University of Bologna

School of Engineering and Architecture

Two Year Master Course in Computer Engineering

Master Thesis
in
Protocols and Architecture for Space Networks M

DTN DISCOVERY AND ROUTING: FROM SPACE APPLICATIONS TO TERRESTRIAL NETWORKS

Student:
Michele Rodolfi

Supervisor:
Prof. Ing. Carlo Caini

Co-supervisor:
Scott Burleigh (JPL NASA)

Session III
Academic Year 2014/2015

ABSTRACT

This thesis deals with the Delay-/Disruption-Tolerant Networking (DTN) architecture, which was designed to support communications within “challenged networks”: environments where the TCP/IP protocol stack may be ineffective due to long round-trip-times, high packet loss ratio and link disruption. Challenged networks are very heterogeneous and examples of them vary from interplanetary networks to Mobile Ad-Hoc Networks (MANETs). The main implementations of the DTN architecture are DTN2, the reference implementation, and ION, developed by NASA JPL, until now more oriented to space applications. A significant difference between space and terrestrial networks is that while in space nodes movements and contacts are deterministic, as related to the movement of planets and spacecrafts, terrestrial mobile nodes in MANETs or in wireless sensor networks do not generally have any prior knowledge about contacts, which most of the times are related to independent movements of nodes, and more commonly, also of network topology. This leads to the adoption of completely different routing strategies: deterministic for space networks and opportunistic for terrestrial mobile networks. A DTN implementation should be effective within the DTN environment heterogeneity, consequently NASA JPL has recently decided to extend ION in order to support also non-deterministic scenarios. To this purpose, during my thesis work, carried out at NASA -JPL in Pasadena under the guide of my co-supervisor, Scott Burleigh, I have worked on different topics all related to this common aim. Firstly, I tested the brand new IP Neighbor Discovery (IPND) ION implementation: bugs have been fixed and the official “man” page written from scratch. Then, my research has focused on the integration of the existing deterministic ION Contact Graph Routing (CGR) into ”The ONE” DTN simulator, using the Java Native Interface (JNI) as a bridge between the Java code of the simulator and the C code of ION. The ION native libraries have been adapted to work within The ONE environment in order to allow the CGR code to work in The ONE without any modifications, thus root avoiding all disadvantages related to the development of a parallel implementation

of CGR in Java. Furthermore, after the careful analysis of some mobile trace datasets, I supported my co-supervisor in the development of an opportunistic extension of CGR (OCGR) , to be implemented as an ION module first, and then integrated into The ONE. Preliminary tests carried out with OCGR in The ONE seem to have proved the potential of OCGR: once properly tuned, it could become a valid competitor of the most renowned opportunistic solutions, while maintaining its undiscussed superiority when applied to deterministic environments.

PREFAZIONE

L'argomento di questa tesi è l'architettura di rete Delay-/Disruption-Tolerant Networking (DTN), progettata per operare nelle reti “challenged”, dove la suite di protocolli TCP/IP risulta inefficace a causa di lunghi ritardi di propagazione del segnale, interruzioni e disturbi di canale, ecc. Esempi di reti “challenged” variano dalle reti interplanetarie alle Mobile Ad-Hoc Networks (MANETs). Le principali implementazioni dell'architettura DTN sono DTN2, implementazione di riferimento, e ION, sviluppata da NASA JPL per applicazioni spaziali. Una grande differenza tra reti spaziali e terrestri è che nello spazio i movimenti dei nodi sono deterministici, mentre non lo sono per i nodi mobili terrestri, i quali generalmente non conoscono la topologia della rete. Questo ha portato allo sviluppo di diversi algoritmi di routing: deterministici per le reti spaziali e opportunistici per quelle terrestri. NASA JPL ha recentemente deciso di estendere l'ambito di applicazione di ION per supportare anche scenari non deterministici. Durante la tesi, svolta presso NASA JPL, mi sono occupato di argomenti diversi, tutti finalizzati a questo obiettivo. Inizialmente ho testato la nuova implementazione dell'algoritmo IP Neighbor Discovery (IPND) di ION, corretti i bug e prodotta la documentazione ufficiale. Quindi ho contribuito ad integrare il Contact Graph Routing (CGR) di ION nel simulatore DTN “ONE” utilizzando la Java Native Interface (JNI) come ponte tra il codice Java di ONE e il codice C di ION. In particolare ho adattato tutte le librerie di ION necessarie per far funzionare CGR all'interno dell'ambiente di ONE. Infine, dopo aver analizzato un dataset di tracce reali di nodi mobili, ho contribuito a progettare e a sviluppare OCGR, estensione opportunistica del CGR, quindi ne ho curato l'integrazione in ONE. I risultati preliminari sembrano confermare la validità di OCGR che, una volta messo a punto, può diventare un valido concorrente ai più rinomati algoritmi opportunistici.

Index

1 INTRODUCTION

1.1 DTN architecture

The Delay-/Disruption-Tolerant Networking (DTN) architecture has been designed to allow communications in those scenarios where the TCP/IP protocols are not able to work, called “challenged networks”. In fact, the Internet architecture is based on some fundamental assumptions that do not hold in these scenarios. These assumptions are: short round trip times (RTT), low channel error rates, connected end-to-end path from the source to the destination, channel symmetry, etc. If at least one of these assumption is not verified, the network is defined ‘challenged’. Examples of challenged scenarios are: interplanetary networks, where the RTTs are in the order of minutes and the connected path to the destination is not always feasible (if a lander is on the opposite side of a planet with respect to an orbiter), Mobile Ad-Hoc Networks (MANETs), where the network may be often partitioned making not possible an end-to-end connected transmission path.

The DTN architecture has been standardized and it is described in the [RFC4838]. The main aspects are summed up below:

- A new layer is added to the TCP/IP communication stack, between the application layer and transport layer on and nodes: it is named bundle layer, and a packet at this layer is called bundle. The bundle layer can also be present in some intermediate nodes, where it is on top of Transport and the last of the stack, as Application is only on end nodes. The correspondent protocol of this layer is the “Bundle Protocol” [RFC5050].
- The main goal of this layer is hiding the underlying layers to the application layer, in order to have the opportunity to create a heterogeneous network. In this way, an application just needs to interact with the bundle protocol, without

caring about the transport protocols under it.

- DTN nodes have the capability to store data, since 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. To make communications possible it becomes necessary to store data at DTN nodes, waiting for the availability of the next hop: the technique just described is called store and forward and is a characteristic feature of DTN. A bundle, once received by a node, can be stored for a long period (even 24 hours), until the next path becomes available. This mechanism, when coupled with the "Custody Option" is also really useful in case of loss recovery with very long RTTs, as the bundle can be retransmitted by an intermediate node, called custodian, instead of the source.
- Two kinds of fragmentation, proactive and reactive, are supported. The former is used to cope with scheduled intermittent contacts. A contact is an opportunity of transmission between two nodes. Since contacts are limited in time, the maximum amount of data that can be transferred during a contact, called contact volume or contact capacity, is limited. As a result, if the bundle dimension is larger than a contact volume, the proactive fragmentation splits the bundle a priori, i.e. before the end of the contact. The reactive fragmentation, by contrast, is used a posteriori, for example when a disruption, i.e. a random unavailability of the link occurs, and allows the sender to re-transmit only the bundle part that was not received before the disruption.

1.2 DTN routing

Bundle routing is a open research field in the DTN community. Indeed, there are several problems that could affect the computation of a route, such as network partitioning, scheduled intermittent links (available only in certain moments), limited storage in the intermediate nodes (the price to pay for store and forwarding benefits), high delay and disruption in the routing information exchange among nodes in the network.

Since DTN networks are highly heterogeneous, the best routing policy could be

different in various regions. In fact, given different scenarios, is really difficult to design a general and optimal routing algorithm. Instead, a routing algorithm could be created in order to cope with different scenario-specific issues. For example, while in a certain environment the most important factor could be the bundle delivery time; in another one it could be the overhead ratio. For these reasons, the best solution could be a hybrid solution, which involves more than one routing scheme, and requires some nodes to act as a gateway between different router domains. For example, a quite common case is a network divided into two parts: a well – connected one that uses a normal static routing algorithm, and a challenged part that uses a scenario-specific router.

DTN literature enumerates a large number of totally different routing algorithms, usually cataloged into two families, considering how much the algorithm knows 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.

Among the first group, we have algorithms based on an exchange of calculations and measurement which allow them to a real time update of best route and forwarding decisions. Moreover, these kind of approaches usually use a flooding–based strategy, replicating the messages a number of times dependent from their algorithm. This group of algorithm works very good in networks with high node mobility and no storage problem (flooding based, so several copies of every message are possible), giving high delivery success rate, even if the current network state is not known.

Examples of opportunistic replication-based routing algorithms are:

- Epidemic routing [Vahdat and Becker, 2000], 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 [Grasic et al., 2010] 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 [Balasubramanian et al., 2007] 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.

Before talking about the deterministic algorithms, and moving from the "zero knowledge" to the "complete knowledge" of the network, some authors in [S. Jain et al., 2004] consider an intermediate group of algorithms belonging to a so called "partial knowledge" category. In this group we find Minimum Expected Delay (MED), which uses statistical information about contacts, Earliest Delivery (ED) which uses information about contacts and its two further optimizations, Earliest Deliver with Local Queue (EDLQ) and Earliest Delivery with All Queue (EDAQ), which respectively add information about local and all queues.

As we said, a hybrid solution is usually used, and it could require some nodes to act as gateways between different router domains. For example, a quite common case is a network divided into two parts: a well – connected one that uses a normal static routing algorithm, and a challenged part that uses one of the aforementioned algorithms.

Finally, in the deterministic group, we find algorithms that work well when contact are predictable, achieving the best delivery rate and saving as much bandwidth and buffer space as possible. These kind of algorithms have to consider also the aforementioned contacts could fail to occur, leading the algorithms themselves to a re-computation of the whole topology and best possible route. The most important ones are MARVIN and the Contact Graph Router (CGR). These algorithms use contact predictions, spread all

over the network, in order to build network graphs and take forward decision on a hop-to-basis. In particular, MARVIN encodes information about the operational environment and infers contact opportunities from this knowledge; CGR, instead, obtains information about contacts from the “contact plan”, and tries to calculate the best path considering different routing metrics and performance indicators [Burleigh et al., 2015].

1.3 The ONE simulator

Performing tests on real opportunistic DTN environment is really hard due to the non-deterministic and heterogeneous nature of nodes movements and capabilities. Therefore to verify the validity of a opportunistic routing protocol the environment needs to be synthesized and simulated.

The ONE simulator is an opportunistic networking evaluation system that offers a variety of tools to create complex mobility scenarios that come closer to reality to many other synthetic mobility models. GPS map data provides the scenario setting and node groups with numerous different parameters are used to model a wide variety of independent node activities and capabilities. ONE provides different node movement models and simulates several DTN routing algorithms such as Epidemic, Spray-and-Wait, PROPHET, etc.

The ONE simulator uses Java programming language and allows to add routing algorithms by extending the built in routing classes.

2 DTN DISCOVERY (IPND)

2.1 Introduction

DTNs make no presumption about network topology, routing or availability. DTNs therefore attempt to provide communication in challenged environments where, for instance, contemporaneous end-to-end paths do not exist. Example of such DTNs arise in a variety of context including mobile social networks, space communications, rural message delivery, military networks, etc. [IPND]

In such dynamic and not deterministic scenarios, the identity and meeting schedule of participating nodes is not known in advance. Therefore the ability to dynamically discover other DTN nodes becomes a key factor for routing and services purposes. The Internet Protocol Neighbor Discovery (IPND) is a standard specified in the DTN IP Neighbor Discovery draft published in 2012. In contrast to link and physical layer discovery, IPND enables a general form of neighbor discovery across a heterogeneous range of links, as are often found in DTNs. IPND is particularly useful in mobile, ad-hoc DTN environment where meeting opportunities are not known a priori and connections may appear or disappear without warning. For example, two mobile nodes might come into radio distance of each other, discover the new connection, and move data along that connection before physically disconnecting.

In addition to neighbor discovery (i.e. contact discovery), it is often valuable to simultaneously discover services available from that neighbor. Example of DTN services include a neighbor's available Convergence Layer Adapters (CLAs) and their parameters (e.g. TCP CLA), available routers (e.g. Prophet), tunnels, etc. It is usually useful to decouple service discovery from neighbor discovery for efficiency and generality. For example, upon discovering a neighbor, a DTN node might initiate a separate negotiation process to establish 1-hop connectivity via a particular convergence

layer, perform routing setup, exchange availability information, etc.

IPND beacons thus optionally advertise a node's available services while maintaining the ability to decouple node and service discovery as necessary. This flexibility is important to various DTN use scenarios where connection opportunities may be limited (thus necessitating an atomic message for all availability information), bandwidth might be scarce (thus implying that service discovery should be an independent negotiation to lower bacon overhead), or connections have very large round trip times.

2.2 IPND protocol

An IPND beacon is a small UDP message in the IP underlay that advertises the presence of a node and optionally its available services (such as routers or convergence layers). A beacon can be sent as either IP unicast, multicast or broadcast UDP packet.

2.2.1 Broadcast, multicast and unicast beacons

Broadcast beacons are designed to reach unknown neighbors in the local network (within the boundaries where the broadcast packet transmission is limited). Multicast beacons extend the scope of beacon dissemination to include different networks across routed boundaries. Unicast beacons are sent only to explicitly known and enumerated neighbors.

Upon discovering a neighbor and its services, a node can establish a connection to the new neighbor via an IP-based Convergence Layer Adapter.

Generally the IP address of a potential neighbor is not known in advance. In this case, IPND beacons are sent to broadcast or multicast destination addresses. However, since multicast or broadcast discovery may not be always feasible over the Internet, the IP addresses of potential neighbors reachable only across multiple underlay hops must be explicitly enumerated for discovery. In fact, while the neighbor address is already known, its availability is not.

2.2.2 Beacon period

An IPND node should send beacons periodically. The time interval between beacon

transmission is configurable, and should be set to a reasonable value with respect to the network conditions. The beacon period should be advertised within the beacon itself so that any neighbor can use this information to determine the state of the sender.

2.2.3 Beacon format

The beacon message contains the following fields:

- Version: the version number of the IPND that construct the beacon. This version field is incremented if either the IPND protocol is modified or the Bundle Protocol version is incremented. In this way the field can also used to determine the BP version supported by a potential DTN neighbor.
- Flags: four flags are currently defined:
  - Source EID present: indicates that the source node EID is advertised in the beacon. (this flag should always be set).
  - Service block present: indicates that a service block is present.
  - Neighborhood Bloom Filter (NBF) present: indicates that a NBF is present within the service block.
  - Beacon period present: indicates that the beacon period is advertised.
- Beacon sequence number: integer field incremented once for each beacon transmitted to a particular destination address.
- EID length: the length of the beacon source canonical EID.
- Canonical EID: the canonical end node identifier of the neighbor advertised by the beacon. It is represented as a Uniform Resource Identifier.
- Service block: optional announced services in the beacon.
- Beacon period: optional field indicating the sender's current beacon interval in seconds. A value of zero means that the beacon period is undefined.

Service block

A beacon can optionally include a service block used to advertise service availability on

the sender node. While the service block is intended to contain representations of available CLAs, routers, a NBF, etc., it can also accommodate implementation specific services provided by the advertising node.

In fact, while the source IP address of the beacon is sufficient to identify the neighbor at the IP level, it cannot inform via which transport mechanism (TCP or UDP) or via which transport port the neighbor is offering a connection. Likewise, nodes do not know which routers are running on a remote node. Therefore, a beacon may contain a service block which serves to notify nodes about the availability of these services.

Service definition

IPND uses Tag Length Value (TLV) encoding scheme to define the advertised services. This provides for the standardization of services definitions using a format that focuses on simplicity, flexibility and efficiency. IPND-SD-TLV structures are composed by three parts:

- Tag: a numeric token which identifies the structure.
- Length: a numeric value which specifies the size of the content block.
- Value: the content block, which contains the value(s) described by the tag.

The detailed composition and usage of the IPND-SD-TLV structures is described in [IPND].

Services

A service is an IPND-SD-TLV structure that represents an advertisement for a DTN-related resource available on the beacon source node. Each service type has a unique tag number in order to identify it within the service block.

An IPND node must support the service definitions for TCP-CLA-v4 and UDP-CLA-v4; that is, a node must support the standard definitions for TCP CLA advertisements and UDP CLA advertisements, respectively. The structure of these service definitions contains the IP address and the transport port via which the advertising node is accepting a connection or receiving packets. Moreover, a IPND node may support the TCP-CLA-v6, CLA-UDP-v6, TCP-CLA-HN, UDP-CLA-HN, that is a node may

support the definition for TCP and UDP CLAs where the address is indicate as either an IPv6 address or a string (hostname).

Finally a node may support any implementation-specific service.

Neighborhood Bloom Filter

Many routing protocols work correctly only when links are bidirectional. While in wired IP networks link bi-directionality can often be presumed, this is not true for other type of networks, such as Mobile Ad-Hoc Networks (MANETs). In fact if a node receives beacon from a neighbor over a wireless medium, it is not generally safe to assume that the link is bidirectional. MANETs often have links that are only unidirectional due to differences in antennas, transmit power, hardware variability, multi-path effects, etc.

In order to efficiently determine link bi-directionality, a node represents the set of its 1-hop neighbors using a Bloom Filter, referred to as the Neighborhood Bloom Filter (NBF). Upon receiving a beacon from a neighbor that contains NBF service information, a node can quickly determine whether it is in the neighbor's NBF set, and thereby determine whether the link is bidirectional.

The detailed description of the NBF service definition is depicted in [IPND].

2.2.4 Disconnection discovery

An IPND node should maintain state over all existing neighbors, that is maintaining a current neighbor set. When IPND discover a new neighbor, it adds it to the current neighbor set. Likewise, IPND removes from the set stale neighbors after the defined neighbor receive timeout period elapses without receiving any beacon messages from a particular neighbor.

Upon detecting that a neighbor is no longer available, IPND may informs the CLAs that the neighbor is gone.

2.3 ION IPND implementation

ION IPND implementation has been developed in 2015 and distributed starting from the

ION 3.4.0 release. The ION IPND module is a daemon that manages beacon sending and reception. This module allows the node to send and receive beacon messages using unicast, multicast or broadcast IP addresses. Beacons are used for the discovery of neighbors and may be used to advertise services that are present and available on nodes, such as routing algorithms or CLAs.

The following ION IPND description is taken from the official ION IPND man page that I wrote. [ION]

ION IPND module is configured using a *.rc configuration file. The name of the configuration file must be passed as the sole command-line argument to the `ipnd` command when the daemon is started. Commands are interpreted line by line, with exactly one command per line. The formats and effects of the ION IPND management commands are described below.

Usage

```
ipnd config_file_name
```

Commands

- `1`

  The initialize command. This must be the first command.

- `#`

  Comment line. Lines beginning with # are not interpreted.

- `e` *{ 1 | 0 }*

  Echo control. Setting echo to 1 causes all output printed by IPND to be logged into *ion.log*. Setting echo to 0 disables this behavior. Default is 1.

- `m eid` *eid*

  Local EID. This command sets the advertised BP endpoint ID by which the node will identify itself in beacon messages.

- `m announce period` *{ 1 | 0 }*

Announce period control. Setting to 1 causes all beacons messages sent to contain beacon period. Setting to 0 disables this behavior. Default is 1.

- `m announce eid` *{ 1 | 0 }*

  Announce EID control. Setting to 1 causes all beacons messages sent to contain source EID. Setting to 0 disables this behavior. This should be always set to 1. Default is 1.

- `m interval unicast` *interval*

  Unicast interval. This command sets the beacon messages period on unicast transmissions. Time interval is expressed in seconds. Default is 5.

- `m interval multicast` *interval*

  Multicast interval. This command sets the beacon messages period on multicast transmissions. Time interval is expressed in seconds. Default is 7.

- `m interval broadcast` *interval*

  Broadcastcast interval. This command sets the beacon messages period on broadcast transmissions. Time interval is expressed in seconds. Default is 11.

- `m multicast ttl` *ttl*

  Multicast time-to-live. This command sets the multicast outgoing beacon messages' time to live, in seconds. Default is 255.

- `m svcdef` *id name child_name:child_type ...*

  Service definition. This command specifies definitions of "services", which are dynamically defined beacon message data structures indicating the capabilities of the beacon message sender. *id* is a service-identifying number in the range 128-255. *name* is the name of the service type that is being defined. The definition of the structure of the service is a sequence of elements, each of which is a *name:type* pair. Each *child_type* must be the name of a standard or previously defined service type. Infinite recursion is supported.

- `a svcadv` *name child_name:child_value ...*

Service advertising command. This command defines which services will be advertised and with which values. All types of formats for values are supported (e.g. 999, 0345 (octal), 0x999 (hex), -1e-9, 0.32, etc.). For a service that contains only a single element, it is not necessary to provide that element's name. E.g. it is enough to write *Booleans:true* instead of *Booleans:BooleanValues:B:true*, as *BooleanValues* is the only child of *Booleans* and *B* is the only child of *BooleanValues*.

- `a listen` *listen_socket_spec*

  Listen socket specification command. This command asserts, in the form *IP_address:port_number*, the specification for a socket at which the IPND daemon is to listen for incoming beacons. The address can be an unicast, a multicast or a broadcast address. If a multicast address is provided all the configured unicast addresses will listen for multicast packets in that group. If a broadcast address is provided all the unicast addresses will listen for broadcasted packets.

- `a destination` *destination_socket_spec*

  Destination socket specification command. This command asserts the specification for a socket to which the IPND daemon is to send beacons. It can be an unicast, a multicast or a broadcast address.

- `s`

  The start command. This command starts the IPND daemon for the local ION node.

Examples

```
m scvdef 128 FooRouter Seed:SeedVal BaseWeight:WeightVal
RootHash:bytes
```

Defines a new service called FooRouter comprising 3 elements. SeedVal and WeightVal are user defined services that must be already defined.

```
m svcdef 129 SeedVal Value:fixed16

m svcdef 130 WeightVal Value:fixed16

m svcdef 128 FooRouter Seed:SeedVal BaseWeight:WeightVal
RootHash:bytes

m svcdef 150 FixedValuesList F16:fixed16 F32:fixed32
F64:fixed64

m svcdef 131 VariableValuesList U64:uint64 S64:sint64

m svcdef 132 BooleanValues B:boolean

m svcdef 133 FloatValuesList F:float D:double

m svcdef 135 IntegersList FixedValues:FixedValuesList
VariableValues:VariableValuesList

m svcdef 136 NumbersList Integers:IntegersList
Floats:FloatValuesList

m svcdef 140 HugeService CLAv4:CLA-TCP-v4
Booleans:BooleanValues Numbers:NumbersList FR:FooRouter

a svcadv HugeService CLAv4:IP:10.1.0.10 CLAv4:Port:4444
Booleans:true FR:Seed:0x5432 FR:BaseWeight:13
FR:RootHash:BEEF Numbers:Integers:FixedValues:F16:0x16
Numbers:Integers:FixedValues:F32:0x32
Numbers:Integers:FixedValues:F64:0x1234567890ABCDEF
Numbers:Floats:F:0.32 Numbers:Floats:D:-1e-6
Numbers:Integers:VariableValues:U64:18446744073704783380
Numbers:Integers:VariableValues:S64:-4611686018422619668
```

This shows how to define multiple nested services and how to advertise them.

3 CGR INTEGRATION INTO ONE

The Contact Graph Routing algorithm code is contained in the ION distribution, developed and maintained mainly by Scott Burleigh at NASA Caltech Jet Propulsion Laboratory. The CGR logic is coded into the file *libcgr.c* and we wanted to integrate this file into the ONE simulator with the minimum modification. In fact, like all libraries, the CGR library is constantly evolving, and we do not want to modify the CGR code in the ONE simulator every time a new CGR version is released. Moreover, translating the whole CGR written in C into a brand new Java module would necessarily introduce some unwanted differences with respect to the original, such as new bugs. Therefore, we preferred to integrate the original ION CGR library into ONE using Java Native Interface.

The code in *libcgr.c*, as most of the code of ION, uses many library functions whose purpose is to provide an abstract, reliable, optimized and platform independent view of operative system's resources, such as memory and persistent storage, and an easy to use set of structures, such us linked lists and red and black trees. These libraries are initialized at ION startup and are used by every ION module.

Our aim was to integrate CGR into ONE using the *libcgr.c* file as it is. However, this code has been designed to run in an ION environment, with all the utility libraries available; by contrast, into ONE we lack all of them. To use the actual ION environment in the simulation would be a waste of resources. Instead, we decided to simulate the ION environment in ONE, so that the *libcgr.c* code can run as it was in ION.

3.1 The ICI package

3.1.1 The lyst library

The first library we need to simulate in order to allow *libcgr.c* to run in ONE is the *lyst* library. This library provides a set of functions for manipulating generalized doubly

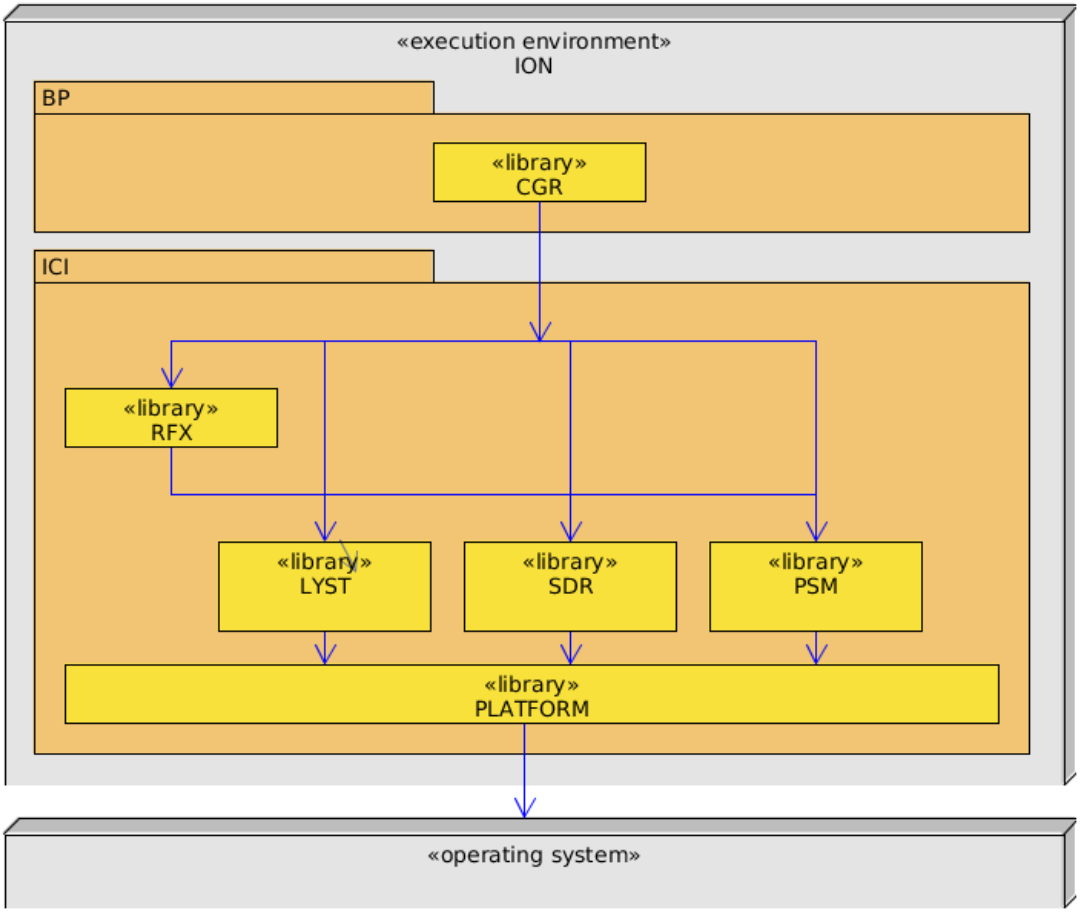

Image 1: Original ION libraries scheme

linked lists. Since ONE already works in Java and since this environment provides classes for manipulating doubly linked lists, we decided to use these classes for simulating the lyst library through JNI function calls.

Lyst function declarations are in the *lyst.h* header file, implementations in *lyst.c* . In our simulated environment we can use the same header file, as the CGR library calls the lyst functions based on the signatures described there, but we need to change the implementation in order to use JNI to access to the actual lists managed by the Java virtual machine (JVM). The lyst library defines two structures: *Lyst* and *LystElt*; the former contains information about the list, the latter contains a pointer to the list to which is belongs and a pointer to the generic data to which it refers. These structures are

never directly accessed from the *libcgr.c* code, but always from lyst functions. Thanks to that we could redefine these structures as *jobject*, a JNI type used for generic Java objects.

On the Java side of the lyst library simulation, we defined two classes: *cgr_jni.lyst.Lyst* and *cgr_jni.lyst.LystElt*. We can compare them to the respective lyst original structures. The *LystElt* class only contains a reference to the list that owns the element, and a pointer to the memory area where the data this element refers to resides. The pointer to the data is converted from *void** to *long*. The *Lyst* class contains a *java.utils.LinkedList* object that stores the actual list and several static methods for manipulating the list. The lyst library permits to set a custom delete function invoked on element deletion in order to allow the user to free the memory used by the object referred by the deleted list element. This behavior is recreated in our Java implementation saving a reference to the delete function in the Java *Lyst* object. The static methods in the *Lyst* class have the same signature as the functions declared in *lyst.h*.

We did not need to implement all the functions of the lyst library but only those used by *libcgr.c*. For this reason we did not implement the list sorting mechanism. In our *lyst.c* file basically every function wraps a JNI call to the respective static method of the *Lyst* class.

The lyst library could have been alternatively developed without JNI by adapting the original *lyst.c* file; this would have been a simpler solution. However, the method chosen made me get started with JNI use and I think that the big effort was worthwhile.

3.1.2 The PSM library

An instrumental library in the ION environment is the PSM (Personal Space Management) library, which provides functions that support high level memory management and memory partitioning. PSM is designed to be faster and more efficient than standard `malloc()/free()` and provides a memory management abstraction that insulates applications from differences in the management of private versus shared memory. This aims to enhance the portability of the code to those systems where separate tasks using a shared memory partition are given different base addresses with

which to access the partition, such as Solaris. Basically PSM provides function for translating absolute memory addresses to partition based offset addresses and vice versa. It also provides a useful catalog abstraction that maps strings into addresses in a partition. The catalog abstraction allows the user to save a memory address with a string as key and later to locate the address by using the key. [ION]

In our case code portability and memory efficiency are not essential, therefore for our use the standard `malloc()/free()` functions are adequate. However, partition abstraction and catalog functionality are still necessary to allow several ION nodes to run simultaneously on a unique host machine. Since an ION node uses a PSM partition catalog to store working structures, we need a different PSM catalog for every simulated node.

As done for the lyst library, we decided to use the original header file and to modify the source file. Likewise, we decided to use the Java collections support to implement the PSM partition management. Thus we created the *cgr_jni.psm.PsmPartition* class, which contains a set of addresses and a map of entries (name, value) where the name is a string and the value is an address. Addresses are C pointers, converted into *Long* Java type. This class provides a method to add addresses to the address set and entries to the catalog, or to remove them, and a method to retrieve an address from the catalog given its name. Since a node can have multiple PSM partitions, we created the *PsmPartitionNodeManager* class, which contains a map of entries (*id*, *PsmPartition*) and methods to create, destroy and manage partitions. This class describes the PSM partitions status of a single simulated node. Then we created a utility class *PsmPartitionManager* with static functionality. This class contains a map of triples (*id*, *PsmPartitionNodeManager*) where the id is the node number and provides methods for creating, destroying and retrieving PSM partitions globally. Thus, every partition has an id that identifies it within the partitions of the node and every node partition set is identified globally by the node number.

On the C side we redefined the *PsmPartition* structure as a *jobject* that actually point to a *PsmPartition* Java object. Then in the *psm.c* we implemented the PSM functions in order to wrap JNI calls to *PsmPartition* class methods. Every time a memory area is

allocated within the partition, its pointer is saved into the PSM partition object and likewise for memory deallocations and catalog operations. When a memory area is freed, the catalog entry is removed. In this implementation we used standard `malloc()` and `free()`, as memory management operations and the pointer to partition offset conversion are just type casts.

3.1.3 The smlist and smrbt libraries

The *smlist* library provides functions to create, manipulate and destroy doubly linked lists in shared memory. Like the *lyst* library does, *smlist* uses two types of object: list objects and element objects. However, as these objects are stored in shared memory, which is managed by the PSM library, pointer to these objects are carried as *PsmAddress* values. Basically this library provides the same functionality as *lyst,* but it uses PSM managed shared memory. For this reason, it also provides mutex to build thread safe lists. [ION]

The *smrbt* library provides functions to create, manipulate and destroy red-black balanced binary trees in shared memory. *Smrbt* uses two types of objects: *Rbt* objects and *Node* objects. As these objects are stored in shared memory, which is managed by PSM, pointers to these objects are carried as *PsmAddress* values. [ION]

Since we already had our PSM implementation, we were able to use the original *smlist.c* file. The ONE simulator is a single-thread environment, therefore we could get rid of the complexity added by list mutex management by converting mutex functions to no op in the *platform_sm.c* file.

3.1.4 The SDR library

The SDR (Simple Data Recorder) library contains functions that support the use of an abstract data recording device for persistent storage of data. The SDR abstraction insulates software not only from specific characteristics of any single storage device but also from some kinds of persistent data storage and retrieval chores. The underlying principle is that an SDR provides standardized support for user data organization at object granularity, with direct access to persistent user data objects, rather than supporting user data organization only at file granularity and requiring the user to

implement access to the data objects contained within those files. [ION]

As in our simulations we do not need to use persistent storage and the SDR functionality, the simulation of the PSM library is enough. SDR provides a set of function to work with lists and catalogs exactly as PSM does. In addition, since SDR manages the persistent storage and input/output operations may be complex, it offers a transaction mechanism to protect data integrity across a series of reads and writes. Thus every transaction management function has been transformed in no-op and every other one has been redirected to the respective PSM function. To do so the *sdrxn.h* header file has been changed and the types *Address* and *Object* used by SDR have been both redefined as *PsmAddress* and the type *Sdr* as *PsmPartition*. The file structure is the same as in the official ION distribution, but all source files have been modified in order to implement the changes previously described.

3.1.5 The RFX library

The RFX library contains functions to manage ION contacts and ranges. This library uses PSM, SDR and Lyst functions. Since we have implemented a simulated version of these libraries, the *rfx.c* and *rfx.h* files can be imported from the ION release without any changes.

3.1.6 Utilities

The ICI package implements also a small set of utilities to support real-time interaction with the ION environment. The main utility is the standalone program *ionadmin* to manage contacts and ranges. It can be used by inserting commands either from prompt or a configuration file. Since a contact plan must be provided to every node in the simulation, *ionadmin* must be used to initiate their status. Basically *ionadmin* performs the command parsing and the invocation of the correct RFX function for adding/deleting/listing contacts and ranges. Because in our simulation contact and range information are not stored in persistent but in volatile memory, *ionadmin* must be used within the simulator. For this purpose we can use the `runIonAdmin()` function, to read a contact plan from a file, or the `processLine()` function, to execute just one command. These functions are in *ionadmin.c*, which is identical to the original but small

compilation fixes.

The *utils.c* file contains some initialization and finalization functions used by *ion.c*. The ION runtime structures initialized here are *IonDB* (Ion database) and *IonVdb* (Ion volatile database). The former is stored in the SDR space (persistent) and contains information about the contact plan and ranges; the latter, is in the PSM space and contains the same information as *IonDB,* but differently organized in order to provide a faster access speed. Moreover, the *utils.c* file contains functions for managing the Java *PsmPartition* objects used by each ION node, as *Sdr* and *PsmPartion*.

3.2 The BP package

The only part of the ION BP package (the part of ION that contains Bundle Protocol related functions)that is necessary to import in ONE is the CGR library, implemented in the *libcgr.c* file. Our primary aim is to make possible the use in ONE of the original ION file without any modification. This is a fundamental requirement to avoid by root all the problems related to a parallel implementation in Java: code inconsistency and maintenance cost.

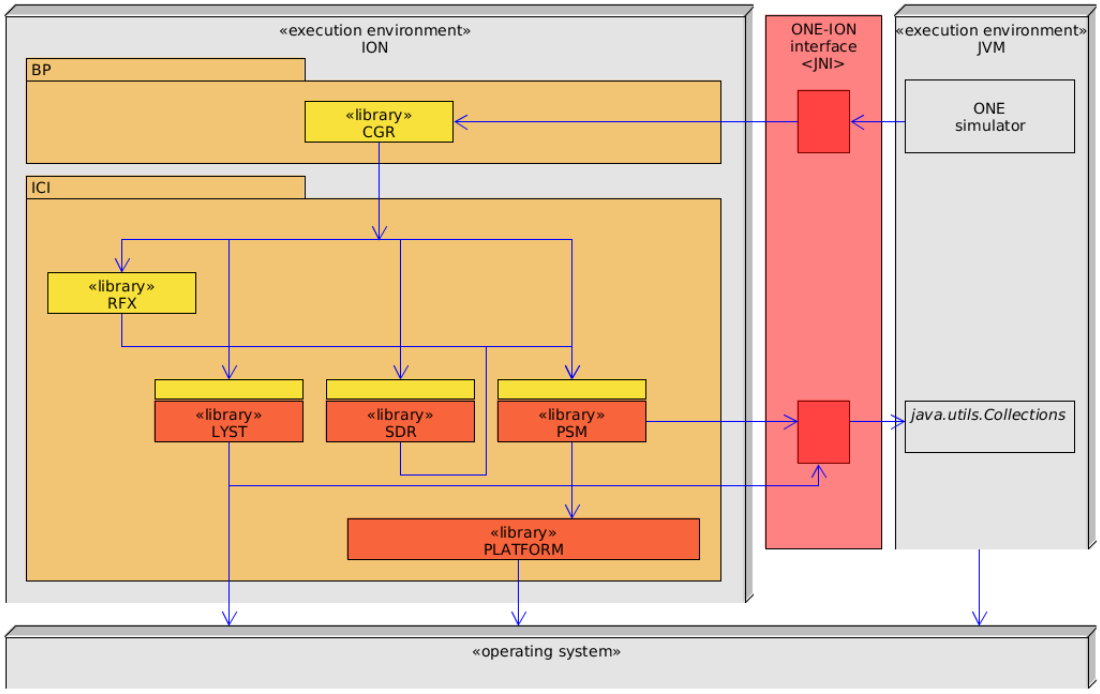

Image 2: Adaptation of ION libraries. Red blocks are the modified ones.

The *libcgr.c* file requires a set of header files, all in the *include* directory. Moreover, it calls several functions of *libbpP.c* file; these functions are supposed to interface CGR with the ION runtime environment, therefore, we need to redirect them to ONE simulator runtime environment.

3.3 The ONE to ION interface

The ONE to ION interface code is contained in the *jni_interface* directory. This code contains the entry point functions for ONE to interact with the simulated ION environment and vice versa. It also provides functions to manage multi threading operations and initialization/finalization of ION nodes.

The result of the compilation of our ION adaptation is a shared library that can be used by the Java VM hosting the ONE simulator. Before performing route calculations, the library needs to be initialized both globally, in order to provide a consistent runtime environment, and locally, i.e. for each node.

This library has to provide entry points for the ONE simulator running in the Java VM to use the ION adaptation. The entry points are native functions invoked by the Java code.

3.3.1 Global initialization

Even if the ONE simulator is single-threaded, the ION adaptation library has been designed to work in a multi-threaded environment, assuming that each node runs on a single thread at a time. This can be useful if ONE will eventually provide support to multi-threaded simulations. Every information about the local node is thus stored in special thread-specific data area: thread-specific variables are managed by the `pthread_setspecific()` and `pthread_getspecific()` function of the pthread library. Each thread-specific variable is identified by a key stored in a global variable.

There are 3 thread-specific keys defined in *shared.h*:

- *nodenum_key* the local node number. It is updated every time ONE invokes an ION function, i.e. in every native function called by Java.

- `j`*niEnv_key* a reference to the current JNI environment. It is updated every time a native function is called by the Java environment and it is used to invoke JNI methods from native code (such as `psm_locate()`).
- *interfaceInfo_key* this key refers to a structure containing informations that needs to be local to the node but global with respect to the function invocations.

The *init_global.c* source file contains functions that manage the global-level and node-level initialization and the simulated time.

ION uses the system time as a reference and it works in real time, while ONE, being a simulator, uses a non real time reference where time 0 is the simulation start instant. Unless we want to set all our experiments back in the '70s (the origin of the POSIX time) we need to convert the ONE time to the ION time. To do so when the ION environment is initialized, the current system time is stored in a global variable and its value is taken as reference as the ONE simulation start time. Every ONE time is thus considered as an offset with respect to this reference time. Fortunately, *libcgr.c* uses the function `getUTCTime()` in *ion.c* to get the current system time, so it is enough to modify the call to `getSimulatedUTCTime()` in *init_global.c*, which returns the current simulated time from ONE via a JNI call.

3.3.2 Node initialization

To understand the interface between ONE and ION it is important to know how simultaneous operations on multiple nodes are supported. Each ION node uses one SDR partition and one PSM partition. Since our SDR implementation basically relies on PSM, we can conclude that each ION nodes uses 2 PSM partitions. These partitions are managed by the Java class *PsmPartitionManager* and each node must maintain a reference to both partitions. The file *bp/utils.c* defines the structure *IonPartitions* as that:

```
typedef struct
{
     PsmPartition partition[2];
     uvast nodeNbr;
} IonPartitions;
```

This structure contains the number of the local node and the two PSM partitions used by the node as a reference to the *PsmPartition* Java objects. The Java class *PsmPartitionManager* has a global knowledge of all partitions of all nodes. Thus to get either the PSM or SDR partition of a node, the Java partition manager needs to be invoked via JNI. Since the invocation of a JNI method carries considerable overhead, the partitions of the current local node are stored it the *IonPartitions* structure. The *utils.c* source file contains the functions for managing this structure.

The ION code and in particular the *libcgr.c* code retrieves the working partitions using the functions `getIonwm()` for the PSM partition and `getIonsdr()` for the SDR partition, respectively. Both of them are defined in *ion.c*, thus we have redefined them to use `getIonWm()` and `getIonSdr()` in *ici/utils.c* instead. These functions use the partition informations cached in the *IonPartitions* structure to return the needed ION partition. Every time a node partition is requested, if the current node number stored in the thread-specific space is different from the one registered in the *IonPartitions* structure, the whole structure is updated to contain the current local node partitions.

The main reason for having those two partitions is to store and catalog the *IonDB* object in the SDR partition and the *IonVdb* object in the PSM partition. These two objects are both initialized during the node initialization. The functions to initialize, finalize, and retrieve the objects at runtime are defined in *ici/utils.c* .

3.3.3 Java entry points

The Java class *cgr_jni.Libcgr* declare a series of native methods used as entry points to the ION library. All of these methods are defined in the source file *cgr_jni_Libcgr.c*. Each function performs environmental checks and updates before invoking the actual working function. In brief, each function updates the current node number reference in the thread-specific space, so that every subsequent invocation of partition related or node management functions uses the right structures. In addition to that, each function updates the thread-specific JNI environment reference, so that every subsequent JNI call from native code uses the right reference.

The entry points are used for node initialization and finalization, contact plan updating

and contact graph route calculation.

3.3.4 ONE to ION interface functions

The file *ONEtoION_interface.c* contains several functions that support information exchanges from the Java runtime to the ION library and vice versa. The functions defined in this file retrieve information from the ONE environment, such as bundle source and destination, creation time, time-to-live and payload size. In addition to that it supports CGR specific bundle fields that have been recreated in ONE as *Message* properties, such as *deliveryConfidence* and *xmitCopies.* There are also functions that support outducts information exchanges. All these functions contain JNI calls to specific methods of the *cgr_jni.IONInterface* class.

In this file are also defined the functions that convert Java *Message* objects into C *Bundle* structures and Java *Outduct* objects into C *Outduct* structures. In these conversions only the structure fields used by CGR are set with sensible values; every other field is set to 0.

ONEtoION_interface.c also contains a set of functions and a structure that manage the workflow of the CGR simulation.

The structure *InterfaceInfo* is defined as follow:

```
struct InterfaceInfo_t {
     jobject currentMessage;
     Object outductList;
     Object protocol;
     int forwardResult;
};
typedef struct InterfaceInfo_t InterfaceInfo;
```

This structure contains information whose scope is a single `cgrForward()` invocation. The *currentMessage* variable contains a reference to the Java *Message* that needs to be forwarded. This reference is needed in case the bundle should be enqueued in an outduct or put into limbo. The *outductList* contains a SDR list filled by CGR. This list is needed because the code in *libcgr.c* pretends to have outduct references stored in a SDR list; since we do not want to change the CGR code, we have to recreate that list

and make it available to CGR. The *protocol* variable contain a reference to a *ClProtocol* structure. ION uses this structure to store protocol information, such as overhead per frame, frame size and nominal transmission rate. The CGR library does not use this information but still does a null check on the outduct protocol variable, so a dummy *ClProtocol* structure is created and referenced into the outducts structures. The reference is stored into the *InterfaceInfo* structure so that every outduct can reference the same object. The *forwardResult* variable is an integer that indicates the outduct that the current bundle has been forwarded to.

3.3.5 CGR work flow

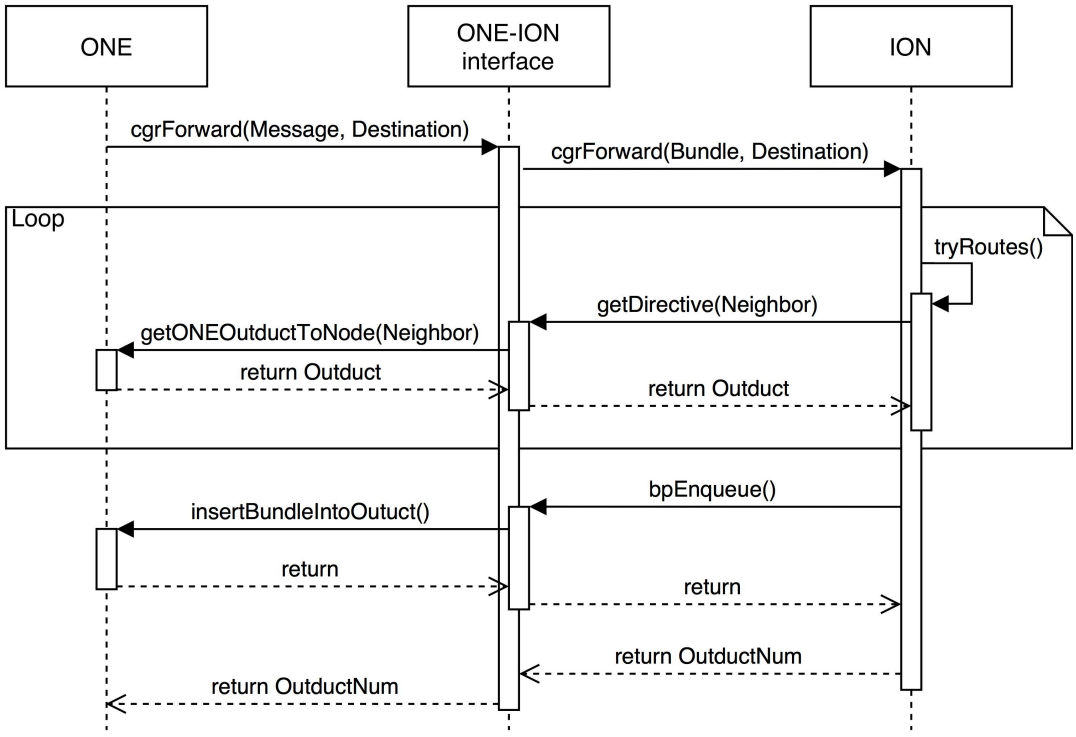

Image 3: sequence diagram of a cgrForward() routine (function names and signatures have been renamed for a better reading)

When a simulated node needs to forward a bundle, it calls the `cgrForwardONE()` function via the entry point previously described. This function performs the initialization of the *InterfaceInfo* structure and the conversion of the *Message* Java object to a *Bundle* C structure. Subsequently, it invokes *libcgr.c* `cgr_forward()` passing the converted bundle as a parameter; it also passes as a parameter a function

pointer to `getONEDirective()`. This function is defined in the *ONEtoION_interface.c* file and it is used by *libcgr.c* to retrieve the outduct reference from ONE. If CGR succeeds in finding a route to destination for the current bundle, it invokes `bpEnqueueONE(),` which performs a JNI call to the Java method that enqueues a bundle into an outduct in ONE. It also updates the route expiration time, i.e. the time until the node tries to forward the bundle to the selected neighbor; if the bundle cannot be sent within the expiration time, the route must be recalculated. The `bpEnqueueONE()` function stores in the *InterfaceInfo* structure the number of the outduct to which the bundle has been enqueued and, finally, `cgrForwardONE()` returns this value (or 0 if the bundle is in limbo). This is an easy way to provide ONE with a result that indicates if a route has been eventually found or not and the outduct that the bundle has been enqueued to. The whole ION adaptation work flow is based on the assumption that every simulated node puts each bundle into limbo before invoking CGR. In this way, if no route is found and `bpEnqueueONE()` is not called, the bundle is already in limbo and no further operations are needed.

4 OPPORTUNISTIC CGR

4.1 Motivations

The main purpose of CGR in ION is to calculate deterministic routes for bundles. ION indeed was developed mainly to deal with deep space networks, where contacts between nodes (orbiting spacecraft, satellites, landers and rovers) are well known in advance. Since DTN architecture should also work with all the challenging networks where deterministic routing is not a choice, such as terrestrial mobile networks, ION needs to provide a mechanism to perform probabilistic and opportunistic routing if it wants to cover also these scenarios.

The current CGR implementation (ION 3.4.1) supports probabilistic contacts that have to be inserted manually into the contact plan. A probabilistic contact is a contact which probability (confidence) is less than 1.0. CGR, while computing the routes, takes into account the probability of each route hop and based on the resulting route probability (delivery confidence) it decides whether forward the bundle on one route only or to replicate the bundle and send it via multiple routes. This feature can be exploited to implement Opportunistic CGR by providing an algorithm that automatically fills the contact plan with predicted contacts.

There are many opportunistic routing algorithm in literature [caini2011]: from the basic ones (Epidemic, Spray and wait), which are basically controlled flooding mechanism, to the smartest ones, like PROPHET. The Opportunistic Contact Graph Routing algorithm wants to be an extension of the classic CGR, therefore it uses the same concept of contacts and uses the same route calculation algorithm based on Dijkstra search. The main difference with respect to classic CGR is that the contact plan can contain non-deterministic contacts (i.e. contacts with confidence less than 1.0), and that these contacts are automatically inserted in the contact plan by a contacts prediction algorithm

that tries to guess the next contacts based on previous encounters history. The base idea is that the more often a node had a contact with a specific neighbor, the more likely it is going to encounter it again; this assumption is at the basis of other opportunistic routing algorithm, like PROPHET. In addition to that, the prediction algorithm tries to guess the start time of next contacts and their capacity. This is necessary because we do not want to modify the route calculation mechanism based on contact plan, thus even if the predicted contacts can never have a confidence of 1.0, we need to insert them in the contact plan with fixed start and end times and transmission rate.

4.2 The algorithm

The OCGR algorithm has been firstly conceived by Scott Burleigh; then, I contributed to its design during its implementation in ION by means of a continuous and intense exchange of ideas with its inventor. Datasets analysis have shown that contacts properties of a pair of nodes are not completely random, but most of the times they follow a certain probability distribution. The contact properties we are interested in are contact duration (interval between start and end instant of a contact), contact gap (interval between the end instant of a contact and the start instant of the next one) and the nominal transmission rate. On the base of previous contacts history, we can find the mean and the standard deviation of those contact properties for a pair of nodes. The mean value of the previous contacts properties will be used as the actual properties of the predicted contacts while the standard deviation give us an indication of the history randomness: with a low standard deviation we can say that the contacts history for a pair of node is likely following a pattern, so we can have a higher confidence on the predicted contacts. On the other hand, if the standard deviation is high, we can assume that the history is kind of random, thus we can attribute a low confidence to the predicted contacts. The actual link between mean value and variance depends form the kind of distribution; here we have assumed that those two values can be related, thus the threshold chosen to establish if the standard deviation is high or low is the mean value itself: i.e. if the standard deviation is lower than the mean the confidence of the predicted contact will be higher.

We also want to take into account the number of previous contacts while calculating the

confidence of a predicted contacts series. In fact, when we have a short contact history, the mean and standard deviation values are less significant, therefore the confidence of a predicted contact has to be low. Therefore, in order to compute the final confidence of a predicted contact, we define the following parameters for each sequence of contact history entries comprising all and only entries for some single sending node and some single receiving node:

- Base confidence: is the confidence we initially attribute to the series of predicted contacts. It can be high or low:
  - High base confidence: attributed when the contact duration standard deviation is less than the mean **and** the contacts gap standard deviation is less than the mean. It is temporarily defined as 0.2.
  - Low base confidence: attributed otherwise. This means that the history is random. It is temporarily defined as 0.05.
- net confidence: it is the final confidence of the predicted contacts. It is:

$$1.0 - (1.0 - \text{base confidence})^N$$

  where N is the number of contacts. This means that the more entries are in the contact history, the higher will be the predicted contacts confidence.

These parameters are just a first guess and no studies have been done to prove that they are somewhat acceptable, due to lack in time. In fact, all these parameters need to be empirically tuned, a work that will require effort and time before reaching the most appropriate values.

Guessing the correct confidence for a predicted contact is a key issue for OCGR performances. In fact if the contacts confidence is overestimated, OGCR will often find a high confidence route for a bundle and will enqueue it to a specific outduct without trying any alternative route. If the chosen route turns out to be wrong, the bundle will wast much time stuck in a dead end outduct, without the possibility to be forwarded via a better route. On the other hand, it the contacts confidence is underestimated, OCGR will often find a low confidence route for a bundle, thus it will enqueue it to multiple outducts, possibly causing network congestion and overhead.

We define “prediction horizon” for a pair of nodes the instant calculated as the current time plus the difference between the current time and the start time of the firs contact in the history log, related to that pair of nodes. This is the end time of our prediction: we only predict into the future as far as we can see into the past.

Therefore for each pair of nodes the contact prediction algorithm inserts into the contact plan several predicted contacts which durations, gaps, and capacity are the mean values of the previous registered contacts, until the prediction horizon is hit. The confidence of those predicted contacts is based on the calculated standard deviation of the previous contacts durations and gaps.

CGR then uses its existing probabilistic route calculation algorithm to decide which neighbor the bundle should be forwarded to.

4.3 The implementation

In order to support this new opportunistic routing algorithm the ION code needs to be changed. The main features we need to implement are the contact history log and the contact prediction algorithm. In addition other little modifications are needed to make the code consistent.

4.3.1 Confidence

The current ION version (3.4.1) uses the term “probability” to refer to non-deterministic contacts likeliness of happening. We have no theoretical basis to allocate a specific probability to any element of predicted contacts and route calculations, but we can freely assert that we feel a given level of confidence in each prediction. Therefore all references to “probability” in ION 3.4.1 have been changed to “confidence” in this experimental OCGR version. Also the *cgrBets*, *cgrBetsCount*, and *deliveryProb* fields in the *Bundle* structure have been renamed *xmitCopies, xmitCopiesCount* and *dlvConfidence*.

4.3.2 Database modifications

Contact Plan

The ION contact plan continues to reside in the ION database (the *IonDB* object in the SDR partition), listing anticipated intervals of contact between nodes and intervals of times when the distances ("ranges") between nodes are as noted.

A contact with confidence value less than 1.0 is termed "predicted contact". Predicted contacts can be added to the database manually via ionadmin as before, but they normally should be generated or deleted automatically by the contact prediction algorithm.

The contact automatically inserted and terminated by the "eureka" library, which acts as an interface between ION and the neighbor discovery daemon, always have confidence level 1.0; they are termed "discovered contacts" and can be identified as such by the value of the *discovered* flag, newly added to the *IonContact* and *IonCXref* structures. The stop time of a discovered contacts is initially set to MAX_POSIX_TIME.

Contact history

New *contactLog* (contact history) lists are added to the ION database, one for the contacts reported by the sending node and one for the contacts reported by the receiving node, including all completed discovered contacts that the current node has personally experienced or that have been reported to it by other nodes. The contact history log contains only discovered contacts that have already terminated. OCGR want to list only known facts in the contact history; the contact history is the base for the contact prediction algorithm and since the prediction result is probabilistic by construction, OCGR does not want to add any level of uncertainness in the prediction base that would dramatically lower the prediction confidence. The only tolerated uncertain value is the stop time of a discovered contact. In fact, while the start time of a discovered contact is certain, as identified by the neighbor discovery in the same moment for both the sender and the receiver nodes, the stop time can be different between the two nodes as often identified by a timeout expiration of the neighbor discovery daemon. We consider the stop time reported by the sender node as more accurate than the one reported by the

receiver node; this is the reason why we implemented the contact history as a double list: the entries in the sender list have higher priority than the ones in the receiver list. An entry in the receiver list is used for the prediction if and only if there is not the corresponding entry in the sender list.

In order to facilitate read and write operations within the contact history log, entries in each list are sorted by:

- Sending node (ascending)
- Receiving node (ascending)
- Contact start time (ascending)

Contact history administrative record

A new contact history administrative record is defined. Its data are two sequences of contact history entries: all entries in the SENDER *contactLog* list, followed by all entries in the RECEIVER *contactLog* list, followed by all discovered contact currently in the contact plan other than the contact with the node to which the record is sent.

This administrative record is not yet implemented and will be developed as soon as the simulations confirm that OCGR can be a valuable opportunistic routing strategy.

4.3.3 Library modifications

The RFX library

The RFX library manages the contact insertion and deletion and now it has been modified to manage the contact history log and to implement the contact prediction mechanism as well. Therefore this is the library that has undergone the biggest modifications.

The new function `rfx_discovered_contacts()` is added; it removes every discovered contact in the contact plan that constitutes a contact with the indicated peer node.

Whenever `rfx_insert_contact()` is called, the new contact is checked for overlap with an existing contact. If the new contact's confidence level Is 1.0 (managed

or discovered), every predicted contact with which it overlaps is automatically deleted. This means that every insertion of a discovered contact will erase all predicted contacts for the affected sender/receiver, because the discovered contact's end time is MAX_POSIX_TIME, thus it overlaps with everything by definition. Any other overlap causes the new contact to be discarded rather than inserted.

Whenever `rfx_remove_contact()` removes a discovered contact whose sender/receiver node pair includes the local node, the new function `rfx_log_discovered_contact()` is called; the function adds a contact history list entry. Whenever any discovered contact is deleted, the new function `rfx_predict_contacts()` is called for the affected nodes pair: the discovered contact that caused the pair's predicted contacts to be removed due to overlap when it was inserted is now gone, so predicted contacts for this node must now be reinstated.

New functions `rfx_predict_contacts()` and `rfx_predict_all_contacts()`, described later, are added.

The EUREKA library

The eureka library, which provides functions supporting contact discovery, has been modified to implement operations triggered by a neighbor discovery. In particular, whenever the eureka library adds a new egress plan, it triggers the generation and transmission of the contact history administrative record to that neighboring node.

Whenever the eureka library discovers that a contact from the current node to another node has been lost, it passes that node to the new function `rfx_remove_discovered_contact()` noted above: because the current node is no longer in contact with that node, it has also lost the knowledge about other nodes with which it is in discovered contact.

The information exchange between two neighbors actually has not been implemented yet; it will be implemented as soon as simulations confirm that OCGR can be a valuable opportunistic routing strategy.

libbpP.c

The processing of administrative records has been modified: when a contact history administrative record is received:

- Every discovered contact in that record is inserted into the contact plan with stop time set to MAX_POSIX_TIME.
- Every contact history entry in that record that is not already included in the node's corresponding *contactLog* list is inserted into that list.
- The `rfx_predict_all_contacts()` function of *rfx.c* is invoked.

Contact prediction

The new `rfx_predict_all_contacts()` function performs the actions listed below. Note that `rfx_predict_contacts()` does the same, but only for a single sender/receiver pair, i.e., a single prediction sequence:

- All predicted contacts are removed from the contact plan.
- A *prediction base* is dynamically constructed from the *contactLog* lists, the SENDER list followed by the RECEIVER list. Each element of each *contactLog* list, in order, is inserted in the reconstructed contact plan in the usual way. Inserting all SENDER log entries before the RECEIVER log entries ensures that a contact reported by a receiving node that has also been reported by the sending node is excluded from the contact plan due to time overlap; the report from the sending node is always assumed to be more accurate. Elements of the prediction base are ordered by:
    - Sending node
    - Receiving node
    - Start time

For each element of the prediction base, the *duration* of the element is the contact's Stop time minus its Start time and the *volume (or capacity)* of the element is the contact's duration multiplied by its nominal data rate.

- A *prediction sequence* is any sequence of entries in the prediction base comprising all, and only, entries for some single sending node and some single receiving node. A *gap* in a prediction sequence is the time interval between the Stop time of some entry in the prediction sequence and the Start time of the next entry in the prediction sequence.
- For each prediction sequence:
    - The mean duration MC of all contacts in the prediction sequence is computed.
    - The corresponding standard deviation DC is computed.
    - The mean duration MG of all gaps in the prediction sequence is computed. If the prediction sequence contains no gaps, then MG is zero.
    - The corresponding standard deviation DG is computed, except that if MG is zero then DG is zero.
    - The mean capacity MV of all contacts in the prediction sequence is computed.
    - If DC < MC and DG < MG then the contacts appear to be somewhat non-random and we assert our *base confidence* for this prediction sequence to be 0.2; otherwise we detect no discernible pattern in the contacts and our base confidence is 0.05. (These values are just a first guess; they need to be tuned as we experiment with the system.)
    - Our *net confidence* for this prediction sequence is 1.0 – (1.0 – base confidence)^N where N is the number of contacts in the prediction sequence.
    - The *prediction horizon* for this prediction sequence is the current time plus the difference between the current time and the Start time of the first contact in the prediction sequence. (That is, we only predict into the future as far as we can see into the past.)
    - We then insert predicted contacts as follows:

- Set Time to the Stop time of the last contact in the sequence.
- Until Time is greater than the prediction horizon:
    - Predicted gap's Start time is Time. Predicted gap's Stop time is its Start time plus MG, minus DG; if the computed Stop time is less than the computed Start time, set the Stop time to the Start time (i.e., the predicted gap duration is zero). Gap duration is intentionally underestimated.
    - Predicted contact's Start time is the predicted gap's Stop time. Predicted contact's Stop time is its Start time plus MC, plus DC; contact duration is intentionally overestimated. Predicted contact's data rate is MV divided by predicted contact duration (Stop time minus Start time). If the predicted contact's data rate is greater than 1 byte per second and its Start time is greater than the current time, set the predicted contact's confidence level to the net confidence for this prediction sequence and insert the predicted contact into the contact plan.
    - Set Time to the Stop time of the predicted contact.

The CGR library

The *libcgr.c* source file has been modified to get rid of "ranges" for discovered and predicted contacts. The reason is that CGR wants every contact to happen in a interval where a range is defined. The range indicates the light distance between a nodes pair, i.e. the time it takes to the light to travel from a node to its neighbor. If the contact plan contains a contact scheduled in a moment when no ranges are defined or if the contact is not completely scheduled within a range, this contact will not be taken into account for route calculation.

Assuming that in an opportunistic environment such as a terrestrial mobile network the light distance between two neighbors can be ignored, OCGR needn't ranges for

discovered and predicted contacts and for each discovered or predicted contact it assumes the light distance between the nodes pair as 0.

4.4 Integration into ONE

To integrate the new Opportunistic Contact Graph Router protocol into ONE we needed to extend the classic CGR integration. On the C side we had to provide new entry points in order to support the information exchange between nodes that discover each other and the contact prediction. On the Java side we created the *OpportunisticContactGraphRouting* class that extends the *ContactGraphRouting* class.

4.4.1 Simulating contact history exchange

Since ION is not actually running in the simulator, the population of the contact plan with predicted contacts and the information exchange between neighbors must be simulated. Therefore, in addition to the new ION libraries modifications, we need to create an additional simulation library that performs as follows:

- Whenever the simulated start of a contact between nodes A and B occurs:
  - All current discovered contacts in the contact plan of node A are copied into the contact plan of node B, and vice versa.
  - All entries in each *contactLog* of node A are copied in the corresponding *contactLog* of node B, and vice versa.
  - The `rfx_predict_all_contacts()` function is invoked on both node A and node B.
  - Operation of the eureka library is simulated:
    - New discovered contacts (in both directions between the two affected nodes) are inserted into the contact plans of both nodes.
  - At node A and node B, for each bundle currently in limbo, the `cgr_forward()` function is performed.
- Whenever the simulated termination of a contact between nodes A and B occurs:

- The `rfx_remove_discovered_contacts()` function is invoked at both nodes. This has the effect of removing the discovered contact(s) and updating the local contact history.

4.4.2 The native code

cgr_jni_Libocgr.c

The new entry points are defined in the *cgr_jni_Libocgr.c* source file and basically they wrap calls to the operational functions defined in the *chsim.c* source file. The entry point functions perform the same environment updates as the ones described in the CGR integration. The functions defined in *cgr_jni_Libocgr.c* reflect the methods of the *cgr_jni.Libocgr.c* java class; they are:

- `predictContacts()` called upon the discovery of a new contact. This function triggers the contact prediction based on the new contact history enhanced by the contact history exchange between the two neighbors.
- `exchangeCurrentDiscoveredContacts()` called upon the discovery of a new contact. This function triggers the simulation of the discovered contacts exchange between the two neighbors.
- `exchangeContactHistory()` called upon the discovery of a new contact. This function triggers the simulation of the contact history exchange between the two neighbors.
- `contactDiscoveryLost()` called upon the lost of a discovered contact. This function triggers the deletion of a discovered contact from the contact plan and the insertion of it in the contact history log.
- `applyDiscoveryInfo()` this function was defined to support a new discovery contacts exchange protocol now disbanded as considered premature optimization. The function has not been deleted to be easily re-enabled whenever this protocol will be useful.

The aim of this functions is to support the simulation of the discovery management that in a real ION framework should be done by the eureka library.

chsim.c

The file *chsim.c* defines the functions used to simulate the information exchange between two nodes that acquire or lose a connection. While the *cgr_jni_Libocgr.c* source file only defines the entry points for the ONE framework, here the real operational functions are defined.

In order to simulate the current discovered contact information exchange between two nodes we need to look through the whole contact plan of a node, store the found discovered contacts in a list and insert each contact of the list in the contact plan of the peer node using the specific RFX function. The `exchangeCurrentDiscoveredContacts()` function performs the information exchange in both ways, so it is supposed to be invoked only once per pair of nodes. The RFX function `rfx_insert_contact()` insert the contacts in the contact plan and takes care of possible duplicated contacts.

Likewise, to simulate the contact history exchange, we need to look through the contact history log of a node, copy all the entries in a list and insert them in the history log of the peer node using the specific RFX function `rfx_log_discovered_contact()`. This function is used when a node lose a discovered contact and it takes care of possible duplicated entries as well. The `exchange_contact_history()` function is supposed to be invoked only once per pair of node since it performs the contact history exchange in both ways.

The RFX functions we use to insert discovered contacts and history log entries are defined in the *ici/rfx.c*, file that we don't want to modify. Every RFX function uses the PSM and SDR partitions of the local node so if we want to copy the history log of node A to the node B we need to set the thread-specific local node number reference to A, read the history log from the *IonDB* object, copy all the entries in a list, set the thread-specific local node number to B and call the RFX function to insert in the history log of node B all the entries earlier saved in the list. This can be done because the thread-specific local node number represent the node the ION code is managing: every time this value changes, the PSM and SDR partitions references are updated to point the new local node runtime space.

The `insertDiscoveredContact()` is invoked when a node discover a new neighbor and opens a connection to it; it uses the RFX function `rfx_insert_contact()` to insert a new discovered contact and its symmetric one. Respectively the `contactLost()` function is invoked when a node lose a connection to a neighbor and it uses the RFX function `rfx_remove_discovered_contacts()` to remove the discovered contact from the contact plan and to insert it in the contact history log.

The function `predictContacts()` is invoked after any information exchange between two nodes and it uses the RFX function `rfx_predict_all_contacts()` to trigger the contact prediction algorithm that will fill the contact plan with probabilistic contacts based on the contact history.

The functions `notifyNeighbors()` and `applyDiscoveryInfo()` simulate a discovered contacts exchange protocol now disbanded as considered premature optimization.

4.4.3 The Java code (ONE extension)

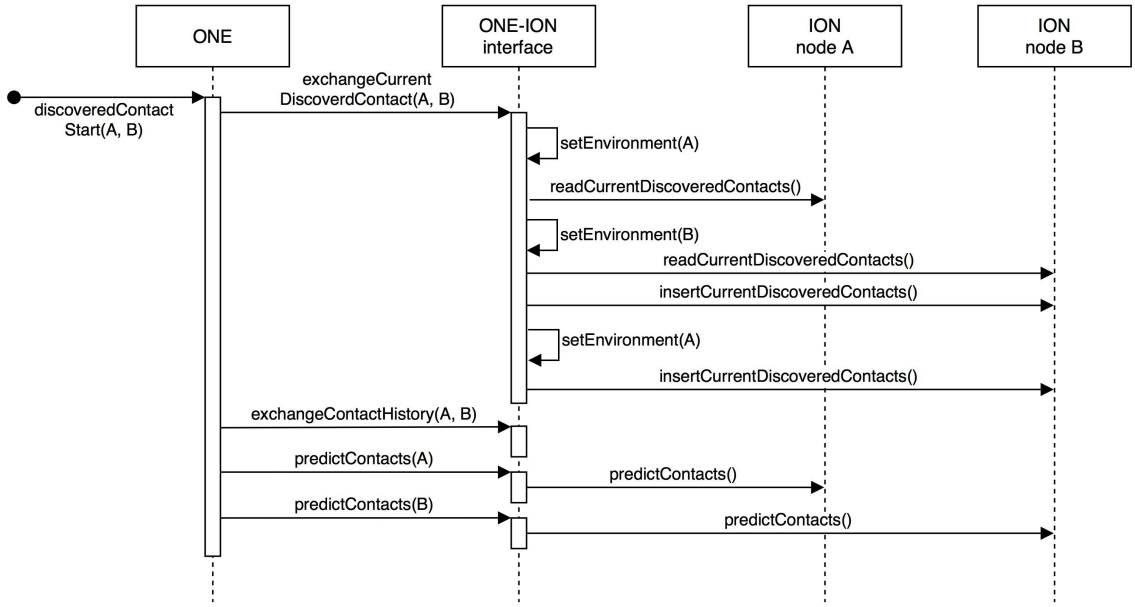

Image 4: sequence diagram of function invocations triggered by the discovery of a new contact. The exchangeContactHistory() has not been expanded as it behaves similarly to the exchangeCurrentDiscoveredContact(). Function names and signatures have been renamed for a better reading.

Since we already extended ONE to support the simulation of the CGR, in order to simulate OCGR as well we need to extend the *ContactGraphRouting* 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 bundle.

Contact discovery

- The method `discoveredContactStart()` has been implemented. It 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 method `discoveredContactEnd()` has been implemented. It 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 method `changedConnection()` on both ends of the connection.

This method has been overridden in our class. It invokes:

- The method `discoveredContactStart()` if the connection is up.
- The method `discoveredContactEnd()` if the connection is down.

Epidemic drop back

Our simulation of the OCGR provides a epidemic drop back mode that can be enable 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 of the contact plan. It 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 could not find any route to the bundle destination. 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 can not 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
- repeat the previous step until either the transfer successfully starts or there are no more bundle 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.

The OGCR specific MessageStatsReport

ONE can provide a series of report as result of the simulations. The main report is the *MessageStatsReport* that contains statistical informations about the simulation such as the number of bundles created, forwarded and delivered, the overhead ratio and the delivery probability. Each report type is defined in a class by ONE and the compilation of a specific report must be requested in the settings file before starting the simulation. We implemented a OCGR specific *MessageStatsReport* called *OCGRMessageStatsReport* that shows different counters for the OCGR-forwarded bundles and for the epidemic-forwarded ones, in addition to the cumulative counters. This report is implemented in the *report.OCGRMessageStatsReport* class, that extends the *report.MessageStatsReport* class, and it is enabled in the settings file like all the others reports. This report does not work with other routers than the *OpportunisticContactGraphRouter*.

4.4.4 ONE settings for OpportunisticContactGraphRouter

The *OpportunisticContactGraphRouter* like other ONE routers can be initialized with settings read by ONE from a settings file at the beginning of the simulation. *OpportunisticContactGraphRouter* supports the following settings:

- *epidemicDropBack* if set to true the epidemic drop back mode is enabled.

Default is true.

- *preventCGRForward* if set to true the function `cgrForward()` will never be invoked. This is useful only for test and debug purposes. Default is false.
- *debug* if set to true ONE will print useful debug informations to the standard output. Default is false.

4.5 Optimizations

4.5.1 Symptoms

The first tests revealed that the simulation speed of OCGR in ONE is way slower than the other protocols speed. For example the same simulation would take a few minutes to finish with PROPHET routing while it would take days to finish with OCGR. This is due to the fact that while the simulation runs, the contact history of each nodes 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, thus, with a huge contact plan, this results to be really slow.

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, until the whole system memory becomes full and the operative system throws a memory error. Thus the simulation cannot finish. In order to have any result from the simulations, we needed to optimize the code and the algorithm to be faster and less memory hungry.

4.5.2 Contact prediction optimization

The total number of contact plan entries depends mainly on how many contacts are inserted by the contact prediction algorithm. In fact for each nodes pair it can insert as many contacts as the number of contact history entries that involve the same nodes pair. We can optimize this behavior performing as follows for each nodes pair:

- Instead of inserting all the predicted contacts in the contact plan only one contact

will be inserted.

- 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, while before it was depending also 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.

4.5.3 Route calculation optimization

The CGR library defines three different payload classes and performs route calculation for each one of them. Each payload class defines a contact capacity floor threshold: every contact whose capacity 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 performing 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 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 contacts capacity, 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.

5 CONCLUSIONS

This thesis has been carried out at NASA JPL in Pasadena (California), under the direct guide of my co-supervisor, Scott Burleigh, leader of the DTN research in NASA. As that, it was natural to focus the thesis work on the most urgent topic, which at present is the extension of the application field of the ION DTN implementation, from space networks to terrestrial non-deterministic environments, such as MANETs. In brief, the aim is to transfer, once again, the results of the most advanced aerospace research to the terrestrial field, as done so many times in the aerospace history.

Neighbor and service discovery capabilities may not be necessary in space environments, where node contacts can be scheduled in advance, but they are an instrumental feature in non-deterministic environments. For this reason, I started my work by testing the brand new ION IPND implementation , removing bugs and writing the official documentation (main page).

Then I moved on routing, another interesting topic. Firstly, I integrated into “The ONE” the ION CGR algorithm. This one guarantees optimal performances in deterministic networks, but it is not operable, as it is, in an opportunistic environment. Therefore, starting from the analysis of a mobility trace dataset, I collaborated with Scott Burleigh to the development of the Opportunistic CGR (OCGR) extension, and then I have integrated it, into ”The ONE” DTN simulator, by extending the previously developed CGR integration. Preliminary simulation results show that OCGR seems to have a great potential: once properly tuned, it could become a serious competitor of the best opportunistic routing algorithms, while maintaining its dominance in the deterministic space environments.

APPENDIX 1: COMPILATION AND SIMULATION

Files and directories organization

The ION integration for ONE that support the simulation of CGR and OCGR comes within a single packet that contains the Java classes that extend the ONE framework and the native code that simulates the ION environment.

The Java classes

The Java code is organized following the Java standard guideline for packages and classes: each file contains a class and its name is *ClassName.java* and 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 folder *src*.

Since we want to use our classes in the ONE framework, we needed to use the same packages used by ONE. The packages and classes used directly by ONE are:

- package *routing*: classes *OpportunisticContactGraphRouting* and *ContactGraphRouting*
- package *test*: classes *OpportunisticContactGraphRoutingTest*, *ContactGraphRoutingTest* and *TestUtilsForCGR*
- package *report*: class *OCGRMessageStatsReport.*

All the other classes manage the JNI interaction with the ION integration native code and are defined within the *cgr_jni* package.

The native code

The C source and header files are in the *ion_cgr_jni* folder that tries to follow 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 exactly as in the original ION distribution.
- *ici* folder: contains all the source files of the ICI libraries and the needed headers exactly as in the original ION distribution.
- *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 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.

Native library compilation

The provided *Makefile* in the *ion_cgr_jni* directory takes care of the compilation. The make program should be invoked as follows:

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

The ONE classpath is the root directory of the packages containing the *.class* files result of ONE compilation. If DEBUG is defined, the CGR debug prints are enabled. In addition, the environment variable $JAVA_HOME needs to be set in order to let the compilator to find the JNI header files. It is usually set to */usr/lib/jvm/java-8-oracle/* depending on which version of JRE is installed. If this variable is not set, the compilation fails.

Also, this *Makefile* assumes that the Java classes of the *cgr_jni* package are compiled in the *bin* directory, as Eclipse does. If this is not the case, the classpath of these classes needs to be appended to the ONE classpath parameter upon make invocation as follows:

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

ONE modifications

Even if we tried to keep our code completely separate from the ONE code in order to distribute our package as a external module easily integrable into a existing ONE installation, we needed to perform a few changes in the ONE code to make the ION adaptation to work.

Mandatory modifications

Without these modification, the ION adaptation will not work. 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 modifications are needed because ION can not handle a node whose 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.

  ```
  java -Xmx512M -cp
  .:lib/ECLA.jar:lib/DTNConsoleConnection.jar:
  $CGR_JNI_CLASSPATH core.DTNSim $*
  ```

  This modification is needed because we need to tell the JVM where to find our ION integration classes, and we do that by adding an environment variable to the Java classpath. In this way whenever we change the location of the ION integration classes we do not need to change this file again.

Optional modifications

The following changes are needed only if we want to use the OCGR specific message stats report. This is not necessary and the simulations can be done with the original code.

- Class *report.MessageStatsReport* line 178, modify as follows:

```
        write(statsText);
        write(getMoreInfo());
        super.done();
    }

    protected String getMoreInfo() {
        return "";
    }
```

That is: define the `getMoreInfo()` method and write the string returned to the output stream before closing it. This method is overridden in our *report.OCGRMessageStatsReport* class.

Running the simulations

ONE provides two ways to performs simulations: graphic mode and batch mode. The graphic mode opens a graphic user interface that shows the nodes moving in the map and allows the user to interact with the simulation with the pause, step and fast forward buttons. The batch mode allows the user to perform a series of simulations automatically changing a parameter (such as bundle size or expiration time) each run. While the first mode is used to see how the nodes move, the second one is used to do actual performance tests.

For both modes ONE is started with the *one.sh* script. Before invoking the script we need to set the $LD_LIBRARY_PATH and the $CGR_JNI_CLASSPATH environment variables. The first informs the JVM where to find the *libcgr_jni.so* native library, the second informs the JVM where to find our Java classes. To set these variables the following commands need to be execute:

```
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.

Running batch simulations

In order to simplify the simulation set up and results analysis processes, a utility script has been developed. The script name is *batch_test.sh* and it is in the *simulations* folder. 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] 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.

ACKNOWLEDGMENTS

I would first like to thank my thesis co-supervisor Scott Burleigh of NASA JPL for letting me work on this ambitious research project directly from inside the JPL, one of the coolest place in the world, and for his availability any time I needed a hint or a help.

I would also like to thank the School of Engineering and Architecture of the University of Bologna for providing me with a scholarship in order to conduct this research abroad.

Then I would like to thank my family for always supporting me.

Finally I would like to thank all the people that made this amazing experience memorable: Lourdes, Alfredo and all my extended Mexican family; my loyal bearded companion Giulio; the dream traveler Felicitas; my office mate Lorenzo and all the people of the awesome JPL lunch crew; the crazy friends of the PCC; and all those who I cannot list because they would be too many for a regular “acknowledgment” page.

Thank you to all of you.

Michele Rodolfi

BIBLIOGRAPHY

[A. Keränen, 2010] Keränen, Ari, Teemu Kärkkäinen, and Jörg Ott. "Simulating Mobility and DTNs with the ONE." Journal of Communications 5.2 (2010): 92-105.

[Apolonnio et al., 2013] P. Apollonio, C Caini, M Lülf “DTN LEO satellite communications through ground stations and GEO relays” - Personal satellite services, 2013

[Balasubramanian et al., 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 et al., 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.

[Burleigh et al., 2015] Scott Burleigh; Giuseppe Araniti; Nikolaos Bezirgiannidis; Edward Birrane; Igor Bisio; Carlo Caini; Marius Feldmann; Mario Marchese; John Segui; Kiyohisa Suzuki “Contact graph routing in DTN space networks: overview, enhancements and performance”

[DTNRG] DTN Research Group, http://www.dtnrg.org/wiki/Code

[E. Birrane et al., 2012] E Birrane, S Burleigh, N Kasch “Analysis of the contact graph routing algorithm: Bounding interplanetary paths” - Acta Astronautica, 2012

[Grasic et al., 2010]Grasic, Samo, et al. "The evolution of a DTN routing protocol-PRoPHETv2."Proceeding. ACM, 2011.

[I. Bisio et al., 2008] Bisio, Igor, Mario Marchese, and Tomaso De Cola. "Congestion aware routing strategies for DTN-based interplanetary networks." Global Telecommunications Conference, 2008. IEEE GLOBECOM 2008. IEEE. IEEE, 2008.

[ION] ION manual at https://sourceforge.net/

[IPND] DTN IP Neighbor Discovery (IPND). IETF draft.

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

[PROPHET] Probabilistic Routing Protocol for Intermittently Connected Networks, Mar 2006

[RFC4838] Delay-Tolerant Networking, http://tools.ietf.org/html/rfc4838

[RFC5050] Bundle Protocol Specification, http://tools.ietf.org/html/rfc5050

[RFC5325] Licklider Transmission Protocol – Motivation, http://tools.ietf.org/html/rfc5325

[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”.

[T. Spyropoulos et al., 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 and Becker, 2000] Vahdat, Amin, and David Becker. "Epidemic Routing for Partially-Connected Ad Hoc Networks."