POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Fornitura di connettività ottimizzata in un'infrastruttura DTN service-oriented

Relatore
prof. Fulvio Risso

Correlatore:
ing. Gabriele Castellano

Candidato
Matteo Ottolini

Aprile 2020

Dobbiamo avere il coraggio di seguire il nostro cuore e la nostra intuizione. In qualche modo, essi sanno che cosa vogliamo realmente diventare. Tutto il resto è secondario.

Ringraziamenti

I ringraziamenti da fare, arrivati a questo punto, sono moltissimi; vorrei cominciare con il Professor Fulvio Risso, che mi ha permesso di partecipare a questo progetto il quale sicuramente ha migliorato le mie conoscenze sotto tutti i punti di vista e Gabriele Castellano che mi ha aiutato sulla validazione dell'elaborato e ascoltato le mie lamentele. Un grande riconoscimento va a Calogero Carrabbotta che ogni giorno mi ha sostenuto ad aiutato a portare avanti questo progetto, un collega ma soprattutto amico. Un ringraziamento anche a Riccardo Loti che mi ha permesso di entrare nel mondo di Tierra e a tutte le persone che ci lavorano per la loro accoglienza e riguardo nei miei confronti.

Un ringraziamento, sicuramente il più importante, alla mia bellissima e numerosa famiglia, ma soprattutto a mia madre e mio padre che hanno sempre creduto in me e permesso con il loro sforzo il raggiungimento di questo importante obbiettivo. Il loro insegnamento è quello che più di ogni altro è riuscito a portarmi dove sono arrivato.

Ultimi, ma non ultimi, a tutti i numerosi amici che mi circondano e riempiono le mie giornate e senza i quali non sarebbe stato emotivamente facile concludere con serenità questo percorso. Chiedo scusa ai miei coinquilini per tutte quelle volte che li ho disturbati con rumori provenienti dalla mia stanza per via dei mancati obbiettivi progettuali quotidiani raggiunti, ma sono sicuro anche di avergli strappato numerosi sorrisi.

Grazie a tutti per avermi ascoltato nei momenti di sconforto e per la forza che mi avete dato, senza di voi non sarei riuscito.

Indice

Capitolo 1

Introduzione

La sempre crescente diffusione di dispositivi IoT, in concomitanza con la rapida evoluzione delle tecnologie di comunicazione wireless, introducono una serie di nuove sfide dal punto di vista del networking. È sempre più comune imbattersi in contesti in cui le entità coinvolte nella comunicazione sono in continuo movimento e caratterizzate, nella maggior parte dei casi, da risorse hardware limitate. In tali scenari, identificabili con il nome di "challenged networks", non è possibile fare affidamento ai paradigmi di comunicazione tradizionali. Le comunicazioni in Internet, ad esempio, si fondano sull'assunzione secondo cui, in ogni istante, è garantita l'esistenza di almeno un percorso end-to-end tra la sorgente e la destinazione del traffico, inoltre i link tendono a essere stabili. Questo non è assolutamente garantito nel contesto di una challenged network, che è invece caratterizzata da continue interruzioni o addirittura assenza di connettività per un periodo indefinito di tempo, perciò non è mai assicurato un percorso stabile tra sorgente e destinazione. Oltre che alla mobilità dei dispositivi, la connettività intermittente può essere dovuta a fattori intrinseci dell'ambiente in cui la challenged network è collocata, come la presenza di ostacoli che si interpongono tra i dispositivi atti a comunicare, oppure il verificarsi di fenomeni atmosferici e ambientali avversi. La connettività intermittente porta con sé una serie di ulteriori problemi, come il partizionamento della rete, ritardi lunghi e/o variabili, alto tasso di perdita d'informazioni, i quali rendono la comunicazione ancora più complessa. Gli scenari soggetti a tali problematiche sono molteplici e spaziano dall'ambito militare, all'interplanetario, alle reti di sensori nelle aree non dotate di alcuna infrastruttura di telecomunicazione, come ad esempio siti edili, agricoli o per l'allevamento del bestiame, in zone montuose o rurali. La maggior parte delle applicazioni IoT esistenti suppongono di operare su reti connesse, caratterizzate da ritardi minimi o perlomeno trascurabili. Poiché la modifica di tali applicazioni richiederebbe uno sforzo materialmente insostenibile, date le assunzioni precedenti, l'approccio più semplice è quello d'introdurre un framework comune

a tutti i nodi della rete, una sorta d'interfaccia, che permetta di far fronte alle esigenze tipiche delle challenged network.

Il paradigma del *Delay-Tolerant Networking* rappresenta una potenziale soluzione del problema. Il suo obiettivo principale è quello di garantire, con una buona probabilità, che un pacchetto giunga da sorgente a destinazione, nonostante la temporanea mancanza di un percorso completo tra esse. Il perseguimento di tale obiettivo è raggiunto mediante l'utilizzo di un meccanismo asincrono d'inoltro dei messaggi, che utilizza un approccio molto simile a quello adottato per la posta elettronica, noto con il nome di *store-and-forward message switching*. Secondo quest'approccio, un messaggio viene mantenuto localmente fin quando non risulta possibile consegnarlo direttamente a destinazione, oppure inoltrarlo a qualche altro nodo intermedio, ritenuto un potenziale next-hop verso la destinazione. Una DTN è, quindi, una rete formata da dispositivi fissi e mobili quali sensori e attuatori a bordo di veicoli, terminali utente, unità computazionali ai confini della rete, etc., dotati di una o più interfacce di rete in grado di comunicare tra di loro tramite connessioni opportunistiche, senza fare affidamento su un'infrastruttura. Una funzione fondamentale, in questo tipo di rete, è quella di permettere alle applicazioni presenti sui vari dispositivi mobili di scambiarsi messaggi, informazioni e usufruire dei servizi presenti sulla rete in modo ottimale, tenendo conto dei possibili ritardi di comunicazione.

L'obiettivo della tesi sarà quello di studiare e presentare diversi protocolli di routing esistenti in letteratura e adattare il migliore al framework IBR-DTN. Tra i vari protocolli di routing disponibili in letteratura, verrà in un primo momento implementato il più promettente. In seguito, verranno proposti e valutati dei miglioramenti allo scopo di adattare meglio l'algoritmo scelto alla particolare natura service-oriented dello scenario applicativo proposto da Tierra Telematics, andando di fatto a definire un algoritmo di routing che tiene conto dei SLA (Service Layer Agreements - Accordi sul Livello del Servizio). Verranno infine analizzati i risultati ottenuti e discusse le possibili direzioni per ulteriori miglioramenti e sviluppi futuri.

Capitolo 2

IBR-DTN

In questo capitolo verranno descritte le caratteristiche e la struttura del framework IBR-DTN, nel quale parte delle informazioni riportate sono tratte dall'elaborato di tesi *Comunicazione self-optimized tra dispositivi eterogeneamente connessi ad una rete con tolleranza ai ritardi* [1].

2.1 Delay/Disruption Tolerant Networking

Le *Delay/Disruption Tolerant Network* (DTN) definiscono un'architettura end-to-end capace di fornire connettività nelle cosiddette "challenged networks". Queste reti sono caratterizzate da connettività intermittente, nodi di tipologia eterogena e condizioni di rete molto diverse. Il concetto di Delay Tolerant Networking nasce nell'ambito delle comunicazioni interplanetarie, ma attualmente trova moltissime applicazioni in ambito commerciale, scientifico, militare e di servizio pubblico. I protocolli Internet tradizionali non riescono a fornire comunicazione efficientemente, in quanto le assunzioni sulla quale sono basati non sono valide per questa particolare tipologia di reti. Oggigiorno, infatti, è sempre più comune scontrarsi con scenari applicativi in cui i dispositivi che devono comunicare sono in movimento e operano a potenza limitata, questo può portare all'interruzione di un collegamento per la presenza di un ostacolo, oppure, in certe situazioni l'interruzione del link fisico al fine di preservare energia. La conseguenza di questi fenomeni di connettività intermittente, è un naturale partizionamento della rete.

In tali scenari, la comunicazione mediante i protocolli basati su IP è particolarmente inefficiente. Il protocollo IP, si basa sull'idea che in ogni istante esista un percorso end-to-end che colleghi sorgente e destinazione di un pacchetto. Questo non è assolutamente ipotizzabile in una "challenged network", che è invece caratterizzata da connettività intermittente, ritardi lunghi e/o variabili, alto tasso di

errori e asimmetria nelle trasmissioni. Basta pensare a TCP/IP, il suo utilizzo per comunicare all'interno di una rete instabile causerebbe un numero significativo di dati persi. Infatti, nel caso di un pacchetto che non possa essere inoltrato immediatamente, il TCP assumerà il congestionamento della rete, scarterà il pacchetto e proverà a ritrasmetterlo abbassando gradualmente la velocità di ritrasmissione, fino a chiudere la sessione nel caso di intermittenza troppo elevata.

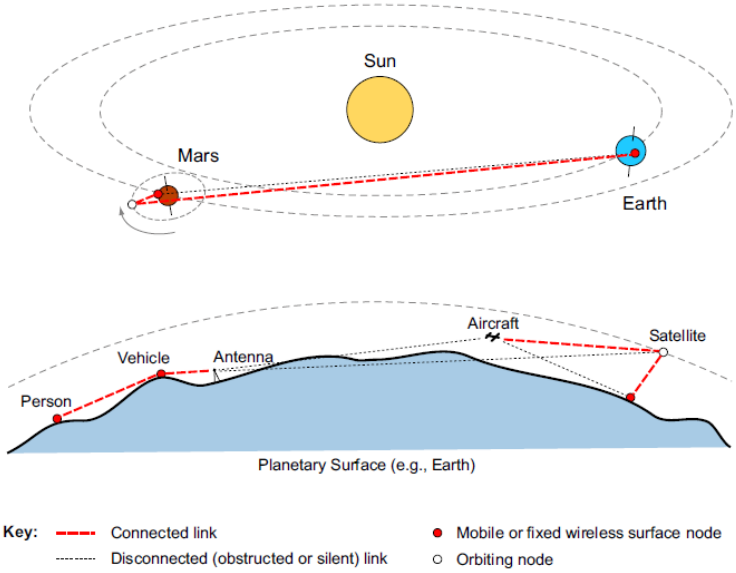


Figura 2.1. Esempi di contatti pianificati (comunicazioni interplanetarie) e opportunistici (comunicazioni sulla superficie terrestre)

Inoltre, parlando di connettività intermittente, è opportuno far distinzione tra i contatti pianificati e quelli opportunistici (figura 2.1). Lo scenario tipico dei contatti pianificati o *scheduled* è quello dello spazio, in cui i nodi si muovono su percorsi orbitali predicibili, tanto che è possibile prevedere o ricevere gli instanti in cui occuperanno le loro future posizioni e quindi organizzare le future sessioni di comunicazione. I contatti di tipo pianificato, perciò, richiedono la sincronizzazione temporale dell'intera DTN. Per contatti opportunistici, invece, si intendono i contatti tra un trasmettitore e un ricevitore in istanti non programmati. E' il caso di persone, veicoli, aerei o satelliti che potrebbero voler scambiare informazioni quando risultato in linea di vista e abbastanza vicini da poter comunicare usando la loro potenza, seppure limitata.

Per far fronte alle problematiche tipiche delle "challenged networks" e trarre beneficio dai contatti pianificati e/o opportunistici, le DTN utilizzano la tecnica dello

store-and-forward message switching. Secondo questo paradigma, analogo al meccanismo utilizzato per la posta elettronica, interi messaggi o frammenti di essi sono spostati dallo storage di un nodo a quello di un altro, lungo un percorso che potenzialmente conduce alla destinazione. Quando un nodo riceve un pacchetto, esso viene inoltrato immediatamente se possibile, oppure memorizzato localmente per essere trasmesso in futuro. Per questo motivo, ogni router DTN deve disporre di un supporto che permetta di memorizzare i messaggi per un tempo indefinito (un hard disk, ad esempio), garantendo la persistenza dell'informazione. Questo è in contrapposizione rispetto a quanto accade nei router IP, che utilizzano dei buffer di memoria per accodare i pacchetti in attesa di essere inoltrati, garantendone una persistenza dell'ordine dei millisecondi. E' necessario che lo storage sia persistente poiché alcuni link di comunicazione potrebbero essere non disponibili per lunghi periodi di tempo, nelle situazioni in cui venga richiesta la ritrasmissione di un messaggio oppure nel caso di un nodo che trasmetta e/o riceva i dati molto più velocemente di un suo vicino.

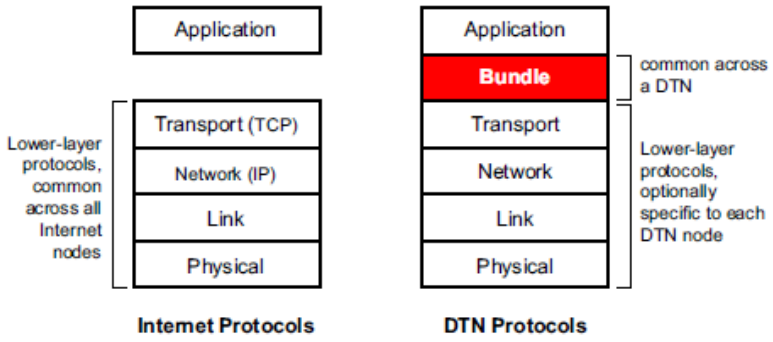


Figura 2.2. Confronto fra uno stack Internet (a sinistra) e uno stack DTN (a destra)

La DTN realizza una rete "overlay" introducendo un nuovo livello di astrazione, il *Bundle Layer*, che estende lo stack di rete dei nodi partecipanti alla DTN, ponendosi tra il livello applicativo e il livello trasporto. L'obiettivo principale di questo layer è quello di rendere i programmi applicativi agnostici rispetto ai livelli di trasporto utilizzati, favorendo la creazione di reti eterogenee. Due nodi che vogliono instaurare una comunicazione interagiranno con il Bundle Layer, senza preoccuparsi della natura dei protocolli utilizzati nei livelli inferiori. Il bundle layer sarà responsabile dell'instradamento di questi messaggi, detti appunto Bundle, da sorgente a destinazione. Le DTN utilizzano un modello non-conversazionale asincrono, in contrasto al meccanismo di comunicazione richiesta/risposta tipico della famiglia TCP/IP. I protocolli conversazionali, come il TCP, implicano lunghi RTT e spesso falliscono. Il Bundle Layer comunica tramite un protocollo non-conversazionale che

minimizza i round trips necessari a confermare le trasmissioni, rendendo opzionali gli acknowledgment.

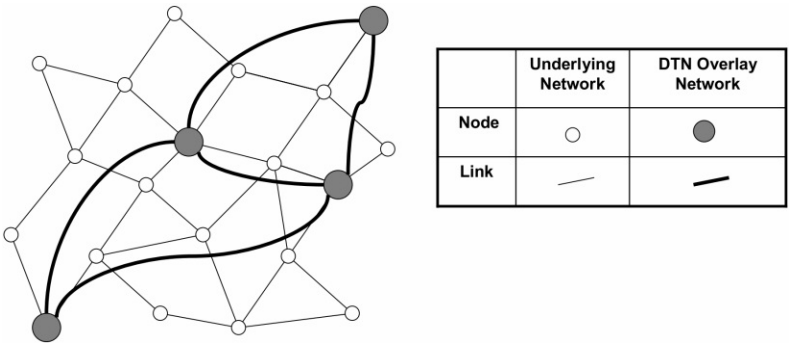


Figura 2.3. La rete DTN in overlay su un altro tipo di rete

La peculiarità di posizionarsi fra il layer di trasporto e il layer applicativo permette l'uso di DTN per creare dei proxy applicativi. Prendiamo come esempio applicazioni che girano su TCP/IP, esse tipicamente usano le API socket Berkeley, e non hanno accesso ai servizi di DTN. Inoltre se volessero usarli dovrebbero essere scritte in maniera da essere tolleranti ad interruzioni e ritardi, e potrebbero avere bisogno di numerosi scambi di messaggi per effettuare le proprie operazioni, come SMTP. Riscrivere le applicazioni per sfruttare le API, richiederebbe modifiche a tutte le applicazioni. L'altro uso che possiamo ipotizzare di DTN è quello di creare un Application Layer Gateway. Esso sarebbe un terminatore di protocollo, e prenderebbe le informazioni necessarie per ricreare lo stesso dialogo avuto con il client, così da riproporlo al server e ottenee la risposta desiderata.[2]

2.2 Bundle Protocol

Il *Bundle Protocol*[3] è un protocollo sperimentale, corrispondente allo Bundle Layer dell'architettura DTN, sviluppato all'interno del Delay Tolerant Networking Research Group (DTNRG) dell'IRTF.

2.2.1 Architettura

Nel contesto delle DTN, con il termine *bundle node* si indica un'entità capace di ricevere e trasmettere bundle. Secondo le specifiche del Bundle Protocol, un *bundle node* è concettualmente costituito da tre componenti fondamentali:

- **Bundle Protocol Agent (BPA)**: è il fornitore dei servizi del bundle protocol. Il modo con cui tali servizi sono offerti dipende dalla sua implementazione. Infatti, il BPA può essere implementato in hardware, come libreria condivisa tra più nodi su una singola macchina, come un processo(un demone) con cui i nodi su una o più macchine possono interagire tramite meccanismi di comunicazione tra processi o comunicazione di rete (Es. Socket Berkeley).

- **Convergence Layer Adapter (CLA)**: invia e riceve i bundle per conto del BPA, sfruttando i servizi offerti da un qualche protocollo di trasporto (Es. TCP, UDP, RFCOMM, etc). Il modo in cui il CLA gestisce la trasmissione dei bundle dipende dal protocollo che adottata al livello sottostante.

- **Application Agent (AA)**: utilizza i servizi del bundle layer per comunicare. L'AA è generalmente composto da due elementi, uno amministrativo e uno applicativo. L'elemento amministrativo costruisce e richiede la trasmissione di record amministrativi (status report e segnali di custodia) e processa i segnali di custodia ricevuti dal nodo. Tipicamente è integrato nell'implementazione del BPA. L'elemento applicativo, invece, costruisce, trasmette e processa i dati applicativi veri e propri e può essere implementato in software o in hardware. La comunicazione tra l'elemento applicativo dell'AA e il BPA avviene tramite l'interfaccia di servizio esposta da quest'ultimo. Un nodo che ha solo funzione di "router" può non avere un alcun elemento applicativo.

I principali servizi che un BPA dovrebbe fornire all'AA di un nodo sono i seguenti:

- registrazione di un nodo ad un endpoint;
- terminazione della registrazione;
- trasmissione di un bundle ad uno specifico endpoint;
- annullamento della trasmissione;
- consegna di un bundle ricevuto.

2.2.2 Incapsulamento

Il Bundle Protocol estende la gerarchia dell'incapsulamento realizzata dai protocolli Internet, semplicemente incapsulandoli senza alterarne i dati. La figura 2.4 mostra un esempio di incapsulamento dei protocolli TCP/IP.

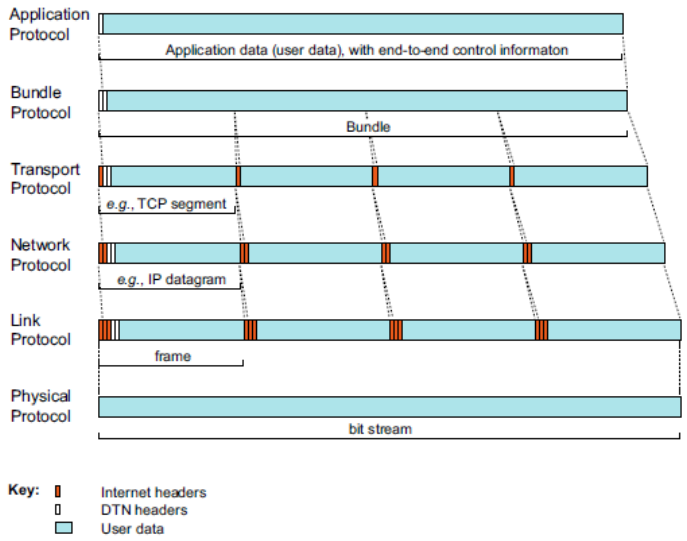


Figura 2.4. Incapsulamento dei protocolli TCP/IP nel Bundle Protocol

Nel caso di bundle troppo grandi, il bundle layer dovrebbe essere in grado di suddividere i messaggi in più frammenti, in maniera abbastanza simile a come il livello IP frammenta i propri pacchetti. In caso di frammentazione, è compito del nodo destinazione quello di riassemblare i frammenti nell'ordine corretto, in modo da ottenere il bundle originario.

2.2.3 Frammentazione

Per assicurarsi che i volumi di contatto siano usati pienamente e per evitare la ritrasmissione di Bundle parzialmente inoltrati, DTN offre un meccanismo di frammentazione. Due tipi di frammentazione sono previsti da DTN: proattiva reattiva. La frammentazione *proattiva* avviene su scelta arbitraria da parte di un nodo inoltrante il Bundle, sarà poi compito del nodo, o nodi, destinatari il riassemblaggio dei frammenti. La frammentazione *reattiva* avviene invece a seguito del non completo trasferimento di un bundle verso un nodo. Il nodo ricevente deciderà di trattare la porzione ricevuta come se fosse un frammento, e il mittente inviare la parte rimanente come se fosse un secondo frammento, direttamente al ricevente o passando da altri nodi se dovesse cambiare la topologia. Solo la frammentazione *proattiva* è di obbligatoria implementazione. La frammentazione a livello di Bundle Protocol è supportata grazie all'uso di un header che indica la lunghezza e l'offset del frammento rispetto al bundle originario, secondo un meccanismo simile a quello utilizzato in IP. I frammenti originati a partire dallo stesso bundle saranno identificati da sorgente, destinazione e tempo di creazione. Per un Bundle è inoltre

possibile richiedere la non frammentazione tramite uno dei Control Flag del primary Block. Inoltre tutti i blocchi prima del payload sono inseriti nel frammento di offset minore, e quelli dopo il blocco di payload sono inserite nel frammento di offset maggiore.

2.2.4 Indirizzamento

La sorgente e la destinazione di un bundle sono identificati da un *Endpoint Identifier* (EID). Ogni EID è conforme al formato Uniform Resource Identifier (URI) ed è composto da due parti: <scheme-name>:<scheme-specific part (SSP)>. La lunghezza di entrambi i campi non deve eccedere i 1023 bytes. Gli schemi di rappresentazione proposti per l'EID sono molteplici, ma convenzionalmente sono usati schemi conformi allo schema URI (Unified Resource Identifier), e caratterizzati da uno <scheme-specific part> suddiviso in due porzioni: la prima indicante il nodo, la seconda il *demux-token*, ovvero una singola applicazione. Uno degli schemi più diffusi è quello identificato dalla stringa `dtn`, che assume la forma `dtn://node/demux-token`. Mentre la presenza del `node` è obbligatoria, il `demux-token` può anche non esserci, come nel caso di bundle amministrativi diretti al BPA del nodo. Un EID tipicamente rappresenta un solo nodo (o meglio un applicazione su un solo nodo) ed è detto Singleton, ma può anche rappresentare un gruppo di nodi DTN, "multicast" o "anycast", gruppi contenti più nodi

2.2.5 Formato di un bundle

Ogni bundle è costituito dalla concatenazione di almeno due blocchi. Il primo blocco della sequenza, o *primary block*, contiene informazioni analoghe a quelle di un'intestazione IP, necessarie all'instradamento del bundle verso destinazione. Ogni bundle può avere un solo primary block, ma può essere seguito da una serie di blocchi per supportare le estensioni del protocollo, come il Bundle Security Protocol (BSP). Può esistere nei blocchi successivi al primo, al massimo un blocco di payload. La maggior parte dei campi hanno lunghezza variabile e utilizzano una notazione compatta detta *self-delimiting numerical values* (SDNVs) (rif.), estendibili e scalabile per una diversa varietà di protocolli di rete e dimensioni di payload.

Primary Block

Come si può notare in figura 2.5, oltre a versione, lunghezza del blocco, sorgente e destinzione, il primary block contiene una serie di informazioni tipiche del Bundle Protocol.

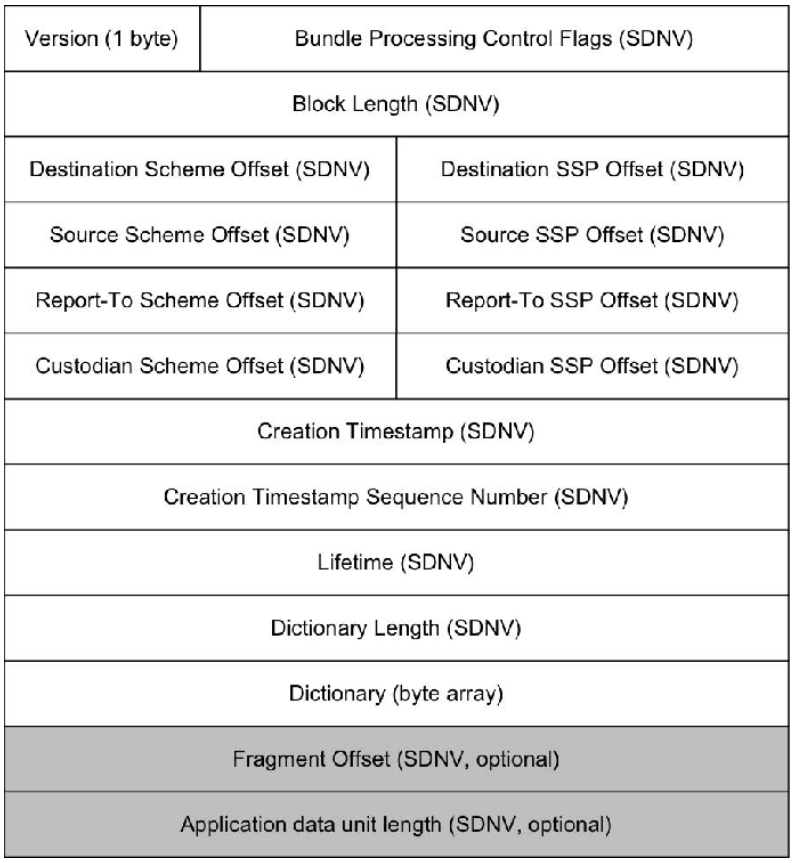


Figura 2.5. Formato del primary block di un bundle

Bundle Processing Control Flag I *Bundle Processing Control Flags* costituiscono una stringa di bit utili al processamento del bundle. Sono suddivisi in 3 categorie:

- General [0-6]: specificano informazioni di carattere generale sul bundle, ad esempio, se è regolare o amministrativo, lo stato di frammentazione, se la destinazione è un EID singleton, se sono richiesti acknowledgment o trasferimento di custodia

- Class of Service [7-13]: specificano la priorità del bundle, dove un valore elevato indica una priorità elevata, e altre informazioni utili al routing del pacchetto.

- Status Report [14-20]: specificano i report richiesti per questo bundle, ad esempio se è richiesto il reporti di consegna, di inoltro, di accettazione di custdia, ecc..

Priorità Dei bit "Class of Service" due vengono usati per definire la priorità del Bundle. Tipicamente vale solo tra bundle aventi la stessa sorgente, e può non essere rispettata nei confronti di bundle con sorgente diversa Tre sono i valori fino ad ora adoperati:

- Bulk: indicate bundle che devono essere spediti con il minimo dello sforzo, consegnati solo al termine della consegna di tutti bundle con la stessa sorgente e destinazione
- Normal: per i bundle che vengono spediti prima di quelli a priorità Bulk
- Expedited: per i bundle con priorità maggiore, da essere spediti prima di quelli con priorità Normal e Bulk

Endpoints Il primary block include quattro EID di lunghezza variabile, ognuno codificato tramite una coppia di offset: uno per lo schema, l'altro per la SSP. Tali offset non sono altro che puntatori alle stringhe rappresentanti gli EID memorizzate all'interno del dizionario posizionato successivamente nel blocco.

- Source: contiene l'endpoint dalla quale proviene il bundle
- Destination: è l'endpoint di destinazione del bundle
- Report-to: indica il nodo a cui inviare gli status report per eventi che coinvolgono il bundle
- Custodian: identifica l'ultimo nodo che ha accettato la custodia del bundle

Poiché gli EID costituiscono la maggior parte dei byte di overhead dovuti al Bundle Protocol, il dizionario rappresenta un meccanismo per ridurre la quantità di spazio necessario alla loro memorizzazione. Ad esempio, nel caso in cui l' EID sorgente e report-to coincidano, compariranno due riferimenti a tale EID, ma un'unica stringa all'interno del dizionario.

Tempo Altre informazioni significative per l'elaborazione di un bundle sono il *creation timestamp* e il *lifetime*. Il *creation timestamp* indica il tempo di creazione del bundle, espresso come il numero di secondi trascorsi dall'inizio dell'anno 2000 nel fuso orario UTC. Questo valore è calcolato l'istante in cui il BPA riceve la richiesta di trasmissione. Il *lifetime*, invece, rappresenta il tempo di vita del bundle, espresso come offset rispetto al tempo di creazione. L'uso del lifetime permette di eliminare i bundle in eccesso all'interno della rete, in quanto, ogni volta che un nodo riceve un bundle che ha terminato il suo tempo di vita, lo scarta. Poiché sia il *creation timestamp* che il *lifetime* utilizzano il tempo reale, è necessario che i nodi partecipanti alla DTN siano sincronizzati, seppure in maniera grossolana.

Altri blocchi

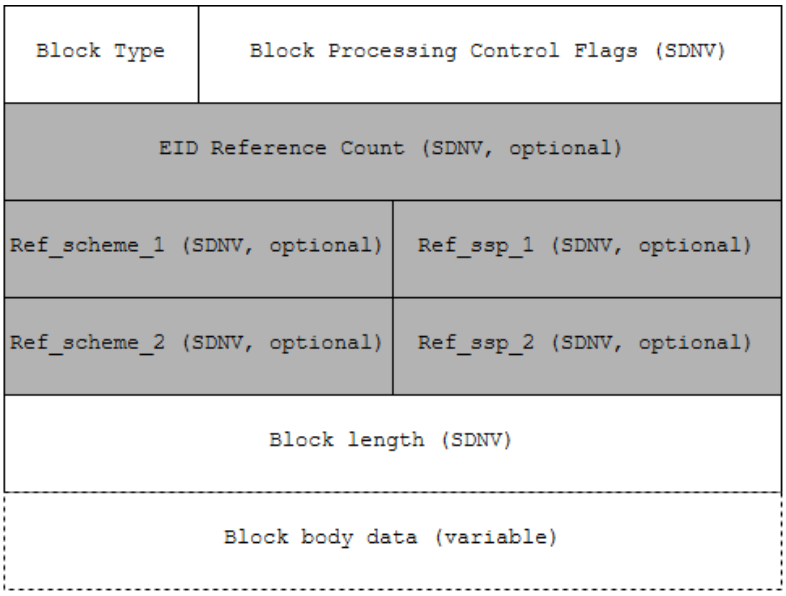


Figura 2.6. Formato generico di un blocco secondario di un bundle

Oltre al Primary Block all'interno di un bundle possono essere inseriti diversi altri blocchi. Come si può notare in figura 2.6, ognuno di questi blocchi è identificato dal *Block Type*, una stringa di 8 bit. Il valore '1' indica un blocco payload e un bundle ne può contenere massimo uno, i valori tra 192 e 255 sono ad uso sperimentale e privato, mentre i restanti sono riservati per usi futuri. Tutti i blocchi diversi da quello primario e dal payload sono detti extension block. Poi sono ci sono i flag di controllo del blocco, che danno indicazioni su come il blocco deve essere trattato. Infine completano il blocco il body e la loro lunghezza. E' inoltre possibile inserire il riferimento ad alcuni EID contenuti nel dizionario. Un contatore ne terra traccia e due puntatori, uno all'inizio dello schema e uno all'inizio dell'SSP nel dizionario per ogni entry.

Block Processing Control Flag I *Block Processing Control Flags* costituiscono una stringa di bit utili al processamento del blocco. E' un campo SDNV attualmente formato da 7 bit, indicanti alcuni particolari accorgimenti sul blocco. Per esempio abbiamo la possibilità di replicare il blocco in ogni frammento (in caso di frammentazione), indicare di scartare il blocco o l'intero bundle o inviare un report se non si è in grado di processare il blocco, se contiene degli EID-Reference, e sopratutto il flag che indica se è l'ultimo blocco del Bundle. Il bit di replicazione nei frammenti però non può essere settato a uno sui blocchi successivi a quello di payload

2.2.6 Affidabilità delle trasmissioni

Le DTN supportano meccanismi di ritrasmissione di dati persi e/o corrotti sia a livello dei protocolli di trasporto che a livello di Bundle Protocol. Tuttavia, poiché le DTN presentano tipicamente un'eterogeneità nei protocolli di trasporto utilizzati dai nodi, l'affidabilità deve essere realizzata a livello di Bundle Protocol, mediante un meccanismo di ritrasmissione da nodo a nodo detto *trasferimento in custodia*. Di base, quando il custode corrente di un bundle deve inoltrarlo, richiede il trasferimento in custodia e fa partire un timer di ritrasmissione. Se il BPA del nodo ricevente decide di accettare la custodia, invia un acknowledgement al mittente. Se non viene ricevuto alcun acknowledgement prima della scadenza del timer, il bundle viene ritrasmesso. Il valore del timer di ritrasmissione può essere distribuito ai nodi insieme alle informazioni di routing o calcolato localmente dai nodi stessi, secondo la loro esperienza passata. Il custode corrente di un bundle rappresenta quindi il nodo responsabile di mantenere il bundle in memoria persistente finché esso non viene ricevuto da un nuovo custode. Non è detto che uno nodo della DTN debba obbligatoriamente offrire il servizio di trasferimento in custodia. Un nodo potrebbe, ad esempio, rifiutare una richiesta di trasferimento in custodia per la mancanza di risorse disponibili, per una questione di policy o di implementazione. Tuttavia, in un contesto in cui si voglia minimizzare il numero di perdite, sarebbe opportuno che tutti i nodi utilizzassero il trasferimento in custodia, a patto che esistano le risorse di storage necessarie e che la frequenza di generazione dei bundle non superi quella di consegna, oltre che la capacità di buffering della rete. Dunque, il meccanismo di trasferimento in custodia, combinato con l'utilizzo di storage persistente sui nodi intermedi, permette di delegare la responsabilità di trasferimenti affidabili a porzioni della rete piuttosto che al mittente del bundle. Purtroppo, questo non è sufficiente a garantire l'affidabilità delle trasmissioni, ma solo a migliorarla. Un ulteriore passo può essere compiuto utilizzando il return receipt, un messaggio che conferma la consegna a destinazione di un bundle destinato al mittente dello stesso. Tuttavia, un'eccessiva quantità di bundle o frammenti di essi può portare ad un eccessivo consumo le risorse di storage disponibili, congestionando la DTN. In caso di congestionamento, un nodo può adottare diverse strategie: eliminare dallo storage le copie di bundle che hanno terminato il loro tempo di vita, attività che dovrebbe essere intrapresa comunque con regolarità, trasferire dei bundle ad altri, non accettare bundle con trasferimento in custodia, piuttosto che bundle regolari, eliminare bundle non scaduti, anche se il nodo ne è il custode. L'utilizzo di quest'ultima opzione è assolutamente sconsigliato, poiché chiaramente contraddittoria rispetto ai principi cardine delle DTN.

2.3 IBR-DTN Node

2.3.1 Introduzione

IBR-DTN è l'applicativo scelto per la realizzazione di una infrastruttura DTN che una volta installato su dispositivi ne permette l'inserimento in rete e gestisce la comunicazione via bundle. Modulare e leggera IBR-DTN è stata creata dal gruppo di ricerca sulle DTN del Technische Universität Braunschweig. Studiato per essere installato su sistemi embedded fornisce allo sviluppatore un framework per creare applicazioni DTN[4].

2.3.2 L'architettura

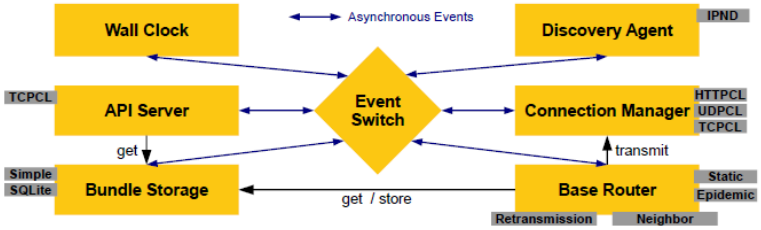


Figura 2.7. Architettura IBR-DTN

La versione di IBR-DTN per i sistemi operativi tradizionali è stata sviluppata in C++. Come si può notare in figura2.7, l'implementazione del bundle protocol di IBR-DTN è contraddistinta da un'organizzazione fortemente modulare, tale da permettere agli sviluppatori di estendere il software in maniera semplice e poco invasiva. Il Bundle Protocol Agent è implementato come processo demone ed espone una API basata su socket che le applicazioni possono contattare per interagire con il Bundle Layer. Di default l'API è disponibile alla porta TCP 4550 in formato testuale e binario. Per maggiori informazioni sulle funzionalità esposte si può fare riferimento alla documentazione[5].

Event Switch I moduli sono collegati in maniera flessibile e comunicano tra loro tramite un meccanismo basato su eventi, rendendo fondamentale l'*Event Switch*, incaricato di affidare la gestione dei singoli eventi ai sotto-moduli corrispondenti. Tutti i moduli possono ricevere o scatenare eventi per comunicare con le altre parti del software. Nell'implementazione attuale sono integrati una serie di eventi per notificare le operazioni di storage, la presenza e scomparsa di nodi nel vicinato, le operazioni di routing dei bundle, ecc.

Discovery Agent Un altro componente di vitale importanza è il *Discovery Agent*, responsabile di scoprire i nodi nel vicinato. Sotto l'ipotesi di voler far comunicare nodi IP, IBR-DTN utilizza un modulo che implementa il protocollo DTN IP Neighbor Discovery (IPND)[6]. Tale modulo rimane in ascolto di piccoli datagrammi UDP detti *beacon*, utilizzati dai nodi per annunciare la propria presenza ai vicini, e periodicamente si annuncia tramite i medesimi datagrammi. I *beacon* sono spediti ad un indirizzo IP multicast noto (e specificabile in configurazione) e contengono l'EID del mittente, per permettere a chi lo riceve di effettuare il binding tra EID e indirizzo IP del vicino.

Connection Manager e Convergence Layer Ad occuparsi della gestione della gestione delle connessioni con i nodi vicini e dell'invio e della ricezione di bundle è il modulo *Connection Manager*. Il *Connection Manager* a sua volta per l'implementazione del trasferimento di informazioni sfrutta diversi *convergence layer*. Come descritto dall'RFC 5050[3] sulle Delay Tolerant Network, sono i *convergence layer* a occuparsi della comunicazione tra due nodi. Ognuno di essi definisce un'interfaccia verso il livello di trasporto sottostante, permettendo il trasferimento di bundle astraendosi dai protocolli di livello inferiore. I convergence layer utilizzati dal sono specificati nella configurazione del demone. Attualmente IBR-DTN offre convergence layer per TCP/IP[7], UDP/IP, HTTP, IEEE 802.15.4 LoWPAN. Esiste anche un'estensione del TCP/IP CL per il supporto a TLS.

Bundle Storage Poiché le DTN sono basate sul paradigma store-and-forward, ogni nodo deve essere capace di memorizzare bundle per un certo periodo di tempo. In IBR-DTN l'interazione con lo storage è gestita da *Bundle Storage*, modulo che fornisce primitive per la lettura, cancellazione e memorizzazione dei bundle da/verso lo storage. Sono supportati diversi meccanismi di memorizzazione: in memoria RAM, su disco (file-system) e su basi di dati SQLite.

Base Router Il routing dei bundle è invece realizzato dal modulo *Base Router*, che si occupa di gestire il forwarding dei bundle che ha in carico. Il *Base Router* suddivide il proprio lavoro tra i diversi moduli di routing. Ognuno di essi implementa uno specifico algoritmo di routing DTN ed è agganciato al Base Router come una sorta di plugin. Tutti i moduli di routing sono notificati dal *Discovery Agent* al verificarsi di eventi legati al vicinato del nodo e dal *Bundle Storage* nel momento in cui un nuovo bundle giunge al demone. Il modulo di routing che si riterrà responsabile dell'inoltro del bundle contatterà poi il Connection Manager per attivare il convergence layer opportuno. IBR-DTN presenta moduli per il supporto al routing statico, epidemico e PRoPHET e grazie a questo lavoro di tesi MaxProp.

API server L'interazione con IBR-DTN, e quindi la parte più utile al completamento di questa tesi, è ottunuta tramite l'API server. L'API server espone su una interfaccia socket, configurabile tramite config file, un protocollo testuale con cui effettuare richieste al core dell'applicativo. I comandi da inviare devono seguire la specifica logica e sintassi dell'azione da eseguire, all'invio di un comando e all'inserimento di tutte le informazioni che esso richiede, si riceve sempre un response formato come $<status - code>$ $<message>$ *[Additional data]*, come mostrato in figura 2.8

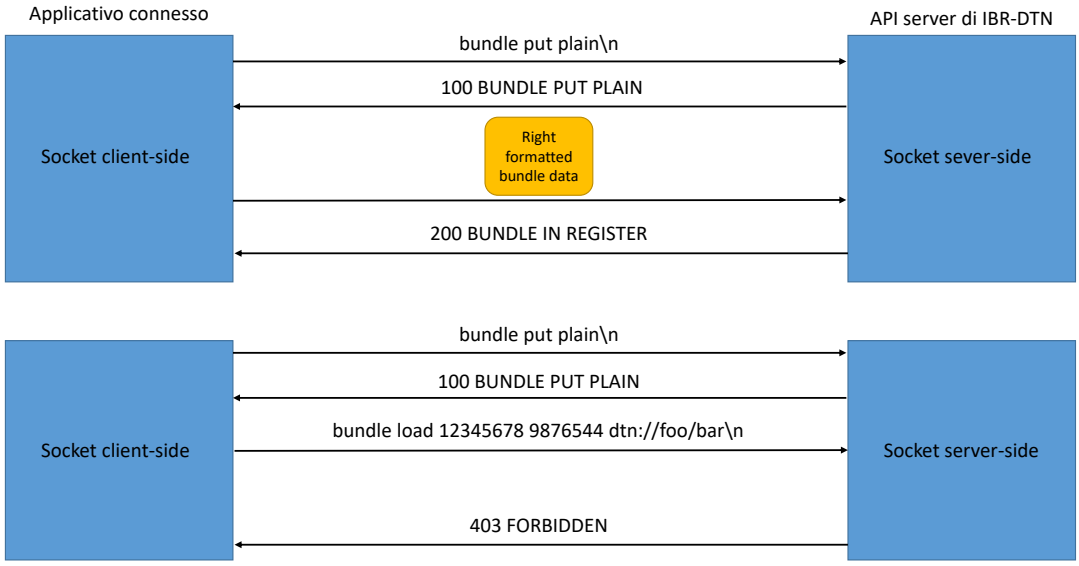


Figura 2.8. Interazione API Server

Le api disponibili sono visibili sul repository ufficiale di IBR-DTN [https://www.ibr.cs.tu-bs.de/projects/ibr-dtn/apidoc/0.12/api.pdf]

Service Discovery É un modulo creato per propagare i servizi presenti su ogni nodo, all'interno della rete DTN. Con servizi di intende la descrizione delle capacità di un'applicazione, residente sul nodo, che può utilizzare la rete DTN per inviare e/o ricevere dati. La Service Discovery permette a tutti i nodi la conoscenza di tale servizio grazie a messaggi multicast inviati ad un indirizzo DTN di gruppo. Grazie alla Service Discovery le applicazoni possono selezionare il nodo DTN che più si avvicina alle loro richieste ed iniziare ad inviare pacchetti verso di esso.

2.3.3 Installazione e avvio

In questo paragrafo sono illustrati i passaggi necessari all'installazione e avvio del demone IBR-DTN. Le procedure utilizzate sono valide per distribuzioni Linux, Debian e derivate (Raspbian).

Installazione Il primo passo consiste nell'installazione delle librerie necessarie. In realtà, l'installazione di alcune librerie elencate in seguito è opzionale, in quanto esse risultano utili nel momento in cui si aggiungano moduli opzionali.

```
$ apt-get install build-essential libssl-dev zlib1g-dev libsqlite3-dev libcurl4-gnutls-dev
    libdaemon-dev automake autoconf pkg-config libtool libcppunit-dev libnl-3-dev
    libnl-cli-3-dev libnl-genl-3-dev libnl-nf-3-dev libnl-route-3-dev libarchive-dev git
```

L'installazione e l'attivazione/disattivazione dei moduli opzionali di IBR-DTN ora prevede l'utilizzo di CMAKE. Come da prassi, dopo aver creato la cartella *build* in */ibedtn/ibrdtn*, si procederà a lanciare il comando **cmake ..** con l'opzione per attivare tutti i moduli che il sistema operativo può supportare. Quindi dopo aver clonato il repository, si esegue la configurazione, la compilazione e l'installazione dei sorgenti:

```
$ cd ibrdtn-repository/ibrdtn
$ mkdir build
$ cd build
$ cmake .. -DBUILD_ALLFEATURES=ON
$ make
$ make install
$ ldconfig
```

Nel caso in cui si voglia avere un controllo più specifico delle opzioni, basterà utilizzare le variabili cmake

```
$ cmake .. -DBUILD_<OPTIONS>=ON/OFF
```

Il dettaglio di tutte le opzioni possibili è riportato nel file *CMakeList.txt* all'interno della cartella */ibrdtn/ibrdtn* presente nel repository.

Avvio Dopo aver completato l'installazione, è possibile avviare il demone IBR-DTN utilizzando il comando `dtnd`. Tale comando, invocato senza opzioni, avvia il demone utilizzando la configurazione di default. E' possibile utilizzare l'opzione `-i` per specificare l'interfaccia di rete alla quale associare il processo demone, oppure `-v` per abilitare la stampa dei messaggi di log, `-d` per scegliere il livello di log che verranno stampati, etc. Un esempio del comando:

```
$ dtnd -i eth0 -v
```

Con questa conquesta combinazione di parametri avremo il binding sull'interfaccia di rete *eth0* e log sulla console per le informazioni pricipali Per un elenco completo delle opzioni disponibili utilizzare il flag `-h`. Una volta avviato, il demone IBR-DTN rileverà automaticamente la presenza di demoni in esecuzione su macchine direttamente raggiungibili tramite il modulo di IP Neighbor Discovery e simultaneamente, annuncerà il suo EID locale per essere scoperto dagli altri. Nella configurazione di default, tale EID utilizza lo schema DTN e il nome della macchina locale come SSP, nella forma `dtn://hostname`.

2.3.4 Configurazione

Per modificare il comportamento predefinito del demone è necessario specificare i parametri da utilizzare all'interno di un file di configurazione. Un esempio di configurazione è reperibile al path `ibrdtn/daemon/etc/ibrdtnd.conf`, all'interno del repository utilizzato per l'installazione, o all'indirizzo [8]. In seguito sono illustrate le parti più significative.

```
# the local eid of the dtn node
# default is the hostname
local_uri = dtn://node.dtn
```

permette di personalizzare l'EID locale, se non specificato IBR-DTN ne creerà uno per noi secondo la formattazione standard degli URI dtn `dtn://hostname`.

```
# defines the storage module to use
# default is "simple" using memory or disk (depending on storage_path)
# storage strategy. if compiled with sqlite support, you could change
# this to sqlite to use a sql database for bundles.
storage = default
```

Definisce in che modo salvare fino alla scadenza del TTL i bundle. In memoria volatile (RAM) o su disco se specificato un path di salvataggio.

```
# a list (seperated by spaces) of names for convergence layer instances.
net_interfaces = lan0 lan1 hci0

# configuration for a convergence layer named lan0
net_lan0_type = tcp # we want to use TCP as protocol
net_lan0_interface = wlan0 # listen on interface eth0
net_lan0_port = 4556 # with port 4556 (default)
```

```
8
9  # configuration for a convergence layer named lan1
10 net_lan1_type = tcp # we want to use TCP as protocol
11 net_lan1_interface = eth0 # listen on interface eth0
12 net_lan1_port = 4557 # with port 4557
13
14 # configuration for a convergence layer named hci0
15 net_hci0_type = bluetooth # we want to use bluetooth extension
16 net_hci0_interface = hci0 # listen on interface hci0
17 net_hci0_port = 10 # on channel 10
```

Avendo a disposizione diversi convergence layer è possibile listare nel file di configurazione tutte le interfacce disponibili sul dispositivo. IBR-DTN proverà, dunque, a fare un bind sui protocolli scelti, permettendo così la comunicazione su bundle.

```
1 # routing strategy
2 # values: default | epidemic | flooding | prophet | none
3 # In the "default" the daemon only delivers bundles to neighbors and static
4 # available nodes. The alternative module "epidemic" spread all bundles to
5 # all available neighbors. Flooding works like epidemic, but do not send the
6 # own summary vector to neighbors. Prophet forwards based on the probability
7 # to encounter other nodes (see RFC 6693).
8 routing = epidemic
```

specifica l'algoritmo di routing da utilizzare scegliendo tra le opzioni mostrate nei commenti.

```
1 # forward bundles to other nodes (yes/no)
2 routing_forwarding = yes
```

abilita/disabilita l'inoltro di bundle da parte del nodo.

```
1 # forward singleton bundles directly if the destination is a neighbor
2 routing_prefer_direct = yes
```

abilita/disabilita l'inoltro diretto alla destinazione di un bundle se questa è raggiungibile direttamente.

2.3.5 Configurazione della time synchronization

La sincronia temporale è un punto molto critico della configurazione di IBR-DTN. Nel momento in cui si utilizzano dei dispositivi reali, che non hanno la possibilità di avere un orologio sempre sincronizzato con il resto del mondo, è necessario disattivarla.

Attivare la time synchronization significa avere la possibilità di scartare i bundle all'arrivo se questi sono troppo vecchi e quindi ritenuti inutili, ma questo è un

comportamento che può portare all'impossibilità di comunicazione tra i dispositivi DTN. La rete DTN in quanto tale non prevede che i nodi all'interno abbiano perennemente accesso ad un servizio di time synchronization esterno come ad esempio l'NTP e non è in alcun modo garantito che i device abbiano l'orologio interno settato entro un certo ritardo. L'esempio possibile è quello di un dispositivo con sola connessione bluetooth che viene utilizzato per poche ore, per poi essere riacceso molto tempo più avanti. L'orologio di sistema di quest'ultimo non sarà mai sincronizzato con il resto della rete, perciò se la time synchronization viene attivata il dispositivo non creerà mai dei bundle validi all'interno della rete.
È perciò consigliabile disattivare tale comportamento.

```
# set to yes if this node is connected to a high precision time reference
# like GPS, DCF77, NTP, etc.
#
time_reference = no
```

2.3.6 Applicativi di interazione con IBR-DTN

Al fine di sperimentare l'utilizzo del DTN Bundle Protocol, oltre al processo demone, il software IBR-DTN mette a disposizione una serie di tool a linea di comando. `dtnping` invia dei bundle ad uno specifico EID destinazione e si mette in attesa delle risposte, misurando il tempo di andata/ritorno. `dtnsend` e `dtnrecv` permettono il trasferimento di file tra nodi DTN. Qualora si voglia testare l'API testuale esposta dal demone, è possibile usare strumenti come `telnet` o `netcat`, come nell'esempio seguente:

```
$ telnet localhost 4550
Trying ::1...
Connected to localhost.
Escape character is '^]'.
IBR-DTN 0.11.0 (build dfb7402) API 1.0
protocol management
200 SWITCHED TO MANAGEMENT
neighbor list
200 NEIGHBOR LIST
dtn://neighbor1
dtn://neighbor2
```

In questo esempio, dopo essersi collegati al demone IBR-DTN in esecuzione localmente alla porta 4550, si invoca il comando `protocol management` per accedere alla API di Management. È possibile richiedere la lista dei nodi DTN adiacenti al nodo locale, utilizzando il comando `neighbor list`, come riportato in esempio o inviare comandi per la gestione dei bundle.

Capitolo 3

Routing in DTN

Poiché le reti DTN operano in ambienti dove connettività intermittente e lunghi ritardi sono molto frequenti, il routing deve essere capace di mantenere alte le performance in condizioni estreme. Per questo motivo è stato proposto un nuovo paradigma di routing chiamato *store-carry-and-forward* come estensione dello *store-and-forward* utilizzato in Internet.

In *store-carry-and-forward*, quando un nodo DTN riceve dati da trasmettere alla destinazione, decide se inoltrarlo al nodo "vicino" direttamente connesso o salvarlo nella sua memoria, in attesa di un'opportunità di inoltro più adeguata. C'è da notare che il concetto di *link* usato nelle reti tradizionali, vieni sostituito dal concetto di *contacts* che viene stabilito quando dei nodi DTN si incontrano. Data la natura mutevole della rete, anche questi *contacts* varieranno. I *contacts* in una DTN possono essere classificati in:

- **Scheduled contacts**, riscontrabili in una DTN dove i nodi si incontrano secondo uno schema conosciuto a-priori, per esempio in scenari spaziali dove le orbite dei pianeti e dei satelliti sono conosciute.
- **Probabilistic encounters** seguono una qualche distribuzione statistica che determina la probabilità per un nodo di incontrarne un altro. Questa distribuzione di probabilità può essere basata sulla storia degli incontri passati.
- **Random contacts** caratterizzano una DTN dove non ci sono conoscenze a-priori e ogni nodo può incontrarne un altro in modo del tutto casuale.

Siccome la connessione end-to-end non è garantita, le decisioni di routing sono prese hop-by-hop basandosi su una varietà di fattori. In letteratura sono presenti moltissimi studi su diversi protocolli di routing per reti di tipo Delay-Tolerant Network. L'obbiettivo è quello di scegliere un protocollo di routing che ci permetta

di costruire la topologia della rete e determinare il percorso migliore tra sorgente e destinazione. In questo capitolo verranno illustrati gli approcci, le tecniche di routing e i diversi protocolli presenti in letteratura [9].

3.1 Routing: approcci e tecniche

3.1.1 Approcci di routing

Per la costruzione topologia e il funzionamento di una rete DTN sono possibili quattro tipi di approcci: **pro-attivo**, **reattivo**, **ibrido** e **gerarchico**.

Approccio Pro-attivo Questi protocolli sono *table driven*: ogni nodo possiede una tabella, aggiornata costantemente, con le informazioni sui collegamenti verso tutti gli altri nodi. In tali protocolli, dunque, lo stato delle tabelle deve rimanere il più aggiornato possibile, anche in mancanza di un effettivo utilizzo della rete. Si riesce così ad ottenere un forwarding immediato dei pacchetti, a discapito però di un overhead di comunicazione considerevole, anche quando non vi è traffico da smaltire in rete, necessario a tenere aggiornate le tabelle.

Approccio Reattivo Con questo approccio, invece, i nodi ignorano totalmente la posizione del destinatario fin quando non vi devono comunicare; il routing viene stabilito su richiesta (*on-demand).* Si garantisce, quindi, un'assenza del traffico di controllo, se non vi sono delle sessioni dati in corso, ma ogni nuova sessione di comunicazione sarà soggetta ad una latenza iniziale necessaria alla route discovery.

Approccio Ibrido In questo caso si cerca di ottimizzare e sintetizzare i precedenti due, usando il primo per i nodi "vicini", e il secondo per quelli "lontani". In questo caso resta comunque da dare una soddisfacente definizione della caratteristica vicino/lontano.

Approccio Gerarchico Questo approccio riduce l'overhead di segnalazione dei protocolli puramente pro-attivi, suddividendo i nodi in classi e mantenendo nelle tabelle una entry per ognuna di esse. In tal modo è possibile ridurre significativamente l'overhead di segnalazione. In dettaglio, si suddivide la rete in cluster, in cui viene eletto un capo cluster: si sacrifica così parte della flessibilità della rete ad hoc, ma si ottiene una struttura centralizzata, che rende più semplice la gestione dei vari aspetti della comunicazione e che consente inoltre di implementare protocolli maggiormente scalabili. All'interno del cluster, per avere dei percorsi sempre

disponibili, si usano tecniche pro-attive, mentre per le comunicazioni inter-cluster si usano tecniche reattive, in cui i pacchetti passano da un capo cluster all'altro sino a raggiungere il cluster destinatario.

3.1.2 Tecniche di routing

Le modalità con cui si scambiano i bundle in una rete, sono essenzialmente due: attraverso la replica dei pacchetti (**Replication-based routing**) e quello basato sull'inoltro (**Forwarding-based routing**); solo i protocolli basati sulla replica, nel nostro caso, saranno studiati più a fondo nella sezione di testing 3.4.

Replication-based routing [9] Consiste nel replicare un messaggio con l'obiettivo di ridurne il ritardo di consegna e aumentarne il rapporto. Il nodo in questione conserverà il messaggio ricevuto e ne genererà un altro identico da inoltrare sulla rete verso i nodi vicini: in questo modo la comunicazione sarà più veloce, ma ci sarà uno spreco di risorse di rete. Le tecniche che si basano sulla replica a loro volta possono essere divise in due categorie:

- **Unlimited Replication**, dove tutti i nodi sono autorizzati a creare illimitate copie dei messaggi e mandarli attraverso la rete alla ricerca della destinazione. Nel peggiore dei casi il messaggio viene replicato *n-1* volte prima di raggiungere la destinazione, dove n è il numero di nodi della rete. In questa categoria i protocolli che studieremo sono:
  - *Epidemic*
  - *Epidemic Oracle*
  - *PRoPHET*
  - *PRoPHET V2*
  - *MaxProp*
- **Quota-Based Replication**, dove il numero di messaggi replicati nella rete è limitato e specificato come parametro. In questa categoria i protocolli che studieremo sono:
  - *Spray & Wait Vanilla*
  - *Spray & Wait Binary*

Forwarding-based routing [9] Consiste nell'inoltrare un singolo messaggio attraverso il percorso migliore verso la destinazione, ovvero nel prendere il pacchetto ricevuto e inoltrarlo direttamente verso il next hop sul percorso verso la destinazione senza replicarlo, in questo modo: si può ottenere uno minor spreco delle risorse di rete, ma un rapporto di consegna molto più alto. I protocolli che studieremo sono:

- *First Contact*
- *DTLSR (Delay-Tolerant Link State Routing)*
- *CGR (Contact Graph Routing)*

3.2 Protocolli Replication-based

In questa sezione studieremo i protocolli di routing basati sulla replica dei pacchetti, caratterizzati da performance più elevate in termini di percentuale di consegna, overhead, ritardo di consegna, costo computazionale e tempo medio di un pacchetto nel buffer.

Epidemic e Epidemic Oracle In questi protocolli di routing ogni nodo replica il messaggio ad ogni altro che incontra. I messaggi, quindi, verranno potenzialmente replicati su tutti i nodi della rete. La destinazione avrà una probabilità di ricezione del messaggio pari al 100%, tranne nei casi in cui ci si ritroverà in una rete formata da nodi con buffer limitati, che potrebbero quindi scartare pacchetti per via della loro saturazione o sufficientemente ampi, ma non in grado di replicare tutti i bundle per via del limitato contatto. La perdita dei pacchetti, prima dell'arrivo a destinazione può anche essere dovuta al raggiungimento della soglia del Time To Live (TTL).

Il protocollo Epidemic funziona come segue: due nodi che si incontrano iniziano con lo scambio della lista dei bundle ID memorizzati all'interno dei propri buffer e, dopo aver controllato quali di questi ID il proprio "vicino" non ha, ogni nodo replica i pacchetti con ID mancante. Alla fine entrambi i nodi avranno gli stessi messaggi all'interno dei loro buffer. L'approccio del protocollo Epidemic routing è molto semplice d'altro canto, però, consuma una grande quantità di risorse per via delle numerose repliche di messaggi circolanti all'interno della rete. Questo porterà, quindi, ad un grande consumo di spazio nei buffer, di banda e di potenza richiesta.

Un altro problema del protocollo Epidemic è la continua disseminazione di copie di messaggi nella rete, anche quando la destinazione viene raggiunta. Per ovviare a questo problema è stato proposto il concetto di "death certificates". La versione

Epidemic Oracle, implementa appunto questo concetto, inviando dei messaggi di Acknowledge in modo da garantire al nodo "vicino" la cancellazione di tale pacchetto dal proprio buffer per avvenuta ricezione. Generalmente questi pacchetti di notifica sono più piccoli dei messaggi originali [9].

PRoPHET **P**robabilistic **Ro**uting **P**rotocol using **H**istory of **E**ncounters and **T**ransitivity [10]. Questo algoritmo è, anche esso, un algoritmo di routing basato sulla replica illimitata dei pacchetti; Sfrutta la non casualità in una rete DTN, mantenendo una lista di probabilità di incontro di altri nodi ed utilizzando, appunto, queste probabilità per la spedizione e la replica dei messaggi ad un nodo avente una probabilità di consegna, verso la destinazione, migliore della sua.

Il nodo di inoltro N, memorizza la probabilità per ogni destinazione $\rho(N, D)$, dove N è il nodo di inoltro e D la destinazione. Nel caso in cui non si conosca nessuna probabilità per una destinazione, $\rho(N, D)$, si assumerà essere zero. Le probabilità sono ricalcolate ad ogni incontro secondo queste regole:

- Quando un nodo N incontra un altro nodo B, la probabilità per B aumenta secondo la regola $\rho(N, B)_{new} = \rho(N, B)_{old} + (1 - \rho(N, B)_{old}) * X_{encounters}$ dove $X_{encounters}$ è una costante
- Le probabilità per tutte le destinazioni D sono calcolate come $\rho(N, D)_{new} = \rho(N, D)_{old} * Y^k$ dove Y è una costante e k è il numero di unita di tempo trascorse dall'ultimo calcolo
- Le probabilità sono scambiate tra N e B e la proprietà transitiva delle probabilità è usata per aggiornare la probabilità verso la destinazione D come $\rho(B, D)_{new} = \rho(B, D)_{old} + (1 - \rho(B, D)_{old}) * \rho(N, B) * \rho(B, D) * \beta$, dove β è una costante scalare

A contrario dell'algoritmo precedente, PRoPHET è sicuramente più complesso, produce meno copie all'interno della rete con, però, un rapporto di consegna più basso e uno stazionamento dei pacchetti all'interno del buffer più lungo. Esiste una seconda versione di PRoPHET, **PRoPHET V2**, dove i calcoli delle probabilità sono eseguiti in modo differente, ottenendo risultati più realistici [11].

MaxProp anche conosciuto come *Maximum Priority* [12] è come l'algoritmo precedente, basato sul calcolo di una probabilità per stabilire quale nodo debba prendere in carico il pacchetto da spedire verso la destinazione. Ogni nodo possiede una lista di nodi "vicini" con una probabilità di incontro ρ. Ad ogni incontro del nodo corrente con i suoi vicini, la probabilità associata viene modificata come segue: $\rho = (\alpha + 1)/2$ dove α rappresenta la probabilità avuta fino al momento del nuovo

incontro. Una volta calcolata la nuova probabilità tutte le altre, riferite ai restanti vicini, vengono normalizzate, $\rho_i = \alpha_i/2$, in modo da garantire che la somma totale di esse sia sempre pari ad uno, $\sum_{i=0}^{n} \rho_i = 1$. Ogni nodo memorizza, inoltre, per ogni destinazione D, i relativi percorsi con il correlato costo di raggiungibilità, determinato come $C = \sum_{i=0}^{n}(1 - \rho_i)$ con n numero di nodi da sorgente a destinazione; vengono utilizzate, quindi, le probabilità precedentemente calcolate per la stima del percorso.

I nodi "vicini" si scambiano le informazioni immagazzinate sui percorsi disponibili, in modo da sfruttarle nella fase di forwarding, per scegliere il percorso più adatto al raggiungimento della destinazione. I messaggi vengono ordinati secondo il costo più basso, all'interno del buffer di inoltro, e spediti in ordine dal meno al più costoso, che di conseguenza sarà scartato in caso di buffer pieno. I nodi destinazione inviano un messaggio di Acknowledge per certificare l'arrivo del messaggio e garantire ai nodi di inoltro la cancellazione del pacchetto dai propri database.

Come per il protocollo precedente, si contraddistingue per: la sua più elevata complessità algoritmica, la replica del messaggio a nodi specifici con, però, un rapporto di consegna più basso rispetto al primo.

Spray & Wait Questo protocollo, a contrario degli altri, definisce un massimo numero di copie replicabili in rete. Inizialmente ogni volta che la sorgente genera un nuovo messaggio, il protocollo di routing gli assegna un numero t, che rappresenta il numero massimo di copie permesse in rete. Come dice il nome è composto da due fasi:

- Fase di **Spray**, la sorgente *sprays* ("spruzza") una copia al primo nodo di inoltro che incontra. Quando il nodo di inoltro riceve la copia del messaggio entra nella fase di *Wait*
- Fase di **Wait**, il nodo di inoltro salva il messaggio fino all'incontro del nodo destinazione

La precedente descrizione è la versione di base **Vanilla** di *S&W* dove solo la sorgente può replicare t messaggi ad un altro nodo che non sia la destinazione.

Una variante di *S&W Vanilla*, chiamata **Binary**, costruisce un albero binario che permette alle copie dei messaggi di essere inoltrate da nodi diversi dalla sorgente. La sorgente, quindi, parte con L copie del messaggio e ne replica $N = L/2$ ad ogni nodo che incontra, con L che si aggiorna alla sua metà dopo ogni invio. Ogni nodo che possiede N copie con $N > 1$ quando incontra un nodo B può replicare, a sua volta, metà delle copie e memorizzare l'altra metà. Quando rimane con una sola copia del messaggio si entra nella modalità *direct transmission*, dove il nodo replica la singola copia ai nodi incontrati, fino alla destinazione.

A contrario dei precedenti è meno complesso a livello algoritmico, ma sicuramente meno efficace in termine di consegna, soprattutto la versione *Vanilla* [9].

3.3 Protocolli Forwarding-based

Come sottolineato in precedenza, questi algoritmi non saranno studiati tutti fino in fondo nella fase di testing, meritano, però, una spiegazione data la loro menzione nei capitoli successivi.

First Comtact Questo algoritmo inoltra il pacchetto ad ogni nodo incontrato e di conseguenza ogni nodo farà lo stesso, meritava un paragone con gli altri sopra elencati per la sua estrema semplicità e il bassissimo spreco di risorse [9].

DTLSR É abbreviazione di *Delay-Tolerant Link State Routing*, un estensione di *Link-State Routing*. DTLSR considera che ogni nodo, nel sistema, viene assegnato ad un area e il protocollo di Link-State opererà esclusivamente su quell'area. Quando ci sono dei cambiamenti, il protocollo annuncia in flooding il cambiamento alla rete. Ogni nodo mantiene un grafo, che rappresenta la corrente composizione della rete e utilizza un algoritmo, basato sulla scelta del percorso migliore, per cercare la via tra sorgente e destinazione dove far passare il messaggio [9].

Contact Graph Routing (CGR) Il CGR sfrutta il fatto che ogni nodo conosce in anticipo tutte le connessioni, che in seguito chiameremo contatti della rete, poiché i movimenti di ogni singolo nodo sono stati configurati a priori. Grazie a questa conoscenza è possibile evitare la fase di dialogo iniziale tra due nodi in quanto entrambi saranno già al corrente della possibilità di comunicazione. Ricordiamo che anche una semplice fase di accordo iniziale in questo ambiente può essere complicata a causa di elevato RTT e tasso d'errore.

La base del CGR è il *Contact Plan*, ovvero la lista dei contatti fra i nodi della rete; permette all'algoritmo di determinare a priori quando poter trasmettere verso un certo nodo. I *Contact Plan* sono contenuti in un file di configurazione. All'interno del *Contact Plan* sono presenti messaggi di due tipi: *Contact* e *Range*. A partire dai messaggi del *Contact Plan* ogni nodo crea localmente una tabella di routing. Le tabelle di routing contengono una lista di percorsi verso tutti i possibili nodi destinazione che vengono riportati almeno una volta nel *Contact Plan*. Ogni percorso è caratterizzato da un nodo iniziale e dalla lista di tutti gli altri nodi verso la destinazione. Le rotte all'interno di una *route list* sono elencate in ordine crescente

di costo per i bundle che vengono trasmessi. Il costo può dipendere da metriche differenti, come il tempo di consegna. Un collegamento fra due nodi è destinato ad aprirsi e chiudersi varie volte, ad istanti di tempo definiti a priori. Non esiste un percorso migliore fra sorgente e destinazione, ma un percorso migliore ad ogni istante di tempo. Possiamo definire quattro fasi dell'algoritmo: calcolo delle rotte (Dijkstra), selezione delle rotte accettabili, selezione di una rotta (massimo) per ogni prossimo nodo e inserimento del bundle in coda [9].

3.4 Routing: Testing dei protocolli

In questa sezione verranno presentati i test e i risultati ottenuti simulando su di una rete i protocolli sopra descritti, al fine di produrre dei benchmark preliminari per individuare l'algoritmo migliore tra quelli disponibili. La simulazione è stata eseguita con l'ausilio del simulatore *The ONE*.

3.4.1 The ONE simulator

Il simulatore ONE è stato sviluppato: nei progetti SIN-DTN e CAT-DTN supportati dal Nokia Research Center (Finlandia), nei progetti TEKES ICT-SHOK Future Internet e IoT-SHOK, RESMAN e Picking Digital Pockets (PDP) dell'Accademia di Finlandia, in SCAMPI progetto della Comunità europea (Seventh Framework Programme) e supportato da EIT ICT Labs. Questo simulatore permette all'utente di creare una rete DTN e di simulare il funzionamento di alcuni algoritmi di routing [13].

Il software, sviluppato in linguaggio Java in maniera modulare, permette agli utenti di sviluppare nuove funzioni o estendere quelle presenti facilmente utilizzando le interfacce base.

Gli elementi base di ogni simulazione sono i nodi. Un nodo può essere in grado di ricevere, memorizzare ed inviare messaggi e inoltre può muoversi nel mondo simulato secondo diversi schemi. Ogni scenario è composto da vari gruppi di nodi, ogni gruppo può avere configurazioni diverse come il modello di movimento, la strategia di instradamento dei messaggi, la grandezza del buffer, il tempo di vita dei messaggi creati, la velocità di movimento e l'interfaccia di trasmissione dei dati che determina la velocità di invio/ricezione ed il raggio entro il quale è possibile avere un contatto. L'ambiente di simulazione The ONE è in grado di:

- visualizzare sia la mobilità che i messaggi passanti in tempo reale, attraverso la sua interfaccia grafica
- importare dati sulla mobilità da percorsi reali (strade di città, paesi ...).

- produrre una varietà distinta di report dallo spostamento dei nodi al passaggio dei messaggi e, inoltre, statistiche di carattere generali.

- rendere possibile il setting delle impostazioni di simulazione, attraverso un file di configurazione

3.4.2 Scenario generale

Data la prospettiva di utilizzo di IBR-DTN in aree di lavoro molto grandi, come ad esempio: campi agricoli, pascoli o aree di costruzione; si è deciso di effettuare delle simulazioni di durata di 8 ore, rappresentante una fascia oraria lavorativa giornaliera, in un'area di $50 \times 100\ m^2$.

La comunicazione fisica tra i diversi nodi viene simulata dalla tecnologia bluetooth. Le interfacce hanno una velocità di $2Mbps$ e una portata massima di 10 metri. La dimensione dei buffer dei nodi è tra i 50-100 MB, ovvero ci sono dei nodi che, in modo casuale, avranno una grandezza presa tra quel range di valori. Ad ogni pacchetto generato è stato applicato un *Time To Live* (TTL) di 300 minuti allo scadere del quale i pacchetti vengono scartati. Ogni nodo all'interno della rete si muove in modo casuale nella propria area di lavoro, secondo il pattern *Random-WayPoint*, oppure secondo un percorso prefissato. Ogni nodo in modo casuale può generare pacchetti ogni 25/35 secondi verso una destinazione casuale. Per far si che due nodi si scambino dati, le loro aree operative devono sovrapporsi almeno della metà, come mostrato in figura 3.1.

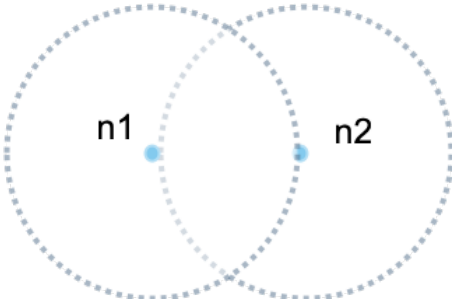


Figura 3.1. Sovrapposizione aree operative

I test sono stati divisi in diversi scenari, varianti per la forma della rete e il percorso dei nodi. I risultati sono stati costruiti su 4 parametri:

- Percentuale di consegna **(Delivery Percentage)**, ovvero il numero di pacchetti consegnati rispetto a quelli spediti $D_{prob} = np_{delivered}/np_{sent} * 100$

- **Relay Overhead**, ovvero la percentuale di repliche (inteso come numero di hop attraversati dai bundle per raggiungere la destinazione meno il numero di consegne) rispetto al numero di pacchetti consegnati $O = (n_{relayed} - np_{delivered})/np_{delivered} * 100$, da ora in poi lo chiameremo semplicemente **Overhead**

- Tempo di consegna (**Delivery Time**), ovvero il tempo che un pacchetto impiega per raggiungere la destinazione $D_t = t_{arrival} - t_{creation}$

- Tempo medio nel buffer (**Average Time Buffer**), ovvero il tempo medio che un pacchetto trascorre nel buffer prima di essere inviato $AVG_{t-buffer} = t_{delete} - t_{arrival}$

I nodi della rete sono stati suddivisi in tre categorie:

- *Nodi neri*, posti ai vertici dell'area, che simulano dei nodi stazionari, ad esempio nodi che forniscono servizi come: sensori di temperatura, di movimento, etc;

- *Nodi blu*, posti in diverse aree più piccole di quella principale, che simulano nodi in movimento, ad esempio macchinari di lavoro e/o persone che operano in piccole porzioni del campo;

- *Nodi rossi*, posti nell'area principale ricoprente l'intero campo di lavoro, che simulano macchinari e/o persone lavorare spostandosi sull'intero campo.

3.4.3 Scenario 1

Il primo scenario, come si evince in figura 3.2, rappresenta una rete composta da 19 nodi, 4 dei quali stazionari (*neri*), nei vertici dell'area, 10 nodi mobili ad una velocità tra 0-3 Km/h nelle aree più piccole (*blu*), 4 delle quali sovrapposte di circa 10 metri con quella centrale, e altri 5 nodi mobili ad un velocità tra 0-5 Km/h nell'area più grande che comprende l'intera zona di simulazione (rossi).

In figura 3.3 si può notare come le percentuali di consegna siano tutte molto elevate, ma non raggiungono mai il 100%, a causa del fatto che i buffer erano di dimensione limitata e quindi molti pacchetti sono stati scartati per via delle code piene o della scadenza del TTL. L'overhead nei casi di algoritmi epidemici risulta essere molto alto, dovuto alla replica continua di pacchetti a tutti i nodi incontrati; gli altri algoritmi presentano un overhead decisamente più basso e simile tra loro. Nei grafici riguardanti i tempi di consegna si può notare come Epidemic Oracle sia il più veloce, come ci si aspettava, dato il suo comportamento di replica continua al contrario degli altri che ricercano il percorso migliore.

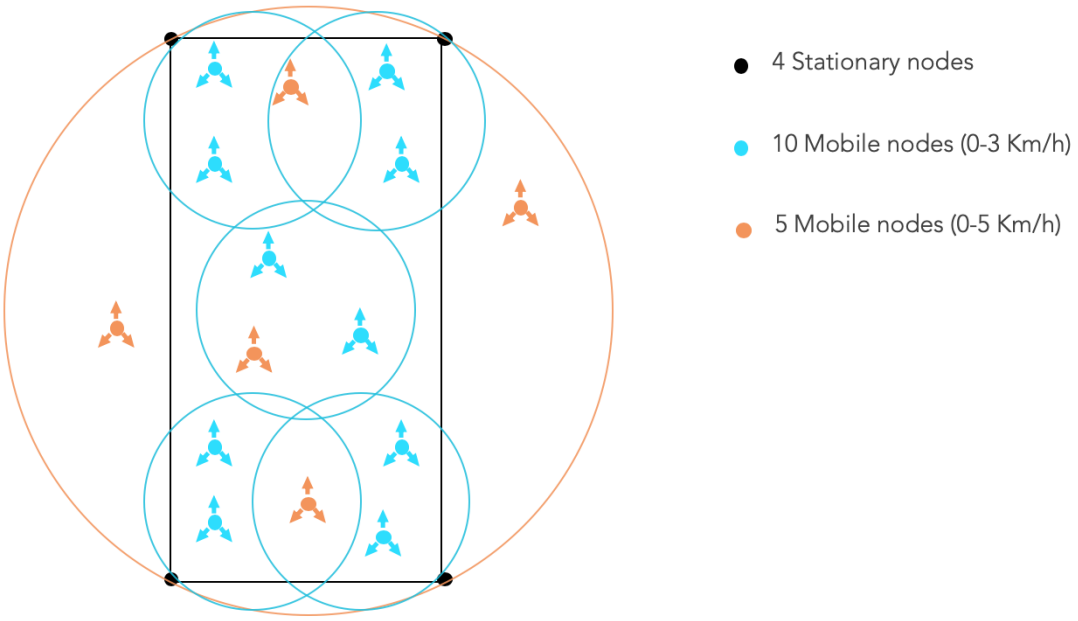


Figura 3.2. Scenario 1

Per quanto riguarda i tempi di stazionamento nei buffer risultano essere più performanti gli algoritmi che utilizzano un sistema di Acknowledge per lo svuotamento delle code, ad eccezione di First Contact, che è un algoritmo di forwarding e che quindi non memorizza i pacchetti all'interno della coda dopo la loro trasmissione. In questo scenario sono state testate anche delle comunicazioni dirette tra nodi scelti non in maniera casuale, facenti parte di aree uguali e aree diverse. Nella prima simulazione si è voluto testare la comunicazione tra due nodi stazionari (*neri*) posti in parti opposte dell'area; nella seconda si è voluto testare la comunicazione tra due nodi facenti parte di differenti aree *blu* comunicanti direttamente; nella terza, ed ultima, si è simulata la comunicazione tra due nodi facenti parte di differenti aree *blu* non comunicanti.

In tutti e tre i casi sono stati testati sempre gli stessi parametri: *Delivery Probability, Overhead, Delivery Time, Average Time Buffer*, portando alla luce che, in media sui quattro parametri, l'algoritmo più performante è risultato essere MaxProp.

Scenario 1.2

In questo caso (Figura 3.4) si è voluto rendere meno casuale lo spostamento dei nodi facenti parte dell'area *rossa*, ovvero quelli con possibilità di movimento sull'intera area. Sono stati creati, quindi, dei percorsi non casuali e rettilinei, da nord a

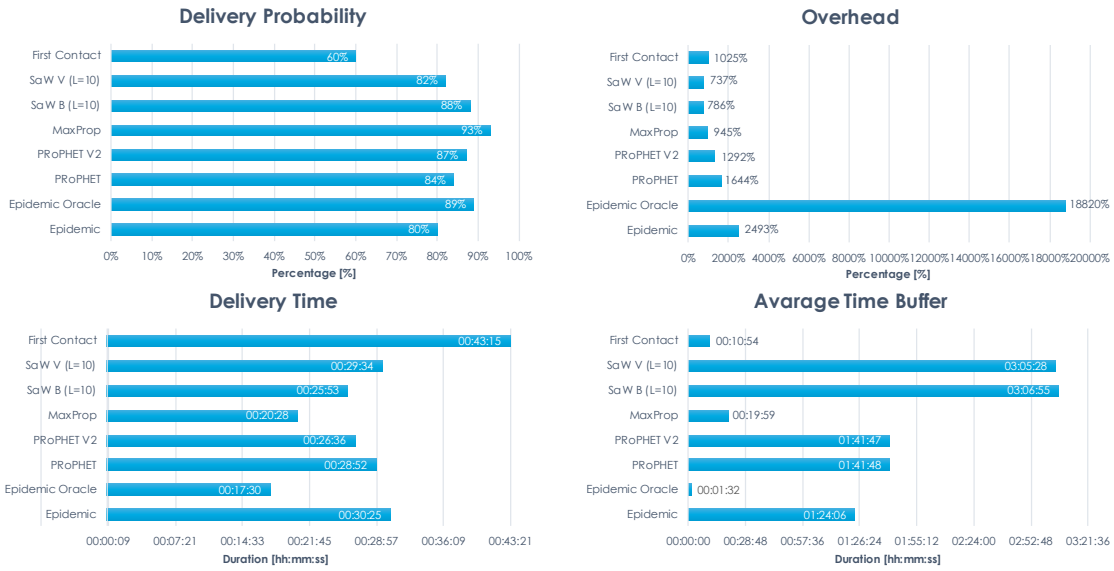


Figura 3.3. Scenario 1 risultati

sud dell'area, simulando, ad esempio, trattori in movimento verticale su un campo agricolo.

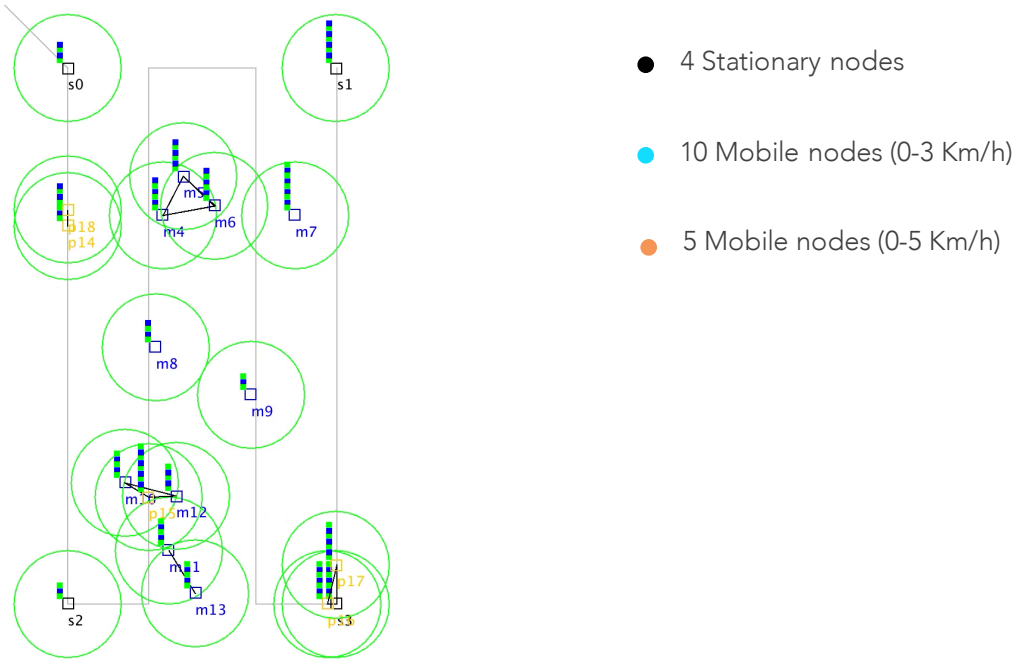


Figura 3.4. Scenario 1.2

I risultati sono pressoché identici ai precedenti; si può quindi constatare che il movimento casuale dei nodi che verrà utilizzato anche nelle future simulazioni all'interno delle aree non è un fattore rilevante di cambiamento di risultati.

3.4.4 Scenario 2

Il secondo scenario (Figura 3.5) rappresenta una rete composta da 14 nodi: 4 dei quali stazionari, *neri*, nei vertici dell'area, 10 nodi mobili ad una velocità tra 0-3 Km/h nelle aree più piccole, *blu*, 4 delle quali sovrapposte di circa 10 metri con quella centrale. Rispetto alle simulazioni precedenti si è voluto eliminare l'area più grande, *rossa*, in cui i nodi si spostavano su tutta la mappa. Questa scelta è dovuta al fatto che: si vuole capire quanto lo scambio di informazioni tra i nodi degrada, data la presenza di minori collegamenti sull'intera area che garantirebbero a nodi lontani una migliore comunicazione.

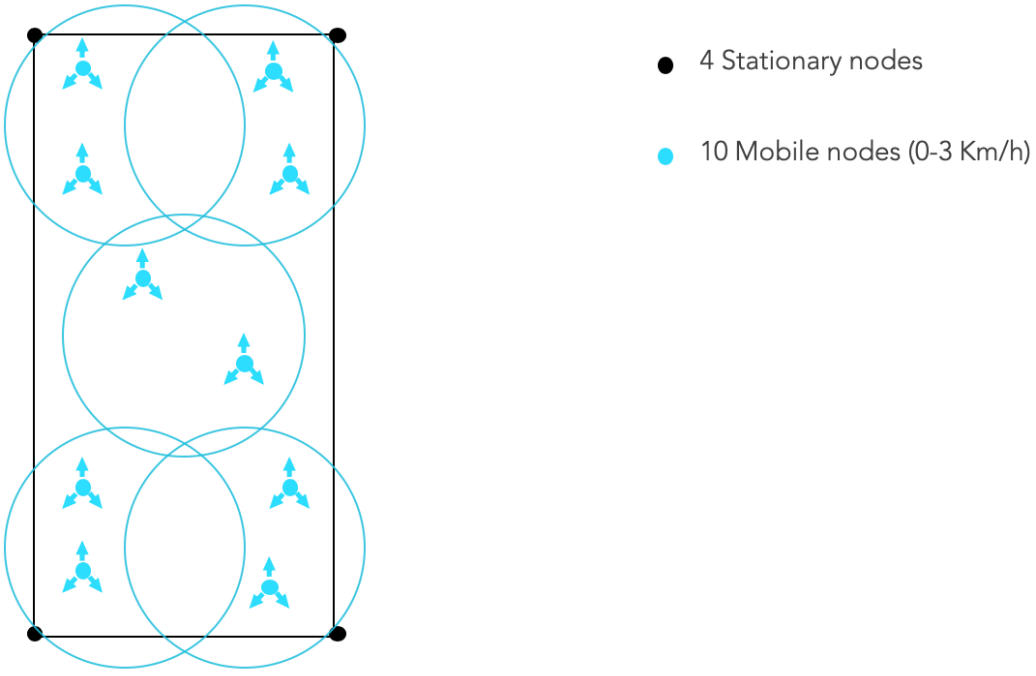


Figura 3.5. Scenario 2

In figura 3.6, si può notare, rispetto allo senario precedente, come le percentuali di consegna si siano abbassate, mantenendo però invariate le conclusioni per cui il protocollo epidemico risulta essere il più prestazionale date le sue caratteristiche, seguito da MaxProp.

L'overhead vede le sue percentuali abbassate dato il minor numero di nodi e di conseguenza il minor numero di repliche, ma mantiene le conclusioni fatte per lo scenario precedente, dove Epidemic Oracle risulta saturare troppo le risorse.

I tempi di consegna e di stazionamento nei buffer risultano aumentati, come ci si aspettava, e in scala uguali alle simulazioni precedenti.

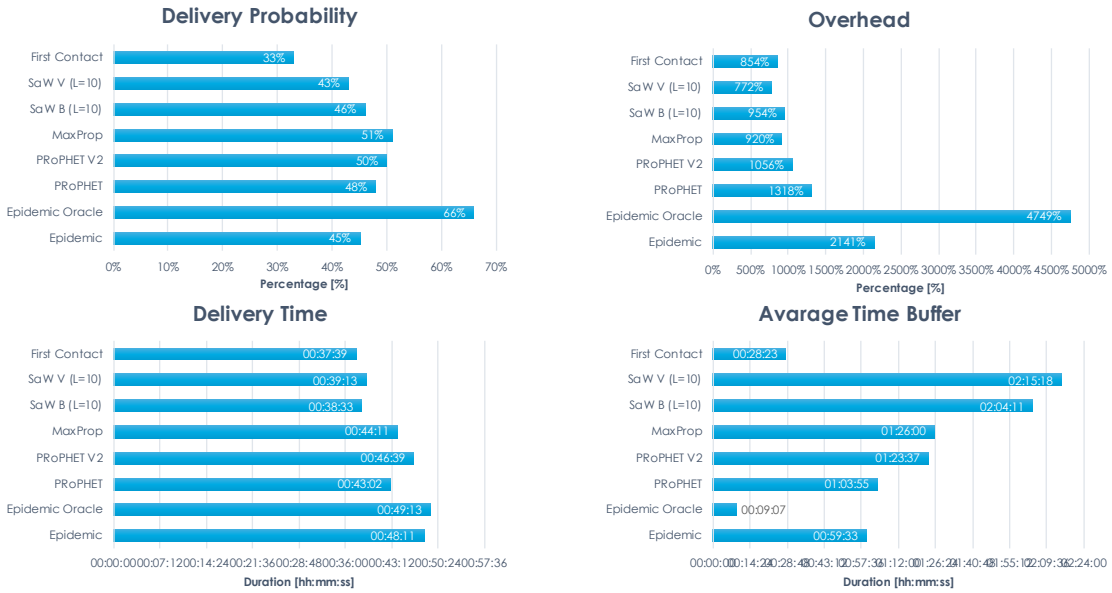


Figura 3.6. Scenario 2 risultati

3.4.5 Scenario 3

Nel terzo scenario (Figura 3.7), si è deciso di replicare la simulazione dello scenario uno, variando la dimensione delle aree più piccole, *blu*.

In questo caso le 4 aree non sono più sovrapposte di circa 10 metri con quella centrale, ma bensì separate di circa 10. Questo per simulare la connessione dei nodi, e quindi lo scambio di informazioni, solamente nel caso in cui i nodi facenti parte di aree diverse si dovessero trovare esattamente sul perimetro della suddetta. Il resto della configurazione rimane invariato rispetto allo scenario 1.

I risultati, in questo caso, si avvicinano molto a quelli dello scenario 1. Le percentuali vedono sempre Epidemic Oracle e MaxProp nelle posizioni più alte. L'overhead sempre molto alto per i protocolli Epidemic e i tempi leggermente più alti, rispetto a prima, ma sempre con Epidemic e MaxProp come protocolli più performanti.

In questo caso, come da obbiettivo, si può constatare che la riduzione delle aree, e quindi le probabilità ridotte di incontro dei nodi, hanno influito sulle performance della trasmissione nella rete.

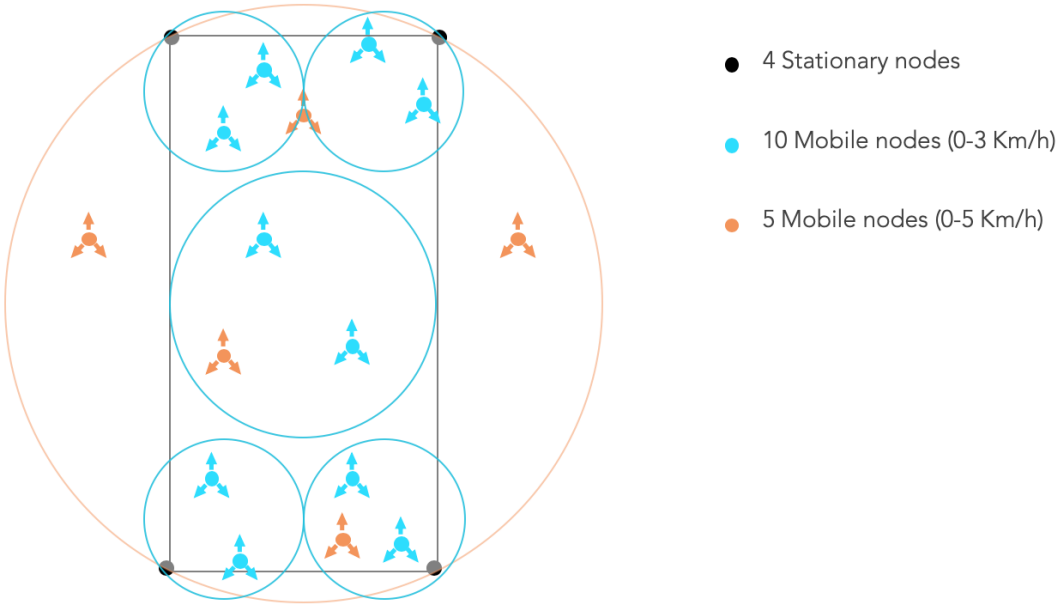


Figura 3.7. Scenario 3

3.4.6 Scenario 4

Nello scenario quattro (Figura 3.8), come nel precedente, si è voluto replicare il test con le aree più piccole sempre separate da quella centrale. In questo caso, si è ridotto ancora di più la probabilità di incontrare nodi per la trasmissione di pacchetti, eliminando l'area più grande, *rossa*, comprendente i 5 nodi mobili a velocità 0-5 Km/h.

Le drastiche modifiche alla rete hanno prodotto dei risultati sempre in scala rispetto ai precedenti (Figura 3.9) ma di gran lunga meno performanti. Le probabilità di consegna e l'overhead si sono ridotte drasticamente rispetto alle simulazioni recedenti. I tempi di consegna e di stazionamento nei buffer si sono alzati. Le considerazioni fatte sui vari protocolli rimangono però invariate. L'obbiettivo di questa simulazione è stato raggiunto, dato che si è riuscito a dimostrare che in condizioni di una rete poco connessa le performance risultano essere basse. Probabilmente i pacchetti tra una sorgente e una destinazione, poste in parti opposte dell'area, hanno una difficoltà elevata ad essere consegnati, dato che due nodi per scambiarsi

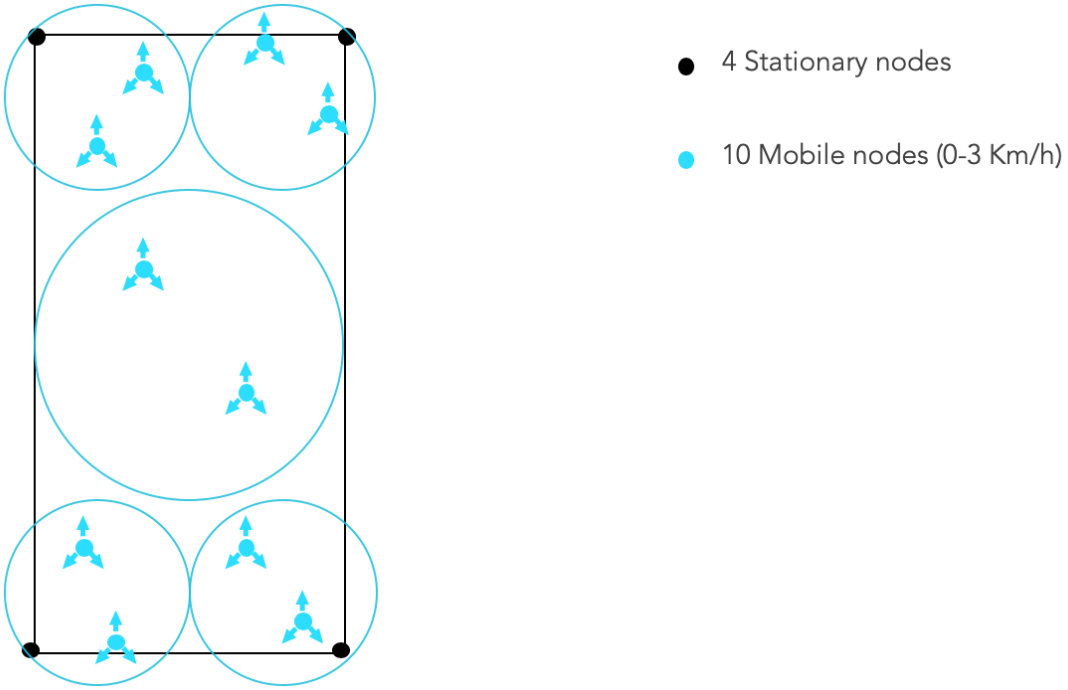


Figura 3.8. Scenario 4

informazioni devono essere, entrambi, sul perimetro della propria area, in modo trovarsi all'interno dell'area operativa del suddetto nodo.

3.4.7 Conclusioni

Possiamo dire che in media per tutti gli scenari presentati basandosi sui 4 parametri di: *Delivery Probability, Overhead, Delivery Time e Average Time Buffer*; **MaxProp** risulta essere quello più performante.

Nella figura 3.10 si possono vedere i risultati di MaxProp nei rispettivi scenari. Con l'obbiettivo di ottimizzare la rete DTN, si è scelto di creare, quindi, un'implementazione di *MaxProp* all'interno di IBR-DTN. L'implementazione per la parte di messaggistica ha subito delle modifiche, dovendo sottostare alla struttura di IBR-DTN, mantenendo, però, intatta la parte matematica ed algoritmica.

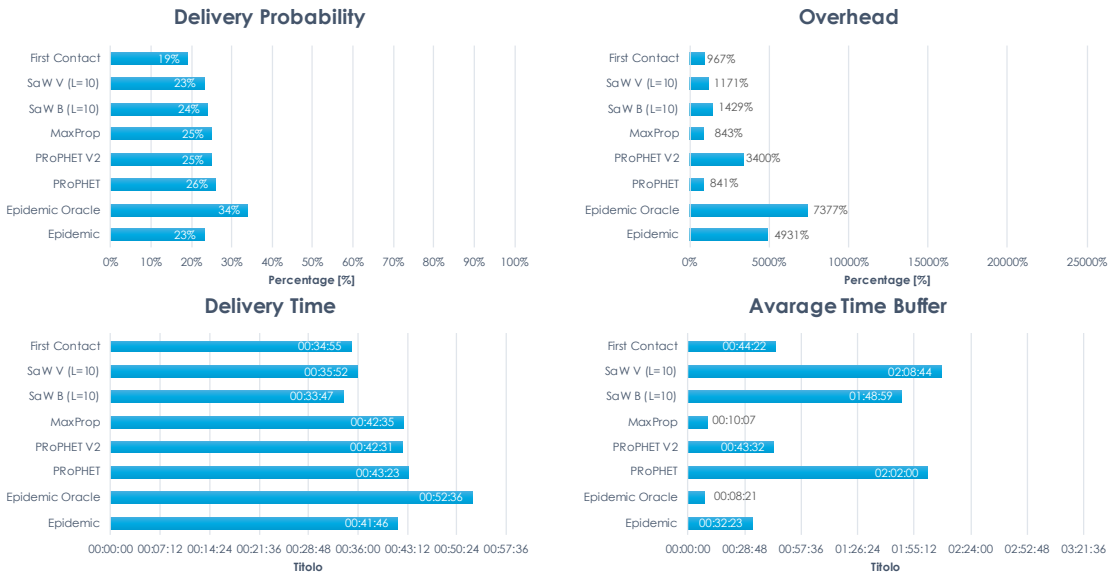


Figura 3.9. Scenario 4 risultati

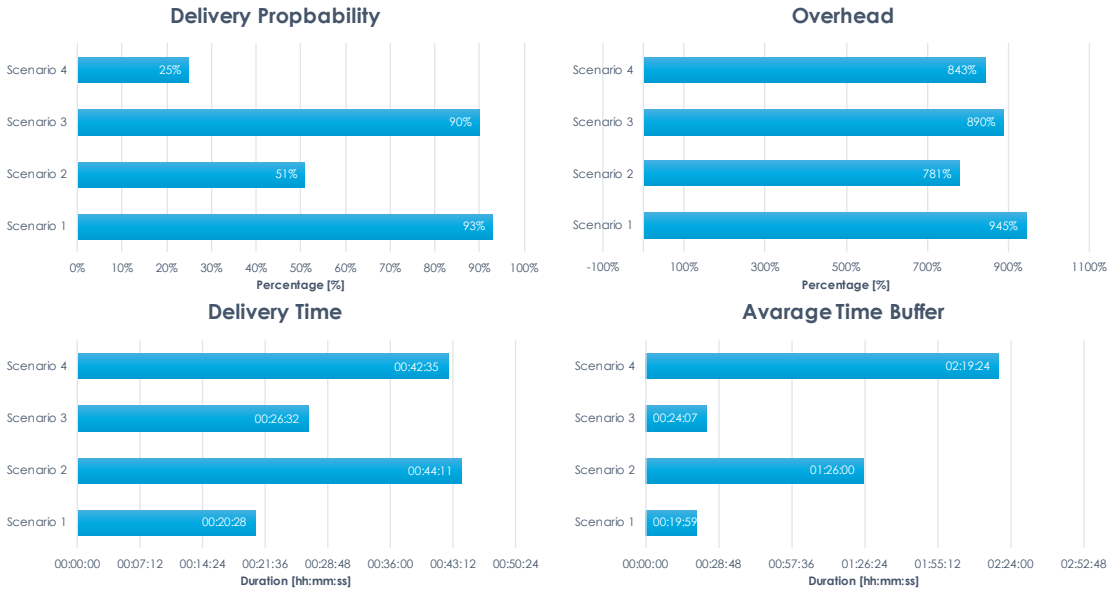


Figura 3.10. Validazione di MaxProp nei quattro scenari

Capitolo 4

Routing in altre piattaforme DTN

In questo capitolo illustreremo i progetti esistenti che sfruttano le potenzialità di una rete DTN, oltre al già citato IBR-DTN, ed i protocolli di routing utilizzati, in modo da avere un confronto sulle scelte fatte. Il protocollo di routing è una parte fondamentale per la costruzione della rete, essendo il motore che ci garantisce lo scambio di informazioni tra i nodi facenti parte di essa.

4.1 DTN2

É stata, inizialmente, sviluppata in C++ e include un simulatore DTN. Può operare con i sistemi operativi: Linux, MAC, Win e FreeBSD. DTN2 è disponibile liberamente con licenza OpenSource. Fornisce estensioni per il routing, storeage e convergence layer attraverso interfacce XML. In particolare sono supportati i seguenti protocolli di routing: PRoPHET (3.2), Epidemic (3.2) e DTLSR (3.3). Il modulo di routing è implementato all'interno della componente *Bundle Router*, che include la scelta dei percorsi e le policies di scheduling per l'inoltro dei pacchetti. Il *Bundle Router* collabora con la componente di *Bundle forwarder* che prende le decisioni di routing. Quest'ultima, durante il processo di routing, interagisce con altre componenti per completare lo scambio di dati nella rete [9].

4.2 ION-DTN

Quest'applicazione è stata sviluppata dalla NASA per la costruzione di una rete DTN che garantisse ai veicoli spaziali le comunicazioni. Questa tecnologia si basa su conoscenze a priori dei movimenti dei nodi facenti parte della rete. La distribuzione del software Interplanetary Overlay Network (ION) è un'implementazione dell'architettura DTN (Delay-Tolerant Networking), come descritto in Internet RFC 4838

[14]. L'obiettivo principale di ION è quello di fornire una suite completa di protocolli per facilitare e automatizzare la comunicazione tra nodi spaziali, nodi sulla superficie di altri pianeti e nodi sulla superficie terrestre. La distribuzione è progettata in modo da permettere un inserimento delle funzionalità DTN anche all'interno di sistemi "embedded". Questo software per il modulo di routing, prevede l'utilizzo del protocollo *Contact Graph Routing* (3.3), basato sull'inoltro e non sulla replica del messaggio [9].

4.3 Bytewalla

É un implementazione Java che sfruttando il protocollo bundle mette in comunicazione dispositivi ANDROID in aree rurali. Questi dispositivi vengono, ad esempio, utilizzati per trasportare dati dalla città ai villaggi circostanti e viceversa. Una volta che il dispositivo ANDROID si trova nel villaggio, riceve bundle dal server via 802.11, mentre quando ritorna in città i bundle sono caricati attraverso gateway APs. É un applicazione basata su DTN2 (4.1) ed utilizza principalmente PRoPHET (3.2) come protocollo di routing [9].

4.4 ContikiDTN

É una piattaforma che implementa un'architettura DTN attraverso il sistema *Contiki* OS [15] che permette la trasmissione di bundle in IEEE 802.15.14. É basato su DTN2 ed implementa un convergence layer TCP che usa messaggi di handshake prima delle trasmissioni. I protocolli di routing utilizzati sono: PRoPHET (3.2), Epidemic (3.2) e DTLSR (3.3) [9].

4.5 6LoWDTN

É un livello di trasporto, funzionante sulle Low-power Wireless Personal Area Networks (LoWPAN), ed è stato adattato per supportare IPv6; è basato sul sistema operativo Contiki [15]. É responsabile delle decisioni di routing e di conseguenza sull'inoltro dei bundle. La decisione è basata su una metrica chiamata Estimated Delivery Delay (EDD), stima del ritardo totale dall'origine alla destinazione attraverso una metrica scelta (occupazione del buffer, residuo della batteria, ...). I nodi si scambiano dei link local broadcast UDP beacon per tenere aggiornate le informazioni riguardanti i link e la metrica utilizzata per fare forwarding [9]. Sono disponibili due protocolli di routing per distribuzioni su larga scala: LOADng, standardizzato dall'ITU e RPL standardizzato dal gruppo di lavoro IETF ROLL.

RPL è un protocollo di routing per reti wireless a basso consumo energetico e generalmente suscettibile alla perdita di pacchetti. È un protocollo pro attivo basato su vettori di distanza e funziona su IEEE 802.15.4, ottimizzato per la comunicazione multi-hop e molti-a-uno, ma supporta anche i messaggi one-to-one [16]. LOADng è un protocollo di routing reattivo basato sul rilevamento del percorso mediante dei messaggi di richiesta e risposta verso il vicino. Pertanto, quando un nodo desidera inviare un messaggio di dati e il percorso verso la destinazione è sconosciuto, dovrebbe iniziare, prima, un nuovo processo per il suo rilevamento [17].

4.6 CoAP over BP

CoAP [18] fornisce un protocollo a livello applicativo che consente ai dispositivi con risorse limitate di interagire in modo asincrono. É progettato per comunicazione machine-to-machine ed è conforme con l'architettura REST. Definisce un semplice livello di messaggi che viaggia sopra UDP. CoAP usa un interazione di tipo request/response tra client e server. La versione quindi di CoAP over BP viene definita come BoAP che è basta su IBR-DTN (2.3), ma implementata in Java, dove ogni messaggio viene incapsulato in bundle. I protocolli utilizzati sono gli stessi che possono essere scelti in IBR-DTN, quindi PRoPHET (3.2) ed Epidemic (3.2) [9].

4.7 Conclusioni

Per concludere possiamo dire che, la maggior parte dei progetti con un infrastruttura di tipo DTN, utilizzano i protocolli Epidemic e PRoPHET, essendo protocolli standardizzati. Nelle nostre simulazioni è emerso, però, che MaxProp risulta essere più performante rispetto ai due appena citati.

Capitolo 5

MaxProp routing su IBR-DTN

Date le considerazioni fatte ed i risultati ottenuti, è stato deciso di implementare l'algoritmo MaxProp nel framework IBR-DTN. In questo capitolo viene presentato il funzionamento dettagliato dell'algoritmo, descritto in 3.2, dal punto di vista teorico e la sua integrazione, grazie ad IBR-DTN, all'interno di Aether.

5.1 Funzionamento e caratteristiche

Ogni nodo all'interno della nostra rete, per comunicare con un altro, esegue l'applicazione Aether. Nell'istante in cui un nodo si accende, se non risulta essere la prima volta, può caricare dallo store, memoria o file, le informazioni riguardanti la topologia della rete e i bundle salvati, non ancora inviati o in attesa di un Acknowledge, che aveva immagazzinato prima del suo spegnimento; può anche decidere di non fare il restore delle informazioni salvate in precedenza e partire come un nodo "pulito".

Ogni nodo possiede due database fondamentali: uno utilizzato per conoscere il suo intorno, quindi sapere chi sono i suoi vicini in un dato istante di tempo, e l'altro riguardante tutti i percorsi disponibili attraverso i quali raggiungere le destinazioni della rete conosciute. Una volta che il nodo si è acceso, nell'istante in cui ne incontra un altro, avviene la fase di handshake. Questa viene utilizzata per scambiare e conoscere le informazioni riguardati la topologia della rete che ha e che, invece, deve aggiornare grazie alle conoscenze dei suoi vicini. Una volta finita la fase di scoperta, il nodo in questione può occuparsi del forwarding dei bundle secondo le regole stabilite dal protocollo.

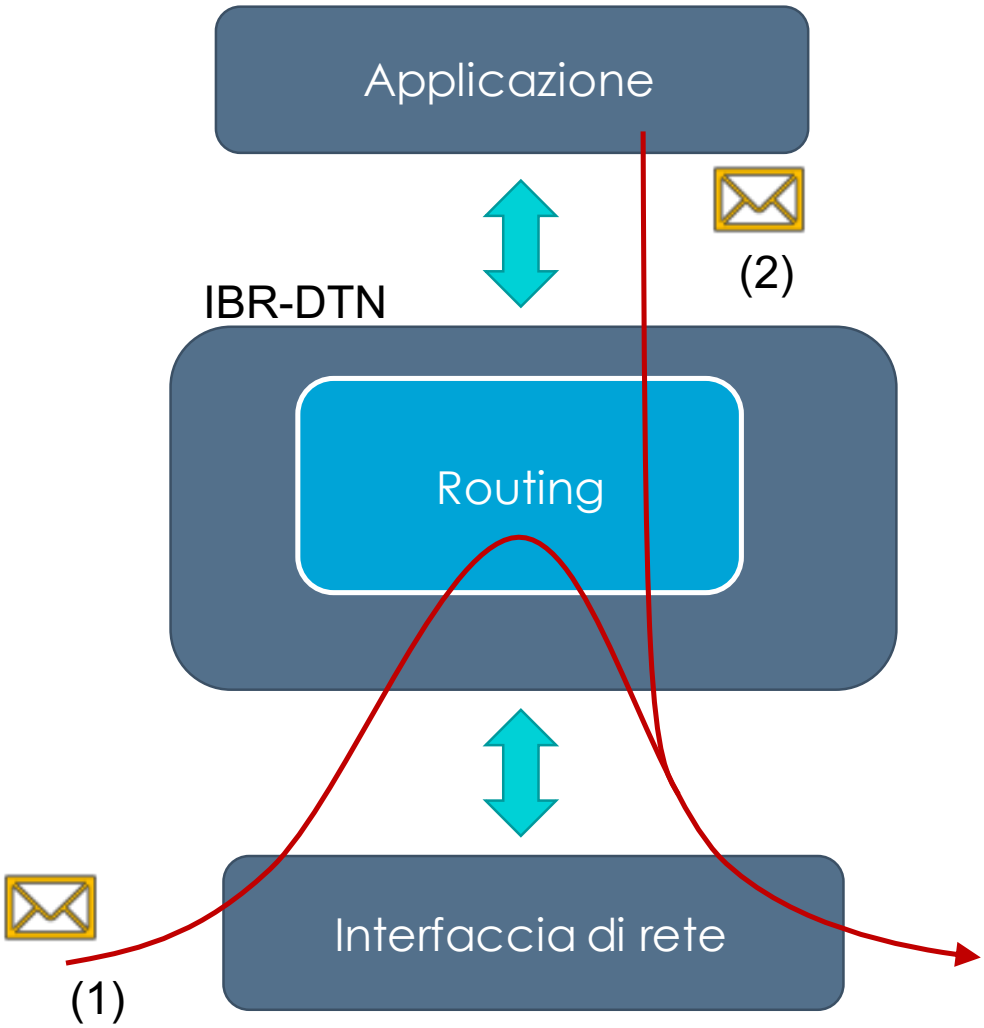


Figura 5.1. Architettura generale

5.1.1 Architettura Generale

In figura 5.1 si può vedere come il modulo di routing sia integrato all'interno del framework IBR-DTN. Si occupa di fare il forwarding dei bundle generati da altri nodi facenti parte della rete (1) ed introdurre quelli creati dal nodo stesso (2). All'interno di IBR-DTN sono presenti anche altri moduli, come la Service Discovery, che forniscono funzionalità differenti e comunicano con il modulo di routing per garantire al meglio lo scambio di informazione nella rete. Le funzioni utili al forwarding implementate nell'algoritmo le vedremo nel dettaglio nel capitolo successivo.

5.1.2 Database

Per far si che ogni nodo conosca la topologia statistica della rete e possa decidere verso chi inoltrare i propri bundle e quelli provenienti da altre sorgenti, deve avere delle conoscenze di ciò che lo circonda. Ogni nodo contiene, quindi, tre informazioni importanti che vengono aggiornate durante la fase di handshake (5.1.3):

- *Tabella dei vicini*
- *Tabella dei percorsi*
- *Tabella degli Acknowledgement*

Tabella dei vicini

Memorizza l'indirizzo di ogni nodo che si incontra ed associato ad una probabilità, secondo la regola descritta in 3.2. In questo modo si otterrà una lista di tutti i nodi incontrati con le rispettive probabilità, la somma delle quali sarà uguale a uno. Questa informazioni ci consente di capire quale nodo sia quello che più probabilmente incontreremo in futuro e verso il quale potrebbe essere meglio inoltrare un bundle nel caso in cui faccia parte del percorso verso la destinazione. Le informazioni all'intero di questa tabella scadono quando le probabilità scendono fino ad arrivare a zero.

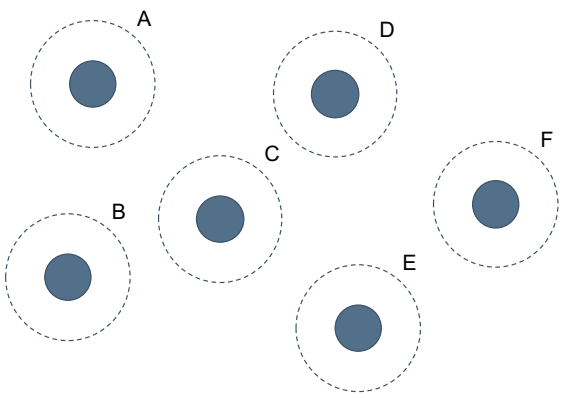


Tabella nodo F

dtn://D	0,75
dtn://E	0,125
dtn://C	0,125

Tabella nodo B

dtn://A	0,5
dtn://C	0,5

Figura 5.2. Tabella dei vicini

La figura 5.2 rappresenta una rete di sei nodi in un istante di tempo in cui non stanno comunicando, perché al di fuori dei range di comunicazione, ma che in precedenza si sono scambiati delle informazioni. Possiamo vedere che il nodo F avrà incontrato più volte il nodo D rispetto ai nodi E e C e non ha mai incontrato

direttamente gli altri nodi, questo non vuol dire, però, che non avrà informazioni a riguardo, ma lo vedremo nella Tabella dei Percorsi (5.1.2). Per quanto riguarda, invece il nodo B, vediamo che ha incontrato lo stesso numero di volte i nodi A e C. Come si evince dalla figura, la somma delle probabilità nelle tabelle risulta essere pari a uno.

Tabella dei percorsi

Questa tabella memorizza tutti i percorsi verso le destinazioni della rete attraverso i vicini, associati ad un valore di costo determinato secondo le regole descritte in 3.2. Ogni nodo avrà, quindi, una lista di percorsi, con il relativo costo, attraverso i quali far passare i bundle verso una data destinazione. Sarà possibile avere anche più percorsi per la stessa destinazione tra i quali il nodo potrà scegliere quello migliore secondo le policies disponibili. Quando scade l'informazione nella *Tabella dei vicini* relativa ad un nodo "vicino", vengono di conseguenza eliminati anche tutti i percorsi passanti per esso.

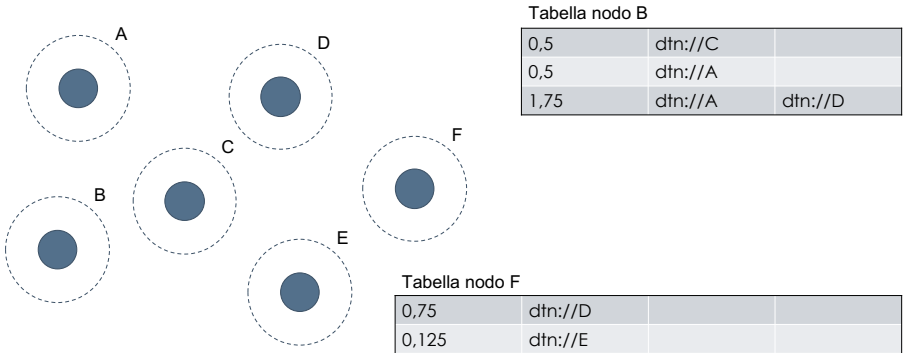


Figura 5.3. Tabella dei percorsi

Nella figura 5.3 possiamo vedere che il nodo F avrà le informazioni riguardanti i percorsi aggiornati dopo l'incontro dei nodi D, E e C. Nella tabella sono presenti quindi i percorsi diretti verso i suoi vicini, con costo pari a $1 - p_i$ dove p_i è la probabilità di incontro di quel nodo e poi avrà i percorsi verso altre destinazioni derivati da una conoscenza dei nodi C e D verso la rete, con costo pari a $C = \sum_{i=0}^{n}(1 - \rho_i)$, dove n sono i nodi del percorso. Per quanto riguarda la tabella del nodo B valgono le stesse considerazioni avrà quindi i percorsi verso i nodi direttamente connessi e un percorso reso noto delle informazioni che A ha della rete. Inizialmente, confrontando le tabelle dei percorsi di nodi diversi, si potrà

riscontrare che due porzioni dello stesso percorso potrebbero non avere lo stesso costo. Questi valori risultano essere diversi poiché le probabilità di incontro che modificano i costi sono in continuo aggiornamento all'interno del nodo stesso, ma non vengono comunicate ai nodi vicini poiché causerebbero una fase di handshake infinita. I vicini sapranno solamente se si aggiunge o toglie qualcuno dal vicinato, ma non se cambia la sua probabilità di incontro.

Tabella degli Acknowledgement

Questa tabella memorizza tutti gli acknowledge riferiti ai bundle in circolazione. Una volta che il bundle arriva a destinazione, il sistema aggiunge all'interno di questa tabella l'informazione che il messaggio è stato ricevuto. Questo viene utilizzato per far sapere alla sorgente ed ai nodi intermedi, responsabili della replica, che il bundle ha raggiunto la destinazione e che possono terminare la fase di inoltro per quel singolo pacchetto ed eliminarlo dallo store. Un bundle può essere eliminato anche nel caso in cui scada il suo tempo di vita, parametro modificabile in fase di configurazione, anche se effettivamente non risulta essere stata notificata la sua ricezione tramite acknowledge.

5.1.3 Handshake

La fase di handshake può essere scatenata dall'incontro di un nuovo nodo, oppure da un segnale periodico che viene mandato ad ogni istante di tempo configurabile prima dell'avvio. La fase di handshake porta allo scambio di tre tipologie diverse di messaggi:

- *Handshake Request*
- *Handshake Response*
- *Handshake Notification*

Nella immagine 5.4, raffigurante un ipotetico esempio di rete formata da tre nodi, vogliamo illustrare i messaggi che vengono scambiati prendendo come riferimento quelli per il nodo B. Quando il nodo B entra in contatto con altri due ipotetici nodi, A e C, viene mandato un messaggio di tipo *Handshake Request* all'indirizzo di broadcast, cosi che entrambi, A e C, ricevano la richiesta di handshake. Una volta ricevuto il pacchetto di *Handshake Request* entrambi i nodi, A e C, risponderanno con una *Handshake Response*. La ricezione di un messaggio di tipo *Handshake Response*, può creare una successiva generazione di pacchetti di tipo *Handshake Notification*, se all'interno della risposta ricevuta si sono ottenuti dei dati sufficienti

a modificare le informazioni, riguardanti la rete, possedute fino a quel momento. Nel caso in cui si sono ricevute informazioni sufficienti da modificare il contenuto dei dati memorizzati, il nodo B genererà una messaggio di *Handshake Notification* all'indirizzo di broadcast, che suggerirà ai nodi A e C di generare a loro volta il processo di scambio di *Handshake Request e Handshake Response* per aggiornare le loro informazioni. Questo accade poiché il nodo B, avendo modificato i suoi database, informerà i vicini che potrebbero esserci state delle modifiche delle quali anche loro devono essere a conoscenza.

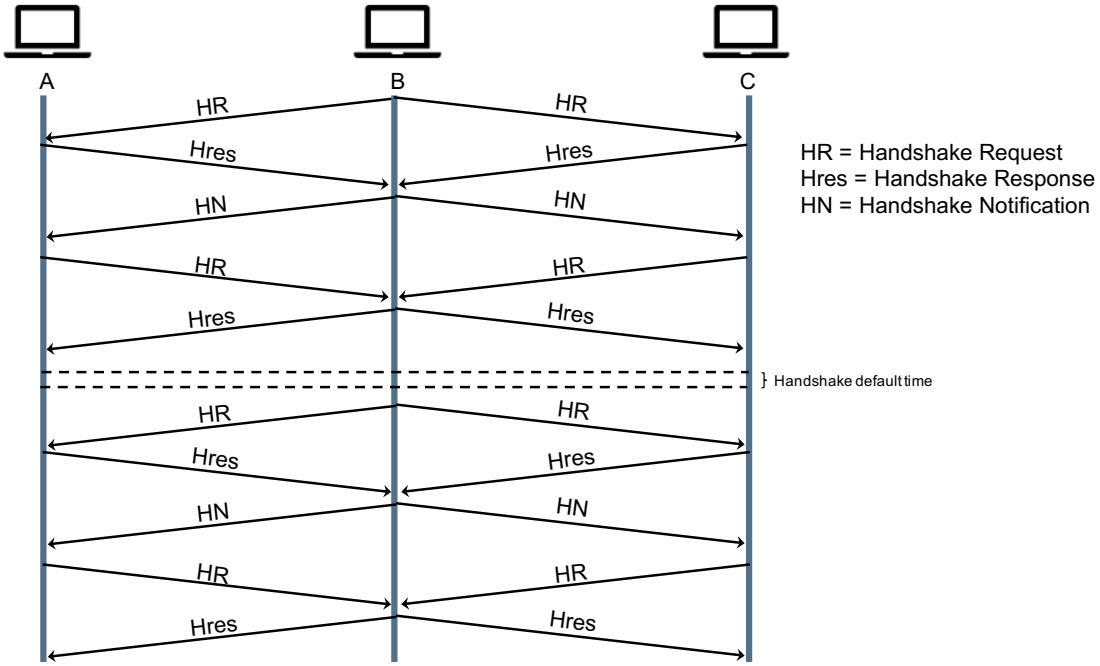


Figura 5.4. Messaggi durante la fase di handshake

Il processo di handshake spiegato, viene utilizzato per far si che ogni nodo possa aggiornare le informazioni riguardanti la topologia della rete e quindi i database descritti in 5.1.2.

Handshake Request Questo messaggio consiste in una richiesta, da parte del nodo che la esercita, di ricevere il database, contenete la *Tabella dei vicini, Tabella dei percorsi e Tabella degli acknowledgement*, del proprio o dei propri vicini.

Handshake Response Questo messaggio, inviato in risposta ad una *Handshake Request*, contiene le informazioni richieste, riguardanti il proprio database. La ricezione di questo messaggio genererà un processo di handshake, dentro al quale

avverranno le modifiche del proprio database con quello ricevuto dal vicino. Nel caso in cui il database venga modificato si genererà, appunto, un messaggio di *Handshake Notification*.

Handshake Notification Quando un nodo riceve questo messaggio, si sottintende che il nodo sorgente, abbia modificato il proprio database. Per modifica si intende che la tabella dei vicini possa aver subito un'aggiunta o un'eliminazione di uno o più nodi, oppure che la tabella dei percorsi abbia aggiunto o tolto uno o più percorsi o che qualche percorso sia stato modificato nel numero di nodi attraversabili. Questo significa che la topologia della rete potrebbe essere cambiata e che si possano essere aggiunti nuovi percorsi. Per questo motivo vengono sollecitati i vicini ad eseguire un processo di handshake per rinnovare i propri database ed essere quindi tutti aggiornati allo stesso istante di tempo. A loro volta i nodi che hanno ricevuto questo messaggio e che hanno eseguito un processo di handshake possono generare un messaggio di *Handshake Notification* nel caso in cui abbiano modificato il proprio database.

Handshake periodico Questa fase di handshake racchiude possibilmente tutti i messaggi descritti fin'ora, ma non viene scatenato dall'accensione, da una notifica o da un incontro. Periodicamente, dopo un certo istante di tempo definito in fase di configurazione iniziale prima dell'avvio, viene avviato il processo di handshake riportato nella figura 5.4, garantendo, cosi, che anche i nodi fermi e sempre connessi si possano comunque aggiornare.

5.1.4 Forwarding

La fase di forwarding rappresenta la parte fondamentale dell'algoritmo, dove vengono prese le decisioni sull'inoltro dei bundle e verso quale percorso. Ogni volta che viene generato un pacchetto o deve essere inoltrato quello generato da qualche altra sorgente, prima viene salvato all'interno dello store dei bundle e cancellato solamente quando verrà ricevuto l'acknowledge di ricezione da parte della destinazione. Ogni volta, quindi, l'algoritmo controlla all'interno dello store, tra tutti i bundle memorizzati, quali sono quelli che devono essere inoltrati ai vicini per raggiungere la destinazione finale. L'algoritmo si occuperà di inoltrare i bundle broadcast verso tutti i suoi vicini e quelli unicast solo verso il vicino corretto. I bundle che hanno come destinazione finale un nodo "vicino" vengono spediti direttamente, mentre gli altri subiscono un controllo. Vengono inseriti all'interno del buffer di inoltro solamente i bundle per i quali: non è stato ricevuto un acknowledge ed esiste un percorso verso la destinazione passante per uno dei vicini. La scelta del percorso può essere fatta secondo due strategie, specificabile nella fase di configurazione:

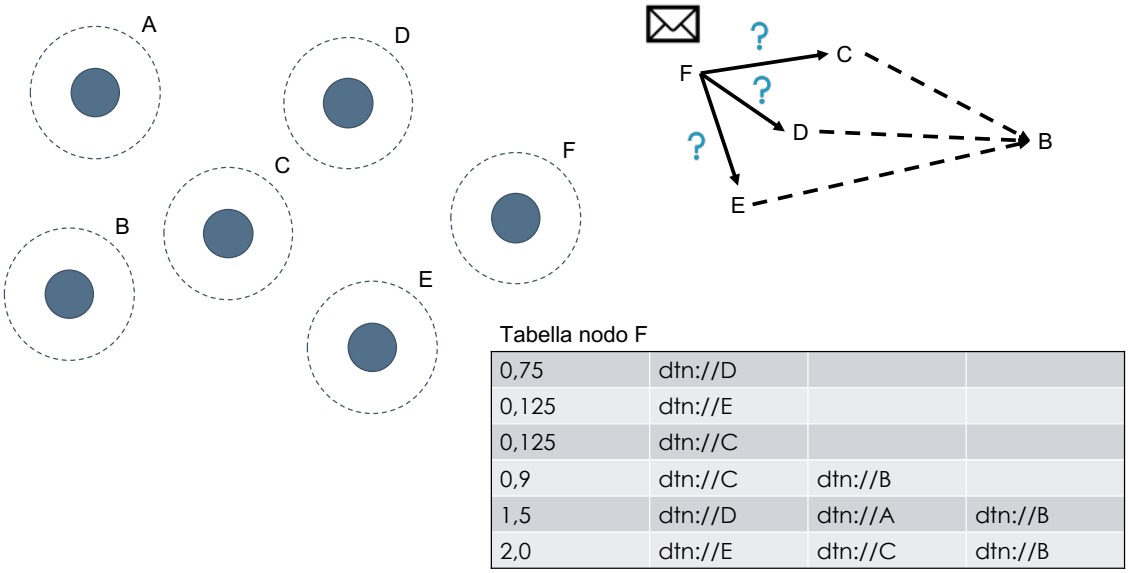


Tabella nodo F

0,75	dtn://D		
0,125	dtn://E		
0,125	dtn://C		
0,9	dtn://C	dtn://B	
1,5	dtn://D	dtn://A	dtn://B
2,0	dtn://E	dtn://C	dtn://B

Figura 5.5. Forwarding: quale percorso scegliere?

- **Most Encountered/Drop Least Encountered (MEDLE) Strategy**, tra tutti i percorsi disponibili verso la destinazione, attraverso i vicini, viene scelto quello con costo inferiore. Questa strategia è migliore quando si ha a che vedere con buffer ridotti. Inoltre, alla lunga, potrebbe generare congestioni nella rete data la continua scelta dei soliti percorsi verso la destinazione

- **Random Strategy**, tra tutti i percorsi disponibili verso la destinazione, attraverso i vicini, ne viene scelto uno in modo casuale. Questo garantisce che alla lunga sia meno probabile riscontrare delle congestioni in rete, non prendendo in considerazione il percorso con costo migliore. Risulta, quindi, meno ottimale sotto questo punto di vista

Una volta scelto verso chi inoltrare il pacchetto, l'algoritmo deve inserire il bundle nel buffer di inoltro. A questo punto va fatta una distinzione sui bundle, in base al numero di hop attraversati. Questo permette l'inserimento dei bundle in due sezioni distinte del buffer, *NEW* o *NORMAL* (Figura 5.6). É importante tener presente che la due liste in realtà fanno parte della stessa struttura che pero viene divisa in due. I bundle presenti all'interno della lista *NEW* verranno ordinati in base al numero di hop attraversati in modo crescente, cosi da garantire a un pacchetto appena generato di entrare subito all'interno della rete; mentre quelli all'interno della lista *NORMAL* verranno ordinati in base al costo del loro percorso verso la destinazione, sempre in modo crescente. Successivamente verrà, quindi, svuotata prima la lista *NEW* e poi quella *NORMAL*.

Il calcolo della soglia p che divide i due buffer viene eseguito ad ogni ciclo d'invio. Vengono presi come parametri di calcolo la dimensione del buffer b e la quantità media di byte trasferiti x. A questo punto se la quantità media di byte trasferiti è minore della metà della dimensione del buffer, $x < b/2$, allora la soglia di divisione sarà pari alla quantità media di byte trasferiti, $p = x$. Se, invece, la quantità media dei byte trasferiti risulta essere maggiore della metà della dimensione del buffer, ma minore dell'intera dimensione, $b/2 <= x < b$, allora la soglia sarà pari al minimo tra la quantità media di byte trasferiti e la sottrazione tra dimensione del buffer e quantità media di byte trasferiti, $p = min(x, b - x)$. L'ultima possibilità prevede che se la quantità media di byte trasferiti risulta essere maggiore della dimensione del buffer, $b < x$, allora la soglia sarà pari a zero, $p = 0$.

Nel caso in cui si voglia inserire un pacchetto all'interno del buffer pieno, prima si andrà ad eliminare il pacchetto messo in ultima posizione che sarà quello con costo di raggiungibilità della destinazione più alto.

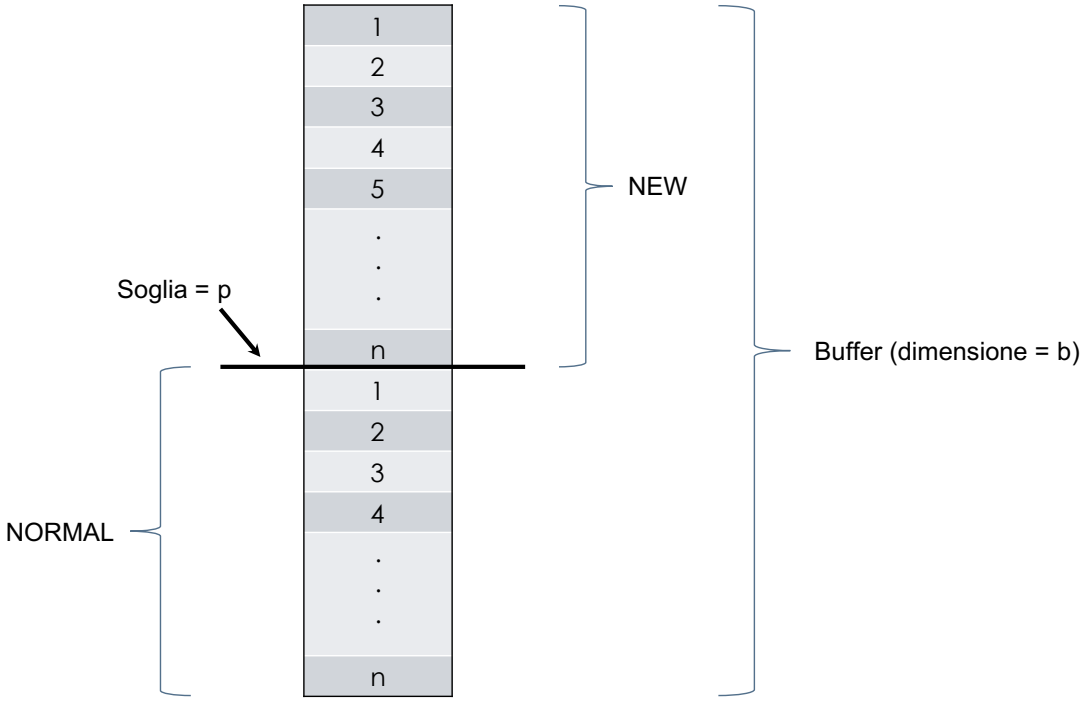


Figura 5.6. Buffer di inoltro composto da da lista NEW e NORMAL

5.1.5 Store e Restore

Le fasi di Store e Restore garantiscono la possibilità ad un nodo di accendersi e spegnersi, senza perdere le informazioni precedentemente raccolte. Quando un nodo

si accende, è possibile scegliere se ripartire con le stesse informazioni memorizzate precedentemente o se riiniziare con tutti i database vuoti. La fase di Restore carica da file o dalla memoria tutte le informazioni legate ai database (5.1.2) ed ai bundle rimasti in sospeso prima dello spegnimento. La fase di Store invece avviene con un periodo configurabile prima del lancio dell'applicazione e consiste nel salvare le informazioni legate al database e a i bundle ancora da inviare in file o in memoria.

5.1.6 Configurazione

Durante la fase di avvio di IBR-DTN si può, attraverso un file di configurazione, impostare alcuni parametri che consentiranno all'algoritmo di MaxProp di avere delle regole definibili ad ogni diverso lancio dell'applicazione. I parametri che possono essere configurati sono:

- **routing** *[epidemic, prophet, maxprop]*: il parametro routing permette di selezionare quale algoritmo di routing si vuol far girare sull'applicazione
- **routing_forwarding** *[yes, no]*: permette di scegliere se si vuole inoltrare un pacchetto verso un alto nodo oppure no
- **routing_accept_nonsingleton** *[yes, no]*: permette di scegliere se accettare anche pacchetti con destinazione broadcast
- **routing_prefer_direct** *[yes, no]*: permette di poter specificare se si vuole inoltrare direttamente un bundle, senza basarsi sulle politiche di routing, nel caso in cui la destinazione sia un vicino
- **limit_bundles_in_transit** *[0...n]*: permette si specificare un numero di bundle massimo in transito nello stesso instante di tempo
- **limit_lifetime** *[0...n]*: permette di specificare la durata di vita di un bundle in secondi, dopo la quale può essere eliminato dallo store
- **maxprop_p_enconuter_first** *[0...1]*: permette di specificare quale deve essere il valore iniziale della probabilità quando un nodo ne incontra un altro che ancora non aveva nel suo database. Di default pari a 1.
- **maxprop_forwarding_strategy** *[MEDLE, Random]*: permette di scegliere quale strategia far utilizzare all'algoritmo
- **maxprop_push_notification** *[yes, no]*: permette di poter scegliere se utilizzare il messaggio di *Handshake Notification* oppure no
- **maxprop_storeTime** *[0...n]*: permette di poter specificare ogni quanti secondi eseguire lo store

- **maxprop_next_exchange_timeout** *[0...n]*: permette di poter scegliere ogni quanti secondi effettuare il processo di hansdhake
- **maxprop_limit_storage** *[0...n]*: permette di definire la dimensione massima del buffer di invio
- **maxprop_restore** *[yes, no]*: permette di poter scegliere se fare il restore in fase di avvio oppure no
- **storage_path**: permette di specificare dove salvare i bundle pendenti ancora da inviare e le informazioni topologiche della rete

Un esempio di configurazione è riportata in figura 5.7.

```
routing = maxprop
routing_forwarding = yes
routing_accept_nonsingleton = yes
routing_prefer_direct = yes
limit_lifetime = 600000
limit_bundles_in_transit = 100

maxprop_p_enconuter_first = 1
maxprop_forwarding_strategy = MEDLE
maxprop_push_notification = yes
maxprop_restore = no
maxprop_storeTime = 30
maxprop_next_exchange_timeout = 180
maxprop_limit_storage = 256000
```

Figura 5.7. Esempio di configurazione iniziale dell'applicazione Aether

5.2 Struttura interna

In questa sezione andremo ad analizzare nel dettaglio la struttura ed i meccanismi interni di cui è composto il modulo di routing e come li usa per svolgere tutte le sue funzioni.

5.2.1 Architettura Interna

L'architettura dell'algoritmo di MaxProp all'interno di un nodo IBR-DTN è sostanzialmente composta da cinque classi principali e altre classi minori contenenti informazioni di contorno. Le classi principali sono:

- **MaxPropRoutingExtension**, è la classe principale dove vengono svolte le operazioni più importanti
- **DeliveryPredictabilityMapMaxProp**, è la classe che gestisce il database dell'algoritmo (5.1.2 e 5.1.2)
- **ForwardingStrategyMaxProp**, è la classe che implementa le diverse strategie di forwarding descritte in (5.1.4)
- **AcknowledgementSetMaxProp**, è la classe che contiene parte del database riferito solo a quella delle tabelle di acknowledge (5.1.2)
- **RoutingResultMaxProp**, è la classe che implementa la struttura del buffer di inoltro dei pacchetti (5.1.4)

Le classi interne al MaxProp routing collaborano tra di loro, con quelle appartenenti all'intero modulo di routing e con quelle esterne al fine di garantire un instradamento dei pacchetti ottimizzato, come possiamo vedere in figura 5.8.

5.2.2 MaxPropRoutingExtension

Questa è la classe principale per il funzionamento dell'algoritmo; implementa al suo interno diversi metodi che vengono chiamati dalla classe di base *BaseRouter* (2.3.1). Durante l'avvio di Aether la classe *NativeDaemon*, facente parte delle classi di IBR-DTN esterne al modulo di routing, in base al tipo di configurazione data, decide di far partire l'algoritmo di routing scelto. Nel nostro caso dopo aver specificato che l'algoritmo scelto sia MaxProp, *NativeDaemon* chiama il costruttore della classe *MaxPropRoutingExtension* passandogli come parametri quelli descritti in 5.1.6 inizializzando di conseguenza tutte le variabili. Di seguito sono riportate le principali funzioni di questa classe:

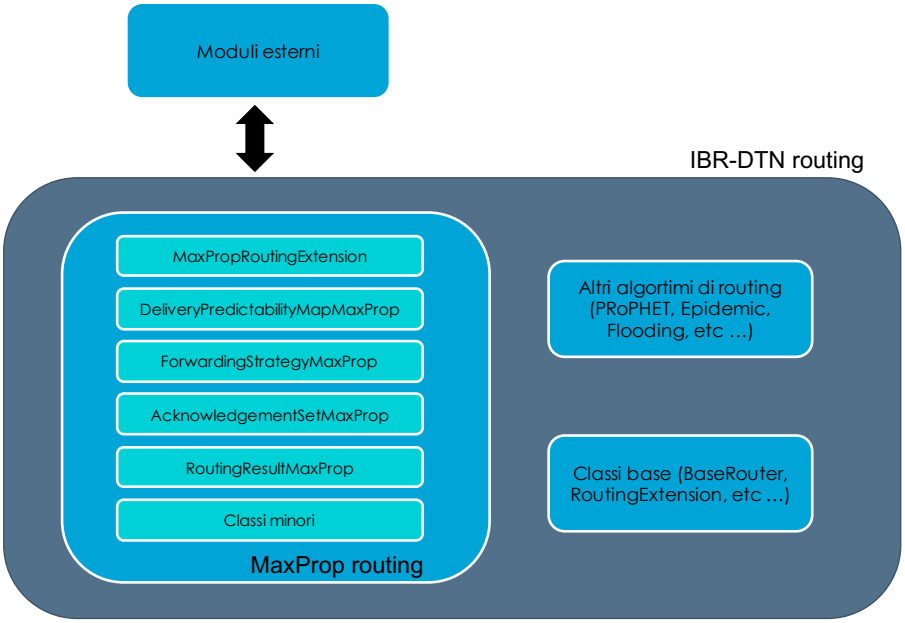


Figura 5.8. Architettura interna del modulo di routing di IBR-DTN

- **componentUp**: viene chiamata per far partire il thread principale ed iscriverlo a gli eventi per i quali sarà interpellato quando verranno scatenati. Gli eventi principali a cui MaxProp è iscritto sono 3: *NodeHandshakeEvent*, utilizzato nella fase di handshake, *TimeEvent*, utilizzato per eseguire le operazioni periodiche, *BundlePurgeEvent*, utilizzato per aggiungere un bundle alla tabella degli acknowledge dopo aver constatato di essere la destinazione finale. Dopo la fase di iscrizione a gli eventi, se abilitata in configurazione, viene avviata la fase di restore con le informazioni salvate in precedenza.

- **componentDown**: in questa fase viene disiscritto il modulo dagli eventi precedentemente descritti, fatto lo store delle informazioni, se possibile, e terminato il processo.

- **requestHandshake**: questa funzione viene chiamata in fase di handshake per decidere cosa richiedere all'interno del messaggio di *Handshake Request*. In questa fase viene richiesto il database del nodo vicino (5.1.2). Nella porzione di codice sottostante si può vedere come viene fatta questa richiesta:

```
1 handshake.addRequest(neighborsTable, pathsTable);
2 handshake.addRequest(acknowledgementsTable);
```

- **responseHandshake**: una volta ricevuta la richiesta di handshake, verrà generata una risposta. Vengono controllate quali informazioni sono state

richieste dal nodo vicino e di conseguenza confezionate e inviate nel seguente modo:

```
if (request.hasRequest(neighborsTable))
{
  response.addItem(new neighborTable(_neighborTable));
}
if (request.hasRequest(pathsTable))
{
  response.addItem(new pathsTable(_pathsTable));
}
if (request.hasRequest(acknowledgementsTable))
{
  response.addItem(new acknowledgementsTable(_acknowledgementsTable));
}
```

- **processHandshake**: in questa funzione viene eseguita tutta la fase di aggiornamento del database al termine della ricezione di un *HandshakeResponse*; descritta nel dettaglio in (5.2.2).

- **eventDataChanged**: questa funzione viene lanciata dal modulo di *BaseRouter* o da alcune funzioni descritte in seguito per informare che ci sarebbe un bundle da inviare verso il next hop suggerito (un nodo vicino). In questo caso viene inserita questa informazione in una coda, (*_taskqueue*), che verrà processata poi nella funzione di run e fatti i relativi controlli per il forwarding del bundle in questione.

```
_taskqueue.push(new bundle_to(peer) );
```

- **eventTransferSlotChanged**: questa funzione viene eseguita per re-inserire all'interno della coda dei prossimi bundle da inviare, quelli che precedentemente non sono stati inviati per motivi di rigetto all'interno della funzione run. Proveranno ad essere inviati all'iterazione successiva, essendo sono stati salvati in una coda temporanea e non scartati. Viene quindi chiamata la funzione *eventDataChanged* per ogni bundle salvato in questa coda provvisoria.

- **eventBundleQueued**: chiama anche essa *eventDataChanged* che inserisce all'interno della coda, (*_taskqueue*), tutti i next hop suggeriti per inviare i bundle in attesa

- **eventTransferCompleted**: viene chiamata quando avviene un trasferimento e aggiorna la quantità di byte trasferiti, che servirà poi per calcolare il threshold della lista *NEW e NORMAL* (5.1.4)

- **run**: questa funzione è quella più importante, è il cuore dell'algoritmo di routing e si occupa di fare il forwarding dei pacchetti; verrà descritta nel dettaglio in (5.2.2)

- **updateNeighbor**: questa funzione viene lanciata dal *Process Handshake* per aggiornare il database; ; verrà descritta nel dettaglio in (5.2.2)

- **store**: questa funzione si occupa di eseguire lo store delle informazioni del database di MaxProp e dei pacchetti non ancora inoltrati

- **restore**: questa funzione si occupa di eseguire il restore delle informazioni

Eventi

- **NodeHandshakeEvent**, lo scatenamento di questo evento viene eseguito durante la fase di handshake, al punto in cui deve essere inviato il pacchetto contenente delle informazioni riguardanti l'handshake. Catturato l'evento, si occuperà di inserire all'interno di una coda, *_taskqueue*, il next hop suggerito, verso il quale dovrà essere inviato il pacchetto di handshake nella funzione di *run*.

```
13 _taskqueue.push(new bundle_handshake_to(handshake.peer));
```

- **TimeEvent**, questo evento viene scatenato periodicamente ogni secondo. Catturato l'evento, la funzione si occuperà di controllare se ci sono dei pacchetti scaduti, e di conseguenza da cancellare, all'interno della tabella degli acknowledge:

```
1 _acknowledgementSetMaxProp.expire(time.getTimestamp());
```

  Successivamente si occuperà di controllare se sia passato il tempo necessario per eseguire il periodico store delle informazioni. Infine controlla se sia il momento per eseguire il prossimo periodico handshake:

```
1 if ((_next_exchange_timestamp > 0) && (_next_exchange_timestamp < now))
2 {
3   _taskqueue.push(new handshake);
4   _next_exchange_timestamp = now + _next_exchange_timeout;
5 }
```

- **BundlePurgeEvent**, questo evento viene scatenato quando il nodo in questione riconosce di essere la destinazione finale e quindi inserisce il bundle nella tabella degli acknowledge in modo da cancellarlo e far sapere anche agli altri nodi della rete che possono fare lo stesso, nel caso in cui avessero memorizzato tale bundle.

processHandshake

La funzione di *processHandshake*, è una delle funzioni più importanti, perché si occupa di leggere le risposte alle *Request Handshake* inviate ai nodi vicini. Nella prima fase estrae, dalla *Response Handshake* inviata dal vicino, la tabella dei percorsi e dei vicini, le memorizza e chiama la funzione *updateNeighbor* per aggiornare il suo database con le informazioni appena ricevute.

```
1 neighbor_map = response.get<Tables>();
2 ...
3 updateNeighbor(neighbor_name, neighbor_map);
```

Successivamente si occupa di controllare anche la tabella degli acknowledge ricevuta nell'*Handshake response* insieme alle altre e unirla a quella corrente. A questo punto controlla se ci sono dei bundle nello storage che deve cancellare perché arrivati destinazione. Possiamo specificare che i bundle verranno cancellati dallo storage solo quando la destinazione avrà inserito tale informazione nella tabella degli acknowledge in modo da farle fare il giro della rete.

run

É la funzione principale, dove viene svolta la parte più importante del forwarding. Questa funzione non muore mai, se non al termine del processo main. Lo scopo è quello di processare la coda *_taskqueue* contenente tutti i nodi vicini per i quali è possibile che ci sia un pacchetto da inoltrare, da ora in avanti li chiameremo **Task**. Ci sono due tipi di task: **SearchNextBundleTask** e **NextExchangeTask**. Il primo utilizzato per informare il modulo della richiesta di inoltro di una qualsiasi categoria di bundle, il secondo, invece, riferito a quei bundle che vengono generati dall'handshake periodico.

Il primo passo consiste nel calcolare il numero di task presenti in quel momento all'interno della coda e successivamente vengono estratti dalla medesima; nel frattempo la coda può essere riempita lo stesso con nuovi task che verranno però processati al prossimo ciclo.

```
1 Task t = _taskqueue.take();
```

Dopo aver estratto il task dalla coda, bisogna capire di che tipo sia. Se il task è di tipo **SearchNextBundleTask**, quindi riferito ad un vicino che potrà trattare un bundle di carattere generale, viene caricato dalla memoria tutto lo store dei bundle memorizzati che potrebbero avere il nodo vicino, identificato dal task, come prossimo next hop.

```
1 SearchNextBundleTask task = dynamic_cast<SearchNextBundleTask>(t);
2 ...
```

```
3 NeighborDatabase db = getNeighborDB();
4 NeighborDatabase::NeighborEntry &entry = db.get(task.eid, true);
```

A questo punto entra in funzione la parte vera e propria dell'algoritmo di forwarding, dove verrà presa la decisione di scelta o meno di inoltro verso un determinato nodo e di conseguenza di inserimento all'interno del buffer di uscita.

```
1 const BundleFilter filter(entry, _forwardingStrategy, database, neighbors, ... , hopThreshold,
      _acknowledgementsTable);
```

Questa operazione viene eseguita, passando i pacchetti presi dallo store in precedenza, attraverso un filtro. Il filtro deciderà se il pacchetto in questione verrà inoltrato verso il task che si sta processando oppure no. Questo significa che verrà deciso se il task che rappresenta un vicino sia quello giusto verso il quale inoltrare il pacchetto. All'interno del filtro, inizialmente, si controlla se il pacchetto ha attraversato già il numero di hop massimi. In caso positivo si annulla il forwarding.

```
1 if (bunlde.hopcount == 0)
2 {
3     return false;
4 }
```

Successivamente viene controllato se il bundle in questione è destinato al nodo corrente:

```
1 if (bundle.destination == me && meta.no_broadcast)
2 {
3     return false;
4 }
```

se questi controlli danno esito positivo verrà annullato l'inoltro verso il task processato. In seguito viene controllato se il bundle ha come destinazione finale un nodo direttamente connesso, nemmeno in questo caso viene preso in considerazione perché se ne occuperà un altro modulo di IBR-DTN ad inoltrarlo. Ciò avviene perché essendo direttamente connesso, possiede un percorso privilegiato che non ha bisogno del processo di forwarding e scelta di strategia.

```
1 if (task == meta.destination.getNode())
2 {
3     return false;
4 }
```

Vengono, inoltre fatti dei controlli sulla validità del pacchetto; non vengono inoltrati i pacchetti verso il next hop che è anche la sorgente e quelli già presenti nella tabella degli acknowledge. A questo punto, se la destinazione del pacchetto è un singolo nodo e non un gruppo (broadcast), viene scelto se inoltrare effettivamente il pacchetto verso il nodo identificato dal task oppure no, attraverso delle scelte di

strategia. La funzione di *shallForward* chiama una funzione di scelta del percorso diversa a seconda di quale strategia è stata impostata in fase di configurazione e restituisce il next hop giusto, verso il quale inoltrare il pacchetto con il relativo costo.

```
if (meta.no_broadcast)
{
  [nexthop, cost] = _strategy.shallForward(_database, bundle);
  pathCost = cost;
  if (nexthop != task)
      return false;
  if (cost == -1)
    return false;
}
```

Le diverse strategie, descritte nel capitolo 5.1.4, verranno spiegate in seguito nel dettaglio (5.2.4). Una volta deciso se il task in questione è idoneo a ricevere il pacchetto, bisogna scegliere se inserire il bundle nella lista *NEW o NORMAL* (5.1.4). Inizialmente è stato calcolato il threshold che caratterizzerà la distinzione tra i pacchetti che verranno inseriti in una lista o nell'altra. Questo calcolo, come spiegato in 5.1.4, ha bisogno di sapere la media dei byte trasferiti fino a quel momento e la dimensione del buffer:

```
hopThreshold = _forwardingStrategy->Threshold(bufferSize, AVGtransferredBytes);
```

Ora se il numero di hop attraversati è minore del threshold calcolato, i pacchetti verranno inseriti nella lista *NEW*, altrimenti verranno inseriti nella lista *NORMAL*:

```
if(realHopCount <= _hopThreshold)
{
    buffer.putNewerBundles(realHopCount, bundle, task);
}
else
{
    buffer.put(pathCost, bundle, task);
}
return true;
```

Nel caso in cui il buffer fosse pieno, viene cancellato il bundle con costo di consegna più alto, che risulterà trovarsi in ultima posizione:

```
if(bufferIsFull(buffer.bufferSize, buffer, bundle.getPayloadLength()))
    buffer.pop();
```

A questo punto il bundle si ritroverà nel buffer di inoltro e sarà pronto alla trasmissione. Se invece si tratta di un **NextExchangeTask**, il procedimento è differente. In questo caso si tratta, solamente, di inviare a tutti i nodi vicini il pacchetto di richiesta di handshake:

```
dynamic_cast<NextExchangeTask&>(*t);
...
for(it = neighbors.begin(); it != neighbors.end(); ++it)
    doHandshake(it->id);
```

Arrivati a questo punto bisogna procedere con l'invio effettivo dei bundle all'interno del buffer di inoltro. Prima dell'invio bisogna procedere con l'ordinamento della lista *NEW e NORMAL* in base al numero di hop e ai costi, rispettivamente. Per inviare i bundle presenti nel buffer, si svuotano in ordine la lista *NEW* e la lista *NORMAL* attraverso la funzione *transferTo*:

```
buffer._listNEW.sort();
buffer._listNORMAL.sort();
for(auto itl = list._listNEW.begin(); itl != list._listNEW.end(); ++itl)
{
...
    transferTo((*itl).second._nextHop, (*itl).second._meta, (*itl).second._protocol);
...
}
...
for (auto iter = list._listNORMAL.begin(); iter != list._listNORMAL.end(); ++iter)
{
...
    transferTo((*iter).second._nextHop, (*iter).second._meta, (*iter).second._protocol);
...
}
```

Nel caso in cui il trasferimento si fosse bloccato, i bundle che dovevano ancora essere inviati vengono presi e inseriti in una lista di bundle pendenti. I bundle pendenti all'interno della lista verranno successivamente riconsiderati per l'inoltro, al ciclo successivo, come spiegato in precedenza per il metodo *eventTransferSlotChanged*.

```
size_t sizeNEW = list._listOfNewerBundles.size(), sizeNORMAL = list._listNORMAL.size();
if(i < sizeNEW){
  for(k = i; k < sizeNEW; k++){
      _pending_peers.insert(list._listNEW.back().second._nextHop);
    list._listNEW.pop_back()
  }
}
if(j < sizeNORMAL){
  for(h = j; h < sizeNORMAL; h++){
      _pending_peers.insert(list._listNORMAL.back().second._nextHop);
    list._listNORMAL.pop_back();
  }
}
```

updateNeighbor

Questa funzione è molto importante perché, scatenata dal *processHandshake*, chiama le funzioni del database per il suo aggiornamento. Come prima cosa, se il parametro delle notifiche è stato impostato su *yes*, durante la configurazione, viene fatto un controllo dello stato della tabella dei vicini e dei percorsi. Per controllo sullo stato si intende una verifica di quali nodi e quali percorsi sono rispettivamente presenti nella tabella dei vicini e nella tabella dei percorsi. In seguito, viene chiamata la funzione di aggiornamento delle tabelle, che verrà spiegata nel dettaglio in 5.2.3.

```
database.updateTables(neighbor, neighbor_database, _p_encounter_first);
```

Una volta eseguito l'aggiornamento del database, viene rifatto il controllo sullo stato della tabella dei vicini e dei percorsi, e se si trovano delle differenze, rispetto allo stato calcolato prima dell'aggiornamento, viene inviata una *Handshake Notification*. Questo avviene per far si che anche i vicini ad un nodo che ha subito dei cambiamenti possano aggiornarsi e rimanere tutti sincronizzati.

```
if (new_endpoints.size() > 0 || old_endpoints.size() > 0)
{
...
   const dtn::data::Timestamp now = dtn::utils::Clock::getMonotonicTimestamp();
   _next_exchange_timestamp = now + _next_exchange_timeout;
   pushHandshakeUpdated( handshake_notification );
...
}
```

Durante l'invio di questa notifica, viene fatto ripartire il timer dell'handshake periodico, di modo da non eseguire un altro handshake troppo ravvicinato.

5.2.3 DeliveryPredictabilityMapMaxProp

Questa classe gestisce tutto il funzionamento del database, caricamento, aggiornamento e accesso a tutte le strutture dati che l'algoritmo MaxProp necessita per funzionare. Viene utilizzata esclusivamente dal modulo di rouitng MaxProp. Durante l'avvio di *MaxPropRoutingExtension* viene inizializzato il database, chiamando il costruttore della classe, e passati le variabili di base prelevate dal file di configurazione (5.1.6). All'interno di questa classe troveremo le nostre strutture base: *Tabella dei vicini* (5.2) e *Tabella dei percorsi* (5.3).

```
std::map<dtn::data::EID, float> probTable;
std::vector<PathsCostMapParameter> pathsCostMap;
```

La classe **PathsCostMapParameter** contiene il costo e la lista dei nodi di ogni singolo percorso verso le destinazioni conosciute.

Le funzioni principali di *DeliveryPredictabilityMapMaxProp* sono:

- **getLength, serialize, deserialize**: queste tre funzioni, vengono utilizzate per inviare e leggere le informazioni trasmesse o ricevute da un nodo vicino. Permettono di serializzare le informazioni attraverso delle regole precise, in modo che il ricevente attraverso il deserialze possa sapere come estrapolarle dal pacchetto e memorizzarle
- **updateTables**: aggiorna la tabella dei percorsi e dei vicini; descritta nel dettaglio in 5.2.3
- **store, restore**: queste funzioni vengono utilizzate per salvare e caricare le informazioni del database. Vengono chiamate dalle omonime funzioni presenti nella classe *MaxProproutingExtension* ed eseguono il vero e proprio compito di store e restore da e per la memoria.
- **getPaths**: questa funzione, data una destinazione, controlla se nella tabella dei percorsi esiste una strada verso essa. In ogni caso ritorna una lista di nodi, rappresentanti i next hop e il loro costo verso la destinazione. Se non sono stati trovati percorsi, la lista sarà vuota, altrimenti ci saranno uno o più next hop disponibili per raggiungere la destinazione. Viene lasciato come primo della lista il nodo con minor costo di percorso verso la destinazione.

```
std::list<nexthop, cost> paths;
for(auto it = _pathsTable.begin(); it != _pathsTable.end(); ++it){
    ...
    if(it->_path.back().sameHost(destination) && value > it->_cost){
        value = it->_cost;
        nextHop = it->_path.front().getString();
        const dtn::data::EID nextHop(nextHopSTR);
        paths.push_front(nextHop, it->_cost);
    }
    else if(it->_path.back().sameHost(destination)){
        nextHopSTR = it->_path.front().getString();
        const dtn::data::EID nextHop(nextHopSTR);
        paths.push_back(nextHop, it->_cost);
    }
}
return paths;
```

updateTables

Questa funzione è la più importante. Si occupa di tenere aggiornate la tabella dei percorsi e dei vicini. Inizialmente viene modificata la tabella dei vicini, viene presa

la probabilità di incontro del nodo appena incrociato, già memorizzata in precedenza, e la si modifica. Se è un nodo nuovo viene inserito direttamente attraverso le regole dell'algoritmo. Il calcolo delle probabilità, sfrutta i meccanismi spiegati in 3.2.

```
try{
  old_probability = getProbTable(host_b);
     _neighborsTable.erase(host_b);
  new_probability = (old_probability+1)/2;
  alreadyExistNode = true;
} catch (ValueNotFoundException&) {
  old_probability = p_encounter_first;
  new_probability = (old_probability+1)/2;
}
setNeighborsTable(neighbor_name,new_probability);
```

Una volta modificato il valore del nodo incontrato, bisogna normalizzare tutti gli altri, in modo da avere una probabilità totale pari a uno. A questo punto vanno modificati, anche, tutti i costi dei percorsi passanti per i nodi di cui si è appena variata la probabilità di incontro, con i valori appena ottenuti dopo la normalizzazione.

```
for(auto it = _neighborsTable.begin(); it != _neighborsTable.end(); ++it)
{
  if(_neighborsTable.size() > 1){
    if(alreadyExistNode && it->first == neighbor_name){
      ...
      continue;
    }
    float old_pathProb = it->second;
      it->second = it->second/2;
    if(_pathsTable.size() > 0)
      for(auto itpath = _pathsTable.begin(); itpath != _pathsTable.end(); ++itpath){
        if(itpath->_path.front().sameHost(it->first)){
          itpath->_cost -= (1 - old_pathProb);
          itpath->_cost += (1 - it->second);
          ...
        }
      }
  }
}
```

Ora è necessario aggiornare i percorsi con le informazioni contenute nella tabella ricevuta dal vicino. Vengono presi tutti quelli passanti per il nodo appena incontrato e cancellati. Successivamente dalla tabella dei percorsi ottenuta dal vicino, vengono presi tutti i quelli, non passanti per il nodo corrente e con percorsi più lunghi passanti per nodi già conosciuti, e aggiunti nella propria tabella dei percorsi mettendo come next hop il nodo incontrato. Viene, inoltre, aggiornato il costo in base alla probabilità di incontro calcolata in precedenza.

```
if(alreadyExistNode){
  _pathsTable::iterator it = _pathsTable.begin();
  while (it != _pathsTable.end()) {
    if(it->_path.front().sameHost(host_b)){
      it = _pathsTable.erase(it);
    } else {
       ++it;
    }
  }
}

if(_pathsTableNeighbor.size() != 0){
  for(auto it = _pathsTableNeighbor.begin(); it != _pathsTableNeighbor.end(); ++it){
    bool pathNotPassesThroughtMe = true, pathNotNecessary = false;
    for (auto itl = it->_path.begin(); itl != it->_path.end(); ++itl)
      if(itl->sameHost(local)){
        pathNotPassesThroughtMe = false;
        break;
      }
    for (auto itmyMap = _pathsTable.begin(); itmyMap != _pathsTable.end(); itmyMap++){
      if(std::find(it->_path.begin(), it->_path.end(), itmyMap->_path.front()) !=
          it->_path.end()){
        pathNotNecessary = true;
        break;
      }
    }

    if(pathNotPassesThroughtMe && !pathNotNecessary){
      std::list<EID> newPath = it->_path;
      newPath.push_front(host_b);
      float pathCost = it->_cost + (1 - getNeighborsTable(host_b));
        ...
      _pathsTable.push_back(PathsCostMapParameter(pathCost, newPath));
    }
  }
}

std::list<EID> newList;
newList.push_back(neighbor_name);
_pathsTable.push_back(PathsCostMapParameter((1 - getNeighborsTable(neighbor_name)),
    newList));
```

Dopo aver aggiornato il database, inserendo e modificando valori, potrebbe capitare di dover cancellare qualche entry, essendo stato raggiunto il threshold minimo pari a zero. In questo caso vanno cancellati tutti i nodi dalla tabella dei vicini e i percorsi passanti per esso, se hanno, appunto, una probabilità di incontro pari a zero.

5.2.4 ForwardingStrategyMaxProp

All'interno di questa classe, vengono gestite le diverse strategie. A seconda di quale strategia è stata scelta in fase di configurazione, verrà fatto partire uno dei tre metodi:

- **pathFinder_ME_DLE**, questa strategia permette di scegliere come next hop quello con il costo minore verso la destinazione. Se non dovesse esistere un nodo verso la destinazione ne viene ritornato uno vuoto, e di conseguenza il pacchetto non verrà spedito.

```
    std::list<EID, float> listNewNodes = dpm.getPaths(destination);
  if(!listNewNodes.empty()){
    const EID newNode(listNewNodes.front().first);
    return [newNode, listNewNodes.front().cost];
  }
  else{
    const EID newNode;
    return [newNode, −1.0];
  }
```

- **pathFinder_Random**, questa strategia, invece, in un insieme di next hop possibili verso la destinazione, non sceglie quello con il costo minore, ma ne sceglierà uno casuale tra essi. Questo garantisce di evitare delle congestioni dovute alla scelta del solito percorso data la stessa destinazione. Il codice utilizzato per la scelta è pressoché lo stesso della strategia precedente, con l'aggiunta di una randomizzazione della lista dei next hop possibili verso la destinazione.

```
    std::random_device randomizer;
  std::mt19937 mt(randomizer());

  if(listNewNodes.empty()){
    dtn::data::EID nodeEmpty;
    return [newNode, −1.0];
  }

  if(listNewNodes.size() == 1)
    return newNodes.front();

  shuffle(listNewNodes.begin(), listNewNodes.end(), mt);
  return listNewNodes.front();
```

Un altro metodo importante in questa classe è **newPacketThreshold** che viene sfruttato per calcolare il nuovo threshold, utilizzato per decidere poi se un pacchetto finirà nella parte del buffer *NEW o NORMAL*. Il calcolo viene eseguito come

spiegato in 5.1.4 e di seguito possiamo vedere la sua implementazione all'interno del codice.

```
if(AVGTrasferredBytes < bufferSize/2){
  return (int) AVGTrasferredBytes;
}
else if(AVGTrasferredBytes < bufferSize && bufferSize/2 <= AVGTrasferredBytes){
  size_t min = AVGTrasferredBytes;

  if ((bufferSize - AVGTrasferredBytes) < min)
    min = bufferSize - AVGTrasferredBytes;
  return (int) min;
}
else
  return 0;
```

5.2.5 AcknowledgementSetMaxProp

Questa classe raffigura una parte del database, (5.1.2), riferita alla tabella degli acknowledge. Presenta una lista di bundle di cui è stato ricevuto l'acknowledge che si vogliono propagare al vicino in modo da eseguire la loro cancellazione dallo storage. Ci sono quattro metodi principali:

- **add**, permette di aggiungere un bundle all'interno della lista
- **merge**, permette di unire due liste di bundle differenti, ad esempio quella del nodo corrente con quella inviata dal vicino
- **has**, permetta di sapere se è presente, all'interno della lista, uno specifico bundle
- **expire**, permette di cancellare dalla lista un bundle con il tempo di vita scaduto

5.2.6 RoutingResultMaxProp

All'interno di questa classe vengono definite le due liste *NEW e NORMAL* che compongono il buffer di inoltro. Entrambe le liste, anche se divise, rappresentano lo stesso buffer di inoltro, quindi la sua dimensione è la somma delle due. La dimensione totale del buffer di inoltro viene decisa in fase di configurazione; una volta che viene raggiunta la dimensione massima, vengono eliminati i bundle partendo dal fondo del buffer, quelli con costo di raggiungibilità della destinazione più alto. La lista *NEW* viene utilizzata per controllare quali pacchetti hanno attraversato meno hop rispetto agli altri, e inviati in ordine crescente.

```
std::list<std::pair<size_t, TransferMaxPropInformation>> listOfNewBundles
```

Utilizza il metodo **putNewerBundles** per inserire le informazioni al suo interno.

```
push_back(std::make_pair(hopCount, TransferMaxPropInformation(bundle, p, nextHop)));
```

La lista *NORMAL* viene utilizzata, invece, per controllare quali pacchetti hanno un costo verso la destinazione minore ed inviarli in ordine crescente.

```
std::list<std::pair<float, TransferMaxPropInformation>> listOfNormalBundles
```

Utilizza il metodo **put** per inserire le informazioni al suo interno.

```
push_back(std::make_pair(prob, TransferMaxPropInformation(bundle, p, nextHop)));
```

Viene infine utilizzato il metodo **pop** per eliminare l'ultimo elemento dall'intero buffer.

TransferMaxPropInformation

Questa classe la troviamo come secondo elemento all'interno delle liste *NEW e NORMAL*. Ci serve per identificare i parametri fondamentali per l'inoltro dei pacchetti: il bundle vero e proprio da inviare, il next hop verso il quale inoltrare il pacchetto e il protocollo da utilizzare.

```
dtn::data::MetaBundle _meta;
dtn::core::Node::Protocol _protocol;
dtn::data::EID _nextHop;
```

Capitolo 6

MaxProp routing Service-Oriented su IBR-DTN

In questo capitolo si presenterà la strategia di forwarding service-oriented che è stata implementata sulla base dell'algoritmo già esistente, illustrato nel capitolo precedente, di MaxProp routing. Questa scelta nasce dal fatto che Aether viene utilizzata da applicazioni che vogliono fornire un servizio all'interno di una rete dove i nodi sono posizionati in aree limitate con movimenti circoscritti ad aree note più piccole. Si pensi ad esempio ad un cantiere di lavoro, dove i nodi si muovono seguendo dei percorsi probabili e che spesso saranno in connessione. In una rete DTN basarsi sulle latenze tra i nodi potrebbe essere un ossimoro, ma date le premesse sullo scenario in cui si pensa di volere utilizzare l'applicazione si è voluto implementare una strategia che tenesse conto della latenza tra il nodo sorgente e destinazione per la scelta del percorso migliore.

6.1 Funzionamento e caratteristiche

Come accennato nella premessa, questa nuova strategia è stata aggiunta alle altre già presenti nell'algoritmo. Le grandi modifiche rispetto a quello spiegato nei capitoli precedenti riguardano il database, con l'aggiunta di nuove strutture per la memorizzazione dei dati sui tempi, e la fase di forwarding, con l'implementazione della nuova strategia vera e propria per la scelta del percorso più idoneo. Di seguito verranno introdotte e spiegate le modifiche a livello generale attuate all'algoritmo, valgono tutte le caratteristiche di base specificate nel capitolo precedente e rimangono funzionanti anche in questo caso.

6.1.1 Database

Per farsi che ogni nodo conosca la latenza verso i suoi vicini, è stata aggiunta una nuova struttura in grado di immagazzinare tale informazione.

- Tabella delle latenze

La **Tabella dei percorsi** ha subito delle modifiche in modo da mantenere anche l'informazione della latenza totale dei percorsi memorizzati. In questo caso i messaggi di Handshake saranno gli stessi, ma nella tabella dei percorsi ci sarà un informazione in più, riguardante appunto la latenza dei percorsi da sorgente a destinazione.

Tabella delle latenze

Consiste nel memorizzare l'indirizzo di ogni nodo che si incontra e la media esponenziale mobile calcolata sulla base dei vari incontri effettuati come: $Avg = (1 - a) * Avg + a * s$, dove a è un coefficiente e s è il campione riferito al tempo calcolato durante l'incontro. Se i nodi sono in connessione il coefficiente s varrà n, $s = n$ dove n è un valore di tempo base scelto in configurazione, presumibilmente pari a un secondo. Altrimenti, se non dovessero essere in connessione, sarà pari al tempo trascorso dall'ultimo incontro, $s = time_{lastSeen} - time_{now}$. In questo modo si avranno dei valori raffiguranti la latenza tra un nodo e i suoi vicini. Il coefficiente a, come s, è una costante scelta in fase di configurazione.

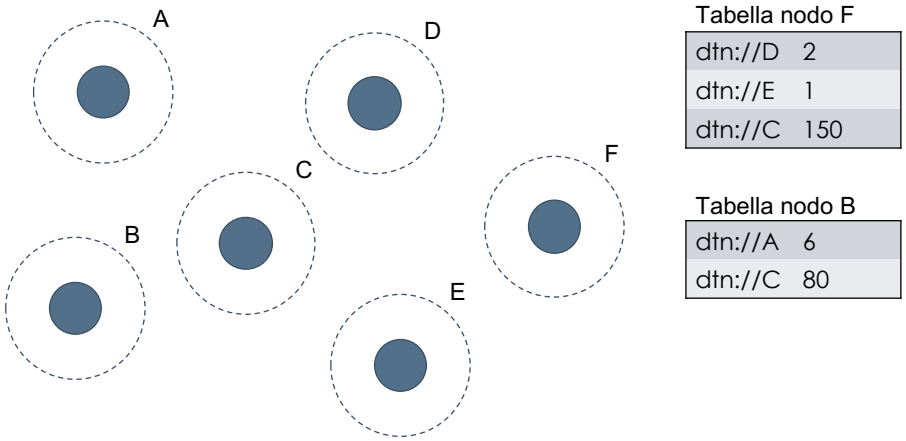


Figura 6.1. Tabella delle latenze

La figura 6.1 rappresenta una rete di sei nodi in un istante di tempo in cui non stanno comunicando, perché al di fuori dei range di comunicazione, ma che

in precedenza si sono scambiati delle informazioni. Possiamo vedere, ad esempio, che il nodo F avrà incontrato con più frequenza i nodi D ed E, avendo dei tempi di latenza più bassi, ma che cresceranno non essendo più connessi. Questi valori tengono in considerazione il reale tempo in cui sono stati connessi e in cui non lo sono stati. Forniscono un attendibile tempo di latenza medio tra un incontro e l'altro.

Tabella dei percorsi

Questa tabella consente di memorizzare tutti i percorsi verso le destinazioni della rete attraverso i vicini, associati ad un valore di costo determinato secondo le regole descritte in 3.2 e ad un valore di latenza. Ogni nodo avrà, quindi, una lista di percorsi, con la relativa latenza, attraverso i quali far passare i pacchetti verso una data destinazione e potrà scegliere quello migliore; sarà possibile avere anche più percorsi per la stessa destinazione.

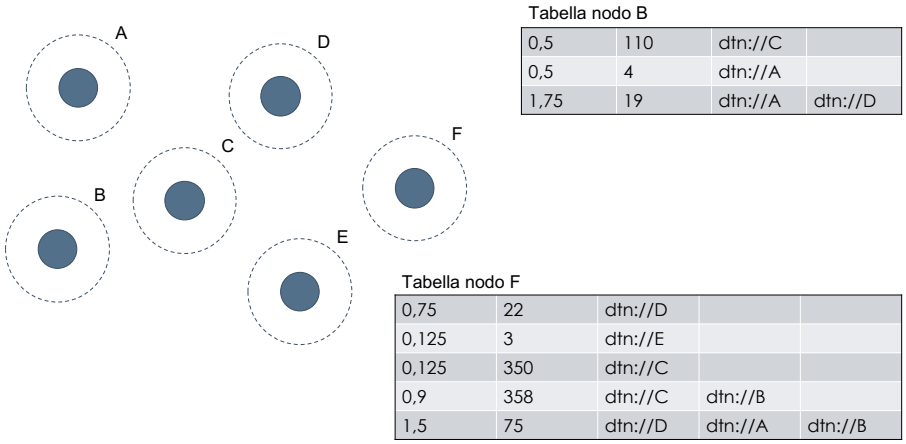


Figura 6.2. Tabella dei percorsi

Nella figura 6.2 possiamo vedere che il nodo F avrà le informazioni riguardanti i percorsi aggiornati dopo l'incontro dei nodi D, E e C. Nella tabella sono presenti quindi i percorsi diretti verso i suoi vicini, con costo pari a $1 - p_i$ dove p_i è la probabilità di incontro di quel nodo e latenza pari alla media calcolata nella *Tabella delle latenze*. Sono presenti, inoltre, i percorsi verso altre destinazioni derivati da una conoscenza dei nodi C e D verso la rete, con costo pari a $C = \sum_{i=0}^{n}(1 - \rho_i)$, dove n sono i nodi del percorso. In questo caso è presente inoltre la latenza del percorso calcolata come la latenza che il nodo vicino aveva verso la destinazione più quella tra il nodo corrente e il vicino, $L = \sum_{i=0}^{n}(\gamma_i)$, dove γ è la latenza tra ogni nodo del percorso e n il numero di nodi. Ogni volta che viene inserito un valore

nella *Tabella delle latenze* la media della latenza verso quel nodo sarà differente, vengono, quindi, ogni volta, aggiornate le latenze verso la destinazione avente come next hop il nodo appena incontrato.

6.1.2 Forwarding

La fase di forwarding rimane esattamente come specificato nel capitolo precedente (5.1.4), con l'aggiunta di una nuova strategia:

- **Latency Based**, l'obbiettivo di tale strategia è quello di poter stimare una latenza tra sorgente e destinazione e poi scegliere quale percorso è meglio in fase di forwarding. L'applicazione che decide di inviare un pacchetto dovrà specificare entro quanto vorrebbe che questo pacchetto arrivi a destinazione. Dato questo parametro, ogni hop controllerà le stime verso le destinazioni e sceglierà un percorso tra quelli risultati idonei. La scelta del percorso è basata sul controllo della tabella dei percorsi calcolata e aggiornata in precedenza. Ogni volta che il pacchetto attraversa un nodo, la latenza massima specificata nel pacchetto viene decrementata della latenza effettiva impiegata per attraversare il link.

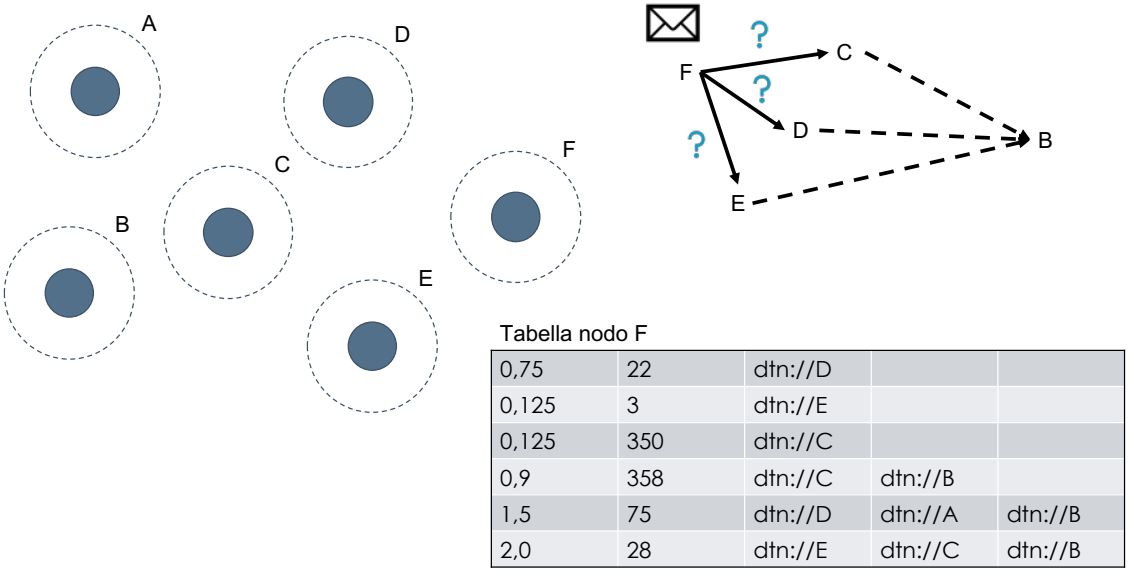


Figura 6.3. Forwarding: quale percorso scegliere?

Come possiamo vedere dalla figura 6.3, se il nodo F dovesse scegliere verso quale next hop mandare il pacchetto con latenza massima pari a 50 e con destinazione B,

l'algoritmo sceglierebbe di inoltrarlo al nodo E invece che al nodo D. Se la latenza massima fosse stata pari a 20, il nodo F non avrebbe inoltrato il bundle a nessuno. Come ultimo caso, se la latenza fosse stata 100, il nodo F avrebbe inoltrato al nodo D o E scelto in modo casuale.

6.1.3 Configurazione

Modificando il file di configurazione sarà possibile specificare la nuova strategia. Sarà necessario, inoltre, specificare il valore degli altri parametri utili al corretto funzionamento della strategia. Di seguito sono riportati i parametri da utilizzare:

- **maxprop_forwarding_strategy** *[MEDLE, Random, Latency]*: permette di scegliere quale strategia far utilizzare all'algoritmo
- **maxprop_A** *[0...1]*: permette di specificare la constante a, tipicamente pari a 0.375
- **maxprop_baseValue** *[0...n]*: permette di specificare il valore di s nel caso in cui i nodi siano connessi, di default sarà nell'intorno di un secondo

6.2 Struttura interna

In questa sezione andremo ad analizzare nel dettaglio la struttura ed i meccanismi interni che sono stati modificati e aggiunti per soddisfare l'inserimento della nuova strategia. Verranno analizzate le modifiche effettuate all'interno dell'architettura interna del modulo di routing di IBR-DTN.

6.2.1 MaxPropRoutingExtension

All'interno di questa classe è stato modificato solamente il metodo che scatta all'evento di **TimeEvent**, nel quale ogni secondo viene verificata la presenza dei vicini e attuato il calcolo necessario per la definizione della latenza. Se il vicino risulta già registrato, si esegue il calcolo della latenza media, altrimenti, viene prima inserita la entry del nuovo vicino nella *Tabella delle latenze*.

```
1 const std::set<Node> neighbor_list = getNeighbors();
2 for(const auto & neighbor : neighbor_list){
3     _database.setLatencyTable(neighbor);
4 }
5 _database.calculateLatency(neighbor_list);
```

Successivamente, vanno aggiornate le latenze all'interno della *Tabella dei percorsi*, secondo i valori appena inseriti all'interno della *Tabella delle latenze*.

```
1 _database.update_latency();
```

6.2.2 DeliveryPredictabilityMapMaxProp

Le modifiche sulle strutture del database sono state effettuate in questa classe. É stata aggiunta la *Tabella delle latenze* e modificata quella dei percorsi.

```
1 std::vector<PathsCostMapParameter> pathsCostMap;
2 std::map<dtn::data::EID, std::vector<size_t>> latencyTable;
```

La classe **PathsCostMapParameter** vede l'aggiunta del nuovo parametro di latenza associato al singolo percorso.

Nella classe *DeliveryPredictabilityMapMaxProp* sono stati, inoltre, inseriti quattro metodi fondamentali:

- **getLatencyTableNew**, questo metodo permette di ottenere il valore di latenza medio attuale dopo l'inserimento dell'ultimo tempo di incontro.

```
1     for(auto it = _latencyTable.begin(); it != _latencyTable.end(); ++it){
2     if(it->first.getEID().sameHost(neighbor)){
3       return it->second._avg.get();
4     }
5   }
```

- **getLatencyTableOld**, questo metodo permette di ottenere il valore di latenza medio all'istante precedente rispetto all'inserimento dell'ultimo tempo di incontro nella *Tabella delle latenze*.

```
1     for(auto it = _latencyTable.begin(); it != _latencyTable.end(); ++it){
2     if(it->first.getEID().sameHost(neighbor)){
3       return it->second._lastavg.get();
4     }
5   }
```

- **update_latency**, permette di utilizzare le due funzioni sopra elencate, per aggiornare la latenza dei percorsi, all'interno dell'omonima tabella, passanti per il next hop appena incontrato

```
1     if(_pathsTable.size() > 0)
2     for(const auto & node : _latencyTable){
3       for(auto itpath = _pathsTable.begin(); itpath != _pathsTable.end(); ++itpath){
4         if(itpath->_path.front().sameHost(node.name.getEID())){
```

```
5             if(itpath->_latency < getLatencyTableOld(node.name.getEID()))
6               itpath->_latency = 0;
7             else
8               itpath->_latency -= getLatencyTableOld(node.name.getEID());
9             itpath->_latency += getLatencyTableNew(node.name.getEID());
10          }
11        }
12    }
```

- **setLatencyTable**, permette di inserire un nuovo nodo nella *Tabella delle latenze*, associato al nodo "vicino" appena incontrato, senza alcuno valore di latenza

- **calculateLatency**, permette di calcolare la latenza nel caso in cui: i nodi siano connessi e in quello in cui non lo siano, con due formule diverse, come spiegato in 6.1.

```
1     for(auto & entry : _latencyTable){
2     if(neighbor_list.find(entry.first) != neighbor_list.end()){
3       entry.second._isPresent = true;
4       entry.second._lastSeen = dtn::utils::Clock::getMonotonicTimestamp();
5       entry.second.calculateAVG(_baseValue);
6     }
7     else{
8       entry.second._isPresent = false;
9       entry.second.calculateAVG(dtn::utils::Clock::getMonotonicTimestamp());
10    }
11  }
```

Nel primo caso verrà calcolata la media con $s = n$, dove n sarà la costante specificata in configurazione *maxprop_baseValue*. Nel secondo caso verrà calcolata con il tempo trascorso dall'ultimo incontro.

```
1     void calculateAVG(Timestamp sampleLatency){
2     if(!_isPresent){
3       Timestamp sampleLatencytmp = sampleLatency - _lastSeen;
4       _avg = (1-_a)*_avg+_a*sampleLatencytmp;
5     }
6   }
7   void calculateAVG(float sampleLatency){
8     if(_isPresent){
9       _avg = (1-_a)*_avg+_a*sampleLatency;
10    }
11  }
```

6.2.3 ForwardingStrategyMaxProp

All'interno di questa classe è stato implementato la strategia vera e propria di forwarding basata sulla latenza. Consiste nel controllare se all'interno del bundle è presente il parametro settato dall'applicazione che ci consente di capire entro quanto massimo vorrebbe che il pacchetto arrivasse a destinazione. Successivamente bisogna controllare se esiste un percorso, nella *Tabella dei percorsi*, che ha una latenza inferiore al parametro specificato nel bundle. Quindi si prendono tutti i next hop possibili verso la destinazione desiderata e vengono scartati tutti quelli che hanno un percorso alle spalle con latenza troppo alta. Tra quelli disponibili ne viene successivamente scelto uno in modo casuale.

```
if(bundle.hasLatency()){
  if(!listNewNodes.empty()){
    for(const std::pair<EID, float> & next_hop : listNewNodes){
      for (auto it = dpm.begin_pathsTable(); it != dpm.end_pathsTable(); it++){
        if(it->_path.front().sameHost(next_hop.name)){
          if(it->_latency < (float)latencyMax){
            listLatencyNodes.push_back(next_hop);
            break;
          }
        }
      }
    }
  }
  else{
    const dtn::data::EID newNode;
    return std::make_pair(newNode, -1.0);
  }
}
```

Può capitare che non venga trovato alcun percorso verso la destinazione che soddisfi il vincolo sulla latenza. In questo caso il bundle in questione verrà mantenuto nello storage fino alla sua scadenza o all'eventuale incontro di un nuovo nodo che fornisca un percorso accettabile. Se, però, il tempo specificato per arrivare alla destinazione risulta essere pari a zero allora il pacchetto viene scartato.

```
if(latencyMax <= 0){
  dtn::core::BundlePurgeEvent::raise(bundle,
      dtn::core::BundlePurgeEvent::NO_ROUTE_KNOWN);
  const dtn::data::EID newNode;
  return std::make_pair(newNode, -1.0);
}
```

Nel caso in cui non è stato specificato nessun parametro di tempo all'interno del bundle dall'applicazione, si procede al forwarding con un metodo Random come spiegato nel capitolo precedente (5.1.4).

6.2.4 LatencyBlock e TimestampLatencyBlock

Per consentire all'applicazione di inserire il parametro del tempo di latenza massimo all'interno del bundle, è stato necessario creare un modo per inserire due nuovi blocchi (2.2.5) al suo interno che garantissero questa operazione. Il *Latency Block* garantisce quindi all'applicazione di inserire la latenza massima entro cui si desidera ricevere il pacchetto. Il *Timestamp Latency Block* permette di inserire il tempo in cui è stato generato il pacchetto e garantire all'applicazione di ridurre la latenza ad ogni hop sostituendo il tempo inserito nell'hop precedente con quello corrente. Il metodo più significativo in *Latency Block* è **calculateLatency** che svolge le operazioni di aggiornamento della latenza massima dato il timestamp dall'omonimo blocco e memorizza il tempo attuale per calcolar la permanenza del pacchetto nello storage.

```
if(time − now > _latency)
   _latency = 0;
else
   _latency −= time − now;
_time = now;
```

La chiamata a questa funzione avviene nel **Bundle Core** nel momento in cui arriva il bundle e deve essere memorizzato all'interno dello storage in attesa del forwarding. Di seguito viene presentato il codice che esegue questo processo:

```
try {
   TimestampLatencyBlock &timestampblock = bundle.find<TimestampLatencyBlock>();
   time = timestampblock.getTimestamp();
   now = Time.now();
   timestampblock.setTimestamp(now);
   try {
      LatencyBlock &latency = bundle.find<LatencyBlock>();
      latency.calculateLatency(time, now);
   } catch (const NoSuchBlockFoundException&) { };
} catch (const NoSuchBlockFoundException&) { };
```

Viene, quindi, preso il tempo dal blocco Timestamp, ne viene fatta la differenza con il timestamp attuale e il valore ottenuto viene sottratto alla latenza estratta dal blocco Latency. A questo punto si aggiorna il timestamp nell'omonimo blocco, con il valore *now*, e si può procedere con la fase di forwarding.

Capitolo 7

Risultati Ottenuti

In questo capitolo vengono presentati i test effettuati e commentati i relativi risultati. A tal proposito è stato ricreato uno scenario simile alle simulazioni fatte nella sezione 3.4. Come specificato in precedenza l'applicazione Aether è pensata per essere sfruttata in cantieri lavorativi, e per questo motivo sono stati fatti i test simulando aree di certe dimensioni, organizzate ad hoc. Per le simulazioni è stato utilizzato *LEPTON* [19].

7.1 LEPTON simulator

LEPTON è una piattaforma di emulazione progettata principalmente per consentire agli sviluppatori di software di rete opportunistiche (ovvero middleware e / o applicazioni) di eseguire i propri sistemi software con mobilità simulata. A differenza di altre piattaforme di emulazione, LEPTON non richiede apparecchiature di rete particolari e non richiede nemmeno la distribuzione di macchine virtuali su uno o più host. Una semplice postazione laptop o desktop che esegue Linux può supportare facilmente esperimenti basati sull'emulazione che coinvolgono fino a duecento nodi, e su qualsiasi cluster di macchine Linux è possibile eseguire esperimenti più grandi. Essendo un emulatore anziché un simulatore, LEPTON ha lo scopo di guidare la comunicazione tra istanze con funzionalità complete di un sistema di rete opportunistico (un sistema OppNet a.k.a.), ciascuna istanza determina il comportamento di un nodo di sistema (SN) durante la simulazione. É possibile costruire uno scenario ad hoc per il proprio caso specifico e nel nostro caso è stato possibile installare su ogni nodo il nostro framework IBR-DTN e le relative componenti di supporto per eseguire i test di invio bundle.

7.2 Scenario

Come per le simulazione della sezione 3.4, si è scelta un'area di lavoro di 50×100 m^2, che simulasse un cantiere. I nodi, al suo interno, rappresentano macchinari di lavoro e/o persone che si spostano sull'intera area. In questo caso, a contrario di quelli precedenti, c'è stato un riadattamento delle aree di lavoro, per cercare di rendere il più possibile realistica la simulazione.

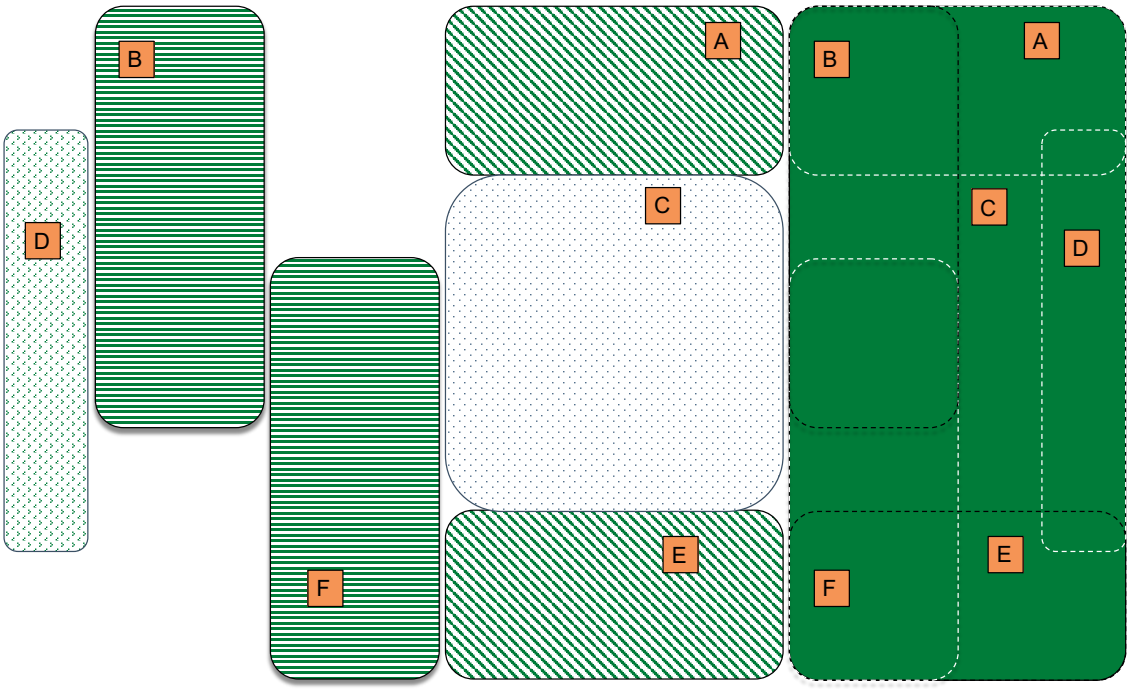


Figura 7.1. Scenario

Come si può vedere dalla figura 7.1, l'area verde, che rappresenta l'intero campo di lavoro, è stata suddivisa in 6 aree più piccole. Le aree più piccole sono state pensate per rappresentare una porzione di terreno su cui i dispositivi potessero muoversi e raffigurare quelli presenti in uno scenario reale. I nodi in ogni area possono muoversi secondo il pattern *RandomWayPoint*, che consente un movimento casuale, e arrivare nelle parti sovrapposte per comunicare con quelli appartenenti ad aree diverse. Anche in questo caso, come nelle simulazioni della sezione 3.4, sono presenti:

- *Nodi Statici*, quattro nodi posti agli angoli dell'area (che chiameremo nodi NERI)
- *Nodi mobili*, confinati all'interno delle aree più piccole (che chiameremo nodi BLU), composti dal 70% dei nodi mobili totali

- *Nodi mobili*, sull'intera area (che chiameremo nodi ROSSI), composti dal 30% dei nodi mobili totali

La quantità di nodi all'interno di ogni singola area, è stata calcolata in base alla loro dimensione. Il 70% dei nodi mobili (BLU) è stato suddiviso uniformemente sull'intera superficie di lavoro.

Ogni gruppo, nodi NERI, BLU e ROSSI, avrà una suddivisione interna, in base alla velocità di movimento e alla categoria che il nodo rappresenta. Nella figura

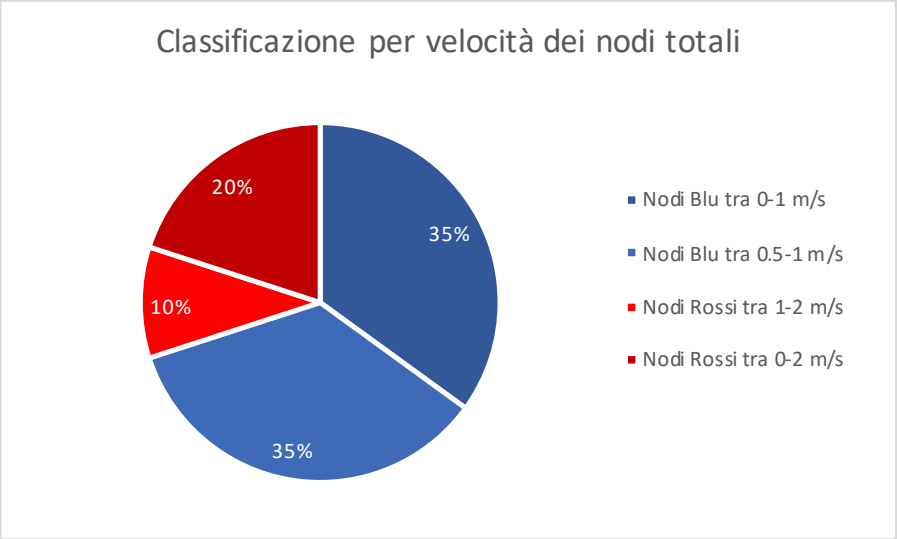


Figura 7.2. Suddivisione del numero di nodi per velocità di movimento

7.2, si può vedere la suddivisione dei nodi mobili, spiegata nei paragrafi successivi, attraverso il parametro di velocità.

Nodi NERI Questo gruppo di nodi vuole raffigurare quelli presenti sui 4 angoli dell'area, posti a 5 metri dal perimetro. Nel nostro scenario rappresentano dei nodi base ai quali sono assegnati gruppi di nodi di altre categorie, diventando dei nodi riferimento. Il nodo situato a nord ovest dell'area sarà il nodo base per l'area B e per il 25% dei nodi ROSSI, presi in modo casuale. Il nodo a nord est dell'area sarà il nodo base per l'area A, l'area D e un altro 25% dei nodi ROSSI, presi sempre in modo casuale. Per quanto riguarda il nodo a sud ovest, sarà il nodo base per l'area F e un 25% dei nodi ROSSI, presi in modo casuale. L'ultimo nodo situato a sud est dell'area sarà la base per l'area E, l'area C e il restante 25% dei nodi ROSSI. Questi nodi base si occuperanno di ricevere i messaggi inviati dai nodi del proprio gruppo ed, inoltre, invieranno, ogni 20 minuti un messaggio senza scadenza ad ognuno di essi.

Nodi BLU I nodi blu sono quelli confinati all'interno di una singola area, e corrispondono al 70% dei nodi totali. Metà di essi si muove ad una velocità che varia tra 0.5-1 m/s, mentre la restante metà tra 0-1 m/s. Inoltre, il 40% di questi nodi si comporta da sensori, un ulteriore 40% da attuatori, mentre il restante 20% svolge la sola funzione di trasporto (Figura 7.3).

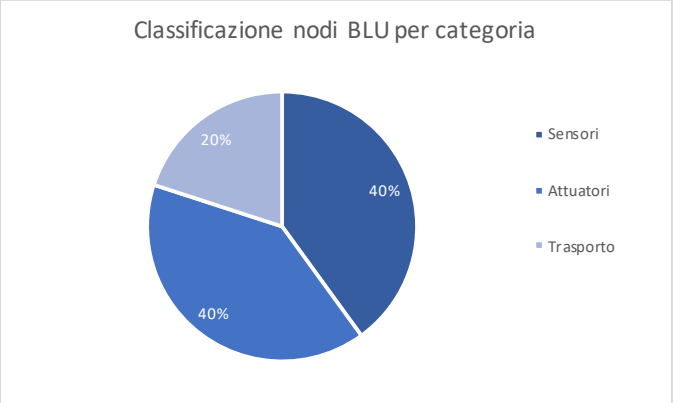


Figura 7.3. Suddivisione dei nodi BLU per categoria

I sensori, si occuperanno di ricevere dati e inviare:

- Ogni 60 secondi:
  - un messaggio, senza scadenza, al nodo base assegnato
  - un messaggio, con scadenza, ad un nodo BLU (sensore o attuatore) assegnato all'inizio della simulazione, facente parte di qualsiasi area
  - un messaggio, con scadenza, ad un nodo BLU (attuatore) assegnato all'inizio della simulazione, facente parte della stessa area con probabilità di inoltro del 50% (ovvero esiste il 50% di possibilità che il pacchetto parta oppure no)
- Ogni 5 minuti:
  - un messaggio, con scadenza, ad un nodo BLU (attuatore) facente parte della stessa area
  - un messaggio, con scadenza, ad un nodo ROSSO (attuatore) casuale
  - un messaggio, con scadenza, a due nodi BLU (attuatori) casuali, facente parte di un area non adiacente
- Ad ogni ricezione:

– un messaggio, con scadenza, di risposta alla ricezione di un pacchetto generato da un altro nodo

Gli attuatori si occuperanno di ricevere dati e inviare ogni 5 minuti:

- un messaggio, con scadenza, ad un nodo BLU (sensore) scelto in modo casuale
- un messaggio, con scadenza, ad un nodo BLU (sensore) scelto in modo casuale, facente parte della stessa area o adiacente
- un messaggio, con scadenza, ad un nodo BLU (attuatore) casuale facente parte di un area non adiacente

I messaggi inviati ogni 5 minuti avranno un ritardo di invio di un numero di secondi casuale preso tra 0 e 5 minuti, allo scopo di evitare sincronizzazioni.

Nodi ROSSI I nodi blu sono quelli confinati all'interno di una singola area, e corrispondono al 70% dei nodi totali. Un terzo di essi si muove ad una velocità che varia tra 0-2 m/s, mentre i restanti due terzi tra 1-2 m/s. Inoltre, il 100% dei nodi che si sposta ad una velocità tra 1-2 m/s si comporta da sensori, il 50% dei nodi che si sposta ad una velocità tra 0-2 m/s da attuatori e il restante 50% svolge la sola funzione di trasporto (Figura 7.4).

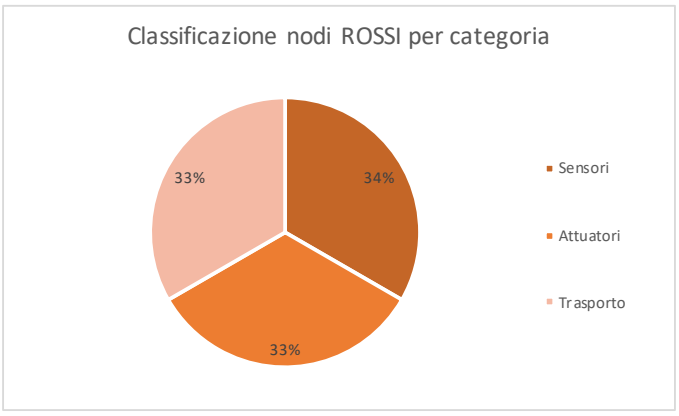


Figura 7.4. Suddivisione dei nodi ROSSI per categoria

I sensori, si occuperanno di ricevere dati e inviare ogni 60 secondi al loro nodo base, un messaggio senza scadenza. Gli attuatori saranno solamente in fase di ascolto.

7.3 Test

I test sono stati eseguiti sullo scenario sopra descritto, per quanto riguarda gli algoritmi di routing: **Epidemic, MaxProp MEDLE e MaxProp Latency Based**. Sono state eseguite due tipi di simulazioni: una con 20 nodi mobili, ed una con 40; per un totale di 24 e 44 nodi rispettivamente, contando anche i nodi statici posti nei quattro angoli. In entrambi i casi si vogliono evidenziare le caratteristiche degli algoritmi in termini di:

- *Percentuali di consegna.* quanti pacchetti sono effettivamente arrivati a destinazione, $\frac{N°pacchetti\ ricevuti}{N°pacchetti\ inviati} * 100$
- *Tempi di consegna*, tempo trascorso dal momento dell'invio a quello di avvenuta ricezione, $t_{finale} - t_{iniziale}$
- *Utilizzo della memoria.* quanti bundle sono stati memorizzati all'interno del nodo durate la simulazione
- *Overhead.* quante sono le repliche che circolano all'interno della rete rispetto agli effettivi pacchetti utili, $\frac{N°repliche\ in\ rete}{N°pacchetti\ utili}$

7.3.1 Test: Simulazione 24 nodi

Questa simulazione prevede una rete composta da 24 nodi, 4 fermi e 20 mobili. Dei 20 nodi mobili: 14 sono BLU e 6 ROSSI. Nella figura 7.5, si può vedere un esempio di configurazione della rete durante la simulazione, con le suddivisioni per aree secondo le regole di 7.2. Si può cogliere che i nodi in comunicazione sono collegati da una linea rossa molto marcata, mentre i nodi vicini ma che non stanno trasferendo alcun dato, sono collegati da una linea arancione più sottile. Vedremo come gli algoritmi *Epidemic, MaxProp Most Encountered/Drop Least Encountered e MaxProp Latency Based* si comportano attraverso le metriche descritte in 7.3. Per quanto riguarda il protocollo *MaxProp* si è deciso di impostare: un tempo di handshake periodico pari a circa 8 minuti, il tempo di store periodico pari a 10 minuti ed è stata disattivata la funzione di *Handshake Notification.*

Simulazione preliminare

In un primo momento è stata eseguita una simulazione preliminare per capire il comportamento della rete e determinare i vari parametri di configurazione che testassero al meglio i diversi protocolli nel nostro scenario; questa simulazione è stata eseguita sull'algoritmo di *MaxProp MEDLE.*

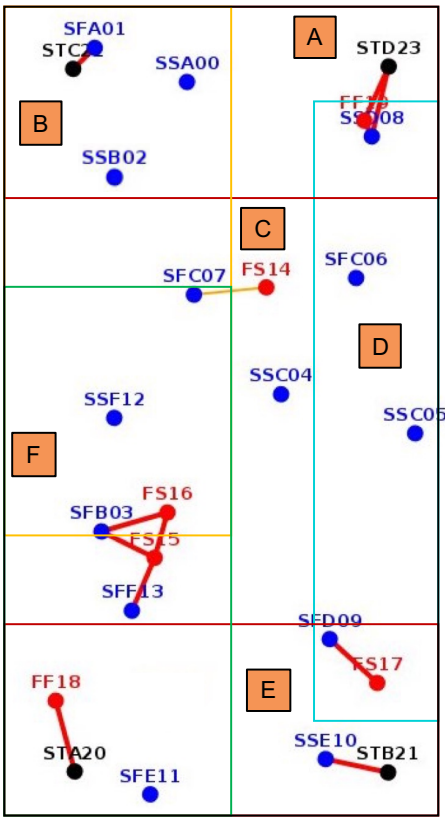


Figura 7.5. Esempio rete da 24 nodi, durante la simulazione

Dalla figura 7.6, possiamo notare la distribuzione dei tempi di consegna dei bundle senza scadenze, ne vincoli sul tempo di consegna; considerando che *MaxProp Latency Based* è l'unico a sfruttare effettivamente questo parametro. Il grafico è servito, per l'appunto, a determinare quali fossero i vincoli di tempo da impostare ai pacchetti delle diverse aree, che facessero notare la differenza tra bundle arrivati in orario e non, in modo da mettere a confronto la strategia *Latency Based* sviluppata con le altre. Il tempo di vita del pacchetto è stato impostato pari ad un ora, dopo aver constatato che il 95% di essi arriva a destinazione entro 35/40 minuti. Da questo grafico, successivamente, sono stati calcolati i vincoli sui tempi da inserire nei bundle, secondo queste regole:

- Per i messaggi destinati a nodi della stessa area: 75% di probabilità di avere una scadenza in [2, 4] minuti (random uniforme), altrimenti scadenza infinita
- Per i messaggi destinati a nodi di aree vicine: 75% di probabilità di avere una scadenza in [3, 8] minuti (random uniforme), altrimenti scadenza infinita

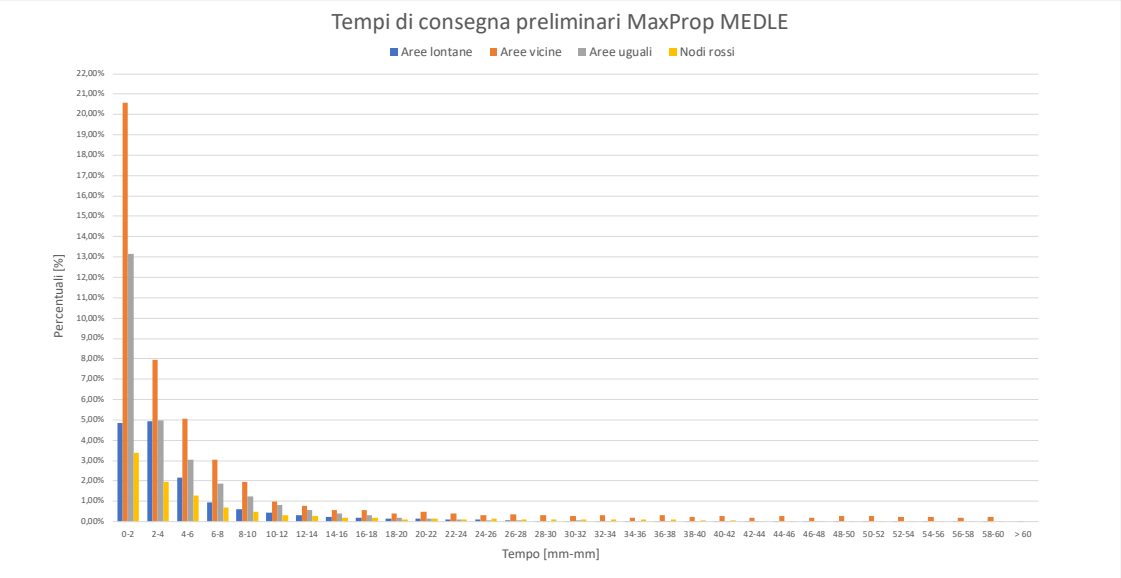


Figura 7.6. Tempi di consegna preliminari

- Per i messaggi destinati a nodi di aree lontane: 75% di probabilità di avere una scadenza in [4, 10] minuti (random uniforme), altrimenti scadenza infinita
- Per i messaggi destinati a nodi ai rossi (no area): 100% di probabilità di avere una scadenza in [4, 8] minuti

Simulazione finale: Tempi di consegna

Il tempo di consegna consiste nel misurare quanto tempo è trascorso dall'invio del pacchetto, da parte della sorgente, fino all'avvenuta ricezione alla destinazione, per tutti questi grafici le percentuali sono riferite al totale dei pacchetti inviati.

Nel grafico riportato in 7.7, possiamo notare la distribuzione dei pacchetti nel tempo riferita ai tre algoritmi proposti. Il protocollo *Epidemic* risulta essere il più veloce, quello che consegna i pacchetti alla destinazione nel minor tempo. Nei grafici successivi si può notare, invece, la distribuzione dei pacchetti a seconda della posizione della destinazione rispetto alla relativa sorgente. Saranno quindi divisi i pacchetti tra sorgente e destinazione della stessa area, di aree vicine e di aree lontane. Un ulteriore divisione rappresenterà le comunicazioni con e verso i nodi mobili sull'intera area.

Come si può vedere dalla figura 7.8, i tempi di consegna del protocollo *Epidemic*, sono molto veloci, in media il 95% dei pacchetti di ogni sotto-area, arriva a destinazione in 3-4 minuti, il restante la raggiunge entro massimo 10 minuti.

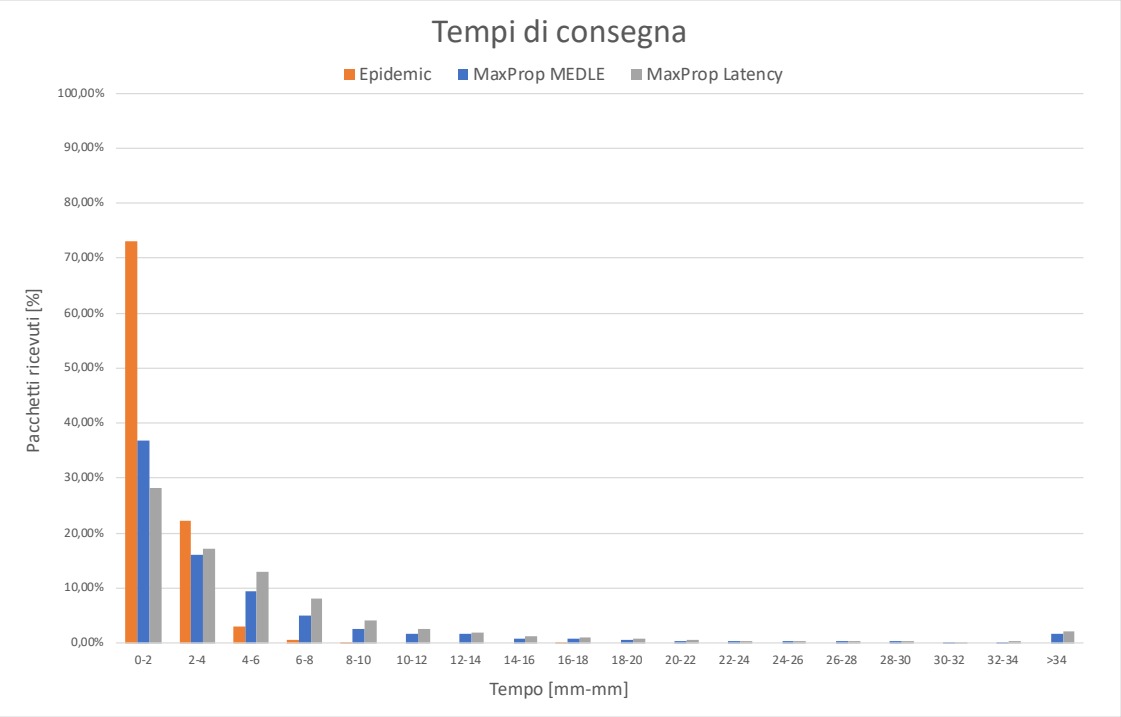


Figura 7.7. Tempi di consegna

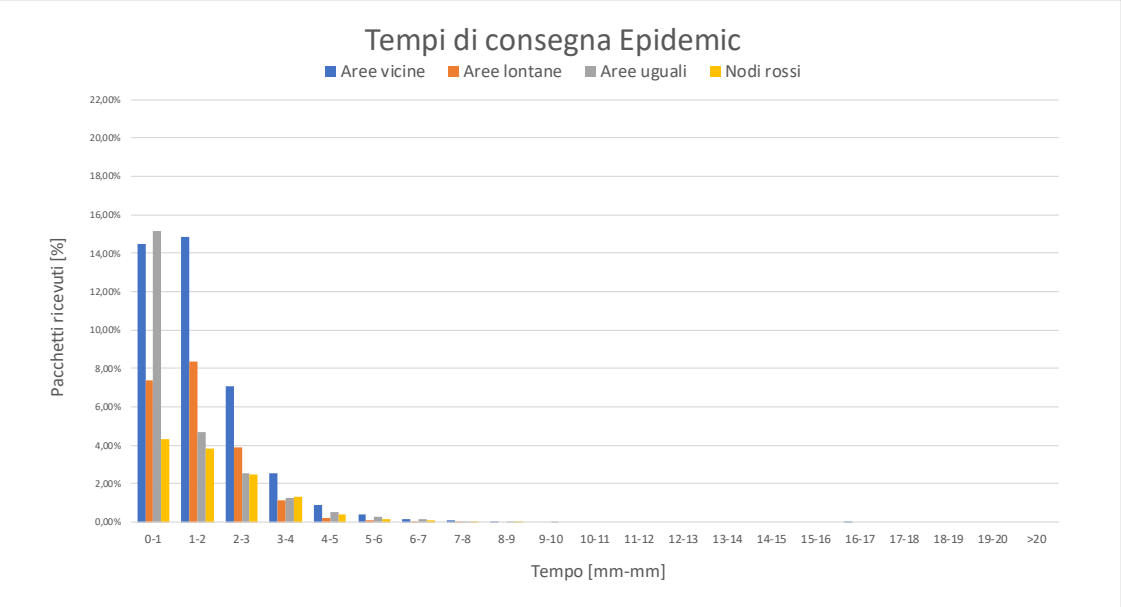


Figura 7.8. Tempi di consegna Epidemic

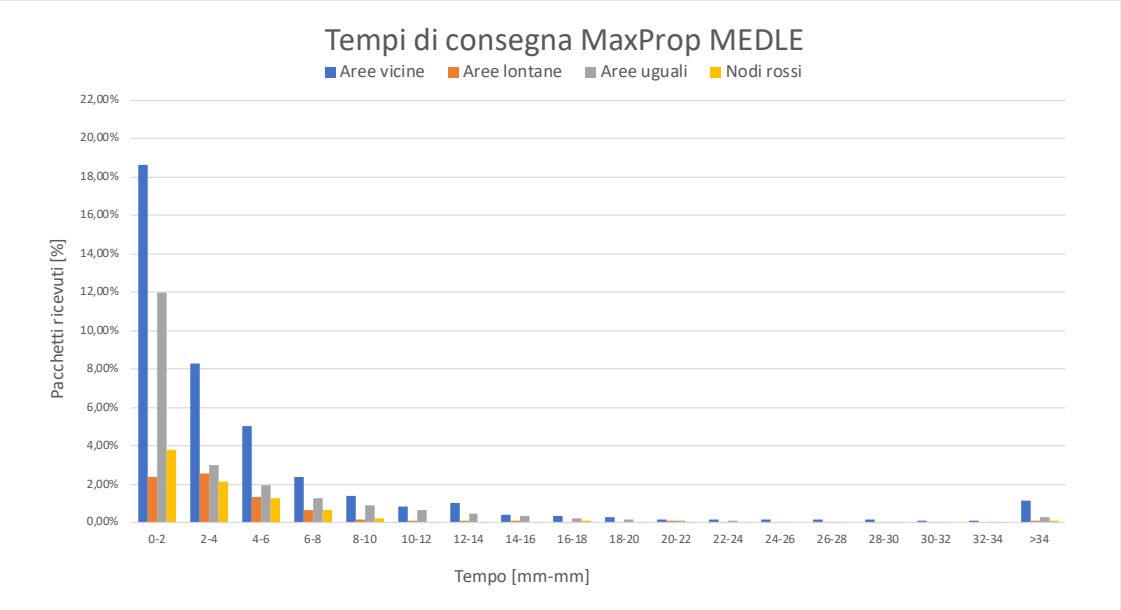


Figura 7.9. Tempi di consegna MaxProp MEDLE

Nella figura 7.9, invece, notiamo i tempi di consegna del protocollo *MaxProp* con strategia *MEDLE*. In questo caso il 95% dei pacchetti in media nelle quattro sotto-aree arriva a destinazione in 14-16 minuti. Risulta essere leggermente più lento rispetto al precedente sia per le caratteristiche di scelta del percorso che lo distinguono, ma anche per via delle numerose operazioni prima dell'inoltro che deve eseguire: come la scelta del percorso migliore per ogni bundle, l'ordinamento del buffer e la replica. La più onerosa computazione per il mantenimento delle conoscenze della rete non è un parametro da sottovalutare nei termini del tempo di consegna.

I risultati del protocollo *MaxProp Latency* riportati in figura 7.10, hanno il parametro $a = 0{,}375$. Possiamo notare che in questo caso arrivano a destinazione meno pacchetti rispetto ai precedenti, dovuto al fatto che i vincoli sui tempi impediscano ai bundle di arrivare a destinazione. Per tutti e tre gli algoritmi, quasi la totalità dei bundle viene ricevuta entro 25 minuti.

Simulazione finale: Percentuali di consegna

La percentuale di consegna, consiste nello stabilire quanti dei pacchetti inviati, sono effettivamente arrivati a destinazione. Ogni pacchetto ha al suo interno una latenza determinata secondo i calcoli riportati in 7.3.1, questo permette di sapere quanti dei pacchetti ricevuti sono arrivati prima del tempo stabilito. Questa caratteristica

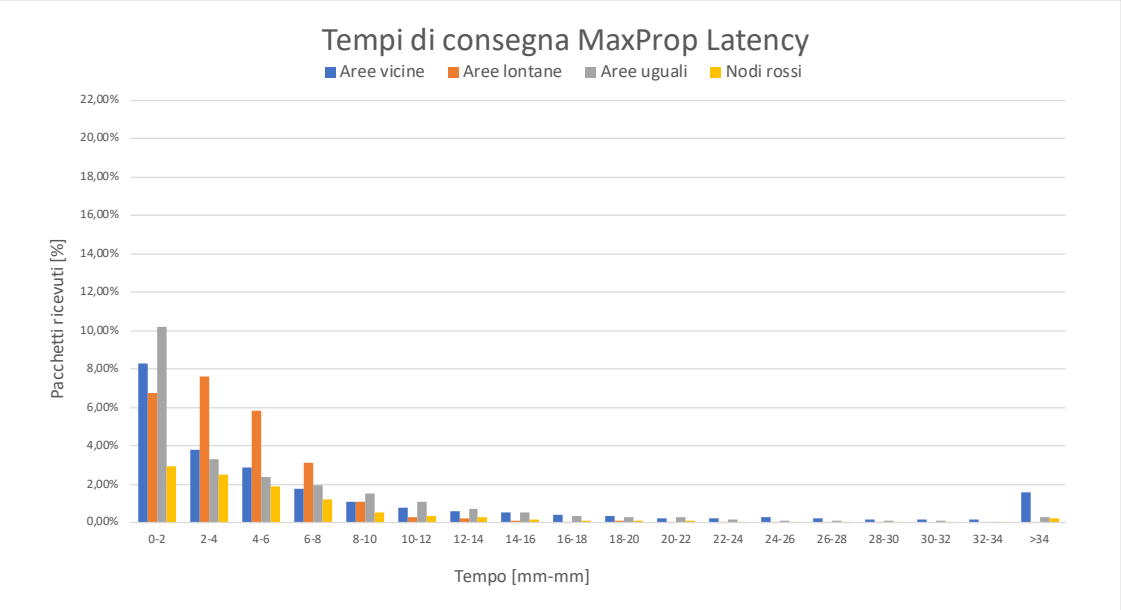


Figura 7.10. Tempi di consegna MaxProp Latency

viene utilizzata per simulare una rete in cui i vari nodi possono decidere entro quanto tempo vorrebbero che il pacchetto arrivasse a destinazione. Nel caso di *Epidemic e MaxProp MEDLE* vedremo la distinzione tra pacchetti arrivati in orario e fuori orario, mentre per quanto riguarda *MaxProp Latency Based* non sarà possibile notare questa differenza perché quelli in ritardo vengono scartati.

Nella figura 7.11, possiamo vedere il confronto delle percentuali di consegna tra i tre algoritmi. *Epidemic e MaxProp MEDLE* hanno una percentuale di consegna generale molto alta, mentre per *MaxProp Latency Based* è più bassa. Questa differenza dimostra come i bundle arrivati in ritardo nella versione *MEDLE* non arrivino affatto quando si usa *Latency Based*. Il ritardo dei pacchetti nei protocolli di *MaxProp*, come spiegato in precedenza, è dovuto alle differenti scelte sui percorsi e al costo di computazione interno dell'algoritmo che deve eseguire svariate operazioni prima di replicare un singolo bundle. Possiamo notare pero come la percentuale di consegna dei bundle arrivati in orario in *Latency Based* sia superiore a quella di *MEDLE*, questo perché molti bundle vengono scartati e di conseguenza ogni nodo ha meno bundle su cui eseguire le varie operazioni di inoltro ed inoltre i percorsi sono meno congestionati. In *Latency Based* si nota che esiste una piccola percentuale di pacchetti arrivati in ritardo questo perché IBR-DTN, nel caso in cui si accorge che la destinazione di un bundle è un nodo vicino, prende in carico l'inoltro senza interrogare nessun tipo di algoritmo di routing. Per questo motivo è possibile che una piccola percentuale di pacchetti scaduti non sia stata scartata in tempo prima dell'incontro.

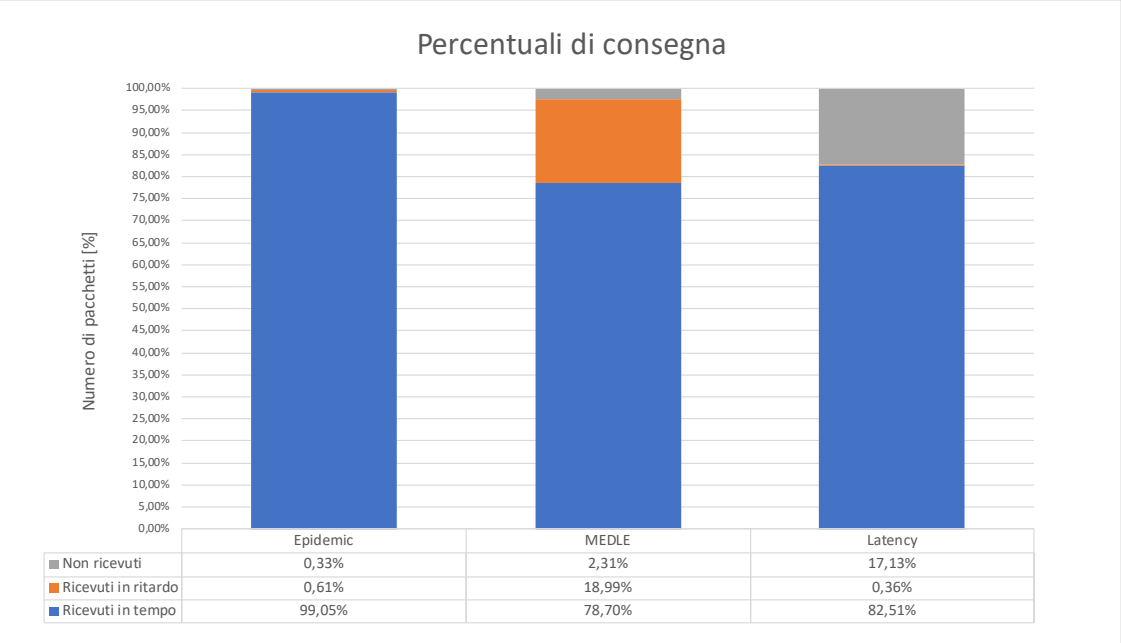


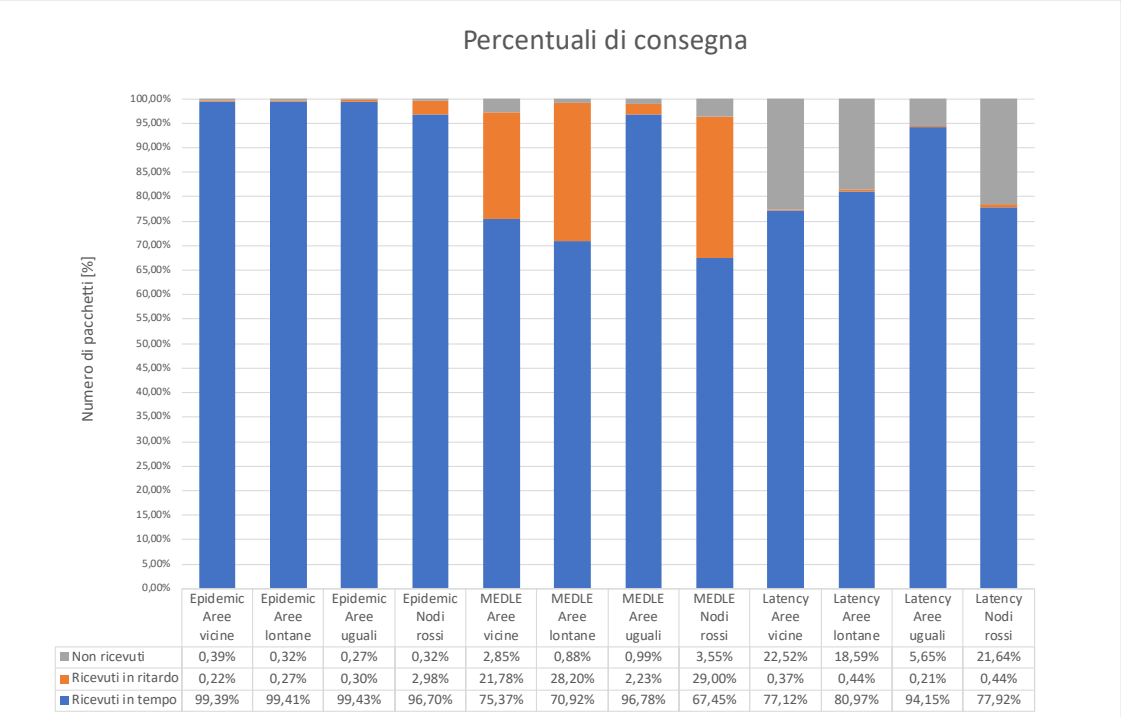
Figura 7.11. Percentuali di consegna

Nella figura 7.12, vediamo le percentuali di consegna divise per aree. Le aree più soggette a ritardi sono: quelle vicine, lontane e riferite ai nodi rossi. Valgono anche in questo caso le stesse considerazioni fatte per il grafico precedente (7.11). Una riflessione ulteriore va fatta sui nodi rossi, che hanno delle percentuali inferiori rispetto alle altre aree. Questo potrebbe essere causato dal fatto che muovendosi su tutta l'area continuamente possono trovarsi nella situazione di essere per poco tempo in contatto con i nodi incontrati e quindi non finire lo scambio dei bundle presenti nei buffer. Potrebbe capitare anche che le code siano piene dato che l'algoritmo li individui come nodi preferenziali su cui inviare bundle per raggiungere la destinazione con costo minimo. Inoltre spostandosi su tutta l'area potrebbero incontrare meno frequentemente i nodi stazionari e posizionati i zone limitrofe dell'area.

Simulazione finale: Overhead

L'overhead permette di stabilire quanti dei pacchetti presenti nella rete sono effettivamente utili rispetto alle repliche generate, che sprecano le risorse della DTN. In questo caso, nello stesso grafico, potremo notare come ogni algoritmo occupa la rete con le proprie repliche.

Nella figura 7.13, notiamo le differenze sulle quantità di repliche generate da ogni algoritmo ad intervalli di 20 minuti. *Epidmeic* mostra i suoi veri punti deboli;



	Epidemic Aree vicine	Epidemic Aree lontane	Epidemic Aree uguali	Epidemic Nodi rossi	MEDLE Aree vicine	MEDLE Aree lontane	MEDLE Aree uguali	MEDLE Nodi rossi	Latency Aree vicine	Latency Aree lontane	Latency Aree uguali	Latency Nodi rossi
Non ricevuti	0,39%	0,32%	0,27%	0,32%	2,85%	0,88%	0,99%	3,55%	22,52%	18,59%	5,65%	21,64%
Ricevuti in ritardo	0,22%	0,27%	0,30%	2,98%	21,78%	28,20%	2,23%	29,00%	0,37%	0,44%	0,21%	0,44%
Ricevuti in tempo	99,39%	99,41%	99,43%	96,70%	75,37%	70,92%	96,78%	67,45%	77,12%	80,97%	94,15%	77,92%

Figura 7.12. Percentuali di consegna divise per aree

con una media di 57000 repliche, satura le risorse della rete. *MaxProp MEDLE* ha una media di repliche intorno a 1600, mentre *MaxProp Latency* 2600, questa differenza nasce dal fatto che in *Latency* i nodi iniziali continueranno a replicare bundle, non avendo una latenza minore o pari a zero, di cui non si riceverà mai un messaggio di ACK e che non potranno essere cancellati dagli storage.

Simulazione finale: Utilizzo della memoria

L'utilizzo della memoria è un parametro che va di pari passo con l'overhead, poiché più alta sarà la quantità di pacchetti in rete, più alto saranno i pacchetti da memorizzare. Questo parametro, quindi, ci dice quanto spreco di risorse di storage ci saranno per ogni algoritmo. Anche in questo caso vediamo un unico grafico che esprime come ogni protocollo influisce sulla memorizzazione dei dati ad intervalli di 20 minuti.

Come per quanto riguarda l'overhead, anche nell'utilizzo della memoria, *Epidemic* dimostra di occupare numerose risorse all'interno della rete con una media di bundle in memoria pari a circa 790 per dispositivo. *MEDLE e Latency*, invece, risultano essere pressoché identici con una media di pacchetti di circa 20, considerando che eliminano i bundle dai buffer una volta ricevuti gli ACK e non solo

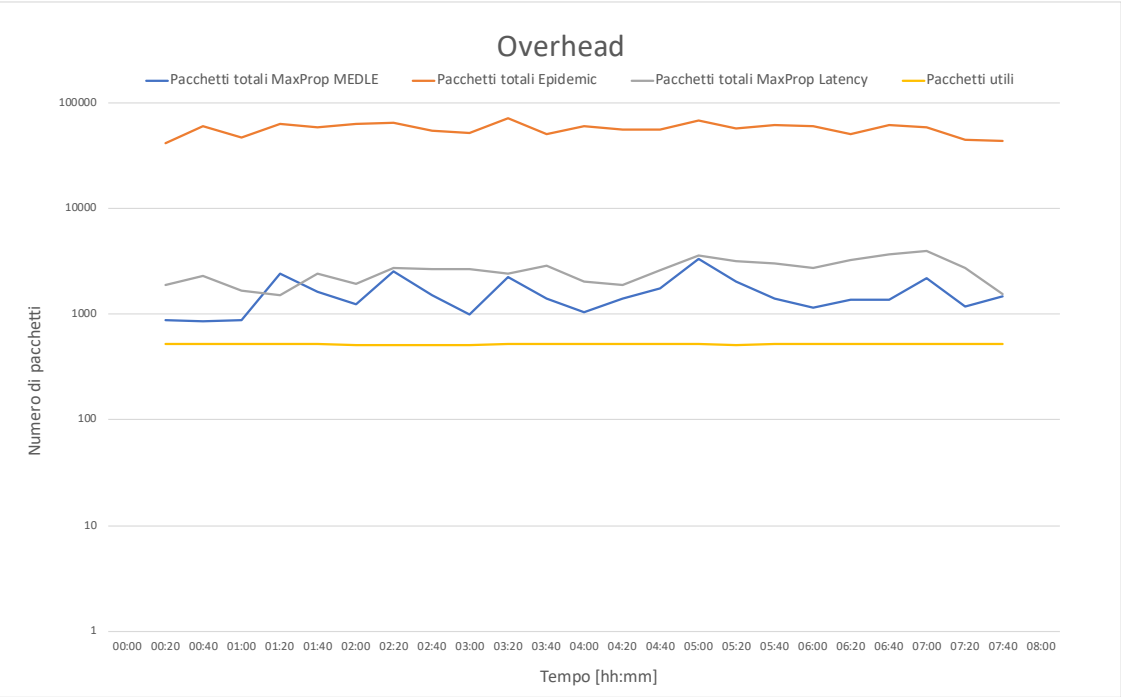


Figura 7.13. Overhead

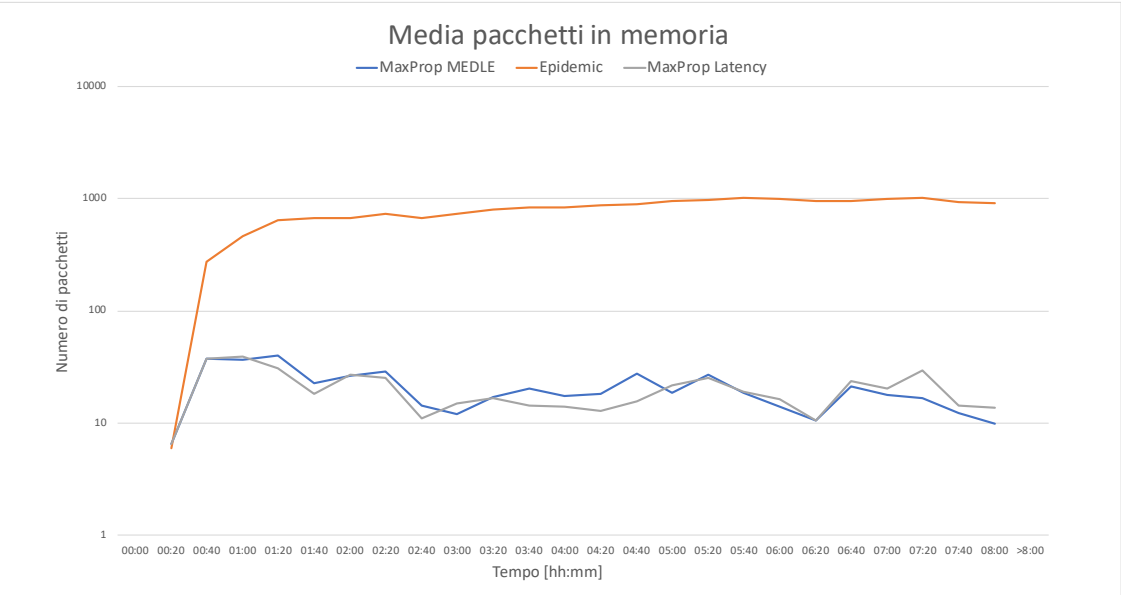


Figura 7.14. Storage

dopo la loro scadenza, come invece fa *Epidemic*. Ci si aspetterebbe che *MEDLE e Latency*, seppur di poco, abbiano un numero di pacchetti differente in memoria. Per dimostrare, però, la bontà del risultato possiamo dire che il numero di pacchetti scartati dal primo dopo la ricezione delle risposte di ACK, sia pari a quello dei bundle eliminati per scadenza dal secondo più l'arrivo degli ACK.

7.3.2 Test: Simulazione 44 nodi

Questa simulazione prevede una rete composta da 44 nodi, 4 fermi e 40 mobili. Dei 40 nodi mobili: 28 sono BLU e 12 ROSSI, divisi secondo le percentuali riportate nella sezione 7.2. Vedremo, anche in questo caso, come gli algoritmi *Epidemic, MaxProp Most Encountered/Drop Least Encountered e MaxProp Latency Based* si comportano attraverso i parametri descritti in 7.3. Valgono le stesse considerazioni fatte in 7.3.1. Si precisa che i risultati relativi ad *Epidemic* in questo caso risultano compromessi, a causa del fatto che questo algoritmo, eseguito su 44 nodi emulati, ha saturato le risorse della macchina host.

Simulazione finale: Tempi di consegna

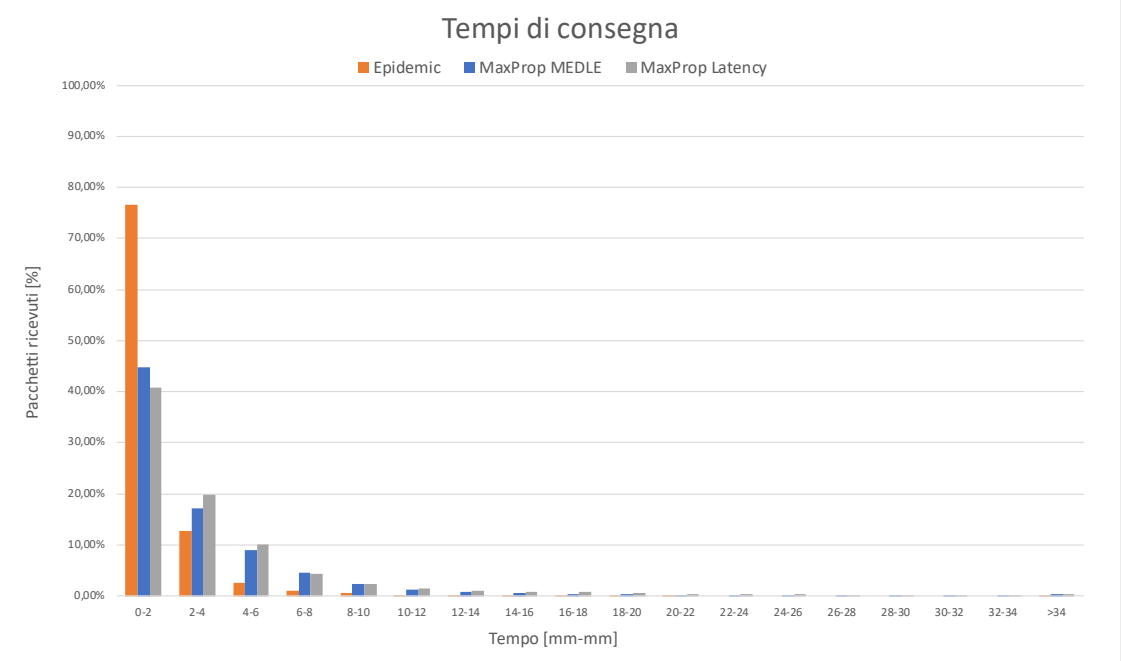
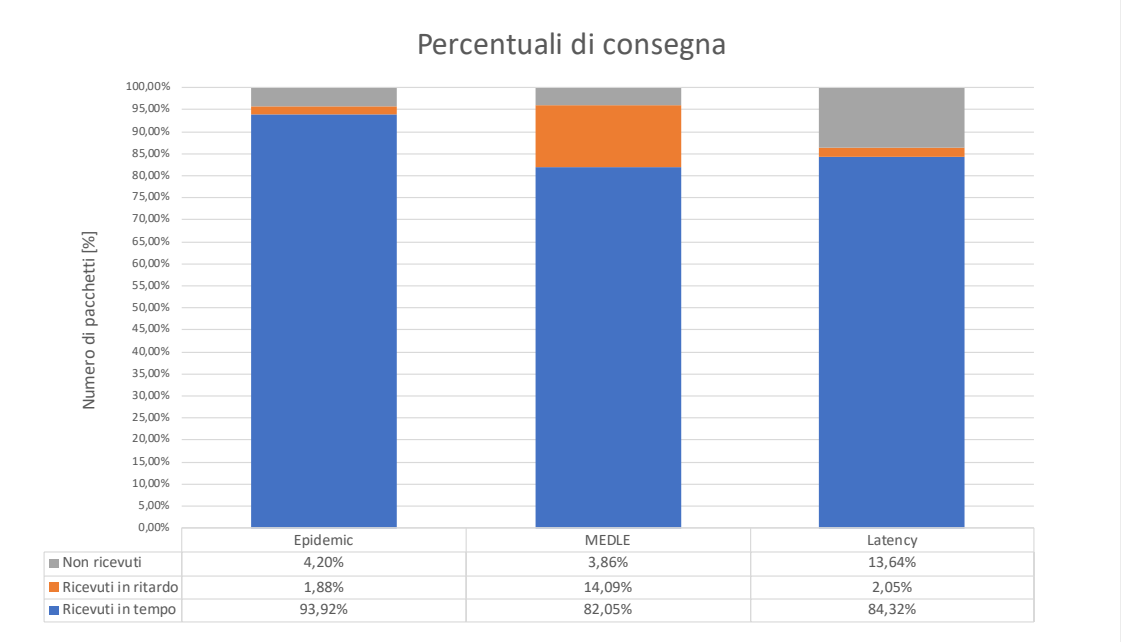


Figura 7.15. Tempi di consegna

Nel grafico riportato in 7.15, possiamo notare la distribuzione dei pacchetti nel tempo riferita ai tre algoritmi proposti. Il protocollo *Epidemic* risulta essere, anche in questo caso, il più veloce, quello che consegna i pacchetti alla destinazione nel minor tempo. Anche con questo scenario possiamo notare che le conclusioni non si discostano molto da quello detto in 7.3.1.

Simulazione finale: Percentuali di consegna



	Epidemic	MEDLE	Latency
Non ricevuti	4,20%	3,86%	13,64%
Ricevuti in ritardo	1,88%	14,09%	2,05%
Ricevuti in tempo	93,92%	82,05%	84,32%

Figura 7.16. Percentuali di consegna

Nella figura 7.16, possiamo notare che le percentuali di consegna di *MaxProp MEDLE e latency* siano molto simili rispetto al caso precedente, 7.3.1, ma il protocollo *Epidemic* ha una percentuale di pacchetti persi maggiore, data l'elevata necessità di risorse per sostenere il carico algoritmico, come premesso in precedenza. Questa informazione potrebbe suggerirci che il protocollo *Epidmeic* non sia adatto a certi tipi di scenari, a contrario degli altri due.

Simulazione finale: Overhead e Utilizzo della memoria

Overhead e utilizzo della memoria vedono variazioni significative soprattutto nell'aumento del numero di pacchetti in *Epidemic*; si può apprezzare, nelle figure 7.17 e 7.18, come la variazione tra il numero di bundle memorizzati e replicati sia ancora

più marcata, di un ordine di grandezza rispetto al solo raddoppio dei pacchetti utili. Viene confermato quanto detto nei paragrafi 7.3.1 e 7.3.1, per quanto riguarda le motivazioni sulle differenze, sottolineando come in questo caso tra *MEDLE* e *Latency* lo scarto sia leggermente più elevato.

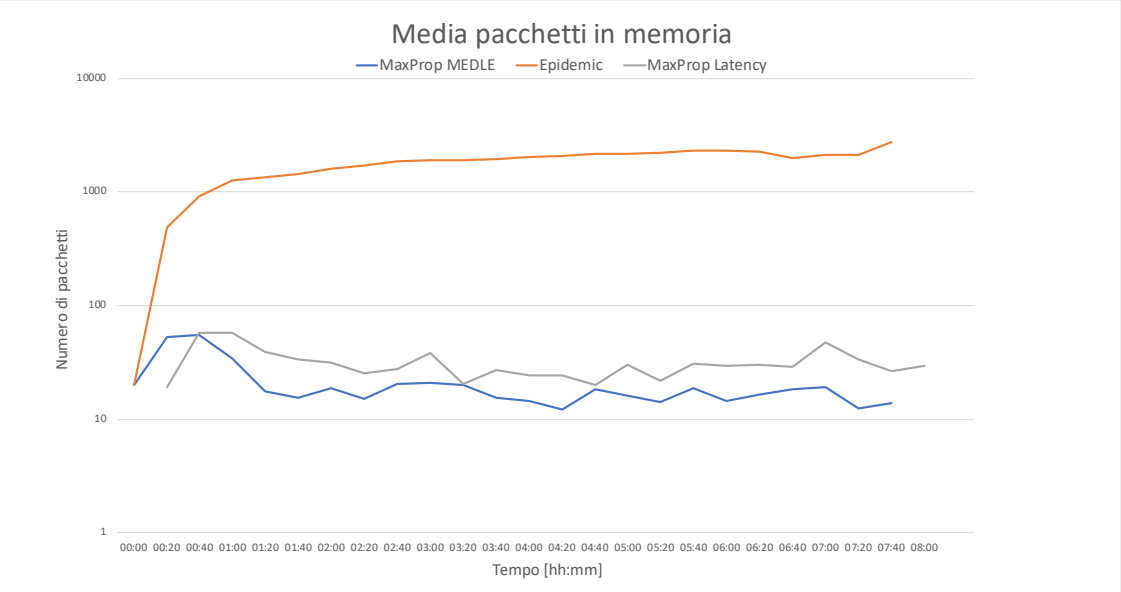


Figura 7.17. Storage

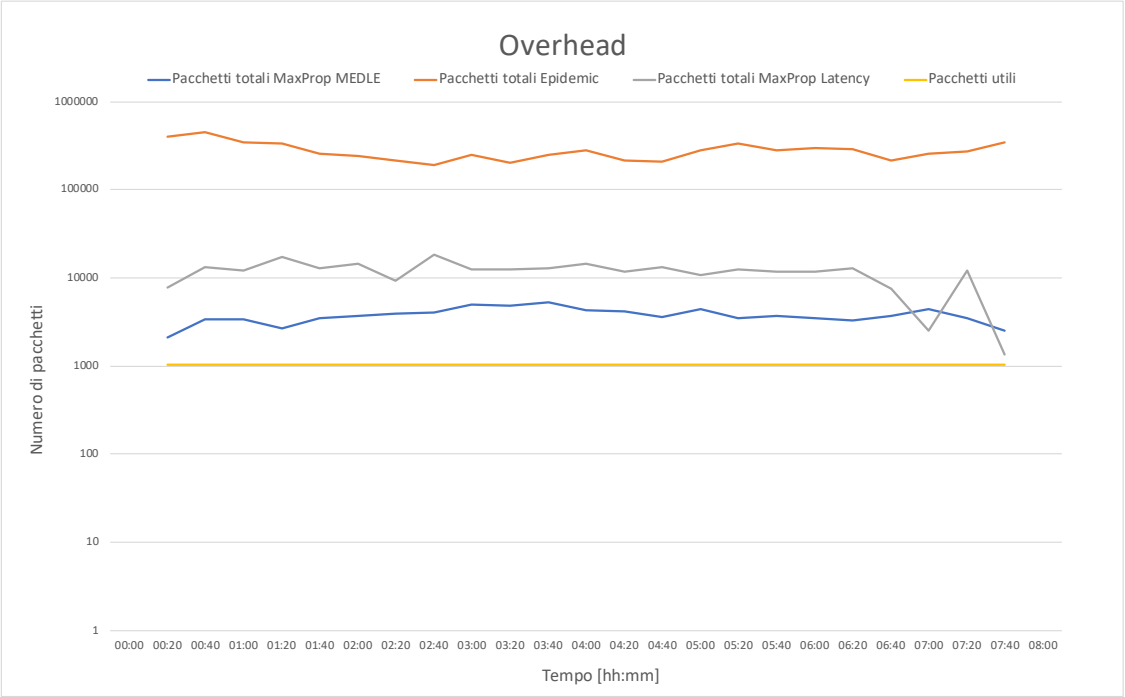


Figura 7.18. Overhead

Capitolo 8

Conclusioni

Il lavoro svolto in questa tesi implementa un protocollo di routing orientato ai servizi in una rete disrupted-tolerant, all'interno dell'applicativo Aether. In questo momento MaxProp può essere configurato in tre differenti versioni semplicemente attraverso un file di configurazione. La sua versatilità è sicuramente un punto a favore rispetto a protocolli già esistenti. Le versioni *MEDLE e Random* sono quelle classiche, basate sulla scelta del percorso in base ai costi ottenuti dalle probabilità di incontro; il secondo spesso utile in una rete di piccole dimensioni che vede il problema del congestionamento all'ordine del giorno. La versione *Latency Based* è quella sviluppata tenendo conto dello scenario descritto nei capitoli precedenti; questa tiene conto delle informazioni aggiuntive reperibili tramite interazione con la service discovery di Aether, e risulta molto valida in ambienti di cui si conoscono la composizione e i ruoli dei vari nodi.

Dati i risulti ottenuti possiamo constatare che *MaxProp* è una valida ed intelligente alternativa ad *Epidemic* il quale garantisce le percentuali di consegna più alte, ma richiede un alto consumo di risorse. *MaxProp*, invece, continua a mantenere la percentuali di consegna molto alte con un minor spreco di risorse, a discapito di un costo computazionale maggiore.

Come ulteriori sviluppi è necessario ottimizzare la computazione algoritmica e migliorare la fase di notification che è stato riscontrato essere molto onerosa. Per quanto riguarda la strategia *Latency Based* è necessario studiare una strategia più accurata per stimare la latenza dei percorsi, avendo riscontrato che sia sovrastimandole che sottostimandole i risultati peggiorano visibilmente.

Bibliografia

[1] C. Carrabbotta. Delay tolerant network in: Comunicazione self-optimized tra dispositivi eterogeneamente connessi ad una rete con tolleranza ai ritardi. Tesi, 2019.

[2] K. Scott. Disruption tolerant networking proxies for on-the-move tactical networks. In *MILCOM 2005 - 2005 IEEE Military Communications Conference*, 2005.

[3] K. Scott and S. Burleigh. Bundle protocol specification. RFC 5050, IETF, November 2007.

[4] S. Schildt, J. Morgenroth, W.B. P¨ o ttner, and L. Wolf. Ibr-dtn: A lightweight, modular and highly portable bundle protocol implementation. *Electronic Communications of the EASST, Volume 37: Kommunikation in Verteilten Systemen 2011*, 2011.

[5] Johannes Morgenroth. Ibr-dtn api.

[6] D. Ellard and D. Brown. Dtn ip neighbor discovery (ipnd). Internet-Draft draft-irtf-dtnrg-ipnd-01, IETF, March 2010.

[7] M. Demmer, J. Ott, and S. Perreault. Delay-tolerant networking tcp convergence-layer protocol. RFC 7242, IETF, June 2014.

[8] Johannes Morgenroth. Ibr-dtn configuration file example.

[9] K. Obraczka A. P. Da Silva, S. Burleigh. *Delay Distruption Tolerant Networks*. CRC Press, New York, NY, USA, 2018.

[10] A. Doria Technicalities E. Davies Folly Consulting S. Grasic Lulea University of Technology A. Lindgren, SICS. Probabilistic routing protocol for intermittently connected networks. RFC 6693, IRTF, August 2012.

[11] A. Doria Technicalities E. Davies Folly Consulting S. Grasic Lulea University of Technology A. Lindgren, SICS. Probabilistic routing protocol for intermittently connected networks draft-irtf-dtnrg-prophet-09. DRAFT 09, April 2011.

[12] D. Jensen J. Burgess, B. Gallagher and B. N. Levine. Maxprop: Routing for vehicle-based disruption-tolerant networks.

[13] Ari Keränen, Jörg Ott, and Teemu Kärkkäinen. The ONE Simulator for DTN Protocol Evaluation. Technical report, ICST, New York, NY, USA, 2009.

[14] S. Burleigh A. Hooke L. Torgerson NASA/Jet Propulsion Laboratory R. Durst K. Scott The MITRE Corporation K. Fall Intel Corporation H. Weiss SPARTA Inc. V. Cerf, Google/Jet Propulsion Laboratory. Delay-Tolerant Networking Architecture. RFC 4838. Technical report, IETF, 2007.

[15] Contiki: The open source OS for the internet of things. `http://www.contiki-os.org/`, 2018.

[16] Rpl (ipv6 routing protocol for llns). `https://en.wikipedia.org/wiki/RPL_(IPv6_Routing_Protocol_for_LLNs)`.

[17] Loadng-iot: An enhanced routing protocol for internet of things applications over low power networks. `https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6339092/)`.

[18] K. Hartke C. Bormann Universitaet Bremen TZI Z. Shelby, ARM. The Constrained application protocol, RFC 7252. `https://tools.ietf.org/html/rfc7252`, 2014.

[19] Université Bretagne Sud France team CASA, Laboratory IRISA. Lepton: a lightweight emulation platform for opportunistic networking. `https://casa-irisa.univ-ubs.fr/lepton/index.html`.