Alma Mater Studiorum · University of Bologna

SCHOOL OF ENGINEERING AND ARCHITECTURE
Department of Computer Science and Engineering
Master's degree in Computer Engineering

Master's thesis
in
Protocols and Architectures for Space Networks M

Hierarchical Inter-Regional Routing Algorithm for Interplanetary Networks

Defended by:
Nicola Alessi

Supervisor:
Prof. Carlo Caini
Co-Supervisor:
Scott Burleigh (NASA JPL)

Academic Year 2017-2018

To my beloved sister,
an endless source of inspiration.
To my parents,
constituent elements of my life.

Abstract

Le comunicazioni spaziali (satelliti geostazionari, satelliti ad orbita bassa, comunicazioni interplanetarie o con lo spazio profondo) sono caratterizzate spesso da lunghi ritardi, perdite elevate e connettività intermittente con frequenti interruzioni. I protocolli TCP/IP risultano inadatti ad affrontare questi problemi.
Per le comunicazioni via satellite geostazionario sono state create soluzioni dedicate, come i Performance Enhancing Proxies (PEP), le quali tentano di affrontare il problema cercando di continuare ad utilizzare i protocolli TCP/IP. Nonostante questo sforzo abbia portato a buoni risultati per le comunicazioni via satellite geostazionario, in cui il Round Trip Time (RTT) è dell'ordine di circa 600ms, è risultato essere difficile adattare questa tecnologia alle reti interplanetarie o nello spazio profondo (Deep Space Network, DSN) dove i ritardi di propagazione sono molto maggiori.
Verso la fine degli anni '90 Vint Cerf, già progettista insieme a Bob Kahn dei protocolli TCP/IP (1978), insieme ad altri ricercatori della NASA JPL (Jet Propulsion Laboratory) iniziò a studiare la fattibilità di estendere la rete Internet terrestre per poterla usare nelle reti spaziali: è nato così il concetto di Internet Interplanetaria (InterPlanetary Networking, IPN). Negli anni successivi si è capito che reti di altra natura presentavano caratteristiche simili alle reti interplanetarie. Per identificare questo tipo di reti in cui i protocolli Internet tradizionali falliscono è stato coniato il termine "Challenged Networks". L'idea di IPN si evolve così in Delay-/Disruption-Tolerant Networking (DTN), con l'obiettivo di fornire un'architettura di rete adatta ad affrontare i problemi di tutte le reti challenged, non solo di quelle interplanetarie.
L'architettura DTN e i suoi protocolli sono stati inizialmente sviluppati dall' Internet Research Task Force (IRTF) DTN Research Group (DTNRG). Esso è stato recentemente sciolto a seguito della costituzione di un analogo gruppo all'interno di Internet Engineering Task Force. Il passaggio da "Research" a "Engineering" è indicativo del livello di maturità nel frattempo raggiunto dagli standard. Un altro gruppo di ricerca importante che lavora sull'architettura DTN è il Consultative Committee on Space Data Systems (CCSDS) DTN Working Group, che si interessa per di proporre nuove specifiche finalizzate all'utilizzo dell'architettura DTN in ambito spaziale. CCSDS è un organismo composto dalle maggiori agenzie spaziali, comprese NASA ed ESA.
Tra i vari aspetti in cui le reti DTN differiscono dai protocolli TCP/IP abbiamo il modo in cui viene effettuato routing, ovvero l'instradamento dei dati inviati da una sorgente ad un destinatario. Il Contact Graph Routing(CGR) è l'algoritmo di routing proposto per le reti DTN in ambito interplanetario. L'aspetto che contraddistingue il CGR dagli algoritmi di routing classici è che esso costruisce una rotta di "contatti" (ovvero delle possibilità di comunicazione programmate), anzichè costruire un percorso di nodi. Questa caratteristica si rivela essere efficace nell'ambito delle reti spaziali, dove i contatti sono

noti a priori.
Nonostante il CGR sia molto efficiente e spesso in grado di trovare il percorso ottimo, ovvero quello che permette di arrivare nel più breve tempo possibile a destinazione, esso presenta dei problemi di scalabilità. Infatti, con l'aumentare del numero dei contatti il suo tempo di esecuzione tende a crescere in modo esponenziale. In questa tesi viene proposto un algoritmo di routing chiamato Hierarchical Inter-regional Routing (HIRR) che ha l'obiettivo di mitigare il problema di scalabilità del CGR dividendo i nodi della rete in diverse regioni amministrative, in cui l'utilizzo del CGR non risulta essere critico. Lo scopo principale di HIRR è quindi quello di cercare di trarre il massimo beneficio dal CGR, accettando un ragionevole compromesso fra ottimalità delle rotte e tempo di calcolo.
Questa tesi è stata svolta al Jet Propulsion Laboratory della NASA, situato a Pasadena in California, aderendo al Visiting Student Research Program (VSRP).

Contents

1 Introduction **1**

1.1 Delay/Disruption Tolerant Networking (DTN) overview 1
1.2 The DTN Architecture 2
1.2.1 The Bundle layer 2
1.3 Contact Graph Routing (CGR) overview 3
1.3.1 Other routing approaches 5

2 Hierarchical Inter-Regional Routing (HIRR) **6**

2.1 Motivations 6
2.2 Organization of the hierarchical region structure 7
2.2.1 One region 7
2.2.2 Two layers 8
2.2.3 Three layers (and more) 9
2.3 Rationale of the Hierarchical Inter-Regional Routing algorithm 10
2.4 Intermittent connectivity and reachability 11
2.4.1 Viable routes and passageways 12
2.4.2 Dynamic topology and Internet comparison 12
2.5 Overview of the HIRR algorithm 13
2.6 Algorithm implementation 17
2.6.1 Blacklisting 17
2.6.2 Search strategy 18
2.6.3 Backward route learning 20
2.6.4 Forward route learning 21
2.6.5 Pseudocode 24
2.7 HIRR Variants 27
2.7.1 Auto rediscovery 27
2.7.2 2-way handshake learning 27
2.8 Algorithm Analysis 28
2.8.1 Level representation 28
2.8.2 Derivation 29

3 JRegion prototype **33**
3.1 Motivations 33
3.2 Software architecture 34
3.2.1 Software model description 34
3.2.2 Running environment description 39
3.3 Usage 41
3.3.1 Interpreter commands 41

4 HIRR implementation within ION **53**
4.1 ION Overview 53
4.2 ION Design Principles 54
4.2.1 Inter-task communication 54
4.2.2 Zero-copy objects (ZCO) 54
4.2.3 Personal Space Management (PSM) 55
4.2.4 Simple Data Recorder (SDR) 55
4.3 HIRR implementation 55
4.3.1 Forwarding Strategy 56
4.3.2 HIRR model 57
4.3.3 bprc configuration file 58
4.3.4 ipnfw.c functions 59

5 ION Inter-regional Test Suite **62**
5.1 Motivations 62
5.2 Scripts 63
5.3 Experiments 65
5.3.1 Scenario 65
5.3.2 JRegion and ION configuration files 66
5.3.3 Experiment 1 67
5.3.4 Experiment 2 69
5.3.5 Experiment 3 70
5.3.6 Experiment 4 70

Conclusions **72**

Acknowledgments **73**

Chapter 1

Introduction

1.1 Delay/Disruption Tolerant Networking (DTN) overview

The communication protocols currently used in the Internet, in particular the ones from the TCP/IP suite, were developed according to the following basic assumptions [5]:

1. there is always a continuous path between source and destination;
2. packet losses are few and are mainly caused by network congestion;
3. the transmission delay is small, so end-to-end retransmission mechanisms are a suitable means of recovering lost data;
4. the network is homogeneous: each node supports the TCP/IP suite protocols.

When the TCP/IP protocols are used in networks where one or more of these assumptions are not met, the results are often unsatisfactory. For this reason, these are sometimes called "challenged networks"[6].
In Inter Planetary Networking (IPN) scenarios, delays are on the order of minutes or hours (because of long interplanetary distances), the communication links are disrupted (because of the motion of the planets), and the data loss rate can be relatively high [4]. Initially, only the space scenario was considered in IPN Special Interest Group deliberations, but it soon became clear that some terrestrial networks were also "challenged" [8]. From that point, the IPN architecture evolved in the Delay-/Disruption Tolerant Networking (DTN) architecture, described in RFC4838 [7].
The application domain of the concepts originally developed for the IPN networks has been extended to all challenged networks, such as sensor networks in oceans, military tactical networks, satellite networks and so on.
The basic idea of DTN is analogous to the delivery of parcels: instead of having continuous data flows or small packets (at most a few thousand bytes), the communication is

based on "bundles", which are packets that can potentially reach large sizes. If links are intermittent, as in space, these bundles can be stored by DTN nodes for long periods, whenever necessary, waiting for the next opportunity of being transmitted to the following node on the path to the destination, i.e. for the next "contact".
Of course, the aim of DTN is not to solve the physical constraints of challenged networks (which cannot be eliminated) but to find the best way to cope with them, i.e. to make communications possible in spite of them. For example, one limitation that derives from long delays is that it is unwise to build applications that require many interactions. For example, even with DTN you cannot conveniently make a phone call to Mars, as you cannot eliminate the delay (the one way delay is in the range 3-23 min), but you can send e-mails or transfer files.
The more severe the challenges, the greater the benefit of using DTN rather than internet protocols. At the same time, the DTN architecture does not present significant disadvantages should the challenges be less severe than expected.
The DTN architecture and its protocols were first standardized by the Internet Research Task Force (IRTF) DTN Research Group (DTNRG). The DTNRG gave way to the Internet Engineering Task Force (IETF) DTN working group (DTNWG) in 2015, in recognition of the high level of maturity reached by the protocols. Standardization for space applications is at present carried out in parallel to IETF by the Consultative Committee on Space Data Systems (CCSDS), which is supported by almost all space agencies, including NASA and ESA.

1.2 The DTN Architecture

1.2.1 The Bundle layer

In the DTN architecture, described in RFC4838 [7], a new layer, called "Bundle layer", is inserted between Transport and Application layers. Devices implementing the protocol at this layer, called Bundle Protocol (BP)[10], are called "DTN nodes".
An application developed for DTN networks can send messages of arbitrary size called Application Data Units (ADUs). Each such message is encapsulated by the bundle protocol in one Protocol Data Unit (PDU) called a "bundle". Each bundle is composed of two or more "blocks". The compulsory blocks are the "primary block", which is the mandatory portion of the bundle header, and the "payload" block. Additional "extension" blocks may follow the primary block to support additional features, such as the Bundle Security Protocol (BSP). The bundle protocol is in charge of forwarding a bundle from a DTN node, also called a Bundle Protocol Agent (BPA), to the next one. Each node on the path can store bundles, if necessary, before forwarding them to the next node on the path. This mechanism is called "Store and Forward" and it is one of the key features of the DTN architecture. Any required re-transmissions in an ARQ

(Automatic Repeat re-Quest) scheme may come from an intermediate node, and no end-to-end connection is required between the sender and destination. Bundles can survive a machine reboot or crash, if they are stored persistently. This approach is a departure from the internet protocols, where a packet that cannot be immediately forwarded is discarded. There is a sub-layer below the bundle layer called the "Convergence Layer", which contains "Convergence Layer Adapters" that work as an interface to underlying "Convergence Layer Protocols", which are typically transport protocols. Note that the insertion of the Bundle Layer redefines the role played by Transport, which is no longer end-to-end between source and destination but rather is limited to a single DTN hop (i.e. between two topologically adjacent DTN nodes). This allows for the use of different transport protocols on different segments of a heterogeneous network, which is required in order to cope with the different impairments that can be found on these diferent path segments. For example, LTP (Licklider Transmission Protocol) can be used on space links, while on terrestrial links it is more convenient to use TCP or UDP.

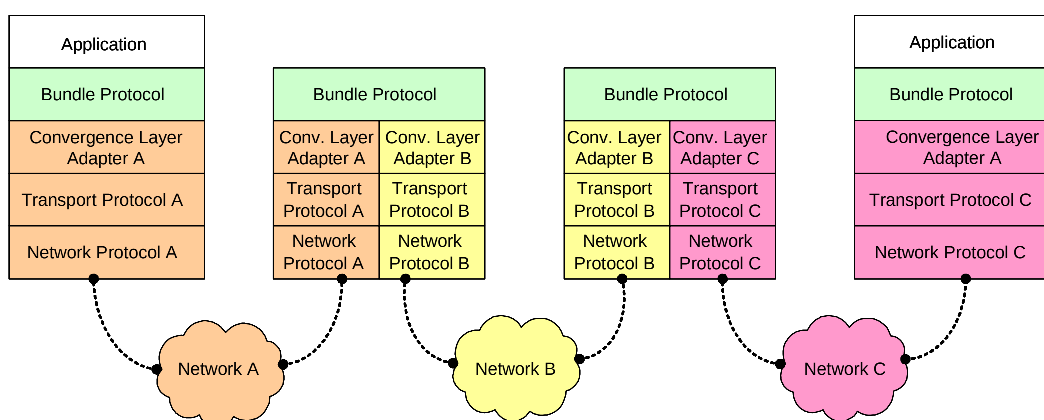

Figure 1.1: DTN Architecture and protocol stack.

1.3 Contact Graph Routing (CGR) overview

Routing is the procedure that selects the most suitable route by which to forward a packet from the source node S to the destination node D in a network. In DTN, this procedure is usually based on heuristic approaches that lead to sub-optimal solutions. This happens because the nodes in the network can use only local information while taking a routing decision. If a node could have information on the status of the other nodes, the solution would be optimal.
Routing in "challenged networks" is much more complex than in Internet, because link intermittency and long delays prevent a rapid exchange of information between nodes, which is essential in ordinary routing protocols.

Contact Graph Routing (CGR) [1] is a routing algorithm designed by NASA-JPL and proposed for DTN networks. Since the DTN networks are mostly characterized by predictable disruptions and changes cannot be propagated in a short time, in CGR the topology of a DTN network is defined as a sequence of "contacts", which represent the opportunities for transmission between two nodes. The contacts are realized by the motion of nodes. This motion is assumed random in terrestrial networks, while is deterministic in space, as it derives from the motion of planets and spacecraft. This dichotomy between random and deterministic necessarily leads to completely different routing algorithms for the two environments. As the space contacts are predictable, each DTN node in space owns a list of contacts, known as "Contact Plan". Starting from the contact plan, the task of CGR is to find the most suitable route (series of contacts) from source to destination, based on some routing metrics such as the earliest delivery time. As the network state may change over time while bundles are traversing the network (which could last hours, instead of a few milliseconds as in Internet), the best route may change while bundles are on the flight. For this reason, CGR is performed on each node to recompute the routes using local information.
Here we will limit the discussion of CGR to a few key points, referring the reader to [1] for a more comprehensive treatment. A helpful way to understand CGR is to note that the CGR routing problem is similar to planning the best sequence of airline flights to a distant destination. In this analogy, contacts are flights and bundles are passengers.

- Each node uses the contact plan information (flight schedules) to build a "contact graph" and then computes from this graph a routing table listing the plausible routes (sequences of flights to arrive at destination);

- For each bundle (passenger), CGR checks the available routes and chooses the best. In contrast to booking flights, however, once the route is selected CGR uses only the information related to the first contact: it selects the receiving node of that contact as the best proximate node to which to send the bundle and it sets a time limit ("forfeit" time) for successful transmission. In our analogy, this is equivalent to booking the first flight only. The reason for this divergent behavior is given in the next bullet.

- For safety, the best route is recomputed at each node along the path to destination. This is justified by the fact that, by contrast to flight booking programs, DTN nodes cannot inform other nodes of their decisions in advance, thus all nodes on the end-to-end path are unaware of other nodes' bookings on connecting "flights".

- CGR enforces bundle priority. This means that a higher priority bundle has the right to use a contact already fully allocated to lower priority bundles. This, on the other hand, may lead to the need to reforward lower priority bundles. This is addressed as soon as possible by the recently introduced "Overbooking management" policy [2].

- If a bundle is not transmitted before its "forfeit" time elapses, it is immediately queued for another pass through the routing procedure.
- In the route selection process, recent versions of CGR take account of data already scheduled for transmission to proximate nodes in order to determine the corresponding queueing delay. This consideration is termed "Earliest Transmission Opportunity" awareness[3], which is limited to the first hop). The same mechanism allows CGR to skip a route if the bundle's estimated volume consumption is larger than the residual volume of the first contact for its level of priority - by analogy, if the flight is fully booked by passengers of the same or higher priority.

Last, it is important to point out that CGR evolves constantly, and new features or variations of the basic protocol are introduced at almost any new release of ION. The latest version is going to be standardized by CCSDS as SABR (Scheduled Aware Bundle Routing).

1.3.1 Other routing approaches

One of the main research activity in DTN networks is to find strategies to cope with the new routing constraints. The proposed approaches to the DTN routing are different on the basis of the amount of information that a node holds: the two opposites are those with zero configuration information (as known as "opportunistic routing") to those with perfect knowledge of the contacts, but not of buffer occupancy at remote nodes (as known as "deterministic routing"). Opportunistic approaches usually are based on flooding strategies such that the messages are typically duplicated either a fixed number of times or else a variable number of times based on contact probability. If the probability of having a contact establishment is high, the delivery success rate is higher than approaches that rely on the accuracy of current network state information. The most significant opportunistic routing algorithms are single-hop multi-cast forwarding (Spray and Wait), distribute application messages (carriers) to hosts within connected portions of ad hoc networks (Epidemic Routing), and probabilistic analysis of predicted node contact (PRoPHET). The details about these algorithms are out of the scope of this thesis.
If the network is characterized by predictable contacts, a deterministic algorithm can improve performance and avoid wasting bandwidth. The Contact Graph Routing (CGR) algorithm is a deterministic algorithm. It assumes that the nodes hold a list of contacts, known as a "contact plan". As the routing decisions are taken according to the contact plan, this algorithm is intended as a perfect knowledge approach to the DTN routing problem. Even if the initial formulation of CGR assumes that each node has full knowledge, it was also extended to work in less perfect knowledge systems.

Chapter 2

Hierarchical Inter-Regional Routing (HIRR)

2.1 Motivations

One of the limitations of CGR is that when the number of contacts becomes too high, the execution of the routing procedure becomes computationally infeasible. To find how to improve CGR is still an open challenge for the research community, which has proposed different approaches including the addition of optimizations and shortcuts to CGR [2] integrating with other alternative routing algorithms [1], and introducing a hierarchical routing structure.
Following this last approach, an inter-regional Routing (IRR) algorithm, designed to complement CGR, is proposed in this thesis. The key idea is to divide the nodes into "regions", each consisting of fewer nodes so that the usage of the CGR algorithm is feasible within each region. In other words, CGR use should be limited to Intra Regional Routing between nodes of the same region. By contrast, the inter-regional Routing algorithm proposed here should be in charge of the routing between several regions. If two nodes do not belong to the same region, the inter-regional procedure will try to find a route to the destination region. Region crossing should be performed through special nodes called "passageways", which should function as a gateway from one region to another one. The task of the passageways (which are the only nodes allowed to belong to two regions) is to connect regions and to allow forwarding between them. In spite of their name, the "regions" have no geographical meaning because otherwise the benefits of CGR (in terms of routing through contacts) would be lost. It is worth stressing once again here that a CGR route consists of a series of contacts, not of a series of nodes as in an Internet route. The series of contacts implies a geographical route, but it is not equivalent, as it is impossible to derive the series of contacts (the CGR route) solely from the list of nodes (the geographical route).

The idea of using regions in a routing protocol is not new. For example, in Internet the equivalent of inter-regional routing is performed by *BGP-4* (Border Gateway Protocol version 4), described in the RFC4271 [9]. Unfortunately, *BGP-4* is an algorithm that performs many negotiations between gateways; therefore, it is not suitable for DTN networks. The idea of splitting the network into different regions was considered in early DTN concepts, but there was never any agreement on the basis on which regions should be formed.
The inter-regional Routing algorithm proposed here follows a new approach that presents a few advantages with respect to the previous ones. First, the complexity of the algorithm is very low. Each passageway holds a list of banned destinations (blacklist of other passageways) and the cost of a routing decision is equivalent to a search for the destination through this list. For this reason, by contrast to CGR, the overall time and space complexity grow linearly with the number of nodes in the network. Second, routes are self-forming and each node does not need to know the entire network but only its peers. Third, the algorithm, even if distributed, produces consistent blacklists and tends to converge to a state where every pair of known nodes, belonging to different regions, are reachable via one direct inter-regional route. A node can also detect if a destination is not present in the network (never existed or removed) or, in one variant of the protocol, detect its possible later appearance in another region.

2.2 Organization of the hierarchical region structure

In order to introduce the principles of the inter-regional routing, it is opportune to represent the hierarchical region structure in a graphical way, which can be conveniently done step by step. However, instead of presenting the final structure, it is more suitable to build step by step the inter-regional hierarchical topology.

2.2.1 One region

Let us assume that at the starting point there is just one region, i.e. that all nodes are cited in the same contact plan. Both source and destination necessarily belong to the sole region (e.g. R_0), and the only possible routing is thus intra-regional (Figure 2.1).
As links inside the region are potentially intermittent, an intra-regional route must be a contact route. It is the task of the intra-region routing algorithm, such as CGR/SABR, to find the best contact route, from source to destination.

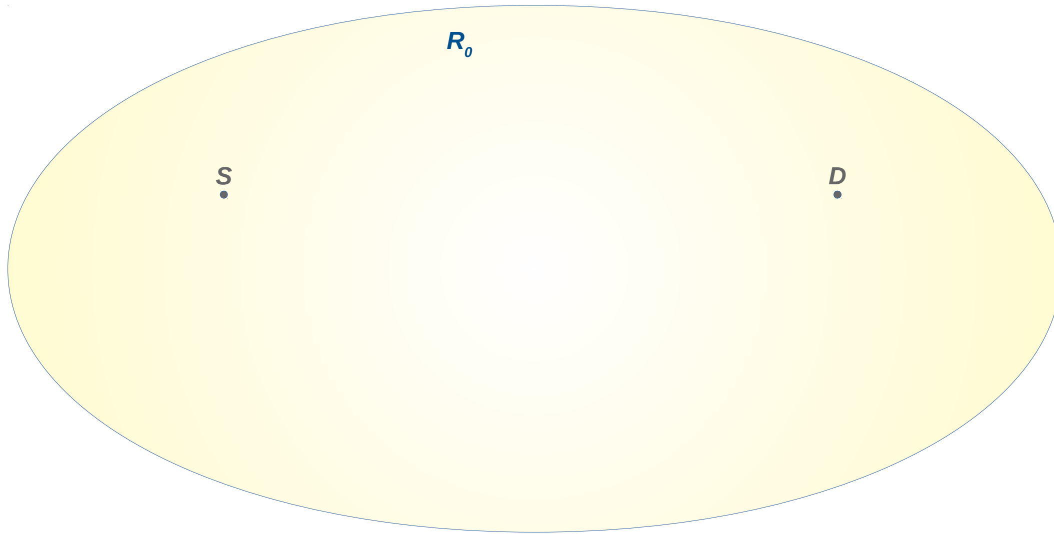

Figure 2.1: An initial situation where all nodes belong to the same region R_0, i.e. they share the same contact plan. As links are potentially intermittent, a route is always a contact route, to be found by the intra-regional algorithm, such as CGR-SABR.

2.2.2 Two layers

As the number of nodes increases, the processing time of the intra-regional algorithm may become excessive. It is, therefore, necessary to divide the set of nodes into several administrative subsets, called regions. These regions do not correspond to geographical regions (e.g. they could consist of nodes belonging to a space agency, or to different departments of the same space agency).
The network's "root" region R_0 functions as a backbone for the new "leaf" regions, such as R_1 and R_2 in the example shown in Figure 2.2. Each region has (and is defined by) its own contact plan. Regions do not overlap, which means that ordinary nodes can belong to only one region (i.e. they can appear only in the corresponding contact plan). Passageways are the exception, as they by definition belong to two regions, and thus they must be cited in a special way in both the contact plans. In the example, P_1 belongs to the R_1 region, which is its "home" region, but it is also projected on R_0, which is termed its "outer" region; P_1 is cited in R_1 contact plan as the only possible passageway to the upper region R_0; the same holds true for P_2, with reference to R_2.
Now if both source and destination belong to the same region (R_0, R_1 or R_2) the routing is purely intra-regional and everything goes as before. Vice versa, if the source is in one region, e.g. R_1, and the destination in another, e.g. R_2, the full route is inter-regional. More precisely, an inter-regional route can be defined as the sequence of nodes

$S - P_1 - P_2 - D$, consisting of the two end nodes and of the two traversed passageways, P_1 and P_2. The task of the inter-regional routing is to find the inter-regional section, i.e. the sequence of passageways between source and destination, i.e. in this very simple case $P_1 - P_2$. The inter-regional routing operates at a higher level with respect to the intra-regional routing algorithm, which is not replaced but just confined inside one region. More precisely, our inter-regional route consists of three regional legs, $S - P_1$, $P_1 - P_2$ and $P_2 - D$. On each regional leg, it is the task of the intra-regional routing algorithm to find the best sequence of contacts from the start to the end point. From S and P_1 the best sequence of contacts is derived from the R_1 contact plan, for $P_1 - P_2$, from R_0, and for $P_2 - D$ from R_2 one.

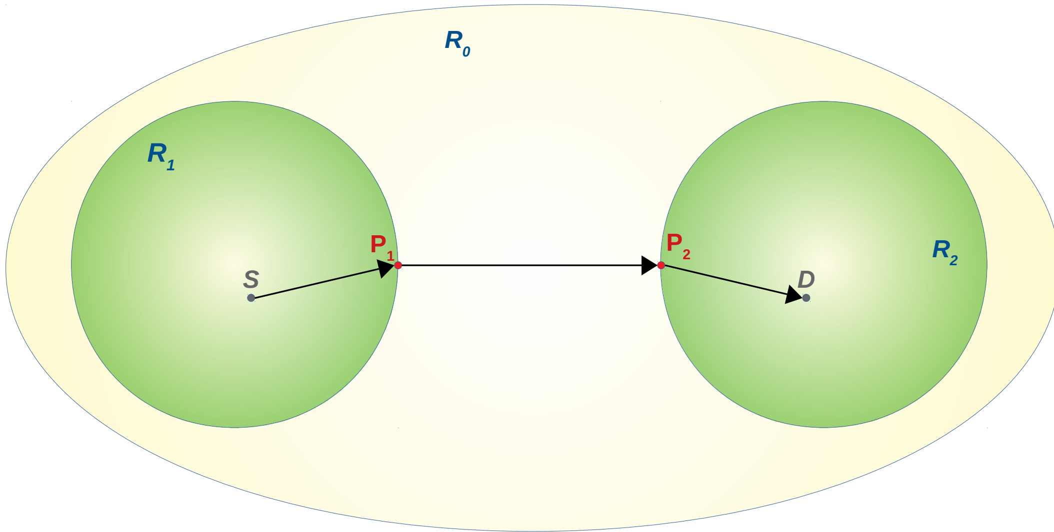

Figure 2.2: The inter-regional route consists of three regional legs, $S - P_1$, $P_1 - P_2$ and $P_2 - D$. On each regional leg, it is the task of the intra-regional routing algorithm to find the best sequence of contacts from the start to the end point. Possible intermediate nodes inside each leg are not reported because the sequence of nodes could vary depending on the sequence of contacts (CGR route) selected.

2.2.3 Three layers (and more)

Let us assume that after a while, as the number of nodes increases, it is necessary to introduce a third regional layer. For example, by adding the sub-regions R_{11}, R_{12} and R_{13} (passageway P_{11}, P_{12} and P_{13}), and also R_{21} and R_{22} to R_2 (passageway P_{21} and P_{22}) to our previous topology, we can obtain the three layer structure presented in fig. 2.3. Assuming that the source is on R_{11} and the destination on R_{22}, the inter-regional route becomes $S - P_{11} - P_1 - P_2 - P_{22} - D$. It is the task of inter-regional routing algorithm to

find inter-regional section, i.e. the sequence of passageways $P_{11}-P_1-P_2-P_{22}$. Source and destination can also be closer to each other, in terms of the number of passageways to be traversed. For example, if the destination was on R_0, the inter-regional section would be just $P_{11}-P_1$. Analogously, if on R_{12}, it would be $P_{11}-P_{12}$ and so on. Before proceeding, it is convenient to note that the region structure just designed can be represented as a tree, where elements are regions.
In our example, R_0 is the root; R_{11}, R_{12},R_{13}, R_{21} and R_{22} are leaf nodes; R_1 and R_2 are intermediate nodes. The tree structure is very useful, as it can represent in a very synthetic way the regions to be traversed from source to destination, i.e. the inter-regional routing problem. We could go on by adding a fourth layer of regions, without, however, adding anything new. We will thus prefer to terminate here the algorithm overview, letting to next section the task of going in depth on crucial points.

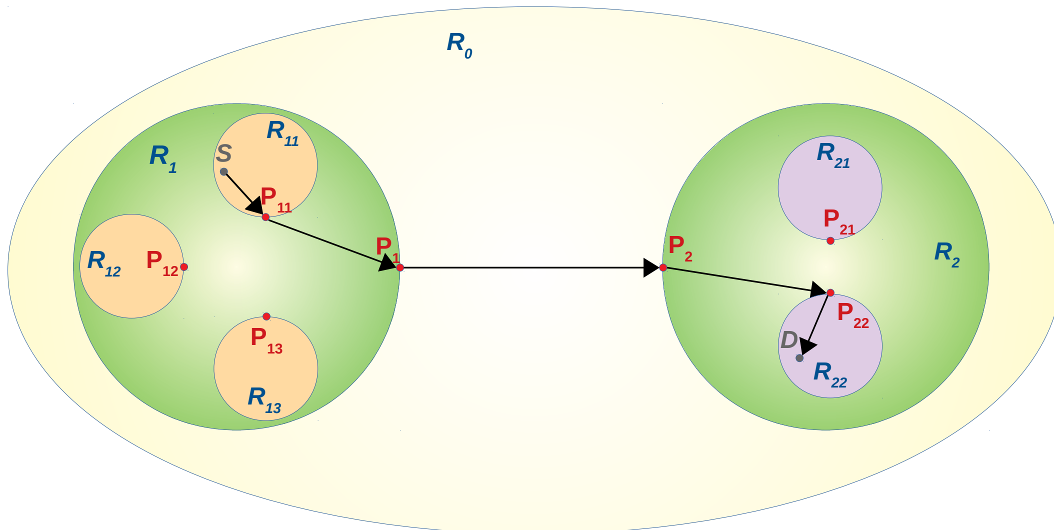

Figure 2.3: The source S and the destination D are in different regions, so the inter-regional routing algorithm is involved. The resulting inter-regional route is $S-P_{11}-P_1-P_2-P_{22}-D$ and its inter-regional section the same except the end points.

2.3 Rationale of the Hierarchical Inter-Regional Routing algorithm

The hierarchical structure just described shows clearly that with this design there is always just one inter-regional section (one possible sequence of passageways) between source and destination. Thus, since the inter-regional section is unique, we could wonder

why it is necessary to design an inter-regional algorithm to find it. In order to answer this question, we must first point out that:

1. As stated in RFC4838 [7] and RFC5050 [10], DTN EID are names not addresses. From a node EID it is therefore impossible to infer to which region the node belongs to.

2. The topology (i.e. the region tree and related border passageways) is assumed unknown to single entities in the scenario considered here. This design choice was made because, in order to improve the scalability, we want to rely on a distributed architecture with independent regions. In this structure, the entity in charge of the management of region R_X should only:

   (a) provide a contact plan only for nodes in its region (which can be done autonomously);

   (b) find an agreement with managers of upper and lower regions about which nodes should be chosen as border passageways.

In conclusion, although there is a unique inter-regional route from source to destination, this route is unknown to the source and must be built step-by-step. More precisely, we must discover:

(a) which region the destination belongs to;

(b) the inter-regional segment to reach it.

These two aspects, although logically distinct, are tackled together by the inter-regional routing algorithm presented in details the next sections.

2.4 Intermittent connectivity and reachability

In the previous description, we have implicitly assumed that is possible to conveniently transfer bundles on each regional leg ($S-P_1$, P_1-P_2,..., P_N-D). How can this be taken for granted, given that each leg, which can involve many intermediate nodes and DTN links, is affected by intermittent connectivity? By the way, this is why it is necessary to use CGR to find the route between $S-P_1$, or P_1 and P_2 and so on! In fact, the possibility of transferring bundles cannot be taken for granted for any given pair of nodes belonging to the same region. For the specific purposes of inter-regional routing, it is necessary to abstract from contacts, and to introduce the concept of "reachability".
We say that given two nodes, A and B, belonging to the same region (i.e, sharing the same contact plan) B is reachable from A only if, despite intermittent connectivity, it is possible to transfer a reasonable amount of bundles of reasonable size in a reasonable

time, from A to B. This triple condition is deliberately but also necessarily expressed in qualitative terms, as it is impossible to give a quantitative general definition of the three "reasonable conditions", valid for all possible deployments. The quantitative definition of them is thus left to network managers.
Note that reachability, for the sake of generality, is defined as unidirectional, although it will be bidirectional in most cases. In the following, not to make the explanation too clumsy, reachability will be assumed bidirectional; modifications for the unidirectional case are trivial and left to the interested reader.

2.4.1 Viable routes and passageways

A leg of an inter-regional route is said "viable" if its end nodes (e.g. $S - P_1$, or $P_1 - P_2$, ... ,or $P_N - D$) are "reachable". An inter-regional route is said "viable" if all its legs are viable.
In other words, to have a viable route each node of an inter-regional route, e.g. $S - P_x - P_y - D$, must be reachable from the adjacent ones. To this end, there are two conditions to be met:

1. the only passageway to the upper region and the possibly many passageways to lower regions must all be assumed "reachable" from all nodes of the region that can be source or destination of the inter-regional traffic. This to ensure that the terminal legs of the inter-regional paths $S - P_x$ and $P_y - D$ are "viable".
2. passageways of the same regions must all be reachable from/to each other. This to have the internal legs viable as well.

It is obvious that in a real deployment, great care must be posed to passageways selection in order to have viable inter-regional routes. The proposed inter-regional algorithm is based on this condition.

2.4.2 Dynamic topology and Internet comparison

The association of an EID to a region and the inter-regional route to get it from another region, once discovered, are assumed constant for a relatively long period. Therefore, the route can be used for inter-regional routing of subsequent traffic (form the same source or from other sources belonging to the same region).
For scalability reasons, it is necessary to avoid recalculating inter-regional routes for each bundle. At the same time, the inter-regional routing algorithm must be flexible enough to tackle small-scale variations of the topology, such as the creation of a new region, or the moving of one node to another region. For the sake of efficiency, these variations should happen rarely; in other words, the pace of these variations should be small with respect to the pace of inter-regional traffic.

There are, however, two important differences from our algorithm and routing on the Internet. First, Internet nodes (NIC cards) are identified by IP addresses, while DTN nodes by DTN EIDs, which are names and not addresses, as highlighted before. Second, in Internet dynamic routes are calculated a priori, i.e. independently of traffic and by means of a continuous exchange of information among nodes, while this is impractical in DTN networks. Therefore, the proposed algorithm will discover the routes only when and if there is a bundle to be routed from a source S to a destination D that belong to two different regions.

2.5 Overview of the HIRR algorithm

In order to make inter-regional routing decisions, each passageway holds a blacklist table containing the excluded passageways for each destination node. Let us consider the same topology shown in Figure 2.3 and start from an initial state where the blacklist tables are empty on every node. We want to send a bundle from S to D, belonging to R_{11} and R_{22} respectively, and show how HIRR learns the inter-regional route $S-P_{11}-P_1-P_2-P_{22}-D$. We call this route "forward route" because it is the route from S to D, in contrast to the reverse route $D - P_{22} - P_2 - P_1 - P_{11} - S$, which is called "backward route". Although the HIRR main goal is to learn the forward route, we will show that it can also learn the backward route without additional cost.
In Figure 2.4 we can see the firsts steps of the algorithm. Initially, S checks if its home region R_{11} contains D. As R_{11} does not contain D, HIRR is executed instead of CGR. As S is a terminal node (i.e., it is not a passageway), the only thing that it can do is to try to reach the passageway P_{11} (there is just one, as R_{11} is a leaf region). When P_{11} receives the bundle (red arrow from S in the Figure 2.4), it checks if one of its two regions contains D. In this case, neither its home region R_{11} nor its outer region R_1 contains D. If D exists, it can be reached only by exploring other regions.
This exploration starts with P_{11} sending a replica (by using CGR) of the bundle to: a) all passageways that can bring to the sub-regions of R_1, such as P_{12} and P_{13}; b) the sole passageway that can bring to the sole upper region R_1, i.e. P_1 (blue arrows in Figure 2.4). All these three passageways are the "peers" of P_{11}.
Of course, this example cannot show all the HIRR scenarios. For example, we started from a situation where all the blacklist table of each node are empty, and R11 is a leaf region. If P_{12} had been in the blacklist of P_{11} for the node D, the copy from P_{11} to P_{12} would have not been sent. If R_{11} had had one or more sub-regions, P_{11} also would have sent the replica to them.

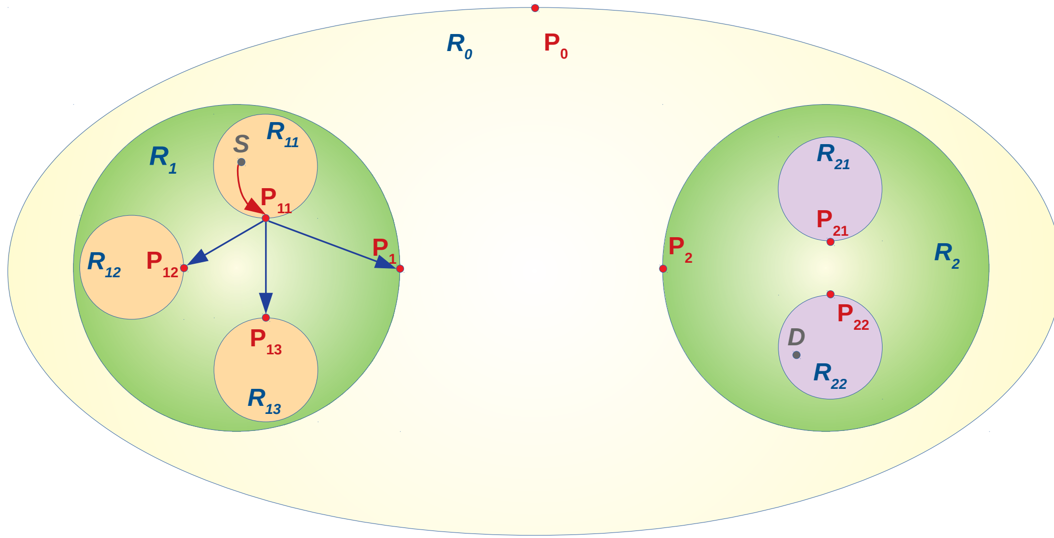

Figure 2.4: The source node S forwards the bundle to its home region passageway P_{11} (red arrow). Then, HIRR is executed instead of CGR on P_{11} because neither its home region (R_{11}) nor its outer region (R_1) contain D. Replica of the bundles are created and sent P_{12}, P_{13} and P_1. (flooding to peer passageways).

As soon as the bundle replica arrives they learn some information, even before execute CGR or HIRR:

- P_{12} learns that both P_{13} and P_1 do not contain S;
- P_{13} learns that both P_{12} and P_1 do not contain S;
- P_1 learns that both P_{12} and P_13 do not contain S.

This represents the backward route learning described above.
Let us continue with the description of the next phases of the Forward route learning. P_{12} checks if its home region R_{12} or its outer region R_1 contain D, which, unfortunately, is not the case. As R_{12} has not any sub-regions (it is a leaf), it is immediately clear that choosing P_{12} to reach D is wrong. For this reason, both P_{12} sends (by using CGR) a blacklist message to P_{11}. The same happens for P13 (see Figure 2.5). As soon as P_{11} receives both blacklist messages, it can create, learning by exclusion, the first leg $P_{11} - P_1$ of its inter-regional route (showed in Figure 2.3). This is the first result of the forward route learning. The situation is different for P_1. Initially it checks if its home or outer regions contain D but, unfortunately, neither its home region R_{12} or its outer region R_1 contain D. Unlike the previous case, however, P_1 has some peers that must be explored (i.e., P_0 and P_2). For this reason, unlike P_{12} and $P13$, P_1 cannot provide an immediate

feedback to P_{11}. In this case, as D belongs to R_{22} (which is a region reachable from P_1), it never provides an answer to P_{11}, as shown in figure 2.5.

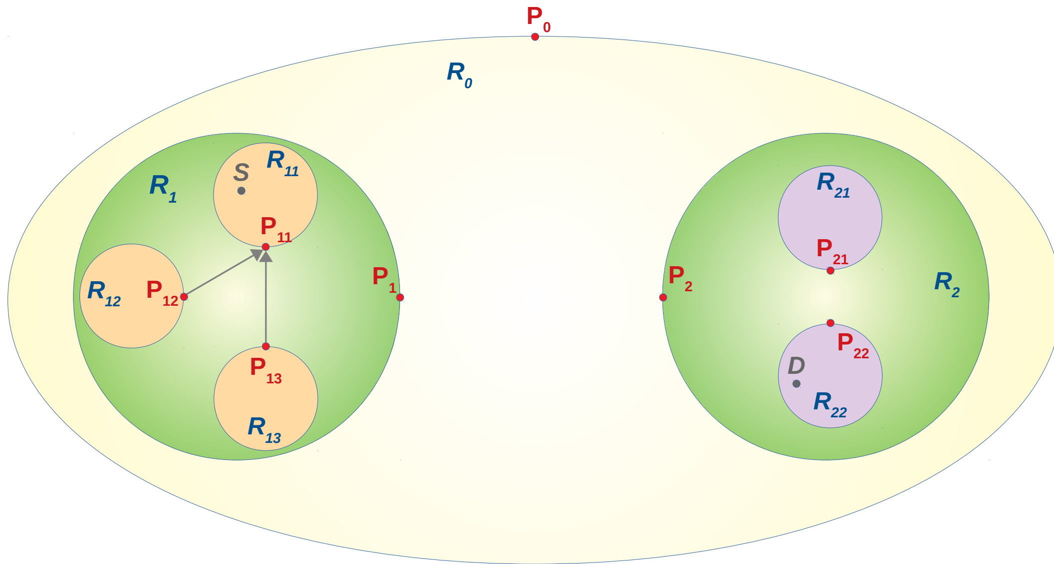

Figure 2.5: P_{12}, P_{13} and P_1 have received a bundle replica from P_{11}. Both P_{11} and P_{12} react sending a blacklist message to P_{11}. P_1 has not enough information to answer, therefore it waits to learn about its peers (i.e., P_0 and P_2).

Now, as described in Figure 2.6 with blue arrows, P_1 sends a replica of the bundle (using CGR) to P_0 and P_2. Using backward learning, P_0 learns that P_2 the node S is not reachable via P_2 and similarly P_2 learns that S is not reachable via P_0.
Similarly to what previously done by P_{12} and P_{13} in figure 2.5, P_0 first checks if D belong to its region R_0. Note that being R_0 on top of the tree, P_0, by contrast to all other passageways, has not an outer region. As D does not belong to R_0, P_0 then sends a blacklist message back to P_1 (see the gray arrow in Figure 2.6).

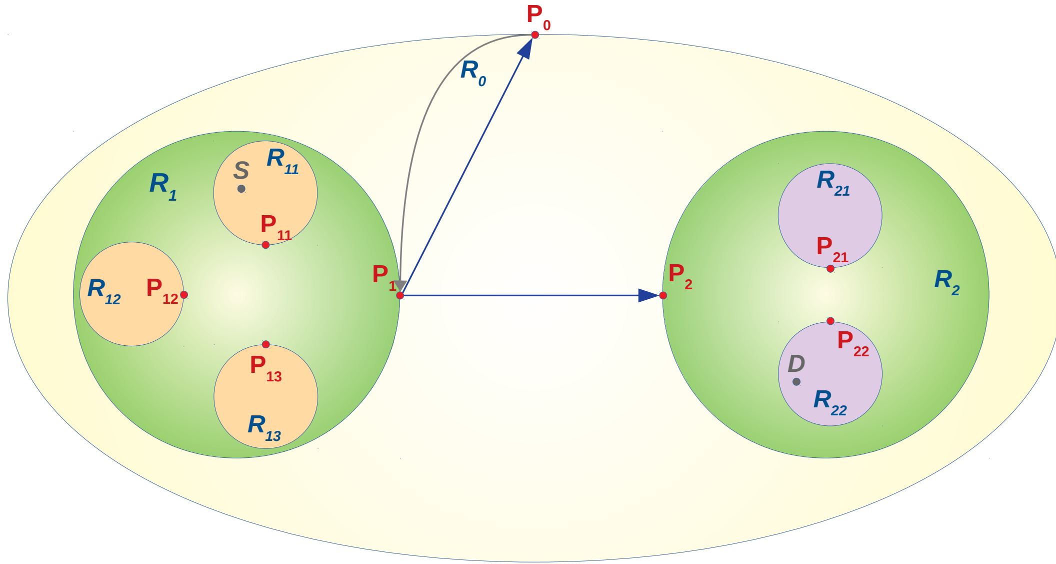

Figure 2.6: P_1 is the only passageway that can continue the forwarding. It sends a bundle replica to both P_0 and P_2 (blue arrows). As P_0 cannot continue the forwarding, after it receives the bundle from P_1, it forwards a blacklist message to P_1 (gray arrow).

As soon as P_2 receives its bundle replica, it first checks that D does not belong to its home region (R_2), then it has to explore its inner regions (its behavior is the dual of P_1, it has to explore below, instead of above), by sending a replica to P_{21} and P_{22} (see the blue arrows in figure 2.7). When P_{21} receives the replica, it learns (backward route learning) that S is not reachable via P_{22}. Similarly, P_{22} learns that S is not reachable via P_{21}.
As already happened several times before, after checking the presence of D in its regions (by checking its contact plan), P_{21} sends a blacklist message back to P_2, as shown by the gray arrow in Figure 2.7. By contrast, when P_{22} receives the bundle replica, the usual check for the presence of D within its contact plan succeeds, and the intra-regional routing algorithm (in this case CGR) can be used to reach D (see the red arrow in Figure 2.7). The first exploration is now complete. Note that, although it could appear logic to send back a positive confirmation ("I have it"), this is not strictly necessary, as the correct path can be learned at this point by exclusion (only one possibility is left to each passageway, as all others have been blacklisted).
Now each passageway has learned the next routing decision that follows directly the unique inter-regional route between S and D using the forward route learning. The direct consequence is that if the source sends another bundle, it will be routed directly to the destination, following its inter-regional route (see Figure 2.3). Moreover, if D wants to send a response back to S, as it is likely, there is no need to explore the

topology again because all intermediate passageways have already learned the route in the reverse direction (it can be seen by swapping the direction of the arrows in Figure 2.3) thanks to the backward route learning.

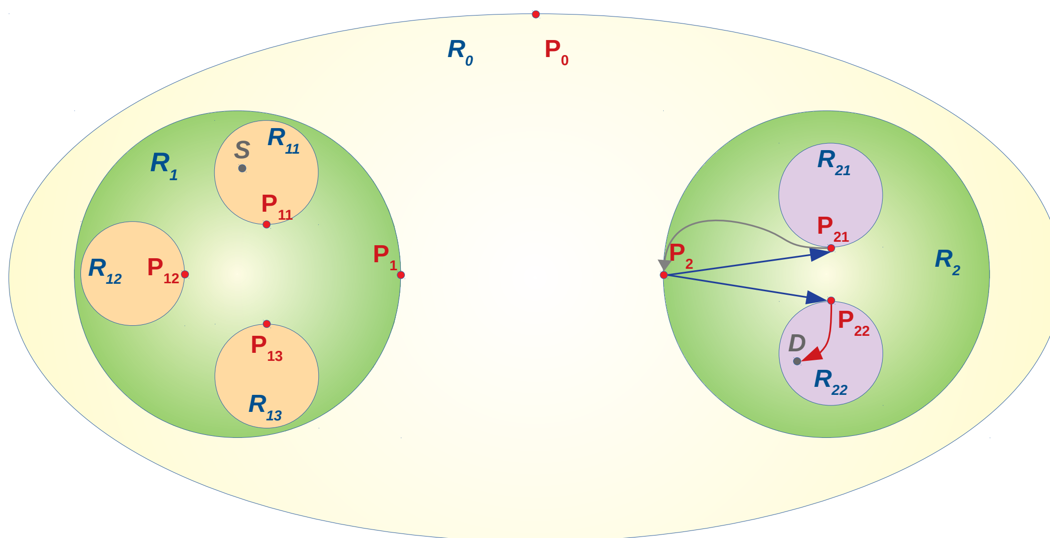

Figure 2.7: P_2 sends (by using CGR) a bundle replica to both P_{21} and P_{22} to explore regions R_{21} and R_{22}. Since D is not in R_{12}, and R_{12} is a leaf, P_{21} sends back a blacklist message to P_2. By contrast, P_{22} finds D in its home region, thus it can use CGR to reach it. P_{22} does not send back any positive acknowledgment, as not strictly necessary.

2.6 Algorithm implementation

2.6.1 Blacklisting

Each node maintains a list of entries (P, N), where P is a passageway and N is a generic node, which could also be a passageway. The meaning is: P has blacklisted N. The purpose of this blacklist is to prevent a node from forwarding future bundles to passageways that have declared they cannot reach the destination. This blacklist must be checked every time that a passageway wants to send a bundle replica to one of its peer passageways. If the blacklist contains the destination D related to the peer P, the bundle replica is not sent to that peer.

2.6.2 Search strategy

Given a source, this is the mechanism that propagates a replica of a given bundle across all the topology. The bundle replica is sent to the only peer passageways that are not blacklisted. If the destination is not blacklisted by any passageways, the bundle is globally propagated among every passageway of the topology, as shown in Figure 2.8. By contrast, if all the blacklist records are properly set the bundle follows the unique inter-regional route, as shown in Figure 2.3.
To this end, every bundle is equipped with a field named "prior passageway", which is a reference (e.g. its node name) to the previous forwarder passageway (if any).
In order to perform the exploration, a passageway who receives a bundle infers the provenance of prior passageway matching its name with the regions declared in the contact plan. In this case, the provenance can assume one of these three values:

- **home**: if the prior passageway belongs to the home region of the current passageway;
- **outer**: if the prior passageway belongs to the outer region of the current passageway;
- **unknown**: if the prior passageway is not defined. This happens only when the current passageway is the first of the inter-regional route.

For example, considering the inter-regional route $S - P_{11} - P_1 - P_2 - P_{21} - D$ of the topology shown above in Figure 2.3:

- the prior passageway of the bundle received at P_{11} is not specified, therefore its provenance value is *unknown*;
- the prior passageway of the bundle received at P_1 is P_{11} and its provenance value is *home*;
- the prior passageway of the bundle received at P_2 is P_1 and its provenance value is *outer*;
- the prior passageway of the bundle received at P_{22} is P_2 and its provenance value is *outer*;

Similarly, it is computed also the provenance value of every peer passageways. For example, using the same example as before (Figure 2.3) we have that:

- P_{12}, P_{13} and P_1 are peer passageways of P_{11} and their provenance value is *home*;
- $P_{11}, P_{12}, P_{13}, P_0$ and P_2 are peer passageways of P_1. The provenance value of P_{11}, P_{12} and P_{13} is *home*; the provenance value of P_0 and P_2 is *outer*.

- P_{21}, P_{22}, P_0 and P_1 are peer passageways of P_2. The provenance value of P_{21}, P_{22} is *home*; the provenance value of P_0 and P_1 is *outer*.

- P_{21} and P_2 are peer passageways of P_{11} and their provenance value is *home*;

The routing decision is made comparing the provenance value of the prior passageway with the provenance value of each peer passageway. The exploration rule is:
For each non-blacklisted peer passageway, if its provenance value is different from the provenance value of the prior passageway, a bundle replica is sent.
Note that for the peer passageways, the provenance value *unknown* is impossible because it is supposed that the peer passageways are always well defined in the contact plan. This implies that if the provenance value of the prior passageway is unknown, the bundle replica is spread in every direction (because the value *unknown* is always different from both *home* or *outer*). The Figure 2.8 resumes the spreading of the bundle replicas described above and introduced step-by-step in the previous sections.

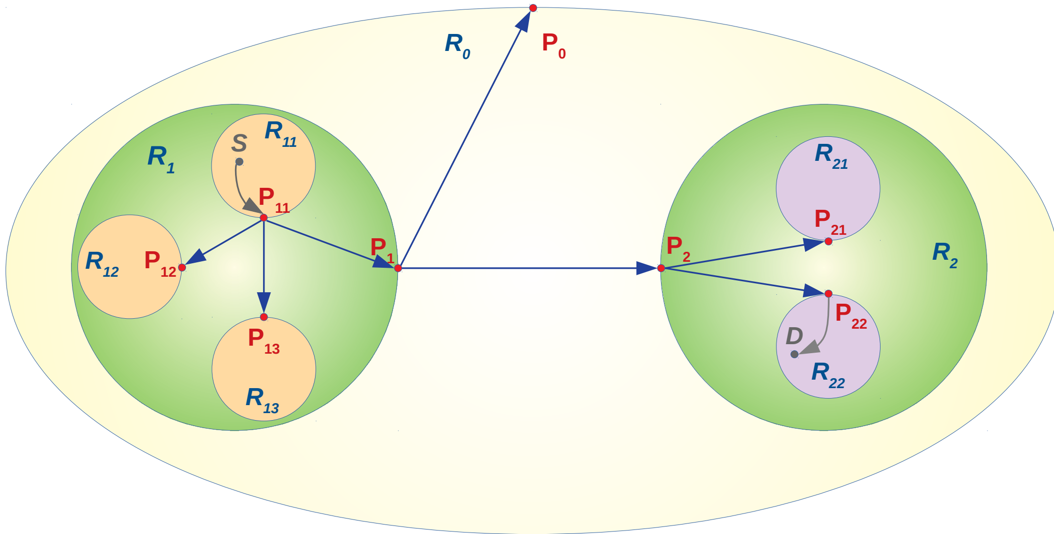

Figure 2.8: The source node S belongs to the region R_{11} and the destination node D belongs to the region R_{22}. S initially tries to reach D using CGR, but it fails because D is not in its home region. The blue arrows show how the bundle is forwarded between the passageways. When P_{22} receives the bundle it can finally forward it to D. Note that CGR is used in every forwarding step between two adjacent passageways.

The Figure 2.8 resumes the spreading of bundle replicas described above and introduced step-by-step in the previous sections. It is important to note that the blue arrows in that figure form a spanning tree of the topology, as shown in Figure 2.9.

What happens to the spanning tree if we move the destination from R_{22} to R_{21}? Nothing: the spanning tree is always the same! The spanning tree that can be generated by P_{11} is the same no matter which is the destination, therefore it is an intrinsic property of P_{11}. As a consequence of the rules defined for the exploration, this spanning tree is unique. These considerations can be extended to each passageway of the topology. The exploration can be seen as the theoretical problem well known as "tree traversal", i.e. the process of visiting each node in a tree data structure exactly once. The described search strategy can be seen as a procedure that explores the topology using the breadth-first approach.

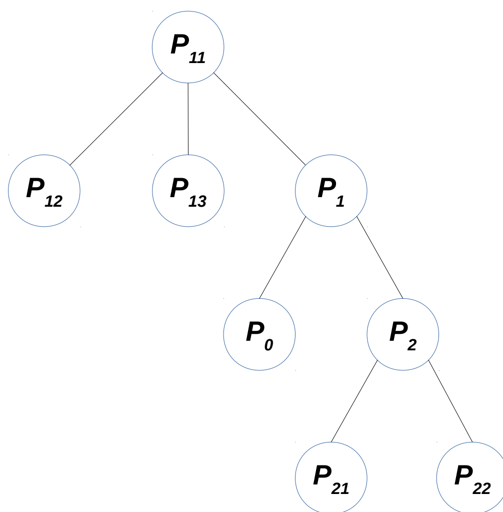

Figure 2.9: As the search strategy breaks all the loops it creates a spanning tree among the passageways involved. Every source node in the same region generates the same spanning tree, which does not depend on the destination node.

2.6.3 Backward route learning

In the previous sections, we have seen how each passageway learns the backward path. When a node receives a bundle, each peer passageway (except for the prior one) cannot belong to a region which contains the source node. In other words, the prior passageway is for sure the only one that can forward something addressed to the current bundle source node. For this reason, is possible to add to the blacklist the entry $(P_x, source)$, for each P_x different from the prior passageway.
For example, in figure 2.7 we have seen that P_2 learns using the backward route learning that P_0, P_{21} and P_{22} are not part of the inter-regional route from D to S.
Looking at the topology from a global point of view, we have seen that after sending a

bundle from a source S to a destination D, the network learns the inter-regional route from D to S.
However, the backward route learning is more powerful than this: if the bundle traverse the entire topology, the network learns also how to reach S from any source node.
In order to understand how much the backward route learning is effective, it is possible to compute a lower bound for the minimum number of different bundles required to reach the convergent state, using the sole backward route learning. Let us assume that all the nodes belong to a region that doesn't have nested regions. This assumption is optimistic, so it is the reason why it is a lower bound instead of a precise value. Lets suppose to have M nodes in the topology, called $N_1, N_2, \ldots, N_M$. If $N_1, N_2, ..., N_{M-1}$ send a bundle to N_M, every passageways learns the inter-regional route from $N_1, N_2, ..., N_{M-1}$ to N_M, by using the backward route learning. From this moment, every bundle sent from N_M follows the unique inter-regional route to the destination. In order to create the missing routes, we have to send a bundle from N_M to all the other ones ($N_1, N_2, ..., N_{M-1}$). We have sent:

- $N - 1$ bundles sent from $N_1, N_2, ..., N_{M-1}$ to N_M
- $N - 1$ bundles sent from to N_M to $N_1, N_2, ..., N_{M-1}$.

The total number of sent bundles is $2(N - 1)$, which is a lower bound for the minimum number of different bundles required to reach the convergent state, using the sole backward route learning.

2.6.4 Forward route learning

As we have seen in Figure 2.5, a node that receives a bundle replica can send a blacklist message to the prior passageway.
A passageway can send blacklist signals for two reasons:

1. after it receives a bundle replica it is unable to perform any forwarding. In this case, a blacklist signal is sent back to the prior passageway.
2. after it receives a blacklist signals from all the nodes in one of its region (home or outer). In this case, blacklist signals are sent to every passageway with the provenance value different to the provenance value of the prior passageway.

The former case always happened in the previous examples (see Figures 2.5, 2.6 and 2.7) and it is resumed by the Figure 2.10. The latter case is required when there is a region that contains other sub-regions, which is never crossed by the inter-regional route. If this happens, a recursive blacklisting is triggered from the leaf sub-regions and the route can be learned correctly.
For example, as shown in Figure 2.11, if we send a bundle from a source S belonging to

R_{11}, to a destination D belonging to R_{13}, the region R_2 is never crossed by the inter-regional route that we want to learn (i.e., $S - P_{11} - P_{13} - D$). In order to learn the leg $P_{11} - P_{13}$, the passageway P_{11} have to add in its blacklist P_1 and P_{12}. Nothing new for P_1, which sends a blacklist message to P_{11} as usual (shown in Figure 2.11 using gray arrows). To send a blacklist message from P_1 to P_{11}, we need that P_1 has blacklisted both P_0 and P_2; similarly P_2 waits to receive both blacklist messages from P_{21} and P_{22} (as shown in Figure 2.11 using orange arrows). For this reason, the recursion is triggered by the passageways of leaf sub-regions (P_{21} and P_{22}) and it propagates back until a blacklist message is received by P_{11}.
The only question left is: why the recursion step is propagating blacklist messages to the entire region instead of the answer only to the prior passageway?
As this rule is triggered by the receiving of a blacklist message, we have lost track of the passageway who have sent the bundle replica. For this reason, the blacklist messages are propagated to the entire region. We found two ways to solve this problem:

1. introduce a table that keeps track of the processed inter-regional bundles, and retrieves from this table the passageway that has sent the replica;

2. each passageway injects the traveled path inside the bundle replicas, and retrieve from this field the passageway that has sent the replica.

As both solutions add a lot of complexity and the information carried by extra blacklist messages is correct (even if not strictly required), we decided to accept this bandwidth wasting and keep the algorithm simple.
The forward route learning guarantee to let the network learns the direct route from S to D. This technique does not interfere with the backward learning, therefore HIRR implements both.
In order to understand how much the forward route learning is effective, it is possible to compute a lower bound for the minimum number of different bundles required to reach the convergent state, using the sole forward route learning. Lets suppose to have M nodes in the topology, called $N_1, N_2, \dots, N_M$. In the worst case, it is required to combine without repetitions all the M nodes, taken two-by-two and considering the order. The total number of sent bundles is $M(M-1)$, which is a lower bound for the minimum number of different bundles replica required to reach the convergent state, using the sole forward route learning.

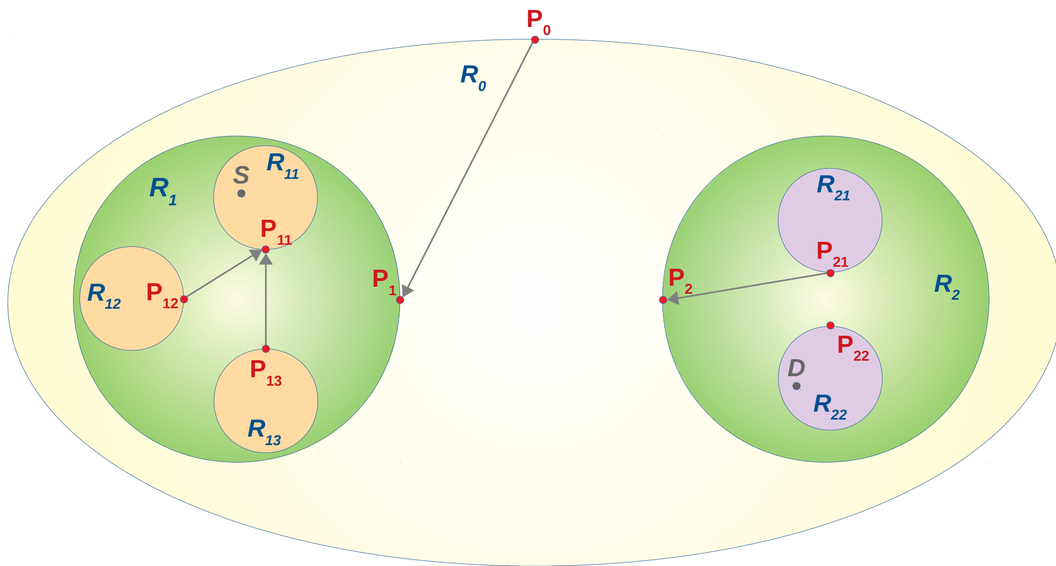

Figure 2.10: The node S, which is part of R_{11}, sends a bundle to D, which is part of R_{22}. The gray arrows show the blacklist message sent by the passageways. This Figure resumes the blacklisting shown in Figures 2.5, 2.6 and 2.7.

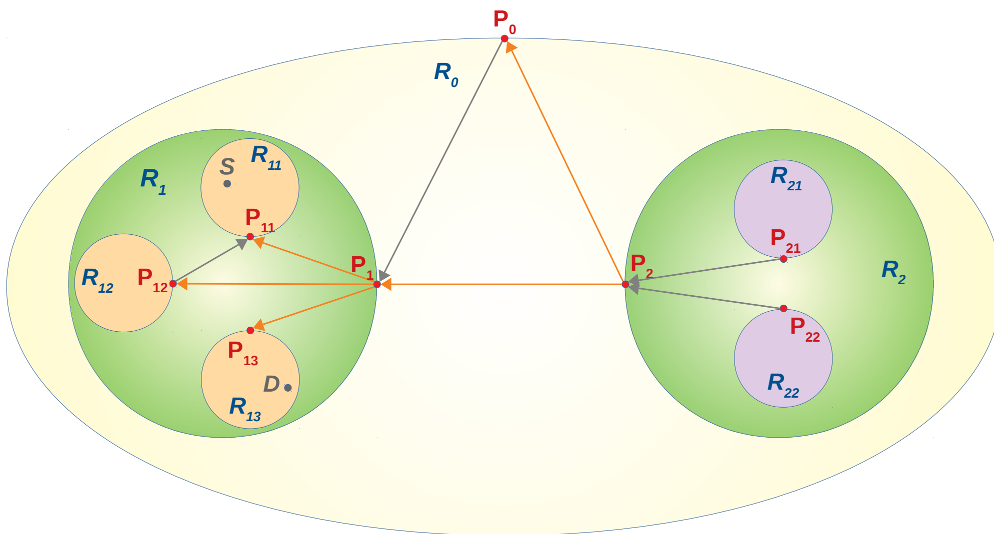

Figure 2.11: The node S, which is part of R_{11}, sends a bundle to D, which is part of R_{13}. The gray arrows are blacklist signal generated by the former case, the orange ones are generated by the latter case (i.e. recursion).

2.6.5 Pseudocode

Description: This function implements the backward route learning described above. In order to get the most benefit from this function, it must be invoked before executing any intra-regional routing algorithm.

```
procedure BACKWARDLEARNING(bundle)
    source ← bundle.sourceNode
    priorPw ← bundle.priorPassageway
    for each passageway in PeerPassagewayList do
        if passageway ≠ priorPw then
            Add to blacklist the entry (passageway, source)
```

Description: This function combines the search strategy described above with the forward route learning. For this reason, by first it sends a bundle replica to every non-blacklisted passageway with opposite provenance respect to the prior passageway provenance. If nothing was sent, it forwards a blacklist bundle to every passageway within the same region of the prior passageway.
This function must be invoked if the contact plan of the current node does not contain any reference to the destination node, i.e. if the destination is not part of any of the current node regions.
Note that this is a "best effort" version of HIRR: if a CGR failure occurs, this procedure continues its execution, returning a failure value at the end of the function, to inform the application that something went wrong.

```
function interRegionalDataBundleHandling(bundle)
    source ← bundle.sourceNode
    dest ← bundle.destinationNode
    priorPw ← bundle.priorPassageway
    nothingSent ← TRUE
    outcome ← SUCCESS
    if am I a terminal node? then
        return CGRforward(homeRegionPassageway, bundle)

    for each peer in PeerPassagewayList do
        if blacklist contains the entry (peer, dest) then
            continue
        if priorPw.provenance = peer.provenance then
            continue
        if CGRforward(peer, bundle) = FAILURE then
            outcome ← FAILURE
        nothingSent ← FALSE

    if nothingSent = TRUE then
        blacklistBundle ← createBlacklistBundle(dest)
        if CGRforward(priorPw, blacklistBundle) = FAILURE then
            outcome ← FAILURE
    return outcome
```

Description: This function must be invoked when a blacklist bundle arrives. The goal is, if it is necessary, to recursively propagate the blacklist signal. The function will return a success value if the incoming blacklist bundle was processed correctly; otherwise, it will return a failure.

```
function interRegionalBlacklistBundleHandling(bundle)
    source ← bundle.sourceNode
    dest ← bundle.destinationNode
    priorPw ← bundle.priorPassageway
    outcome ← SUCCESS
    if blacklist contains the entry (priorPw, dest) then
        return SUCCESS
    Add to blacklist the entry (priorPw, dest)
    if currentRegion contains dest then
        return SUCCESS
    if blacklistContainsAll(dest, priorPw.provenance) then
        for each peer in PeerPassagewayList do
            if blacklist contains (peer, dest) then
                continue
            if priorPw.provenance = peer.provenance then
                continue
            blacklistBundle ← createBlacklistBundle(dest)
            if CGRforward(peer, blacklistBundle) = FAILURE then
                outcome ← FAILURE
    return outcome
```

Description: This function returns true if all peer passageways that belong to the specified *provenances* have blacklisted N, 0 otherwise.

```
function blacklistContainsAll(N, region)
    for each peer in PeerPassagewayList do
        if priorPw.provenance ≠ peer.provenance then
            continue
        if blacklist doesn't contains (peer, N) then
            return FALSE
    return TRUE
```

2.7 HIRR Variants

2.7.1 Auto rediscovery

If a source node S sends a bundle to a destination D that does not belong to the network (never existed or removed), each node will eventually learn that there is not any passageway with an inter-regional route to D.
The problem is that if D later appears in the network, no one tries to reach it. A trivial solution could be the following: at the beginning of the InterRegionalForward function, it should be checked if the bundle is destined to a "dead end" (i.e. if D is banned from all the peer passageways). In this case, all entries (P_x, D) should be removed from the blacklist and the algorithm re-run as the first time. The disadvantage is that this solution exposes the architecture to a denial of service (DoS) attack if a node deliberately kept injecting bundle destined to a non-existent destination, as each attempt would trigger flooding of the network. This point needs further investigation.

2.7.2 2-way handshake learning

Instead of sending blacklist messages, there is another way to learn the inter-regional routes using the sole backward route learning.
Suppose we want to learn both the forward and backward inter-regional routes between S and D. The whole process starts with a bundle probe spreading to the entire topology, similarly as shown in Figure 2.8. Unlike before, here no passageway sends back any blacklist message. When the probe starting from S arrives at D, by using the backward route learning, all the passageways have learned the inter-regional route from D to S. As we are not sending any blacklist messages, the inter-regional route from S to D is not learned yet. In order to learn this route, D sends back to S another probe. As the inter-regional route from D to S is formed, this response does not flood the network, but it follows its unique inter-regional route, as shown in Figure 2.12. When S receives this probe, the inter-regional route from S to D is being learned.
At the end of the process, this 2-way handshake produces very similar results to standard HIRR. Apparently, this procedure seems to be way better than the standard HIRR, which propagates back the blacklist messages every time and also sometimes it needs a recursion, wasting more bandwidth.
One of the problems of this approach is that, until S receives a response, it can not send any bundle to D. As the DTN networks are often characterized by long RTT, this procedure is a bottleneck that freezes the routing until the inter-regional route is discovered. Another problem is that if the D does not exist, S can wait indefinitely and a timeout must be set to give up assuming that the node is not part of the network. Finding the timeout value is critical because: a) if it is underestimated, it produces wrong results because if the timer expires the passageway assumes that the destination

does not exist while the answer has not arrived yet; b) if it is overestimated, the bundles addressed to D are pending for a long time and this can saturate the buffers.
This variant can be a starting point for future works, but at present, it needs further investigation.

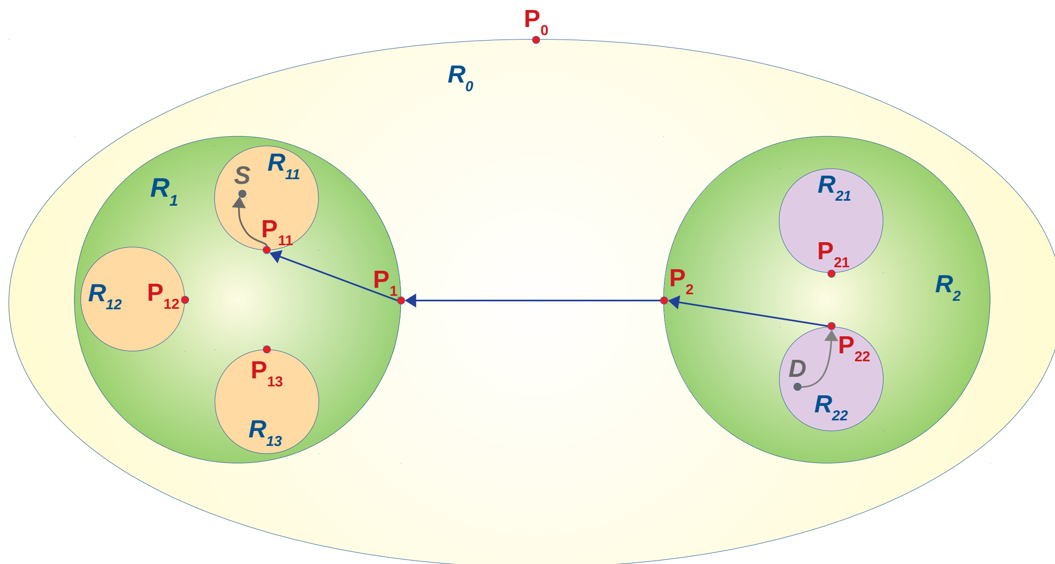

Figure 2.12: After receiving the probe, D respond to S with another probe, which follows the unique inter-regional route learned thanks to the previous one.

2.8 Algorithm Analysis

2.8.1 Level representation

Despite the hierarchical representation described above is intuitive, there is another way to represent the topology, organizing the passageways by levels.
An example is given by Figure 2.13, which shows the same topology used in all the previous examples (e.g. Figure 2.3). Each node of the graph represents a passageway and the arcs between two nodes are their bidirectional reachability. This results in less intuitive representation respect to the previous one, but it has the advantage of keeping simple the formal analysis of HIRR.
The rules for drawing this graph are the following:

1. each passageway can be connected to just one higher level node;

2. each passageway can share the same higher level node with other nodes from the

same level; when this happens, all the involved nodes are fully connected one to each other;

3. each passageway can have connections to many nodes from the lower level;

4. connections from non-adjacent levels are not allowed.

The graph obtained applying these rules (e.g. Figure 2.13) has a shape that reminds a tree, but because of the presence of horizontal connections, the graph is not a tree, therefore is not possible to use the same terminology.
In this graph a passageway can have one connection to only one higher level node (analogous to a "father" for the trees), many connections to nodes from the same level (analogous to the "siblings" for the trees) and many connections to nodes from lower level (analogous to "children" for the trees).
Note that, from the perspective of each passageway, the union of the passageways in the same level with the ones in the higher level represents the outer region, while the passageways in the lower layer represent the home region. The Figure 2.9 shows that the search strategy creates a spanning tree of this graph and the inter-regional route is the only existing path between the first passageway (i.e., the root of the tree) and the last passageway.

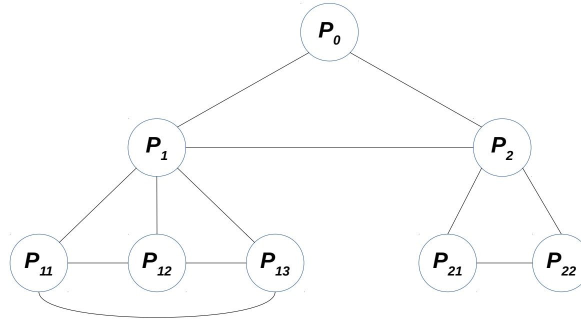

Figure 2.13: In the level representation, the resulting graph is organized by levels. Each node has visibility of exactly 3 levels called: "same level", "higher level" and "lower level". For example, from the perspective of the node P_1 we have that: P_2 belongs to its same level, P_0 belongs to its higher level and P_{11}, P_{12} and P_{13} belong to its lower level.

2.8.2 Derivation

The level representation was useful to derive the final version of the exploration strategy. After conjecturing various rules that allow to each start node to traverse the graph without cycles, a brute-force program was written to validate the results. The idea is to

create an algorithm capable to visit all the nodes once, by taking advantage of the full connection between levels.
A minimal version of the exploration strategy is the following:

```
function searchStrategy(source, priorPassageway)
    for each peer in PeerPassagewayList do
        if skipForwarding(bundle) then
            continue
        forwardBundle(peer, bundle)
```

The goal of the brute force program is to derive a *skipForwarding*() function that allows visiting all the nodes once without cycles. We can notice that the domain of the variables is finite, therefore is possible to represent the *skipForwarding*() function as a truth table. A topology is given as input, and the brute force tries every possible truth tables. A truth table is considered valid if it represents a function able to visit each node of the topology without cycles, no matter which is the source or the destination.
In particular, we have:

- The *skipForwarding*() function returns two values: $\{\text{TRUE}, \text{FALSE}\}$.
- The domain of the peer passageway provenance is: $\{lower, same, higher\}$.
- The domain of the prior passageway provenance is: $\{unknown, lower, same, higher\}$.

The truth table can be written as a table with three columns, the first two for the prior and peer passageways level and the last one for the outcome of the skipForwarding() function. As the domain size is 2,3,4 respectively for the output of *skipForwarding*() function, peer passageway provenance and prior passageway provenance, the number of possible truth tables for the *skipForwarding*() function is $2^{3*4} = 4096$. Since this number is small, the brute force approach is one of the faster ways to solve the problem. In the end, it emerged that if the topology is trivial several solutions may exist, but if it is sufficiently complex only one solution exists. The solution is resumed on the following truth table.

Prior passageway level	Peer passageway level	skipForwarding
unknown	lower	FALSE
unknown	same	FALSE
unknown	higher	FALSE
lower	lower	TRUE
lower	same	FALSE
lower	higher	FALSE
same	lower	FALSE
same	same	TRUE
same	higher	TRUE
higher	lower	FALSE
higher	same	TRUE
higher	higher	TRUE

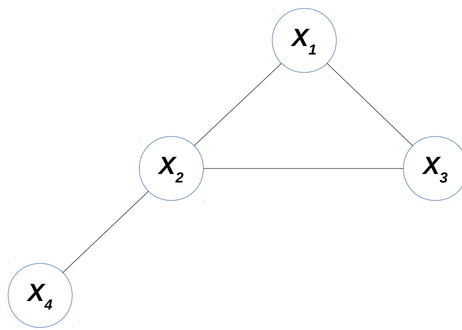

Figure 2.14: The smallest topology where the brute force gives a general unique solution.

The first observation about this truth table is that the values "same" and "higher" are always interchangeable. For this reason, we collapsed these two values in only one value. This observation is the key point that allowed us to define the idea of "home region" and the "outer region". The home region corresponds to the lower level. The outer region corresponds to the union between the same and the higher levels. Using these substitutions, the previous truth table is simplified and it carries the same information as before.

Prior passageway provenance	Peer passageway provenance	skipForwarding
unknown	outer	FALSE
unknown	home	FALSE
outer	outer	TRUE
outer	home	FALSE
home	outer	FALSE
home	home	TRUE

Analyzing this new truth table, it becomes clear that the $skipForwarding()$ function returns true if the prior passageway provenance is equal to the peer passageway provenance, false otherwise.

Chapter 3

JRegion prototype

3.1 Motivations

During the developing of the HIRR algorithm, it was necessary to implement and test several approaches to the inter-regional routing problem. Unfortunately, current DTN implementations (including ION) are not designed for rapid prototyping purposes.
In particular, writing experimental code for ION is challenging because:

- ION is written in C, therefore the memory is managed manually;
- the software is complex as it implements a lot of features that are irrelevant to the development and testing of the inter-regional routing algorithm.
- starting a single node requires to launch several daemons, which implies that starting different nodes on the same machine is a slow process (up to 10 seconds each); moreover, the total amount of memory is high;
- each node needs its own configuration files, which makes the managing of a high number of nodes impractical.

JRegion is a prototype written in Java specifically designed to avoid the problems described above.
The main goal of this prototype is to make possible a rapid evaluation of the inter-regional algorithms that can be implemented on top of CGR. JRegion is not intended as an alternative DTN implementation and is designed to run all nodes only on a single machine. Since we need to simulate several independent DTN nodes using as less memory as possible and making them easy to create and to destroy, each DTN node is implemented as a single thread. The communication between threads is made by Java *BlockingQueues*, which provide for the producer-consumer synchronization.
Each thread manages an input queue (i.e. its induct) and a list of the input queues of

its peer nodes (i.e. its outducts).
As JRegion is not a DTN implementation, these queues are not scheduled and they are supposed to be always available. At present, the prototype is not able to emulate the channel intermittencies, but it can be easily extended if necessary. Moreover, the prototype also lacks the CGR implementation, which is replaced by a fake intra-regional routing algorithm that is always able to reach the desired node (if in the same region). Of course, if CGR were used, the prototype should also handle the case of a CGR failure.
As the prototype always run on a single machine, an interpreter was designed to dynamically interact with the system. When running, it provides a command line as a user interface. The interpreter accepts several commands, for example to define the topology, to add or remove nodes, to send a bundle from a source to a destination etc. It can also read instructions from a file, therefore any interaction with the prototype can be scripted.
JRegion is designed to be highly extensible using Java class inheritance and polymorphism. In fact, the prototype contains several experimental inter-regional routing algorithms, among which HIRR was eventually selected thanks to its advantages.
By contrast to ION, the compilation is usually performed in background by the IDE and its time is negligible. Moreover, efficient memory management is out of the scope of this prototype, therefore we can rely on the Java garbage collector without managing the memory.
The following table resumes the differences between ION and JRegion.

	ION	**JRegion**
Language	C	Java
Memory management	manual (SDR,PSM, ...)	automatic (Java garbage collector)
Compilation time	up to a few minutes	negligible
Each node runs	a set of processes	one thread
Memory usage for each node	high (few megabytes)	low (few kilobites)
Time required to start a node	up to 10 seconds	few milliseconds
Configuration	different for each node	centralized
DTN stack implementation	full stack	not implemented yet
CGR implementation	present	not implemented yet
Nodes can be instantiated to	same or different machines	same machine

3.2 Software architecture

3.2.1 Software model description

- **Bundle class:** it is a minimal representation of a bundle, which has no payload and contains only the essential information for the inter-regional routing.

More precisely, it contains:

- a bundle identifier, stored in the *ID* attribute;
- references to the source, the destination and the prior passageway, stored respectively in the *source*, *destination* and *priorPassageway* attributes;
- the bundle type, stored in the *type* attribute, which can be "DATA" (for regular bundles) or "BLACKLIST" (for the passageway blacklist messages).

This class declares only the constructor, getter and setter methods.

- **Duct class:** it is a minimal representation of a duct. As JRegion does not contain any abstraction regarding contacts, each connection is assumed stable and without disruption. Even if this may appear as a strong assumption, it may represent a realistic scenario because the inter-regional routing algorithms are executed on top of CGR, which is in charge of managing scheduled link intermittency.
This class implements the communication between nodes exchanging bundle class instances between threads, using the producer-consumer synchronization. In order to keep the implementation simple, the direction (input or output) of ducts is not implemented. Each node has a reference to an input queue (i.e. its induct) and also to a list of output queues (i.e. its outducts). This class has an attribute declared as a *BlockingQueue* (which is an interface provided by Java) and it is instantiated by the concrete class *LinkedBlockingQueue*, which implements a thread-safe queue with infinite capacity. The Duct class has only two methods: *receive*() and *send*(). If the queue is empty the receive() method is blocking. As the queue has infinite capacity, the send() method is never blocking. This class has no getters and setters because it manages the access to the only class attribute (queue), which must be protected from external usages.

- **Region class:** it is a representation of regions. More precisely, it contains:

  - the name of the region, stored in the *name* private attribute;
  - a list of node that belongs to the region, stored in the *nodes* private attribute.
  - a reference to its enclosing region, stored in the *enclosingRegion* private attribute;
  - a list of nested regions, stored in the *nestedRegions* private attribute;

  Despite the fact that regions can be seen as nodes of a tree, to avoid any ambiguity we will reserve the term "node" to the DTN nodes. Thus, we will use the term *enclosingRegion* to refer to the father node of a region (i.e., its encompassing region), and *nestedRegions* to the children nodes (i.e., its sub-regions).
  This class contains one constructor and multiple getter and setter methods. It also contains the following methods needed for the region management:

- ```
  private void setEnclosingRegion(Region region)
  ```
  This method sets the enclosing region of the current region.
- ```
  public void addNestedRegion(Region region)
  ```
  This method adds a nested region in the current region.
- ```
  public void addNode(Node node)
  ```
  This method adds a node in the current region.
- ```
  public void removeNode(Node node)
  ```
  This method removes a node from the current region.

- **Node class:** it is a minimal implementation of a DTN node. In more details, it contains:
  - a node name, in the *name* private attribute ;
  - a reference to its region, in the *region* private attribute;
  - a reference to its induct, in the *induct* private attribute;
  - a list of vents (always empty for the terminal nodes), in the *vents* private attribute;
  - a blacklist, in the *blacklist* private attribute;
  - a registry of received bundles, in the *receivedBundles* private attribute;
  - a reference to the inter-regional algorithm in use, stored in *irrAlgorithm* private attribute.

  The *Node* class is one of the most complex classes in the prototype. Each *Node* instance runs a thread that emulates a DTN node: every thread listens on the node induct waiting for new bundles. When a bundle arrives, it can be processed in different ways, depending on the situation:
  - the bundle is discarded, if it is a duplicate of a bundle previously received;
  - the bundle is delivered, if the node match with the bundle destination;
  - the intra-regional routing algorithm is called, if the destination belongs to the same region;
  - vice versa, the inter-regional routing algorithm is called, if the destination does not belong to the same region.

  This class contains one constructor, getter and setter methods. It contains also the following public methods needed for the management of a node:

- ```java
  public void addVent(Node vent)
  ```
  This method adds the specified vent in the current node.

- ```java
  public void removeVent(Node vent)
  ```
  This method removes the specified vent from the current node.

- ```java
  public void addBlacklistEntry(Node P, Node N)
  ```
  This method adds the entry (P, N) in the blacklist of the current node.

- ```java
  public void clearBlacklist()
  ```
  This method deletes all the blacklist entries of the current node.

The incoming bundles are processed with several methods, including the following:

- ```java
  @Override
  public void run()
  ```
  It is the thread main function, overridden from the Java Thread class. It contains an infinite loop that uses the *receive*() function to retrieve a bundle from the induct. After the incoming bundle arrival, this function manages the bundle calling the following methods: *standardDataBundleHandling*() *interRegionalDataBundleHandling*() and *blacklistBundleHandling*().

- ```java
  private boolean standardDataBundleHandling(Bundle
      bundle)
  ```
  This method performs the regular bundle processing. It attempts to treat the incoming bundle as a regular bundle. First, if the current node is the bundle destination, the bundle is delivered and this function returns true. Second, it calls the *backwardRouteLearning*() method. Third, if the destination belongs to the same region of the node, perform the intra-regional routing algorithm. If all of these attempt fails, the method returns false and the *interRegionalDataBundleHandling*() method is called.

- ```java
  private void interRegionalDataBundleHandling(Bundle
      bundle)
  ```
  This method is called if the regular DTN processing (described above) fails. Its behavior is described in detail in the "pseudocode" section of the previous chapter.

- ```java
  private void
      interRegionalBlacklistBundleHandling(Bundle
      bundle)
  ```

This method is called when a blacklist bundle is received. Its behavior is described in the "pseudocode" section of the previous chapter.

- ```
  private boolean intraRegionalForwarding(Bundle
     bundle)
  ```

  It simulates an intra-regional algorithm (e.g., CGR). If the environment variable $HOT\_POTATO\_ROUTING\_ENABLED$ is set to *true*, the bundle is forwarded to a random node within the same region of the current node. This process is repeated for all the nodes inside the region until the bundle reaches its destination. By contrast, if $HOT\_POTATO\_ROUTING\_ENABLED$ is set to $false$ the bundle is forwarded directly to the destination. This function returns a success value if the destination node belongs to the same region of the current node. Otherwise, it returns a failure value.

- **IRRAlgorithm class:** It is an abstract class used to implement the inter-regional routing algorithms. Each algorithm concrete instance can be created using the factory method *makeAlgorithm*(). The concrete algorithms extend this class overriding the missing methods:

  - ```
    public abstract List<Node>
       getDataFloodingNodes(Bundle bundle)
    ```

    This method returns a list of the vents selected for the bundle data forwarding.

  - ```
    public abstract List<Node>
       getBlacklistFloodingNodes(Bundle bundle)
    ```

    This method returns a list of the vents selected for the blacklist messages forwarding after receiving a data bundle.

  - ```
    public abstract List<Node>
       getBlacklistFloodingRecursionNodes(Bundle bundle)
    ```

    This method returns a list of the vents selected for the blacklist messages forwarding after receiving a blacklist bundle. This method is basically responsible for the implementation of the recursions after receiving a blacklist message.

  There is also the *isDeadEnd*() method used to implement the "auto rediscovery" variant described in the previous chapter. Overriding this method is optional because it has a default implementation. At present, JRegion contains several concrete classes corresponding to all the experimental inter-regional routing algorithms developed and tested. One of them is called *HirrAlgorithm* and it implements the *HIRR* algorithm presented in this thesis.

3.2.2 Running environment description

- **Environment:** This class represents the global view of a running instance of the software managing the declared regions and nodes of a given topology.
  The declaration of regions and nodes is realized using two java *HashMap* attributes. The first one, named *regions*, represents the declared regions within the environment creating a correspondence between the name of each region and its corresponding object. The second one, named *nodes*, represents the declared nodes within the environment creating a correspondence between the name of each node and its corresponding object. Since nodes and regions can be added and removed dynamically from the environment, this class is responsible for keeping coherent the overall state of the declared nodes and regions. It contains also the following set of global variables used by the entire program:

  - ```
    public static boolean HOT_POTATO_ROUTING_ENABLED
    ```
    If this variable is set to true, the intra-regional routing procedure forwards the bundle to a random node within the same region. This process is repeated until the destination is reached. If this variable is set to false, the bundle is always forwarded directly to the next inter-regional hop or to the destination (if it belongs to the same region of the current node). This can be used in order to emulate the CGR routing inside a region. The default value suggested is false.

  - ```
    public static boolean BUNDLE_REGISTRY_ENABLED
    ```
    it is used in order to break the loops. If this variable is set to true, each node remembers the transit of each bundle. If a bundle arrives twice, it will be discarded. The default value suggested is true.

  - ```
    public static boolean BACKWARD_LEARNING_ENABLED
    ```
    If this variable is set to true the backward learning is performed, it is skipped otherwise. The default value suggested is true.

  - ```
    public static boolean AUTO_REDISCOVERY_ENABLED
    ```
    If this variable is set to true the "auto rediscovery" variant explained in the previous chapter is enabled, it will be skipped otherwise. The default value suggested is false.

  - ```
    public static String IRR_ALGORITHM
    ```
    It is used for the inter-regional algorithm selection. The available values for this variable are: *treeirr*, *fctirr* and *hirr*. The first two are experimental
    ```

algorithms that are not documented in this thesis. They are still present in JRegion as examples that show how to extend the prototype. For this reason, *hirr* is the suggested default value.

- ```
  public static long GLOBAL_FORWARDING_DELAY
  ```

  It is the delay added to each forwarding expressed in milliseconds. The suggested default value is 0 because the prototype provides the same output results even with bigger values. Higher values can be useful in experiments regarding the performances in terms of average delivery time.

This class defines also several methods to manage the declared regions and nodes:

- ```
  public void declareNode(String regionName, String
      nodeName)
  ```

  This method is used to declare a node given its name inside the specified region.

- ```
  public void declareRegion(String name)
  ```

  This method is used to declare a region given its name.

- ```
  public void removeNode(String nodeName)
  ```

  This method kills the corresponding thread and removes the specified node from the declared nodes.

- ```
  public void destroyRegion(String regionName)
  ```

  This method recursively removes a region and its nested regions from the environment. Each node inside the specified region and its nested regions is also removed from the declared nodes and the corresponding thread is killed.

- ```
  public void nestRegions(String external, String
      nested)
  ```

  This method is used to nest a specified region in another one. Both regions must be previously declared.

- ```
  public void addVent(String from, String to)
  ```

  This methods are used to add a mono-directional vent between nodes belonging to different regions.

- ```
  public void addBidirectionalVent(String n1, String
      n2)
  ```

This methods are used to add a bidirectional vent between nodes belonging to different regions.

–
```java
public void sendBundle(String from, String to,
    boolean waitForDelivery)
```

This method is used to send a bundle from a previously declared node to another one. The source node must be previously declared but the destination may not exist. If *blocking* parameter is true this command waits for the end of any interaction before exit, otherwise the method return immediately and the threads start to work independently.

–
```java
public void clearAllBlacklists()
```

This method deletes all the blacklist entries of every node in order to be able to start a new experiment from scratch without restarting the interpreter.

- **Interpreter class:** The interpreter is an executable class that directly performs instructions written in a command line prompt or in a script file, without requiring them previously to have been compiled. Each instance of this class refers to an instance of the Environment class, stored in the *env* attribute.
  When the interpreter is started it executes a default configuration file called "$default.conf$". If this file does not exist the console prints an error message and the execution continues using an hard-coded configuration. The core of this class is represented by the public method called $parseInputLine()$, which takes the string $inputLine$ as a parameter. This string is split into two tokens using the space character as a delimiter: the first token corresponds to the name of the command that is executed; the second token is the list of parameters. At this point, the interpreter dynamically invokes a method having the same name of the command using the Java Reflection, passing the second token as a parameter. The called method is in charge to parse the parameter list and to execute the corresponding command over the environment.
  This mechanism allows extending the interpreter by adding new methods. For example, if we want to define a new command called *test*, we just need to add the method $test(String\ params)$ to the interpreter class. Inside the method, we can interact with the environment to implement the behavior of the *test* command. After this, launching the interpreter we will be able to use the *test* command typing in the command line (or in a script file) $test\ parameter1\ parameter2\ ...\ parameterN$.

3.3 Usage

3.3.1 Interpreter commands

Command: region R_1 R_2 ... R_N

This command is used to declare multiple regions by their name.
The environment is updated with the new references to the specified regions.
Once declared the regions are not connected to each other and they do not contain any node (see the *nest* and *add* commands for this purpose).

Usage example

```
region R1 R2 R3 R4
```

This command defines 4 regions respectively called R1,R2,R3 and R4

Command: nest R_E R_{N_1} R_{N_2} ... R_{R_M}

This command is used to nest multiple regions R_{N_x} to an enclosing region R_E.
Before using the *nest* command, the specified regions must be previously declared using the *region* command.
The environment is updated adding the references to the specified region nesting.

Usage example

```
region R1 R11 R12 R13
nest R1 R11 R12 R13
```

After declaring $R1$, $R11$, $R12$ and $R13$ with the *region* command, the *nest* command set $R11$,$R12$ and $R13$ as sub-region of $R1$.

Command: add R N_1 N_2 **...** N_M

Add inside region R the specified nodes N_1, N_2, ... N_M.
The node declaration is implicit to this command.
For each first-declared node, a new *Node* instance is created and added to the environment and its thread is immediately executed.

Usage example

region $R1$ $R2$
add $R1$ $N1$
add $R2$ $N2$ $N3$

The first instruction declares the regions $R1$ and $R2$.
The node $N1$ is created and added to the region $R1$ by the second instruction.
The nodes $N2$ and $N3$ are added to the region $R2$ by the third instruction.

Command: vent N_{from} N_{to}

This command requires exactly two parameters, named N_{from} and N_{to}.
It adds a vent from N_{from} to N_{to}.
This vent is not bidirectional, therefore the vent from N_{to} to N_{from} is not declared implicitly.
N_{from} and N_{to} must belong to different regions.
If a node holds a bidirectional vent toward another node, it is automatically considered a passageway.

Usage example

```
region R1 R2
add R1 N1
add R2 N2
vent N1 N2
```

The first instruction defines the regions $R1$ and $R2$.
The second and the third instructions add respectively the node $N1$ to $R1$ and the node $N2$ to $R2$.
The last instruction declares a vent from $N1$ to $N2$.
Note that this does not imply the existence of a vent from $N2$ to $N1$.

Command: doublevent N_1 N_2

This command requires exactly two parameters, named N_1 and N_2.
It adds a bidirectional vent between N_1 and N_2.
N_1 and N_2 must belong to different regions.
If a node holds a bidirectional vent toward another node, it is automatically considered a passageway.

Usage example

region *R*1 *R*2
add *R*1 *N*1
add *R*2 *N*2
doublevent *N*1 *N*2

The first instruction defines the regions *R*1 and *R*2.
The second and the third instructions add respectively the node *N*1 to *R*1 and the node *N*2 to *R*2.
The last instruction declares a bidirectional vent from *N*1 to *N*2. In other words, this command declares also the vent from *N*2 to *N*1.

Command: runscript *FileName*

This command reads line by line the specified file, executing sequentially the instructions.
runscript accepts the same commands defined for the command line prompt. This command can be used for creating scripts that contain scenario definitions and/or experiments.

Usage example

runscript *test_scenario*

The file *test_scenario* is read line by line, and each line is executed by the interpreter.

Command: send *Source Destination*

This command sends a bundle from the *Source* node to the *Destination* node.
By contrast of *syncsend*, this command is not blocking, therefore the interpreter instantaneously gives back the prompt to the user or to the input script.
The output format is the same as *syncsend* command.

Usage example

region $R1$ $R2$
add $R1$ $N1$ $N2$
add $R2$ $N3$ $N4$ $N5$
doublevent $N1$ $N3$
send $N1$ $N2$
    node N1: received DATA from node: USER
    node N2: received DATA from node: N1
    node N2: bundle DELIVERED!!
send $N2$ $N4$
    node N2: received DATA from node: USER
    node N1: received DATA from node: N2
    node N3: received DATA from node: N1
    node N4: received DATA from node: N3
    node N4: bundle DELIVERED!!

The first command adds the regions $R1$ and $R2$. The second and the third add some nodes to these regions.
The fourth command adds a bidirectional vent between the nodes $N1$ and $N2$, which are now considered as passageways.
The fifth command sends a bundle from $N1$ to $N3$, and the last one sends a bundle from $N2$ to $N4$. As the nodes $N1$ and $N2$ belong to the same region, the first bundle sending is intra-regional.
By contrast, as the nodes $N2$ and $N4$ belong to different regions, the second bundle sending is inter-regional.
Note that, as the *send* command is not blocking, both commands runs concurrently.

Command: syncsend *Source Destination*

This command sends a bundle from the *Source* node to the *Destination* node.
By contrast of *send*, this command is blocking, therefore the interpreter waits for the end of the command before giving back the prompt to the user or to the input script.
The output format is the same as *send* command.

Usage example

region $R1$ $R2$
add $R1$ $N1$ $N2$
add $R2$ $N3$ $N4$ $N5$
doublevent $N1$ $N3$
syncsend $N1$ $N2$
  node N1: received DATA from node: USER bundle ID: 346861221
  node N2: received DATA from node: N1 bundle ID: 346861221
  node N2: bundle DELIVERED!!
syncsend $N2$ $N4$
  node N2: received DATA from node: USER bundle ID: 1188392295
  node N1: received DATA from node: N2 bundle ID: 1188392295
  node N3: received DATA from node: N1 bundle ID: 1188392295
  node N4: received DATA from node: N3 bundle ID: 1188392295
  node N4: bundle DELIVERED!!

The first command adds the regions $R1$ and $R2$. The second and the third add some nodes to these regions.
The fourth command adds a bidirectional vent between the nodes $N1$ and $N2$, which are now considered as passageways.
The fifth command sends a bundle from $N1$ to $N3$, and the last one sends a bundle from $N2$ to $N4$. As the nodes $N1$ and $N2$ belong to the same region, the first bundle sending is intra-regional.
By contrast, as the nodes $N2$ and $N4$ belong to different regions, the second bundle sending is inter-regional.
As the *syncsend* command is blocking, before going on the last command wait for the end of the execution of the previous one.

Command: kill N

This command terminates the thread related to the N node and remove it from the environment. The specified node must already exist.

Usage example

region $R1$
add $R1$ $N1$ $N2$ $N3$
kill $N1$
node N1: Thread exit...

After the first two instructions, the region $R1$ contains three nodes named respectively $N1$, $N2$ and $N3$. The last instruction terminates the thread related to the node $N1$ and remove its references from the environment.

Command: remove *Region*

Destroy recursively a region killing also all the node contained in each one.
The specified region must exist.

Usage example

region $R1$ $R2$ $R3$
nest $R1$ $R2$ $R3$
add $R1$ $N1$ $N2$ $N3$
add $R2$ $N4$ $N5$
add $R3$ $N6$ $N7$ $N8$
remove $R1$

```
node N4: Thread exit...
node N2: Thread exit...
node N7: Thread exit...
node N5: Thread exit...
node N6: Thread exit...
node N8: Thread exit...
node N1: Thread exit...
node N3: Thread exit...
```

The first instruction will define three regions, respectively named $R1$, $R2$ and $R3$.
The second one will nest $R2$ and $R3$ in $R1$.
The *add* instructions are adding some nodes within the regions defined.
As the *remove* works recursively, all the nodes related to the region R1 and its subregions are killed.
The environment is updated removing both nodes and regions involved in the recursion.

Command: var $VariableName = value$

Used to assign change value to the environment variables. The list of the available variables is described in the previous section.

Usage example

var $GLOBAL_FORWARDING_DELAY = 150$

Set the global delay to 150 milliseconds.
The environment variable is updated, therefore the changes are applied instantaneously.

Command: route

For each node, it prints a human readable output of its blacklist content. This command does not expect any parameters.

Usage example

var $BACKWARD_LEARNING_ENABLED = true$
region $R0$ $R1$ $R2$
add $R0$ 90
add $R1$ 91 10 11 12
add $R2$ 92 20 21 22
doublevent 90 91
doublevent 91 92
send 10 20
  node 10: received DATA from node: USER
  node 91: received DATA from node: 10
  node 90: received DATA from node: 91
  node 91: received SIGNAL from node: 90 BLACKLIST node: 20
  node 92: received DATA from node: 91
  node 20: received DATA from node: 92
  node 20: bundle DELIVERED!!
routes
  node 91:
  — 90 HAS BLACKLISTED 10
  — 92 HAS BLACKLISTED 10
  — 90 HAS BLACKLISTED 20

Command: generate

Generate ION configuration files representing the same scenario currently in use. A folder called *scenario* will contain the resulting output. The previous content of that folder will be wiped. During this generation, contacts are assumed to always continue. The goal is to obtain configuration files that let ION act exactly as the JRegion. More realistic scenario can be created using the generated configurations as template.

Usage example

```
runscript test_scenario
generate
```

The script *test_scenario* contains the definition of a generic scenario, in terms of regions, nodes and vents.
After loading the scenario, ION configuration files are generated.
The *scenario* will contain the result.

Chapter 4

HIRR implementation within ION

4.1 ION Overview

The Interplanetary Overlay Network (ION) is an implementation of the DTN architecture developed by NASA Jet Propulsion Laboratory (JPL), described in RFC4838[7]. It is specifically designed for embedded systems, in particular for robotic spacecraft. This software is the proposed DTN implementation for future space missions, with the aim of reducing the cost and risk in communications by simplifying the construction of automated communication networks.
ION is a DTN implementation that is able to work in an interplanetary network environment and has been written taking into account the following constraints:

- **Link constraints**: all interplanetary communications are wireless and they can be slow and asymmetric. The reason behind this is that the available electrical power for spacecraft is limited and antennae are small, so emitted signals are usually weak. This limits the transmission speed from a spacecraft to Earth to bit rates in the range from 256 kbps to 6 Mbps (higher values can however be reached with optical links, whose study is on going). Even if the electrical power provided to transmitters on Earth is much greater, the sensitivity of receivers on spacecraft is constrained by limited power and the size of the antenna. Because in the previous missions the volume of command traffic that had to be sent to spacecraft was less than the volume of telemetry the spacecraft was expected to return, old spacecraft receivers have been designed for lower data rates from Earth to the spacecraft, to bit rates on the order of 1kbps to 2 kbps.

- **Processor constraints**: the computing capability of the spacecraft CPUs is very low compared to terrestrial ones. As before, one of the causes is that the spacecraft have limited electrical power. Another problem is that the deep space environment is characterized by intense radiations that make impractical the use of regular

processors. For this reason, in order to minimize errors in computation and storage, these processors and the memory must be designed to be tolerant to radiations. This special treatment is not common for terrestrial application, therefore these processors are much more expensive than the traditional ones.

- **Operating system constraints**: in general, the software written for spacecraft must be highly reliable and have hard real-time processing deadlines, therefore it runs on hard real-time operating systems. As the processing performed by the spacecraft must be highly reliable, the result of computations must be predictable. In this context, the dynamic allocation of memory must be avoided as much as possible because it can introduce unpredictable effects that can compromise the functionalities of the spacecraft risking the failure of the space mission. In addition, many real-time operating systems do not support protected-memory models that are usually present in most of the traditional operating systems. Therefore, all tasks share the same system memory, which can be accessed directly without any protection.

4.2 ION Design Principles

4.2.1 Inter-task communication

ION is designed to take advantage of the shared memory model, using it for inter-task communication purposes. In this way, a data object can be shared between tasks and the access is regulated by semaphores. Semaphore operations are usually as fast as the storage and retrieval of data in memory. Therefore, this inter-task data interchange model is reasonably efficient for flight software. The communication between a sending task and a receiver task is always implemented as following:

- The sending task attempts to take a mutex semaphore. When it succeeds, it appends the data object to a queue implemented as a linked list in shared memory. After this, the mutex semaphore is released and a signal is sent (releasing another semaphore) to announce that the list contains a new element.
- The receiving task, which is waiting for the release of the signal semaphore, resumes its execution. When this happens, it takes the mutex of the queue and extracts the object from the list and give the mutex semaphore back.

4.2.2 Zero-copy objects (ZCO)

Another way to take advantage of the shared memory is the usage of zero-copy objects. Instead of creating multiple copies of the same object for each process, the elements

contained in the linked lists are replaced with pointers. In this way, each object is instantiated only once. This approach reduces the memory usage and eliminates the overhead of copying large objects byte-by-byte. If a task needs to access an object, the pointer is provided and a reference counter variable is increased. A zero-copy object can be destroyed only if its reference counter goes to zero. This works as a protection against any attempt to destroy the object while other tasks are using it.

4.2.3 Personal Space Management (PSM)

Even if in principle the memory is completely shared, in ION there is a library called Personal Space Management (PSM) used to allocate and recover objects within an assigned memory block of fixed size. PSM is a memory manager that provides an abstraction that separates the application from the differences between private and shared memory. It is designed to be more efficient than the common $malloc()$ and $free()$ system calls.

4.2.4 Simple Data Recorder (SDR)

Simple Data Recorder (SDR) is a memory manager similar to PSM that manages the non-volatile storage. In other words, it is implemented as an "object database" management system. It provides the functions to manage data structures of arbitrary complexity in persistent storage, relying on a single fixed-size file. One of the main goals of SDR is to provide an abstraction similar to a DBMS. To this end, it provides a transaction mechanism that guarantees the ACID properties (Atomicity, Consistency, Isolation, and Durability). The aim of SDR is to maintain consistent the state of a node after an unexpected crash or reboot.

4.3 HIRR implementation

Although the HIRR algorithm was tested by means JRegion prototype, there are other practical aspects to consider. For example, the ability of passageways to flood the topology without any protection may represent a critical security problem. During this research activity, a possible solution has been conceived: using a DTN node auto-configuration (DNAC) mechanism to manage the insertion of a node inside a region and exchanging public keys among nodes and passageways using DTKA (Delay-Tolerant Key Administration).
JRegion prototype, described in Chapter 3, proved essential in fast defining the basic characteristics of the HIRR algorithm. The next step was to integrate HIRR in ION. Given the complexity of this task, we decided to divide it into two phases. The former consists in building a preliminary version, a sort of C prototype, to study the interaction of HIRR with the other protocols present in ION, and in particular with CGR. The

latter in building the definitive official implementation, taking advantage of the lessons learned with the C prototype. The first phase has been carried out during this thesis and is described in this chapter. It is a necessary preliminary step towards the definitive implementation, which will follow in the near future, authored by my NASA supervisor, like the rest of the ION code.

4.3.1 Forwarding Strategy

Figure 4.1 shows the new forwarding strategy for an outgoing bundle. The scheme uses a first success approach: all the techniques are applied sequentially and the forwarding procedure ends encountering the first success (shown in green). If all the techniques fail, the bundle is saved into a memory space called limbo with the hope that it can be reforwarded in the future (e.g., after a contact plan update).
The first routing technique is called "sticky routing" and it fails (shown in red) if the bundle does not have a flow label. Each flow label is associated with a preference type (static or dynamic) and a preferred node number. The effect is to forward to the preferred node indicated by the flow label if possible, and to forward to the best available alternative neighbor otherwise. The second routing technique is called "static routing". A static route is a predetermined route computed by the node administrator (e.g. a ground station). This step fails if a static route to the destination of the bundle does not exist. If the destination is cited in the contact plan of the current node, CGR is used. Otherwise, the HIRR procedures are executed.

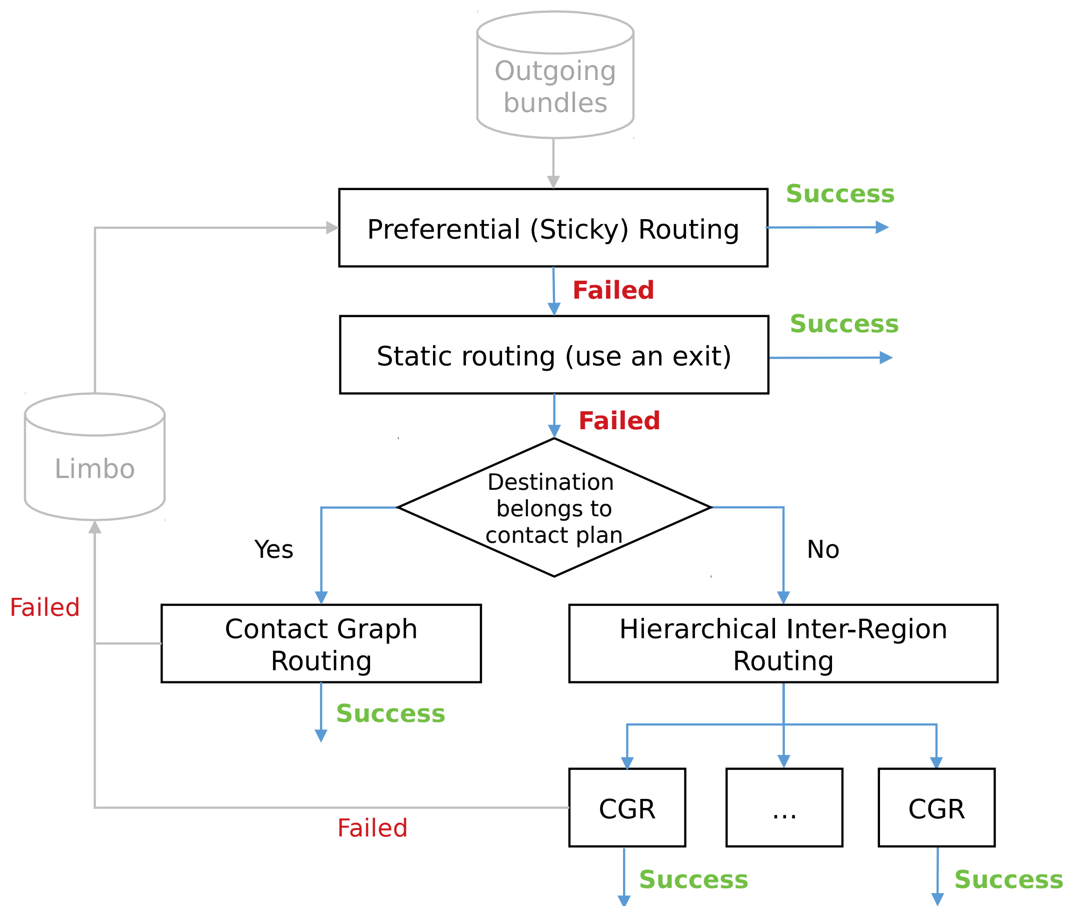

Figure 4.1: New forwarding strategy that includes the hierarchical inter-regional routing.

4.3.2 HIRR model

The *bpP.h* file defines the structures required by the bundle protocol implementation. It has been modified adding the following HIRR structures:

- **Provenance**: it is an enumeration (*enum*) that describes the provenance of a peer passageway. It defines three values: *Home*, *Outer* or *Unknown*.
  As explained in Chapter 2, the value is: *Home* if the peer passageway belongs to the home region of the current passageway; *Outer* if the peer passageway belongs to the outer region of the current passageway; *Unknown* if the peer passageway is not defined (this happens only when the current passageway is the first of the inter-regional route).

- **PeerPassageway**: it is a structure (*struct*) that describes the peer passageways.

Each peer is defined by a node number and a *Provenance*. Passageways are the only nodes that can contain other peer passageways.

- **BundleType**: it is an enumeration (*enum*) that describes the inter-regional bundle type. It defines two values: *Data* or *Blacklist*.

- **BlacklistEntry**: it is a structure (*struct*) that describes the blacklist entries. As explained in Chapter 2, a blacklist entry is an entry (P, N), where P is a peer passageway and N is a node, which could also be a passageway. The meaning is: P has blacklisted N.

The *bpP.h* file contains also the *BpDB* structure, which represents the bundle protocol configuration of a node. For example, this structure contains the contact plan, the induct list, the outduct list etc. SDR contains a singleton instance of this structure, that can be retrieved by the function $getBpConstants()$. This structure has been modified to contain also the following:

- **peerPassageways**: it is an SDR list of $PeerPassageway$ instances. It represents the list of all the known peers of the current passageway. Regular nodes should not declare any peer, otherwise they are treated by HIRR as passageways.

- **blacklist**: it is an SDR list of *BlacklistEntry* instances. It represents the list of all the learned blacklist rules of the current node.

- **defaultPwNbr**: it is an *uvast* variable that stores the default passageway node number.

Both $peerPassageways$ and *blacklist* are SDR list created in $bpInit()$, an initialization function of the bundle protocol defined in the file *libbpP.c*.

4.3.3 bprc configuration file

The program *bpadmin* is an interpreter used to configure the bundle protocol for the current ION node. It can be run specifying a configuration file (*bprc*) or using the command line interface. The most important commands allow to define endpoints, inducts and outducts. See the help (command "h") or the ION manual for more details.
The *bpadmin.c* file is being modified to accept also the definitions of peer passageways and the default passageway. The *bprc* configuration file now can contain the following commands:

- **P nodeNbr**: this command declares the default passageway of a terminal node. For example, to define the node 1 as default passageway for the terminal node 2, the *bprc* file of 2 must contain the command "P 1".

- **p nodeNbr provenance**: this command declares a peer passageway of the current passageway. The provenance can be "O" to define an *Outer* region or "H" to define an *Home* region. For example, to define the peer passageway 3 in the home region of the passageway 1, the *bprc* file of 1 must contain the command "p 3 H".

4.3.4 ipnfw.c functions

The $ipnfw$ program is a daemon launched by *bpadmin* in response to the start command "s" and it is terminated when the interpreter receives the stop command "x". Its goal is to extract bundles from an output bundle queue and forward the bundle to a proximate destination after computing the routing procedures. This file is the core of the routing strategy in ION. It has been modified to perform HIRR if the current region does not contain the bundle destination.

Blacklist functions

- ```
  int blacklistContains(uvast P, uvast N)
  ```

  this function returns 1 if the blacklist contains the entry (P, N), 0 otherwise.

- ```
  int blacklistContainsAll(uvast N,HIRRProvenance provenances)
  ```

  this function returns 1 if all peer passageways that belong to the specified *provenances* have blacklisted N, 0 otherwise.

- ```
  void blacklistAdd(uvast P, uvast N)
  ```

  this function adds the entry (P, N) to the blacklist.

- ```
  void blacklistClearDestination(uvast N)
  ```

  this function removes from the blacklist all the entries that have N as blacklisted destination.

- ```
  void blacklistPrint()
  ```

  this function print the blacklist in a human readable format.

Getter and setter functions

- ```
  uvast getDefaultPassagewayNbr()
  ```

  this function retrieve the *BpDB* structure contained in *SDR* and returns the node number of the default passageway.

- `void setIRRvalues(Bundle *bundle, uvast priorPwNbr, BundleType type)`

  this function set the prior passageway number (*priorPwNbr*) and the bundle *type* to the specified input bundle. It must be called before forwarding a replica to update the HIRR additional information of that bundle.

- `uvast getPriorPassageway(Bundle *bundle)`

  this function returns the prior passageway node number of a given bundle.

- `uvast getSource(Bundle *bundle)`

  this function returns the source node number of a given bundle.

- `uvast getDestination(Bundle *bundle)`

  this function returns the destination node number of a given bundle.

- `BundleType getBundleType(Bundle *bundle)`

  this function returns the type of a given bundle, either *Data* or *Blacklist*.

- `HIRRProvenance getProvenance(uvast nodeNbr)`

  this function returns the provenance of a given bundle, which can be *Home*, *Outer* or *Unknown*.

Utility functions

- `int regionContainsNode(uvast nodeNbr)`

  this function returns 1 if the current region contains the specified node, 0 otherwise.

- `int isDeadEnd(Bundle *bundle)`

  this function returns 1 if none of bundle replicas can not be sent because all the selected peer passageways have blacklisted the bundle destination, 0 otherwise. It is used to implement the auto rediscovery variant described in Chapter 2.

Bundle forwarding functions

- `void forwardBundleReplica(Bundle *bundle, uvast nodeNbr)`

  this function creates a replica of the given bundle and forwards it to the specified node number using CGR.

- `void forwardBlacklistBundle(Bundle *bundle, uvast nodeNbr)`

this function creates a blacklist bundle and forwards it to the specified node number using CGR.

- ```
  int dataBundleFlooding(Bundle *bundle)
  ```

  this function forwards the bundle replica to all peer passageways that both: a) have different region provenance respect to the current passageway; B) have not blacklisted the destination. It returns the total number of forwarded bundles.

- ```
  void signalBundleFlooding(Bundle *bundle, HIRRProvenance provenances)
  ```

  this function forwards a blacklist bundle to each passageway belonging to the specified provenances.

HIRR functions

- ```
  void interRegionalDataBundleHandling(Bundle *bundle)
  ```

  This function performs HIRR for a bundle whose destination does not belong to the current region. Further details are provided in the pseudocode section of Chapter 2.

- ```
  void interRegionalBlacklistBundleHandling(Bundle *bundle)
  ```

  this function is called when a blacklist bundle arrives. If required, it performs the blacklist bundle recursion. Further details are provided in the pseudocode section of Chapter 2.

- ```
  void backwardLearning(Bundle *bundle)
  ```

  this function implements the backward route learning described in Chapter 2. Further details are provided in the pseudocode section of Chapter 2.
  ```

Chapter 5

ION Inter-regional Test Suite

5.1 Motivations

In general, testing the HIRR implementation is a challenging task because it is not possible to conduct meaningful experiments in small topologies. This is not surprising because HIRR is designed to improve the scalability of CGR partitioning the network in regions, thus it does not make much sense to perform experiments with small numbers of nodes and little partitioning. In fact, if our reference topologies had too few nodes it would not be necessary to use an algorithm like HIRR because there would be no scalability issues. Testing the HIRR implementation in ION is much harder than JRegion because, even if we choose a complex topology, we need to cope with other challenges:

- creating the configuration files for all the nodes
- starting a large number of nodes
- dynamically adding or removing nodes from the regions

These challenges have revealed the need to create a test suite. It uses one of the ION features that allows starting several nodes in the same machine without requiring any virtualization techniques. This approach has significant advantages over the use of virtual machines or containers in terms of the amount of the memory and CPU needed to cope with a high number of instantiable nodes. In fact, in an average desktop machine, the maximum number of nodes that can be run is more or less one hundred. Note that this number is large compared to the number of nodes that can be run using virtual machines or containers but is small compared to the number of nodes that can be run in JRegion. The ION test suite for HIRR testing is composed of several bash scripts, which provides a subset of commands equivalent to the ones provided by JRegion. The main goal is to manage the ION nodes in a centralized fashion, similar to the JRegion interpreter. By contrast with JRegion, the regions are statically defined by the contact plans of the

nodes. All the configuration files used in ION were automatically generated with the "*generate*" command of JRegion.

5.2 Scripts

environment.sh

This file contains the functions to create and maintain the environment. It is conceptually the equivalent of *Environment* class of *JRegion* but it contains only a subset of its functions for the sake of simplicity.

- ```
  build()
  ```

  This function has no parameters. It creates the *environment.db* file according to the configuration files of each node in the *nodes* directory. This file contains a line for each node in the topology. Each line contains the region name, the node name and the default passageway name, each separated by a space.

- ```
  getDefaultPassageway()
  ```

  This function takes a region name as parameter and prints the name of the default passageway node of the considered region.

- ```
  addNode()
  ```

  This function takes a region name and a node name as parameters. After deducting the default passageway from the region name, it inserts a line in the *environment.db* containing the given region, node name and the deducted default passageway.

- ```
  deleteNode()
  ```

  This function takes a node name as parameter and deletes from the *environment.db* file the corresponding line (if exists).

- ```
  getRegionNodeList()
  ```

  This function takes a region name as parameter. It scans the *environment.db* file and prints all the node names having the given region name as first word of the lines.

- ```
  whereIs()
  ```

  This function takes as parameter a node name and prints the name of its corresponding region.

- `print()`

  This function has no parameters and prints the content of *environment.db* file.

- `getNodes()`

  This function has no parameters and prints all the node names contained in the *environment.db* file.

start.sh

This script is used to start multiple ION nodes. If it is run without parameters starts all nodes contained the *nodes* folder. Alternatively, a list of node names separated by spaces can be passed as parameters. In this case, only the specified nodes are started.
Each node is started using the only function contained in this script: *startNode*(). This function takes a node name as a parameter and starts the corresponding node in the *nodes* folder. It terminates with a success state only after the specified node is correctly started, or with an error if an error occurred.
Starting a single ION node may require up to 10 seconds, therefore starting sequentially a bunch of nodes may be a problem if the number of nodes to be started is large. For this reason, the *start.sh* script runs in parallel the starting function of each node and it ends when all the nodes are being started correctly. If an error occurred in at least one of the *startNode*() executions, the script brutally terminates printing an error message.

send.sh

This script is used to send a test-file from a source to a destination. It is similar to the JRegion *send* command. This script takes two parameters as input: the first is the node name of the source node, the second is the node name of the destination node. After the test-file is sent, this script is blocking until the destination node receives the file.

add.sh

This script is used to add a node into an existing region. It is similar to the JRegion *add* command. It requires exactly two input parameters: the node name and the region name. If the specified node does not exist, it will be created. If the specified region does not exist, the script terminates giving an error message. In order to create a new node, its ION configuration files are automatically generated. After creating (or reintroducing) a node into a region, this script updates the configuration of all nodes belonging to the specified region adding the missing contacts between them and the specified node. If the whole process succeeds, the environment is updated with the information regarding the specified node.

kill.sh

This script is used to remove the specified node from its region. It is analogous to the JRegion *kill* command, but it presents some differences: JRegion kills the thread that represents the specified DTN node and it removes all its references from the environment; on contrary this script removes every reference (e.g., contacts, outducts, plan, etc.) to the specified node but does not kill its processes. In other words, the specified node is removed isolating it from the rest of its region. This decision was made to simplify the reintroduction of the same node to another region using the *add.sh* script.

5.3 Experiments

5.3.1 Scenario

The following tests are conducted with the same topology introduced in Figure 5.1. This topology is equal to the topology shown in the previous examples, with the difference that the passageway names are numerical. Each region has 3 terminal nodes and one border passageway.
To interpret rapidly the results we used the following naming conventions:

- the region name of a nested region is equal to the enclosing region name followed by a progressive number. For example, the name of regions nested inside R_1 are R_{11}, R_{12}, R_{13}, etc.
- the border passageway node number is 9 followed by the region number. For example, the node number of the passageway on the border of region R_{13} is 913.
- the name of a node is equal to the region number followed by a progressive number. For example, the number of nodes in region R_{21} are $211, 212, 213$, etc.

These name conventions give us the advantage of knowing the information of the provenance of each node by looking at its node name.

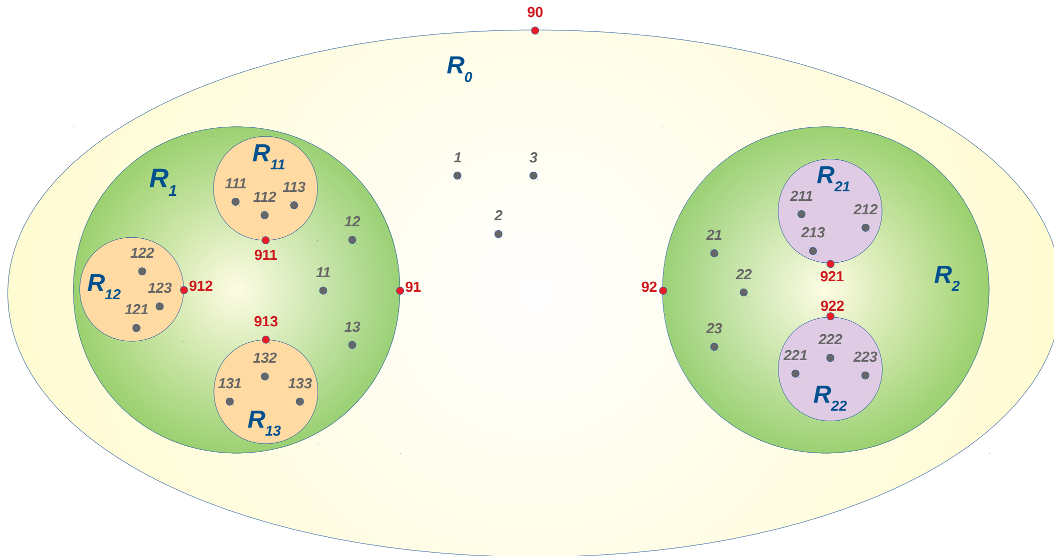

Figure 5.1: Topology used for testing. Passageways are represented as red dots and terminal nodes are represented as gray dots. Each region has 3 terminal nodes and one border passageway.

5.3.2 JRegion and ION configuration files

The topology described in Figure 5.1 is defined by the following JRegion configuration file. After loading this configuration in JRegion, we have generated the equivalent ION configuration files using the *generate* command.

```
# environment variables
var HOT_POTATO_ROUTING_ENABLED = false
var BUNDLE_REGISTRY_ENABLED = true
var GLOBAL_FORWARDING_DELAY = 0
var BACKWARD_LEARNING_ENABLED = true
var AUTO_REDISCOVERY_ENABLED = false
var IRR_ALGORITHM = hirr

#regions
region R0 R1 R2 R11 R12 R13 R21 R22

#nest enclosingRegion nestedR1 nestedR2 nestedR3 ...
nest R0 R1 R2
nest R1 R11 R12 R13
nest R2 R21 R22

#add Region Node1 Node2 Node3 ...
add R0 2 3 4
add R1 11 12 13
```

```
add R2 21 22 23
add R11 111 112 113
add R12 121 122 123
add R13 131 132 133
add R21 211 212 213
add R22 221 222 223

#passageways
add R0 90
add R1 91
add R2 92
add R11 911
add R12 912
add R13 913
add R21 921
add R22 922


#vent Node1 Node2  -> #add a monodirectioal vent from Node1 to Node2
#doublevent Node1 Node2 -> #add a bidirectional vent between Node1 ad Node2

#R0
doublevent 90 91
doublevent 90 92
doublevent 91 92

#R1
doublevent 91 911
doublevent 91 912
doublevent 91 913
doublevent 911 912
doublevent 911 913
doublevent 912 913

#R2
doublevent 92 921
doublevent 92 922
doublevent 921 922
```

5.3.3 Experiment 1

The goal of this experiment is to show the behavior of both JRegion and ION sending a bundle from a source node in region R_{11} and a destination node in region R_{22}, as shown in Chapter 2 at Figures 2.4, 2.5, 2.6 and 2.7. This experiment terminates with a success value if the behavior of passageways matches with the cited figures, a failure otherwise. The following outputs confirm the success of this experiment: a bundle replica is propagated to the entire topology and the passageways not involved in the inter-

regional route send back a blacklist message, as expected from the cited figures. After this experiment, the passageways have learned the unique inter-regional route between the two nodes. Note that the order of the output lines is not predictable, but the content is the same in both cases.

JRegion

```
send 111 222
node 111: received DATA from node: USER
node 911: received DATA from node: 111
node 91: received DATA from node: 911
node 913: received DATA from node: 911
node 912: received DATA from node: 911
node 90: received DATA from node: 91
node 92: received DATA from node: 91
node 911: received SIGNAL from node: 913 BLACKLIST node: 222
node 91: received SIGNAL from node: 90 BLACKLIST node: 222
node 921: received DATA from node: 92
node 922: received DATA from node: 92
node 911: received SIGNAL from node: 912 BLACKLIST node: 222
node 222: received DATA from node: 922
node 222: bundle DELIVERED!!
node 92: received SIGNAL from node: 921 BLACKLIST node: 222
---
Experiment 1 passed.
```

ION

```
./send 111 222
node 111: received DATA from node: USER
node 911: received DATA from node: 111
node 91: received DATA from node: 911
node 913: received DATA from node: 911
node 912: received DATA from node: 911
node 90: received DATA from node: 91
node 92: received DATA from node: 91
node 911: received SIGNAL from node: 913 BLACKLIST node: 222
node 911: received SIGNAL from node: 912 BLACKLIST node: 222
node 922: received DATA from node: 92
node 921: received DATA from node: 92
node 222: received DATA from node: 922
node 222: BUNDLE DELIVERED!
node 92: received SIGNAL from node: 921 BLACKLIST node: 222
node 91: received SIGNAL from node: 90 BLACKLIST node: 222
---
Experiment 1 passed.
```

5.3.4 Experiment 2

The goal of this experiment is to show that if we repeat twice the same operation of the first experiment, the second bundle follows the unique inter-regional route between the source and the destination, as already shown in the Chapter 2 at Figure 2.3. The experiment terminates with a success value if the second bundle follows the unique inter-regional route without generating blacklist bundles, a failure otherwise. The following outputs shows the experiment success in both JRegion and ION.

JRegion

```
send 111 222
node 111: received DATA from node: USER
node 911: received DATA from node: 111
node 91: received DATA from node: 911
node 913: received DATA from node: 911
node 912: received DATA from node: 911
node 90: received DATA from node: 91
node 92: received DATA from node: 91
node 911: received SIGNAL from node: 913 BLACKLIST node: 222
node 91: received SIGNAL from node: 90 BLACKLIST node: 222
node 921: received DATA from node: 92
node 922: received DATA from node: 92
node 911: received SIGNAL from node: 912 BLACKLIST node: 222
node 222: received DATA from node: 922
node 222: bundle DELIVERED!!
node 92: received SIGNAL from node: 921 BLACKLIST node: 222

send 111 222
node 111: received DATA from node: USER
node 911: received DATA from node: 111
node 91: received DATA from node: 911
node 92: received DATA from node: 91
node 922: received DATA from node: 92
node 222: received DATA from node: 922
node 222: bundle DELIVERED!!

___
Experiment 2 passed.
```

ION

```
./send 111 222
node 111: received DATA from node: USER
node 911: received DATA from node: 111
node 91: received DATA from node: 911
node 913: received DATA from node: 911
node 912: received DATA from node: 911
```

```
node 90: received DATA from node: 91
node 92: received DATA from node: 91
node 911: received SIGNAL from node: 913 BLACKLIST node: 222
node 911: received SIGNAL from node: 912 BLACKLIST node: 222
node 922: received DATA from node: 92
node 921: received DATA from node: 92
node 222: received DATA from node: 922
node 222: BUNDLE DELIVERED!
node 92: received SIGNAL from node: 921 BLACKLIST node: 222
node 91: received SIGNAL from node: 90 BLACKLIST node: 222

./send 111 222
node 111: received DATA from node: USER
node 911: received DATA from node: 111
node 91: received DATA from node: 911
node 92: received DATA from node: 91
node 922: received DATA from node: 92
node 222: received DATA from node: 922
node 222: BUNDLE DELIVERED!

---
Experiment 2 passed.
```

5.3.5 Experiment 3

The goal of this experiment is to check the global behavior of HIRR. It has two steps: the first one consists in sending a bundle between all the possible pairs of source and destination in the topology to learn all the existing inter-regional routes; the second step consists in repeating the same procedure and check if all the bundles follow the unique inter-regional route between each pair of source and destination. The experiment fails if during the second step there is at least one bundle that follows multiple inter-regional routes or there is a passageway that sends blacklist bundles. The test terminates with a success message if no anomaly has been detected. As this experiment requires to send a large number of bundles, the output is too much verbose and it is not shown in this document. The outcome was positive and the ION output matches with the JRegion one (except for the order because it is not predictable).

5.3.6 Experiment 4

The goal of this experiment is to check the validity of every possible route. It sends a bundle between two nodes twice: the first sending is used to learn the unique inter-regional route between the nodes, the second is used to check immediately if it was learned correctly. The experiment fails if during the second step the bundle follows multiple inter-regional routes or there is a passageway that sends blacklist bundles. These two steps are repeated for each possible pair of source and destination in the topology.

The test terminates with a success message if no anomaly has been detected. Like the previous experiment, it sends a large number of bundles and the output is too much verbose, therefore is not reported in this document. The outcome was positive and the ION output matches with the JRegion one (except for the order because it is not predictable).

Conclusions

This work was carried out at NASA Jet Propulsion Laboratory located in Pasadena, California, under the Visiting Student Research Program (VSRP). It concerns Inter-Planetary Networking (IPN), which is a particular application case of Delay/Disruption Tolerant Networking (DTN) architecture, based on the introduction of the Bundle Protocol. In particular, it is focused on the problem of routing and describes a possible way to solve the Contact Graph Routing (CGR) scalability problem using an Inter-Regional Routing (IRR) approach.

The proposed solution, named Hierarchical Inter-Regional Routing (HIRR), exploits the idea of region nesting in a hierarchical topology. Here, each DTN node is assigned to an administrative region where bundles can move from a region to the next by passing through special nodes that belong to two regions, called passageways. These nodes learn the next inter-regional leg to the path to the destination, by using a flooding-based mechanism. Once the path is learned, the cost of successive inter-regional routing decisions increases linearly with the number of nodes present in the topology.

In order to support a fast HIRR development, the algorithm was initially implemented as a Java prototype named JRegion. Later, when the algorithm became mature enough, we decided to build a proof-of-concept implementation of HIRR within the latest ION release (3.6.2, Nov 2018). Finally, we carried out a series of tests to prove the correctness of both implementations. The preliminary results show that the CGR scalability problem is virtually solved at the price of some bandwidth wasting in the preliminary phase when inter-regional routes are learned, and of an optimality reduction of the routes found with respect of global use of CGR.

The solution proposed and studied in this thesis has been positively accepted by my NASA mentor, Scott Burleigh, and will be further investigated by his research team with the development of a new implementation with a greater degree of integration with other ION components. My hope is that the work carried out in this thesis can be useful for future space missions.

Acknowledgments

First of all, I would like to thank my thesis supervisor Carlo Caini, a professor from the University of Bologna. During the last years, he involved me in his DTN research activities and gave me precious suggestions for my student career. Over time he became more than a professor: for me he is a guide, a mentor and a friend. Last but not least, he gave me the opportunity to carry out my thesis work in California at NASA Jet Propulsion Laboratory, which was the most important experience of my life. Words are never enough to express my gratitude.
I would like to express my appreciation to my JPL mentor Scott Burleigh and co-mentor Leigh Torgerson for giving me the opportunity to work with them at JPL and helping me with this thesis project.
I also would like to thank my office mate Marc Sanchez Net for our discussions that helped me to improve the quality of this work.
Another thanks goes to Michele Rodolfi for giving me precious advice on how to manage my stay in Pasadena.
Finally, I would like to thank all the friends I met here, who made my abroad experience in California unforgettable.

Bibliography

[1] G. Araniti, N. Bezirgiannidis, E. Birrane, I. Bisio, S. Burleigh, C. Caini, M. Feldmann, M. Marchese, J. Segui, and K. Suzuki. Contact graph routing in dtn space networks: overview, enhancements and performance. *IEEE Communications Magazine*, 53(3):38–46, March 2015.

[2] N. Bezirgiannidis, C. Caini, D. D. P. Montenero, M. Ruggieri, and V. Tsaoussidis. Contact graph routing enhancements for delay tolerant space communications. pages 17–23, Sept 2014.

[3] N. Bezirgiannidis, C. Caini, and V. Tsaoussidis. Analysis of contact graph routing enhancements for dtn space communications. *International Journal of Satellite Communications and Networking*, 34(5):695–709.

[4] S. Burleigh, V. Cerf, R. Durst, K. Fall, A. Hooke, K. Scott, and H. Weiss. The interplanetary internet: A communications infrastructure for mars exploration. *Acta Astronautica*, 53(4):365 – 373, 2003. The New Face of Space Selected Proceedings of the 53rd International Astronautical Federation Congress.

[5] C. Caini, P. Cornice, R. Firrincieli, M. Livini, and D. Lacamera. Analysis of tcp and dtn retransmission algorithms in presence of channel disruptions. In *2009 First International Conference on Advances in Satellite and Space Communications*, pages 174–179, July 2009.

[6] C. Caini, H. Cruickshank, S. Farrell, and M. Marchese. Delay- and disruption-tolerant networking (dtn): An alternative solution for future satellite networking applications. *Proceedings of the IEEE*, 99(11):1980–1997, Nov 2011.

[7] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, and H. Weiss. Delay-tolerant networking architecture. RFC 4838, RFC Editor, April 2007.

[8] K. Fall. A delay-tolerant network architecture for challenged internets. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and*

Protocols for Computer Communications, SIGCOMM '03, pages 27–34, New York, NY, USA, 2003. ACM.

[9] Y. Rekhter, T. Li, and E. S. Hares. A border gateway protocol 4 (bgp-4). RFC 4271, RFC Editor, January 2006.

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