ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE IN COMPUTER ENGINEERING

THESIS

in

Foundations of Telecommunications T

SVILUPPO DI NUOVE FUNZIONI PER IL SUPPORTO DELL'ALGORITMO DI ROUTING OCGR NEL SIMULATORE ONE PER RETI DTN DI TIPO OPPORTUNISTICO (DEVELOPMENT OF NEW FEATURES FOR THE SUPPORT OF THE OCGR ROUTING ALGORITHM IN THE ONE SIMULATOR FOR OPPORTUNISTIC DTN NETWORKS)

CANDIDATE:

Simone Pozza

SUPERVISOR:

Prof. Ing. Carlo Caini

Academic Year 2015/2016

Session II

Prefazione

L'ambiente di ricerca di questa tesi è quello del Delay- and Disruption-Tolerant Networking (DTN), un'architettura di rete progettata per far fronte ai problemi che caratterizzano le cosiddette “challenged networks”: tempi di propagazione elevati, un alto tasso di pedita dei pacchetti e connessioni intermittenti. L'origine di questa architettura risiede nella generalizzazione dei requisiti identificati per Inter-Planetary Networking (IPN), una rete composta da sonde, stazioni spaziali e satelliti, ma sono state ampiamente studiate anche applicazioni terrestri come reti militari tattiche, reti di sensori, reti mobili ad-hoc etc.. Nelle comunicazioni nello spazio profondo 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 tutte queste reti, l'impiego dei protocolli della suite TCP/IP risulta inefficace o inattuabile.

Esistono diverse implementazioni dell'architettura DTN: DTN2, IBR-DTN e ION (Interplanetary Overlay Network), sviluppata da NASA/JPL, per applicazioni spaziali. All'interno di ION è presente l'algoritmo di routing detto Contact Graph Routing (CGR), progettato per operare in ambienti con connettività deterministica e una sua estensione per ambienti non deterministici detta Opportunistic Contact Graph Routing (OCGR). Per lo studio degli algoritmi di routing nelle reti DTN la “Helsinki University of Techology” ha sviluppato il simulatore “The ONE”, che implementa diversi modelli di moto, di generazione dei dati, e permette la visualizzazione in tempo reale tramite interfaccia grafica.

L’obiettivo principale di questa tesi è stato quello di combinare in un unico pacchetto i contributi degli studenti dell’Università di Bologna che mi hanno preceduto lavorando sul tema dell’integrazione di CGR in The ONE. Michele Rodolfi e Jako Jo Messina, durante la tesi di laurea Magistrale in Ingegneria Informatica, utilizzando la Java Native Interface (JNI), hanno adattato tutte le librerie C di ION per far funzionare CGR e OCGR all'interno dell'ambiente Java di The ONE. Successivamente, Alessandro Berlati e Federico Fiorini, durante il loro tirocinio presso il Dipartimento di Ingegneria dell’Energia Elettrica e dell’Informazione (DEI), hanno aggiunto il supporto ai messaggi con priorità e l'Overbooking Management per CGR. Il “merge” dei diversi contributi è stato particolarmente complesso a

causa della frammentazione del codice già sviluppato, e dalla mancanza di una documentazione unitaria. Esso è stato realizzato rispettando un importante principio di progettazione: mantenere al minimo le modifiche necessarie al codice originale di ION e The ONE per evitare di dover modificare il nostro codice ogni volta che una nuova versione dei sofware da cui dipende viene rilasciata.

Una volta unificato il codice di partenza, è stata realizzata una nuova classe di routing per aggiungere il supporto ai messaggi con priorità e l'Overbooking Management a OCGR. Infine, con questa tesi è stata prodotta una documentazione unificata di tutto il codice. La descrizione del codice si concentra sulle classi Java con cui un potenziale utente deve interfacciarsi per poter usare il software. Saranno illustrate le funzionalità di tre gruppi di classi Java, indipendenti dal codice C di ION e quattro classi Java di routing che invece necessitano del codice nativo e quindi fanno uso della JNI, fra le quali quella sviluppata ex novo. Da ultimo due appendici descrivono come installare ed utilizzare tutto il software.

Abstract

This thesis deals with Delay- and Disruption-Tolerant Networking (DTN), a network architecture designed to cope with those problems that characterize the so-called "challenged networks": long round-trip-times, high packet loss ratio and link intermittency. The origin of this architecture lies in the generalization of the requirements identified for Inter-Planetary Networking (IPN), a network composed of probes, space stations and satellites, but a few terrestrial applications have been widely studied too: military tactical networking, sparse sensor networks, mobile ad-hoc networks (MANETs) etc.. While contacts between nodes in deep space communications are deterministic (due to the motion of planets and spacecrafts), contacts in terrestrial networks are generally opportunistic (not known a priori). For all these networks, the employment of the protocols of the TCP / IP suite is inadequate or impossible.

The main implementations of the DTN architecture are: DTN2, IBR-DTN and ION (Interplanetary Overlay Network), developed by NASA/JPL, more oriented to space applications. ION includes a routing algorithm, called Contact Graph Routing (CGR), designed to work in deterministic environments and an extension of CGR, called Opportunistic Contact Graph Routing (OCGR), for non-deterministic environments. In order to analyze routing in challenged networks, the “Helsinki University of Technology” (TKK) developed “The ONE” simulator, which is capable of generating nodes following different movement models, allows messages exchange between these nodes and features a graphical user interface.

The main objective of this thesis was to combine in a single package the contributions of several students of the University of Bologna who worked before me on the integration of CGR into The ONE. Michele Rodolfi and Jako Jo Messina, during their master thesis in Computer Engineering, using the Java Native Interface (JNI), have adapted all ION C libraries to allow CGR and OCGR to work inside the Java environment of The ONE. Subsequently, Alessandro Berlati and Federico Fiorini, during their internship at the Department of Energy Electrical and Information Engineering (DEI), have added the support for priority messages and Overbooking Management to CGR. The merge of the different contributions was particularly complex because of the fragmentation of the existing code, and

the lack of a unified documentation. The merge has been carried out in observance to an important design principle: keep to a minimum the changes necessary to the original code of ION and The ONE to avoid having to modify our code whenever new versions of those softwares (on which it depends) is released.

After unifying the starting code, a new routing class was created to add support for priority messages and Overbooking Management to OCGR. Finally, with this thesis it was produced a unified documentation of all code. The description of the code focuses on those Java classes which a potential user must know in order to use the software. The functionalities of three groups of Java classes, independent of the C code of ION will be illustrated along with those of four routing classes that instead require the native code and make use of the JNI, one of which was developed from scratch. Finally two appendices describe how to install and use the software.

Index

1 Introduction

1.1 Delay- and Disruption-Tolerant Networking (DTN)

The Delay- and Disruption-Tolerant Networking architecture has been designed to allow communications in those scenarios where the TCP/IP protocols alone cannot provide satisfactory performance: networks in which one or more of the fundamental assumptions on which the Internet architecture is based are not held. These assumptions are:

- The end-to-end path between source and destination is always available.
- Round trip times (RTTs) are short.
- Channel bandwidth is symmetrical between up and down directions.
- Channel error rate is low.

Networks were at least one of these conditions are not met, are called “challenged”, and are the most suitable environments for a DTN application. At present, the scenario where the DTN networking is most used, and the main reason for its creation, is the Inter Planetary Networking (IPN), where almost all the assumptions on which the Internet architecture is based are not valid: the RTTs are usually much longer than the terrestrial ones, the packet loss percentage is one order higher than on Earth, or there is a notable bandwidth asymmetry between up and down directions. Other examples of challenged networks are: Mobile Ad-Hoc Networks (MANETs), emergency networks, sensor networks, tactical military networks and underwater networks.

1.1.1 Bundle Protocol (BP)

The DTN architecture introduces an overlay protocol to the normal communication stack, between the application layer and transport or lower layers: the Bundle Protocol (BP). In such an overlay, delays and disruptions can be handled at each DTN “hop” in a path between a sender and a destination. Nodes on the path can also provide the storage necessary for application data before forwarding that to the next node on the path. Thus the main benefit of protocols implementing the DTN architecture is that they do not require the contemporaneous end-to-end connectivity that TCP and other standard Internet transport protocols require in order to reliably transfer application data. The basic unit of data in the BP is a “bundle” which

is a message that carries application layer protocol data units (APDU), sender and destination names, and any additional data required for end-to-end delivery. The BP can interface with different lower layer (usually transport) protocols through “Convergence Layer Adapters” (CLAs) as shown in Figure 1. With the BP, each DTN node on a path may use whatever CLA is best suited for the next forwarding.

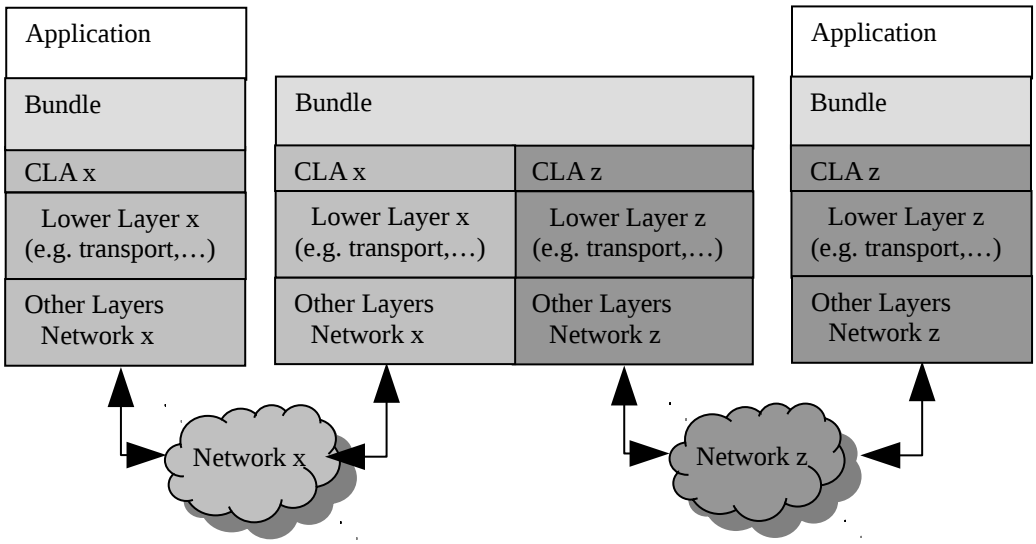

Figure 1: DTN architecture and protocol stack [C. Caini et al., 2011]

1.1.2 Store-and-forward paradigm

In standard networks, which assume continuous connectivity and short delays, routers perform non persistent (short-term) storage and information is persistently stored only at end nodes. This is because, dealing with reliable transmission, information is supposed to be easily retrieved directly from the source. In a DTN network the presence of a continuous end-to-end path between source and destination cannot be taken for granted, as links between consecutive DTN nodes can be intermittent. Therefore, in DTN networks it becomes necessary to store information persistently (long-term) at intermediate DTN nodes waiting for the availability of the next hop: the technique just described is called *store and forward.* A bundle, once received by a node, can be stored for a long period, until the next path becomes available. This mechanism makes DTN much more robust against disruptions, disconnections, and temporary node failures.

On the other hand, in-network bundle storage raises storage congestion issues that still need to

be addressed. While the BP includes some “expiry” controls, so that expired bundles are eventually deleted from in-network storage, there may still be cases where a node does not have sufficient storage available and work on generic and scalable ways to handle this is still ongoing in the DTN community.

1.1.3 Cardinal priorities

The DTN architecture provides three priority levels for bundle delivery: low, medium, and high. These priority levels imply a form of scheduling within DTN node queues: bulk, which concerns lowest priority bundles; normal; and expedited, whose bundles should be shipped prior to bundles of the other classes [C. Caini et al., 2011].

1.2 Routing algorithms in DTN

Challenged networks present several problems that prevent the use of Internet routing algorithms based on an up-to-date comprehensive knowledge of network topology such as: link intermittency, network partitioning, limited storage in the intermediate nodes possibly and long delays in the exchange of routing information among nodes in the network. Many routing algorithms for DTN have been proposed, investigated in simulation, and in some cases tested in operation, but the field remains generally open: no single routing system has emerged as the consensus choice of the DTN research and deployment community, in part because DTN networks are highly heterogeneous. Space networks are characterized by intermittent scheduled connectivity: opportunities for of transmission between nodes are known in advance, and paths are thus deterministic. By contrast, most terrestrial DTNs are characterized by random intermittent connectivity, as contacts typically arise from casual encounters [S. Burleigh et al., 2016]. Given this assumption, totally different routing algorithm were studied and developed, and they were split into two families, considering how much the algorithm knew about the status of the network and its configuration information: *opportunistic algorithms*, where those information were not always updated, and *deterministic algorithms*, which are assumed to have a perfect knowledge of the network. Contact graph routing is possibly the sole DTN routing algorithm designed to cope with deterministic scheduled connectivity, while for opportunistic networks there are many

proposed approaches that usually employ a *flooding-based* strategy, replicating the messages a number of times dependent from their algorithm:

- Epidemic routing [Vahdat and Becker, 2000], which is the easiest routing algorithm, which allows the nodes transmitting bundles every time they encounter a node not carrying a copy of that bundle. Of course it is highly reliable, but storage consuming too, because does not care of avoiding replication at all.
- ProPHET [A. Lindgren et al., 2012] uses the non-randomness of contacts, replicating bundles only if delivery probability is higher than a certain value. The second version, ProPHET v2 is the latest and optimized version.
- Spray – and – Wait [T. Spyropoulos et al., 2005] replicates ("sprays") a limited number of copies in the network and waits until one of the node which received a copy contacts the destination.
- MaxProp [J. Burgess et al., 2006] is based on a priority definition based on likelihoods according to historical data and other complementary mechanism.
- RAPID [A. Balasubramanian et al., 2007 and 2010] also evaluated on the same DTN bus network, uses a random variable that represents the contact between two DTN nodes and replicates bundles in decreasing order of their marginal utility at each transfer opportunity. Utility is measured for three separate metrics aimed at minimizing either the average delivery delay, or the missed bundle deadline beyond which the bundle is no longer useful, or the maximum delivery delay.

1.2.1 Contact graph routing (CGR)

CGR is a dynamic algorithm that computes routes based on the "contact plan," a time-ordered list of scheduled, anticipated changes in the topology of the DTN network. The entries in this list are termed "contacts"; each one is an assertion that a transmission from node X to node Y at nominal data rate R will begin at time T1 and will end at time T2. Note that this assertion implicitly also defines the "volume" (or "capacity") of the contact, which is the maximum amount of data that can be transferred during the contact, given by the product of contact length (T2 – T1) and nominal transmission rate R. Each node uses the contacts in the contact plan to build a "routing table" data structure. A routing table is a list of "route lists," one route

list for every possible destination node in the network. Each route in the route list for node D identifies a path to destination node D, from the local node, that begins with transmission to one of the local node's neighbors in the network, the initial receiving node for the route, termed the route's "entry node." The route list entry for each neighbor contains the best route that begins with transmission to that neighbor. It's important to note that the routes in the route list do not need to be continuous. Each segment of the path is an opportunity to send data from node X to node Y; once a bundle has reached node Y it may well reside in storage at node Y for some length of time, awaiting the start of the opportunity to be forwarded from node Y to node Z, and so on [G. Araniti et al., 2015]. Each route is also associated to a "forfeit time", i.e. the latest time by which the bundle must be forwarded to the route's entry node in order to have any chance of traversing the route itself.

CGR can be successfully applied not only to an Interplanetary Internet, but also to all space flight communication operations, since the communication routes between any pair of "bundle agents" can be inferred from the mission operators' detailed plans rather than discovered via dialogue.

At the time of writing this thesis, two enhacements have been implemented to cope with residual issues: earliest transmission opportunity (ETO) and overbooking management. Only the latter will be described here while further informations about ETO and other proposed enhancements can be found at [G. Araniti et al., 2015].

A bundle may be assigned to a contact that is already fully subscribed, provided that the bundle's priority is higher than that of some of the bundles currently assigned to that contact. The contact oversubscription that derives from this policy is informally called contact "overbooking". In an overbooking example of a future contact, some low priority bundles put in the queue to a proximate node will miss their contact, to accommodate higher priority bundles. This situation is tackled by CGR a posteriori, by re-forwarding the "bumped" bundles once their forfeit time expires (usually at the overbooked contact's end-time). This handling, although robust, is not efficient. By contrast, overbooking management acts a priori, by re-forwarding as soon as possible any bundles that are destined to miss the contact, i.e. immediately after forwarding the higher priority bundle that has caused the oversubscription.

1.2.2 Opportunistic contact graph routing (OCGR)

Opportunistic Contact Graph Routing is an extension to CGR aimed at enlarging its applicability from deterministic space networks to opportunistic terrestrial networks. To extend CGR in support of opportunistic routing, the contact plan has been extended in two ways:

- Non-scheduled contacts may be automatically discovered in real time, offering immediate connectivity to newly discovered neighboring nodes. When these discovered contacts end, their start and stop times and volumes are recorded in a contact log.
- *Confidence* in both scheduled and discovered contacts is always 1, but the contact plan may also include predicted contacts in which we have much less confidence.

Additionally, for each outbound bundle is calculated the confidence that the forwarding activities performed so far will result in delivery of the bundle at its destination prior to bundle expiration. This bundle delivery confidence value is initialized to 0. For any newly discovered contact, the communicating nodes exchange all contact log entries, they then discard all previously computed predicted contacts and use the updated contact history to compute new predicted contacts. The result is a contact plan that can be used for contact graph routing in the usual way, except that our confidence in the resulting forwarding decisions is less than total.

1.3 The ONE simulator

“The ONE” DTN is a Java based simulator that offers the opportunity to test and compare these routing algorithms and, although routing schemes that are suitable for interplanetary and deep space scenarios are not included, it is possible to add new routing protocols just by extending the *ActiveRouter* class, which contains some basic methods and functionalities that almost every kind of router owns. The simulator also offers several (extendable) reports which allow a good analysis of routing performances.

OCGR has been implemented in the current version of the ION DTN package, and that implementation has been integrated into The ONE DTN simulator. The native ION CGR software (including the OCGR extension), written in C, has been imported directly into the Java-based simulator, without modification, by means of Java Native Interface (JNI) classes.

CGR is not simulated in ONE, it is executed. Later in this work will be shown how to run simulations of multiple classes of routers.

2 Extensions indipendent from the native C code

The following classes offer functionalites indipendent from the C native code and therefore can also be used outside the context of the CGR integration into The ONE. At the beginning of each paragraph the Java classes visible to the user are listed as “Main classes”.

2.1 Creation of an ION contact plan

Main Java classes:

- *CPEventLogReport* (Package *report*; class that extends the ONE *EventLogReport* class to include the transmission speed);
- *ContactPlanCreator* (executable; converts a *CPEventLogReport* into an ION contact plan).

2.1.1 Introduction

The contact plan creator consists of a standalone Java application aiming at translating a log file into an ION contact plan. This could be useful in a variety of situations. For example, the movement models of ONE could be exploited to generate a corresponding contact plan to be used outside ONE. To this end, contacts could be generated in ONE by setting a given number of nodes and selecting a suitable movement model; then the log could be converted into a contact plan, ready to be used outside ONE, e.g. in an ION testbed consisting of the same number of nodes. However, the primary application is to overcome the CGR need to have the a priori knowledge of the contact plan; to obtain it before the CGR simulation, a simple trick is to run the simulation twice, starting from the same seed. In the first run a routing algorithm different from CGR, as Epidemic, is used, then the log is converted into a contact plan; this contact plan is used in the second simulation where the CGR is used. By using the same seed, contacts are opened and closed as before, thus they exactly correspond to the contact plan passed to CGR. Note that in this scenario the contact plan is not used to open and close contacts, but only by CGR to make routing decisions. These classes have been developed by Jako Jo Messina; for further information see [J. J. Messina, 2015].

2.1.2 Description

The contact plan creation involves three classes: *CPEventLogReport, ContactPlanCreator and ContactPlanLine*. The latter is just a support class representing a line of the contact plan while *ContactPlanCreator* creates the contact plan starting from a *CPEventLogReport.* It is of crucial importance that the simulation run using CGR, and consequently the contact plan, has the same configuration and seed of the simulation performed to obtain the contact plan, otherwise contacts will open and close to different times. First of all, we need to enter the full path of the input file (the report file generated by ONE at the end of the simulation) and of the output (the ION contact plan). The application will read the whole input file, line by line, selecting only the connection events, and, in particular, extrapolating from the "UP" events the start time of the contact, the two nodes involved, and the transmission speed; then, stop time will be added when the corresponding "DOWN" event is read. For every contact, the application will write in the contact plan file three lines: first, the range line, with the smallest node number first (this to let ION interpret the range as bidirectional), and then the two contact lines, one for each direction (again, this is requested by ION syntax). Note that the transmission speed in ION is given in B/s and that the range of terrestrial contacts (i.e. the one way propagation delay) is assumed 1s. An extract from a contact plan created is given below.

```
a range     +542    +568    5   8   1
a contact   +542    +568    5   8   16000
a contact   +542    +568    8   5   16000
a range     +891    +911    10  8   1
a contact   +891    +911    10  8   16000
a contact   +891    +911    8   10  16000
a range     +1654   +1729   6   1   1
a contact   +1654   +1729   6   1   16000
a contact   +1654   +1729   1   6   16000
a range     +1827   +1882   8   4   1
a contact   +1827   +1882   8   4   16000
a contact   +1827   +1882   4   8   16000
a range     +2273   +2531   3   5   1
a contact   +2273   +2531   3   5   16000
a contact   +2273   +2531   5   3   16000
a range     +2893   +2981   2   9   1
a contact   +2893   +2981   2   9   16000
a contact   +2893   +2981   9   2   16000
a range     +3333   +3397   4   3   1
```

Figure 2: ION contact plan format

2.2 Use of an ION contact plan to open and close contacts in The ONE simulator

Main Java classes:

- *ExtendedExternalEventsQueue* (Package *input*; reads the external contact plan);
- *ExtendedEventLogReport* (Package *report*; creates a log of contacts created by the external contact plan).

2.2.1 Introduction

As ONE was designed for DTN opportunistic networks, all contacts derive from the motion of nodes. To extend its applicability to DTN deterministic networks, we have added the possibility of opening and closing contacts on the basis of an external contact plan, following the ION format. This possibility is extremely useful for both CGR and non CGR routers. To this end, we have exploited the possibility of generating events on the basis of external events. Note that the transmission rates inserted in the contact plan cannot exceed the maximum transmission rate associated to the network interface in ONE configuration files. Moreover, it is paramount to set the Scenario.simulateConnections to “false”, as connections are no more to be generated by the simulator. These classes have been developed by Federico Fiorini. For further information see [F. Fiorini, 2016].

2.2.2 The CPEventsReader e CPConnectionEvent classes

First, the class *CPExternalEventsReader* has been created to read the external contact plan: it implements the interface *ExternalEventsReader* and redefines two methods:

- *readEvents()*: for every contact read in the file it generates the corresponding contact event and puts it in a queue of *ExternalEvent*, given in output. The range instructions are formally checked, and then skipped, as propagation delays are negligible in terrestrial communications and not implemented in ONE.
- *close()*: it closes the *reader* associated to the external contact plan; it must be called after reading the file to avoid wasting memory resources.

Then the class *CPConnectionEvent* has been created by extending the class *ConnectionEvent*. The extension adds the parameter speed (in B/s) and the get method to obtain this speed. The list of external events generated by the class CPEventsReader is just a list of CPConnectionEvent items.

2.2.3 The ExtendedExternalEventsQueue and ExtendedEventQueueHandler classes

Other two classes have been extended to allow the use of an external contact plan.

ExtendedExternalEventsQueue is an extension of *ExternalEventsQueue* where the two following methods are redefined

- *init()*: it creates an association between an instance of this class and one reading class; it has been modified to allow reading from CPEventsReader. This method calls the next one. For the sake of simplicity, it is assumed that the input file cannot be binary, thus omitting one if instruction.
- *readEvents()*:called inside *init()*, it reads the events by means of the associated reader and adds them to the queue of the external events, defined as a field of the class.

The class *ExtendedEventQueueHandler* extends *EventQueueHandler* to manage an event queue of class *ExtendedExternalEventsQueue*.

The constructor *ExtendedEventQueueHandler()*, creates an object, a list of *EventQueue*, which then is filled as specified in the configuration file: if only the setting Events*.class is present, it realizes that the event generator is internal, otherwise, if Events*.filePath is specified it realizes that the events are external and provided in the file (the file must be inside the ONE directory to avoid confusion). In this latter case, it instances one *ExtendedExternalEventsQueue* object, which reads the events and put them into a queue.

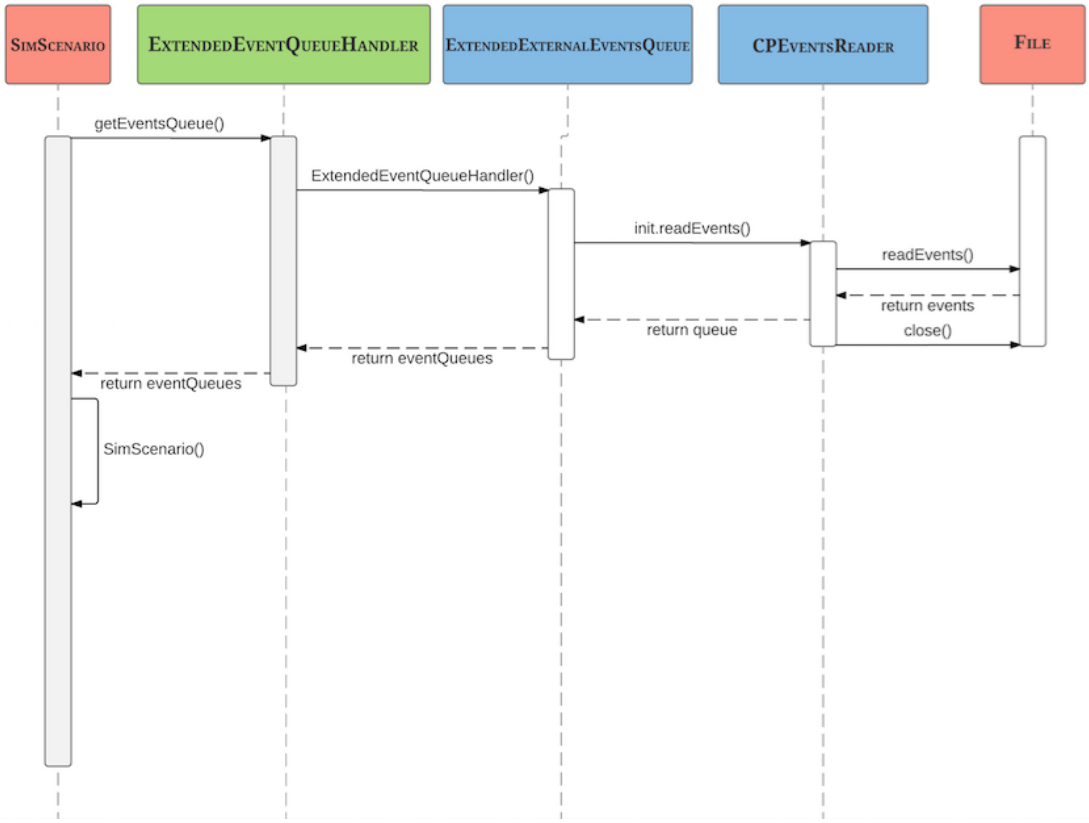

Figure 3: diagram sequence of the methods called by the classes described above.

2.2.4 ExtendedEventLogReport

The class *ExtendedEventLogReport* extends the *EventLogReport* (package *report*) to include the transmission rate specified in the external contact plan. To this end the method *hostsConnected()*, which produce a line in the log for each connection event has been redefined to include the transmission rate.

It is worth noting that the transmission rate inserted in the contact plan cannot exceed the maximum transmission rate associated to the network interface, as specified in the btInterface.transmitSpeed parameter of the configuration file.

2.3 Adding priorities to ONE

Main Java classes:

- *PriorityMessageEventGenerator (*Package *input;* it generates messages with priorities*);*
- *PriorityEpidemicRouter* (Package *routing;* Epidemic router with support of priorities);
- *PriorityMessageStatsReport* (Package *report;* it creates a log with the statistics of bulk, normal and expedited messages).

2.3.1 Introduction

Although three levels of (“cardinal”) priorities (bulk, normal, expedited) are defined in both the DTN architecture [RFC4838] and Bundle Protocol (BP) specifications [RFC5050], the ONE simulator and the routers included in it do not consider any kind of differentiated traffic. By contrast, a finer granularity for the extended class (255 “ordinal” priorities) has been proposed in ECOS BP extensions [Burleigh, “Bundle Protocol Extended Class Of Service (ECOS)” IRTF draft-irtf-dtnrg-ecos-05, work in progress] and implemented in ION. Both cardinal and ordinal priorities are taken into full account by CGR. To fill the gap, we have decided to include priorities in ONE; as usual we have preferred to introduce a few extended classes, instead of modifying the old ones, to maintain full compatibility with all routers already present in ONE. Only cardinal priorities have been introduced for the sake of simplicity. Messages with priorities can be handled by an extended version of the

EpidemicRouter and, if CGR extensions are included, also by priority aware versions of CGR and OCGR (PCGR and POCGR, respectively). These classes have been developed by Federico Fiorini; for further information see [F. Fiorini, 2016].

2.3.2 PriorityMessage e PriorityMessageEventGenerator classes

The *Message* class has been extended to *PriorityMessage*, which just includes the priority integer field, which can assume the values [0,2] corresponding to bulk, normal and expedited cardinal priorities. Moreover, the methods *replicate()* e *copyFrom()*, have also been modified to include the new priority field in the replicated messages.

Once priorities were added to the *Message* class, it was necessary to create a message generator able to generate messages with priorities. To this end the new class *PriorityMessageEventGenerator* extends the *MessageEventGenerator* class. Each instance of this class will have just one priority, according to what is specified in the following settings of the configuration file:

```
Events*.class = PriorityMessageEventGenerator
Events*.priority = 0 | 1 | 2
```

Inside the method *nextEvent()* one *PriorityMessageCreateEvent* object is instanced , which extends the *MessageCreateEvent* class, which creates a message of the wanted priority thanks to the function *createNewMessage()*.

Note that other message generators, with a more restricted application scope, are present in ONE, for example to generate bursts. They could be easily extended to include priorities, too, if deemed necessary.

2.3.3 PriorityEpidemicRouter

As said, routers in The ONE do not enforce priorities. To show that their usefulness is not limited to CGR, and also to check their correct processing, a new class, called *PriorityEpidemicRouter,* has been created. It sends the messages in its transmission buffers first on the basis of their priority, and then, for each priority, on a FIFO logic, i.e. on the basis of their arrival time. A few implementation details are given below.

The *PriorityEpidemicRouter* extends the *ActiveRouter* class; it adds the boolean variable *removing*, which denotes if the next message in the queue is to be removed or sent, and the two methods *get* e *set* to either read or set the value.

The following methods have been redefined:

- *sortByQueueMode()*: it orders the queue on the basis of priorities and then as FIFO. The order is increasing when messages are to be sent, decreasing when to be removed.
- *update():* this method is called every time an update of the simulation must be performed; it calls the method t*ryAllMessagesToAllConnections()* of *ActiveRouter*, which gives the list of messages, allows ordering them according to the criteria just said and finally to send them to connections that are active at that instant.
- *getNextMessageToRemove()*: this method is called when a new message arrives and the buffer is full; the message to drop is the one with the lowest priority and higher waiting time.
- *replicate()*: to generate one *PriorityEpidemicRouter* instead of a plain *EpidemicRouter*.

2.3.4 PriorityMessageStatsReport e testing

The *PriorityMessageStatsReport* (package *report*) extends the class *MessageStatsReport*, to add priorities in statistical reports. This is essential to check the expected behavior, i.e. if higher priority messages obtain higher chances of getting delivered and shorter delivery times.

3 Routing classes

Here will be described classes which use the ION native libraries thanks to the Java Native Interface (JNI): additional code which acts as a bridge between Java and C. The C files and the Java classes which constitutes the JNI were developed by Michele Rodolfi and are described in depth in his thesis [M. Rodolfi, 2015]. All the classes described in this chapter belong to the *routing* package.

3.1 ContactGraphRouter (CGR without priorities)

3.1.1 Outduct and ContactGraphRouter classes

Before starting to write the CGR class it was fundamental creating the *Outduct* object. In ION outducts are the queues towards proximate nodes where bundles are put after having been forwarded by CGR, waiting to be transmitted; to support CGR it was necessary to build an equivalent in ONE. To this end, outducts have been abstracted with a class containing the *DTNHost* indicating the local host, and a queue of messages representing the outduct itself. Then, a few accessory methods and functions for inserting and removing messages from the queue have been added. In every *ContactGraphRouter* all the outducts are collected inside an array. Moreover, every router contains one more outduct, the limbo, hosting the messages whose intended route has expired, or no known route to destination, with respective methods for adding e removing messages from it. Note that this policy differs from the current CGR implementation in ION, where bundles with no known route to destination are purged to save memory space. This drastic policy has not been implemented in ONE, as considered likely too drastic for terrestrial environments, although justified in space communications. Going on in the description of the CGR class, after the constructor and the copy constructor there is the *init()* method. It initializes the router, by calling *super()* thus invoking the same method of the *AbstractRouter* class, and the CGR libraries, by means of *initCGR()* and also reads the contact plan from the file path provided in the ONE setting file.

The method *checkExpiredRoutes()* searches all outducts looking for those messages whose intended route has expired and thus need to be re-forwarded by CGR. First, this method creates a list of all messages whose forfeit time has expired; then each message of this list is

moved from the outduct to the limbo, where is re-forwarded by *cgrForward()*; if a new plausible route is found the bundle is eventually put into the corresponding proximate node outduct. The *update()* method, which is responsible for sending and receiving messages from other routers put messages into the limbo too, then it invokes the *tryRouteForMessageIntoLimbo()* method, which tries to find a route for all messages in the limbo; after that, the *super()* method is invoked, thus continuing with the start of communication for nodes that are connected. The *addToMessages()* method adds a message into the router message queue and puts it into limbo, then calls the father inherited method and invokes *cgrForward()* for forwarding the message. The other methods of the class are overridden methods from the *ActiveRouter* class, which substantially performs as the parent methods except for *removeFromOutducts()*, which removes a message from every outduct. Finally, a few interface methods, such as *initCGR()* and *finalizeCGR()*, call methods directly from the native *libcgr.c* file, to initialize and finalize the router. The finalizing operation is important to deallocate the memory used by CGR. In the *IONInterface* class are implemented the static methods accessed form the Java Native Interface, i.e. the methods used by the native C libraries.

3.1.2 Test classes

The ONE offers a suite of classes useful for testing routing algorithms, movement models and other features. In particular, the *TestUtils()* class and the *AbstractRouter* class can be used to test routing algorithms. To test CGR we have created the *ContactGraphRouterTest* class, which inherits the *AbstractRouter*'s methods and adds to them the *setup()* method and the test functions. Moreover, it was created also the *TestUtilsForCGR* class, which extends *TestUtils* . See [J. J. Messina, 2015] for further information on testing.

3.2 OpportunisticContactGraphRouter (OCGR without priorities)

3.2.1 Introduction

To integrate the Opportunistic Contact Graph Routing algorithm into The ONE what was done for the CGR integration was extended. On the C side, new entry points had to be provided in order to support both the information exchange between nodes that discover each

others and the contact prediction. On the Java side, we created the *OpportunisticContactGraphRouter* class that extends the *ContactGraphRouter* class.

Below only Java classes, developed by Michele Rodolfi will be described. For the native C code (JNI and other code derived from ION), the interested reader is referred to [M. Rodolfi, 2015].

3.2.2 Java classes

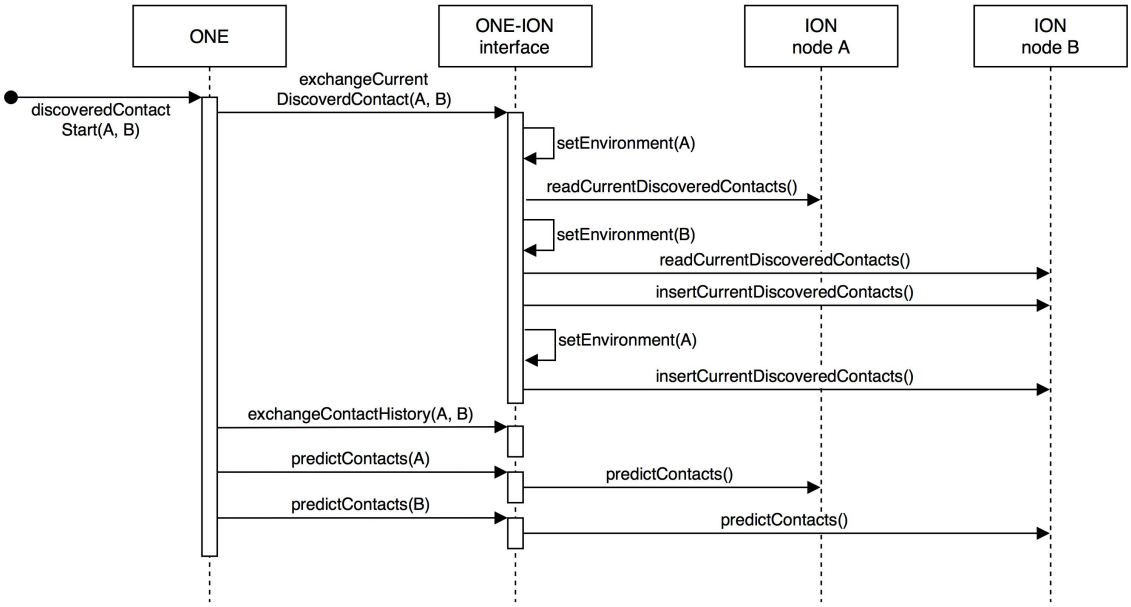

Figure 4: sequence diagram of function invocations triggered by the discovery of a new contact. The exchangeContactHistory() method has not been expanded as it behaves similarly to exchangeCurrentDiscoveredContact(). Function names and signatures have been renamed for a better reading.

Since we already extended ONE to support the simulation of CGR, in order to simulate OCGR we had just to extend the *ContactGraphRouter* class. The new *OpportunisticContactGraphRouter* class basically provides methods to inform the ION libraries of the acquisition or the loss of a discovered contact. It also provide a mechanism to support a epidemic routing drop back, if no routes can be found for a given bundle.

Contact discovery:

- In *OpportunisticContactGraphRouter* the new method *discoveredContactStart()* is invoked whenever a new discovered connection is acquired. It performs:
    - The current discovered contact exchange between the nodes pair. This operation is simulated by the *chsim.c* library that provides the function *exchangeCurrentDiscoveredContacts()*. This function is supposed to be invoked only once per nodes pair, thus it is called only if the local node is the connection's initiator.
    - The contact history exchange between the nodes pair. This operation is simulated by the *chsim.c* library that provides the function *exchangeContactHistory()*. This function is supposed to be invoked only once per nodes pair, thus it is called only if the local node is the connection's initiator.
    - The contact prediction on both nodes.
    - The insertion of the new discovered contact in the contact plan of both nodes.
- The new method of OCGR *discoveredContactEnd()* is invoked whenever a discovered connection is lost. It performs on both nodes the deletion of the discovered contact from the contact plan, the insertion of the discovered contact in the history log and the contact prediction.
- Whenever a connection between two nodes changes status, the ONE framework invokes the OCGR method *changedConnection()* on both ends of the connection. This method has been overridden in our class. It invokes:
    - The OCGR method *discoveredContactStart()* if the connection is up.
    - The OCGR method *discoveredContactEnd()* if the connection is down.

3.2.3 Epidemic drop back

An epidemic drop back mode is provided; it can be enabled to enhance the delivery ratio of bundles in the early stage of the simulation, i.e. when the contact history is too short to support a valuable contact prediction. Generally, the epidemic drop back mode is useful when

a node needs to forward a bundle whose destination cannot be reached using the information in the contact plan. This can be due to the fact that the local node has never encountered the bundle destination node or that the bundle destination node resides in a partitioned area of the network that has never been in touch with the local area.

The epidemic drop back takes control only if OCGR cannot find any route to the bundle destination (more precisely, any “plausible route”, in CGR terminology). If this is the case, the epidemic drop back tries to send the bundle to every neighbor currently in contact with the local node.

In order to implement this mechanism a new property has been added to the *Message* object: the *epidemicFlag* property. This property is a boolean: it is set to true if OCGR could not find a route to the destination for the bundle.

The epidemic drop back mechanism performs as follows:

- Whenever a bundle is created or received, its *epidemicFlag* property is set to false.
- Whenever OCGR cannot find a route for the bundle, the *epidemicFlag* property is set to true.
- Whenever OCGR can find a route for the bundle and the bundle is enqueued in a outduct, the *epidemicFlag* property is set to false.
- Whenever the local node has an active connection with a neighbor and it is not transferring any bundle, for each active connection:
    - It looks for the first bundle in limbo that has the *epidemicFlag* property set to true and it tries to send it to the neighbor.
    - If the transfer successfully starts, the bundle's *epidemicFlag* property is set to false and the node waits for the end of the transfer, otherwise the node tries to send the next bundle in limbo with the *epidemicFlag* property set to true
    - It repeats the previous step until either the transfer successfully starts or there are no more bundles in limbo with the *epidemicFlag* property set to true.

- The reason why a transfer can fail to start is because a peer node can refuse to accept the incoming bundle if it already has a copy of it. If this is the case, the epidemic drop back avoids to send a redundant bundle. This property distinguish epidemic from uncontrolled flooding.

3.2.4 The OGCR specific report

The ONE provides a series of simulation reports. The main report is the *MessageStatsReport*, which contains statistical information such as the number of bundles created, forwarded and delivered, the overhead ratio and the delivery rate. Each report type is defined in one class; the use of a specific report must be requested in the settings file before starting the simulation. We implemented a OCGR specific *MessageStatsReport* called *OCGRMessageStatsReport* to log a series of counters for the OCGR-forwarded bundles and for the epidemic-forwarded ones, in addition to the cumulative counters. This report is implemented in the *OCGRMessageStatsReport* class, which extends the *MessageStatsReport* class. This report works only with the *OpportunisticContactGraphRouter*.

3.2.5 Optimizations

First tests revealed that the OCGR simulation speed in The ONE is drastically much slower than the other routing protocols. For example, a simulation that would take a few minutes with PROPHET routing, it may take days with OCGR. This is due to the fact that while the simulation runs, the contact history of each node becomes longer and the prediction horizon moves further; therefore the contact plan will contain a huge amount of contacts (thousands). The route calculation performs a Dijkstra search through all the contacts in the contact plan, an processing time increase exponentially with the number of contacts. Speed is not the only issue we had to deal with: in fact during a Dijkstra search through a huge contact plan, the structures used to store routes information become very large, and the whole system memory can becomes full, thus causing a memory error. In order to cope with this problems, it was necessary to both optimize the code and change the algorithm to be faster and less memory hungry.

The total number of contact plan entries depends mainly on how many contacts are inserted by the contact prediction algorithm. In fact, for each node pair it can insert as many contacts

as the number of contact history entries that involve the same node pair. We can optimize this behavior performing as follows for each nodes pair:

- Instead of inserting all the predicted contacts in the contact plan, we limit the insertion to just one contact.
    - The start time of this contact is the current time (now).
    - The end time of this contact is the current time plus the prediction horizon (current time minus the start time of the first contact in the contact log).
    - The capacity of this contact is the sum of the capacities of the contacts in the contact log.
    - The confidence of this contact is calculated as before.

This optimization makes the contact plan length depending only on the number of nodes listed in the contact log, and no more on the total contact log length. This is an approximation of the OCGR that speeds up the simulation and reduces the memory usage, while maintaining the functionality and the forwarding ability of the algorithm.

The CGR library defines three different payload classes and performs route calculation for each of them. Each payload class defines a contact volume (or improperly “contact capacity”) floor threshold: every contact whose volume is less than the threshold size for the class is not taken into account in route calculation. The payload classes define the following threshold:

- Payload class 0: 1 kB.
- Payload class 1: 1 MB.
- Payload class 2: 1 GB.

Therefore, instead of repeating three times the Dijkstra search, we limited the route calculation to only the payload class 1, that is: any contact whose capacity is less than 1 MB is omitted from the route calculation. This enhances the route calculation speed but may deprives the bundle of some routes. Anyway The ONE does not support bundle fragmentation and the simulated bundles size is often from 500 kB to 1 MB. Also, with the contact prediction optimization that enlarge the predicted contact volumes, we can say that a contact

whose capacity is less than 1 MB is unlikely to happen or at least not useful. In addition, we limited the route calculation to those routes whose first hop is a discovered contact, i.e. currently active. In fact, if the route's first hop is not a discovered contact, the bundle can not be forwarded.

3.3 PriorityContactGraphRouter

3.3.1 Introduction

This class extends the *ContactGraphRouter* class to include the support of messages with the priority attribute. It required both Java and native code extensions or modifications. It also includes the Overbooking Management CGR enhancement, lacking in the existing CGR class, as it is strictly related to priorities. It is convenient to examine the two extensions separately.

3.3.2 Inclusion of priorities

We started from the *ContactGraphRouter* class, already available, which contains the *Outduct* class which represents a list containing all bundles (messages) waiting for the opportunity of being transmitted to the corresponding node. Each *ContactGraphRouter* has an array of *Outduct* objects, one for each proximate node. When contacts open, the router manages the actual bundle transmission. To include priorities in *ContactGraphRouter*, we have created *PriorityContactGraphRouter* as an extension of *ContactGraphRouter*. The new class contains *PriorityOutduct*, which extends *ContactGraphRouter*.*Outduct*. The new *Priority*Outduct has obviously a different list for each priority class, instead of just one. Then in *PriorityOutduct* all methods related to bundle insertion and removal have been redefined, while in *PriorityContactGraphRouter* those selecting the messages to be sent or to be dropped (taken in priority and arrival order). The method *updateOutducts()*, of the class *PriorityContactGraphRouter*, has been redefined to use *PriorityOuduct* instead of *Outduct* objects.

The second and last class we modified was *IONInterface*. This static class exposes the methods used by C libraries to manage objects such as messages or outducts. In particular, we introduced the methods to obtain the priority of a message and those to obtain the queue

length in byte inside an outduct (backlog), taking priorities into account. Note that for *ContactGraphRouter* these methods returns messages of Normal (1) priority.

The first C file to be modified was *ONEtoION_interface.c*, which contains the calls to the static methods described in the Java section above and the conversions of objects from Java to C. We inserted the procedures to obtain information from Java side:

- *getMessagePriority()*
- *get{Bulk | Normal | Expedited}Backlog()*

The syntax to call Java functions from C is particular; first, *getThreadLocalEnv()* provides us with a pointer to the execution environment; then, by means of this pointer *FindClass()* gives the class of the method we want to call, and then *GetStaticMethodId(),* the method itself. The signatures can be obtained by means of the command *javap -s* passing as an argument the class containing the methods we are looking for. Once all information is available the Java method is called with a function that depends on the type of return (e.g. *CallStaticIntMethod*), passing as arguments the execution environment, the class name, the method name and all the method-specific arguments.

The functions *ion_bundle()* and *ion_outduct()* that convert Java messages and outducts into ION bundles and outducts, were also modified. Concerning bundles, the flags 8 and 7 of the *bundleProcFlags* must be set according to the value returned by *getMessagePriority():* 10 expedited, 01 normal, 00 bulk. Concerning *ion_outduct()*, we need to insert the backlog values for each priority. The following two instructions, referring to bulk, must be repeated also for normal and expedited, paying attention that in this file they are improperly called *std* and *urgent*.

```
long bulkBacklog = getOutductBulkBacklog(jOutduct);
loadScalar(&(outduct->bulkBacklog), bulkBacklog);
```

The last modification has been carried out in *libbpP.c*, where the function *computePriorClaims()* returns both the total number of bytes in the outduct (i.e. the queue length), and then those seen by the bundle to be enqueued, i.e. by skipping those of lower priority. The version developed for *ContactGraphRouter* returned the same value for both

parameters, as priorities were not supported. It now reports the correct values, which are necessarily equal only if the bundle to be enqueued is bulk.

3.3.3 Inclusion of Overbooking Management

The Overbooking Management is a CGR enhancement aiming at managing the overbooking (or “oversubscription”) of a contact, a priori, i.e. before the end of the contact. It is associated to priorities, as overbooking in CGR can only happen as a result of later arrival of higher priority bundles that take the “seats” of lower priority bundles already in the outduct.

The Overbooking Management is an option included in ION in the libcgr.c and its functions work directly on the ION outducts. However, in The ONE the outducts are in the Java environment, thus it resulted convenient to integrate the Overbooking Management feature into the *PriorityContactGraphRouter.* moving a significant part of the code from C to Java. To find the messages to be re-forwarded, i.e. to be removed from an outduct, the Overbooking Management calculates the values *protected* (the bytes to be skipped, as belonging to a subsequent contact) and *overbooked* (i.e. the bytes to be removed from the queue). This values now must be passed from C to Java. To this end, we have created a chain of functions starting from the *cgrforward()* function in *libcgr.c,* where a conditional block *#ifdef* contains the call to the interface. The compilation of this block must be carried out only if *libcgr.c* is compiled for The ONE instead of ION, and it is the only modification introduced in *libcgr.c* to support CGR in ONE. It is worth noting that to minimize the changes to *libcgr.c* is one of our design aims. The chain of functions terminates in the *PriorityContactGraphRouter* where we are inserting the bundle. To this end, a static nested class called *OverbookingStructure* was created inside *PriorityContactGraphRouter*. This new class contains: the outduct reference, the *overbooked* e *protected* values, and two indexes to know from which priority list and from which point of the list the next message must be removed.

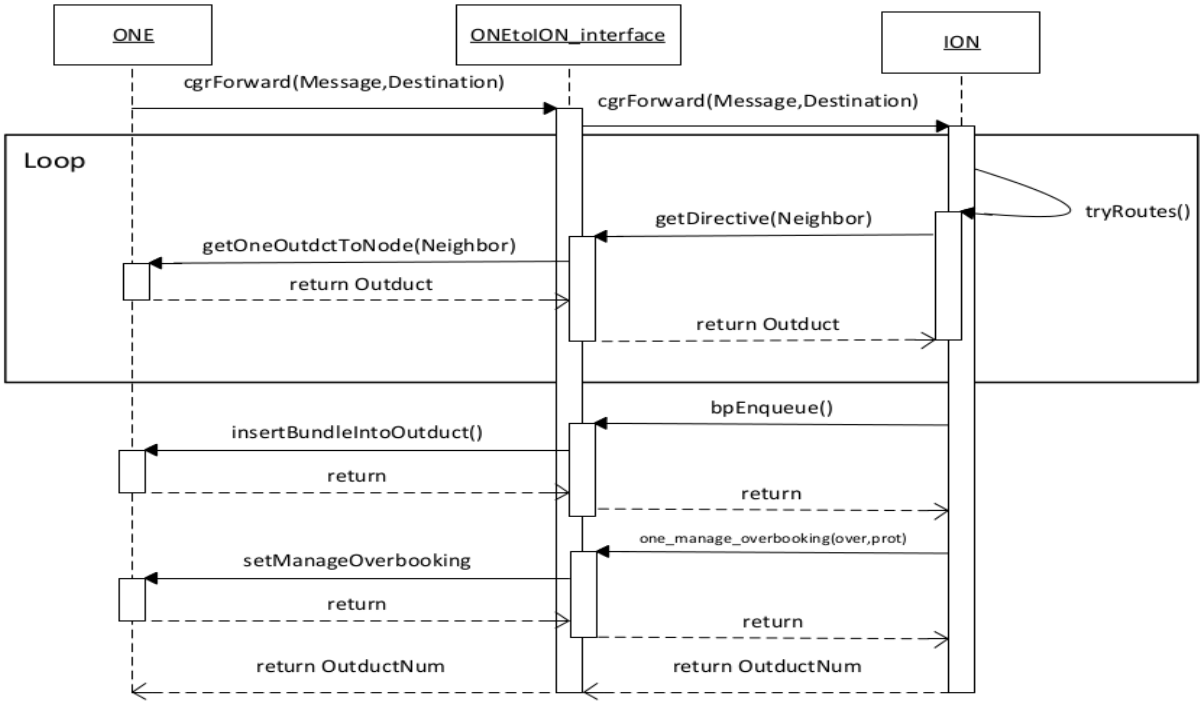

Figure 5: workflow cgrForward

In the work flow above the call to the *setManageOverbooking()* method adds one *OverbookingStructure* instance to a stack. Once the JNI call has finished, the method *ManageOverbooking()*, is called; the first object in the stack is taken and then we have the following cycle: until *protected* is positive, the current bundle is skipped and its dimension is subtracted from *protected*;.the first message bigger than the residual value of *protected* is the first message to be removed from the queue and re-forwarded. The message dimension is subtracted from *overbooked* and *cgrForward()* is invoked passing as arguments the message just removed.

It is worth noting that the re-forwarded bundle has always a priority lower than the priority of the incoming bundle that has caused the overbooking. If the incoming bundle is expedited, it causes the re-forwarding of bulk bundles, until available, and then, if bulk bundles are not enough, of normal bundles. A normal bundle re-forwarded can in turns cause the overbooking of its new contact, but then the chain is broken. In fact, as a result of this second overbooking only bulk bundles can be re-forwarded. The stack allow us to simulate this recursive behavior. When the variable *overbooked* reaches zero, the *OverbookingStructure* object is removed from the stack and if there are not others the simulation is restarted.

3.4 PriorityOpportunisticContactGraphRouter

3.4.1 Introduction

Similarly to *PriorityContactGraphRouter*, this class extends the *OpportunisticContactGraphRouter* class to include the support of messages with the priority attribute and the Overbooking Management support, both lacking in the existing *OpportunisticContactGraphRouter*. Since all modifications required outside the new routing class itself have already been done to include the support of priorities and Overbooking Management in *PriorityContactGraphRouter,* it is convenient to examine the two extensions together.

3.4.2 Inclusion of priorities and Overbooking Management

To include priorities in the *OpportunisticContactGraphRouter* class, we have created the new class *PriorityOpportunisticContactGraphRouter* as an extension of the former. Since Java does not support multiple inheritance, the new class contains the inner class *PriorityOutduct,* like Priority*ContactGraphRouter* does, which extends *ContactGraphRouter*.*Outduct*. The new *PriorityOutduct* has a different message list for each priority class and also the relative getters methods: *get{Bulk | Normal | Expedited}Queue()* and *get{Bulk | Normal | Expedited}Backlog().* Furthermore, all methods related to bundle insertion and removal have been redefined: *getEnqueuedMessageNum(), containsMessage(), insertMessageIntoOutduct()* and *removeMessageFromOutduct()*. In the nesting class *PriorityOpportunisticContactGraphRouter,* as *in PriorityContactGraphRouter,* the methods selecting the messages to be sent or to be dropped (taken in priority and arrival order) have been overridden. They are: *updateOutducts(), replicate(), getMessagesForConnected()* and *checkExpiredRoutes()*.

To include the Overbooking Management it has been sufficient to import the *OverbookingStructure* static class nested in *PriorityContactGraphRouter.* It is worth noting that *PriorityContactGraphRouter.PriorityOutduct* could not be imported the same way, since it is not a static class.

As mentioned before, it was not necessary to carry out any other modifications to other classes or C files.

3.4.3 Testing

In order to check the proper functioning of the *OpportunisticContactGraphRouter*, we carried out a few simulations. Results of one of the most representative are reported below, as an example. This simulation involves a group of twenty nodes (pedestrians), all moving casually along the map of Helsinki and using the same wireless interface. The probabilities of creation of each class of message are as follows: 55% bulk, 35% normal and 15% expedited. The settings file used to obtain this results is included in the cgr-jni-Merge packet and is called “pozza_pocgr_settings.txt”.

```
Message stats for scenario PriorityOpportunisticContactGraphRouter
sim_time: 43200.1000
created: 1139
bulk created 541
normal created 323
expedited created 275

started: 7904
relayed: 7858
bulk relayed 1158
normal relayed 2298
expedited relayed 4402

aborted: 46
dropped: 6689
bulk dropped 1255
normal dropped 2143
expedited dropped 3291

delivered: 371
bulk delivered 108
normal delivered 126
expedited delivered 137

delivery_prob: 0.3257
bulk_delivery_prob: 0.1996
normal_delivery_prob: 0.3901
exp_delivery_prob: 0.4982
latency_avg: 6080.4296
latency_med: 4792.8000
hopcount_avg: 2.5094
hopcount_med: 2
buffertime_avg: 5776.0845
buffertime_med: 4820.7000
```

Figure 6: PriorityMessageStatsReport result of the simulation

Results are satisfactory. In particular, by observing The ONE log shown in Figure 1, it is possible to note that the message delivery probability is in accordance with the assumption that messages with higher priority should have greater likelihood of delivery.

4 Source code organization

The integration of CGR and OCGR into The ONE, carried out in this thesis resulted into a single software packet, called “cgr-jni-Merge”, which contains both the Java classes that extend the ONE framework, the native C code that simulates the ION environment and a few files taken form the ION package, including libcgr.c, with minimal modifications The cgr-jni-Merge packet also contains examples of contact plans (directory resources), settings files (directory simulations) and a script “compile_cgr-jni.sh” to facilitate the compilation of our classes. This software supports CGR with an extension to that allows The ONE to create and forward messages with 3 priority levels, manage contact overbooking and also supports OCGR with the same functionalities.

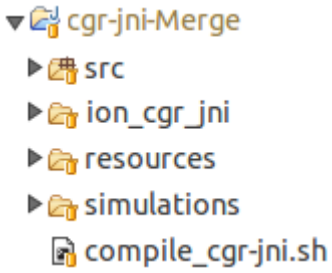

Figure 7: contents of the cgr-jni-Merge packet

4.1 The Java classes

The Java code is organized following the Java standard guideline for packages and classes: each file contains a class and is contained in a folder whose name is the package that contains the class. The root directory of the Java code is the folder *src*.

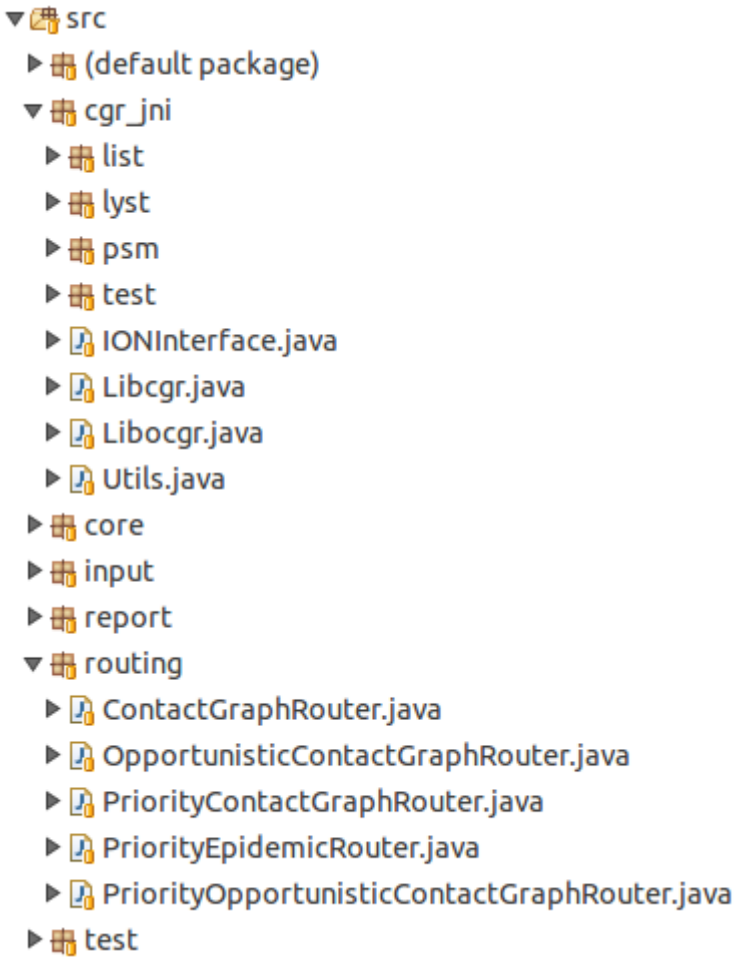

Figure 8: contents of the src directory

Since the Java classes are used in The ONE framework, it is necessary to organize them in the same packages used by The ONE. The packages and classes used directly by The ONE are:

- package *core*: class *PriorityMessage*;
- package *input*: contains classes which implement message events generators;
- package *report*: these classes are used to create summary data of simulation runs, detailed data of connections and messages, and can interface with other programs.
- package *routing*: contains the classes, described in the previous chapter, which implement CGR and OCGR with and without priorities;

- package *test*: contains JUnit tests for *ContactGraphRouter, PriorityContactGraphRouter, OpportunisticContactGraphRouter* and *PriorityEpidemicRouter.*

The *cgr_jni* package contains only classes that manage the JNI interaction with the ION integration native code.

4.2 The C source and header files

The C source and header files are in the *ion_cgr_jni* folder that tries to follow the original ION distribution file organization.

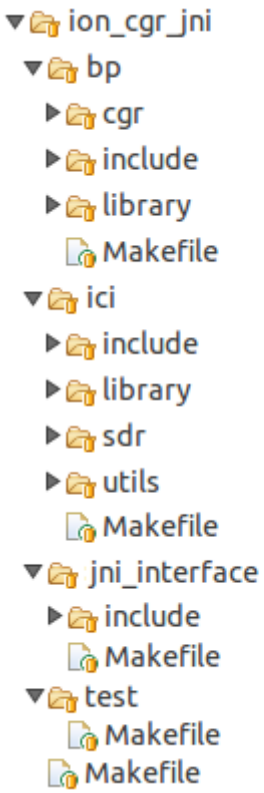

Figure 9: contents of the ion_cgr_jni directory (the jni_interface folder contains files that are not shown)

This folder contains the following sub directories:

- folder *bp*: contains the ION *libcgr.c* source file and all the needed headers exactly as in the original ION distribution.

- folder *ici:* contains all the source files of the ICI libraries and the needed headers exactly as in the original ION distribution.
- folder *jni_interface*: contains all the source and header files that support the JNI interaction between the ONE framework and the ION adaptation.
- folder *test*: contains source and header files used for JNI and simulated libraries tests. These files are not used for simulations.

All the above mentioned folders 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.

5 Conclusions

The goals of this thesis were manifold:

- to combine the contributions of the various students who have worked, before me, at the integration into The ONE of CGR and OCGR, into a single packet;
- to extend the existing code in order to include the support of priority messages and the Overbooking Management for OCGR;
- and lastly, to write the documentation necessary for both new users, who want to test the routing algorithms, and developers, who want to further extend the software.

These goals have been achieved. Moreover, although the original ONE and ION code was not fully preserved, as desired, only a few minimal modifications resulted necessary to include all new routing classes, which is a clear advantage in term of maintenance.

During the merge, it appeared evident that three sets of classes, which provide three different interesting functionalities, such as priorities, contacts export into an external contact plan file, and vice versa, the use of an external contact plan to manage contacts in ONE, are not only independent of CGR, but also of any native code, thus they could be added in the future to The ONE installation.

Concerning the last goal, this thesis does not constitute an exhaustive documentation, as it neglects native code already described in [M. Rodolfi, 2015], to focus on the Java side, and in particular on the Java classes of greatest interest for the user, such as the four classes related to CGR. The two Appendixes, an installation and use guide, and a description of configuration settings, complete the documentation, with the aim of contributing to a possible widespread use of the developed software.

Appendix 1: Installation of the “cgr-jni-Merge” packet

This appendix is a brief user guide to integrate the cgr-jni-Merge packet version, into ONE. The guide is intended both for developers that want to change or extend the code and the users interested only in running simulations. This guide largely extends a previous document, [A. Berlati, 2016]. The ONE simulator with a complete manual can be downloaded from akeranen.github.io/the-one/, while cgr-jni-Merge can be downloaded from github.com/BerlaT/cgr-jni/tree/Merge.

The ONE original package

The ONE folder contains all original Java classes, setting files and two scripts for compilation and running (one.sh and compile.sh respectively). This guide and the code refers to the latest version available at the time of 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.

Mandatory modifications

The ONE code needs to be changed as follows:

- class *core.DTNHost*
  - line 22: initialize nextAddress with 1.
  - line 107: 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 the java command invocation, this variable must be set to allow the JVM to find our Java classes. We also add “bin” in case the user wants to compile the 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.

Optional modifications

The following changes are necessary only if we want to use the class *OCGRMessageStatsReport*. A new method called *getMoreInfo()* must be added to the class *report.MessageStatsReport*. 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 the *report.OCGRMessageStatsReport* class, so it is necessary to add it here, although it returns an empty string.

Compiling and launching The ONE and cgr-jni from command line interface

First, the JAVA 8 compiler (or above) is required. Java version can be checked using

```
java -version
```

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. Check the value of this environment variable using

```
echo $JAVA_HOME
```

Compiling the ONE and our cgr-jni Java classes

The compilation can either be done from a command line interface or in Eclipse. Here we will focus on the compilation from a command line interface (easier for normal users). Developers may prefer the compilation in Eclipse, explained later.

To facilitate the resultion of dependencies and the compilation we can use two scripts. The ONE provides a script, “compile.sh”, which can be used to compile all the ONE classes together. This script put the class files into the “target” folder, but it can be changed. After running “compile.sh” it is necessary to enter the following commands:

```
export ONE_BIN=/path/to/ONE/target
```

where “/path/to/target” denotes the folder containing the ONE .class files and then

```
export CGR_JNI_CLASSPATH=/path/to/cgr-jni-Merge/bin
```

also used in the script one.sh (see below).

Finally, our script “compile_cgr-jni.sh” can be launched to compile our Java classes.

Compiling the native C code

To compile the native C code the use of command line interface is recommended. To this end the user should use the *Makefile* in the cgr-jni-Merge/ion_cgr_jni directory, entering the following make command from the directory mentioned above:

```
make ONE_CLASSPATH=/path/to/ONE/target[DEBUG=1]
```

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

It is important to note that the *Makefile* assumes that the Java classes of the cgr_jni packet 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 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.

The 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 library, the second to the class files of the cgr-jni packet:

```
export LD_LIBRARY_PATH=/path/to/libcgr_jni.so
export CGR_JNI_CLASSPATH=/path/to/cgr-jni-Merge/bin
```

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 /path/to/settings.txt
```

Running batch simulations

In order to simplify the simulation set up and the results analysis, the batch_test.sh 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 The 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] the following three modes are defined:

Buffer: the nodes buffer size changes.

Message: the bundle size changes.

TTL: the bundle time to live changes.

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 The ONE and cgr-jni in 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, all classes must be recompiled. To this end, we show the steps for a specific IDE, Eclipse. It is important to note that it is required Eclipse for Java Enterprise Edition

Importing The ONE and cgr-jni as two different projects

If The 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-Merge/src folder must be linked to The ONE in this way:

1. Right Click on The ONE project > Build Path > Configure Build Path > Left Click on the Source tab > Link Source > select /path/to/cgr-jni-Merge/src
2. Right Click on the new source > Properties > Native Library > Location path: /path/to/cgr-jni-Merge/ion_cgr_jni

Compilation the native C code

Native library compilation must be done as before by using the Makefile in cgr-jni-Merge/ion_cgr_jni.

```
make ONE_CLASSPATH=/path/to/ONE/bin[DEBUG=1]
```

Note that Eclipse puts class files in the *bin* directory and not in the *target* directory like the script “compile.sh” of The ONE does.

Running simulations

After the native library is compiled, 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 the following line is added: “${string_prompt}” in Run > Run Configurations > Java Application > “your application” > Arguments > Program Arguments. In this case, after clicking the run button, a text field is shown where we can add command line arguments like batch simulation (-b) and all the settings files that are needed.

Appendix 2: The ONE settings files

The 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 specified in configuration files that must be passed as arguments when the simulation is launched. This appendix aims to help the user familiarize with the ONE configuration files and in particular with settings specific to CGR and OCGR routing classes. It is worth noting that whenever multiple files are given to The ONE at start up, new settings are added, while old settings are overridden by new settings with the same name.

General settings

The default settings file, default_settings.txt, contains general settings and specific settings should be put in other files Here will be presented some the most important settings; the user is referred to The 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 desired routing algorithm. It is set a transmit interface (btInterface) for all groups. The movement model name is also a java class from The ONE 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 sources and destinations, 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 The 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 The 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 The ONE developers, 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 settings

The following settings are specific to the classes which implements CGR with and without priorities (*ContactGraphRouter* and *PriorityContactGraphRouter* respectivly): the path to contact plan and three message generators, one for each priority class (only the 3 cardinal priorities, bulk, normal expedited, are implemented, denoted as 0, 1 and 2). Router must be set as *PriorityContactGraphRouter* if three *PriorityMessageEventGenerator* are used or as *ContactGraphRouter* if priorities are not needed.

```
Scenario.name = CGR_settings
Group.router = PriorityContactGraphRouter
ContactGraphRouter.ContactPlanPath = /path/to/contact_plan.txt
Report.report1 = PriorityMessageStatsReport
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

The following settings are specific to the classes which implements OCGR with and without priorities (*OpportunisticContactGraphRouter* and *PriorityOpportunisticContactGraphRouter* respectivly). As this routing algorithm 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.

```
Scenario.name = OCGR
Group.router = OpportunisticContactGraphRouter
OpportunisticContactGraphRouter.epidemicDropBack = false
OpportunisticContactGraphRouter.debug = false
Report.report1 = OCGRMessageStatsReport
```

If *PriorityOpportunisticContactGraphRouter* is used, three *PriorityMessageEventGenerator* must also be specified like for *PriorityContactGraphRouter* and the report class must be changed aswell.

```
# one message generator for each priority level: Bulk, Normal and
Expedited
Events.nrof = 3
# generator of "Bulk" messages
Events1.class = PriorityMessageEventGenerator
Events1.interval = 1,160
Events1.size = 100k
Events1.hosts = 1,20
Events1.prefix = B
Events1.priority = 0
# generator of "Normal" messages
Events2.class = PriorityMessageEventGenerator
Events2.interval = 1,255
Events2.size = 50k
Events2.hosts = 1,20
Events2.prefix = N
```

```
Events2.priority = 1
# generator of "Expedited" messages
Events3.class = PriorityMessageEventGenerator
Events3.interval = 1,300
Events3.size = 10k
Events3.hosts = 1,20
Events3.prefix = E
Events3.priority = 2
Report.report1 = PriorityMessageStatsReport
```

Bibliography

[A. Balasubramanian et al., 2007 and 2010] Balasubramanian, Aruna, Brian Levine, and Arun Venkataramani. "DTN routing as a resource allocation problem." ACM SIGCOMM Computer Communication Review 37.4 (2007): 373-384.

[A. Berlati, 2016] A. Berlati, “Gestione dei messaggi con priorità nel simulatore di DTN The ONE”, Internship report, University ofBologna, May 2016.

[A. Lindgren et al., 2012] A. Lindgren, A. Doria E. Davies, and S. Grasic, “Probabilistic Routing Protocol for Intermittently Connected Networks”, Internet RFC 6693, Aug. 2012.

[C. Caini et al., 2011] C. Caini, H. Cruickshank, S. Farrell, M. Marchese, “Delay- and Disruption-Tolerant Networking (DTN): An Alternative Solution for Future Satellite Networking Applications”, Proceedings of IEEE, Vol. 99, N. 11, pp.1980-1997, Nov. 2011.

[F. Fiorini, 2016] F. Fiorini, “Inserimento dell'algoritmo di routing CGR nel simulatore DTN ONE: Contact Plan e priorità in ONE”, Internship report University of Bologna, May 2016.

[G. Araniti et al., 2015] G. Araniti, N. Bezirgiannidis, E. Birrane, I. Bisio, S. Burleigh, C. Caini, M. Feldmann, M. Marchese, J. Segui, and K. Suzuki, “Contact graph routing in DTN space networks: overview, enhancements and performance,” IEEE Communications Magazine, vol. 53, no. 3, March 2015, pp. 38-46.

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

[J. J. Messina, 2015] Jako Jo Messina, “Inclusion of Contact Graph Routing in The ONE DTN simulator”, Master Thesis, University of Bologna, March 2015.

[Keränen, 2000] A. Keränen, J. Ott, and T. Kärkkäinen, “The ONE Simulator for DTN Protocol Evaluation”, in Proceedings of the 2nd International Conference on Simulation Tools and Techniques. New York, NY, USA: ICST, 2009.

[M. Rodolfi, 2015] Michele Rodolfi, “DTN discovery and routing: from space applications to terrestrial networks”, Master Thesis, University of Bologna, March 2015, http://amslaurea.unibo.it/10361/ Accessed 2016-12-14.

[RFC4838] V. Cerf, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss “Delay-Tolerant Networking Architecture”, Internet RFC 4838, Apr. 2007, http://www.rfc-editor.org/rfc/rfc4838.txt Accessed 2016-12-06.

[RFC5050] K. Scott, S. Burleigh, “Bundle Protocl Specification”, Internet RFC 5050, Nov. 2007, http://www.rfc-editor.org/rfc/rfc5050.txt Accessed 2016-12-06.

[S. Burleigh et al., 2016] S. Burleigh, C. Caini, J. J. Messina, M. Rodolfi, “Toward a Unified Routing Framework for Delay-Tolerant Networking”, in Proc. of IEEE WiSEE 2016, Aachen, Germany, Sept. 2016.

[S. Burleigh, 2015] S. Burleigh, “Interplanetary overlay network design and operation V3.3.1,” JPL D-48259, Jet Propulsion Laboratory, California Institute of Technology, CA, May 2015. [Online]: http://sourceforge.net/projects/ion-dtn/files/latest/download

[SATRIA] Deni Yulianti, Satria Mandala, Dewi Naisien, Asri Nagad, YahayaCoulibaly, “Performace comparison of Epidemic, PRoPHET, Spray and Wait, Binary Spray and Wait, and ProPHETv2”.

[T. Spyropoulos et al., 2005] T. Spyropoulos, K Psounis, and C. S. Raghavendra, “Spray and wait: An efficient routing scheme for intermittently connected mobile networks”, in Proc. of 2005 ACM SIGCOMM workshop on Delay-tolerant networking, WDTN’05, 2005, pp. 252.

[Vahdat and Becker, 2000] Amin Vahdat and David Becker, “Epidemic routing for partially connected ad hoc networks”, Technical Report CS-2000-06, Department of Computer Science, Duke University, April 2000-