ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica - Scienza e Ingegneria

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

Fondamenti di Telecomunicazioni T

OPPORTUNISTIC CONTACT GRAPH ROUTING

CANDIDATO

Alessandro Berlati

RELATORE

Prof. Carlo Caini

Anno Accademico 2015/2016

Sessione I

Prefazione

L'ambiente di questa tesi è quello del Delay and Disruption Tolerant Networks (DTN), un'architettura di rete di telecomunicazioni avente come obiettivo le comunicazioni tra nodi di reti dette “challenged”, le quali devono affrontare problemi come tempi di propagazione elevati, alto tasso di errore e periodi di perdita delle connessioni. Esempi di questo tipo di reti sono i sistemi di connessione tra dispositivi nello spazio, come Internet Interplanetaria (IPN, composta da sonde interplanetarie, stazioni spaziali, satelliti), per i quali i classici protocolli (suite TCP/IP) non riescono a far fronte a ritardi significativi o all’oscuramento del segnale causato dall'interposizione dei pianeti. Il Bunde layer, un nuovo livello inserito tra trasporto e applicazione nell’architettura ISO/OSI, ed il protocollo ad esso associato, il Bundle Protocol (BP), sono stati progettati per rendere possibili le comunicazioni in queste reti. I bundle sono i pacchetti, anche di grande dimensione, utilizzati a questo livello. In questa architettura store-and-forward un bundle ricevuto viene prima memorizzato e successivamente inviato al nodo successivo quando possibile. A volte fra la ricezione e l’invio può trascorrere un lungo periodo di tempo, a causa della indisponibilità del collegamento successivo; in questo periodo il bundle resta immagazzinato in un database locale, spesso su memoria permanente per maggiore sicurezza. Esistono varie implementazioni dell'architettura DTN come DTN2, implementazione di riferimento, e ION (Interplanetary Overlay Network), sviluppata da NASA JPL, per utilizzo in applicazioni spaziali; in esse i contatti tra i nodi sono deterministici (perché dovuti al moto dei pianeti e delle navicelle spaziali), a differenza delle reti terrestri nelle quali i contatti sono generalmente opportunistici (non noti a priori). Per questo motivo all’interno di ION è presente un algoritmo di routing, detto CGR (Contact Graph Routing), progettato per operare in ambienti con connettività deterministica. È in fase di ricerca un algoritmo che opera in ambienti non deterministici, OCGR (Opportunistic Contact Graph Routing), che estende CGR.

L’obiettivo di questa tesi è quello di fornire una descrizione dettagliata del funzionamento di OCGR, partendo necessariamente da CGR sul quale è basato, eseguire dei test preliminari, richiesti da NASA JPL, ed analizzarne i risultati per verificare la possibilità di utilizzo e miglioramento dell’algoritmo. Sarà inoltre descritto l’ambiente DTN e i principali algoritmi di routing per ambienti opportunistici. Nella parte conclusiva sarà presentato il simulatore DTN “The ONE”, sviluppato dalla “Helsinki University of Technology” e l’integrazione di

CGR e OCGR al suo interno. Quest'ultima è stata svolta da Michele Rodolfi e Jako Jo Messina durante la tesi di laurea Magistrale in Ingegneria Informatica presso l'Università di Bologna, che ho poi esteso in collaborazione con Federico Fiorini, durante il tirocinio presso il Dipartimento di Ingegneria dell'Energia Elettrica e dell'Informazione (DEI).

Sommario

1 Introduzione

1.1 L'architettura DTN (Delay-/Disruption-Tolerant Networking)

Le comunicazioni in Internet avvengono attraverso la commutazione di pacchetto, di tipo datagram, molto simile all'inoltro della posta: un pacchetto IP rappresenta una parte di un blocco di dati che deve essere trasmessa da sorgente a destinazione, ogni pacchetto può prendere strade diverse per arrivare a destinazione a seconda delle decisioni prese dai nodi intermedi. Il funzionamento di Internet, cioè della suite di protocolli TCP/IP, si basa sui seguenti presupposti fondamentali:

- Un percorso bidirezionale continuo tra sorgente e destinazione è sempre disponibile.
- Round Trip Time (RTT) basso: il tempo di andata e ritorno da sorgente a destinazione è basso e relativamente uniforme in tutta la rete (tipicamente da poche decine a circa 200 millisecondi).
- Velocità di trasmissione simmetrica.
- Basso tasso di errore.

L'architettura DTN è stata progettata per superare i problemi delle cosiddette reti "challenged", ovvero quelle reti in cui uno o più dei quattro punti descritti sopra non sono soddisfatti; in particolare ciò accade sempre dove la connettività è intermittente e di conseguenza, non esiste un percorso continuo tra sorgente e destinazione. Attualmente lo scenario principale delle DTN è l'Internet InterPlanetario (IPN), in questo caso l'intermittenza delle connessioni è dovuta al moto dei pianeti che si interpongono tra i nodi, i RTT sono molto più elevati di quelli terrestri, la percentuale di dati perduti è spesso di un ordine superiore a quella terrestre ed inoltre è spesso presente asimmetria nella larghezza di banda tra i nodi. L'architettura DTN è descritta nel dettaglio in [RFC4838]; le caratteristiche principali sono:

- Definizione di un nuovo strato tra Applicazione e Trasporto, il "Bundle layer", presente nei nodi destinazione, di arrivo ed eventualmente in nodi intermedi, nei quali è l'ultimo strato dello stack, sempre sopra al trasporto. Un pacchetto a questo

livello è chiamato *bundle.* Il protocollo di questo livello è il "Bundle Protocol" descritto in [RFC5050]. Un nodo che implementa il bundle layer è detto *nodo DTN.*

- I nodi DTN possono essere raggruppati in gruppi detti *DTN Endpoints.* Ogni DTN endpoint è identificato da un *Endpoint IDentifier(EID),* un nome univoco espresso usando la sintassi degli URI.
- I bundle devono avere un luogo in cui essere memorizzati (database o memoria permanente) nel caso non possano momentaneamente proseguire verso la destinazione. Infatti non essendo garantita la presenza di un percorso continuo tra sorgente e destinazione, è necessario considerare la possibilità che un bundle debba attendere, anche per un lungo periodo, il prossimo *contatto,* ovvero la prossima opportunità di essere trasferito al nodo successivo. La memoria deve essere disponibile, ben distribuita nella rete e persistente in modo da mantenere i dati in caso di crash. Questo tipo di modello è detto store-and-forward e può essere accompagnato dall'opzione, richiesta dall'applicazione sorgente, di *Custody.* Un bundle trasmesso con la Custody Option attiva, comporta il passaggio della responsabilità di un'eventuale ritrasmissione dal custode attuale al custode successivo, con un meccanismo a staffetta. Un nodo DTN può tuttavia rifiutare la richiesta di Custody, ovvero di diventare il nuovo custode del bundle; ad esempio può non essere abilitato ad accettare richieste di Custody oppure può non accettarle in un particolare momento in quanto congestionato. Se accetta, diventa il nuovo custode, e segnala l'accettazione al vecchio custode, per liberarlo dell'incombenza; a questo punto, la copia del bundle presente nel vecchio custode può essere cancellata. Nel caso non riceva nessuna accettazione di custodia entro un tempo prestabilito, il custode re-invia il bundle. Il vantaggio della custodia è quello di consentire ritrasmissioni a partire da nodi vicini (ad esempio da un gateway su Marte ad un utente su Marte) anziché dalla sorgente (ad esempio sulla Terra), cosa di fondamentale importanza in presenza di lunghi ritardi e di collegamenti intermittenti.
- Due tipi di frammentazione: pro-attiva e reattiva. La prima permette di dividere i bundle in più frammenti, nel caso di contatti schedulati (o predetti) e intermittenti. Un contatto rappresenta, come detto prima, un'opportunità di trasmissione di dati verso un nodo; in base alla velocità di trasmissione e alla durata, un contatto, sarà associato anche ad un *volume* (a volte impropriamente detto *capacità*), ovvero la

quantità massima di byte che possono essere trasmessi in quell'arco di tempo. Conoscendo in anticipo i contatti, quindi i rispettivi volumi, è possibile frammentare a priori i bundle di dimensione maggiore al volume, anche residuo, del contatto durante il quale devono essere trasmessi. La frammentazione reattiva avviene invece nel caso di interruzione non prevista della connessione, in modo da non dover ritrasmettere se parte del bundle è già stata ricevuta, è possibile non ritrasmetterla. Questo tipo di frammentazione deve essere accompagnata da un livello sottostante che fornisca il supporto necessario (trasporto affidabile). In ogni caso il nodo destinatario deve essere in grado di ricostruire il bundle dai frammenti.

- Late binding: diversamente da quanto accade in Internet classico, dove gli indirizzi sono risolti tramite il Domain Name System (DNS) alla sorgente, nelle DTN il binding nome-indirizzo può avvenire anche in nodi intermedi oppure all'endpoint di destinazione. Questa scelta diventa necessaria quando il tempo di trasmissione di un messaggio potrebbe essere maggiore del tempo di validità del binding nome-indirizzo, in questo caso il binding alla sorgente non è possibile. Inoltre il late-binding consente di limitare i dati di mapping da propagare all'interno di una rete con disconnessioni frequenti.
- Esistono tre classi di priorità per i bundle:
    - - *Bulk*: priorità minore, nessun bundle di questa classe viene spedito fino a che sono in coda bundle di un altro tipo.
    - - *Normal*: inviati prima dei bulk ma si comportano allo stesso modo nei confronti degli expedited.
    - - *Expedited*: inviati con precedenza su tutti gli altri.

1.2 Differenze tra Routing DTN e Routing Internet

Considerando quanto introdotto nel paragrafo precedente risulta chiaro che l'instradamento, o routing, dei dati nelle DTN deve essere differente da quello usato in Internet classico. Definiamo il routing come la procedura attraverso la quale si calcola il miglior percorso per trasmettere dati da un nodo sorgente ad un nodo destinazione all'interno di una rete. Ogni nodo solitamente calcola in anticipo le rotte, per i dati che devono essere inviati, basandosi su informazioni di stato della rete come la topologia e/o la lista di tutte le connessioni che avverranno tra i nodi. In una DTN questa lista potrebbe includere informazioni come la velocità di trasmissione dei dati e la capacità archiviazione di ogni nodo. Le informazioni di

stato della rete non sono immutabili, ma possono cambiare nel tempo e in particolare il percorso migliore per raggiungere un nodo potrebbe variare mentre i dati sono in un punto intermedio. Possiamo quindi ridefinire il routing come la procedura con la quale, in ogni nodo nel percorso tra sorgente e destinazione, viene selezionato un nodo vicino verso il quale trasmettere i dati supponendolo nel miglior percorso verso la destinazione. Ci sono a questo punto due scelte: calcolare una nuova rotta ad ogni nodo intermedio, utilizzando le informazioni di stato della rete attuali, oppure continuare sul percorso calcolato precedentemente da un altro nodo. In Internet classico le informazioni sui cambi nella rete si propagano molto velocemente, si presuppone quindi che ogni nodo abbia una sempre una conoscenza quasi perfetta dello stato della rete e di conseguenza possa calcolare il percorso migliore con grande precisione.

Nelle reti "challenged" quanto detto sopra non è valido. Il routing rimane comunque un problema di scelta del percorso (o nodo vicino) migliore ma, data l'intermittenza dei contatti e i lunghi tempi di propagazione del segnale, i cambi nella topologia della rete possono avvenire più rapidamente di quanto le informazioni su di questi ultimi possano essere propagate. Di conseguenza la determinazione del percorso migliore deve considerare che lo stato corrente della rete non è completamente noto e, inoltre, che spesso non è possibile trasmettere tempestivamente i dati. La discontinuità dei collegamenti cambia completamente la natura del problema in quanto non esiste una strada migliore tra sorgente e destinazione ma una strada migliore in ogni istante di tempo. Il routing, di conseguenza non sarà più basato sulla topologia della rete, ma piuttosto sulla sequenza temporale dei contatti tra i nodi.

1.3 Algoritmi di Routing DTN

La progettazione di un algoritmo di routing ottimale può essere molto complicato data l'eterogeneità delle DTN. Parti diverse della stessa rete potrebbero appartenere a scenari in cui le politiche di routing dovrebbero seguire logiche differenti. Un algoritmo di routing potrebbe mirare al minor tempo di consegna così come al minor overhead. La soluzione migliore è quella ibrida; permette di interfacciare parti di rete eterogenee tra loro attraverso un gateway che fa da ponte tra i due domini di nodi. Un caso comune è quello di una rete che permette a nodi terrestri di comunicare con nodi su Marte attraverso delle sonde. È divisa in due parti: una connessa in modo classico (parte terrestre) che usa gli algoritmi noti, l'altra challenged (sonde e nodi su Marte) che utilizza gli algoritmi delle DTN.

In [Jain_2004] vengono distinte tre categorie di approcci per affrontare i problemi di routing nelle DTN: quelli che assumono la minima conoscenza dello stato della rete, detti "opportunistici", quelli che hanno una conoscenza parziale e quelli che assumono la completa conoscenza delle informazioni, detti "deterministici". Nel primo caso gli algoritmi si occupano di ottenere real-time il percorso migliore per il bundle da consegnare, attraverso lo scambio di calcoli tra i nodi. La maggior parte di questi algoritmi adotta varianti variamente temperate della strategia di *flooding*, replicando il messaggio più volte, rendendoli molto dispendiosi in termini di memoria. Se non ci sono problemi di dimensione dei buffer e la mobilità dei nodi è abbastanza alta, gli algoritmi opportunistici offrono una probabilità di consegna elevata, nonostante lo stato della rete non sia noto. Analizziamo più nel dettaglio alcuni di questi algoritmi opportunistici basati sulla replicazione:

- *Epidemic Routing:* [Vahdat_2000] è l'algoritmo più semplice in quanto non utilizza nessuna logica particolare. I bundle vengono trasmessi ad ogni nodo incontrato che non ne possieda già una copia. È un algoritmo affidabile ma, come già detto in precedenza, non preoccupandosi della replicazione dei bundle, tende ad occupare molta memoria.
- *Spray and Wait:* [Spyropoulos_2005] replica il bundle per un certo numero di volte, trasmette queste copie ad altri nodi ed attende fino a che il messaggio non raggiunge la destinazione.
- *MaxProp:* [Burgess_2006] è un algoritmo che segue una priorità di invio derivata dal calcolo della probabilità di consegna, basato sulla storia passata ed altri meccanismi accessori.
- *PRoPHET:* [Grasic_2010] l'algoritmo si basa sul fatto che spesso i contatti non sono completamente casuali e di conseguenza fa una stima sulla probabilità di consegnare un bundle trasmettendolo a un particolare nodo. Invece di replicare in modo casuale i bundle, si esegue il cosiddetto "routing probabilistico". Ogni nodo calcola una "delivery predictability", ovvero la probabilità di consegnare un bundle, attraverso ogni nodo conosciuto, verso ogni possibile destinazione. Quando due nodi che implementano PRoPHET si incontrano, si scambiano le relative "delivery predictability" in modo da aggiornare le informazioni disponibili e in seguito iniziare i trasferimenti. In particolare è utilizzata una regola transitiva secondo la quale, se il nodo B ha una buona probabilità di consegna verso C e il nodo A incontra

frequentemente B, allora i nodi che incontrano A potrebbero consegnare i bundle per C. Se la probabilità di consegna è sufficientemente alta il bundle viene replicato e consegnato al nodo. Una nuova versione, PRoPHETv2, nella quale si mantiene l'algoritmo originale ma viene modificata una formula per il calcolo della probabilità, è correntemente utilizzata visti i migliori risultati sperimentali.

- *RAPID:* [Balasubramanian_2007] è un algoritmo che permette di effettuare routing seguendo metriche differenti (minimizzare il tempo di consegna medio, minimizzare il tempo di consegna massimo, minimizzare il numero di bundle scartati in quanto esaurito il TTL). La replicazione dei pacchetti avviene in ordine di *Utility*. L'utility di replicazione di un bundle è calcolata per ogni contatto e sulla base della metrica adottata.

Nel secondo gruppo di algoritmi, ovvero quelli con conoscenza parziale dello stato della rete, troviamo algoritmi che prendono decisioni ricercando il percorso con il minor costo in termini di tempo. Non basandosi sul traffico nella rete il percorso sarà unico per quella destinazione, in una DTN potrebbe essere una limitazione pesante. Gli algoritmi principali che troviamo in questa categoria, nel dettaglio in [Jain_2004], sono:

- *Minimum Expected Delay (MED:* utilizza informazioni statistiche sui contatti per calcolare il percorso più breve che sarà riutilizzato da ogni bundle con la stessa coppia sorgente – destinazione.
- *Earliest Delivery (ED):* calcola il percorso più breve, attraverso l'algoritmo di Dijkstra, utilizzando informazioni sui contatti ma senza considerare la quantità di memoria libera nei nodi intermedi. Ottimale se i nodi intermedi non hanno messaggi in coda oppure se le capacità dei contatti sono elevate.
- *Earliest Delivery with Local Queuing (EDLQ)* e *Earliest Delivery with All Queues (EDAQ):* miglioramenti di ED che utilizzano, rispettivamente, le dimensioni delle code locali o di tutti i nodi nella rete per calcolare il percorso più breve.

Per ultimi descriviamo gli algoritmi appartenenti al terzo gruppo, i deterministici. Questi algoritmi lavorano in reti nelle quali i contatti sono noti a priori, raggiungendo alte probabilità di consegna con minor spreco di banda e memoria [Araniti_2015]. La lista dei contatti previsti è nota a tutti i nodi, permettendo la costruzione di grafi della rete che saranno

utilizzati per decisioni di routing *hop–by–hop.* Gli algoritmi opportunistici più noti sono MARVIN e Contact Graph Routing (CGR). Il CGR è l'esempio di un algoritmo che lavora con la conoscenza perfetta della rete, sarà trattato nel capitolo successivo, ed è attualmente in fase di testing e miglioramento anche una versione opportunistica, detta OCGR, che è appunto l'oggetto di questa tesi.

2 Il Contact Graph Routing

Il CGR è un algoritmo di routing dinamico, contenuto nella distribuzione ION realizzata da NASA-JPL, per reti DTN caratterizzate da connessioni intermittenti e note a priori. È progettato per operare in reti DTN spaziali, ma può chiaramente funzionale in reti terresti con contati opportunamente pianificati, cioè determinsitici.

2.1 Interplanetary Overlay Network – ION

ION è una distribuzione software che implementa l'architettura DTN. 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".

ION comprende una serie di moduli software, tutti scritti in linguaggio C, di cui elenchiamo i principali:

- *ici (Interplatetary Communication Infrastructure)* è un insieme di librerie fondamentali per le funzionalità comuni di tutti gli altri package come gestione delle liste ad alto livello, gestione della memoria più performante rispetto alle tradizionali *malloc() / free(),* gestione dei contatti.
- *bp (Bundle Protocol)* protocollo già analizzato precedentemente, assicura la trasmissione dei dati in reti "challenged", contiene inoltre l'algoritmo di routing CGR all'interno di *libcgr.c.*
- *ltp (Licklider Transmission Protocol)* protocollo di Trasporto adatto ad ambienti con lunghi ritardi di propagazione; può assicurare sia la trasmissione dei dati affidabile (red), basata su *Acknowledment,* timeout e ritrasmissioni, sia la consegna non affidabile (green). Segue le specifiche di [RFC5326].

I package citati più i rimanenti, ampiamente descritti in [ION], permettono alla distribuzione di fornire funzionalità quali consegna affidabile di file, consegna multicast, gestione della congestione, facilità di monitoraggio delle performance, robustezza, basso overhead ma

soprattutto portabilità e facilità di integrazione con le infrastrutture, eterogenee, di comunicazione sottostanti.

Solitamente gli EID seguono lo schema URI dtn, cioè sono del tipo "dtn://station1.mars.ma/example" ma per le comunicazioni spaziali risulterebbero troppo verbosi, introducendo overhead. Per questo motivo in ION si utilizza lo schema URI ipn, del tipo "ipn:*node_number_service_number"*, più sintetico, che oltretutto si avvicina alla rappresentazione di *indirizzo IP + porta.* Essendo ION progettato per permettere comunicazioni interspaziali, spesso tra nodi molto distanti, deve essere in grado di operare superando due particolari problemi: vincoli sulle comunicazioni e vincoli sulla potenza di calcolo (processori). Il primo vincolo riguarda in particolare le velocità di trasmissione delle informazioni, è necessario tenere presente che la potenza disponibile all'interno di un veicolo spaziale è limitata e le antenne sono relativamente piccole, conseguentemente il segnale irradiato è debole, limitando la velocità di trasmissione dei dati. In media la velocità di trasmissione da un veicolo spaziale verso la terra varia dai 256 kbps a 6 Mbps. Per quanto riguarda le trasmissioni nel senso opposto non è sicuramente un problema la potenza di cui dispongono i trasmettitori terrestri, ma la sensibilità dei ricevitori è limitata. Storicamente il volume dei dati che deve essere inviato verso le navicelle spaziali è molto inferiore rispetto a quello dei dati inviati verso Terra, ne risulta che in media la velocità di ricezione di un veicolo spaziale è nell'ordine di 1 / 2 kbps.

Il vincolo sui processori è dovuto in parte, ancora una volta, al limite sulla potenza elettrica disponibile e in parte alla necessità di protezione dalle radiazioni per evitare errori durante i calcoli. Processori con queste caratteristiche sono inevitabilmente meno performanti (in quanto poco richiesti) di quelli utilizzabili sulla terra. È necessario inoltre fare un uso accurato delle risorse ed evitare l'allocazione dinamica incontrollata in quanto potrebbe produrre errori imprevedibili compromettendo eventualmente una missione, nel caso di sonde autonome.

Per ridurre l'uso delle risorse ION fa largo uso della memoria condivisa, in questo modo, attraverso l'uso di semafori a mutua esclusione, è possibile condividere dati tra processi diversi velocemente ed efficientemente. Un'altra caratteristica di ION è l'uso intensivo di puntatori a strutture complesse, soprattutto per quelle che devono essere inserite in liste, limitando notevolmente le copie dei valori.

2.2 Concetti Fondamentali CGR

Il CGR sfrutta il fatto che ogni nodo conosce in anticipo tutti i contatti della rete, grazie a questa conoscenza è possibile evitare la fase di dialogo iniziale tra due nodi in quanto entrami 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.

Saranno introdotti ora i concetti fondamentali su cui si basa il CGR in modo da poter spiegare più agevolmente l'algoritmo in seguito.

2.2.1 Contact Plan

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 *ionrc* e processati da *ionadmin* che li mantiene successivamente in un database locale.

All'interno del contact plan sono presenti messaggi di due tipi: *contact* e *range.*

I messaggi contact sono composti, in questo specifico ordine da:

- Istante di inizio del contatto, in UTC, espresso in secondi.
- Istante di fine del contatto, in UTC, espresso in secondi.
- Numero ipn del nodo mittente.
- Numero ipn del nodo ricevente.
- La velocità di trasmissione nominale espressa in Byte per secondo.

a contact +456 +600 23 45 256000

Dai messaggi contact viene ricavata la capacità dei contatti, è banalmente la durata del contatto moltiplicata per la velocità di trasmissione, utilizzata successivamente dall'algoritmo.

Le istruzioni di range contengono invece:

- Istante di inizio e fine del contatto in UTC espressi in secondi.
- Numero ipn dei nodi mittente e ricevente.

- La distanza attesa tra i due nodi in questo intervallo di tempo, indicata in secondi-luce. Deve essere intesa come ritardo di propagazione dei dati.

Un esempio di messaggio range è il seguente:

a range +456 +600 23 45 12

La principale differenza tra i due messaggi è che le istruzioni contact sono sempre unidirezionali, devono essere quindi ripetute con i nodi in ordine inverso nel caso di comunicazioni bidirezionali. Le istruzioni di range funzionano diversamente, se il primo nodo indicato è quello con il numero ipn più basso l'istruzione è implicitamente bidirezionale, tuttavia è sempre possibile specificare l'istruzione di range anche per il percorso inverso indicando un tempo di propagazione differente. È necessario fare chiarezza su questo punto in quanto è chiaro che due nodi si trovano alla stessa distanza uno dall'altro, ma gli sviluppatori hanno previsto la possibilità di trasmettere messaggi attraverso percorsi diversi nei due sensi, rendendo necessario specificare tempi di propagazione differenti. Nel caso l'unica istruzione di range sia inserita con il numero del nodo più alto per primo, il link viene considerato unidirezionale, quindi non utilizzabile in senso inverso.

2.2.2 Tabelle di Routing

A partire dai messaggi del contact plan ogni nodo crea localmente una tabella di routing. Le tabelle di routing contengono una lista di *route_list* verso tutti i possibili nodi destinazione che vengono riportati almeno una volta nel contact plan. Una route_list contiene tutte le rotte per uno specifico nodo destinazione ed ogni rotta rappresenta il percorso dal nodo locale a quello destinazione per una determinata *payload class*. La payload class è un limite superiore alla dimensione dei bundle che possono essere trasmessi attraverso quella rotta. Ogni percorso è caratterizzato da un nodo iniziale (*entry node*), e da un contatto iniziale (cioè si ha una diversa route list per ogni diverso contatto verso lo stesso nodo), e dalla lista di tutti gli altri contatti 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, quella utilizzata correntemente in ION è il tempo di consegna, che possono variare tra le diverse versioni di CGR (il CGR ha infatti subito una lunga evoluzione, peraltro non ancora terminata). È importante tuttavia non avere metriche diverse all'interno della stessa rete in quanto potrebbero portare a dei loop con conseguente fallimento della consegna.

2.2.3 Volume

Abbiamo già parlato di volume di un contatto, ovvero la quantità massima di dati che può essere inviata dall'istante di inizio a quello di fine, data appunto dal prodotto tra la durata del contatto e la velocità nominale di trasmissione dei dati. Nel CGR a partire dal concetto di volume vengono introdotti due nuovi termini: *estimated capacity consumption (ECC)* e *residual capacity* dove con capacità si intende il volume. La dimensione di un bundle è data dalla somma di payload e intestazione (header), ma questo valore non coincide con la porzione di capacità del contatto che viene utilizzata in caso di invio. Un bundle infatti viene incapsulato, secondo il classico schema dello stack ISO/OSI, all'interno di un messaggio del livello inferiore e questo a sua volta fino all'ultimo livello. La somma di tutte le intestazioni più la dimensione del bundle è la capacità effettivamente utilizzata in caso di trasmissione e coincide con la ECC. Ovviamente il calcolo di ECC deve tenere conto della diversità dei livelli sottostanti (*convergence-layer*) e stimare questi valori in base all'effettivo protocollo sottostante.

Consideriamo un contatto tra il nodo locale e un altro nodo D. La capacità residua per quel contatto, calcolata per un certo bundle, è la somma delle capacità di quel particolare contatto e tutti quelli precedenti verso lo stesso nodo, meno le ECC di tutti i bundle con priorità maggiore o uguale al bundle in oggetto che sono attualmente in coda verso D.

2.2.4 Outduct

Un outduct non è altro che una coda di bundle che devono essere trasferiti a un particolare nodo. Una volta che l'algoritmo avrà calcolato la rotta migliore per un bundle, lo inserirà nell'outduct per il nodo inziale della rotta. L'outduct è composto in realtà da tre code diverse, una per ogni classe di priorità descritta dal Bundle Protocol, in modo da poter accedere velocemente ai diversi tipi ed estrarre direttamente dalla coda che ci interessa.

2.2.5 Critical Bundle

Un critical bundle è un messaggio urgente, che deve raggiungere la destinazione nel minor tempo possibile. Inviando una sola copia attraverso il nodo che garantisce il tempo di consegna più breve potremmo incorrere in ritardi non previsti. Per questo motivo un bundle critico viene replicato verso tutti i nodi che garantiscono una rotta verso la sua destinazione. Da notare che diverse copie del bundle potrebbero raggiungere il nodo finale.

2.3 L'algoritmo

Un collegamento fra due nodi è destinato ad aprirsi e chiudersi varie volte, ad istanti di tempo definiti a priori. Ogni intervallo, o contatto, sarà definito (se bidirezionale) da una coppia di istruzioni contact. L'intermittenza dei contatti rende completamente diverso il routing CGR da quello classico, perché nell'ambiente considerato dal CGR la topologia della rete è dinamica, quindi non esiste un percorso migliore fra sorgente e destinazione, ma un percorso migliore ad ogni istante di tempo. La maggior complessità è evidente.

Possiamo definire quattro fasi dell'algoritmo:

1. Calcolo delle Rotte (Dijkstra),
2. Selezione delle rotte accettabili,
3. Selezione di una rotta (massimo) per ogni Proximate node,
4. Inserimento del bundle in coda.

Descriviamo nel dettaglio ogni fase supponendo di dover inviare un bundle che ha come destinazione il nodo D. Per prima cosa si esegue un controllo preliminare, se nessun contatto del contact plan riguarda D non è possibile usare il CGR. Se il contact plan è stato modificato da quando è stato eseguito l'ultimo calcolo delle rotte è necessario invalidare tutte le rotte per ogni nodo. A questo punto se la tabella di routing contiene già le rotte calcolate per D si passa alla seconda fase, altrimenti si effettua il calcolo delle rotte.

Fase 1: Calcolo delle rotte.

Si crea una lista di *Proximate Node* (nodi con i quali abbiamo un contatto diretto) ed una lista di *Excluded Node* attraverso i quali non calcoleremo alcuna rotta per questo bundle. All'interno di quest'ultima lista sarà inserito inizialmente il nodo dal quale abbiamo ricevuto il bundle, per evitare cicli, e tutti i nodi che hanno rifiutato precedentemente una richiesta di custody per un bundle con destinazione D.

Si costruisce successivamente un grafo dei contatti a partire dal contact plan. La radice del grafo è un contatto fittizio tra il nodo locale e sé sesso, gli altri vertici rappresentano tutti i contatti ed infine esiste un vertice terminale che è, ancora una volta, un contatto fittizio tra D e sé stesso. Una volta costruito il grafo si eseguono una serie di ricerche di Dijkstra, una per ogni payload class e per ognuna di queste salviamo, nella lista di rotte per D, quella con

il costo più basso. Dopo ogni ricerca si elimina il contatto iniziale della rotta selezionata dal grafo e si esegue un'altra ricerca. Alla prima ricerca fallita si termina il ciclo.

Fase 2: selezione delle rotte accettabili per il bundle corrente.

La fase 1 calcola una lista di rotte che possono essere utilizzate da più bundle, infatti le rotte non vengono ricalcolate a meno di cambi nel contact plan. In questa fase, dunque, sarà sempre disponibile una lista di rotte verso D che verranno analizzate e opportunamente scelte. In sintesi le rotte possono venire scartate in quanto non più valide (tempo di terminazione della rotta passato), se il tempo di arrivo previsto in caso di invio attraverso la particolare rotta è maggiore del TTL del bundle, se l'outduct di partenza è bloccato (persa la connessione) oppure se il payload del bundle è maggiore di quello massimo per la rotta e non può essere frammentato.

Fase 3: Una rotta per ogni Proximate Node

Abbiamo a questo punto le rotte che sono accettabili per il bundle da consegnare. Scorriamo la lista per trovare la rotta migliore per ogni Proximate Node. Per ogni rotta ci sono due possibilità:

- L'entry node non è presente nella lista dei Proximate Node. In questo caso viene semplicemente aggiunto, correlato dal numero di nodi previsti nel percorso e dal *forfeit time* per questa rotta. Il forfeit time indica l'istante dopo il quale la rotta non è più utilizzabile e quindi entro il quale il bundle deve essere stato trasmesso.
- L'entry node della rotta è già nella lista dei Proximate Node. In questo caso, se il tempo di consegna della rotta che stiamo analizzando è minore di quello presente nella lista dei Proximate Node, il tempo viene aggiornato assieme al nuovo numero di nodi ed al nuovo *forfeit time.* Da notare che i tempi di consegna potrebbero coincidere, in questa eventualità verrebbero aggiornati solo gli ultimi due parametri nel caso il numero di hop della rotta in esame sia minore.

Fase 4: Inserimento del bundle in coda

Anche in questa fase abbiamo due possibilità: il bundle da trasmettere è un critical bundle oppure è un bundle normale. Se ci troviamo nel primo caso il bundle è messo in coda, in multipla copia, negli outduct di tutti i Proximate Node, da notare la differenza tra questa

operazione ed un classico *flooding;* infatti con questo algoritmo il bundle non verrà trasmesso a nodi che non hanno alcuna possibilità di consegnarlo. Nel caso di bundle normale invece verrà inserito solo nell'outduct di uno dei Proximate Node disponibili. I criteri per la scelta sono, in quest'ordine di importanza, minor tempo di consegna, minor numero di nodi sul percorso, minor numero ipn dell'entry node.

2.4 Miglioramenti dell'algoritmo

Nel calcolo delle rotte il CGR considera che i bundle possano essere spediti esattamente all'apertura del contatto, non vengono sommati quindi i ritardi dovuti all'invio di bundle che sono già in coda (con priorità maggiore o uguale). ETO è un miglioramento di CGR, che permette di calcolare con una maggior precisione il tempo di consegna di un bundle, considerando i bundle, aventi priorità uguale o maggiore a quello da inviare, che sono già in coda Ovviamente il calcolo è facile per gli outduct del nodo locale, meno per quelli dei nodi seguenti che infatti vengono trattati come in precedenza. Maggiori dettagli tecnici sull'algoritmo possono essere trovati in [Bezirgiannidis_2014], nel quale è descritto anche Overbooking Management, il secondo miglioramento di cui andiamo a parlare. Nell'algoritmo originale del CGR, durante il calcolo della capacità residua, vista da un bundle in base alla sua priorità (ovvero senza calcolare quelli meno prioritari), non si teneva conto dei bundle che, anche se già schedulati, sarebbero stati spediti nel contatto successivo (o non spediti) in quanto "superati" dagli altri più prioritari. Allo scadere del *forfeit time* sarebbe stata ricalcolata una nuova rotta. L'Overbooking management è un miglioramento che permette di ricalcolare immediatamente una rotta per i bundle che vengono scavalcati da quelli prioritari e di conseguenza mancheranno il contatto. Dato che si può determinare a priori quali saranno questi bundle l'overbooking management ricalcolerà la rotta in anticipo e non allo scadere del forfeit time. Questo funzionamento scatta solo quando la capacità residua del contatto vista dal bundle, in base alla sua priorità, è maggiore della sua dimensione, mentre quella totale, ovvero senza tenere conto delle priorità, è minore. Questo significa che una volta inserito il bundle nella coda gli ultimi non saranno inviati in tempo. Abbiamo due tipi di overbooking:

- Partial Overbooking: quando la capacita totale residua è positiva, l'algoritmo toglie dalla coda un bundle alla volta, partendo dall'ultimo, fino a che la dimensione del bundle da inserire non è minore o uguale alla nuova capacità residua totale.

- Total Overbooking: quando la capacità totale residua è nulla, in questo caso non si comincia a togliere i bundle dall'ultimo ma dall'ultimo bundle previsto per il contatto che ci interessa (in pratica potrebbero essercene in coda, nello stesso outduct, cioè verso la stessa destinazione, alcuni associati al contatto successivo al nostro; essi non devono essere tolti).

3 Opportunistic Contact Graph Routing

3.1 ION in ambiente opportunistico

Il CGR è un algoritmo introdotto per calcolare le rotte dei bundle in modo deterministico, utilizzabile principalmente in ambito spaziale dove è facile prevedere in anticipo i contatti tra i nodi e di conseguenza costruirne un grafo. L'architettura DTN ha l'obbiettivo di lavorare in tutte le tipologie di rete challenged, quindi anche in ambienti opportunistici come, ad esempio, le reti cellulari terrestri. ION, per realizzare questo obiettivo, aveva bisogno di un nuovo meccanismo che permettesse di fare routing in entrambi gli ambienti.

Nell'ultima versione di CGR sono supportati i contatti non certi, ovvero che non avverranno sicuramente ma con una probabilità (*confidence)* riportata nel contact plan. Questi contatti probabilistici devono essere inseriti a mano nel contact plan così come si inseriscono anche i contatti certi. Durante il calcolo delle rotte si considera quindi anche la probabilità di consegna (*delivery confidence*) attraverso la quale viene deciso se inviare più copie del bundle attraverso più rotte oppure se inviarne solamente una, nel caso in cui la probabilità di consegna si abbastanza alta.

Sono stati descritti precedentemente algoritmi opportunistici basati su flooding, più o meno controllato, come Epidemic e Spray and Wait, oppure altri che eseguono scelte di routing attraverso meccanismi probabilistici come PRoPHET. Opportunistic Contact Graph Routing (OCGR) è un algoritmo che estende CGR, ed utilizza conseguentemente le ricerche di Dijkstra per il calcolo della rotta migliore, descritte nel capitolo precedente. La sfida è quella di rendere automatico l'inserimento di contatti probabilistici nel contact plan ottenendo delle probabilità di contatto plausibili. OCGR si basa sul principio secondo il quale più volte è stato incontrato un nodo, più alta sarà la probabilità di incontrarlo di nuovo. Attraverso questo principio l'algoritmo calcola la probabilità del verificarsi di un contatto, utilizzando la storia dei contatti precedenti, cercando, diversamente da algoritmi come PRoPHET, di prevederne anche l'istante di inizio, l'istante di fine e la velocità di trasmissione. Questi dati sono fondamentali in quanto non si vuole modificare il modo in cui l'algoritmo del CGR calcola la rotta migliore basandosi sul contact plan. I contatti calcolati saranno quindi aggiunti al contact plan seppure la probabilità che questi avvengano è minore di 1.0.

3.2 L'algoritmo di predizione dei contatti

Dall'analisi dei contatti di reti opportunistiche è risultato che questi non sono completamente casuali. Nella maggior parte dei casi, infatti, contatti tra una coppia di nodi tendono a seguire una certa distribuzione di probabilità. Sulla base di questa scoperta l'algoritmo di predizione dei contatti di OCGR tiene in considerazione tutti i contatti avvenuti precedentemente.

I contatti non schedulati vengono automaticamente scoperti real-time, offrendo possibilità di scambio dei dati immediata. Durante la fase iniziale di un contatto i due nodi si scambiano le liste dei contatti avuti precedentemente. Successivamente entrambi scartano tutti i contatti predetti e li ricalcolano utilizzando anche le nuove informazioni.

La predizione dei contatti è eseguita per ogni coppia di nodi presente nella lista dei contatti precedenti. In particolare ci interessano, per ogni contatto di una coppia di nodi, tre proprietà: la durata del contatto (istante di fine meno istante di inizio), il *gap* tra un contatto e l'altro, ovvero il tempo che intercorre tra la fine di un contatto e l'inizio di quello successivo, e la velocità di trasmissione. Analizzando i contatti precedenti è possibile calcolare media e deviazione standard delle tre proprietà. Le medie saranno utilizzate per calcolare istante di inizio, istante di fine e velocità di trasmissione del contatto previsto, mentre le deviazioni standard ci danno una indicazione sulla casualità dei contatti. Possiamo infatti dire che se la deviazione standard è relativamente bassa allora i contatti stanno seguendo un certo schema ed è probabile che il prossimo contatto avvenga nei tempi previsti dai valori medi calcolati. Se invece la deviazione standard è alta i contatti sono avvenuti in modo più casuale e la probabilità che il prossimo contatto sia quello previsto si abbassa. Tuttavia l'effettivo collegamento tra media e deviazione standard dipende dal tipo di scenario e la soglia con la quale definiamo se la deviazione standard è alta o bassa è un parametro da regolare con attenzione.

Si vuole ovviamente tenere conto anche del numero di contatti che sono avvenuti in precedenza. Abbiamo infatti detto che più volte due nodi sono stati connessi, più la probabilità che si verifichi un altro contatto aumenta. Inoltre se la storia dei contatti è breve la deviazione standard è poco significativa e la confidence di un contatto deve essere conseguentemente più bassa.

Definiamo quindi due parametri:

- *Base confidence:* la probabilità di partenza di una serie di contatti previsti (per una coppia di nodi), può assumere due valori:
    - *High Base Confidence* nel caso la deviazione standard delle durate e gap dei contatti precedenti sia alta.
    - *Low Base Confidence* nel caso contrario.
- *net confidence:* la confidence effettiva dei contatti previsti, calcolata attraverso la formula seguente:

$$1 - (1 - B)^N$$

  Dove N è il numero di contatti precedenti per la coppia di nodi e B la Base Confidence.

In questo modo più è elevato il numero di contatti precedenti più la probabilità del contatto previsto si alza. I due valori di base confidence (0.05 e 0.2 nella versione attuale) e la formula di calcolo della net confidence, ci forniscono dei gradi di libertà per la modifica dell'algoritmo nel caso i risultati non siano ottimali.

Il calcolo della confidence è la parte più importante dell'algoritmo in quanto condiziona le scelte di routing fatte dall'algoritmo. Se la confidence stimata è troppo bassa il bundle viene replicato e trasmesso a più nodi provocando overhead e possibile congestione. Se, invece, la confidence stimata per una rotta è troppo alta l'algoritmo accoda il bundle in uno specifico outduct senza provare altre strade. Il bundle a questo punto potrebbe rimanere bloccato in attesa di un contatto che non si verificherà nel breve periodo mancando tutti i contatti per una rotta migliore.

È definito poi il "prediction horizon", per una coppia di nodi, come l'istante di tempo attuale più la differenza tra l'istante di tempo attuale e l'istante di inizio del primo dei contatti tra i due nodi. Il prediction horizon è il limite massimo per il quale l'algoritmo predice dei contatti, se la storia dei contatti è breve di conseguenza i contatti predetti saranno in numero inferiore evitando contatti troppo azzardati. Per ogni coppia di nodi l'algoritmo calcolerà un numero di contatti, in accordo con il prediction horizon, i quali avranno durata, gap e volume stimati in base ai valori medi di quelli precedenti. La confidence di questi contatti sarà calcolata in base alla deviazione standard di durata e gap dei contatti precedenti.

Il CGR infine calcolerà le rotte migliori attraverso l'algoritmo esistente per decidere se inviare il bundle verso un solo proximate node o replicarlo ed accordarlo in più outduct.

3.2.1 Il log dei contatti

Per la raccolta dei contatti passati sono stati aggiunti al database di ION due *contactLog*: uno per i nodi mittente ed uno per i riceventi, entrambi contenenti tutti i contatti terminati dal nodo locale o passati ad esso da un altro nodo. Per evitare ulteriore incertezza, i contactLog contengono solo contatti che sono già terminati. In questo modo l'algoritmo di predizione, probabilistico per natura, si baserà solamente su dati certi e ben definiti. L'unica valore non perfettamente noto è l'istante di fine di un contatto ed è questo il motivo della presenza di due contactLog. Il tempo di inizio del contatto è esatto in quanto registrato allo stesso momento dal "neighbor discovery" (un demone in attesa di contatti non previsti), è uguale sia per mittente che per ricevente. L'istante di fine del contatto invece potrebbe differire tra i due nodi in quanto identificato dallo scadere di un timeout. L'istante di fine di un contatto viene considerato più attendibile se calcolato da un nodo mittente, per questo motivo le voci all'interno del contactLog dei nodi riceventi vengono considerate solo se non esiste la voce corrispondente nel contactLog dei nodi mittente.

Per facilitare le operazioni di lettura dai contactLog le voci sono elencate in ordine crescente di:

- Numero ipn del nodo mittente.
- Numero ipn del nodo ricevente.
- Istante di inizio del contatto.

4 Simulazioni di OCGR con “The ONE” Simulator

“The ONE Simulator” è un ambiente di simulazione DTN, il cui scopo è lo studio delle prestazioni degli algoritmi di routing, in particolare opportunistici, in cui i contatti fra i nodi sono legati alla distanza, e quindi al movimento dei nodi stessi. Per questo motivo ONE è in grado di generare il movimento di nodi usando diversi modelli di movimento e instradare i messaggi generati attraverso vari algoritmi di routing opportunistico.

In questo capitolo sarà presentata la struttura ed il funzionamento generale del simulatore, sarà data una breve spiegazione di come gli algoritmi CGR ed OCGR sono stati integrati e successivamente saranno presentati e commentati i risultati di simulazioni con OCGR.

4.1 Struttura del software

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. La flessibilità di questo software lo rende perfetto per testare nuove tecnologie e confrontarle con quelle già affermate grazie anche alla possibilità di ripetere simulazioni identiche tra loro.

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.

Ogni nodo è rappresentato da un'istanza di *DTNHost* durante la simulazione, che contiene il suo indirizzo, il router che implementa, le coordinate che indicano dove si trova e dove è diretto, il percorso e la velocità di movimento. I modelli di movimento (package *Movement)* determinano l'algoritmo con il quale i nodi si spostano: in modo casuale o seguendo percorsi definiti da una mappa, che può essere fornita al simulatore in formato WKT, oppure simulando una giornata lavorativa di una persona (dormire a casa, lavorare in ufficio, uscire

la sera) secondo il modello *Working Day Movement*. Gli algoritmi forniti per il movimento su una mappa sono

- *Random Map-Based*: i nodi si muovono casualmente ma seguendo i percorsi forniti dalla mappa.
- *Shortest Map-Based*: i nodi seguono la strada più corta per raggiungere un punto di interesse, casualmente generato, dalla posizione in cui si trovano.
- *Routed Map-Based*: i nodi seguono strade predeterminate.

Il package *Routing* contiene le classi che implementano il router all'interno di ogni singolo nodo. Tutte le classi derivano dalla classe astratta *Message Router* che contiene le mappe dei messaggi in arrivo, quelli da consegnare e quelli arrivati a destinazione. Questa classe è estesa da *Active Router* e *Passive Router,* quest'ultimo non invia messaggi se non sotto indicazione specifica, è quindi utilizzato per il testing di routing controllato o per nodi di sola ricezione. *Active Router* è invece la classe astratta che implementa i metodi per ricevere, eliminare o inviare un messaggio, recuperare le connessioni attive al momento dell'invocazione e recuperare i messaggi nel buffer. Ogni algoritmo di routing fornito dagli sviluppatori estende questa classe ed implementa il metodo *update* chiamato ogni volta che il simulatore aggiorna lo scenario. Gli algoritmi di routing più importanti già implementati sono: *Direct Delivery* (il messaggio viene consegnato solo al destinatario finale), *First Contact* (il messaggio viene inviato solo al primo nodo connesso), *Spray and Wait*, *Epidemic, PRoPHET e PRoPHETv2*. La classe *Message* definisce i messaggi scambiati dai router, gli attributi principali sono il nodo mittente e quello di destinazione, la dimensione in byte e un time to live dopo il quale vengono scartati dal router.

Nel package input sono presenti le classi che generano gli eventi della simulazione. Gli eventi possono essere letti da un file esterno oppure generati internamente come nel caso dei *MessageEventGenerator*. Questi ultimi possono essere casualmente creati sui nodi oppure seguire logiche come, ad esempio, inviare un messaggio verso ogni nodo o da ogni nodo. Gli eventi possono essere di vario tipo, creazione, passaggio, cancellazione di messaggi così come eventi di connessione tra nodi. Fanno parte di questo package anche i vari lettori che ricevono in input mappe per la simulazione, percorsi, punti di interesse e eventi generici.

Il package report contiene le classi che producono i documenti per analizzare le statistiche post-simulazione. Possono essere utilizzati per verificare quali contatti sono avvenuti, il percorso che seguono i messaggi per arrivare a destinazione oppure le statistiche della simulazione, come numero di messaggi creati, consegnati, eliminati, tempi medi di consegna, tempo medio nel buffer, overhead, ecc...

È ovviamente presente anche un package *core*, che contiene la classe cuore di tutto il software DTNSim, in questa classe è contenuto il main e sono svolti tutti i passi preliminari della simulazione. Attraverso le class *World e SimScenario* successivamente vengono inizializzati i nodi, i router e tutto l'ambiente di simulazione (mappe e modelli di movimento).

4.2 Simulazione e Report

Le simulazioni possono essere lanciate in modalità "batch" oppure con interfaccia grafica. I parametri devono essere inseriti in un file di configurazione (`default_settings.txt` di default) che deve essere specificato al momento del lancio della simulazione. Il file è strutturato con comandi del tipo

```
PrimaryNamespace.SecondaryNamespace = value
```

Il file è modificabile liberamente ma è necessario essere precisi nell'inserimento di namespace e valori, in caso contrario la simulazione terminerà con un errore.
I principali parametri sono:

- `Scenario.endTime` che indica la durata della simulazione in secondi, intesi come secondi trascorsi nello scenario simulato (la durata effettiva della simulazione di norma è molto minore, dipendendo dalla potenza di calcolo del processore).
- `Scenario.nrofHostGroups,` il numero di gruppi di host (cioè di tipi di host aventi le stesse caratteristiche).
- `Group.router` per indicare il tipo di router.
- `Group.nrofHost` per indicare il numero di nodi.
- `Events.nrof` è il numero di generatori di eventi.
- `Event*.class` per impostare il tipo di generatore di eventi.

- `Report.nrof` e `Report.”reportSpecifico”` per indicare numero e tipo di report.

E’possibile specificare opzioni comuni a tutti i gruppi oppure relative al singolo gruppo; ad esempio `Group.nrofHost=4` indica che tutti i gruppi avranno 4 nodi, mentre `Group1.router = EpidemicRouter` indica che solo il primo gruppo avrà router di tipo Epidemic.

Se vengono inserite nel file di impostazioni le opportune opzioni, al termine della simulazione si possono trovare nella cartella “reports” i file contenenti dati delle simulazioni.

```
Report.nrofReports = 1
Report.report1 = classeReport
```

Uno dei report più importanti è sicuramente *MessageStatsReport* che contiene tutte le statistiche sui messaggi inviati durante la simulazione (numero, probabiltà di consegna, messaggi creati, scartati). Altri report utili sono *EventLogReport,* che indica ogni evento di connessione della simulazione (analizzando il file prodotto possono essere ricavati i contatti), e *DeliveredMessageStatsReport* con le informazioni dei soli messaggi che sono stati consegnati, compresa la lista dei nodi di passaggio.

Lanciando la simulazione possono essere forniti dei parametri. Nel caso nessun parametro venga passato ONE si avvierà in modalità interfaccia grafica utilizzando il file di setting “default_settings.txt” che, se non trovato, farà terminare la simulazione. Se invece vengono passati uno o più file di configurazione, i comandi al loro interno verranno valutati e sommati a quelli di default_settings (in questo caso se non esiste la simulazione può continuare). Se due comandi vanno in conflitto (stesso nome) il comando letto per ultimo sarà quello utilizzato. Sapendo questo è utile tenere nel file di default tutti gli elementi base, mentre in quelli aggiuntivi i campi che vogliamo far variare nelle simulazioni. Se il primo parametro passato è “-b” ONE sarà eseguito in modalità batch; anche se non sarà possibile vedere lo stato della simulazione che avanza, questa modalità è notevolmente più rapida nell’esecuzione, rendendola preferibile alla modalità GUI. È inoltre possibile specificare quante simulazioni consecutive devono essere eseguite. Un esempio di esecuzione potrebbe essere il seguente:

```
./one.sh -b 2 mysettings epidemicsettings
```

Specifica due esecuzioni, in modalità batch utilizzando i file di configurazione indicati.

4.3 Integrazione di CGR e OCGR

Per il testing di CGR e OCGR era necessario predisporre un testbed con più macchine virtuali ION, che si connettevano tra loro seguendo un contact plan e scambiandosi messaggi. Per quanto efficace, questo metodo può risultare dispendioso e relativamente lento. L'utilizzo di ONE per testare gli algoritmi di ION avrebbe aumentato notevolmente la velocità di simulazione, facilitandone lo sviluppo ed il miglioramento.

Il problema principale da risolvere era il disaccoppiamento tra i linguaggi, infatti mentre ONE è scritto in Java, ION è completamente in C. L'unico modo per non riscrivere tutto il codice dell'algoritmo in Java, soluzione oltretutto non scalabile, era utilizzare la Java Native Interface (JNI). Tramite JNI è possibile invocare funzioni C direttamente dal codice Java e viceversa, in questo modo se il codice del CGR/OCGR viene cambiato, nulla deve essere modificato nella parte Java. Inoltre gli algoritmi utilizzano le informazioni ricavate dall'ambiente ION, era quindi necessario creare delle nuove librerie che recuperassero queste informazioni dall'ambiente di simulazione di ONE. Sono state aggiunte delle classi per ONE (ContactGraphRouter, OpportunisticContactGraphRouter, PriorityMessage, ecc..), anche in modo da poter mantenere i file di configurazione coerenti, e classi di interfacciamento sia lato Java che lato C per lo scambio di informazioni tra i due ambienti.

L'integrazione di CGR è stata svolta da Michele Rodolfi e Jako Jo Messina, due laureandi del corso di Ingegneria Informatica dell'Università di Bologna, e documentata in [Rodolfi_2016] e [Messina_2016], mentre Io e Federico Fiorini ci siamo occupati dell'aggiunta dei messaggi con priorità in ONE e del successivo utilizzo all'interno dell'integrazione, con l'aggiunta di Overbooking Management. L'integrazione di OCGR è stata svolta sempre da Michele Rodolfi ampliando quella già presente di OCGR, nel capitolo successivo saranno analizzati i risultati di alcuni test.

4.4 OCGR: Simulazione e Risultati

Per analizzare i risultati di OCGR, sotto indicazione di Scott Burleigh, a capo della ricerca DTN presso la NASA, ed autore sia di CGR che di OCGR, nonché di ION, sono stati effettuati dei test sperimentali. L'ambiente di simulazione di ONE è stato predisposto come segue:

- La mappa sulla quale si muovono i nodi è quella di Helsinki, default di ONE.
- Quindici nodi divisi in tre gruppi, uno composto da cinque nodi con velocità di movimento maggiore (possono essere indicate come automobili) e gli altri due gruppi composti da nodi più lenti (pedoni).
- La dimensione dei buffer è stata fissata a 5MB
- Il raggio di trasmissione è stato impostato a 10 metri, mentre la velocità di trasmissione a 2Mbps.
- Ogni simulazione ha una durata di 6h.
- Ogni 40-60 secondi un bundle, con sorgente e destinazione scelte in modo casuale, viene generato ed inserito nella rete. Il numero di bundle generati al termine della simulazione è 436.

I file di configurazione di questo scenario sono presentati in Appendice 2. Sono stati svolte successivamente due serie di simulazioni: una con bundle da 100k ed una con bundle da 50k. Per ogni serie sono stati confrontati tre algoritmi di routing opportunistici: Epidemic Routing, PRoPHETv2 ed OCGR. In figura 1 sono mostrate le probabilità di consegna per i tre algoritmi e nelle tabelle 1 e 2 sono presenti tutte le statistiche ottenute.

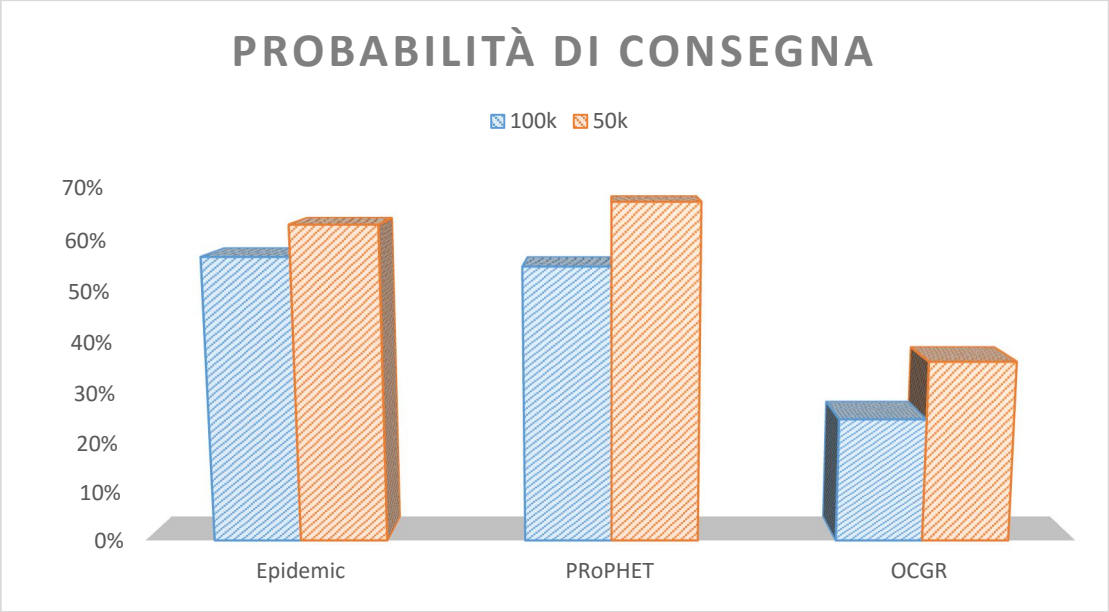

Figura 1Confronto probabilità di consegna

Tabella 1Risultati simulazione 100k

	Epidemic	PRoPHET	OCGR
Bundles started	1660	1387	1133
Bundles relayed	1526	1275	1045
Bundles aborted	134	112	88
Bundles dropped	1105	885	723
Bundles delivered	247	239	108
Delivery probability	0.5665	0.5482	0.2477
Overhead ratio	5.1781	4.3347	8.6759
Average latency (s)	4712.7081	4297.5795	6926.0769
Median latency (s)	3667.2000	3533.8000	6051.9000
Average hop count	1.6842	1.5523	2.0278

Tabella 2Risultati simulazione 50k

	Epidemic	PRoPHET	OCGR
Bundles started	2826	2094	1621
Bundles relayed	2748	2039	1578
Bundles aborted	78	55	43
Bundles dropped	1777	1127	673
Bundles delivered	274	293	158
Delivery probability	0.6284	0.6720	0.3624
Overhead ratio	9.0292	5.9590	8.9873
Average latency (s)	4585.0380	4818.0918	6736.4468
Median latency (s)	3675.5000	4285.9000	6051.4000
Average hop count	1.8905	1.7713	2.1266

In OCGR i bundle che vengono trasmessi sono in numero notevolmente inferiore a PRoPHET ed Epidemic, per via delle decisioni prese dall'algoritmo. Tuttavia la probabilità di consegna rimane bassa e di conseguenza l'overhead è elevato. Il sospetto è quello che nella fase iniziale della simulazione, in cui la storia dei contatti è ancora breve, avvengano delle decisioni di routing poco ottimali. Abbiamo provato di conseguenza a collezionare le

statistiche solo dopo quattro o cinque ore di simulazione, sulle sei totali. Un confronto tra le probabilità di consegna è presentato in Figura 2 mentre i risultati sono riportati in Tabella 3.

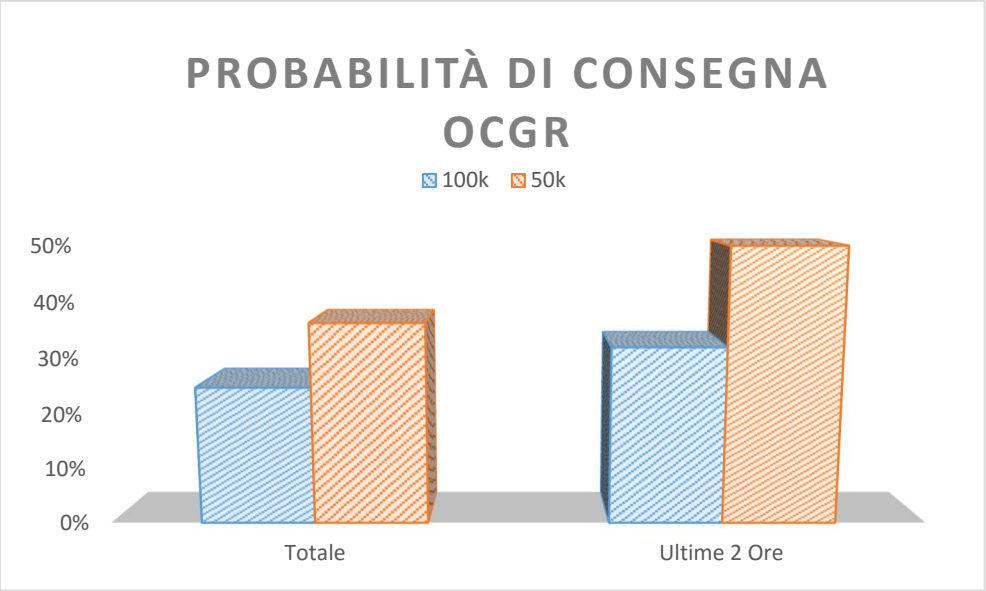

Figura 2 Confronto probabilità di consegna tra simulazione di 6 ore e dati collezionati solo nelle ultime 2 ore

Tabella 3Dati collezionati a simulazione iniziata

OCGR	100k dopo 4h	100k dopo 5h	50k dopo 4h	50k dopo 5h
B. started	596	303	862	424
B. relayed	553	280	841	414
B. aborted	43	23	21	10
B. dropped	665	405	673	624
B. delivered	47	19	73	29
Delivery prob.	0.3197	0.2603	0.4966	0.3973
Overhead ratio	10.7660	13.7368	10.5205	13.2759

I risultati ottenuti collezionando i dati dopo cinque ore di simulazione su sei potrebbero essere poco espressivi; considerando infatti che la latenza media di OCGR è molto superiore ad un'ora, molti dei pacchetti creati in questo lasso di tempo non hanno la possibilità di arrivare a destinazione. I risultati collezionati dopo 4 ore di simulazione mostrano invece un miglioramento sulla probabilità di consegna mentre overhead ancora molto alto. È quindi potenzialmente attendibile la teoria secondo la quale le decisioni prese in fase iniziale sono compromettenti in termini statistici.

I risultati in Tabella 4 e Tabella 5 sono relativi a simulazioni con OCGR della durata di nove e dodici ore. Vogliamo analizzare come si comporta l'algoritmo in simulazioni di durata maggiore. Per semplicità sono riportati solo bundle creati, consegnati e overhead.

Tabella 4Risultati 100k simulazioni 9 e 12 ore

Bundle 100k	OCGR 9h	OCGR 12h
Bundle created	654	870
Bundle delivered	168	182
Delivery probability	0.2569	0.2092
Overhead ratio	8.0119	7.6703

Tabella 5Risultati 50k simulazione 9 e 12 ore

Bundle 50k	OCGR 9h	OCGR 12h
Bundle created	654	870
Bundle delivered	235	255
Delivery probability	0.3593	0.2931
Overhead ratio	8.1149	7.7961

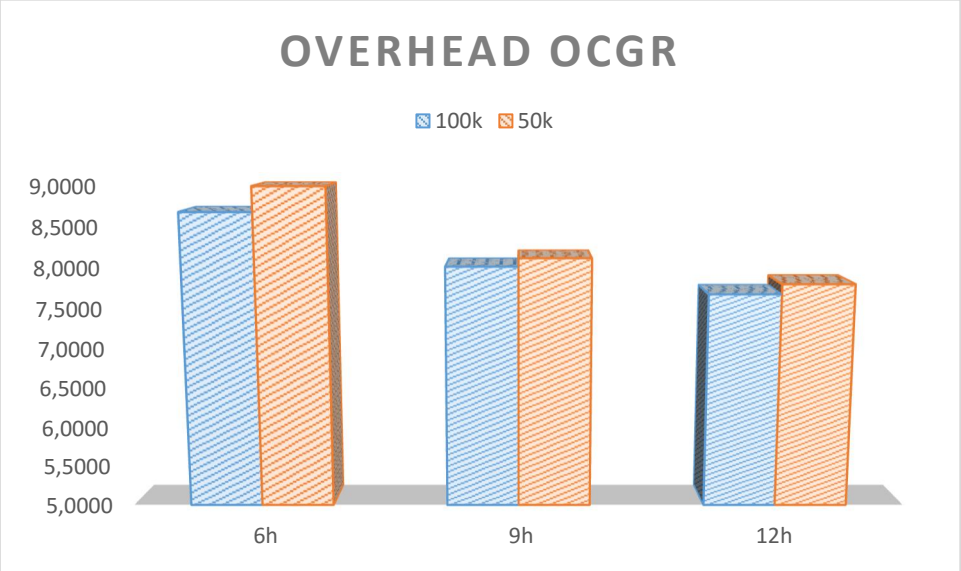

Figura 3Confronto overhead tra simulazioni di durata diversa

Possiamo osservare che più la durata della simulazione aumenta più l'overhead tende a diminuire (Figura 3). Con l'overhead diminuisce drasticamente anche la probabilità di

consegna facendo pensare che il problema potrebbe dipendere dai parametri di OCGR che possiamo variare. Una soluzione proposta e che probabilmente verrà implementata nel breve periodo potrebbe essere quella di far variare la base confidence di un contatto in modo inversamente proporzionale al numero dei contatti precedenti con quel nodo. In questo modo le previsioni saranno più ottimistiche per nodi che hanno una storia dei contatti breve, rendendo il comportamento dell'algoritmo più simile ad epidemic, mentre per nodi che hanno una storia dei contatti più lunga la base confidence sarà definita più bassa. Una volta raggiunti livelli di probabilità di consegna maggiori si dovrà, eventualmente, lavorare sulla riduzione dell'overhead aggiustando la soglia di invio in singola copia o replicazione del bundle.

5 Conclusioni

In questa tesi è stato inizialmente analizzato l'algoritmo Contact Graph Routing, contenuto all'interno della distribuzione ION sviluppata da NASA JPL, che al momento garantisce la miglior probabilità di consegna con minor overhead, in ambienti puramente deterministici. Visti i risultati ottimali di CGR la NASA si è posta l'obiettivo di utilizzare lo stesso algoritmo di calcolo delle rotte in un ambiente opportunistico, in modo da unificare gli approcci. L'Opportunistic Contact Graph Routing, estensione di CGR nata da poco ed ancora in una primissima fase di sviluppo, è l'algoritmo proposto per permettere ad ION di operare all'interno di entrambi gli ambienti. Sotto indicazione di Scott Burleigh, a capo della ricerca DTN presso la NASA, ed autore sia di CGR che di OCGR, nonché di ION, durante lo svolgimento di questa tesi sono stati svolti dei test di valutazione comparativa su di una versione preliminare di OCGR. I risultati ottenuti, mostrati nella tesi, non sono ancora all'altezza dei più performanti fra gli altri algoritmi opportunistici, ma essendo i primi risultati sperimentali essi saranno sicuramente utilizzati in un prossimo futuro per introdurre modifiche e miglioramenti dell'algoritmo. Esistono ancora un'ampia gamma di cambiamenti che possono essere introdotti e valori che possono essere cambiati in modo dinamico, per osservare il comportamento e migliorarlo. Abbiamo notato infatti come i dati cambiano notevolmente anche all'interno di una sola simulazione.

Anche se i risultati sperimentali non sono ancora convincenti, crediamo che l'algoritmo abbia un grande potenziale. Il progetto è molto ambizioso, infatti, se OCGR con gli opportuni miglioramenti si rivelerà efficace negli ambienti opportunistici, sarà possibile utilizzare in un qualsiasi ambiente DTN lo stesso algoritmo di routing, capace di operare sia in ambienti deterministici, sia in ambienti opportunistici, unificando contatti schedulati, scoperti, predetti e continui.

Bibliografia

[Araniti_2015] G. Araniti, N. Bezirgiannidis, E. Birrane, I. Bisio, S. Burleigh, C. Caini, M. Feldmann, M. Marchese, J. Segui, K. Suzuki. “Contact Graph Routing in DTN Space Networks: Overview, Enhancements and Performance”, IEEE Commun. Mag., Vol.53, No.3, pp.38-46, March 2015.

[Balasubramanian_2007] Balasubramanian, Aruna, Brian Levine, and Arun Venkataramani. "DTN routing as a resource allocation problem." ACM SIGCOMM Computer Communication Review 37.4 (2007): 373-384.

[Bezirgiannidis_2014] Bezirgiannidis, N.; Caini, C.; Padalino Montenero, D.D.; Ruggieri, M.; Tsaoussidis, V. "Contact Graph Routing enhancements for delay tolerant space communications", Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop (ASMS/SPSC), 2014 7th, On page(s): 17 – 23.

[Burgess_2006] Burgess, John, et al. "MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks." INFOCOM. Vol. 6. 2006.

[Grasic_2010] Grasic, Samo, et al. "The evolution of a DTN routing protocol-PRoPHETv2."Proceeding. ACM, 2011.

[ION] https://sourceforge.net/projects/ion-dtn/

[Jain_2004] S. Jain, K. Fall, and R. Patra, “Routing in a delay tolerant network,” in *Proc.* ACM SIGCOMM Portland, Aug./Sep. 2004, pp.145-157

[Messina_2016] Jako Jo Messina, “Inclusion of Contact Graph Routing in “The ONE” DTN Simulator”, 2016.

[RFC4838] https://tools.ietf.org/html/rfc4838

[RFC5050] K. Scott, S. Burleigh, “Bundle Protocol Specification”, Internet RFC 5050, Nov. 2007. https://tools.ietf.org/html/rfc5050

[RFC5326] https://tools.ietf.org/html/rfc5326

[Rodolfi_2016] Michele Rodolfi, “DTN discovery and routing: from space applications to terrestrial networks”, 2016

[Spyropoulos_2005] Spyropoulos, Thrasyvoulos, Konstantinos Psounis, and Cauligi S. Raghavendra. "Spray and wait: an efficient routing scheme for intermittently connected mobile networks." Proceedings of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking. ACM, 2005.

[Vahdat_2000] Vahdat, Amin, and David Becker. "Epidemic Routing for Partially-Connected Ad Hoc Networks."

Appendix 1: Integration of the "cgr-jni" package into ONE

Nota: questa appendice è scritta in lingua inglese in quanto andrà a far parte di una guida d'uso per l'utilizzo di OCGR e CGR nelle simulazioni di ONE:

This appendix is a brief user guide to integrate the cgr-jni package, either in its "priority" or "opportunistic" version, into ONE. The guide is intended both for the developer that wants to change or extend the code and the final user interested only in running simulations. This guide largely extends a previous document, [Rodolfi_2016][1].

The ONE original package (downloading and modifications)

The original ONE simulator package with a complete manual can be downloaded from *akeranen.github.io/the-one/,* The ONE folder contains all original Java classes, one setting file and a few scripts for compilation and running (one.sh, compile.sh). This guide and the code refers to the latest version available at thesis writing (v1.6.0).

Although we tried hard to keep our code completely separate from the ONE code, in order to distribute our package as an independent external module, the following changes in the ONE code are necessary or optional.

ONE mandatory modifications

The ONE code needs to be changed as follows:

- class core.DTNHost
  - ◦ line 21: initialize nextAddress with 1.
  - ◦ line 105: assign 1 to nextAddress.

  These two modifications are needed because ION cannot handle a node whose ipn number is 0, therefore we need to start to assign the host address to the node from 1.

- file one.sh: append the environment variable $CGR_JNI_CLASSPATH to the -cp parameter of Java invocation, the user must set this variable to allow the JVM to

[1] [Rodolfi_2016] Michele Rodolfi, "DTN discovery and routing: from space applications to terrestrial networks", 2016

find the CGR and OCGR Java classes. We also add “bin” in case the user wants to compile ONE classes with an IDE, like Eclipse.

```
Java Xmx512M cp
bin:target:lib/ECLA.jar:lib/DTNConsoleConnection.jar:$CGR_JNI
_CLASSPATH core.DTNSim $*
```

In this way, whenever we change the location of our Java classes we do not need to change this file again.

ONE optional modifications

The following changes are necessary only if we want to use the OCGR specific message stats report.

Class **report.MessageStatsReport,** add a new method “getMoreInfo and call it. Below is shown how the code must look like starting from line 178 of MessageStatsReport.java, bold lines are the ones that must be added.

```
178     write(statsText);
179     write(getMoreInfo());
180     super.done();
181     }
182     protected String getMoreInfo(){
183             return"";
184     }
```

This method is overridden in our report.OCGRMessageStatsReport class, so it is necessary to add it here, although it returns an empty string.

The cgr-jni package components

The ION integration in ONE, which supports either CGR or OCGR, comes within one packet that contains both the Java classes that extend the ONE framework and the native “C” code that simulates the ION environment. In the same packet there is also a script to run batch simulations and a few setting files.

As said, at present there are two versions of this software: “priority” and “opportunistic”. The former supports CGR with an extension to ONE that allows ONE to deal with priority

(creation and forwarding of messages with 3 priority levels); the latter, supports OCGR. A possible merging is in the “to-do list”.

Java classes that extend the ONE framework

Our priority/opportunistic classes are organized following the Java standard guidelines for packages and classes: each file contains a class and its name is ClassName.java, each file is contained in a folder whose name is the package that contains the class. The root directory of the Java code is the directory cgr-jni-priority|opportunistic/src. Since we want to use our classes in the ONE framework, we needed to use the same packages used by ONE. Packages are directory with the same name. The packages and classes used directly by ONE are, for OCGR version:

- package routing: classes OpportunisticContactGraphRouter and ContactGraphRouter;
- package test: classes OpportunisticContactGraphRouterTest, ContactGraphRouterTest, and TestUtilsForCGR;
- package report: class OCGRMessageStatsReport.

CGR version:

- package routing: classes ContactGraphRouter and PriorityContactGraphRouter;
- package test: classes ContactGraphRouterTest, PriorityContactGraphRouterTest and TestUtilsForCGR;
- package report: class PriorityMessageStatsReport;
- package input: PriorityMessageEventGenerator and PriorityMessageCreateEvent;
- package core: class PriorityMessage.

All other classes manage the JNI (Java Native Interface) interaction with the ION integration native code and are defined within the cgr_jni package.

The native C code

The C source and header files are in the cgr-jni-priority|opportunistic/ion_cgr_jni directory that tries to follow as much as possible the original ION distribution file organization. This folder contains the following sub directories:

- bp folder: contains the libcgr.c source file and all the needed headers, with the same organization as in ION;
- ici folder: contains all the source files of the ICI libraries and the needed headers, with the same organization as in ION;
- jni_interface folder: contains all the source and header files that support the JNI interaction between the ONE framework and the ION adaptation.
- test folder: contains source and header files used for JNI and simulated libraries tests. Not used for simulations.

All the folders mentioned above and their parent contain a Makefile used for the library compilation. The result of the compilation of the native code is the *libcgr_jni.so* shared library, linked at runtime by the Java virtual machine hosting the ONE framework.

Compiling the code

First, the JAVA 8 compiler (or above) is required. In addition, the environment variable $JAVA_HOME needs to be set in order to let the compiler find the JNI header files. It is usually set to /usr/lib/jvm/java-8-oracle/ depending on which Java version is actually installed. If this variable is not set, the compilation fails.

Compilation of ONE and cgr-jni Java classes

Both ONE and cgr-jni Java classes must be compiled using

```
javac -d destination/folder/path class.java
```

However, ONE provides a script, compile.sh, that could be used to compile all the ONE classes together. This script put the class files into the “target” folder, but it can be changed.

Compilation of the native code

The Makefile in the cgr-jni-priority|opportunistic/ion_cgr_jni directory takes care of compilation of the “C” code. The make program should be invoked, from the directory mentioned above, as follows:

```
make ONE_CLASSPATH=<ONE_classpath>[DEBUG=1]
```

The ONE classpath is the root directory of the packages containing the .class files obtained as a result of the previous JAVA compilation of ONE. The variable DEBUG enables CGR debug prints.

Note that the Makefile assumes by default that the Java classes of our cgr_jni package are put by the Java compiler into the bin directory, as Eclipse does. If this is not the case, the classpath of these classes needs to be added to the make command, as follows:

```
make ONE_CLASSPATH=<ONE_classpath:cgr_jni_classpath>
```

Running the simulations

ONE provides two ways to perform simulations, the graphic mode and the batch mode. The former makes use of a graphic user interface that shows the nodes moving in the map (default Helsinki) and allows the user to interact with the simulation with pause, step and fast forward buttons. The latter, allows the user to perform a series of simulations automatically, by changing one parameter (such as the bundle size or the expiration time) each run.

General settings

ONE is a very powerful and flexible simulator; the user can select different movement models, different routing algorithms, different classes of nodes, etc. All these settings are contained in setting files that must be passed as arguments when the simulation is launched. Default settings can be put in *default_settings.txt,* if this file is in the same folder of *one.sh* (provided by ONE) there is no need to pass it as an argument as it is automatically read. Specific settings should be put in other files, below we will use *cgr-jni.txt.* The description of a setting file would be out of scope here. The reader is referred to The ONE documentation.

cgr-jni specific settings

Before running a simulation, the CGR router variant (either Opportunistic or Priority) and usually also the location of the contact plan must be set, by adding the following lines into *cgr-jni.txt*:

```
Group.Router = [Opportunistic|Priority]ContactGraphRouter
ContactGraphRouter.ContactPlan = path/to/contactplan
```

If PriorityContactGraphRouter is set, it is likely that the user wants also to generate messages with different priority. Every PriorityMessageEventGenerator generate messages with a defined priority. If we want all the three types of messages in the simulation, number of events must be at least 3 and for every generator a different priority must be set. This can be done by setting the following lines in *cgr-jni.txt*:

```
Events.nrof = 3
Events{0|1|2}.class = PriorityMessageEventGenerator
Events{0|1|2}.priority = {0|1|2}
```

Running one simulation

For both CGR modes, ONE is started by means of the one.sh script, provided by ONE. Before invoking the script, we need to set the $LD_LIBRARY_PATH and the $CGR_JNI_CLASSPATH environment variables. The first refers to the location of the libcgr_jni.so native library, the second to our Java classes:

```
export LD_LIBRARY_PATH=/path/to/libcgr_jni.so/dir|
export CGR_JNI_CLASSPATH=/path/to/cgr_jni/classes
```

at this point the one.sh script can be executed passing as parameters the settings file and (if needed) the batch mode options.

```
one.sh -b 1 ./cgr-jni.txt
```

In this case we are launching one simulation in batch mode, using cgr-jni.txt as our setting file.

Running batch simulations

In order to simplify the simulation set up and the result analysis, the batch_test.sh utility script has been developed. This script exploits the ability of ONE to read the simulation settings from separate files in a certain order. In fact ONE reads the settings files in the order they are presented to the command line, and for each setting value read, it overrides any previously read setting with the same name. We define mode of the simulation the parameter that we want to change for each run. According to [SATRIA][2] the following three modes are defined:

- Buffer: the nodes buffer size changes.
- Message: the bundle size changes.
- TTL: the bundle time to live changes.

[2] [SATRIA] Deni Yulianti, Satria Mandala, Dewi Naisien, Asri Nagad, Yahaya Coulibaly, “Performace comparison of Epidemic, PRoPHET, Spray and Wait, Binary Spray and Wait, and PRoPHETv2”.

The simulations show the variations of the performance of a routing algorithm upon specific parameter modifications, but also allows to compare the performances of different routing algorithms running the same parameters. For this reason, the batch script allows to easily choose the routing algorithm we want to use in our simulation: in the simulations directory there is a subdirectory for each router we want to use. In the subdirectory there is the router-specific settings file, that basically define the routing class for the simulation. The simulation is thus invoked passing the settings files in this order: global settings, mode settings, router settings. The output of the simulation is saved in the router folder.

Compiling and launching ONE&cgr-jni from Eclipse

A developer could be interested in a time-saving way to do all the steps written above, considering that every time that a change in the java code is made, classes must be recompiled. To this end, we show the steps for a specific IDE, Eclipse.

If ONE and cgr-jni folders are imported as two different projects, they can be linked and work together, without setting the environment variables every time. First we need to import the files as two projects. After that, the cgr-jni-priority|opportunistic/src folder of cgr-jni must be linked to ONE in this way:

1. Right Click on ONE project ->Build Path ->Configure Build Path
2. Click on the Source tab -> Link Source -> Select path/to/cgr-jni-priority|opportunistic/src
3. Expand the new source -> Native library location -> Edit -> Workspace -> Select path/to cgr-jni-priority|opportunistic/ion_cgr_jni

Native library compilation still must be done as before, by using the Makefile in cgr-jni-priority|opportunistic/ion_cgr_jni without setting the cgr_jni_classpath, as Eclipse compiles by default in the "bin" folder.

```
make ONE_CLASSPATH=<ONE_classpath>[DEBUG=1]
```

After the native library is ready, the simulation can be launched. This can be done, as before, by invoking the one.sh script, or directly from Eclipse. In the latter case the simulation is launched in GUI mode using default settings, unless we add the following line: "${string_prompt}" in Run -> Run Configurations -> Java Application -> "your application" -> Arguments ->Program Arguments. In this case, after clicking the run button, a form is

shown where we can add command line arguments like batch simulation (-b) and all the settings files that are needed.

Appendix 2: Examples of configuration files: default settings, Priority CGR and Opportunistic CGR

Nota: questa appendice è scritta in lingua inglese in quanto andrà a far parte di una guida d'uso per l'utilizzo di OCGR e CGR nelle simulazioni di ONE:

In this appendix three ONE configuration files are shown, to help the user unfamiliar with ONE (default_settings_txt) or with the new setting specific to either cgr-jni version (p-cgr_settings.txt and ocgr_settings.txt). It is worth noting that whenever multiple files are given to ONE at start up, new settings are added, while old settings are overridden by new settings with the same name. These files are ready-to-use, default_settings.txt alone or combined with one of the two others.

Default Settings File – default_settings.txt

The default settings file contains general settings. We will present an example, with a few hopefully useful but not exhaustive comments. The user is referred to ONE documentation for further information.

The following four lines contain the base settings for the simulation scenario; note that the end time is given in seconds. If simulateConnections is false, no connections will start during the simulation. UpdateIntervals indicates, in seconds, the gap between an update of the simulator and the next one.

```
Scenario.name = default_scenario
Scenario.simulateConnections = true
Scenario.updateInterval = 0.1
Scenario.endTime = 43200
```

Below two transmit interfaces are created with transmit speed and range (in meters) well defined. The transmit speed is in kBps so 250kBps are actually 2Mbps. The type of an interface is a java class (ONE package connections), but different types will have different settings; here is shown the *SimpleBroadcastInterface.*

```
btInterface.type = SimpleBroadcastInterface
btInterface.transmitSpeed = 250k
```

```
btInterface.transmitRange = 10
highspeedInterface.type = SimpleBroadcastInterface
highspeedInterface.transmitSpeed = 10M
highspeedInterface.transmitRange = 50
```

The following lines define the groups of nodes used in the simulation and their settings. First default settings, valid for all groups unless specifically overwritten are set. Then specific settings, or settings that override default values, are added for each group. Note that speed is in m/s, TTL in minutes and the router name is the name of the Java class implementing the wanted routing algorithm. It is set a transmit interface (btInterface) for all groups. The movement model name is also a java class from ONE's package Movement.

```
Scenario.nrofHostGroups = 3
Group.movementModel = ShortestPathMapBasedMovement
Group.router = EpidemicRouter
Group.bufferSize = 5M
Group.waitTime = 0, 120
Group.nrofInterfaces = 1
Group.interface1 = btInterface
Group.speed = 0.5, 1.5
Group.msgTtl = 300
Group.nrofHosts = 5
Group1.groupID = p
Group2.groupID = c
Group2.okMaps = 1
Group2.speed = 2.7, 13.9
Group3.groupID = w
```

Below a message generator is defined: the range of the interval between two consecutive messages is in seconds. The host range contains the subset of nodes that can act as source and receiver, all others nodes will be relays. In the example, as each group as a cardinality 5, and there are 3 groups, the total number of nodes is 15 and thus coincides with the

source/destination subset. In simple terms, all nodes will generate/receive messages. In ONE every message has an identifier, every identifier starts with a prefix given by the generator.

```
Events.nrof = 1
Events1.class = MessageEventGenerator
Events1.interval = 40,60
Events1.size = 50k
Events1.hosts = 1, 15
Events1.prefix = M
```

In ONE it is possible to set the random generator seed, rngSeed. Different simulation runs will produce exactly the same results starting from the same seed. This is important to assure the reproducibility of results. Then is set the world size and the warmup time. Warmup indicates how many seconds nodes will move through the map without exchanging messages. The map files used are provided in the ONE folder and are automatically read by the movement model.

```
MovementModel.rngSeed = 1
MovementModel.worldSize = 4500, 3400
MovementModel.warmup = 1000
MapBasedMovement.nrofMapFiles = 4
MapBasedMovement.mapFile1 = data/roads.wkt
MapBasedMovement.mapFile2 = data/main_roads.wkt
MapBasedMovement.mapFile3 = data/pedestrian_paths.wkt
MapBasedMovement.mapFile4 = data/shops.wkt
```

Here we define which reports must be created. Report names correspond to class names, as each class generates one report. It is possible to set a “warmup” period (in s) before the actual start of data collection and also the default report directory.

```
Report.nrofReports = 3
Report.warmup = 0
Report.reportDir = reports/
Report.report1 = MessageStatsReport
```

```
Report.report2 = EventLogReport
Report.report3 = DeliveredMessagesReport
```

The following lines contain some optimization settings suggested by ONE developers (see ONE *World* Class for details), and a few GUI settings. These lines were used in the default settings file provided by ONE.

```
Optimization.cellSizeMult = 5
Optimization.randomizeUpdateOrder = true
GUI.UnderlayImage.fileName = data/helsinki_underlay.png
GUI.UnderlayImage.offset = 64, 20
GUI.UnderlayImage.scale = 4.75
GUI.UnderlayImage.rotate = -0.015
GUI.EventLogPanel.nrofEvents = 100
```

CGR and Priority settings file – p-cgr_settings.txt

This file contains setting specific to the priority CGR router, the path to contact plan, and three message generators, one for each priority class (only the 3 DTN cardinal priorities, bulk, normal expedited, are implemented, denoted as 0, 1 and 2). Router must be set as *PriorityContactGraphRouter.* Note that *ContactGraphRouter* is also available if priorities are not needed).

```
Group.router = PriorityContactGraphRouter
ContactGraphRouter.ContactPlanPath = path/to/contact/plan
Report.report1 = PriorityMessageStatsReport
Scenario.name = CGR_Settings
Events.nrof = 3
Events1.class = PriorityMessageEventGenerator
Events1.interval = 1,160
Events1.size = 100k
Events1.hosts = 1,15
Events1.prefix = B
Events1.priority = 0
Events2.class = PriorityMessageEventGenerator
Events2.interval = 1,255
```

```
Events2.size = 100k
Events2.hosts = 1,15
Events2.prefix = N
Events2.priority = 1
Events3.class = PriorityMessageEventGenerator
Events3.interval = 1,300
Events3.size = 100k
Events3.hosts = 1,15
Events3.prefix = E
Events3.priority = 2
```

OCGR settings file – ocgr_settings.txt

This file contains the settings that are specific to OCGR. First router must be set (*OpportunisticContactGraphRouter).* Then, as this routing is still in a testing phase, there are some particular features. If debug is true, information about the current simulation (found routes, bundles sent, etc.…) are shown in the console; if epidemic dropback is set true and a node finds no routes for a bundle, epidemic routing is performed (only for testing).

```
Scenario.name = OCGR_Router
Group.router = OpportunisticContactGraphRouter
OpportunisticContactGraphRouter.epidemicDropBack = false
OpportunisticContactGraphRouter.debug = false
```