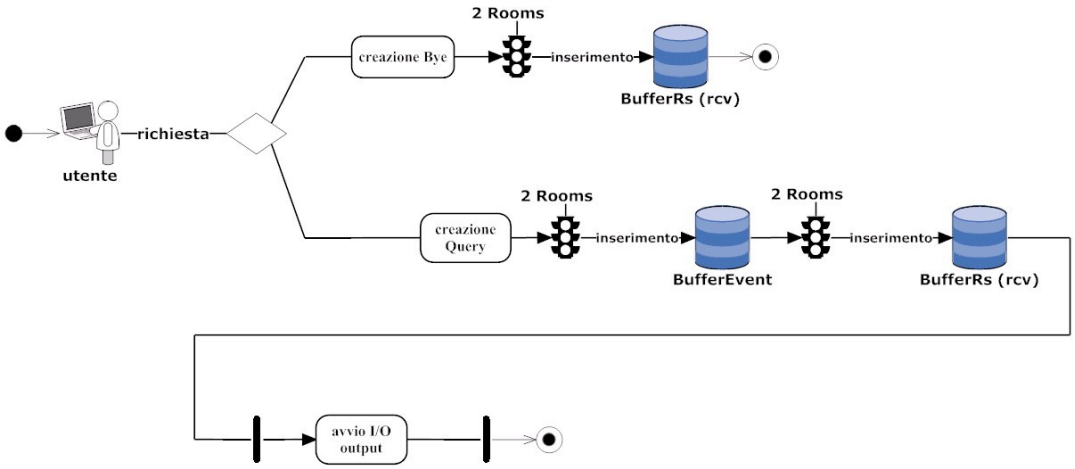


Figura 5.22 Dettagli sulle funzionalità del processo I/O input

All'inizio di ogni ciclo il processo *I/O input* attende la richiesta sottoscritta dall'utente che può essere di due tipi: una richiesta di risorsa oppure una richiesta di uscita.

1. Se il processo *I/O input* riceve una richiesta di risorsa la elabora e genera un messaggio di Query. Dopo aver generato il messaggio inserisce il relativo *Descriptor ID* nel BufferEvent, ottenendo prima il *lock* sulla struttura di sincronizzazione associata e rilasciandolo a fine operazione. Prosegue con l'inserimento del descrittore *msg* (relativo al Query generato) nel BufferRs (coda *rcv*), prima di eseguire l'operazione richiede e infine rilascia il *lock* sulla struttura di sincronizzazione relativa al BufferRS (*rcv*). Infine crea il processo *I/O output*, ne aspetta la terminazione e conclude il suo ciclo.
2. Se il processo *I/O input* riceve una richiesta di uscita crea immediatamente un messaggio di Bye e richiede il *lock* sulla struttura di sincronizzazione del BufferRS (coda *rcv*). Dopo aver ottenuto il *lock* inserisce il descrittore *msg* relativo al Bye nel BufferRs (coda *rcv*), rilascia il *lock* e termina.

5.2.4.2 I/O output

Il processo secondario I/O output permette all'utente di visualizzare i risultati di localizzazione della risorsa da lui richiesta. Il funzionamento del processo I/O output è mostrato in figura 5.23.

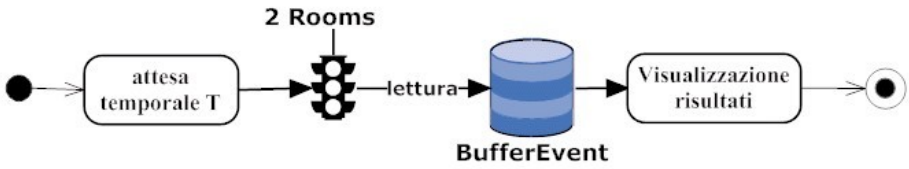


Figura 5.23 Dettagli sulle funzionalità del processo I/O output

L'I/O output attende per un tempo T le QueryHits inviate dai peer, che hanno ricevuto la relativa Query (quella generata dal processo I/O input). Finita l'attesa richiede il lock sulla struttura di sincronizzazione relativa al BufferEvent, legge i risultati provenienti dai messaggi di QueryHits e rilascia il lock. Infine visualizza i risultati di localizzazione sulla risorsa, ottenuti dalla lettura del BufferEvent.

6 L'applicazione 2Rounds

6.1 Descrizione

L'applicazione *2Rounds* è basta sul protocollo 2.0 ed è stata realizzata in linguaggio C[19, 20, 21], per un SO di tipo UNIX.
Permette, grazie alla creazione di una *overlay network* non strutturata, la localizzazione delle risorse richieste dall'utente. L'*overlay network* è altamente dinamica ed è creata con le informazioni contenute nei messaggi di rete, che i vari peer del sistema si scambiano. L'architettura di sistema è decentralizzata pura. La mancanza di un *server di discovery* per le risorse, elimina un possibile *collo di bottiglia,* mentre l'uso del protocollo *2Rounds* 2.0 riduce il *flooding* nella rete generato dai vari messaggi *multicast*.
L'applicazione è stata resa *user-friendly* grazie all'implementazione di un menù testuale che permette all'utente di modificare: il numero di file condivisi e il percorso della cartella dove inserire i file scaricati. Inoltre per testare il funzionamento del *2Rounds* ho realizzato un *server di bootstrap* che permette all'applicazione di funzionare sia in rete che in locale. Naturalmente per realizzare questa versione test (quella contente il server bootstrap), si sono dovute apportare alcune modifiche strutturali al protocollo quali: l'introduzione di un ulteriore famiglia di processi i *processi di Bootstrap,* aventi come processo primario il *dispatcherBootstrap*, utilizzato solo per le connessioni server-peer; e l'introduzione di un alcuni nuovi tipi di messaggi. Nel seguito del capitolo presenteremo brevemente il *dispatcherBootstrap* e le sue funzionalità.
Un'altra differenza tra il protocollo e l'applicazione riguarda il comportamento dell'*I/O input*, nel protocollo si è potuto vedere che il processo gestiva un solo processo secondario di *I/O output*. Questa limitazione non avrebbe consentito all'utente di effettuare una ricerca di risorse, se quella eventualmente sottoscritta precedentemente non si fosse conclusa. L'applicazione permette di sottoscrivere un determinato numero di ricerche contemporaneamente.
Anche il processo di *roundGossip* ha subito una modifica, come abbiamo visto precedentemente il nostro processo primario prima di creare ed attendere il *roundSpeed* un numero prestabilito di volte, effettua delle operazioni su alcune strutture che non sono in relazione con il processo secondario (esclusa l'operazione di creazione di aggiornamento del BufferFanout). Quindi è stato possibile introdurre un ulteriore ciclo di *roundSpeed.* Dopo l'aggiornamento del BufferFanout, il *roundGossip* crea il *roundSpeed* e continua le sue operazioni sulle strutture, alla conclusione delle quali verifica se il processo secondario da lui creato è terminato e se ancora in esecuzione lo attende.

6.1.1 Il processo exit

Nell'algoritmo *2Rounds* non è trattato il problema della terminazione, infatti per permettere alle 4 famiglie di processi di terminare in modo corretto è stato introdotto un nuovo processo ausiliario l'*exit*. Il compito del processo *exit* è favorire la terminazione degli altri processi, senza provocare problemi di sincronizzazione all'applicazione. Inoltre il compito di creare il messaggio di Bye e inserirlo nel BufferRS (coda *rcv*), che era del processo *I/O input* è stato assegnato al processo *exit*. Una volta ricevuta la richiesta di terminazione il processo *I/O input* crea il processo *exit* e termina.

Descrivo le operazioni in sequenza temporale eseguite dal processo di *exit*:

- Crea e inserisce nel BufferRS il Bye,
- Verifica che il processo *I/O input* sia terminato
- Attende un tempo *T* per permettere ai peer di ricevere il messaggio di Bye.
- Invia un segnale per la terminazione ai processi: *disptacherInformations*, *dispatcherMessage* e *roundGossip*.
- De-alloca tutte le strutture.
- Restituisce il controllo al *main* e termina.

Una volta restituito il controllo al *main*, questi attenderà la terminazione del processo *exit*, solo dopo questa operazione l'applicazione terminerà.

6.1.2 Le librerie

Per lo sviluppo dell'applicazione *2Rounds* mi sono servito dell'API del socket di Berkley per quanto riguarda la programmazione di rete [22, 23], mentre ho usato la programmazione *multi-threading* ed i semafori per implementare la parte concorrente [24, 25, 26].

Di seguito mostro alcune delle librerie usate dall'applicazione:

- **pthread.h:** Mette a disposizione tutto l'occorrente per la gestione dei thread nell'ambiente UNIX
- **semaphore.h:** La libreria implementa i semafori di Dijkstra.
- **arpa/inet.h:** Definisce i prototipi delle funzioni per manipolare gli indirizzi IP.
- **netinet/in.h:** Definisce la struttura degli indirizzi Internet.
- **sys/socket.h:** Definisce i simboli che iniziano con PF_ e AF_ ed i prototipi delle funzioni di rete.

6.2 Lo schema leader-follower

La soluzione adottata per l'implementazione del *dispatcher* e del *pool di processi helper*, nelle famiglie di processi Message e Informations, segue lo schema *leader-follower* con *prethreading* e *mutex* sull'*accept.*

Prethreading 2Rounds

Il *thread* principale (*dispatcher*) si mette in ascolto di eventuali connessioni e genera un pool di *threads* (*helpers*) di dimensione statica.

Leder-follower con mutex 2Rounds

Ogni *helpers* cerca di ottenere il *lock* sul *mutex* per lanciare l'*accept* sul *socket* di ascolto. Al più un *thread helper* (detto *leader*) si può trovare in ascolto, mentre gli altri (detti *follower*) sono in attesa di poter accedere alla sezione critica per il *socket* d'ascolto. Il *mutex* garantisce il funzionamento corretto dello schema anche nei sistemi UNIX dove l'*accept* è una funzione di libreria (*kerne*l UNIX derivati da SYSTEM V).

6.2.1 Il thread dispatcher

Nel frammento di codice sono mostrate le operazioni basilari che deve eseguire un thread dispatcher.

```
1.  if ( (  listensd = socket ( AF_INET , SOCK_STREAM , 0 ) ) < 0 ) {...}
2.  if ( bind ( listensd , ( struct sockaddr * ) &addr , sizeof ( servaddr ) ) < 0 ) {...}
3.  if ( listen ( listensd ,  BACKLOG ) < 0 ) {...}
4.  pthread_t   threads[N_THD];
5.  for ( t = 0 ; t < N_THD ; t++ )
6.  {
7.      int  rc ;
8.      rc = pthread_create ( &threads[t] , NULL , func_helper , ( void *) listensd );
9.      if ( rc ){...}
10. }
```

Il *thread dispatcher*:

- Crea il *socket* **listensd** (1) adibito a trasmissioni tipo TCP/IP.
- Effettua la **bind( )** su **listensd** per avvisare il Sistema Operativo di inviargli i dati provenienti

dalla rete, relativi all'ip **addr.sin_addr.s_addr** e alla porta **addr.sin_port** (2).

- Mette il *socket* **listends** in ascolto di eventuali connessioni. **BLACKLOG** identifica il numero di connessioni che possono essere accettate e messe in attesa di essere servite nella coda di richieste di connessione (3).
- Infine crea **N_THD** *threads helpers* a cui passa il socket **listends** (4 – 10).

6.2.2 I threads helpers

Nel frammento di codice sono mostrate le operazioni basilari che deve eseguire un thread helper.

```
1.  for ( ; ; )
2.  {
3.      sem_wait ( sem_accept );
4.          if ( ( connsd = accept( listensd, ( struct sockaddr* )&cliaddr, ( socklen_t * )&len) ) < 0 ) {...}
5.      sem_post ( sem_accept );
6.      recv ( connsd, buff, sizeof ( struct  Messaggio ), 0 );
7.      // elabora il messaggio
8.      close ( connsd ) ;
9.  }
```

Il *threads helper*:

- Chiede il *lock* sul semafono **sem_accept** (3), appena lo ottiene predene dalla coda di *backlog* la prima connessione completa sul *socket* **listends** con la chiamata **accept( )**. Se la coda di *backlog* è vuota rimane bloccato sulla chiamata **accept( )**, finché non viene accettata una nuova connessione (4). Infine rilascia il *lock* (5).
- Riceve il messaggio (6) nella struttura puntata da **buff** di tipo **struct Messaggio**, utilizzando il *socket* **connsd** restituito dalla chiamata **accept( )**. Elabora il messaggio ricevuto (7) ed infine chiude il *socket* **connsd**.
- Ritorna a compete con altri *thread helper* (1) per l'utilizzo dell'**accept( )**.

6.3 L'implementazione del server di boostrap

Per testare l'applicazione *2Rounds* è stato creato un *server di bootstrap* che permette di utilizzare l'applicazione sia in rete che in locale.

6.3.1 I messaggi

Per permettere il dialogo tra il peer ed il *server di boostrap*, sono stati creati dei nuovi messaggi: 0*x*05 ServerbootPing e 0*x*80 ServerbootPeer. Oltre ai due nuovi *Payload Descriptor*, l'*header* non ha subito modifiche. Di seguito vengono mostari i *payload* dei due messaggi.

0*x*05 ServerbootPing

Porta	IPv4
2 byte	*4 byte*

E' usato dal peer per richiedere al *server di bootstrap* delle informazioni di *membership* con cui iniziare a costruire la sua *view*. L'*Age* dell'*header* è posta a zero. Il server risponde con un ServerbootPeer. Se il peer non riesce a costruire la sua *view,* con le informazioni contenute nel messaggio di ServerbootPeer, trascorso un determinato tempo *T* invierà di nuovo un messaggio di ServerbootPing al server. Questa operazione viene ripetuta fino a quando il peer non riesce a creare la sua *view.* Il messaggio ServerbootPing è caratterizzato dai seguenti campi di *payload*:

- **Porta:** La prima porta utilizzata dal nodo che genera il ServerbootPing, rappresentata in formato *network order.*
- **IPv4:** L'indirizzo ip del nodo che genera il ServerbootPing, rappresentato in formato *network order.*

0*x*80 ServerbootPeer

Neighbour 1		.....	Neighbour n		#Neighbour
Porta	IPv4	.....	Porta	IPv4	n
2 byte	*4 byte*	.....	*2 byte*	*4 byte*	*2byte*

E' usato dal *server di bootstrap* per inviare delle informazioni al peer che gli ha inviato un ServerbootPing. L'*Age* dell'*header* è posta a zero. Il messaggio ServerbootPeer è caratterizzato dai seguenti campi di

payload:

- **Porta** *(neigbour)***:** La prima porta utilizzata dal *neighbour*, rappresentato in formato *network order*.
- **Ipv4** *(neigbour)***:** L'indirizzo ip del *neighbour*, rappresentato in formato *network order*.
- **# Neighbour:** il numero di peer inseriti nel messaggio.

6.3.2 Integrazione nell'applicazione 2Rounds

Introdotto il *server di bootstrap* ed i messaggi usati per la comunicazione con i vari peer, rimane da analizzare l'integrazione del server nell'applicazione *2Rounds*.

Il peer

Nella sotto-fase *recupero nodi* (presente nella *fase bootstrap*) il peer contatta con un messaggio di ServerbootPing il *server di bootstrap,* che risponde con un messaggio di ServerbootPeer, contenente i peer da contattare per creare la *view*. Il peer invia ai peer ricevuti nel messaggio di ServerbootPeer un messaggio di BootstrapPing2 e attende gli eventuali BootstrapPoong2.

A questo punto si possono avere due situazioni:

- I messaggi inviati dagli altri peer del sistema sono sufficienti al peer per creare la sua *view.* La sotto-fase di *recupero nodi* termina e la *fase di bootstrap* procede con i successivi passi.
- Il peer non riesce a costruire la propria *view*. Attende un tempo *T* e ripete la sotto-fase di *recupero nodi*.

La figura 6.1 mostra in dettaglio la *sotto-fase recupero nodi*.

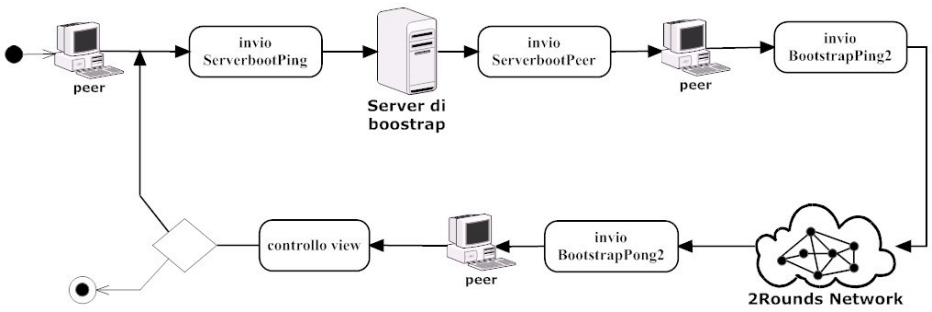


Figura 6.2: La sotto-fase recupero nodi

Il server di bootstrap

Il *server di boostrap* è composto principalmente: da una struttura BufferNodi per l'inserimento dei peer che vogliono accedere alla rete *2Rounds* e da una famiglia di processi per l'invio e la ricezione dei messaggi. La famiglia di processi è composta dal processo primario *dispatcherServerboot* e dai sotto-processi *helperServerboot*. La soluzione adottata per l'implementazione di questa famiglia di processi segue lo schema *leader-follower* con *prethreading* e *mutex* sull'*accept.* La figura 6.2 illustra il comportamento di un *helperServerboot* dopo la ricezione di un messaggio ServerbootPing.

Figura 6.3: Il sotto-processo helperServerboot

L'*helperServerboot* che riceve un messaggio di ServerbootPing, chiede il *lock in lettura* sulla struttura di sincronizzazione per verificare se il peer è già presente nel BufferNodi e finita l'operazione di lettura rilascia il *lock*. Se il peer non risulta presente nel BufferNodi, l'*helperServerboot* chiederà il *lock in scrittura* e lo inserirà, rilasciando il *lock* al termine dell'operazione. In entrambi i casi il sotto-processo invierà un ServerbootPeer al peer e concluderà la gestione della connessione, ritornando a competere per l'*accept*.

7 Conclusioni

Come mostrato in *figura* 7.1 il traffico P2P cresce in modo drammatico. E' stato stimato [CacheLogic , 2006] che il traffico medio P2P su un ISP può essere del 70% [27].

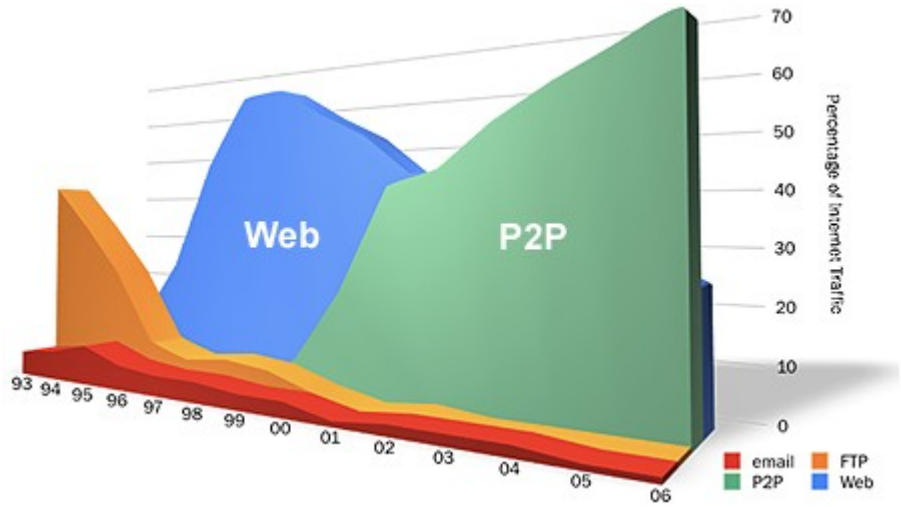


Figura 7.1: Stima [CacheLogic] del traffico P2P sugli ISP

In questo scenario risulta evidente, come anche la sola riduzione del *flooding* dei messaggi di *membership* e di *discovery* delle risorse, apporti un notevole beneficio alla diminuzione del traffico di rete P2P.
Il protocollo 2Round come ho mostrato nel corso della trattazione cerca di preservare le caratteristiche del sistema P2P decentralizzato puro, cercando di limitare il *flooding* dei messaggi di informazione.

Al fine di verificare i miglioramenti apportati dal protocollo 2Rounds ho deciso di implementare un simulatore basato sul protocollo di rete 1.0 . Lo scopo del simulatore è calcolare la percentuale di copertura della *vista limitata* in base ai valore di parametrizzazione forniti in input.

Il simulatore 1.0

Vengono simulati *rg* cicli di *roundGossip* terminati i quali avremo un *overlay* creata connettendo ogni nodo con i propri vicini presenti nel suo fan-out.
Una volta creata l'*overlay* è possibile calcolare la *vista limitata* di un nodo *n*, in questo modo:

- Considerato un nodo *n* della rete creiamo l'insieme N_n, contente tutti i nodi raggiungibili da *n* con un massimo di *r* passi.

- Dall'insieme N_n, *creiamo* un insieme $L_n = N_n \cup N_{n,view}$, dove $N_{n,view}$ è l'insieme costituito dall'unione di tutte le *view* degli elementi di N_n. L'insieme L_n costituisce la *vista limitata* del nodo n.

Considerato *R* l'insieme di tutti i nodi della rete, possiamo definire $p(n) = \frac{card(L_n)}{card(R)} \cdot 100$ ossia la *percentuale di copertura della vista limitata* di *n.*

Se prendiamo un campione *c* di nodi n_i, scelti in modo casuale, definiamo P_c come la percentuale *di copertura della vista limitata* con $P_c = \frac{1}{c}\sum_{j=1}^{c} p(n_j)$.

Il valore di *Pc* rappresenta l'output del simulatore 1.0 .

Di seguito riporto i parametri di input del simulatore:

- **F:** *fan-out* dei nodi.
- **R:** raggio della *vista limitata*.
- **N:** cardinalità della rete.
- **num_neigbour:** numero di descrittori di *membership* presenti in un messaggio di Informations1.
- **max_view:** cardinalità massima della *view* di un nodo.
- **C:** numero di nodi a cui è stata calcolata la *p(n).*
- **NRG:** numero di cicli di *roundGossip;*

Saranno mostrate due simulazioni entrambe eseguite da un calcolatore dual core 2GHz con 3GB di RAM.

Simulazione I

input:

- **F =** 5
- **R =** 6
- **N** = 100.000
- **num_neigbour** = 10
- **max_view** = 1750
- **C** = 500
- **NRG** = 10

output:

P_c = 99.14 %

Simulazione II

input:

- **F** = 5
- **R** = 6
- **N** = 500.000
- **num_neigbour** = 20
- **max_view** = 5000
- **C** = 1000
- **NRG** = 10

output:

P_c = 90.071642%

Possiamo notare che la percentuale di copertura di rete nel primo caso è del 99%, un buon risultato se confrontato con quello del protocollo *Gnutella 0.4* che si aggira intorno al 19-20% in condizioni ideali. Nel secondo caso si è deciso di aumentare il numero di nodi della rete senza modificare i parametri **F** e **R** ottenendo un discreto risultato il 90% di copertura, ottimo se confrontato a quello di Gnutella 4% circa in condizioni ideali.
Se analizziamo il *flooding* nella rete possiamo notare che il protocollo Gnutella per un solo messaggio di ping inviato genera un traffico di circa 19.5K messaggi. Mentre i nodi del sistema 2Rounds generano ciascuno **F** messaggi di Informations1, ogni ciclo di *roundGossip*, contenenti **num_neigbour** descrittori.

Dai test effettuati si può concludere che l'algoritmo 2Rounds diminuisce significativamente il carico di rete dovuto al *flooding* dei messaggi inviati in *multicast*.

8 Bibliografia

[1] Napster. http://en.wikipedia.org/wiki/Napster

[2] Gnutella. http://en.wikipedia.org/wiki/Gnutella

[3] Fast Track. http://en.wikipedia.org/wiki/FastTrack

[4] SETI@home. http://setiathome.berkeley.edu/

[5] GIMPS. http://www.mersenne.org/

[6] AIM: AOL Instant Messenger. http://dashboard.aim.com/aim

[7] ICQ. http://www.icq.com/

[8] V. Cardellini. *Slide del corso di Sistemi Distribuiti*, 2008.

[9] L. Ricci. *Slide del corso di Sistemi P2P*, 2007.

[10] Lawerence Lessing. *Peer to Peer: Harnessing the Power of DisruptiveTechnologies*.

[11] *The Gnutella Protocol Specication v0.4* (Document Revision 1.2).

[12] Enrico Boldrini. *Il procollo Gnutella.*

[13] L. Torrero. *Algoritmi per protocolli P2P*, 2006.

[14] Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, Laurent Massoulié. *Epidemic Information Dissemination in Distributed Systems*, IEE Comuputer Society 2004.

[15] Andrew S. Tanebaum, Maarten Van Steen. *Sistemi Distribuiti, seconda edizione*, Prentice Hall.

[16] S.Pizzutilo, A.Bianchi. *Slide del corso di Architetture e Modellazione dei Sistemi Distribuiti*, 2008-09.

[17] Patrick Th. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, Anne-Marie Kermarrec. *Lightweight Probabilistic Broadcast*. ACM Trans. Comput. Syst. 21(4): 341-374, 2003.

[18] Allen B. Downey. *The Little Book of Semaphores*, version 2.1.5 .

[19] Peter a. Darnell, Philip E. Margolis. *Manuale di programmazione C*, McGraw-Hill.

[20] Harvey M. Deitel, Paul j. Deitel. *C corso completo di programmazione*, seconda edizione, Apogeo

[21] Brian W. Kernighan, Denise M. Ritchie. *C programming Language*, seconda edizione,Prentice Hall.

[22] W. Richard Stevens. *Unix Network Programming Vol. 1 e 2*, seconda edizione,Prentice Hall.

[23] Simone Picardi. *Guida alla programmazione in Linux*, 8 novembre 2007.

[24] Alessandro Dal Palù. *Slide del corso di Laboratorio di Sistemi Operativi*, www.unipr.it/~dalpalu

[25] Raffaele Quitadamo. *Slide seminario Posix Threads: l'evoluzione dei processi UNIX*.

[26] Cotroneo Domenico, De Carlini Ugo, Mazzeo Antonino. *Slide del corso di Sitemi Operativi*.

[27] P4P. http://www.openp4p.net/