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

Unibo-BP: an innovative free software implementation of Bundle Protocol Version 7 (RFC 9171)

Candidate:
Lorenzo Persampieri

Supervisor:
Prof. **Carlo Caini**

Academic Year 2021/2022

ABSTRACT

The BP (*Bundle Protocol*) version 7 has been recently standardized by IETF in RFC 9171, but it is the whole DTN (*Delay-/Disruption-Tolerant Networking*) architecture, of which BP is the core, that is gaining a renewed interest, thanks to its planned adoption in future space missions. This is obviously positive, but at the same time it seems to make space agencies more interested in deployment than in research, with new BP implementations that may challenge the central role played until now by the historical BP reference implementations, such as ION and DTNME. To make Unibo research on DTN independent of space agency decisions, the development of an internal BP implementation was in order. This is the goal of this thesis, which deals with the design and implementation of Unibo-BP: a novel, research-driven BP implementation, to be released as Free Software. Unibo-BP is fully compliant with RFC 9171, as demonstrated by a series of interoperability tests with ION and DTNME, and presents a few innovations, such as the ability to manage remote DTN nodes by means of the BP itself. Unibo-BP is compatible with pre-existing Unibo implementations of CGR (*Contact Graph Routing*) and LTP (*Licklider Transmission Protocol*) thanks to interfaces designed during the thesis. The thesis project also includes an implementation of TCPCLv3 (*TCP Convergence Layer version 3,* RFC 7242), which can be used as an alternative to LTPCL to connect with proximate nodes, especially in terrestrial networks. Summarizing, Unibo-BP is at the heart of a larger project, Unibo-DTN, which aims to implement the main components of a complete DTN stack (BP, TCPCL, LTP, CGR). Moreover, Unibo-BP is compatible with all DTNsuite applications, thanks to an extension of the Unified API library on which DTNsuite applications are based. The hope is that Unibo-BP and all the ancillary programs developed during this thesis will contribute to the growth of DTN popularity in academia and among space agencies.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 CHALLENGED NETWORKS

Space communications are subject to long propagation delays, intermittent connectivity, frequent transmission errors, and downtime.

Terrestrial Internet is based on the TCP/IP architecture, designed to ensure good performance in an environment where the following requirements are met:

1. The round-trip time (RTT), i.e., the time elapsed between sending a packet and acknowledging its receipt, must be short.
2. Throughout the communication session, there must be a continuous path between sender and receiver.
3. Packet losses due to errors introduced by the channel must be negligible.
4. Support for the TCP/IP protocol suite must be provided by all nodes.

Networks that do not meet one or more of these requirements are called "Challenged", as in them the TCP/IP suite has difficulties, or simply cannot, provide satisfactory performance. Among challenged networks we have both space networks and a few peculiar types of terrestrial and maritime networks (sensor networks, emergency networks, military tactical networks, underwater networks, etc.). A new architecture known as DTN (Delay-/Disruption-Tolerant Networking) was thus designed to overcome all mentioned challenges [RFC4838].

1.2 THE DTN ARCHITECTURE

The DTN architecture originates from a generalization of InterPlanetary Networking (IPN) and is designed to be used in the terrestrial or submarine domain as well. This new architecture proposes to solve the problems of challenged networks by inserting a new layer called the "Bundle Layer" between the application and transport layers.

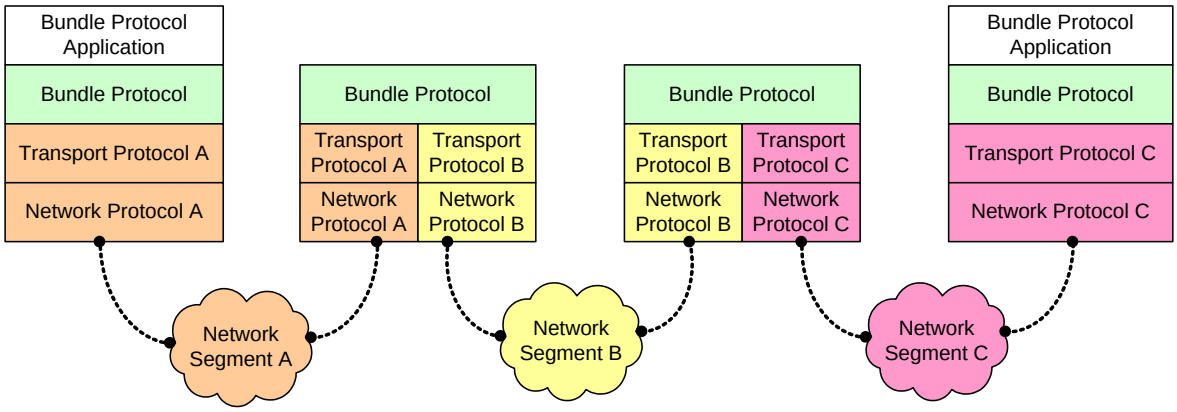


Figure 1.1 - The DTN Architecture

Nodes that implement the Bundle Layer are called DTN nodes, but there is no requirement that this additional layer be implemented by all nodes in the network. In fact, the nodes in the network segment represented by the cloud in the Figure 1.1 can be simple routers in the Internet network if the network segment connecting two DTN nodes is the Internet network. The unit of information exchanged at the Bundle Layer is the Bundle and the protocol that is used at the bundle layer is called the Bundle Protocol (BP); its version 6 was standardized by the Internet Research Task Force (IRTF) in an Experimental RFC [RFC5050] (issued in 2007), and only recently (January 2022) its version 7 has been formalized by the Internet Engineering Task Force (IETF) as a Proposed Standard RFC [RFC9171]. In parallel, the BP has been standardized also by CCSDS (Consultative Committee for Space Data System), a standardization body consisting of all major space Agencies, with “Blue book” (standard document in CCSDS jargon) [CCSDS_BPV6].

A DTN node handles Bundles according to a “store, carry and forward” policy to cope with the possible lack of a continuous path between source and destination nodes. Each node along the path stores incoming Bundles, it may carry them with it (as in “data-mule” or “ferry” environments), and forwards them to the next node on the source-to-destination path, as soon as the corresponding link is available, which may be immediately, or after hours. In fact, in challenged networks it is expected that a node may not have the ability to send Bundles for an arbitrarily long period of time, due to temporarily unavailable links with neighboring nodes.

1.3 ROUTING

A DTN node makes use of routing algorithms at the Bundle Layer level to decide which proximate node it should forward the Bundle to get the Bundle delivered to destination.

DTN routing algorithms can be opportunistic or deterministic, depending on the kind of intermittent connectivity typical of the deployment environment. In the former case, contacts between nodes are supposed random, thus they are based on sending multiple copies of a Bundle to different neighbors, with the goal of increasing the probability of delivery to destination. The main algorithms of this type are Epidemic [Vahdat_2000] Spray and Wait [Spyropoulos_2005], DTLSR [Demmer_2007] and ProPHET [Lindgren_2004] [RFC6693]; all of them implement a sort of controlled flooding, where the bundle is not passed to all nodes encountered, but only to a subset (to avoid a possible network collapse of a pure flooding). They mainly differ in the rule followed to restrict the number of copies [Caini_2011], [Araniti_2015].

However, there are also deterministic routing algorithms, particularly relevant to space communications, which exploit the a priori knowledge of deterministic contacts. They use the information contained in the "contact plan", i.e., the list of next "contacts" and "ranges" compiled in advance by a space agency and distributed to its nodes. In this framework, contacts are unidirectional, being defined as "transmission opportunities from node A to node B"; they obviously refer to a time interval and to a nominal transmission rate (often asymmetric in space links). The expected propagation delay between node A and B, whose knowledge is also essential to routing, is given in "range" instructions. Note that the propagation delay can vary from a few ms for close nodes (e.g., between a Lander and a Rover) to a few minutes (e.g., between Mars and Earth) or more. The main routing algorithm of this type is the Contact Graph Routing, designed by Scott Burleigh of NASA-JPL, whose latest version has been recently standardized by CCSDS as Schedule-Aware Bundle Routing (SABR) [CCSDS_SABR]. The University of Bologna (Unibo), together with DLR (German Aerospace Center), has presented an enhanced version of SABR [Caini_2021] [Birrane_2021] [Persampieri_2020], implemented in Unibo-CGR, the thesis project

developed for my bachelor's degree [Unibo-CGR]. Unibo-CGR was designed to operate with ION, the NASA-JPL suite of DTN protocols, and it has been included in ION official package [ION] by its designer and maintainer, Scott Burleigh, (it can be activated in alternative to the original implementation by means of a "configure" option). However, thanks to its modular design, Unibo-CGR can be coupled with whatever BP implementation by means of a simple interface, as we did during this thesis, to allow Unibo-BP to use SABR (with numerous optional extensions) by means of Unibo-CGR.

1.4 CONVERGENCE LAYER

In DTN jargon, the protocol stack below the bundle layer is improperly called Convergence Layer. At its top, i.e., immediately below the BP and usually above a Transport protocol, we have the Convergence Layer Adapters (CLAs); their aim is to allow BP to abstract from the specific interfaces of the underlying protocols. The two most important convergence layer adapters are the TCP Convergence Layer and the LTP Convergence Layer (note that their official names do not include the "adapter" word, with some language inconsistency).

The TCP Convergence Layer is required to be implemented by a DTN node to send and receive Bundles over Internet. Although its version 4 has been recently standardized [RFC9174] in the thesis we were more interested in version 3 [RFC7242] since it is the version currently implemented by all major BP implementations, thus the only one that can provide a high interoperability to Unibo-BP.

The LTP Convergence Layer [RFC5325] [RFC5326] [CCSDS_LTP] is of particular interest for space communications as it allows the use of the Licklider Transmission Protocol (LTP) as a Transport Layer protocol, below BP. LTP was specifically designed to be used in space, to overcome TCP limitations in the presence of long delays or link intermittency. Unibo-LTP [Bisacchi_2022] [Bisacchi_2021] [Unibo-LTP] is an implementation of Multicolor-LTP, and enhanced version of LTP proposed by Unibo and DLR to be used with ION. Its modular design allowed me to develop an alternative interface to Unibo-BP, thus adding the possibility of using LTP in alternative to TCP as a convergence layer, which is of great importance for space environments.

1.5 DTNSUITE

The DTNsuite [DTNsuite] is a collection of applications, developed and maintained by Unibo, which makes use of the Unified API [UnifiedAPI], a library that allows application developers to use a common "abstracted" API to interact with the underlying BP to perform tasks such as sending or receiving bundles, independently of APIs of specific BP implementations, with obvious advantages in terms of application behavior consistency and maintenance facility.

To do this, the Unified API requires that a special interface be implemented for each supported BP. During this thesis, the support for Unibo-BP was integrated into Unified API, thus making available all DTNsuite applications (DTNperf [Caini_2013], DTNchat, DTNproxy, DTNfog, DTNbox [Bertolazzi_2019]) to Unibo-BP users.

1.6 UNIBO-DTN

Unibo-DTN [Unibo-DTN] is the "umbrella" project aimed at including all major components of a DTN node, some of which, as SABR, LTP and DTNsuite, were previously developed as independent projects. This thesis fills the main gap with the development of BP version 7. Thanks to it, and to interfaces to Unibo-CGR, Unibo-LTP, DTNsuite, all pre-existing modules are now integrated in Unibo-DTN and can be used together. Moreover, the implementation of TCP Convergence Layer Version 3 adds the possibility to interoperate with virtually all other BP implementations.

Now Unibo-DTN contains all the major components needed to deploy a DTN node, with a focus toward DTN nodes operating in space networks. As that, it looks like a Unibo (scaled down) version of ION. However similar, we must highlight that ION and Unibo-DTN have different aims. ION is designed with operation environments in mind, with specific features (e.g., SDR [Burleigh_2007]) to increase robustness in space and a wide variety of additional protocols for network management, for the support of a BP aware public-subscribe architecture and security (BPSec [RFC9172], DTKA [Burleigh_2013]), just to mention only a few elements of a long list. Unibo-DTN, by contrast, is more focused on research and development of DTN pillars, where modularity and varieties

of experimental options are the major design drivers, as obvious coming from an academic environment.

2 BUNDLE PROTOCOL VERSION 7

2.1 BUNDLE NODE

Each bundle node offers the ability to send or receive bundles and is composed of three main components: a Bundle Protocol Agent (BPA), an Application Agent, and a set of zero or more Convergence Layer Adapters.

The Bundle Protocol Agent is the key component of a bundle node as it implements the services offered by the Bundle Protocol. These services are explained in detail in [RFC9171].

The Application Agent uses the services offered by BP to communicate. It is itself composed of two elements: the Administrative Element and the Application-Specific Element. The former deals with receiving or sending Administrative Records, among which are status reports (i.e., information regarding the processing of a specific bundle). The latter deals with sending or receiving Application Data.

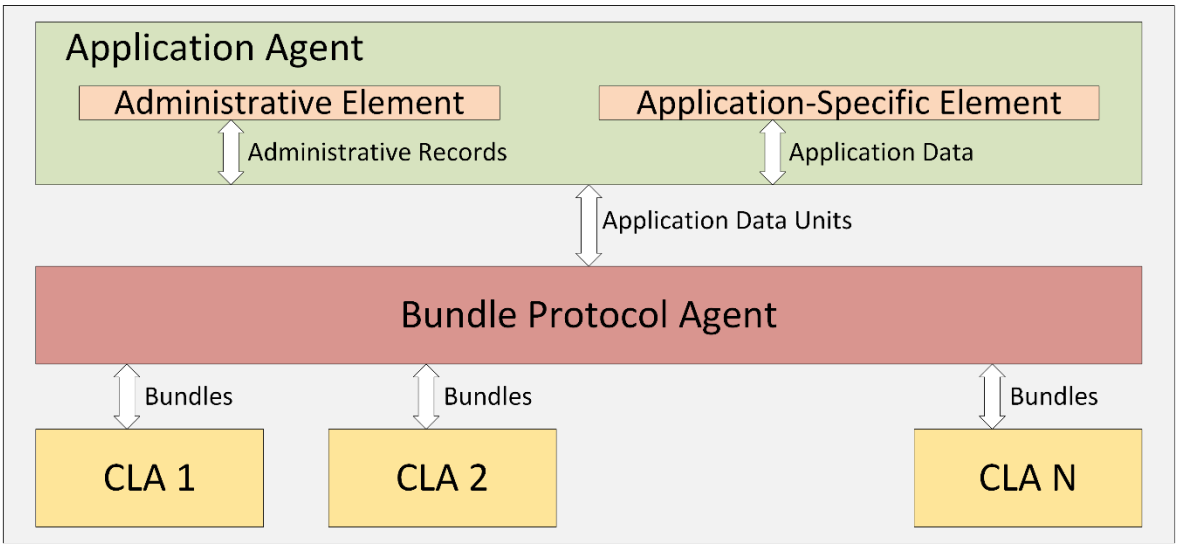


Figure 2.1 - Bundle Node

2.2 ENDPOINT ID

Each bundle node, or service it offers, is represented by an Endpoint ID (EID). Such EIDs are characterized by the scheme they follow (see below) and can be singleton or

not. Specifically, if an EID is not singleton, it means that at any given time zero or more nodes may have registered as belonging to that EID. Otherwise, the EID is singleton, i.e., exactly one node is registered with it. One singleton EID to which each bundle node is implicitly and permanently registered is the EID of the administrative element of that bundle node, called the Node ID. Two main schemes are defined in [RFC9171]: "*dtn*" and "*ipn*".

2.2.1 Scheme “dtn”

Each dtn EID consists of two main data items (alphanumeric strings): the *node name* and the *demux token*. In case the demux token is empty, it means that the EID in question could represent a Node ID. Otherwise, if the demux token starts with the “~” character, the EID in question is not a singleton EID. In all other cases it represents a singleton EID.

```
dtn://<node name>/<demux token>
```

The dtn scheme also supports the concept of null EID represented by the syntax "*dtn:none*".

2.2.2 Scheme “ipn”

Each ipn EID consists of two main data items (non-negative integers): the *node number* and the *service number*. Each ipn EID is implicitly singleton. An EID in the ipn scheme can be used as a Node ID only if the service number is 0.

```
ipn:<node number>.<service number>
```

2.3 BUNDLE FORMAT

A bundle consists of one Primary Block, one Payload Block and zero or more Extension Blocks. In the following, we refer to version 7 unless explicitly stated. The bundle is serialized, for transmission purposes, using a Concise Binary Object Representation (CBOR) encoding [RFC8949]; see [RFC9171] for details.

The Primary Block follows a different structure than the other blocks and represents the bundle header. Below we recall the information contained in the Primary Block.

- The version of the Bundle Protocol that built this bundle.
- The Bundle Processing Control Flags, which are flags representing properties that apply to the entire bundle rather than to a specific block.
- The CRC type: a value between "no Cyclic Redundancy Check is present," "a standard X-25 CRC-16 is present" [CRC16] and "a standard CRC32C (Castagnoli) CRC-32 is present" [RFC4960].
- The EID of the application element of the bundle destination, which can be either singleton or not but never anonymous (e.g., "*dtn:none*").
- The EID of the bundle source, termed the Source Node ID, i.e., the EID of the application element that generated the bundle (e.g., "*ipn:1.1*"), which can be either singleton or the null EID.
- The EID of the report-to of the bundle, i.e., the EID of the administrative element to which the status reports are destined, which may differ from the administrative element of the bundle source or be anonymous in case no status reports are to be generated for that bundle.
- The creation timestamp that includes the DTN Time (number of milliseconds elapsed since the DTN Epoch, i.e., 2000-01-01 00:00:00 +0000 UTC) at which the bundle was generated by the source and the sequence number, which is a monotonically increasing positive integer that can be optionally reset to 0 at each new millisecond.
- The lifetime i.e., a duration in milliseconds that when added to the creation time represents the time point from which the bundle payload will no longer be useful.
- The fragment offset, which is present only if the bundle is a fragment, represents at what offset the first byte contained in the payload of this fragment is in reference to the payload of the original bundle.
- The Total Application Data Unit Length, present only if the bundle is a fragment, represents the total length of the payload of the original bundle.
- The CRC value computed on the encoding of the Primary Block itself only if a CRC type value has been specified.

The Extension Blocks and the Payload Block are encoded in a common format called Canonical Bundle Block. Below we recall the information contained in each Canonical Bundle Block.

- A number identifying the type of the block called the Block Type Code. Within the same bundle there may be multiple blocks of the same type (e.g., Block Integrity Block and Block Confidentiality Block for which see [RFC9172]).
- A unique number that unambiguously identifies this block within the bundle.
- The Block Processing Control Flags, which are flags representing properties that apply only to this specific block.
- The CRC type, just like the one contained in the Primary Block. Different CRC types can be assigned to different blocks.
- The Block-Type-Specific Data i.e., the data contained by the block itself (in the case of the Payload Block this represents the Application Data Unit), along with its length in bytes.
- The CRC value computed on the encoding of this Canonical Bundle Block only if a CRC type value has been specified for this block.

The information needed to uniquely identify a bundle is the Source Node ID and the creation timestamp (comprising both the creation time and the sequence number). In case the bundle in question is a fragment, the fragment offset and fragment length (i.e., length of the Payload of that fragment) are also needed.

2.4 MAJOR IMPLEMENTATIONS

The goal of this thesis is to develop a new implementation of the Bundle Protocol version 7, RFC 9171 compliant, interoperable with all current implementations by means of either LTP or TCPCLv3 and the possible common use of CGR/SABR. The following are the BP implementations taken as references for this thesis primarily as targets for interoperability testing. All of them are released as free software.

Interplanetary Overlay Network (ION) [Burleigh_2007] is an implementation of Bundle Protocol released by NASA JPL (Jet Propulsion Laboratory). It is written in C

[ION] and it includes LTP, TCPCLv3 and CGR (original and Unibo-CGR). It can be used with Unibo-LTP in alternative to original LTP.

DTN Marshall Enterprise (DTNME) is a DTN2 [DTN2] fork made by NASA MSFC (Marshall Space Flight Center). DTN2 was the "reference implementation" of the Bundle Protocol, but it has not been updated since 2011. DTNME, written in C++, adds support for the Bundle Protocol Version 7 to DTN2. Its latest release, 1.2.0 beta, is very recent [DTNME]. It natively includes LTP and TCPCLv3. Unibo-CGR and the support of scheduled contacts can be added by installing Unibo software as described in [Gori_2020].

3 SOFTWARE ARCHITECTURE

This chapter discusses the design details that characterize this new implementation of Bundle Protocol Version 7. Unibo-BP is released as free software under "The GNU General Public License v3.0". The source code is published in a GitLab repository [Unibo-BP].

3.1 MOTIVATIONS

Unibo-BP was created to fill an important gap in the DTN software already developed by Unibo, such as Unibo-LTP, Unibo-CGR and the applications of the DTNsuite. All this software was created to work with other BP implementations, which had its own advantages and disadvantages. Concerning the former, developing software for ION or DTNME was particularly stimulating, as it allowed Unibo researcher and students to get in direct contact with their designers or maintainers, a unique experience in many respects. However, with the years, the increasing of Unibo software modules and the recent retirement from NASA of Scott Burleigh, the ION designer and maintainer, with whom there was a very strict collaboration, has increased also the challenges related to the dependence on software developed and maintained outside Unibo, as detailed in the next paragraph. This has led to the decision of developing a Bundle Protocol implementation fully managed by the University of Bologna, which could integrate in a wider umbrella project, Unibo-DTN, all software modules previously cited. The new BP implementation had to be designed with research in mind, to facilitate DTN research, in particularly at Unibo, in the years to come.

The main disadvantage of developing software to be integrated in third party BP implementations is the difficulty in keeping Unibo code up to date with new releases of BP. A simple change in the BP API would make Unibo software incompatible with the new release and thus it required a prompt response to adequate the software in short time, which was not always possible or easy, as a university has not the human resources of a software house. Even worse, when Unibo software was not an external application or module, but modifications made in the core of the original implementation dictated by research. A new version of the underlying BP

implementation meant the merging of existing Unibo code with the new implementation, a quite demanding task. To avoid this process, we often proposed the integration of our code or the introduction of new features, quite often with success, thanks to the kindness of BP implementation maintainers who integrated our modifications or our modules in their code (e.g., LTP enhancements and Unibo-CGR in ION). Although still possible, this approach has become more difficult in these very recent years, as BP is moving from research to deployment, which has led space agencies to focus more on deployment than on research.

With Unibo-BP these disadvantages are eliminated, as it will make all Unibo software independent of others, giving more freedom to academic research, although Unibo will try hard to continue collaborations with other maintainers.

Moving to more technical considerations, another goal to be achieved by Unibo-BP is the possibility to be used in tests by space agencies, which, with the introduction of high-speed links, makes the efficiency of code execution at runtime a significant aspect. Moreover, it must be considered that the code is going to be maintained and extended over the years primarily by undergraduate thesis students, who will have a very short time available to complete their thesis project. Therefore, it is paramount that the exiting code be easily readable and modifiable, i.e., written in a programming language familiar to them.

Keeping in mind the objectives mentioned above and for other technical reasons listed below, **C++** was chosen as the main programming language to develop Unibo-BP. It offers the following advantages:

- It allows the use of the object-oriented programming paradigm covered extensively in the Computer Engineering degree program at Unibo.
- It is supported by the Standard Template Library (STL), which contains many utility functions and data structures, thus making unnecessary to implement them from scratch or the use of third-party libraries. Problem that would occur if, for example, C were chosen as the programming language.

- It allows libraries written in C to be used natively and in turn Unibo-BP code to be used by programs written in C, albeit with necessary caveats. This is critical for integrating Unibo-BP with preexisting software modules, such as Unibo-CGR, Unibo-LTP, and Unified API, all written in C (for compatibility with ION).

- Finally, the latest standard versions of C++ not only will produce fast code, but also allow programmers to use high-level constructs, conjugating execution efficiency with code clarity and conciseness.

The following figure shows the different software components of the Unibo-DTN stack. During this thesis, the ones shown in solid colors were implemented, while the others were pre-existing and only needed an *ad hoc* interface to be implemented to make them compatible with Unibo-BP.

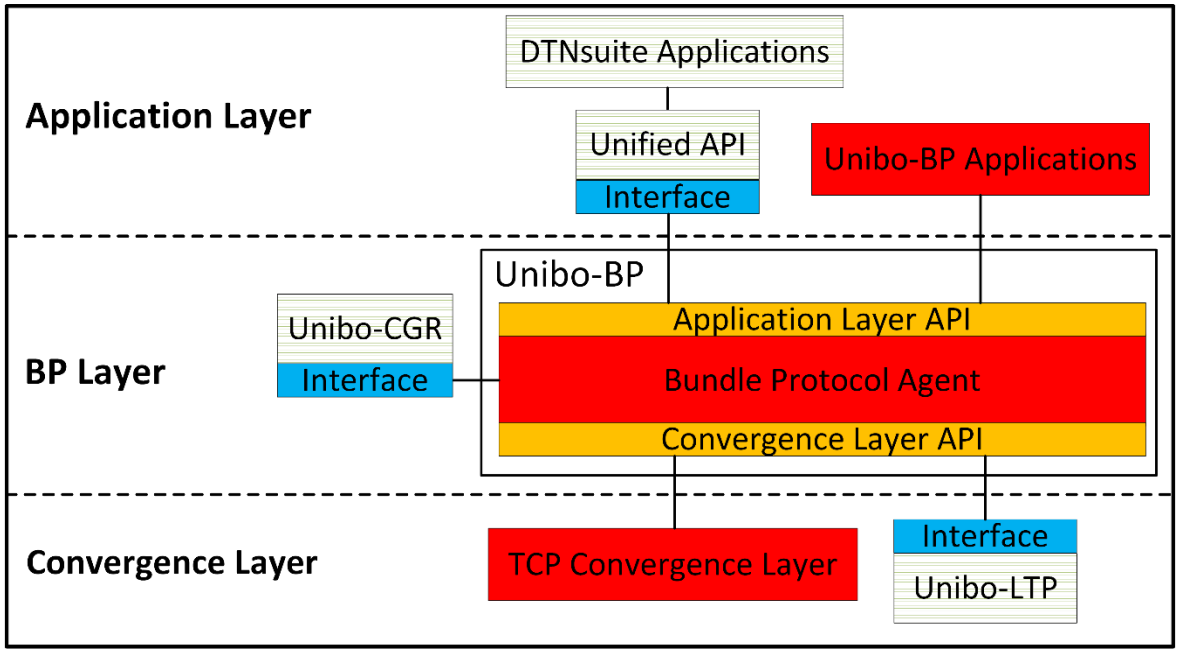


Figure 3.1 – The Unibo-DTN software stack.

3.2 High-Level Design

Unibo-BP is designed to be a highly distributed application, both multi-threaded and multi-process. More specifically, the Bundle Protocol Agent consists of a single process, but with multiple threads. Convergence Layer Adapters are independent processes that will communicate with the BPA by means of Inter-Process Communication. At the upper application layer, we have other processes, divided into administration

programs and applications interested in sending/receiving bundles, which also communicate with the BPA by means of Inter-Process Communication.

3.3 AUXILIARY LIBRARIES

Unibo-BP design is based on the use of a set of core libraries, designed and implemented during this thesis to complement the functionality offered by the C++ standard library. The most important ones are presented below.

3.3.1 The “io” library

The "io" library contains classes that, by means of the composition technique, allow the programmer to abstract from the input/output medium that is present at the lowest level (e.g., a file rather than a socket, or a memory buffer). The “io” library classes are somewhat designed after the equivalent classes of the “java.io” package, generally familiar to most programmers: there are both abstractions of the underlying devices and higher-level utilities, e.g., to perform buffered reads/writes.

The library also contains an implementation of a CBOR encoder/decoder [RFC8949] designed and developed from scratch during this thesis; the CBOR implementation is fully integrated with the rest of the “io” library and by making use of the composition technique. CBOR is the format used for bundle encoding/decoding in BPv7, but in Unibo-BP it has also been extensively used for Inter-Process Communication.

3.3.2 The “storage” library

The "storage" library implements a file-based memory allocator. The library allocator works as the C dynamic memory allocation (malloc), i.e., it limits itself at allocating a persistent memory block of the requested size. The allocated memory block is associated with an identifier, which for code efficiency coincides with the block offset within the file (like the pointer returned by a malloc); moreover, the memory block is tagged with the contained data type (a user-defined integer). At application startup, the library is responsible for retrieving the allocated memory blocks and the tag associated with each block from the file and making it known to the library user so that it can repopulate the volatile structures based on the contents of the persistent data.

This library is used to store bundles, whose dimension is unconstrained by RFC9171, in persistent memory, as well as for other uses.

3.3.3 The “time” library

The "time" library contains a few utility functions, such as to convert from UTC time (in string format) to Unix Time (*time_t* in C/C++ programming languages). It also contains a very important class, called "**dtn_clock**", which performs conversions between the DTN Epoch and system times obtainable via the *system_clock* class offered by C++. The dtn_clock class satisfies the clock requirements of the C++ chrono library.

3.3.4 The “math” library

The "math" library contains several classes and utility functions such as endianness converters, generators of unique thread-safe 64-bit identifiers, calculators of Cyclic Redundancy Check bytes, etc.

3.4 THE “BP” LIBRARY: DATA CLASSES

Several classes have been designed and implemented to abstract from the concepts presented in [RFC9171]. Only the most important are examined below, for treatment’s conciseness.

3.4.1 Endpoint ID

The Bundle Protocol standard stipulates that EIDs can belong to different schemes, which may impair code generality. The **EID** interface has been designed to avoid this risk, by abstracting the EID from the chosen scheme. It also offers a few utility methods to be implemented by the classes derived from it, such as the **IPN** and the **DTN** classes, which implement the homonymous schemes.

3.4.2 Primary Block, Payload Block and Extension Blocks

The **PrimaryBlock** class abstracts the data contained in the bundle primary block, and make them accessible by a set of “setter” and “getter” methods. Analogously, the

CanonicalBundleBlock class represents an abstraction of "Canonical" bundle blocks, from which the **PayloadBlock** and **ExtensionBlock** are derived.

The ExtensionBlock class, itself an abstract class, deserves a brief description. Extension blocks are optionally added to the primary and payload blocks, whose presence is compulsory, to add ancillary information to the bundle. Extensions can be of various types and obey different processing rules: for example, some extension blocks require to be processed when the bundle is received, others when it is about to be serialized upon dispatch, etc. For this reason, the ExtensionBlock class is designed as an abstract class, with virtual methods to be implemented by the derived classes. Each method is then called at the wanted specific phase of the bundle processing. A more detailed view of these methods will be provided in the next chapter.

3.4.3 Bundle

The **Bundle** class contains both volatile data and references to persistent data stored by using the storage library (e.g., the storage ID of the primary and payload blocks belonging to a bundle). Classes such as the PrimaryBlock, ExtensionBlock and PayloadBlock have both volatile and persistent representations. Each Bundle is associated with an identifier that is unique within the node, a 64-bit integer, saved in persistent memory. This identifier is designed to efficiently perform operations such as searching for a bundle within a node for administrative purposes.

3.4.4 Administrative Record

The **AdministrativeRecord** class is an interface for encoding/decoding administrative records of various types. At present there is only one implementation of this interface, the **StatusReport** class, which provides the programmer with getter/setter methods for the status report data [RFC9171] and with CBOR encoding/decoding methods.

3.4.5 Regions, contacts and ranges

Contact plan information, consisting of both contact and ranges, needs to be passed to both Contact Graph Routing and Licklider Transmission Protocol. To this end, Unibo-BP **Contact** and **Range** classes are designed to be used by Unibo-CGR and Unibo-LTP

interfaces. A peculiar innovation is that the EIDs of contact or range sender and receiver nodes are not limited to the "ipn" scheme anymore, but can follow any scheme supported by Unibo-BP. At present, this feature just allows a node to start/stop sending a bundle to another node, independently of the EID scheme, in accordance with contacts, but could be potentially extended to CGR or LTP in the future (e.g., by mapping dtn EIDs to numbers).

To solve the scalability problem of Contact Graph Routing, recent versions of ION have introduced the concept of hierarchical routing [Alessi_2019], based on dividing the network into a tree of regions that communicate with each other by means of certain nodes called "*passageways*", the sole belonging to two regions. Routing inside a Region is left in charge to CGR, as usual, while inter-regional routing is demanded to the brand new Inter-Regional Forwarding (IRF), implemented by Scott Burleigh in an experimental version of ION, kindly passed to us and studied in [Cingolani_2022].

Although IRF is still experimental, several classes have been implemented in Unibo-BP with the goal of being ready for a stable IRF implementation in the near future. The **Region** class contains the number of the region, the set of nodes (Node IDs) belonging to that region, the contacts and ranges between these nodes, and which of these nodes are passageways. Each passageway in turn is represented by the **RegionPassageway** class, which contains the Node ID of the passageway and the region to which that passageway is connected [Alessi_2019] [Cingolani_2022].

3.5 THE "BP" LIBRARY: SERVICES

In Unibo-BP, the Bundle Protocol Agent consists of several submodules, each of which implements a specific function. Most modules are executed as daemons in dedicated threads, others are invoked on demand and executed by the calling thread. As usual, we will limit the treatment to the most important classes.

The **BundleManager** class has the task of maintaining a reference to all bundles currently stored in the local node. To this end, each bundle created or received must be notified to the BundleManager. This class is also responsible, via an internal thread, for deleting bundles from the node when expired.

The **RoutingManager** determines to which neighbor(s) of the current node an incoming bundle must be sent, in order to reach their destination. Incoming bundles are queued by a special thread that is responsible for calculating routes. Once selected the neighbor(s), outgoing bundles are passed to the ForwardingManager.

The **ForwardingManager** tracks potential neighbor nodes, with three different priority queues for each neighbor (bulk, normal, expedited); bundle inserted into a queue by BP, will be then extracted by the CLA connecting the local node to the corresponding neighbor (multiple CLAs between the same two nodes are possible, in which case a choice must be made).

The **DeliveryManager** determines whether the local node is the bundle destination; if so, it passes the bundle to the RegistrationManager. In case of bundle fragmentation, the DeliveryManager must reassemble all fragments into the original bundle before passing it to the DeliveryManager (single fragments cannot be delivered, as stated by [RFC9171]).

The **RegistrationManager** keeps track of endpoints currently registered at the local node and allows applications registered to those endpoints to receive bundles. Registrations can be either in an active or passive state in accordance with [RFC9171]. An active registration in Unibo-BP requires that the application-specific element is connected to the BPA and is bound to a given EID; in contrast, a registration is in passive state if the application-specific element has previously bound to the BPA with a given EID but is no more connected to it at the time the bundle is received. In either case, the bundle is placed in the delivery queue to be later extracted by the application-specific element bound to the destination EID.

The **BPManager** class is a utility class designed to support generic functions, e.g., CBOR encoding or decoding of a bundle, or loading at startup the data stored in persistent memory (by means of the storage library).

The **ContactManager** provides the ability to add/remove/edit contacts and to automatically delete them when expired. Moreover, by means of an “observer” pattern, it allows other threads to register as “observers” of the operations involving contacts.

This way CGR and LTP internal bundle and range structures can be kept aligned with changes made to the BP contact plan.

The **RangeManager** class is analogous to the ContactManager, but for ranges.

The classes **RegionNodeManager** and **RegionPassagewayManager** deal with the registration of nodes within a region, or registration as passageways, respectively.

The **RegionManager** class manages two instances of the Region class: the "home" region, i.e., the region to which the local node belongs, and the "outer" region, i.e., the region that contains the home region.
Each Region class includes an instance of the ContactManager, RangeManager, RegionNodeManager and RegionPassagewayManager classes.

3.6 THE "IPC" LIBRARY

As mentioned above, a Unibo-BP node is composed of multiple processes, which need to communicate. Among the many different possible solutions, we opted for an Inter-Process Communication mechanism. The resulting architecture follows a client/server model and communication between client and server is performed by using Unix Domain Sockets with Stream type protocol.

The "ipc" library contains the utility functions used for encoding the messages exchanged by processes. The message header consists of a message ID (an integer) that represents the type of request and is generally associated with a different payload format. It was decided to use CBOR for the encoding of such messages. Each message is encoded with a CBOR Array (Major Type 4) of 2 elements where the first element is the header (a CBOR Unsigned Integer, Major Type 0) and the second element the payload (any CBOR Major Type). Thus, the payload can be as complex as desired (a CBOR Unsigned Integer rather than a CBOR Array etc.) and its format strictly depends on the type of data we are encoding (e.g., as administrative messages of type "add contact" and "add-range" contain different information, their payload is necessarily different).

3.7 THE "CLIENT" AND "SERVER" LIBRARIES

The "client" library contains the functions used to send requests and receive responses by means of the ipc library. Each request is associated with a method that takes the parameters needed to construct the payload as input and returns the data received from the server as output (or an error message in the form of an exception). This way the Inter-Process Communication mechanism is totally transparent to the client library user.

The "server" library runs servers within the BPA process, receives requests and send replies. There are three servers:

- The manager server, which handles administrative commands (e.g., add a contact or remove a contact). For each connection to the client a single dedicated thread is created by the server.
- The user server, which handles communication with the application-specific element to send or receive bundles. Two threads are created by the server for each connection to the client to handle full-duplex communication (to send and receive bundles at the same time).
- The CLA server, which accepts connections from the external convergence layers and passes them to the "cla" library where for each connection to the client there is a dedicated server that handles the connection. More information about the "cla" library can be found in the next section.

Received requests are decoded according to the message ID contained in the request header. Once the type of request is identified, the payload is decoded with the appropriate function. Most requests are fulfilled by simply returning the result of methods of the bp library classes to the client, while others involve interactions with other libraries, such as to obtain information about the level of persistent memory usage (storage library). Like the requests, the responses sent by the server are encoded in CBOR.

3.8 THE "CLA" LIBRARY

The Convergence Layer Adapter consists of a server running on the BPA process, and of client that implements specific Convergence Layers (e.g., LTP or TCP Convergence Layer). The "cla" library contains the implementation of utility classes for both server and client.

3.8.1 CLA server

On the CLA server side, it is necessary to abstract from the types of Convergence Layers available, as they are very different from each other. Therefore, we defined a special communication protocol between the CLA server and the CLA client to exchange information on contacts, ranges and link openings/closures, as well as to send or receive bundles: some messages involve a response (thus a request/response protocol), while others do not require synchronization and do not involve a response (mainly for efficiency reasons). Moreover, additional information, such as ECOS (Extended Class of Service) fields, is passed to the Convergence Layer (client) when sending a bundle. Such information is currently used only by the LTP Convergence Layer to select the session color to be used (e.g., green or red) but the opportunities offered by passing additional parameters to the lower layer are many and largely unexplored. This is another innovative feature of Unibo-BP which may result useful in future research.

The **Link** class contains abstract information about an outgoing link to a given neighbor. It is worth stressing that multiple links to the same neighbor could be created, either belonging to the same or different CLAs. Each link has a unique identifier, established by the CLA server, used to inform the CLA client which connection to use to send a bundle to the neighbor. The CLA client uses this identifier also to inform the CLA server about openings/closures of the link itself. Each link is also associated with ancillary information regarding reliability or the maximum size of the protocol data unit (serialized bundle) that can be sent on that link.

The **Peer** class deals with managing all links, potentially belonging to various CLAs, directed to a given neighbor. This is the class that interacts with the

ForwardingManager to extract from the BP queue the bundles to be sent toward the neighbor itself. The BP queue differs from the convergence layer queue in that the former does not refer to a specific convergence layer link while the latter does. This means that once a bundle is extracted from the BP queue and sent to a convergence layer it may be placed in a queue before it is sent to the neighbor. Bundles are extracted from the BP queue only if there is an open contact toward the neighbor and if at least one link is currently open. In addition, flow control is applied to try to meet the nominal data rate declared by the current contact, similarly to what done by ION's bpclm daemon. These operations are performed within the Peer class via dedicated threads communicating with each other.

The **CLA** class is created as soon as an external application connects to the CLA server and handles communication with it. Each instance of this class is associated with a unique ID, established by the CLA client and validated by the CLA server (by means of the **CLAManager** class), which allows information to be passed to a given CLA client through the Unibo-BP server. A given message is then sent by the node administrator to the Unibo-BP server, which based on the CLA ID will take care of passing it to the right CLA. This mechanism is particularly useful as it allows remote administration of CLAs through the BP itself as we will see later.

3.8.2 CLA client

On the client CLA side, the **CLAOverIPC** class makes possible to abstract from the IPC mechanism used. By means of a dedicated thread this class listens for messages sent by the server (e.g., a bundle is to be sent) and propagates them to the Convergence Layer by means of the methods offered by an instance of the **BPToCLAController** interface implemented by the CLA client. Different CLAs client may have different needs, which is why it is required that each CLA client implement its own version of the BPToCLAController by means of an appropriate derived class.

3.9 REMOTE ADMINISTRATION

One of the most innovative features of Unibo-BP is the ability to natively administer a node by remote through the BP itself. The idea behind this feature is to send IPC

commands to a Unibo-BP node as a payload of a bundle. An application-specific element listening at a given EID is then responsible for receiving the bundle and propagating the payload to the local node in the form of an administrative message. Having used an interoperable encoding such as CBOR, the Inter-Process Communication presented earlier can be extended to the exchange of messages between different nodes without any modifications. As mentioned earlier this feature is also extended to CLAs connected to a Unibo-BP node.

4 IMPLEMENTATION DETAILS

This chapter provides the reader with an overview of Unibo-BP implementation details.

4.1 LIBRARIES

Each library is organized as follows: below the root directory we have an "include" directory, which contains the headers of public functions and structures, a "src" directory, which contains the source code implementing public and private functions/classes, and an optional "test" directory, in case unit tests (or other) are available.

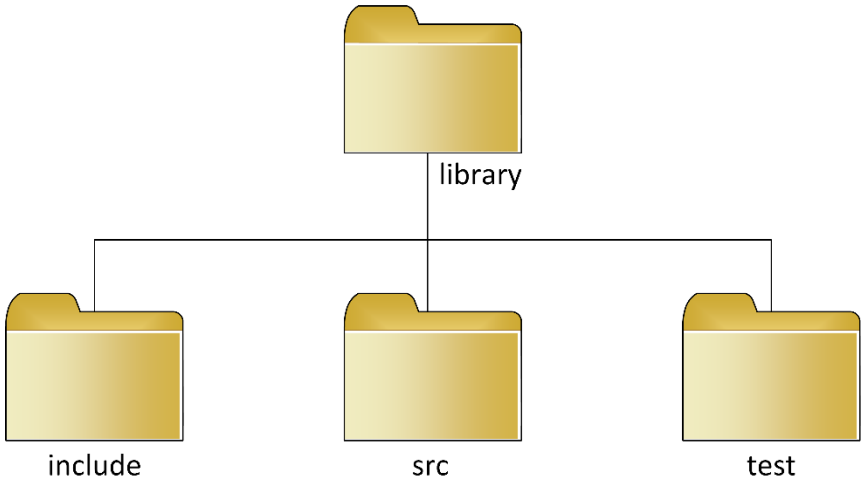


Figure 4.1 - Directory structure for libraries.

4.1.1 Shared symbols

Since Unibo-BP libraries are used by multiple components, we chose to compile them as shared libraries. In order to reduce the size of the symbol table, and thus speed-up dynamic linking, all symbols are hidden by default (by means of the GCC option "-*fvisibility=hidden*"). This choice also helps to avoid unnecessary ABI (Application Binary Interface) breaks when only hidden symbols, internal to the library, are modified.

This, on the other hand, requires making explicit the symbols to be exported (made accessible to library users), which is done by setting the following attribute, recognized by the GCC and Clang compilers (both supported by Unibo-BP respectively starting from version 10 and 11):

```
__attribute__((visibility("default")))
```

Each public symbol must therefore be preceded by this attribute.

To make the declaration of public symbols easier, and possibly to support other compilers in the future, the header-only library "export" and “import” macros were created, which simply mask this attribute in the following way.

```
#define UNIBO_BP_PUBLIC_API_EXPORT
__attribute__((visibility("default")))
#define UNIBO_BP_PUBLIC_API_IMPORT
__attribute__((visibility("default")))
```

Note that working with shared libraries a symbol can not only be exported but also imported into a library and while in the case of GCC and Clang the attribute to be used is the same, other compilers use two distinct attributes. This is why we decided to separate “import” and “export” macros from now.

Finally, we need to determine whether we need to include a public header file (i.e., a header file that exports symbols that can be used by other libraries) to compile the library or to use (i.e., to link) it. In the former case we are interested in exporting symbols, while in the latter we are interested in importing them. We, therefore, defined two additional macros for each library, as shown below with reference to the bp library:

```
#ifdef BUILD_BP_API
#  define BP_API UNIBO_BP_PUBLIC_API_EXPORT
#else
#  define BP_API UNIBO_BP_PUBLIC_API_IMPORT
#endif
```

If the *BUILD_BP_API* macro is defined by the build system, the library is going to be compiled, and therefore public symbols are to be exported. Otherwise, the library is going to be linked and the symbols are to be imported. The *BP_API* macro is the one that is to be used by the bp library developer to declare the symbols to be exported.

Note that the import/export handling is now a build system issue and is totally transparent to the user of the bp library. Only the library developer needs to determine which symbols are public, to declare them as such.

An example of how the PrimaryBlock class is exported/imported using the BP_API macro is shown below. Each symbol is defined in the public headers of the library and thus the library user does not have to do anything beyond using them.

```
class BP_API PrimaryBlock;
```

4.2 ROUTING

As anticipated, the BPA is composed of several daemons that communicate by sending bundles to each other with the goal of increasing the degree of parallelism and to separate the various processing steps of a bundle (e.g., acquisition, routing, forwarding etc.). Each daemon is structured in the same way, in order to communicate. We take routing as an example, first by giving a general explanation and later by showing the communication in detail.

At present there are two types of routing in Unibo-BP: static and dynamic. The former, consists of rules entered manually by the network administrator, as usual, while the latter enforces Contact Graph Routing by means of Unibo-CGR. To simplify the introduction of additional routing types, the generic **IRouter** interface has been defined, to be implemented by each router. The following method is the only one exposed by this interface.

```
virtual bool try_routing(std::shared_ptr<Bundle>& bundle) = 0;
```

This method must return false if no route is found, otherwise true; in the latter case it must also pass the bundle ownership to the **Forwarder** class representing the neighbor selected by the ForwardingManager.

Routers are called sequentially by the RoutingManager daemon until the first returns true or all routers have returned false (in which case the bundle is discarded and a status report, if required, is generated). Currently, static routing is performed first, and dynamic routing is performed next.

4.2.1 RoutingManager

As in other classes, such as the DeliveryManager, the RoutingManager internally runs a daemon that extracts one bundle at a time from the daemon queue and try to find a viable route for it. A synchronization mechanism based on *mutexes* and *condition variables* is used to submit bundles to that daemon. The following is an excerpt of the code used by the RoutingManager daemon, but also by other daemons in Unibo-BP, to wait for a new bundle to be processed.

```
while (true) {
  std::unique_lock lck(data->mtx);
  data->cv.wait(lck, stoken, [data] { return !data-
>bundle_queue.empty(); });
  while (!data->bundle_queue.empty()) {
    std::shared_ptr<Bundle> bundle = std::move(data-
>bundle_queue.front());
    data->bundle_queue.pop();

    /* . . . daemon specific code . . .*/

    if (stoken.stop_requested()) {
      break;
    }
  }
}
```

To insert a bundle into the daemon's queue, also awakening the daemon, if necessary, we use this code.

```
std::unique_lock lck(data->mtx);
if (data->bundle_queue.empty()) {
  data->cv.notify_one();
```

```
}
data->bundle_queue.push(std::move(bundle));
```

4.2.2 StaticRouter

Currently, static routing consists only of a rule type, which states to send a bundle to neighbor X if destined to node Y. The association is done manually by the network administrator. Other types of rules may be defined in the future.

Static routing is implemented by the **StaticRouter** singleton class.

4.2.3 CGRRouter

An interface for Unibo-CGR has been implemented to use Contact Graph Routing in Unibo-BP. The main difference from other existing interfaces for other BP implementations, namely ION and DTNME, is that in Unibo-BP contacts and ranges can be defined in EID schemes different from “ipn”. As CGR necessarily uses numbers to identify nodes, we defined an **EID64BitConverter** interface to perform a bidirectional conversion between an EID in Unibo-BP and a 64-bit integer representing a BP node in Unibo-CGR. The following methods return true if the conversion was successful and false otherwise.

```
virtual bool convert(const EID& input, std::uint64_t& output) noexcept =
0;
virtual bool convert(std::uint64_t input, std::unique_ptr<EID>& output)
= 0;
```

At present, only the **IPN64BitConverter** class has been implemented, which handles the conversion for the ipn scheme (which is of course the simplest case, since an ipn node is already defined as a 64-bit integer).

Contact Graph Routing is accessed through the singleton class **CGRRouter** which internally manages two instances of Unibo-CGR: one for the "home" region and the other for the "outer" region, which is useful dealing with passageways, which must always check if the destination node is in either their home or outer region (i.e., they need to call CGR twice, with different contact plans, one for each region [Alessi_2019],

[Cingolani_2022]). A very interesting and innovative feature of this class, it is that it also exposes several methods to modify the behavior of Unibo-CGR.

```
void change_reference_time(std::chrono::sys_seconds new_reference_time);
void enable_logger();
void disable_logger();
void enable_one_route_per_neighbor(std::uint64_t limit);
void disable_one_route_per_neighbor();
void enable_queue_delay();
void disable_queue_delay();
void enable_proactive_anti_loop();
void disable_proactive_anti_loop();
void enable_reactive_anti_loop();
void disable_reactive_anti_loop();
void enable_moderate_source_routing();
void disable_moderate_source_routing();
```

This means that instead of enabling/disabling experimental features [Caini_2021] by modifying a *config.h* file, the same features can be enabled/disabled at run-time, without the need of recompiling the code, an obvious important advantage for research (changes can be inserted into a general "dotest" script file).

4.3 EXTENSION BLOCKS

Several extension blocks have been implemented in Unibo-BP. Specifically, in addition to those described in [RFC9171] (Bundle Age, Previous Node and Hop Count), we added RGR (Record Geographical Route) and CGRR (CGR-Routes) used by experimental enhancements of SABR included in Unibo-CGR [Caini_2021] [Birrane_2021]. Moreover, we have implemented the ECOS Block [Draft_ECOS], not yet standardized in an RFC, which uses the same encoding format as ION for interoperability reasons.

The following are the virtual functions of the ExtensionBlock class that should be overridden by derived classes.

The **ExtensionBlockOutputMask** type is a bitmask that shows what information in the extension block has been modified by the function. Based on the asserted bits, certain operations are undertaken on the extension block, such as re-serializing the block-type-specific data because the contents of the extension block have been modified (e.g., for the Previous Node Block, before sending the bundle to the neighbor the block is overwritten with the EID of the local node).

The function below is called when the extension block is inserted into a bundle. Currently this occurs only at bundle creation, in which case we have the following syntax:

```
virtual ExtensionBlockOutputMask create(Bundle &bundle);
```

Future policies could require that certain extension blocks be inserted into bundles received from other nodes. In this case we can distinguish many subcases, as shown below.

The function below is called when the bundle is inserted into the BP queue of the neighbor to which it is to be sent:

```
virtual ExtensionBlockOutputMask enqueue(Bundle &bundle);
```

The function below is called when the bundle is extracted from the BP queue because it can be sent to the neighbor (i.e., the contact is open and a link is available).

```
virtual ExtensionBlockOutputMask dequeue(Bundle &bundle);
```

The function below is called as soon as a bundle received from an external node has been decoded.

```
virtual ExtensionBlockOutputMask acquire(Bundle &bundle);
```

This function below is called whenever it is necessary to serialize the contents of the extension block due to changes to it.

```
virtual bool
serialize_block_type_specific_data_content(const Bundle& bundle,
                                           io::CborEncoder &encoder);
```

This function below is called when the bundle has been received from an external node and it is used to decode the content of the block-type-specific data.

```
virtual void
deserialize_block_type_specific_data_content(Bundle& bundle,
                                             io::CborDecoder &decoder);
```

4.4 INTER-PROCESS COMMUNICATION

All messages exchanged by the Inter-Process Communication, described in 3.6, are implemented following a consistent pattern. Requests are grouped into macro-blocks and are delimited by identifiers named "*lower_bound*" and "*upper_bound*" that represent the range of values (open interval) reserved for that block. Each request is then associated with a class, which contains the "*message*" enumerative representing the unique identifier of the type of request, and the functions to encode and decode the payload. The class for the group of messages used by the Application-Specific Element to transmit bundles is shown below, as an example:

```
struct IPC_API ipc_request_user_bundle_transmission {

  enum message : std::uint64_t {
    lower_bound = ipc_user_bundle_transmission_lower_bound,
    bundle_send,
    bundle_cancel,
    /* ... add new values here ...*/
    upper_bound
  };

  static void encode_payload_bundle_send(io::CborEncoder& encoder,
                                         bp::Bundle& bundle);

  static void decode_payload_bundle_send(io::CborDecoder& decoder,
                                         bp::Bundle& bundle);
```

```
  static void encode_payload_bundle_cancel(io::CborEncoder& encoder,
                                           std::uint64_t bundle_unique_id);

  static void decode_payload_bundle_cancel(io::CborDecoder& decoder,
                                          std::uint64_t& bundle_unique_id);

};
```

The implementation of the functions related to the "*bundle_cancel*" request are the following:

```
void ipc_request_user_bundle_transmission::encode_payload_bundle_cancel(
              io::CborEncoder &encoder, std::uint64_tbundle_unique_id) {
 encoder.encode_uint(bundle_unique_id);
}

void ipc_request_user_bundle_transmission::decode_payload_bundle_cancel(
              io::CborDecoder &decoder, uint64_t &bundle_unique_id) {
 bundle_unique_id = decoder.decode_uint();
}
```

On the server side, to simplify the addition of new messages, for each received message the header is decoded and the request identifier is categorized according to the belonging block based on the *lower_bound* and *upper_bound* values of the block (i.e., the request identifier is contained in some block interval); then the function associated with the specific request is selected, as shown in the following code excerpt:

```
switch (request) {
 case ipc_request_user_bundle_transmission::lower_bound:
 break;
 case ipc_request_user_bundle_transmission::bundle_send:
 handle_request_bundle_send(handle);
 return;
 case ipc_request_user_bundle_transmission::bundle_cancel:
 handle_request_bundle_cancel(handle);
 return;
 case ipc_request_user_bundle_transmission::upper_bound:
```

```
 break;
}
```

Each handling function decodes the request payload, performs the associated operation and finally returns the result to the client.

4.5 APPLICATION PROGRAMMING INTERFACE

Unibo-BP is a multi-process application in which the Application-Specific Element and CLAs run on different processes. External BP applications connect to Unibo-BP by means of a public API implemented by a library named "**unibo-bp-api**". This library exposes public functions whose prototypes adhere to the C (and not C++) syntax, so that programs such as Unified API and Unibo-LTP, both written in C, can natively use them. The API's functions are essentially wrappers of the C++ methods offered by the client and cla library classes.

Most functions return an enumerative, named **UniboBPError**, which if different from the value **UniboBP_NoError** implies that an error of some kind occurred during the function execution. Such errors range from unverified preconditions in the input parameters to a possible connection shutdown.

4.5.1 Application-Specific Element API

All Unibo-BP API functions follow the same pattern, so that we can limit the treatment to just a few, such as those used by the Application-Specific Element to connect to the BPA and send or receive bundles.

The **unibo_bp_connect** function, shown below, connects the BP application to the BPA server running in the directory passed as a parameter. If successful, the handle parameter (a pointer to an opaque type) is initialized.

```
UniboBPError unibo_bp_connect(
             const char* directory,
             UniboBPUserHandle* handle);
```

The **unibo_bp_connect_default** has the same functionality but attempts to connect to a server running in the directory path contained in the environment variable **UNIBO_BP** or, if this is undefined, to the working directory.

```
UniboBPError unibo_bp_connect_default(UniboBPUserHandle* handle);
```

The **unibo_bp_disconnect** releases the resources used by the handle parameter and disconnects from the server.

```
void unibo_bp_disconnect(UniboBPUserHandle* handle);
```

The **unibo_bp_get_admin_id** returns as output, via the EID parameter, the administrative endpoint of the local node related to the scheme passed as parameter.

```
UniboBPError unibo_bp_get_admin_id(
             UniboBPUserHandle handle,
             UniboBPScheme scheme,
             UniboBPEID* eid);
```

The **unibo_bp_bind** binds the application-specific element to a specific EID.

```
UniboBPError unibo_bp_bind(
             UniboBPUserHandle handle,
             UniboBPEID* eid);
```

The **unibo_bp_bind_random** is like the previous one but connects to a random EID assigned by the server in the scheme desired by the caller. For the ipn scheme a random service number is assigned, while for the dtn scheme a random demux token is assigned. In both cases a number is randomly chosen.

```
UniboBPError unibo_bp_bind_random(
             UniboBPUserHandle handle,
             UniboBPScheme scheme);
```

The **unibo_bp_get_registered_eid** returns, via the EID parameter, a read-only access to the EID to which the handle is registered. Particularly useful if the *unibo_bp_bind_random()* function has been used to bind to an EID.

```
UniboBPError unibo_bp_get_registered_eid(
             UniboBPUserHandle handle,
             ConstUniboBPEID* eid);
```

The **unibo_bp_get_sender_handle** returns the opaque type that can be used by the thread responsible for sending bundles.

```
UniboBPSenderHandle unibo_bp_get_sender_handle(
                    UniboBPUserHandle handle);
```

The **unibo_bp_send_bundle** sends a bundle using the data contained in the opaque **UniboBPOutboundBundle** type that was previously initialized by the caller through appropriate calls to the "set" functions. If successful, the UniboBPOutboundBundle structure is populated with the creation timestamp data established by the BPA. Note that if “success” is retuned, this does not mean that the bundle has been sent, but simply that the BPA has agreed to send it (it might fail, for example, if the payload of the bundle is too large).

```
UniboBPError unibo_bp_send_bundle(
             UniboBPSenderHandle handle,
             UniboBPOutboundBundle bundle);
```

The **unibo_bp_get_receiver_handle** returns the opaque type that can be used by the thread responsible for receiving bundles.

```
UniboBPReceiverHandle unibo_bp_get_receiver_handle(
                      UniboBPUserHandle handle);
```

The **unibo_bp_receive_bundle_blocking** waits to receive a bundle until a timeout is triggered or another error occurs. On exit, it returns the received bundle via the opaque

UniboBPInboundBundle type, which defines various “get” functions to read the bundle fields.

```
UniboBPError unibo_bp_receive_bundle_blocking(
             UniboBPReceiverHandle handle,
             UniboBPInboundBundle* bundle,
             uint64_t timeout_ms);
```

4.6 CODE BUILDING AND INSTALLATION

CMake is used to generate the build system for Unibo-BP. We decided to use a build system generator, instead of a simple Makefile, to ease the project management and to avoid portability issues. In particular, the choice fell on CMake since it is the “de facto” industry standard for handling C++ projects and is well known by the open-source community.

For user convenience, a Makefile serves as a wrapper for the CMake commands. The following is the typical command sequence for installing Unibo-BP using the Makefile.

```
make init
make
sudo make install
sudo ldconfig
```

Additional options and commands are documented by the *make help* command.

4.7 THE UNIBO-BP NODE

Unibo-BP can start different nodes on the same machines, by associating different directories to each node. In more details, when a Unibo-BP node is started, several directories and files are created in the local filesystem:

- The "**.unibo-bp**" directory it is created in the working directory where the node is started. It acts as parent directory.
- The "**admin**" directory contains the Unix Domain Socket "**admin.sock**" created by the server that handles administrative messages.

- The "**cla**" directory contains the Unix Domain Socket "**cla.sock**" created by the CLA server.
- The "**inbound**" directory contains the bundles (one temporary file for each bundle) received by the Convergence Layer Adapters sent by neighbors.
- The "**outbound**" directory contains the bundles (one temporary file for each bundle) serialized by the local BPA for forwarding purposes.
- The "**storage**" directory contains the "**global.storage**" file, which is the sole file used for persistency.
- The "**user**" directory contains the Unix Domain Socket "**user.sock**" created by the server that handles Application-Specific Element messages.
- The "**unibo-bp.running**" file, which is used as a marker file, to avoid that multiple Unibo-BP nodes are started in the same directory. The presence of this file in a directory means that another Unibo-BP node cannot be started in the same directory. This file is automatically deleted when the node is successfully stopped. In the unlucky case of node crash, this file must be manually removed in order to restart the node in the same directory.

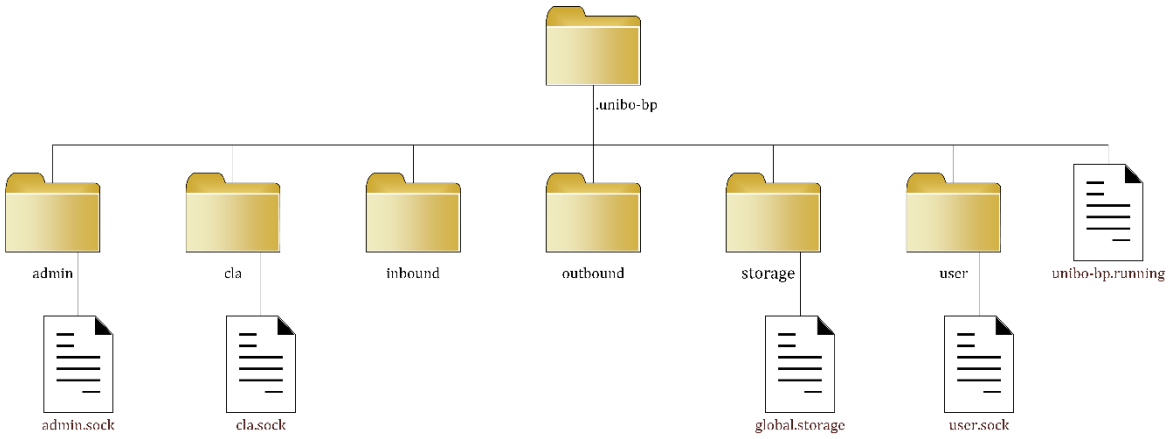


Figure 4.2 - Overview of the ".unibo-bp" directory.

As mentioned, the only limitation is that it is not possible to start multiple nodes in the same directory. However, it is always possible to start multiple nodes within the same machine, as long as different directories are used. This may not seem useful in a deployment scenario, but it can be very convenient for any kind of testing not involving channel challenges: a multiple node scenario can be created without the need of

implementing a testbed with several containers or virtual machines. We used this feature extensively during the development phase of Unibo-BP itself.

5 COMMAND-LINE INTERFACE

Differently from other BP implementations, Unibo-BP is not configured via configuration files, but a series of executables that are installed in the user's operating system can be used for this purpose. Each of these programs provides a command-line interface (CLI) that can be interpreted by a command-line shell, such as "bash"(Bourne Again Shell) [Bash]. It is then up to the user to decide whether to invoke the commands manually from the terminal or to group them within a script for future reuse. This chapter describes the command-line interface containing both administration commands and some utility programs for running tests.

As most GNU/Linux programs, Unibo-BP executables return 0 if successful and print nothing on the standard output; otherwise, if an error occurs, a value other than 0 is returned and an error message is printed on the standard error device. However, this rule is not universal: some of the programs print the result on the standard output if successful, such as the request to output the contact plan loaded on the node. Other commands, when started as daemons, print the PID (Process ID) on the standard output for the user's convenience.

Each program consists of a command name, which may be followed by parameters (optional unless otherwise specified) and subcommands (grouping options or other subcommands), as shown below:

```
<command name> [options] [subcommand name [options]]
```

A command can be followed by one or multiple subcommands; those at the same depth level are mutually exclusive.

A "**--help**" option is present for each command and subcommand; each command also contains the "**license**" subcommand that allows the license of Unibo-BP and third-party libraries to be printed on the standard output.

As CLI readability was the primary concern, all options adhere to the "long" format. Within a command or subcommand, options can be specified in any order.

The commands contained in Unibo-BP are described below. Note that commands involving interaction with the BPA, will attach with the BPA instance executed in the directory path specified by the **UNIBO_BP** environment variable; if this is undefined, they will connect to the BPA executed in the current working directory.

5.1 UNIBO-BP

The "**unibo-bp**" command starts the BPA and consists of two subcommands: "**start**" and "**resume**".

5.1.1 start

The former command starts a new node from scratch, by cleaning any pre-existing storage.

The mandatory parameters supported by this subcommand are listed below.

- **--set-storage-size <integer number>** specifies the size (in bytes) of the file (or memory buffer in case of volatile-only representation) used to store persistent data.
- **--ipn-admin <node id>** specifies the ipn Node ID of this node.
- **--dtn-admin <node id>** specifies the dtn Node ID of this node.

The options supported by this subcommand are listed below.

- **--daemon** executes the process as a daemon returning the command prompt to the user.
- **--set-storage-type <string>** specifies the preferred storage type: *persistent*, *volatile* and *mmap.* Persistent means on files, volatile on memory, and memory-mapped on a volatile cache of data saved on persistent files. Default: mmap.

As an example, the command below starts the BPA from scratch by specifying the local ipn Node ID "ipn:1.0" and dtn Node ID "dtn://vm1.dtn/". A persistent file of 50 MB will be created to serve as storage medium; the storage file is memory-mapped (mmap). Last, the BPA is started as a daemon.

```
unibo-bp start --set-storage-type mmap --set-storage-size 50000000
--ipn-admin ipn:1.0 --dtn-admin dtn://vm1.dtn/ --daemon
```

5.1.2 resume

The resume subcommand restarts a node that was previously stopped, maintaining the same internal configuration. For this purpose, data are read from the BPA persistent memory.

The options are a subset of those already presented for the start subcommand. Specifically, they are: *--daemon*, *--set-storage-type* and *--set-storage-size*.

As an example, below the node is restarted using the same configuration used in the previous run.

```
unibo-bp resume --daemon
```

5.2 UNIBO-BP-ADMIN

The "**unibo-bp-admin**" command groups all administration directives, which are issued as IPC messages. These commands are usually sent to the local node, but they can also be encapsulated in bundles and then sent to other nodes.

The "**--file <file path>**" option must be used to send a remote command. The IPC message is saved in a file (append mode) instead of been sent to the local node. If the specified file does not exist, it is created.

The unibo-bp-admin command consists of a variety of subcategories discussed below.

5.2.1 stop

The "**stop**" subcommand starts a graceful shutdown procedure. Note that the command termination (the return of the prompt), does not mean that the node has already stopped, but only that the shutdown request has been accepted.

```
unibo-bp-admin stop
```

5.2.2 logger

The "**logger**" subcommand is used to administer the Unibo-BP logger, implementing multiple verbosity levels. Log levels are used to declare which log lines are not to be written (i.e., all those with a level less than the current log level) and which are to be written (i.e., all those with a level greater than or equal to the current log level).

- **--get-log-level** permits to retrieve the current log level.
- **--set-log-level <level>** specifies the minimum log level. Log levels range from the most permissive (all) to the least (disabled), they are in order: *all*, *trace*, *debug*, *info*, *exception*, *warning*, *error*, *critical*, *fatal*, *disabled*. Default: *info*.
- **--write-log <level> <string>** writes a log line filtered by the specified log level. Logs are usually written by the BPA itself in the **unibo-bp.log** file created in its current working directory. However, logs can also be sent to the BPA by means of IPC messages. This option sends a log line, tagged with the specified level, to the BPA; then the BPA decides if the log line should be written or not based on the level shipped with the log line itself.
- **--set-self-name <name>** when used with *--write-log* allows the user to set the specific name of whoever is writing that log line.

```
unibo-bp-admin logger --get-log-level
unibo-bp-admin logger –-set-log-level trace
unibo-bp-admin logger --set-self-name cli --write-log info “Hi!”
```

5.2.3 storage

The "**storage**" subcommand is used to get information about Unibo-BP storage status.

- **--get-storage-info** prints information about the storage usage (e.g., number of free bytes etc.).
- **--get-storage-type** returns the storage type.

5.2.4 region

The "**region**" subcommand is used to administer the "home" and "outer" regions of a passageway [Alessi_2019] [Cingolani_2022], by means of the subcommands **home** and **outer**.

The options listed below are available for both subcommands, if not specified otherwise.

- **--set-region-number <number>** to change the current region number.
- **--get-region-number <number>** to retrieve the current region number.
- **--register-node <node id>** to declare a node member of the region.
- **--get-node-list** to retrieve the list of nodes that are members of the region.
- **--get-passageway-list** to retrieve the list of nodes that are passageways.
- **--get-contact-list** to receive the list of (still to expire) contacts belonging to the region.
- **--get-range-list** the same as *--get-contact-list* but for ranges.
- **--deregister-node <node id>** it removes the node from the region, along with all contacts and ranges associated with it. The rationale of this behavior is that in IRF contact plans are regional, i.e., we have a different contact plan in each region; in a region contact plan, only nodes belonging to the region must be cited, i.e., can compare as either sender or receiver nodes of contact or range instructions.
- **--deregister-passageway <node id>** allows the user to remove the passageway status of the node passed as argument.
- **--print-utc-time** to print time in UTC format (e.g., used with *--get-contact-list* and *--get-range-list*). UTC time format is the default, so using this option is not actually required.
- **--print-relative-time <time point>** to print time (e.g., contact start time) in relative format, i.e., with reference to a time "zero" passed as argument. E.g., "+50" means 50 seconds after the reference time. The reference time can be passed either in UTC format or in relative time notation (e.g., "+25" referred to the current time).

- **--register-passageway <node id>** to declare a node passageway.
- **--passageway-region-number** this option is available only for the "*outer*" subcommand; used with *--register-passageway* it permits to declare the home region of the passageway. This because in IRF passageways belong to two regions, the home and the outer, by contrast to all other nodes, which belong to one region only; thus, for passageways it is necessary to specify which region must be considered as their home region.

```
unibo-bp-admin region home --set-region-number 2
unibo-bp-admin region outer --get-region-number
unibo-bp-admin region home --get-node-list
unibo-bp-admin region home --get-passageway-list
unibo-bp-admin region home --get-contact-list --print-relative-time
2022-12-14T18:51:24Z
unibo-bp-admin region home --get-range-list --print-relative-time +10
unibo-bp-admin region outer --register-node ipn:2.0
unibo-bp-admin region outer --deregister-node ipn:2.0
unibo-bp-admin region home --register-passageway ipn:2.0
unibo-bp-admin region outer --register-passageway ipn:2.0 --passageway-
region-number 3
unibo-bp-admin region home --deregister-passageway ipn:2.0
```

5.2.5 routing

The "**routing**" subcommand consists of two (second layer) subcommands: "**static**" and "**cgr**".

The subcommand "**static add**" adds a rule into the static router tables and has the following parameters:

- **--destination <node id>** states the destination for which we are inserting the rule (named *destination rule*).
- **--gateway <node id>** declares the neighbor to which send the bundle if a match is found in the static router tables.

As an example, the following command forces each bundle destined to the node “ipn:2.0” to be forwarded to the neighbor “ipn:3.0”.

```
unibo-bp-admin routing static add --destination ipn:2.0 --gateway
ipn:3.0
```

The subcommand “**static remove**” is the dual of the add command; it has only the following parameter:

- **--destination <node id>** states the destination rule to remove.

As an example, the command below removes the rule associated to the destination node “ipn:2.0”.

```
unibo-bp-admin routing static remove --destination ipn:2.0
```

The subcommand “**static get**” retrieves a table from the static router and has only the following option:

- **--destination-table** permits to retrieve the rules belonging to the *destination table*

```
unibo-bp-admin routing static get --destination-table
```

The subcommand “**cgr**” is used to administer Unibo-CGR features. The options are shown below.

- **--set-reference-time <time point>** to change the Unibo-CGR reference time for logging purposes. The time point can be passed in both relative and UTC format.
- **--enable-logger** to enable the Unibo-CGR logger.
- **--disable-logger**
- **--enable-one-route-per-neighbor [limit]** to enable the “one route per neighbor” enhancement [Caini_2021], with an optional limit to the number of neighbors for which a viable route should be found for each destination.
- **--disable-one-route-per-neighbor**
- **--enable-queue-delay** to enable the “queue delay” enhancement [Caini_2021].

- **--disable-queue-delay**
- **--enable-proactive-anti-loop** to enable the “proactive anti loop” algorithm [Caini_2021].
- **--disable-proactive-anti-loop**
- **--enable-reactive-anti-loop** to enable the “reactive anti loop” enhancement [Caini_2021].
- **--disable-reactive-anti-loop**
- **--enable-moderate-source-routing** to enable the “Moderate Source Routing” enhancement [Birrane_2021].
- **--disable-moderate-source-routing**

The subcommands are summarized in the following reference list:

```
unibo-bp-admin routing cgr --set-reference-time +0
unibo-bp-admin routing cgr --set-reference-time 2022-12-14T18:51:24Z
unibo-bp-admin routing cgr --enable-logger
unibo-bp-admin routing cgr --–disable-logger
unibo-bp-admin routing cgr --enable-one-route-per-neighbor
unibo-bp-admin routing cgr --enable-one-route-per-neighbor 2
unibo-bp-admin routing cgr --disable-one-route-per-neighbor
unibo-bp-admin routing cgr --enable-queue-delay
unibo-bp-admin routing cgr --disable-queue-delay
unibo-bp-admin routing cgr --enable-proactive-anti-loop
unibo-bp-admin routing cgr --disable-proactive-anti-loop
unibo-bp-admin routing cgr --enable-reactive-anti-loop
unibo-bp-admin routing cgr --disable-reactive-anti-loop
unibo-bp-admin routing cgr --enable-moderate-source-routing
unibo-bp-admin routing cgr --disable-moderate-source-routing
```

5.2.6 contact

The "**contact**" subcommand is usually used to add or remove contacts by means of the "**add**" and "**remove**" (second layer) subcommands, respectively; it is also possible to receive information about a specific contact by means of the "**get**" (second layer)

subcommand or to modify contact's information by means of the "**change**" (second layer) subcommand.

Note that by default the contact is inserted/found within the region to which both the contact's sender and receiver nodes belong. If this region is not found, the command fails. It is necessary to register nodes in the common region of the node to be inserted and the local node, before making any change to the contact plan.

Below are the parameters of the "**add**" subcommand, used to insert a contact into the contact plan. Note that contacts between the same pair of "sender" and "receiver" nodes should never overlap in time. Each parameter specifies an element of the contact to be inserted into the contact plan. We remind the user that contacts and ranges are unidirectional.

- **--sender <node id>** specifies the node from which data are sent.
- **--receiver <node id>** specified the node from which data are received.
- **--start-time <time point>** specifies the start of the contact in seconds. Relative time and UTC formats are both supported.
- **--end-time <time point>** specifies the time in seconds at which the contact expires. Relative time and UTC formats are both supported.
- **--xmit-rate <integer>** specifies the nominal transmission rate (bytes/s), when the contact is active.

The parameters listed above are all mandatory, while those listed below are optional.

- **--reference-time <time point>** specifies the reference time to which the start and end times refer, when entered as relative times. The *time point* argument could be passed in both relative time and UTC format.
- **--type <string>** specifies the type of the contact. Only the *scheduled* type is currently supported. Default: s*cheduled*.
- **--confidence <real number>** specifies how confident we are that contact will occur (confidence can be seen as a sort of informal qualitative probability introduced with the opportunistic versions of CGR [Burleigh_2016]). The argument must be a real number in the range [0.0, 1.0]. Default: 1.0.

```
unibo-bp-admin contact add --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --end-time +50 --reference-time 2022-12-14T18:51:24Z --xmit-rate 125000
unibo-bp-admin contact add --sender dtn://node2/ --receiver dtn://nodethree/ --start-time +25 --end-time +50 --reference-time 2022-12-14T18:51:24Z --xmit-rate 125000
```

The "**remove**" subcommand is used to delete a specific contact. The five parameters "*--sender*", "*--receiver*", "*--reference-time*", "*--start-time*" and *"--type"*, which retain the same meaning already discussed for the "add" subcommand, are necessary to uniquely identify the contact. They will be required also by all other commands that works on one specific contact.

```
unibo-bp-admin contact remove --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --reference-time 2022-12-14T18:51:24Z
```

The "**get**" subcommand is used to retrieve the characteristics of a specific contact. It has the same parameters as the "remove" subcommand used to uniquely identify the contact, plus the two optional parameters listed below.

- **--print-utc-time** to print the contact time interval in UTC format.
- **--print-relative-time <time point>** to print the contact time interval in relative format, starting from the time point passed as argument (relative time or UTC).

```
unibo-bp-admin contact get --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --reference-time 2022-12-14T18:51:24Z --print-utc-time
unibo-bp-admin contact get --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --reference-time 2022-12-14T18:51:24Z --print-relative-time 2022-12-14T18:51:20Z
```

The **"change**" subcommand is used to modify the characteristics of a contact. It requires the same parameters as the "remove" subcommand to identify the contact and adds the ones listed below.

- **--new-start-time <time point>** specifies the new contact start time. Relative time and UTC formats are both supported.
- **--new-end-time <time point>** specifies the new contact end time. Relative time and UTC formats are both supported.
- **--new-xmit-rate <integer>** specifies the new data transmission rate.
- **--new-type <string>** specifies the new contact type.
- **--new-confidence <real number>** specifies the new contact confidence.

```
unibo-bp-admin contact get --sender ipn:2.0 --receiver ipn:3.0 --start-
time +25 --reference-time 2022-12-14T18:51:24Z --new-end-time +40
unibo-bp-admin contact get --sender ipn:2.0 --receiver ipn:3.0 --start-
time +25 --reference-time 2022-12-14T18:51:24Z --new-xmit-rate 150000
$ unibo-bp-admin contact get --sender ipn:2.0 --receiver ipn:3.0 --
start-time +25 --reference-time 2022-12-14T18:51:24Z --new-start-time
+30
```

5.2.7 range

The "**range**" subcommand is very similar to the "*contact*" subcommand but operates on ranges.

The "**add**" subcommand shares with the "*contact add*" subcommand the parameters "*--sender"*, "*--receiver"*, "*--reference-time"*, "*--start-time*" and "*--end-time*". Note that to uniquely identify a range we need only the first four. Moreover, it offers the following mandatory parameter:

- **--owlt <integer>** specifies the "one-way-light-time" (i.e., the nominal propagation delay from the sender to the destination), in seconds, of the range.

```
unibo-bp-admin range add --sender ipn:2.0 --receiver ipn:3.0 --start-
time +25 --end-time +50 --reference-time 2022-12-14T18:51:24Z --owlt 1
```

The "**remove**" subcommand shares with the "*add*" subcommand the parameters "*--sender"*, "*--receiver"*, "*--reference-time"* and "*--start-time*", necessary to uniquely identify the range.

```
unibo-bp-admin range remove --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --end-time +50 --reference-time 2022-12-14T18:51:24Z
```

The "**get**" subcommand offers the same parameters of the "*remove*" subcommand, necessary to uniquely identify the range, plus the "*--print-utc-time*" and "*--print-relative-time*" already discussed for the "contact get" subcommand.

```
unibo-bp-admin range get --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --reference-time 2022-12-14T18:51:24Z
unibo-bp-admin range get --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --reference-time 2022-12-14T18:51:24Z --print-relative-time
2022-12-14T18:51:20Z
```

The "**change**" subcommand requires the same options of the "*remove*" subcommand to identify the range, plus the "*--new-start-time*" and "*--new-end-time*" parameters, as for the corresponding "*contact change*" subcommand. It also offers another option discussed below.

- **--new-owlt <integer>** specifies the new "one-way-light-time" of the range.

```
unibo-bp-admin range get --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --reference-time 2022-12-14T18:51:24Z --new-end-time +40
unibo-bp-admin range get --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --reference-time 2022-12-14T18:51:24Z --new-owlt 2
unibo-bp-admin range get --sender ipn:2.0 --receiver ipn:3.0 --start-time +25 --reference-time 2022-12-14T18:51:24Z --new-start-time +30
```

5.2.8 extension

The "**extension**" subcommand allows the user to enable or disable the insertion of experimental extension blocks into the bundles created by the local node. At present it can be used to enable/disable the extensions RGR (Record Geographical Route) and CGRR (CGR-Routes), used by a few extensions of Unibo-CGR. In the future it might be extended to other functionalities.

- **--enable-rgr** enables the RGR extension block.
- **--disable-rgr** disables the RGR extension block.

- **--enable-cgrr** enables the CGRR extension block.
- **--disable-cgrr** disables the CGRR extension block.

```
unibo-bp-admin extension --enable-rgr
unibo-bp-admin extension --disable-rgr
unibo-bp-admin extension --enable-cgrr
unibo-bp-admin extension --disable-cgrr
```

5.2.9 whoami

The "**whoami**" subcommand returns the Node ID of the local node. The scheme of the Node ID can be specified using the "**--scheme <string>**" option (*ipn* and *dtn* are both supported).

```
unibo-bp-admin whoami --scheme ipn
unibo-bp-admin whoami --scheme dtn
```

5.2.10 tcpcl

The "**tcpcl**" subcommand is used to control the TCP Convergence Layer process. The "**--cla-id <integer>**" option is useful to identify a specific TCPCL process in case several instances are present, as they are differentiated by the CLA Identifier (an integer). The default CLA Identifier for the TCPCL is 1000.

The "**induct add**" subcommand spawns a new server listening on specific hostname and port. For this purpose, the optional parameters "**--hostname <string>**" and "**--port <integer>**" are provided to override their defaults, *any* machine address (0.0.0.0 in IP v4) and 4556, respectively.

```
unibo-bp-admin tcpcl induct add
unibo-bp-admin tcpcl induct add --hostname localhost
unibo-bp-admin tcpcl induct add --port 12345
unibo-bp-admin tcpcl induct add --hostname localhost --port 12345
```

The "**outduct add**" subcommand specifies a neighbor to which the TCPCL should connect. The mandatory parameters are listed below.

- **--peer <node id>** specifies the neighbor's Node ID.
- **--hostname <string>** specifies the neighbor's TCPCL hostname (or IP address).
- **--port <integer>** specifies the neighbor's TCPCL port. Default: 4556.

```
unibo-bp-admin tcpcl outduct add --peer ipn:3.0 --hostname 10.0.1.2
unibo-bp-admin tcpcl outduct add --peer ipn:3.0 –-hostname
www.addressresolvedbydns.com
unibo-bp-admin tcpcl outduct add --peer ipn:3.0 --hostname
www.addressresolvedbydns.com --port 23456
```

The "**stop**" subcommand shuts down the TCPCL process.

```
unibo-bp-admin tcpcl stop
```

5.3 UNIBO-BP-REMOTE-ADMIN

The "**unibo-bp-remote-admin**" program is used to enable the reception and execution of remote administration commands. In order to use this feature, the network administrator must only specify the listening EID, to which bundles containing remote administration commands are to be delivered. Note that at present this program works only one-way, i.e., it reads the administration command from the received bundle's payload and send it to the local BPA to which the unibo-bp-remote-admin program is connected; possible replies issued by the local BP are just ignored. This limitation could, however, be easily removed, in new Unibo-BP releases.

The potential of the remote administration via bundles has to be fully investigated yet. Other BP implementations, such as DTNME, offer the administrator the ability to send remote commands to the BPA by means of a TCP connection. This technique has the obvious advantage of being able to administer a BPA without requiring the administrator to run a BPA on his own machine. However, it has the disadvantage of not being able to (natively) administer remotely a node placed in a challenged network. Instead, with Unibo-BP, we wanted to generalize the concept of remote administration to nodes placed in challenged networks as well but requiring the network administrator to execute a BPA on his own machine since remote commands must be sent via bundles. Obviously, to be of some interest in deployment scenarios, the

security issues implicit in the use of remote commands should be counteracted using Bundle Protocol Security [RFC9172], which may be object of future work.

The "**--daemon**" option is the only one actually accepted by this program and has the same meaning as the homonymous option shown for the "*unibo-bp start*" subcommand in 5.1.1.

```
unibo-bp-remote-admin ipn:2.3
unibo-bp-remote-admin ipn:2.3 --daemon
```

Remote commands should be prepared by exploiting the "*--file*" option of the *unibo-bp-admin* program. Then, they should be sent to the remote EID by means of the "*unibo-bp-send*" utility program or any other user-defined program able to use the Unibo-BP API to send a bundle.

5.4 UNIBO-BP-TCPCL

The "**unibo-bp-tcpcl**" program is used to spawn an instance of the TCP Convergence Layer. The "**--cla-id <integer>**" option permits to override the default CLA ID associated with the TCPCL (i.e., 1000) with a user-defined value; this option might be useful if a network administrator wants to spawn multiple processes, all running the TCPCL, in order to balance the computational load between them The CLA ID is used by the "*unibo-bp-admin tcpcl*" command to administer a specific TCPCL. The "**--daemon**" option is also supported with the usual behavior.

```
unibo-bp-tcpcl --daemon
unibo-bp-tcpcl --cla-id 1234 --daemon
```

5.5 UNIBO-BP-PING

The "**unibo-bp-ping**" program is a simple "ping service" over BP. It sends a series of bundles at regular intervals at an "echo service" and expects to receive them back after some time. As the usual ICMP ping program shipped with most Linux distributions, the time elapsed between the creation of the "ping bundle" and the reception of the "echo bundle" is printed on the standard output. When the program terminates, several

statistics are printed on the standard output including the average RTT and the percentage of bundles lost.

This program requires to specify the EID of the echo service and supports the following options:

- **--source <eid>** specifies the source EID of the bundle. Otherwise, an ipn EID is generated using the local ipn node number and a random service number.
- **--count <integer>** stops the program after "count" bundle are sent. With the deadline option (see below), the program waits until the timeout expires, or all bundles are acknowledged, whichever comes first. The default is *unlimited*.
- **--deadline <integer>** forces the program to stop when the specified number of seconds have elapsed. The default is *unlimited*.
- **--interval <integer>** specifies the number of seconds between one bundle and the next. The default is 1 second.

```
unibo-bp-ping ipn:3.7
unibo-bp-ping --source ipn:2.1923 dtn://echonode/echoservice
unibo-bp-ping --interval 2 --deadline 10 ipn:123.456
unibo-bp-ping --count 5 --deadline 10 ipn:123.456
```

5.6 UNIBO-BP-ECHO

The "**unibo-bp-echo**" program is a simple "echo service" over BP. It listens to a specific EID and sends back the received data (the whole payload) to the bundle's source. It is designed to work as a counterpart of the "*unibo-bp-ping*" program, but it can work with any BP ping program that expects to receive back a bundle containing the same payload as the bundle sent.

By default, this program listens to the local node ipn EID with service number 7. The "**--scheme <string>**" option permits to override the scheme, e.g., by specifying "dtn" as argument (in that case, the default demux token "*echo*" is used). The "**--daemon**" option is also supported as usual.

The EID to which the echo service listens can be optionally forced by specifying the desired EID.

```
unibo-bp-echo --daemon
unibo-bp-echo ipn:2.12345 --daemon
unibo-bp-echo dtn://mynode/myechoservice --daemon
unibo-bp-echo --scheme dtn --daemon
```

5.7 UNIBO-BP-SEND

The "**unibo-bp-send**" program allows to issue a single bundle to a user-defined destination EID. Several parameters, shown below, are available.

- **--source <eid>** specifies the bundle's source EID. Otherwise, a random ipn EID is used.
- **--report-to <eid>** specifies the bundle's report-to EID. Otherwise, "*dtn:none*" is used.
- **--destination <eid>** specifies the bundle's destination EID. This parameter is mandatory.
- **--lifetime <integer>** specifies the bundle's lifetime in milliseconds. Otherwise, the default (60000 ms) is used.
- **--payload-file <file path>** specifies the path to a file whose content is to be used as bundle's payload.
- **--payload-string <string>** specifies the string to be used as bundle payload.
- **--do-not-fragment** specifies that the bundle must not be fragmented (i.e., the proper Bundle Processing Control Flag is asserted).
- **--forwarded** specifies that "*forwarded*" status reports are to be generated (i.e., the proper Bundle Processing Control Flag is asserted).
- **--received** specifies that "*received*" status reports are to be generated (i.e., the proper Bundle Processing Control Flag is asserted).
- **--delivered** specifies that "*delivered*" status reports are to be generated (i.e., the proper Bundle Processing Control Flag is asserted).
- **--deleted** specifies that "*deleted*" status reports are to be generated (i.e., the proper Bundle Processing Control Flag is asserted).
- **--priority <string>** specifies the "cardinal" priority (i.e., bulk, normal or expedited).

- **--ordinal <integer>** specifies the ECOS “ordinal” priority (meaningful only if used together with expedited priority; the integer should be in the interval [0,255].).
- **--hop-limit <integer>** specifies the hop limit to be used in the Hop Count Extension Block.

```
unibo-bp-send --destination ipn:123.456 --report-to ipn:789.101112 --
deleted --priority bulk --payload-string “Hello, World!” --hop-limit 10
unibo-bp-send --destination dtn://destinationnode/service --lifetime
1000000 --priority normal --payload-file /path/to/payload/file
```

5.8 UNIBO-BP-SINK

The “**unibo-bp-sink**” program receives bundles and prints their payload content on the standard output; the listening EID must be specified.

```
unibo-bp-sink ipn:2.13464
unibo-bp-sink dtn://mynode/mydemux
```

5.9 UNIBO-BP-UTILITY

The “**unibo-bp-utility**” program is designed as a container of generic utilities. At present, it can be called only with the “**--get-utc-time [relative time]**” option which prints the current time in UTC format on the standard output; this option is designed primarily to be used in conjunction with the "*--reference-time*" option shown for contact and range administration. For user convenience, the optional argument "relative time" can be used to add an offset specified in seconds to the current time: in this case, the program will print the time obtained by adding the offset to the current time.

```
unibo-bp-utility --get-utc-time
unibo-bp-utility --get-utc-time +10
```

6 INTEROPERABILITY TESTS

During the development of Unibo-BP, several tests have been performed: first to verify that DTN nodes deployed with Unibo-BP are able to communicate with each other; later to verify the proper functioning of the interfaces implemented for Unibo-CGR, Unibo-LTP, and Unified-API; and finally, to verify the interoperability with other BP implementations. In this chapter we show some interoperability tests performed with ION and DTNME (both RFC 9171 compliant). We consider a very simple scenario where Unibo-BP plays either the role of source, destination or intermediate node.

6.1 NETWORK TOPOLOGY

Virtualbricks [Virtualbricks], which is a network emulator developed at Unibo, was used to build a multi-node layout (Figure 6.1), where each node is instantiated on a different virtual machine with Debian 11, and the latest ION 4.1.1 and DTNME 1.2.0_Beta (the latest releases available at the writing of this thesis) installed, plus Unibo-DTN software. Although the different BP implementations were available on all machines, in interoperability tests we devoted one virtual machine to each BP implementation, namely **vm1** to DTNME **vm2** to Unibo-BP and **vm3** to ION.

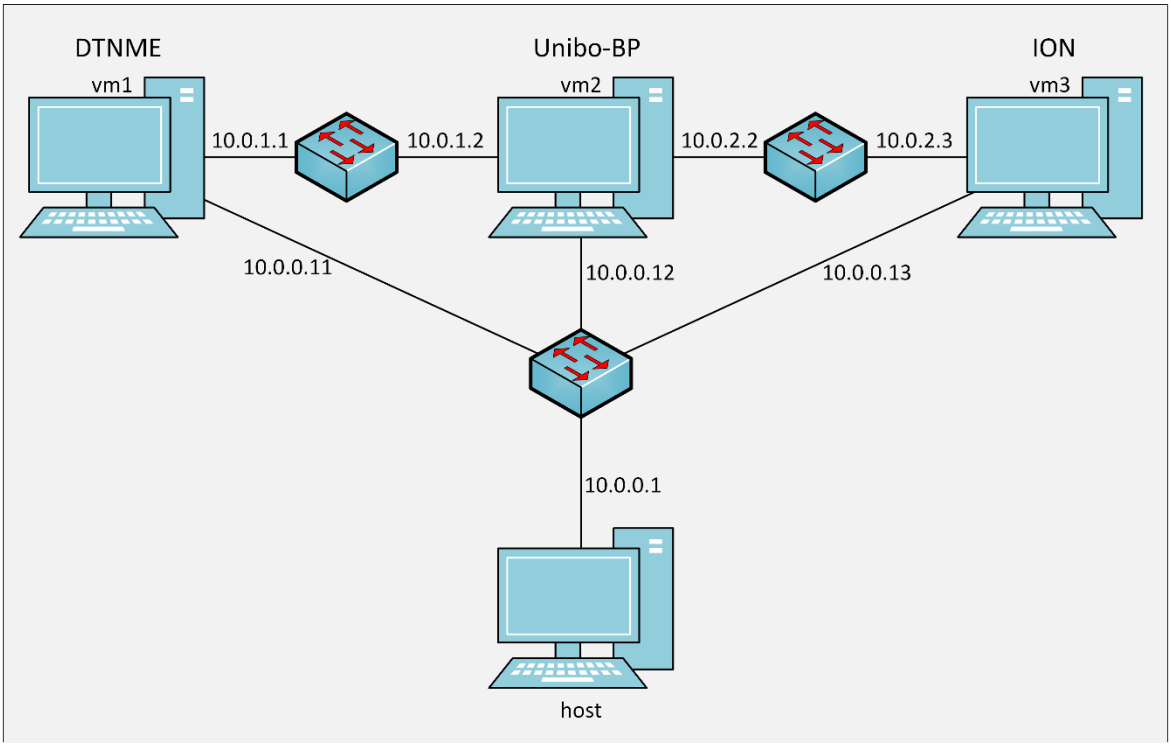


Figure 6.1 - Network topology used to conduct interoperability tests.

The testbed consists of a control network and an experiment network (two DTN hops). The former, subnet **10.0.0.0/24,** is used to control the virtual machines via SSH (Secure Shell) from the host without interfering with test traffic, which exclusively uses the experiment network, i.e., subnets **10.0.1.0/24** and **10.0.2.0/24**. Specifically, the former connects vm1 and vm2, while the latter connects vm2 and vm3. These subnets have a network emulator inside to introduce some challenge between VMs (huge propagation delay, high packet loss rate, etc.). However, as challenges are irrelevant as far as interoperability is concerned, these network emulators are not used in interoperability tests and also the experiment network can be considered ideal.

The table below shows the IP addresses associated to each machine.

Machine	IP Address in subnet 10.0.0.0/24	IP Address in subnet 10.0.1.0/24	IP Address in subnet 10.0.2.0/24
host	10.0.0.1	-	-
vm1	10.0.0.11	10.0.1.1	-
vm2	10.0.0.12	10.0.1.2	10.0.2.2
vm3	10.0.0.13	-	10.0.2.3

Table 1 - IP addresses of machines in the various network segments.

The ipn Node IDs associated to DTNME, Unibo-BP and ION are respectively: **ipn:1.0**, **ipn:2.0**, **ipn:3.0**. DTN nodes communicate bidirectionally by means of the TCP Convergence Layer version 3: ipn:1.0 is connected to ipn:2.0 which in turn is connected to ipn:3.0.

6.2 TESTS DESCRIPTION

Although several choices were possible, DTNperf [Caini_2013] was chosen to carry out the tests, so that we could also serendipitously prove not only the interoperability of Unibo-BP, but also that of DTNsuite applications, such as DTNperf. Moreover, we have the following additional advantages:

- It demonstrates that the Unibo-BP API can be effectively used by external programs (in this case, DTNperf by means of Unified API) for sending and receiving bundles.
- The DTNperf client prints several useful pieces of information on the standard output, including the BP implementation to which it is connected (found at run time on the basis of running processes), its registration EID (the source of the bundles sent) and for each bundle sent, the creation timestamp (time in ms and sequence number). The last three elements uniquely identify a bundle. This way, it is easy to cross-check DTNperf client output with Wireshark [Wireshark].
- The DTNperf server also prints the BP to which it is connected but, more importantly, it prints on the screen the payload size along with the source of each bundle received; more exhaustive information (e.g., the timestamp, to identify the received bundle) can be print by increasing the level of verbosity.

Each test is organized in the following way: the source node starts an instance of the DTNperf client and sends 5 bundles of 1kB each to the destination node where an instance of the DTNperf server is running, by using the following syntax:

```
dtnperf --client -d <destination EID> -D5k -P1k -R1M
```

The destination EID is the only parameter that needs to be changed in different tests. The Data parameter (**-D**) specifies the total number of bytes to transmit (5 kB), while the Payload (**-P)** specifies the payload length of each bundle (1 kB). The Rate (**-R)** specifies the congestion control (rate-based): it is intentionally larger (1 Mbit/s) than the total number of bytes transmitted during the test (in one second) since we are not interested in congestion control.

In all tests the DTNperf server automatically registers with the ipn scheme (chosen automatically by Unified-API library on which DTNperf is based, for ION and Unibo-BP) with service number 2000. The syntax to run the DTNperf server is the following:

```
dtnperf --server
```

For DTNME a slight variation is required to use the "ipn" scheme, as in this case it is necessary to override the default selection of Unified-API, which is dtn for DTN2/DTNME). Moreover, it is also necessary to specify the local ipn node number (1 in the example), as shown below.

```
dtnperf --server --force-eid IPN --ipn-local 1
```

6.3 UNIBO-BP AS A SOURCE NODE

In this section are shown the tests carried out with Unibo-BP acting as the source node. The DTNperf client always runs on vm2, while the server on vm1 or vm3 depending on the BP implementation we want to have on the destination node, DTNME or ION, respectively. The protocols stack is shown in the figure below.

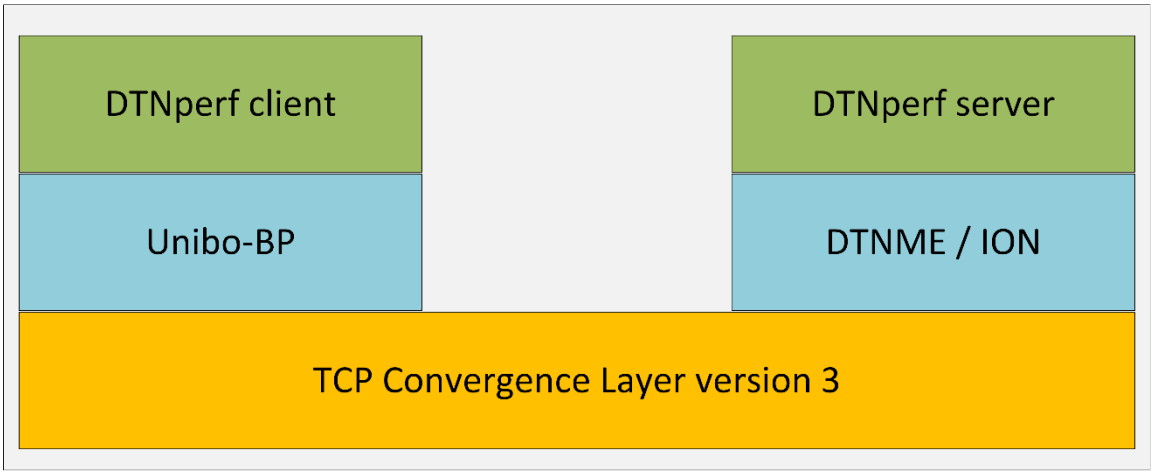


Figure 6.2 – Protocol stack when Unibo-BP runs as source node.

In the following, the output of the DTNperf client and DTNperf server is shown for each test, along with the capture made by Wireshark of the bundles transmitted by means of TCPCL.

6.3.1 Unibo-BP (source) and DTNME (destination)

```
student@vm2: ~/test-msc-thesis-persampieri 94x22
student@vm2:~/test-msc-thesis-persampieri$ dtnperf --client -d ipn:1.2000 -D5k -P1k -R1M
DTNperf version: 4.0.1
Running BP implementation: Unibo-BP
UNIFIED_API version:   2.0.17
Registered at:
char* Eid: "ipn:2.611"
[DEBUG_L0] Bundle timestamp: 727208578701.0
[DEBUG_L0] Bundle timestamp: 727208578709.0
[DEBUG_L0] Bundle timestamp: 727208578717.0
[DEBUG_L0] Bundle timestamp: 727208578725.0
[DEBUG_L0] Bundle timestamp: 727208578734.0

Bundles sent = 5, total data sent = 5.000 kbyte
Total execution time 0.050 seconds
Throughput = 800.000 kbit/s
student@vm2:~/test-msc-thesis-persampieri$
```

Figure 6.3 - DTNperf client executed on the Unibo-BP node. Five bundles, generated by "ipn:2.611", with a payload of 1 kB each were sent to "ipn:1.2000" (the DTNME node running on vm1).

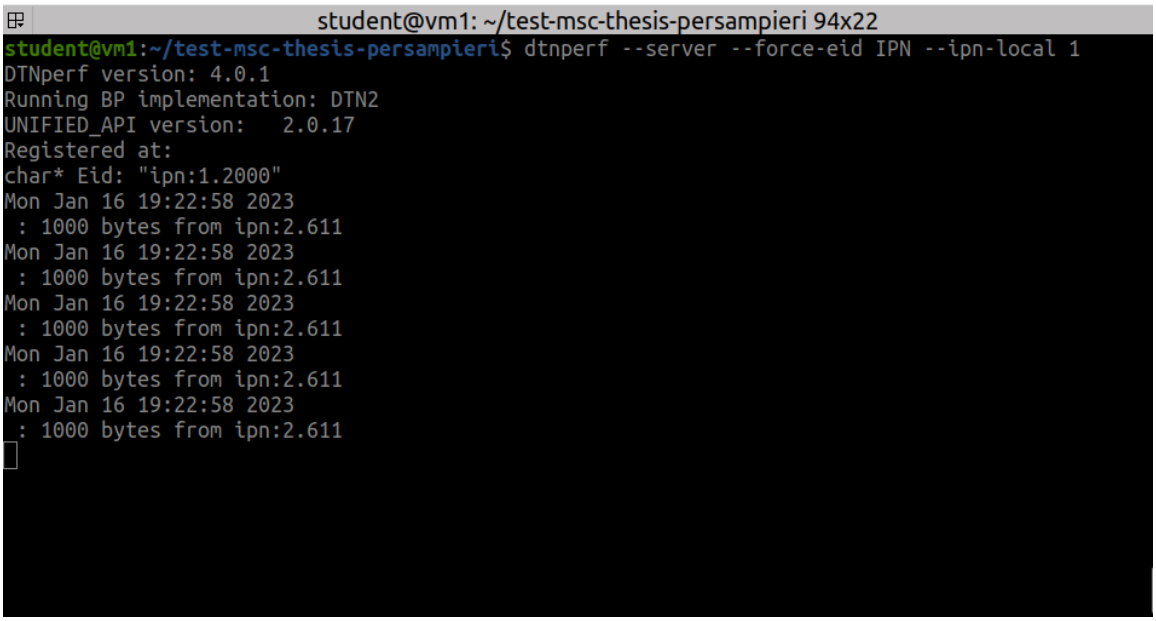


Figure 6.4 – DTNperf server ("ipn:1.2000") executed on the DTNME node. Five bundles, sent by "ipn:2.611", of 1 kB each were received.

No.	Time	Source	Destination	Protocol	Length	Info
51	49.681853	10.0.1.2	10.0.1.1	BPv7	1154	ipn:2.611 → ipn:1.2000
53	49.689182	10.0.1.1	10.0.1.2	TCPCL	69	ACK_SEGMENT
57	49.690409	10.0.1.2	10.0.1.1	BPv7	1154	ipn:2.611 → ipn:1.2000
59	49.691162	10.0.1.1	10.0.1.2	TCPCL	69	ACK_SEGMENT
63	49.698413	10.0.1.2	10.0.1.1	BPv7	1154	ipn:2.611 → ipn:1.2000
65	49.705546	10.0.1.1	10.0.1.2	TCPCL	69	ACK_SEGMENT
69	49.706869	10.0.1.2	10.0.1.1	BPv7	1154	ipn:2.611 → ipn:1.2000
71	49.707469	10.0.1.1	10.0.1.2	TCPCL	69	ACK_SEGMENT
75	49.756262	10.0.1.2	10.0.1.1	BPv7	1154	ipn:2.611 → ipn:1.2000
77	49.758322	10.0.1.1	10.0.1.2	TCPCL	69	ACK_SEGMENT

Figure 6.5 - Wireshark capture of the five bundles sent from "ipn:2.611" (Unibo-BP, DTNperf client) to "ipn:1.2000" (DTNME, DTNperf server).

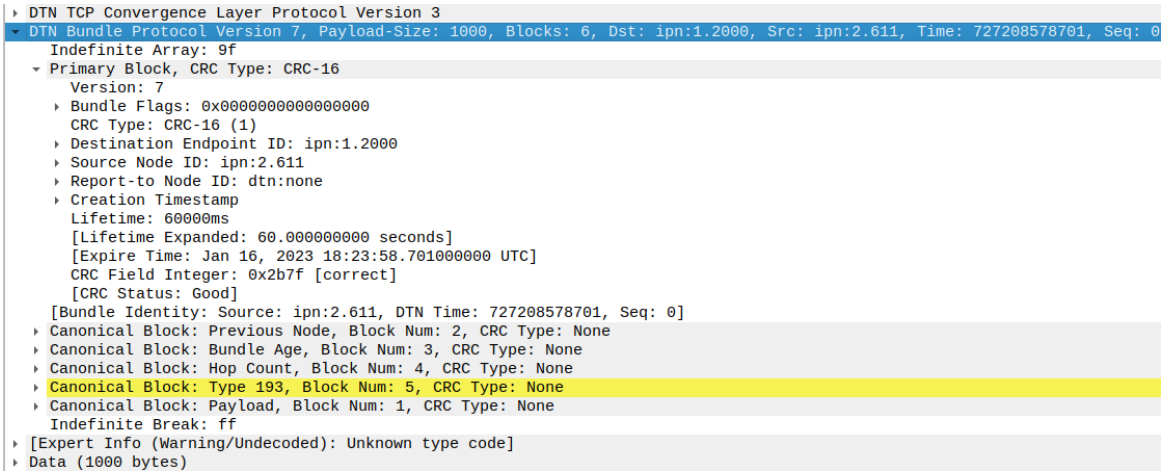


Figure 6.6 - Dissection of the first bundle sent by the DTNperf client ("ipn:2.611"). The Primary Block is expanded.

6.3.2 Unibo-BP (source) and ION (destination)

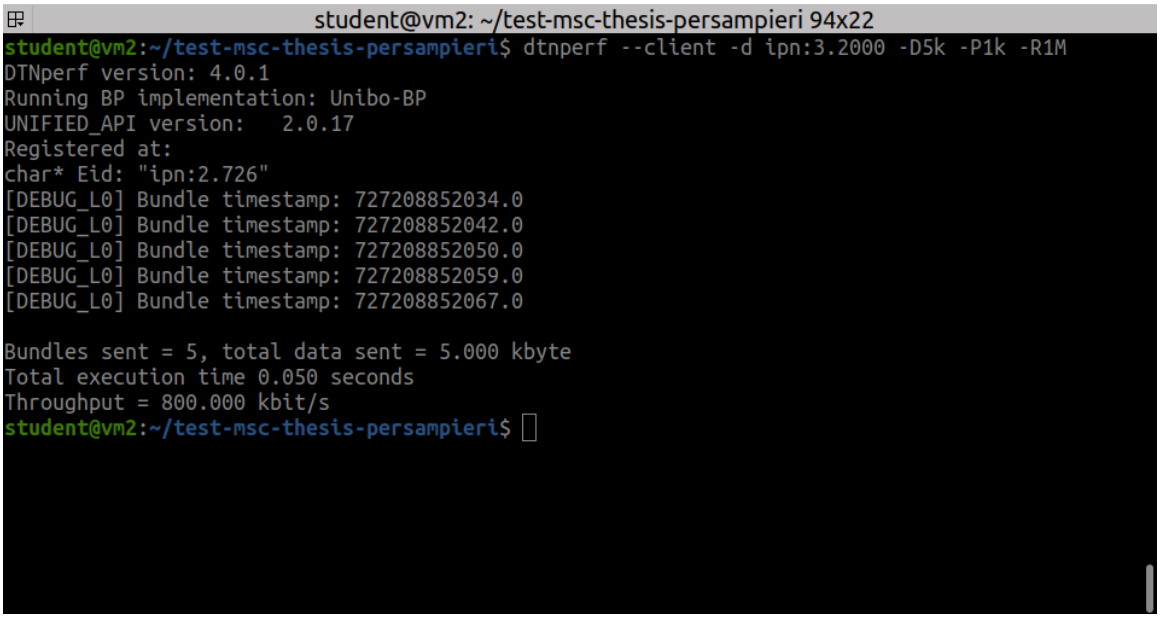


Figure 6.7 - DTNperf client executed on the Unibo-BP node. Five bundles, generated by "ipn:2.726", with a payload of 1 kB each were sent to "ipn:3.2000" (the ION node running on vm3).

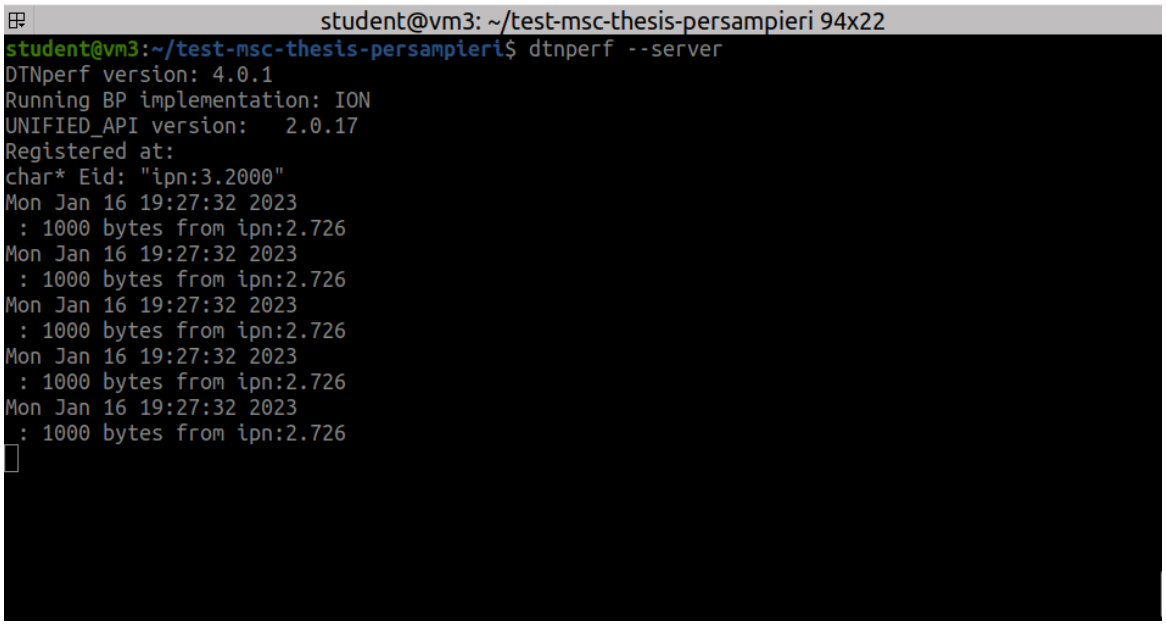


Figure 6.8 - DTNperf server ("ipn:3.2000") executed on the ION node. Five bundles, sent by "ipn:2.726", of 1 kB each were received.

No.	Time	Source	Destination	Protocol	Length	Info
195	322.148770	10.0.2.2	10.0.2.3	BPv7	1154	ipn:2.726 → ipn:3.2000
197	322.149446	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT
201	322.156879	10.0.2.2	10.0.2.3	BPv7	1154	ipn:2.726 → ipn:3.2000
203	322.157341	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT
207	322.165404	10.0.2.2	10.0.2.3	BPv7	1154	ipn:2.726 → ipn:3.2000
209	322.166149	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT
213	322.174390	10.0.2.2	10.0.2.3	BPv7	1154	ipn:2.726 → ipn:3.2000
215	322.175925	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT
219	322.182888	10.0.2.2	10.0.2.3	BPv7	1154	ipn:2.726 → ipn:3.2000
221	322.184418	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT

Figure 6.9 - Wireshark capture of the five bundles sent from "ipn:2.726" (Unibo-BP, DTNperf client) to "ipn:3.2000" (ION, DTNperf server).

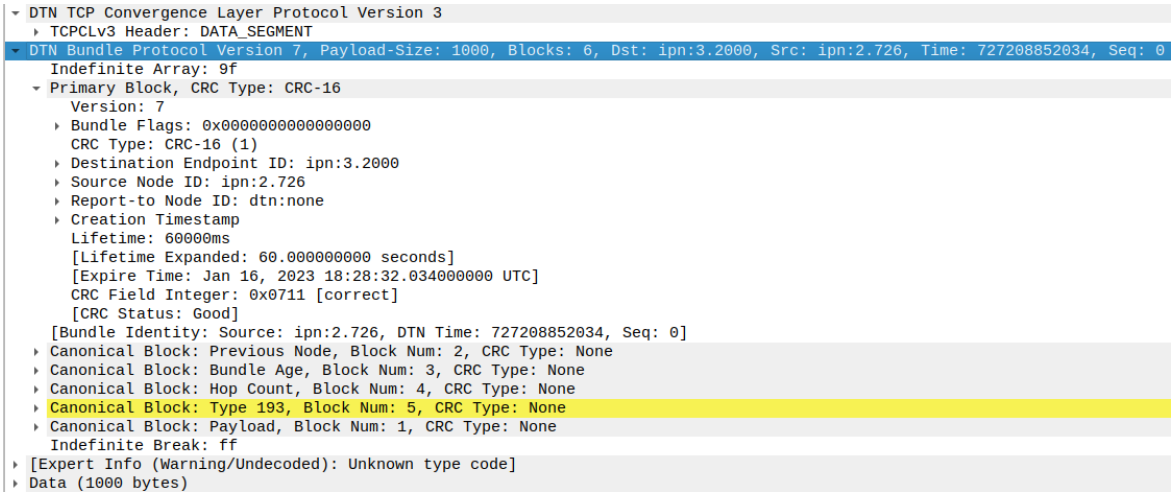


Figure 6.10 - Dissection of the first bundle sent by the DTNperf client ("ipn:2.726"). The Primary Block is expanded.

6.3.3 Test results

Since all bundles sent, 5 to DTNME and 5 to ION, arrived at destination node and were successfully delivered to the server, we can conclude that Unibo-BP is able to correctly format bundles as required by RFC 9171. This result is also confirmed by the fact that Wireshark is able to correctly dissect the bundles, in fact no errors are reported in Wireshark figures. The very same tests also prove the correctness and interoperability of the TCPCL version 3 implementation developed during this thesis.

6.4 UNIBO-BP AS A DESTINATION NODE

In this section are shown the tests carried out with Unibo-BP as destination node. Therefore, the DTNperf server is run on vm2 while the DTNperf client on vm1 or vm3 depending on whether the source node is, respectively, DTNME or ION. The protocols stack is shown in the figure below.

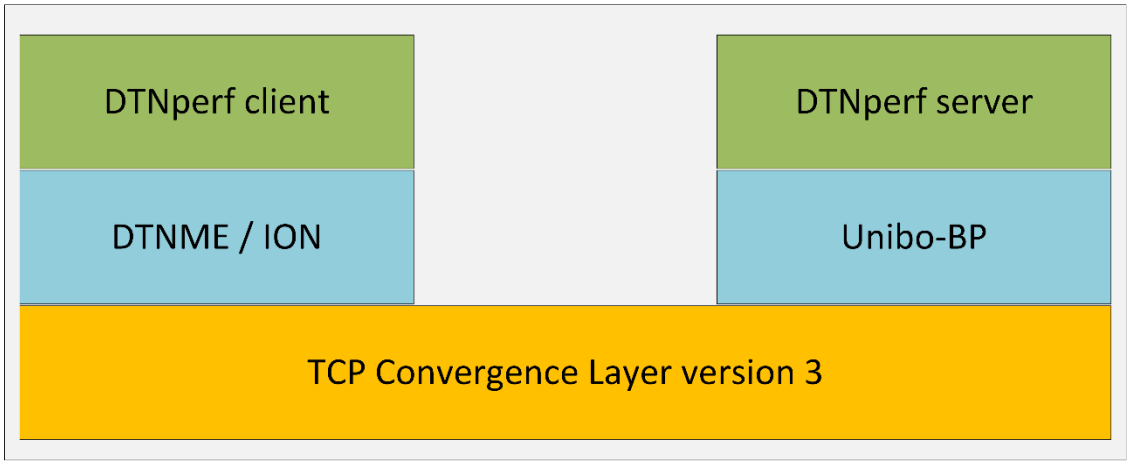


Figure 6.11 - Protocol stack when Unibo-BP runs as destination node.

6.4.1 DTNME (source) and Unibo-BP (destination)

```
student@vm1: ~/test-msc-thesis-persampieri 94x22
student@vm1:~/test-msc-thesis-persampieri$ dtnperf --client -d ipn:2.2000 -D5k -P1k -R1M
DTNperf version: 4.0.1
Running BP implementation: DTN2
UNIFIED_API version:   2.0.17
Registered at:
char* Eid: "dtn://vm1.dtn/dtnperf:/src_580"
[DEBUG_L0] Bundle timestamp: 727208681343.10
[DEBUG_L0] Bundle timestamp: 727208681354.11
[DEBUG_L0] Bundle timestamp: 727208681361.12
[DEBUG_L0] Bundle timestamp: 727208681370.13
[DEBUG_L0] Bundle timestamp: 727208681378.14

Bundles sent = 5, total data sent = 5.000 kbyte
Total execution time 0.054 seconds
Throughput = 740.741 kbit/s
student@vm1:~/test-msc-thesis-persampieri$
```

Figure 6.12 - DTNperf client executed on the DTNME node. Five bundles, generated by "dtn://vm1.dtn/dtnperf:/src_580", with a payload of 1 kB each were sent to "ipn:2.2000" (the Unibo-BP node running on vm2).

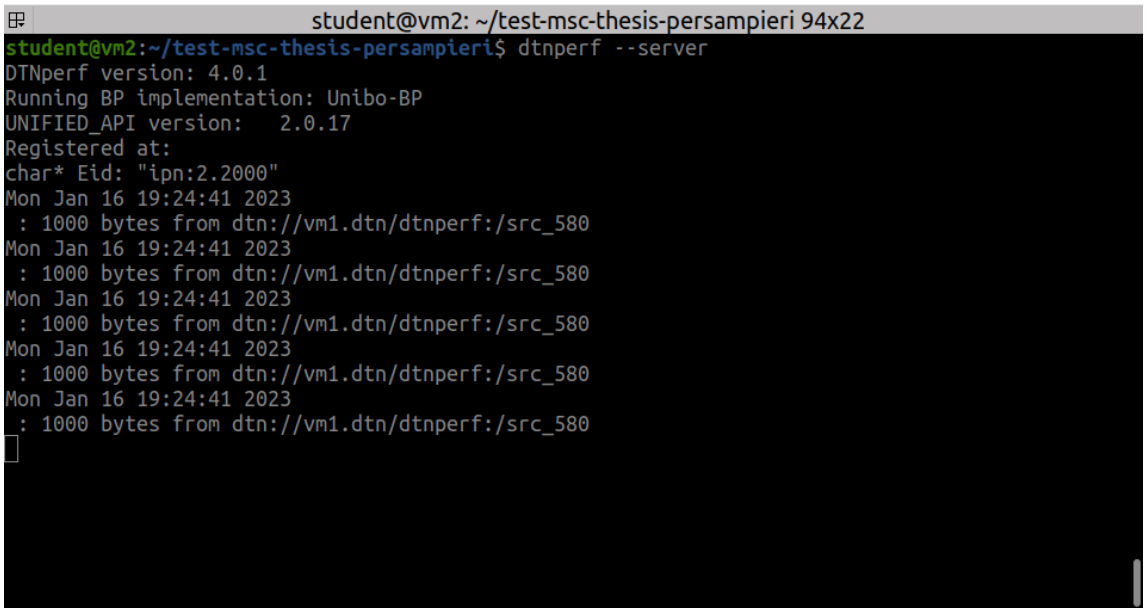


Figure 6.13 - DTNperf server ("ipn:2.2000") executed on the Unibo-BP node. Five bundles, sent by "dtn://vm1.dtn/dtnperf:/src_580", of 1 kB each were received.

No.	Time	Source	Destination	Protocol	Length	Info
175	152.313072	10.0.1.1	10.0.1.2	BPv7	1153	dtn://vm1.dtn/dtnperf:/src_580 → ipn:2.2000
177	152.313566	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT
179	152.317594	10.0.1.1	10.0.1.2	BPv7	1153	dtn://vm1.dtn/dtnperf:/src_580 → ipn:2.2000
180	152.317707	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT
182	152.325284	10.0.1.1	10.0.1.2	BPv7	1153	dtn://vm1.dtn/dtnperf:/src_580 → ipn:2.2000
183	152.325392	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT
185	152.333736	10.0.1.1	10.0.1.2	BPv7	1153	dtn://vm1.dtn/dtnperf:/src_580 → ipn:2.2000
186	152.333972	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT
188	152.342098	10.0.1.1	10.0.1.2	BPv7	1153	dtn://vm1.dtn/dtnperf:/src_580 → ipn:2.2000
189	152.342326	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT

Figure 6.14 - Wireshark capture of the five bundles sent from "dtn://vm1.dtn/dtnperf:/src_580" (DTNME, DTNperf client) to "ipn:2.2000" (Unibo-BP, DTNperf server).

```
▸ DTN TCP Convergence Layer Protocol Version 3
▾ DTN Bundle Protocol Version 7, Payload-Size: 1000, Blocks: 6, Dst: ipn:1.2000, Src: ipn:2.611, Time: 727208578701, Seq: 0
    Indefinite Array: 9f
  ▾ Primary Block, CRC Type: CRC-16
      Version: 7
    ▸ Bundle Flags: 0x0000000000000000
      CRC Type: CRC-16 (1)
    ▸ Destination Endpoint ID: ipn:1.2000
    ▸ Source Node ID: ipn:2.611
    ▸ Report-to Node ID: dtn:none
    ▸ Creation Timestamp
      Lifetime: 60000ms
      [Lifetime Expanded: 60.000000000 seconds]
      [Expire Time: Jan 16, 2023 18:23:58.701000000 UTC]
      CRC Field Integer: 0x2b7f [correct]
      [CRC Status: Good]
    [Bundle Identity: Source: ipn:2.611, DTN Time: 727208578701, Seq: 0]
  ▸ Canonical Block: Previous Node, Block Num: 2, CRC Type: None
  ▸ Canonical Block: Bundle Age, Block Num: 3, CRC Type: None
  ▸ Canonical Block: Hop Count, Block Num: 4, CRC Type: None
  ▸ Canonical Block: Type 193, Block Num: 5, CRC Type: None
  ▸ Canonical Block: Payload, Block Num: 1, CRC Type: None
    Indefinite Break: ff
▸ [Expert Info (Warning/Undecoded): Unknown type code]
▸ Data (1000 bytes)
```

Figure 6.15 - Dissection of the first bundle sent by the DTNperf client ("dtn://vm1.dtn/dtnperf:/src_580"). The Primary Block is expanded.

6.4.2 ION (source) and Unibo-BP (destination)

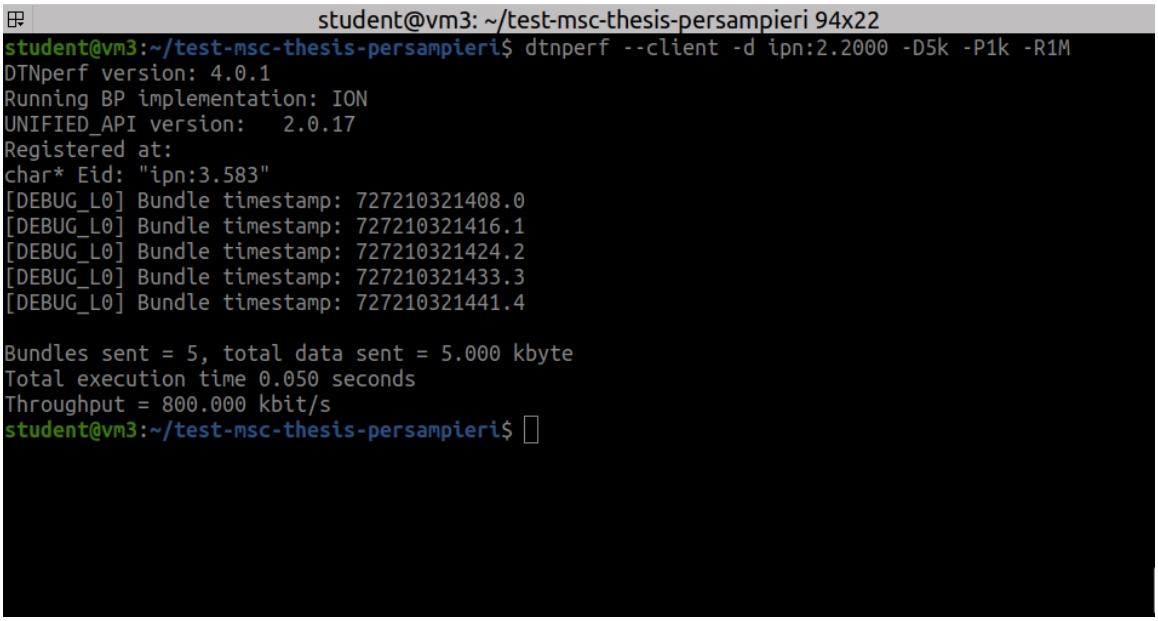


Figure 6.16 – DTNperf client executed on the ION node. Five bundles, generated by "ipn:3.583", with a payload of 1 kB each were sent to "ipn:2.2000" (the Unibo-BP node running on vm2).

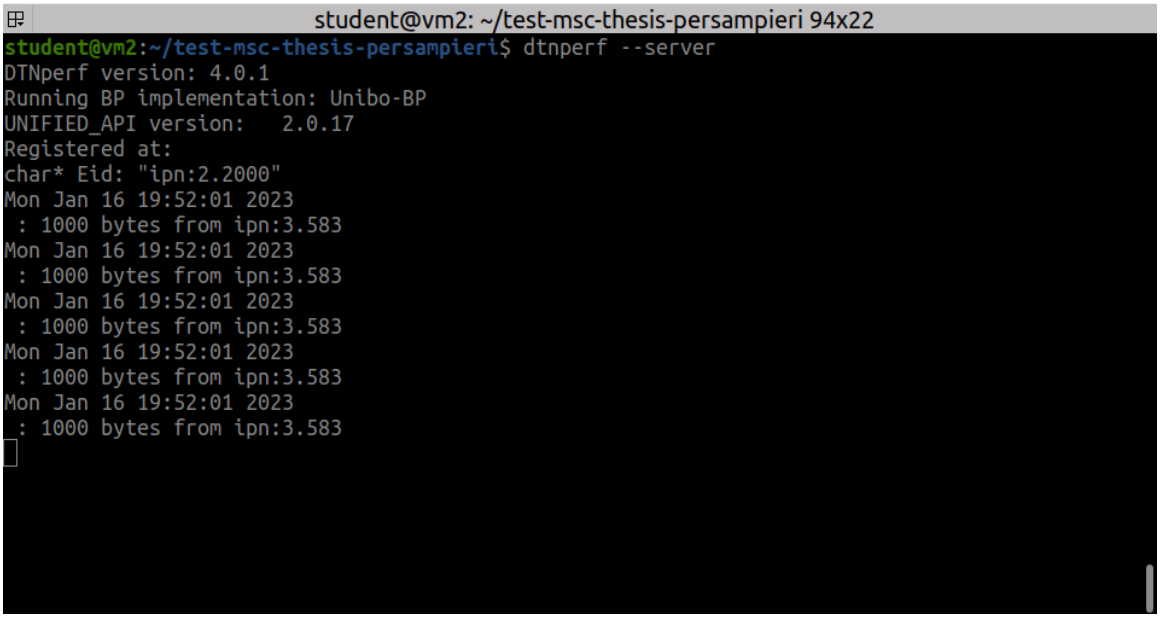


Figure 6.17 - DTNperf server ("ipn:2.2000") executed on the Unibo-BP node. Five bundles, sent by "ipn:3.583", of 1 kB each were received.

No.	Time	Source	Destination	Protocol	Length	Info
1005	1791.506939	10.0.2.3	10.0.2.2	BPv7	1145	ipn:3.583 → ipn:2.2000
1007	1791.507081	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT
1011	1791.514515	10.0.2.3	10.0.2.2	BPv7	1145	ipn:3.583 → ipn:2.2000
1013	1791.514611	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT
1017	1791.522672	10.0.2.3	10.0.2.2	BPv7	1145	ipn:3.583 → ipn:2.2000
1019	1791.522770	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT
1021	1791.524020	10.0.2.3	10.0.2.2	TCPCL	67	KEEPALIVE
1024	1791.539948	10.0.2.3	10.0.2.2	BPv7	1514	ipn:3.583 → ipn:2.2000
1026	1791.540295	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT
1027	1791.540943	10.0.2.3	10.0.2.2	BPv7	779	ipn:3.583 → ipn:2.2000
1029	1791.541654	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT

Figure 6.18 - Wireshark capture of the five bundles sent from "ipn:3.583" (ION, DTNperf client) to "ipn:2.2000" (Unibo-BP, DTNperf server).

```
▾ DTN TCP Convergence Layer Protocol Version 3
  ▸ TCPCLv3 Header: DATA_SEGMENT
▾ DTN Bundle Protocol Version 7, Payload-Size: 1000, Blocks: 5, Dst: ipn:2.2000, Src: ipn:3.583, Time: 727210321408, Seq: 0
    Indefinite Array: 9f
  ▾ Primary Block, CRC Type: CRC-16
      Version: 7
    ▸ Bundle Flags: 0x0000000000000040, Status time requested in reports
      CRC Type: CRC-16 (1)
    ▸ Destination Endpoint ID: ipn:2.2000
    ▸ Source Node ID: ipn:3.583
    ▸ Report-to Node ID: dtn:none
    ▸ Creation Timestamp
      Lifetime: 60000ms
      [Lifetime Expanded: 60.000000000 seconds]
      [Expire Time: Jan 16, 2023 18:53:01.408000000 UTC]
      CRC Field Integer: 0x93e4 [correct]
      [CRC Status: Good]
    [Bundle Identity: Source: ipn:3.583, DTN Time: 727210321408, Seq: 0]
  ▸ Canonical Block: Previous Node, Block Num: 2, CRC Type: None
  ▸ Canonical Block: Type 193, Block Num: 3, CRC Type: None
  ▸ Canonical Block: Bundle Age, Block Num: 4, CRC Type: None
  ▸ Canonical Block: Payload, Block Num: 1, CRC Type: None
    Indefinite Break: ff
▸ [Expert Info (Warning/Undecoded): Unknown type code]
▸ Data (1000 bytes)
```

Figure 6.19 - Dissection of the first bundle sent by the DTNperf client ("ipn:3.583"). The Primary Block is expanded.

6.4.3 Test results

All bundles sent, 5 from ION and 5 from DTNME, arrived at their destination and were correctly delivered, therefore we can conclude that Unibo-BP can correctly decode bundles formatted as required by RFC 9171. We have also verified that our TCPCL version 3 can correctly receive bundles sent by peers.

6.5 UNIBO-BP AS A ROUTER NODE

In this section are shown the tests carried out with Unibo-BP in an intermediate position, i.e., acting as a DTN router. There are no applications running above Unibo-BP, while DTNperf client and server are executed on the DTNME and ION nodes, alternatively, to carry out a test in both directions. Each bundle sent by DTNME is first forwarded to Unibo-BP and then it is forwarded (by Unibo-BP) to ION; then vice versa.

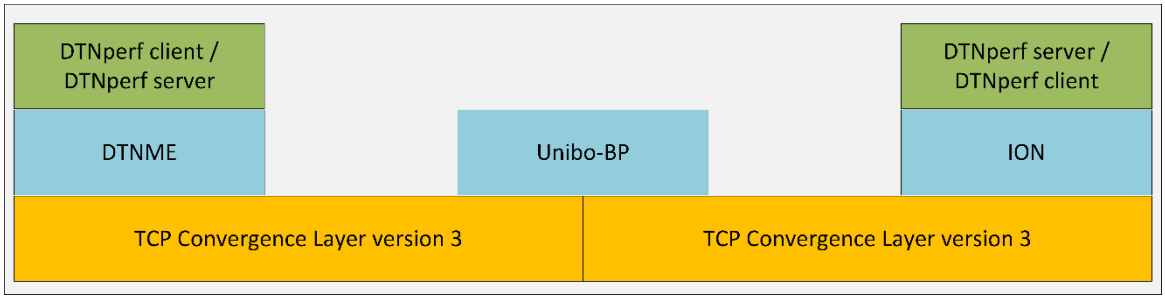


Figure 6.20 - Protocol stack when Unibo-BP runs as router node.

6.5.1 DTNME (source), Unibo-BP (router), ION (destination)

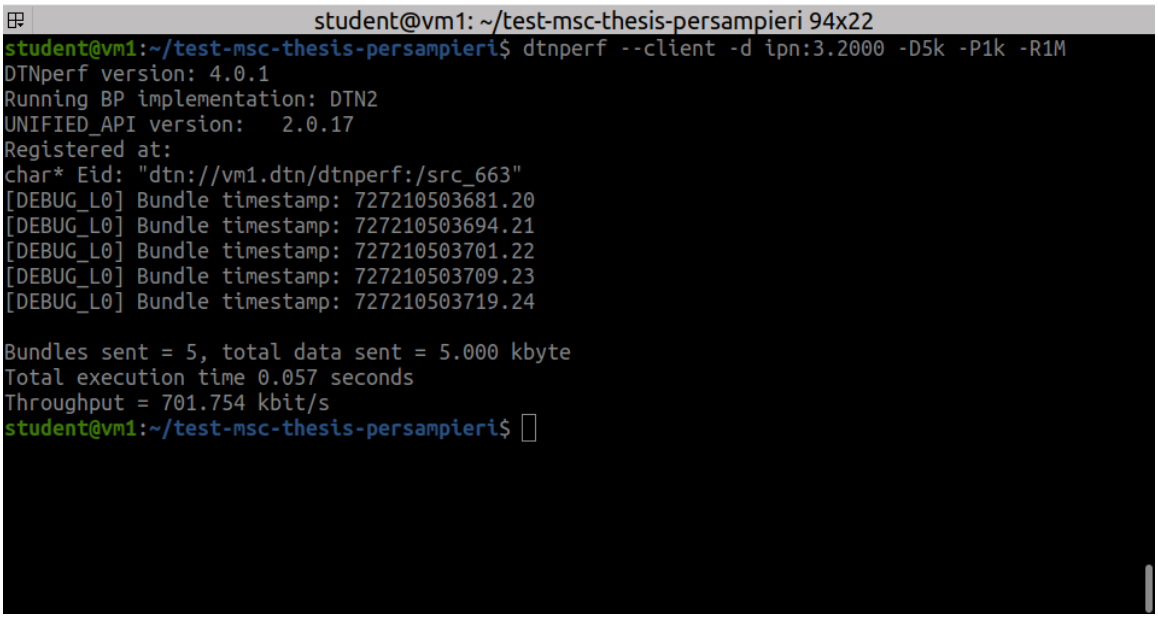


Figure 6.21 - DTNperf client executed on the DTNME node. Five bundles, generated by "dtn://vm1.dtn/dtnperf:/src_663", with a payload of 1 kB each were sent to "ipn:3.2000" (the ION node running on vm3).

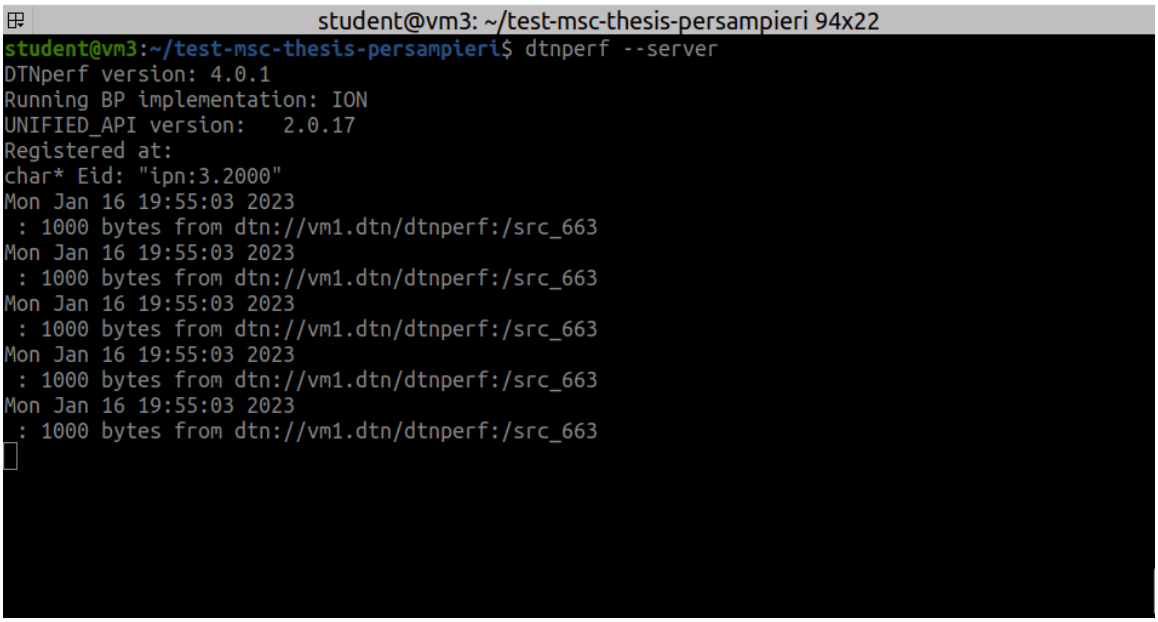


Figure 6.22 - DTNperf server ("ipn:3.2000") executed on the ION node. Five bundles, sent by "dtn://vm1.dtn/dtnperf:/src_663", of 1 kB each were received.

No.	Time	Source	Destination	Protocol	Length	Info
1537	1974.648240	10.0.1.1	10.0.1.2	BPv7	1153	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1539	1974.648545	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT
1541	1974.657531	10.0.1.1	10.0.1.2	BPv7	1153	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1543	1974.657771	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT
1545	1974.664506	10.0.1.1	10.0.1.2	BPv7	1153	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1547	1974.664796	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT
1549	1974.673513	10.0.1.1	10.0.1.2	BPv7	1153	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1550	1974.673716	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT
1552	1974.682991	10.0.1.1	10.0.1.2	BPv7	1154	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1553	1974.683198	10.0.1.2	10.0.1.1	TCPCL	69	ACK_SEGMENT

Figure 6.23 - Wireshark capture of the five bundles sent from "dtn://vm1.dtn/dtnperf:/src_663" (DTNME, DTNperf client) to "ipn:3.2000" (ION, DTNperf server). As can be seen from the destination IP address the bundles were forwarded to the intermediate router (Unibo-BP).

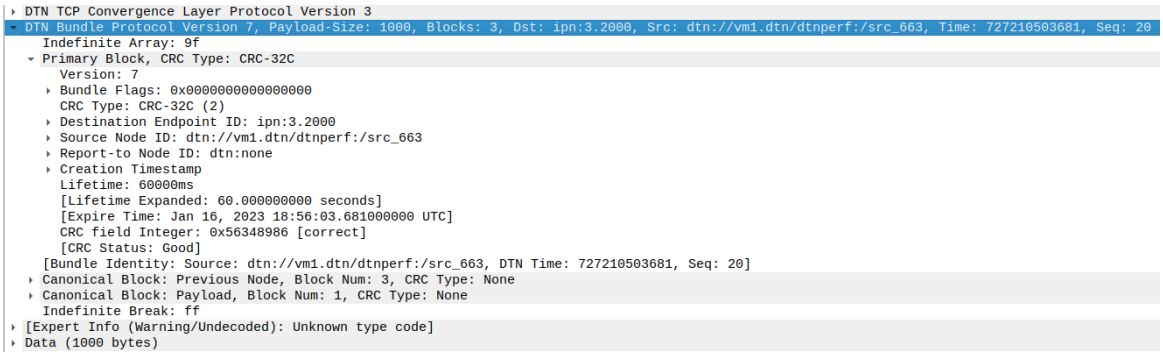


Figure 6.24 - Dissection of the first bundle sent by the DTNperf client ("dtn://vm1.dtn/dtnperf:/src_663") captured on the DTNME / Unibo-BP segment. The Primary Block is expanded.

No.	Time	Source	Destination	Protocol	Length	Info
1155	1973.783342	10.0.2.2	10.0.2.3	BPv7	1150	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1157	1973.784103	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT
1161	1973.792302	10.0.2.2	10.0.2.3	BPv7	1150	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1163	1973.792744	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT
1167	1973.799232	10.0.2.2	10.0.2.3	BPv7	1150	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1169	1973.799714	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT
1173	1973.808300	10.0.2.2	10.0.2.3	BPv7	1150	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1175	1973.809011	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT
1179	1973.817795	10.0.2.2	10.0.2.3	BPv7	1151	dtn://vm1.dtn/dtnperf:/src_663 → ipn:3.2000
1181	1973.818305	10.0.2.3	10.0.2.2	TCPCL	69	ACK_SEGMENT

Figure 6.25 - Wireshark capture of the five bundles sent from "dtn://vm1.dtn/dtnperf:/src_663" (DTNME, DTNperf client) to "ipn:3.2000" (ION, DTNperf server). As can be seen from the source IP address the bundles were forwarded by the intermediate router (Unibo-BP).

```
▾ DTN TCP Convergence Layer Protocol Version 3
  ▸ TCPCLv3 Header: DATA_SEGMENT
▾ DTN Bundle Protocol Version 7, Payload-Size: 1000, Blocks: 3, Dst: ipn:3.2000, Src: dtn://vm1.dtn/dtnperf:/src_663, Time: 727210503681, Seq: 20
    Indefinite Array: 9f
  ▾ Primary Block, CRC Type: CRC-32C
      Version: 7
    ▸ Bundle Flags: 0x0000000000000000
      CRC Type: CRC-32C (2)
    ▸ Destination Endpoint ID: ipn:3.2000
    ▸ Source Node ID: dtn://vm1.dtn/dtnperf:/src_663
    ▸ Report-to Node ID: dtn:none
    ▸ Creation Timestamp
      Lifetime: 60000ms
      [Lifetime Expanded: 60.000000000 seconds]
      [Expire Time: Jan 16, 2023 18:56:03.681000000 UTC]
      CRC field Integer: 0x56348986 [correct]
      [CRC Status: Good]
    [Bundle Identity: Source: dtn://vm1.dtn/dtnperf:/src_663, DTN Time: 727210503681, Seq: 20]
  ▸ Canonical Block: Previous Node, Block Num: 3, CRC Type: None
  ▸ Canonical Block: Payload, Block Num: 1, CRC Type: None
    Indefinite Break: ff
▸ [Expert Info (Warning/Undecoded): Unknown type code]
▸ Data (1000 bytes)
```

Figure 6.26 - Dissection of the first bundle sent by the DTNperf client ("dtn://vm1.dtn/dtnperf:/src_663") captured on the Unibo-BP / ION segment. The Primary Block is expanded.

6.5.2 ION (source), Unibo-BP (router), DTNME (destination)

```
student@vm3: ~/test-msc-thesis-persampieri 94x22
student@vm3:~/test-msc-thesis-persampieri$ dtnperf --client -d ipn:1.2000 -D5k -P1k -R1M
DTNperf version: 4.0.1
Running BP implementation: ION
UNIFIED_API version:   2.0.17
Registered at:
char* Eid: "ipn:3.664"
[DEBUG_L0] Bundle timestamp: 727210555856.10
[DEBUG_L0] Bundle timestamp: 727210555961.11
[DEBUG_L0] Bundle timestamp: 727210555964.12
[DEBUG_L0] Bundle timestamp: 727210555967.13
[DEBUG_L0] Bundle timestamp: 727210555970.14

Bundles sent = 5, total data sent = 5.000 kbyte
Total execution time 0.117 seconds
Throughput = 341.880 kbit/s
student@vm3:~/test-msc-thesis-persampieri$
```

Figure 6.27 - DTNperf client executed on the ION node. Five bundles, generated by "ipn:3.664", with a payload of 1 kB each were sent to "ipn:1.2000" (the DTNME node running on vm1).

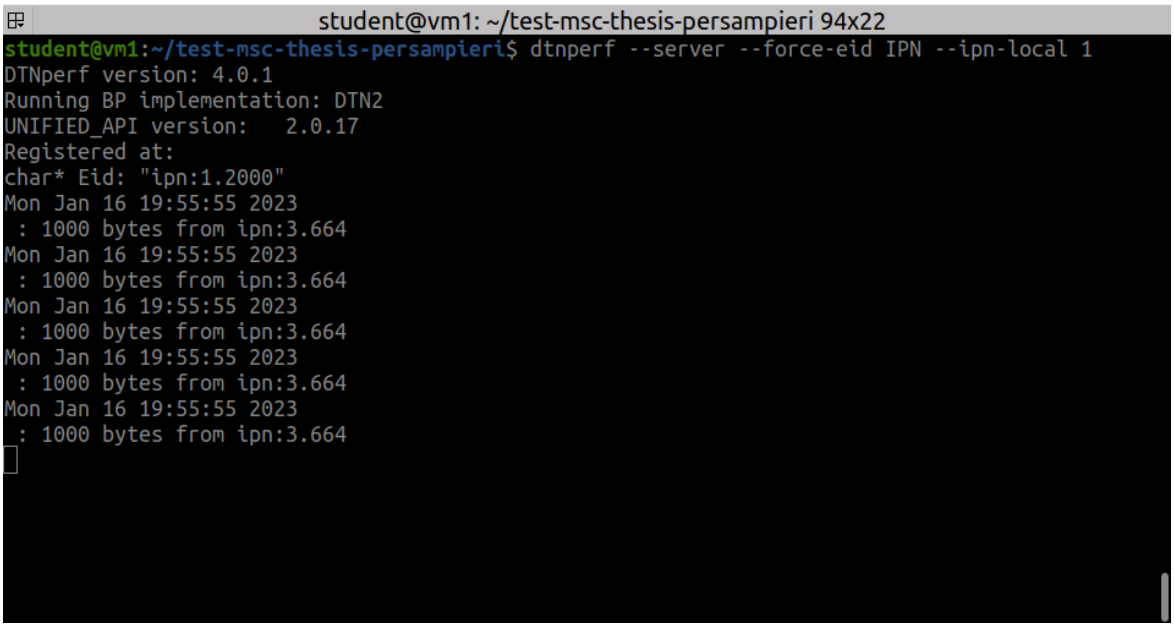


Figure 6.28 - DTNperf server ("ipn:1.2000") executed on the DTNME node. Five bundles, sent by "ipn:3.664", of 1 kB each were received.

No.	Time	Source	Destination	Protocol	Length	Info
1213	2026.056867	10.0.2.3	10.0.2.2	BPv7	1146	ipn:3.664 → ipn:1.2000
1215	2026.057235	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT
1219	2026.061113	10.0.2.3	10.0.2.2	BPv7	1145	ipn:3.664 → ipn:1.2000
1221	2026.061433	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT
1225	2026.064681	10.0.2.3	10.0.2.2	BPv7	1145	ipn:3.664 → ipn:1.2000
1227	2026.065097	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT
1230	2026.067449	10.0.2.3	10.0.2.2	BPv7	1514	ipn:3.664 → ipn:1.2000
1232	2026.067584	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT
1233	2026.067879	10.0.2.3	10.0.2.2	BPv7	779	ipn:3.664 → ipn:1.2000
1236	2026.109699	10.0.2.2	10.0.2.3	TCPCL	69	ACK_SEGMENT

Figure 6.29 - Wireshark capture of the five bundles sent from "ipn:3.664" (ION, DTNperf client) to "ipn:1.2000" (DTNME, DTNperf server). As can be seen from the destination IP address the bundles were forwarded to the intermediate router (Unibo-BP).

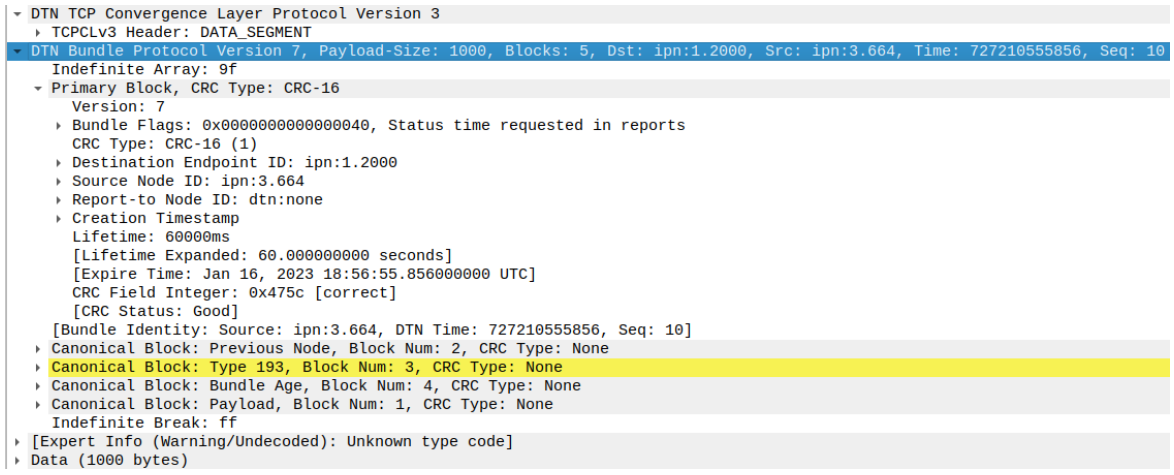
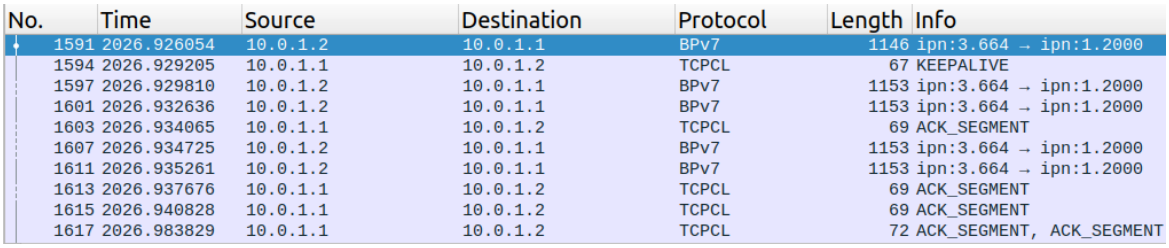
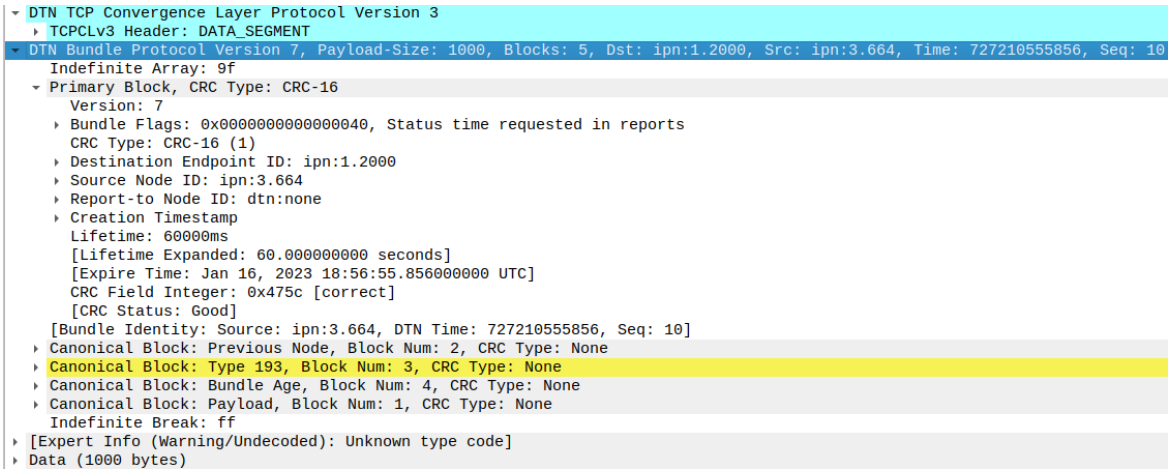


Figure 6.30 - Dissection of the first bundle sent by the DTNperf client ("ipn:3.664") captured on the ION / Unibo-BP segment. The Primary Block is expanded.



No.	Time	Source	Destination	Protocol	Length	Info
1591	2026.926054	10.0.1.2	10.0.1.1	BPv7	1146	ipn:3.664 → ipn:1.2000
1594	2026.929205	10.0.1.1	10.0.1.2	TCPCL	67	KEEPALIVE
1597	2026.929810	10.0.1.2	10.0.1.1	BPv7	1153	ipn:3.664 → ipn:1.2000
1601	2026.932636	10.0.1.2	10.0.1.1	BPv7	1153	ipn:3.664 → ipn:1.2000
1603	2026.934065	10.0.1.1	10.0.1.2	TCPCL	69	ACK_SEGMENT
1607	2026.934725	10.0.1.2	10.0.1.1	BPv7	1153	ipn:3.664 → ipn:1.2000
1611	2026.935261	10.0.1.2	10.0.1.1	BPv7	1153	ipn:3.664 → ipn:1.2000
1613	2026.937676	10.0.1.1	10.0.1.2	TCPCL	69	ACK_SEGMENT
1615	2026.940828	10.0.1.1	10.0.1.2	TCPCL	69	ACK_SEGMENT
1617	2026.983829	10.0.1.1	10.0.1.2	TCPCL	72	ACK_SEGMENT, ACK_SEGMENT

Figure 6.31 - Wireshark capture of the five bundles sent from "ipn:3.664" (ION, DTNperf client) to "ipn:1.2000" (DTNME, DTNperf server). As can be seen from the source IP address the bundles were forwarded by the intermediate router (Unibo-BP).



```
▾ DTN TCP Convergence Layer Protocol Version 3
  ▸ TCPCLv3 Header: DATA_SEGMENT
▾ DTN Bundle Protocol Version 7, Payload-Size: 1000, Blocks: 5, Dst: ipn:1.2000, Src: ipn:3.664, Time: 727210555856, Seq: 10
    Indefinite Array: 9f
  ▾ Primary Block, CRC Type: CRC-16
      Version: 7
    ▸ Bundle Flags: 0x0000000000000040, Status time requested in reports
      CRC Type: CRC-16 (1)
    ▸ Destination Endpoint ID: ipn:1.2000
    ▸ Source Node ID: ipn:3.664
    ▸ Report-to Node ID: dtn:none
    ▸ Creation Timestamp
      Lifetime: 60000ms
      [Lifetime Expanded: 60.000000000 seconds]
      [Expire Time: Jan 16, 2023 18:56:55.856000000 UTC]
      CRC Field Integer: 0x475c [correct]
      [CRC Status: Good]
    [Bundle Identity: Source: ipn:3.664, DTN Time: 727210555856, Seq: 10]
  ▸ Canonical Block: Previous Node, Block Num: 2, CRC Type: None
  ▸ Canonical Block: Type 193, Block Num: 3, CRC Type: None
  ▸ Canonical Block: Bundle Age, Block Num: 4, CRC Type: None
  ▸ Canonical Block: Payload, Block Num: 1, CRC Type: None
    Indefinite Break: ff
▸ [Expert Info (Warning/Undecoded): Unknown type code]
▸ Data (1000 bytes)
```

Figure 6.32 - Dissection of the first bundle sent by the DTNperf client ("ipn:3.664") captured on the Unibo-BP / DTNME segment. The Primary Block is expanded.

6.5.3 Test results

All bundles sent, 5 from DTNME to ION and 5 from ION to DTNME, arrived at their destination, forwarded by the intermediate Unibo-BP node before reaching their destination. Therefore, we can conclude that Unibo-BP is compliant with RFC 9171 even when operating as a router.

In particular, it is worth noting to observe (from Figure 6.24 and Figure 6.26, or from Figure 6.30 and Figure 6.32) that the Primary Block remained unchanged after traversing the Unibo-BP node as expected, since RFC 9171 states that the Primary Block of each bundle shall be immutable.

7 CONCLUSIONS

The purpose of this thesis was to design and create a novel implementation of the Bundle Protocol version 7, recently standardized by RFC 9171. This new implementation, named Unibo-BP, is now complete and has been released as Free Software on a public Git repository.

Unibo-BP is fully compliant with RFC 9171 and interoperability with other BP implementations has been demonstrated by means of a wide variety of tests with the two reference implementations ION (designed and maintained by NASA-JPL) and DTNME (maintained by NASA-MSFC). The tests shown in the thesis confirm that a Unibo-BP node can correctly operates as a source of bundles, as a destination, or as a router node. These tests also demonstrated the correctness of the implementation of the TCP Convergence Layer version 3 (RFC 7242) developed during this thesis.

Among the novel characteristics of Unibo-BP, we cite the possibility to natively manage a Unibo-BP node remotely by sending commands via the Bundle Protocol itself. With Unibo-BP, it is therefore possible to manage remote nodes even if they are deployed in challenged networks, i.e., subjected to long delays and link disruption. Another peculiarity is that Unibo-BP accepts contacts and ranges where the EID of dtn nodes follows schemes different from the “ipn” one, which may help in extending the scope of CGR and LTP outside of the space networks, where the use of the “ipn” scheme is dictated by the CCSDS standard.

Unibo-BP is at the center of a broader project, called Unibo-DTN, which also includes Unibo-CGR and Unibo-LTP, with which Unibo-BP is fully compatible thanks to the implementation of *ad hoc* interfaces developed during this thesis. Another interface was written to extend the support of the Unified API library to Unibo-BP, which makes compatible with Unibo-BP all applications of the DTNsuite, such as DTNperf, DTNbox, DTNfog, DTNproxy and DTNchat, thus making Unibo-BP the central pillar of a whole DTN ecosystem encompassing DTN protocols, routing algorithms and applications.

Our expectation is that Unibo-BP and Unibo-DTN can result a useful tool for all DTN researchers, thus contributing to a wider adoption of the DTN architecture. A second, more ambitious hope is that it may be evaluated by space agencies for interoperability tests or in experimental deployments.

BIBLIOGRAPHY

[Alessi_2019] Alessi, Nicola (2019) "*Hierarchical Inter-Regional Routing Algorithm for Interplanetary Networks*". [Laurea magistrale], Università di Bologna, Corso di Studio in Ingegneria informatica [LM-DM270] <https://amslaurea.unibo.it/17468/>.

[Araniti_2015] G. Araniti, N. Bezirgiannidis, E. Birrane, I. Bisio, S. Burleigh, C. Caini, M. Feldmann, M. Marchese, J. Segui, K. Suzuki. "*Contact Graph Routing in DTN Space Networks: Overview, Enhancements and Performance*", IEEE Commun. Mag., Vol.53, No.3, pp.38-46, March 2015, DOI: 10.1109/MCOM.2015.7060480.

[Bash] Web site: https://www.gnu.org/software/bash/.

[Bertolazzi_2019] M. Bertolazzi, C. Caini and N. Castellazzi, "*DTNbox: a DTN Application for Peer-to-Peer Directory Synchronization*", 2019 Wireless Days (WD), 2019, pp. 1-4, doi: 10.1109/WD.2019.8734214.

[Birrane_2021] E. J. Birrane, C. Caini, G. M. De Cola, F. Marchetti, L. Mazzuca, L. Persampieri, "*Opportunities and limits of moderate source routing in delay-/disruption-tolerant networking space networks*", Int J Satell Commun Network. Early publication, vol.40, no.6, pp. 428- 444, Aug.2021. doi:10.1002/sat.1421.

[Bisacchi_2021] Bisacchi, Andrea (2021) "*Progetto di una implementazione veloce del Licklider Transmission Protocol per link ottici*". [Laurea magistrale], Università di Bologna, Corso di Studio in Ingegneria informatica [LM-DM270] <https://amslaurea.unibo.it/22383/>.

[Bisacchi_2022] A. Bisacchi, C. Caini and T. de Cola, "*Multicolor Licklider Transmission Protocol: An LTP Version for Future Interplanetary Links*", in IEEE Transactions on Aerospace and Electronic Systems, vol. 58, no. 5, pp. 3859-3869, Oct. 2022, doi: 10.1109/TAES.2022.3176847. (Open Source).

[Burleigh_2007] S. Burleigh, "*Interplanetary Overlay Network: An Implementation of the DTN Bundle Protocol*", 2007 4th IEEE Consumer Communications and Networking Conference, 2007, pp. 222-226, doi: 10.1109/CCNC.2007.51.

[Burleigh_2013] Burleigh, Scott, 2013, "*Delay-tolerant security key agreement (DTKA)*", https://hdl.handle.net/2014/44334, Root, V1.

[Burleigh_2016] S. Burleigh, C. Caini, J. J. Messina and M. Rodolfi, "*Toward a unified routing framework for delay-tolerant networking*", 2016 IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE), Aachen, Germany, 2016, pp. 82-86, doi: 10.1109/WiSEE.2016.7877309.

[Caini_2011] C. Caini, H. Cruickshank, S. Farrell and M. Marchese, "*Delay- and Disruption-Tolerant Networking (DTN): An Alternative Solution for Future Satellite*

Networking Applications", in Proceedings of the IEEE, vol. 99, no. 11, pp. 1980-1997, Nov. 2011, doi: 10.1109/JPROC.2011.2158378.

[Caini_2013] C. Caini, A. d'Amico and M. Rodolfi, "*DTNperf_3: A further enhanced tool for Delay-/Disruption- Tolerant Networking Performance evaluation*", 2013 IEEE Global Communications Conference (GLOBECOM), 2013, pp. 3009-3015, doi: 10.1109/GLOCOM.2013.6831533.

[Caini_2021] C. Caini, G. M. De Cola, L. Persampieri, "*Schedule-Aware Bundle Routing: Analysis and Enhancements*", International Journal of Satellite Communications and Networking, vol. 39, no.3, pp. 237-243, May/June 2021. DOI: 10.1002/sat.1384.

[CCSDS_BPV6] CCSDS 734.2-B-1, "*CCSDS Bundle Protocol Specification*", recommended standard, Blue Book, September 2015, https://public.ccsds.org/Pubs/734x2b1.pdf.

[CCSDS_LTP] CCSDS 734.1-B-1, "*Licklider Transmission Protocol (LTP) for CCSDS*", recommended standard, Blue Book, May 2015, https://public.ccsds.org/Pubs/734x1b1.pdf.

[CCSDS_SABR] CCSDS 734.3-B-1, "*Schedule-Aware Bundle Routing*", recommended standard, Blue Book, July 2019, https://public.ccsds.org/Pubs/734x3b1.pdf.

[Cingolani_2022] A. Cingolani, "*Studio delle funzionalità IRF e DNAC di ION*", Tesi di Laurea in Ingegneria Informatica, Università di Bologna, 2022.

[CRC16] ITU-T, "*X.25: Interface between Data Terminal Equipment (DTE) and Data Circuit-terminating Equipment (DCE) for terminals operating in the packet mode and connected to public data networks by dedicated circuit*", p. 9, Section 2.2.7.4, ITU-T Recommendation X.25, October 1996, <https://www.itu.int/rec/T-REC-X.25-199610-I/>.

[Demmer_2007] M. Demmer, K. Fall, "*DTLSR: Delay Tolerant Routing for Developing Regions*", ACM SIGCOMM Workshop on Networked Systems in Developing Regions (NSDR), Kyoto, Japan, 2007.

[Draft_ECOS] S. Burleigh "*Bundle Protocol Extended Class of Service (ECOS)*", IETF Draft, May 2021, https://tools.ietf.org/html/draft-burleigh-dtn-ecos-00.

[DTN2] Web site: https://sourceforge.net/projects/dtn/.

[DTNME] Web site: https://github.com/nasa/DTNME.

[DTNsuite] Web site: https://gitlab.com/dtnsuite.

[Gori_2020] G. Gori, "*Inserimento dell'algoritmo di routing CGR-SABR in DTN2*", Tesi di Laurea in Ingegneria Informatica, Università di Bologna, 2020.

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

[Lindgren_2004] Lindgren, A., Doria, A., Schelén, O. (2004). "*Probabilistic Routing in Intermittently Connected Networks*". In: Dini, P., Lorenz, P., de Souza, J.N. (eds) Service Assurance with Partial and Intermittent Resources. SAPIR 2004. Lecture Notes in Computer Science, vol 3126. Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-540-27767-5_24.

[Persampieri_2020] L. Persampieri, "*Unibo-CGR: una nuova implementazione dell'algoritmo di routing CGR/SABR*", Tesi di Laurea in Ingegneria Informatica, Università di Bologna, 2020.

[RFC4838] Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K., Fall, K., and H. Weiss, "*Delay-Tolerant Networking Architecture*", RFC 4838, DOI 10.17487/RFC4838, April 2007, <https://www.rfc-editor.org/info/rfc4838>.

[RFC4960] Stewart, R., Ed., "*Stream Control Transmission Protocol*", RFC 4960, DOI 10.17487/RFC4960, September 2007, <https://www.rfc-editor.org/info/rfc4960>.

[RFC5050] Scott, K. and S. Burleigh, "*Bundle Protocol Specification*", RFC 5050, DOI 10.17487/RFC5050, November 2007, <https://www.rfc-editor.org/info/rfc5050>.

[RFC5325] Burleigh, S., Ramadas, M., and S. Farrell, "*Licklider Transmission Protocol - Motivation*", RFC 5325, DOI 10.17487/RFC5325, September 2008, <https://www.rfc-editor.org/info/rfc5325>.

[RFC5326] Ramadas, M., Burleigh, S., and S. Farrell, "*Licklider Transmission Protocol - Specification*", RFC 5326, DOI 10.17487/RFC5326, September 2008, <https://www.rfc-editor.org/info/rfc5326>.

[RFC6693] Lindgren, A., Doria, A., Davies, E., and S. Grasic, "*Probabilistic Routing Protocol for Intermittently Connected Networks*", RFC 6693, DOI 10.17487/RFC6693, August 2012, <https://www.rfc-editor.org/info/rfc6693>.

[RFC7242] Demmer, M., Ott, J., and S. Perreault, "*Delay-Tolerant Networking TCP Convergence-Layer Protocol*", RFC 7242, DOI 10.17487/RFC7242, June 2014, <https://www.rfc-editor.org/info/rfc7242>.

[RFC8949] Bormann, C. and P. Hoffman, "*Concise Binary Object Representation (CBOR)*", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <https://www.rfc-editor.org/info/rfc8949>.

[RFC9171] Burleigh, S., Fall, K., Birrane, E., and III, "*Bundle Protocol Version 7*", RFC 9171, DOI 10.17487/RFC9171, January 2022, <https://www.rfc-

editor.org/info/rfc9171>.

[RFC9172] Birrane, E., III, and K. McKeever, "*Bundle Protocol Security (BPSec)*", RFC 9172, DOI 10.17487/RFC9172, January 2022, <https://www.rfc-editor.org/info/rfc9172>.

[RFC9174] Sipos, B., Demmer, M., Ott, J., and S. Perreault, "*Delay-Tolerant Networking TCP Convergence-Layer Protocol Version 4*", RFC 9174, DOI 10.17487/RFC9174, January 2022, <https://www.rfc-editor.org/info/rfc9174>.

[Spyropoulos_2005] T. Spyropoulos, K Psounis, and C. S. Raghavendra, "*Spray and wait: An efficient routing scheme for intermittently connected mobile networks*", in Proc. of 2005 ACM SIGCOMM workshop on Delay-tolerant networking, WDTN'05, 2005, pp. 252-259.

[Unibo-BP] Web site: https://gitlab.com/unibo-dtn/unibo-bp.

[Unibo-CGR] Web site: https://gitlab.com/unibo-dtn/unibo-cgr.

[Unibo-DTN] Web site: https://gitlab.com/unibo-dtn.

[Unibo-LTP] Web site: https://gitlab.com/unibo-dtn/ltp.

[UnifiedAPI] A. Bisacchi, C. Caini and S. Lanzoni, "*Design and Implementation of a Bundle Protocol Unified API*" 2022 11th Advanced Satellite Multimedia Systems Conference and the 17th Signal Processing for Space Communications Workshop (ASMS/SPSC), 2022, pp. 1-6, doi: 10.1109/ASMS/SPSC55670.2022.9914734.

[Vahdat_2000] A. Vahdat and D. Becker, "*Epidemic Routing for Partially-Connected Ad Hoc Networks*", Duke Technical Report CS-2000-06, July 2000.

[Virtualbricks] P. Apollonio, C. Caini, M. Giusti, D. Lacamera, "*Virtualbricks for DTN satellite communications research and education*", in Proc. of PSATS 2014, Genoa, Italy, July 2014, pp. 1-14. DOI: 10.1007/978-3-319-47081-8_7.

[Wireshark] Web site: https://www.wireshark.org/.

APPENDIX A: NODES CONFIGURATION

The following are the configurations used to carry out interoperability tests. For DTNME and ION only the excerpts deemed most relevant are reported, while a complete configuration is shown for Unibo-BP.

DTNME

```
route local_eid_ipn ipn:1.0
route set type static
interface add tcp0 tcp
link add ltvm2 10.0.1.2 ALWAYSON tcp3
route add ipn:2.* ltvm2
route add ipn:3.* ltvm2
```

ION

The commands below are extracted from the **ionrc** configuration file interpreted by the **ionadmin** program.

```
a range +0 +36000 1 2 1
a range +0 +36000 2 3 1
a contact +1 +36000 1 2 125000
a contact +1 +36000 2 1 125000
a contact +1 +36000 3 2 125000
a contact +1 +36000 2 3 125000
```

The commands below are extracted from the **bprc** configuration file interpreted by the **bpadmin** program.

```
a scheme ipn 'ipnfw' 'ipnadminep'
a protocol tcp 1400 100 100000000
a induct tcp 0.0.0.0 tcpcli
a outduct tcp 10.0.2.2 ''
```

Below is the content of the **ipnrc** configuration file interpreted by the **ipnadmin** program.

```
a plan 2 tcp/10.0.2.2
```

UNIBO-BP

```
#!/bin/bash

REFERENCE_TIME=$(unibo-bp-utility --get-utc-time +0)

unibo-bp start --set-storage-size 50000000 --dtn-admin dtn://vm2.dtn/ --
ipn-admin ipn:2.0 --daemon

sleep 5



unibo-bp-admin region home --register-node ipn:1.0

unibo-bp-admin region home --register-node ipn:2.0

unibo-bp-admin region home --register-node ipn:3.0



unibo-bp-admin range add   --start-time +0  --end-time +36000 --sender
ipn:1.0 --receiver ipn:2.0 --owlt 1  --reference-time $REFERENCE_TIME

unibo-bp-admin range add   --start-time +0  --end-time +36000 --sender
ipn:2.0 --receiver ipn:1.0 --owlt 1  --reference-time $REFERENCE_TIME

unibo-bp-admin range add   --start-time +0  --end-time +36000 --sender
ipn:2.0 --receiver ipn:3.0 --owlt 1  --reference-time $REFERENCE_TIME

unibo-bp-admin range add   --start-time +0  --end-time +36000 --sender
ipn:3.0 --receiver ipn:2.0 --owlt 1  --reference-time $REFERENCE_TIME

unibo-bp-admin contact add --start-time +0  --end-time +36000 --sender
ipn:1.0 --receiver ipn:2.0 --xmit-rate 125000  --reference-time
$REFERENCE_TIME
```

```
unibo-bp-admin contact add --start-time +0  --end-time +36000 --sender
ipn:2.0 --receiver ipn:1.0 --xmit-rate 125000  --reference-time
$REFERENCE_TIME

unibo-bp-admin contact add --start-time +0  --end-time +36000 --sender
ipn:2.0 --receiver ipn:3.0 --xmit-rate 125000  --reference-time
$REFERENCE_TIME

unibo-bp-admin contact add --start-time +0  --end-time +36000 --sender
ipn:3.0 --receiver ipn:2.0 --xmit-rate 125000  --reference-time
$REFERENCE_TIME



unibo-bp-tcpcl --daemon

sleep 5

unibo-bp-admin tcpcl induct add

unibo-bp-admin tcpcl outduct add --peer ipn:1.0 --hostname 10.0.1.1

unibo-bp-admin tcpcl outduct add --peer ipn:3.0 --hostname 10.0.2.3
```