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: enhancements and extensions

Candidate:

Alberto Genovese

Supervisor:

Prof. **Carlo Caini**

Academic Year 2022/2023

ABSTRACT

The development of Unibo-BP has represented a big step forward in the field of *Delay-/Disruption-Tolerant Networking* (DTN) research at University of Bologna. The fact that Unibo-BP is a research-driven, fully compliant implementation of the *Bundle Protocol* (BP) version 7 (BPv7), highlights Unibo's commitment to staying at the forefront of DTN research. The modularity of Unibo-BP code allows researchers to add new features or modifications to existing components in a relatively easy way. Dealing with Unibo-BP, which is a big and complex project, a preliminary necessary step was to understand how Unibo-BP and its ecosystem (in particular *Unibo-LTP* and *Unified API*) work; to this end, learn how to debug in the best way all the project was crucial. Once achieved the necessary familiarity with the code, the first goal was to improve the *Unibo-LTP interface to Unibo-BP*; thanks to this improvement it is now possible to map the bundle QoS to the different services ("colors") offered by LTP. The second goal, was to add the *Metadata bundle extension* in Unibo-BP. This not only required to implement the extension, but also to modify its original format, conceived for BPv6, to make it compatible with BPv7. To allow *DTNsuite applications* to use the new Metadata extensions, it was also necessary to modify the *Unified API library* on which all DTNsuite applications are based (in particular, the Unified API interface to Unibo-BP had to be augmented). In conclusion, this thesis was the first to introduce modifications and improvements to Unibo-BP (and related software), but hopefully not the last; the hope is that the work done can be useful to future students and researchers.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 THE DTN ARCHITECTURE

The foundation of the Delay-/Disruption-Tolerant Networking (DTN) architecture stems from a generalization of InterPlanetary Networking (IPN) and is crafted to facilitate communication in demanding scenarios, particularly within networks characterized by prolonged delays and/or sporadic connectivity, called "**challenged networks**". These conditions pose challenges for conventional Internet protocols, hindering effective communication. In response to these challenges, the DTN architecture brings an innovative solution by introducing a distinct layer known as **Bundle Layer**. This layer is strategically positioned between the Application and Transport layers, marking a fundamental departure from the traditional Internet architecture. The incorporation of the Bundle Layer addresses the intricacies of challenged networks, offering a framework that thrives in environments where conventional protocols struggle.

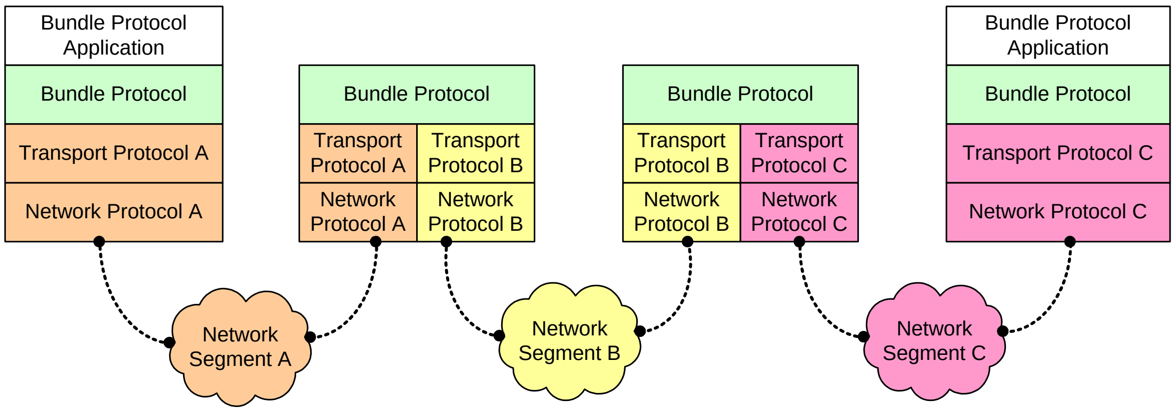


Figure 1 - The DTN Architecture (from [Persampieri 2023])

The entities responsible for executing the Bundle Layer are denoted as **DTN nodes**. Importantly, it is not mandatory for every node within the network to incorporate this supplementary layer; the nodes situated in the Network Segment, as illustrated by the cloud in Figure 1, can function as basic routers within the Internet network.

The unit of information exchanged at the Bundle Layer is the **bundle** and the protocol used is called **Bundle Protocol** (BP).

A DTN node adheres to a "**store, carry, and forward**" policy to effectively manage bundles, a strategy designed to address the potential absence of a continuous path between source and destination nodes. Each node encountered along the communication path follows these rules:

- Store: upon receiving incoming bundles, a node store them locally until the path is available.
- Carry: each node has the capability to carry the bundles along with it.
- Forward: the node forwards the stored bundles to the next node along the source-to-destination path when the corresponding link is available (after minutes or hours, it doesn't matter).

In challenged networks, a node may encounter limitations in sending bundles for an indeterminate duration. This limitation could arise due to factors such as temporarily unavailable links.

1.2 Bundle Protocol

This protocol stands at the cornerstone of the DTN architecture, with its initial version known as version 6 (BPv6) being standardized by the Internet Research Task Force (IRTF) in an Experimental Request for Comments (RFC) [RFC5050] released in 2007. Only in January 2022 its new version 7 has been formalized by the Internet Engineering Task Force (IETF), which is derived from IRTF, as a Proposed Standard RFC [RFC9171].

The introduction of **BPv7** prompted updates to the BPv6 implementation and gave rise to several new implementations, including ION by NASA-JPL (Jet Propulsion Laboratory) [ION] and DTNME by NASA-MSFC (Marshall Space Flight Centre) [DTNME]. These implementations have found practical application in communication between the International Space Station (ISS) and Earth, highlighting the adaptability and relevance of the Bundle Protocol in space communication systems.

Each bundle node can send or receive bundles and is composed by:

- Bundle Protocol Agent (BPA),
- Application Agent,
- a set of zero or more Convergence Layer Adapters (CLA).

The **Bundle Protocol Agent** stands as the pivotal component within a bundle node, playing a central role in implementing the services provided by BP, which are explained in detail in the original documentation [RFC9171].

The **Application Agent** uses the services offered by BP to communicate. It's composed by two elements:

- Administrative Element,
- Application-Specific Element.

The first pertains to reception or transmission of Administrative Records, encompassing elements such as "Status Reports" that provide information regarding the processing of a particular bundle. The latter deals with sending or receiving Application Data. For a quick overview, refer to Figure 2.

In addition, every bundle node, or the services it provides, is uniquely identified by an **Endpoint ID** (EID). This identification is crucial for communication, facilitating the participants in recognizing the sender and receiver of bundles. Endpoint IDs are distinguished by the scheme they adhere to, either "*dtn*" or "*ipn*", as specified in [RFC9171].

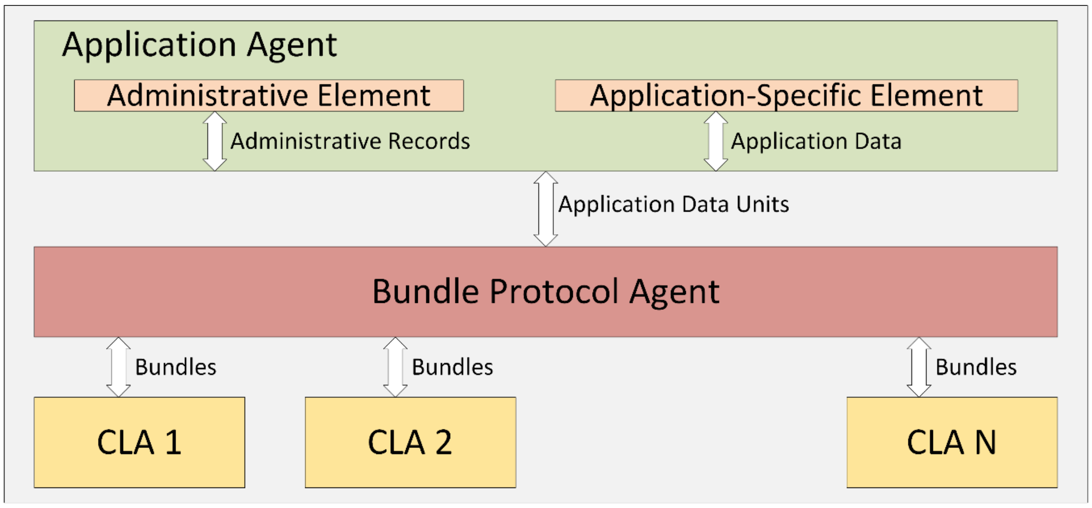


Figure 2 - Bundle Node structure (from [Persampieri_2023])

1.2.1 Bundle format

A bundle is composed by:

- one Primary Block,
- one Payload Block,
- zero or more Extension Blocks.

It's noteworthy that each bundle, for the purpose of transmission, undergoes serialization using **Concise Binary Object Representation** (CBOR) encoding, as outlined in [RFC8949]. For more comprehensive details on this encoding process, you can refer to [RFC9171].

The Primary Block has a different structure with respect to the other blocks and represents the **bundle header**.

The Payload Block and the Extension Block(s) (particularly the last one on which this thesis is focused) are encoded in a common format called **Canonical Bundle Block**, as reported in [RFC9171]. Below, the information contained in each block of this type:

- Block Type Code: a numeric identifier signifying the category of the block. Multiple blocks of the same type can coexist within the same bundle.
- A distinctive numeric identifier for this block within the bundle.

- Block Processing Control Flags: flags indicating characteristics unique to this particular block.
- CRC type: different CRC types can be assigned to different blocks.
- Block-type-specific data: the data encapsulated by the block, inclusive of its length.
- CRC value calculated based on the encoding of this Canonical Bundle Block, contingent upon the specification of a CRC type for this block.

1.3 Licklider Transmission Protocol

The Licklider Transport Protocol (LTP) [CCSDS LTP] [RFC5325] [RFC5326] serves as the designated "**Convergence Layer**" within Interplanetary networks following the Delay-/Disruption-Tolerant architecture. Specifically designed to accommodate long-delay scheduled intermittent links, LTP offers the flexibility of providing either a reliable or an unreliable service, denoted by "**Red**" and "**Green**" parts, respectively. LTP has the versatility to operate on top of either UDP or CCSDS space protocols.

In the context of the DTN architecture, on top of LTP, the Bundle Protocol is assumed to be running. It's noteworthy that other protocols could also be employed. Additionally, there may be an interface between BP and LTP, primarily facilitating bundle aggregation, playing a key role in managing the interaction between the BP and the LTP.

Here are the key features that differentiate LTP from TCP, with a focus on minimizing chattiness:

- No connection-establishment phase.
- Rate-based transmission speed.
- Unidirectional data flow.
- Bundles provided by BP are encapsulated within LTP blocks; then will be transmitted by independent LTP sessions running in parallel.
- An LTP block is divided into several LTP segments, each of which passed to UDP.
- Acknowledgments (i.e. report segments) are triggered only by data segments flagged as "checkpoints," usually at block end.

Certainly, the colors mentioned, i.e. “Red” and “Green”, can be interpreted as representing different Quality of Service (QoS) levels within the context of the LTP. Establishing a clear one-to-one mapping between the QoS required for transferring a bundle and the session color is a key aspect of LTP's design. As an illustrative example, if the bundle has to be transferred reliably, a red session is initiated; consequently, the bundle is encapsulated into a “Red block” indicating that the transmission should prioritize reliability. Conversely, if the unreliable service is required, the new session should be green and, in this case, the bundle will be inserted into a “Green block”, signaling that the transmission has to focus on best-effort.

1.4 DTNSUITE

The DTNsuite [DTNsuite], developed by the University of Bologna, is a comprehensive set of applications that leverages the Unified API [UnifiedAPI], formerly known as the Abstraction Layer.

DTNsuite applications (like DTNperf and many others) seamlessly integrate with major BP implementations such as DTN2/DTNME [DTNME], ION [ION], IBR-DTN [IBR-DTN], µD3TN [µD3TN] and Unibo-BP [Unibo-BP].

DTNperf [Caini 2013] is a powerful program for performance evaluation in DTN environments. It has 3 execution modes: “*client*”, which runs on the source, “*server*”, on the destination, and “*monitor*”. The first and the last, can also be used standalone. The client sends bundle, the server receives them (and if requested by the client, acknowledge them), and the monitor collects the status reports in a *.csv* file, which can be elaborated as a spreadsheet.

1.4.1 The Unified API

The Unified API [UnifiedAPI] serves as an intermediary abstraction layer designed to uncouple DTN applications from the specific APIs of individual BP implementations. It is designed to support all BP implementations, as well as other DTN applications. This integrated approach streamlines the development and maintenance of DTN applications, fostering consistency across various BP implementations and contributing to the broader ecosystem of Delay-/Disruption-Tolerant Networking.

The primary goal of the Unified API is to enable developers to perform tasks like sending or receiving bundles in a manner that is independent of the specific APIs of underlying BP implementations. This approach offers several advantages, including enhanced consistency in application behavior and improved maintenance efficiency.

The Unified API is structured into three layers, as illustrated in Figure 3:

- Top layer: contains the functions to be called by DTNsuite applications.
- Intermediate layer: functions as a "*clutch*", ensuring the independence of the top layer from specific BP implementations.
- Bottom layer: comprises interfaces with specific APIs.

Despite this layered design may appear intricate, it simplifies the process of adding support for new BP implementations. Introducing support for a new BP involves writing a new interface at the bottom layer (and making minor adjustments at the intermediate layer). This streamlined approach was employed for Unibo-BP, and its interface with the Unified API is depicted at the bottom right of Figure 3.

Thanks to this **standardized interface** with Unified API, all DTNsuite applications can seamlessly operate with Unibo-BP, requiring no modifications to their code. Notably, these applications can be compiled exclusively for Unibo-BP or simultaneously for multiple BP implementations. In the latter case, the same executable file is versatile enough to operate on any subset of the five supported BP implementations. This flexibility proves especially advantageous for conducting interoperability tests across different BP implementations.

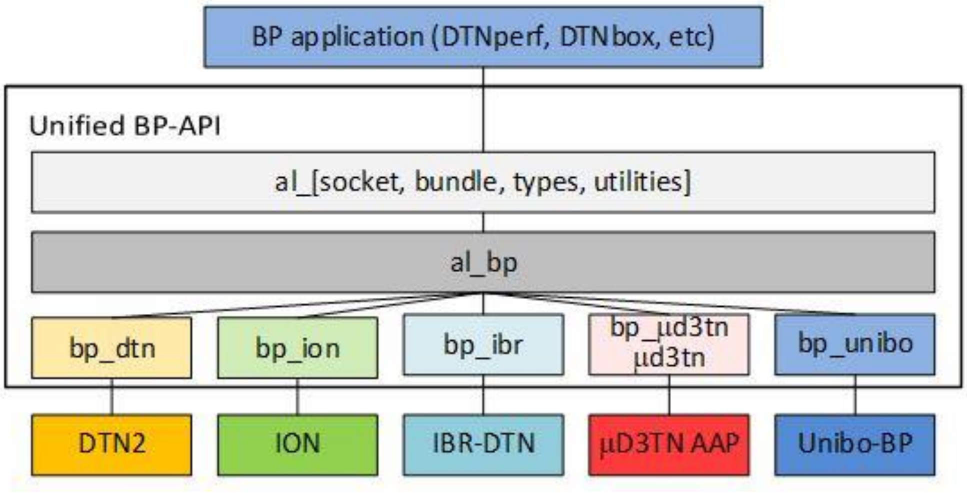


Figure 3 – Unified API's scheme of the hierarchical structure (from [Caini 2024])

1.5 UNIBO-DTN

Unibo-DTN [Unibo-DTN] encompasses all the crucial components of a DTN node developed by the University of Bologna. This comprehensive project incorporates the essential elements required for deploying a DTN node, with a specific emphasis on nodes operating within space networks. At present, this umbrella project includes the following specific projects: Unibo-BP [Unibo-BP], Unibo-CGR [Unibo-CGR], Unibo-LTP [Unibo-LTP], and ECLSA [ECLSA]. All of them have a dedicated repository on GitLab, from which they can be downloaded as Open Source under the GPLv3 license.

2 UNIBO-BP OVERVIEW

This chapter will explain more in detail how the Unibo-BP code was designed and how is organized, following the description presented in [Caini 2023] [Persampieri 2023]. Unibo-BP is released as free software under GPLv3 license and can be downloaded from a GitLab repository [Unibo-BP].

2.1 UNIBO-BP DESIGN AND ECOSYSTEM

The decision to implement Unibo-BP in C++20 reflects a deliberate choice to **enhance the independence** of Unibo software. The aim is to facilitate collaboration while ensuring that Unibo-BP can evolve independently to meet the specific needs and objectives of academic research over the years. Several factors contribute to the choice of C++ as the programming language for Unibo-BP:

- Reliability and modifiability, which makes easier for new users to understand and work with the codebase.
- Language familiarity.
- Maintainability over time, because it supports the development of robust, scalable, and maintainable codebases, ensuring that the software can be kept up-to-date.
- Compatibility with collaborators.

Unibo-BP assumes a central role within the broader Unibo-DTN umbrella project, which also encompasses Unibo-CGR [Unibo-CGR] and Unibo-LTP [Unibo-LTP], for example. The latter is an external, independent module within the Unibo-DTN framework, and, unlike Unibo-CGR, Unibo-BP does not have a direct dependency on Unibo-LTP for compilation. Unibo-BP is designed to be a versatile component that can also be used by the DTNsuite project applications [DTNsuite] through the Unified API [UnifiedAPI].

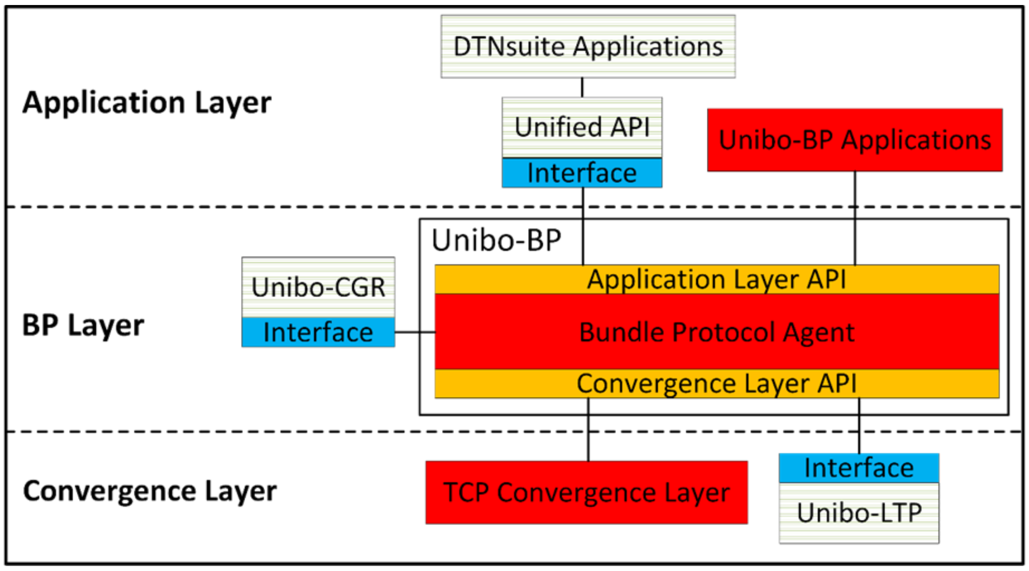


Figure 4 - The Unibo-BP ecosystem (from [Caini 2023])

The architectural representation in Figure 4 illustrates the **ecosystem** with Unibo-BP positioned at the core (because it implements the BPA), as the entity managing a DTN node. The BPA and Unibo-CGR are interconnected through an interface, which facilitates the exchange of information between the 2 components. The internal modules above and below the BPA interact with it through private APIs written in C++. These private APIs provide a structured and controlled means for communication, allowing for the seamless integration of various components within the ecosystem.

Below the BPA there is Unibo-LTP, written in C; to facilitate its integration, a new, specific interface has been developed, on the basis of public API functions written in C "**wrapping**" the private C++ functions. The same holds true for Unified API, which is above Unibo-BP.

The BPA is implemented as a single process with multiple threads. CLAs, on the other hand, are independent processes. **Inter-Process Communication** (IPC) mechanisms are employed for communication between these processes and threads, allowing them to exchange information and coordinate actions. Processes at the upper Application Layer are interested in sending/receiving bundles, and they also communicate with the BPA using IPC.

2.2 UNIBO-BP CLASSES

Unibo-BP, as already said in the previous section, is written in C++20, which, in this very recent version, introduces a range of innovative constructs, and Unibo-BP takes advantage of these features for enhanced expressiveness, readability and efficiency in code design.

Most classes within Unibo-BP are organized into **libraries**, a practice that enhances the clarity of the code structure. The use of libraries allows for logical grouping of related classes, making it easier for developers to navigate and understand the code. To avoid duplication of binary code, reduce the weight of processes, and minimize memory usage, Unibo-BP prefers **dynamic libraries** over static ones, which are loaded into memory at runtime, enabling multiple processes to share the same code in a more memory-efficient manner. Unibo-BP libraries are divided into "BP-specific" and "auxiliary".

2.2.1 BP-specific libraries

2.2.1.1 The BP library

Unibo-BP has a well-structured architecture; one of the most important libraries is the **BP library**, as it contains the classes that implement the data structures and the services of BPv7 [RFC9171]. It is worth stressing that Unibo-BP classes also cover several bundle extensions, some cited in RFC9171, such as "Previous Node", "Bundle Age", "Hop Count", other defined in RFC drafts, as "ECOS", and other designed by Unibo in support of Unibo-CGR, as "CGR Route" and "Geographical Route". One of the aims of the thesis was to add to these a **Metadata** extension, after updating to BPv7 the specifics for BPv6 in [RFC6258]. Let us summarize and highlight the key classes which have been used in this project:

- The ***ExtensionBlock*** class, itself an abstract class, representing extension blocks, which are optionally added to the primary and payload blocks to provide ancillary information. This class is designed to abstract diverse extension types, each governed by specific processing rules, with virtual methods to be implemented by the derived classes. These methods are subsequently invoked at the wanted phase of bundle processing.

- The ***Bundle*** class contains both volatile data and references to persistent data. For example, the *ExtensionBlock* class possesses both volatile and enduring representations. Each Bundle is associated with a 64-bit integer identifier unique in the node (stored in persistent memory), and references to persistent data are stored using the storage library (described later).

Within Unibo-BP, the BPA is composed of multiple submodules, dedicated to specific functions. While certain modules operate as daemons within dedicated threads, others are invoked as needed. Presented below is a concise overview of the key classes employed in the project:

- The ***BundleManager*** class has the task of retaining references to all bundles presently stored in the local node. To this end, each newly created or received bundle must be reported to the *BundleManager*. Additionally, this class manages the removal of expired bundles from the node, facilitated by an internal thread.

- The ***BPManager*** class is a utility class designed to support generic functions, e.g. CBOR encoding or decoding of a bundle.

2.2.1.2 Other BP-specific libraries

Going back to BP-specific libraries, there is the **IPC** (Inter-Process Communication) library, which facilitates communications between the different processes of Unibo-BP. Each message is encoded with a CBOR array of 2 elements: the first element serves as the header (a CBOR Unsigned Integer) and the second element represents the payload (any CBOR Major Type); the latter can be as complex as desired, whose format depends on the type of data that are encoded.

Then there are the **Client** and **Server** libraries. The former contains the functions used to send requests and receive responses; the latter, instead, are used to run servers within the BPA process.

Again, the **CLA** (Convergence Layer Adapter) library consists of a server (for the BPA process) and a client, implementing a specific Convergence Layer.

Last, the **API** (Application and Convergence Layer) library, that contains Unibo-BP public functions. These serve as C wrappers of internal C++ private methods, enabling external programs written in C (like Unified API and Unibo-LTP) to interface easily with Unibo-BP.

2.2.2 Auxiliary libraries

The core libraries introduced in Unibo-BP, developed to complement the functionality offered by the C++ standard library, play a crucial role in enhancing and extending the capabilities of the software. The most important ones are listed below:

- The "***io***" library: it contains classes that allow the programmer to abstract from the input/output medium at the lowest level.
- The "***storage***" library: it implements a file-based memory allocator.
- The "***time***" library: it contains some utility functions, i.e. to convert from UTC time to Unix Time.
- The "***math***" library: it contains several classes and utility functions such as endianness converters and generators of unique thread-safe 64-bit identifiers.

3 UNIBO-BP DEBUGGING IN VISUAL STUDIO CODE

This chapter is entirely dedicated to Unibo-BP debugging, because this procedure was crucial during the thesis, to better understand existing Unibo-BP code and to check the addition introduced. A few remarks are in order before proceeding. First, we need to stress that we preferred Visual Studio Code to Eclipse, because of its better support of CMake, used by Unibo-BP Makefiles. Second, before examining Unibo-BP debug, it is necessary to explain how multiple nodes can run on the same node, with Unibo-BP, as this feature was largely used in debug. Dealing with networking, is actually common to have at least two nodes, one acting as a sender and the other as a receiver. The possibility to debug one while the other run on the same machine is an obvious advantage. Third, we need to show how Unibo-BP can be imported in Visual Studio Code. Last, we can finally examine debug steps.

3.1 USE OF UNIBO-BP SCENARIOS

Unibo-BP is designed to run either one or multiple instances of the Bundle Protocol Agent on the same PC and corresponding configuration files to be used as templates are provided in the "***config_scripts***" directory of the Unibo-BP code (Figure 5). Here we focus on the case of multiple instances, for the mentioned reasons. One set of DTN nodes (BP Agents) forms a ***Scenario*** in Unibo-BP terminology. The "*/config_scripts/n_nodes*" directory contains the "***start-scenario***" and the ***stop-scenario***" scripts, plus one directory for each Scenario.

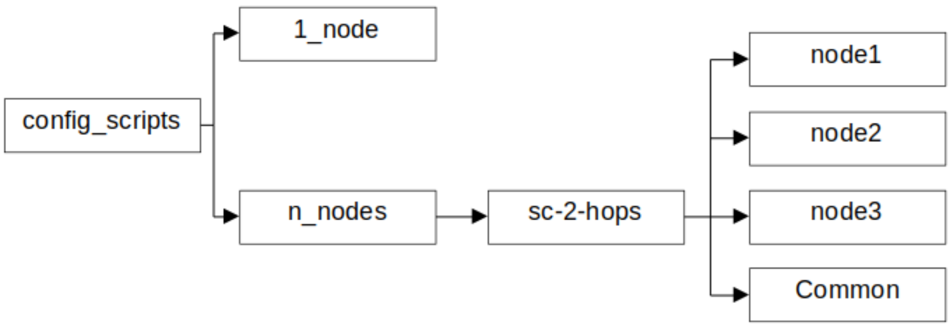


Figure 5 – Unibo-BP "config_scripts" directory design

A Scenario is, in turn, a collection of directories containing scripts. For each scenario there is the "***Common***" directory, which contains scripts common to all nodes (i.e. to configure contact-plan, routing etc.), and one additional directory for each node, which contains the scripts to start the node and others, e.g. to start services such as "*echo*" or "*ping*" on that specific node. Below the structure of the ***sc-2-hops***, considered here as a reference:

```
drwxrwxr-x 2 carlo carlo 4096 feb  4  2023 Common

drwxrwxr-x 2 carlo carlo 4096 feb  4  2023 node1

drwxrwxr-x 2 carlo carlo 4096 feb  4  2023 node2

drwxrwxr-x 2 carlo carlo 4096 feb  4  2023 node3

-rw-rw-r-- 1 carlo carlo   18 feb  4  2023 nodes.txt
```

It consists of three nodes, named as the directories (*node1*, *node2*, *node3*) forming two DTN hops (node1-node2 and node2-node3). The text file "*nodes.txt*" simply contains the names of the nodes.

3.1.1 Scenario's script files

3.1.1.1 The "start-scenario" script

Let us examine first the *start-scenario* script, which requires in input the name of the desired scenario ($1). As similar scripts, it checks first whether the command is written in the right way, then it unsets the UNIBO_BP environment variable, normally used when one instance of the BP Agent is launched. If set, this variable would disable the default mechanism that links each command/application to the BP Agent instance supposed to be running in the same directory. This mechanism is essential when dealing with multiple instances and is used in all the scripts. Pay attention, however, that the mechanism fails if the path from which the BP Agent is launched is longer than about 80 characters. To be safe, it is suggested to copy the *config_scripts* directory, in "*/home/[your-name]/*" directory, to have a shorter path.

After that, the *start-scenario* script continues by calling the "*nodes-array.sh*" script with the line "`source nodes-array.sh "$1"`". This script reads the *nodes.txt* file of the scenario specified in $1, containing the names of all the nodes of the scenario, and then puts the node names in the array “nodes”; as the internal script is called with the “*source*” command, the array is available inside the calling script.

The script continues with a few “for” cycles. The first starts all nodes by moving into each node directory and by launching from here a new instance of the BP Agent. To send commands to this BP Agent it will be enough to launch commands from this directory.

```
for node in ${nodes[@]}; do

      cd ../${node}

      # clear pid file

      > pids

        echo "Starting ${node}..."

        ./start-daemon.sh | awk '{print $2}' >> pids

done

echo "done"

echo

sleep 5
```

All other cycles use this mechanism and call from each node directory first the configuration scripts contained in the “*Common*” directory (*cla.sh*, *region.sh*, *extension.sh*, *routing.sh*, *contact-plan.sh*):

```
for node in ${nodes[@]}; do

        cd ../${node}

        ../Common/cla.sh

        ../Common/region.sh

        ../Common/extension.sh

        ../Common/routing.sh

        ../Common/contact-plan.sh

done
```

```
echo "done"

echo
```

Then inducts:

```
for node in ${nodes[@]}; do

        cd ../${node}

        ./induct.sh

done

echo "done"

echo
```

Outducts:

```
for node in ${nodes[@]}; do

        cd ../${node}

        ./outduct.sh

done

echo "done"

echo
```

And services:

```
echo "Setup services..."

for node in ${nodes[@]}; do

        cd ../${node}

        ./service.sh

done

echo "done"

echo
```

3.1.1.2 The “stop-scenario” script

This script is the dual of *start-scenario* and it has in common the first part, i.e. it first unsets the UNIBO_BP environment variable and calls the *nodes-array.sh* script. Then, for each node, enters in the node directory from which the corresponding instance of the BP Agent was launched, and launches the “*stop*” command:

```
for node in ${nodes[@]}; do
      cd ../${node}
      unibo-bp-admin stop
done
```

3.1.1.3 The “force-stop” and the “cleanup-scenario” scripts

The script is the same as the previous one, but it performs a “hard” stop, i.e. it brutally kills Unibo-BP daemons using the “*kill*” command instead of “*unibo-bp-admin stop*”:

```
for node in ${nodes[@]} ; do
      cd ../${node}
      while IFS= read -r pid; do
            if [ -z $pid ] ; then continue ; fi
            kill -9 $pid
      done < pids
done
```

Although effective, the *force-stop* script, as any hard kill, leaves set the node hidden "switch" files (*.unibo-bp/unibo-bp.running*) designed to prevent the user from launching multiple BP Agent instances in the same directory. To restart the Scenario, it is therefore necessary to cancel these files. This is accomplished by the *cleanup-scenario* script. Note that switch files are automatically cancelled at the end of a “soft” closing:

```
for node in ${nodes[@]}; do
      cd ../${node}
      rm -f .unibo-bp/unibo-bp.running
```

```
done
```

3.1.2 Node's script files

Each node has a separate directory with scripts necessary for starting and configuring it. In the next examples, we will refer to node 1.

3.1.2.1 start-daemon.sh

This script starts the node 1 BP Agent (daemon) on the current directory. It consists of a single command, with a few parameters, like the storage size and the node EIDs, one for the "*dtn*" and one for the "*ipn*" scheme (both are compulsory in Unibo-BP):

```
unibo-bp start --set-storage-size 50000000 --dtn-admin dtn://node1.dtn/
--ipn-admin ipn:1.0 –daemon
```

3.1.2.2 induct.sh

This script sets a tcpcl induct (a server), on port 4556:

```
unibo-bp-admin tcpcl induct add --port 4556
```

3.1.2.3 service.sh

In the absence of other options, the first "*echo*" program will register with the local node number and default service number 7, e.g. as "*ipn:1.7*" on node 1. The second, with the demux token "*echo*", appended to the "*dtn*" the EID of the local node, e.g. as "*dtn://node1.dtn/echo*". The program does what expected, i.e. it sends back any bundle it receives, and thus it is suited to work in partnership with the "*ping*" program:

```
unibo-bp-echo --scheme ipn --daemon | awk '{print $2}' >> pids

unibo-bp-echo --scheme dtn --daemon | awk '{print $2}' >> pids
```

3.1.3 Start and stop of a Unibo-BP Scenario

From the *config_scripts* directory, we have to launch the following command to start the preferred scenario, i.e. *sc-2-hops* in our case:

```
./start-scenario n_nodes/sc-2-hops
```

After few seconds you should have all nodes started and configured, ready to be used in your experiments.

To stop the scenario, i.e. all nodes:

```
./stop-scenario n_nodes/sc-2-hops
```

You can also call the *force-stop-scenario* and *cleanup-scenario* scripts (in the same way as stop-scenario) if something goes wrong, to be sure that at the next launch everything will start correctly.

The script *force-stop-scenario* brutal kill the processes associated to Unibo-BP by calling the command “`kill -9`” followed by the PID (Process Identifier) of the process to stop. It is called in this way:

```
./force-stop-scenario n_nodes/sc-2-hops
```

The *cleanup-scenario script*, instead, deletes the node hidden "switch" files (*.unibo-bp/unibo-bp.running*) designed to prevent the user from launching multiple BP Agent instances in the same directory:

```
./cleanup-scenario n_nodes/sc-2-hops
```

3.2 IMPORTING UNIBO-BP IN VISUAL STUDIO CODE

Unibo-BP is written in C++ 20 and contains a Makefile which is used to setup the project in the initial stage. To be sure that the project will be correctly imported, double check to have downloaded both the C++ and the CMake extensions of Visual Studio Code (Figure 6). This can be done by selecting the Extensions icon in the left column of VS Code and then by writing the name of the desired extension; download the C++ and the CMake extensions one after the other.

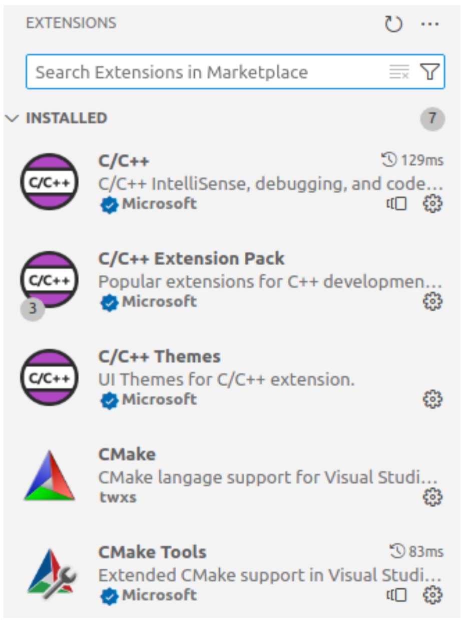

EXTENSIONS
Search Extensions in Marketplace
INSTALLED
7
C/C++
129ms
C/C++ IntelliSense, debugging, and code...
Microsoft
C/C++ Extension Pack
Popular extensions for C++ developmen...
Microsoft
C/C++ Themes
UI Themes for C/C++ extension.
Microsoft
CMake
CMake langage support for Visual Studi...
twxs
CMake Tools
83ms
Extended CMake support in Visual Studi...
Microsoft

Figure 6 – VS Code Extensions panel

Once CMake is installed, at the bottom of the IDE window you will see some commands related to it and one of them is associated to the "configure presets": you have to select "ide-debug-ninja". If ninja is not installed, you can install it in Ubuntu with the following command: "`sudo apt install ninja-build`". Always speaking about important commands, another one is the "bolt", which is near to the configure preset and necessary to start the build.

After this operation, you are ready to import the project (i.e. the folder containing the project), but, previously, the project must be already cloned from GitLab in an appropriate directory (*/sources/git/Unibo-DTN/unibo-bp*) by using the "`git clone`" command, followed by the link found in the GitLab project's web page.

VS Code does not copy source files but will use those in the original directory. Select the Explorer icon (the one at the top) in the left column and open the folder of the project you want to import, unibo-bp in this case, as you can see in Figure 7.

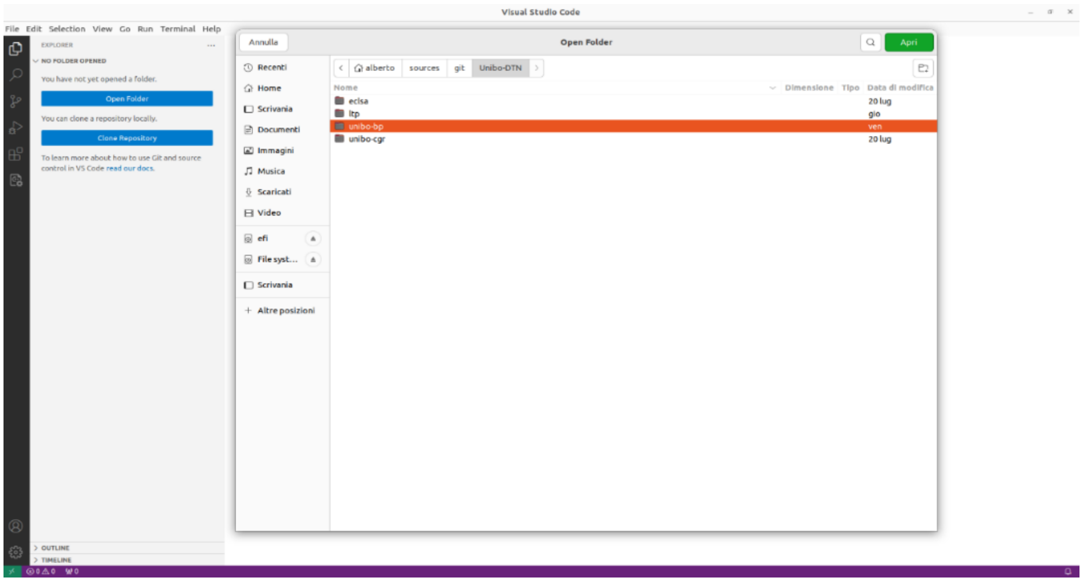
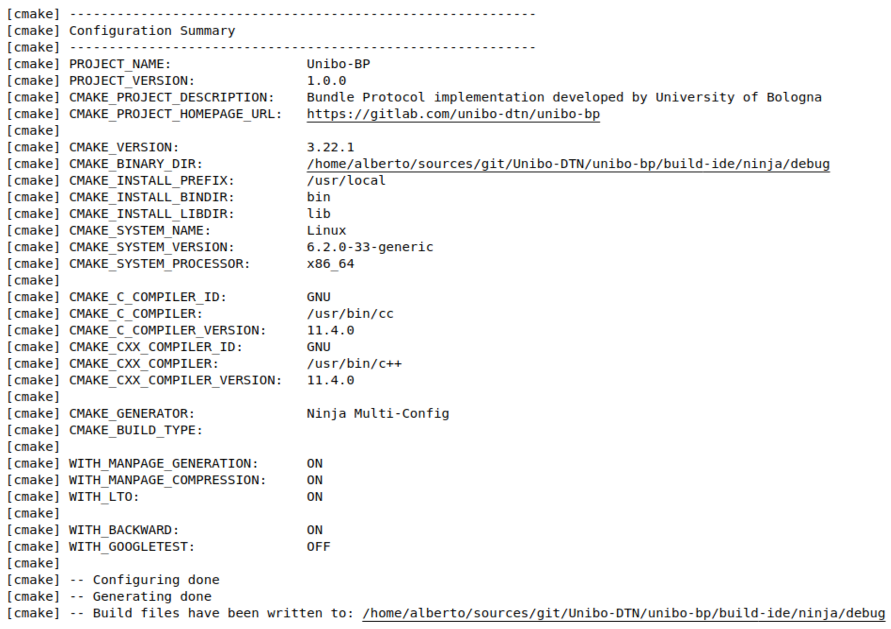


Figure 7 - Import of the Unibo-BP project



```
[cmake] -----------------------------------------------------------
[cmake] Configuration Summary
[cmake] -----------------------------------------------------------
[cmake] PROJECT_NAME:                 Unibo-BP
[cmake] PROJECT_VERSION:              1.0.0
[cmake] CMAKE_PROJECT_DESCRIPTION:    Bundle Protocol implementation developed by University of Bologna
[cmake] CMAKE_PROJECT_HOMEPAGE_URL:   https://gitlab.com/unibo-dtn/unibo-bp
[cmake]
[cmake] CMAKE_VERSION:                3.22.1
[cmake] CMAKE_BINARY_DIR:             /home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug
[cmake] CMAKE_INSTALL_PREFIX:         /usr/local
[cmake] CMAKE_INSTALL_BINDIR:         bin
[cmake] CMAKE_INSTALL_LIBDIR:         lib
[cmake] CMAKE_SYSTEM_NAME:            Linux
[cmake] CMAKE_SYSTEM_VERSION:         6.2.0-33-generic
[cmake] CMAKE_SYSTEM_PROCESSOR:       x86_64
[cmake]
[cmake] CMAKE_C_COMPILER_ID:          GNU
[cmake] CMAKE_C_COMPILER:             /usr/bin/cc
[cmake] CMAKE_C_COMPILER_VERSION:     11.4.0
[cmake] CMAKE_CXX_COMPILER_ID:        GNU
[cmake] CMAKE_CXX_COMPILER:           /usr/bin/c++
[cmake] CMAKE_CXX_COMPILER_VERSION:   11.4.0
[cmake]
[cmake] CMAKE_GENERATOR:              Ninja Multi-Config
[cmake] CMAKE_BUILD_TYPE:
[cmake]
[cmake] WITH_MANPAGE_GENERATION:      ON
[cmake] WITH_MANPAGE_COMPRESSION:     ON
[cmake] WITH_LTO:                     ON
[cmake]
[cmake] WITH_BACKWARD:                ON
[cmake] WITH_GOOGLETEST:              OFF
[cmake]
[cmake] -- Configuring done
[cmake] -- Generating done
[cmake] -- Build files have been written to: /home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug
```

Figure 8 - CMake configuration summary

After this step, at the bottom of the window you should see the process of the CMake tool which automatically starts the configuration of the project by checking for example the CMake version, the project's name and other parameters that you can see in Figure 8. If the configuration and the generation are done, everything should be ok.

3.3 UNIBO-BP DEBUGGING

3.3.1 Aim, challenges, and solution

Let us consider the *sc-2-hops* scenario, consisting of node 1, 2 and 3, and assume that we want to debug the BP Agent of node 1. The challenge is that the latter must be launched inside the IDE, while other nodes must be launched from a terminal, as usual.

The solution is in two steps:

1. start node 1 in the IDE, after having conveniently prepared the debug configuration (two .json files are needed);
2. start all other nodes with the *start-scenario* script, by adding the name of the node already started by the IDE as a second parameter:

```
./start-scenario sc-2-hops node1
```

3.3.2 Preliminary operations

Firstly, it is convenient to copy the original Scenario directory in *"/config_files/n_nodes"* to the "***test-dev***" directory. This because the 2 directories are made for 2 different goals: the latter is created for the debug, the former, instead, to collect template scripts. As the *test-dev* directory is in the "***.gitignore***" file of Unibo-BP, all files in it are never added to Unibo-BP package (it is supposed they are temporary). After the copy mentioned at the beginning of the chapter, you have to make a little modify to the *start-scenario* script, because as you can see from the previous chapter, the line used to call this script contains one more parameter, i.e. the node name. So, the modify to make is essentially the following: you have to check if the name of the node that is going to be started is equal to the parameter "`$2`", that corresponds to the node called by the terminal; if true, the script will activate only the remaining two nodes. The code below has already the modify made:

```
for node in ${nodes[@]}; do

      cd ../${node}

      # clear pid file
      > pids
      if(node != $2) {

        echo "Starting ${node}..."

        ./start-daemon.sh | awk '{print $2}' >> pids
      }

done

echo "done"

echo

sleep 5
```

Secondly, a preliminary build of the project is mandatory.

Then you need to add a new Debug Configuration in the Run and Debug menu (left column in VS Code). To this end, open the Run and Debug menu and at the top of it, click the arrow pointing downwards and click “Add Configuration…”. In this way the *launch.json* file is automatically opened (it is in a new directory called *.vscode*). Although this file is unique for all debug configurations, it contains a different section for each configuration. For example, if we want to build 3 debug configurations, one for each node of the *sc-2-hops*, this file will contain 3 sections, as in Figure 9.

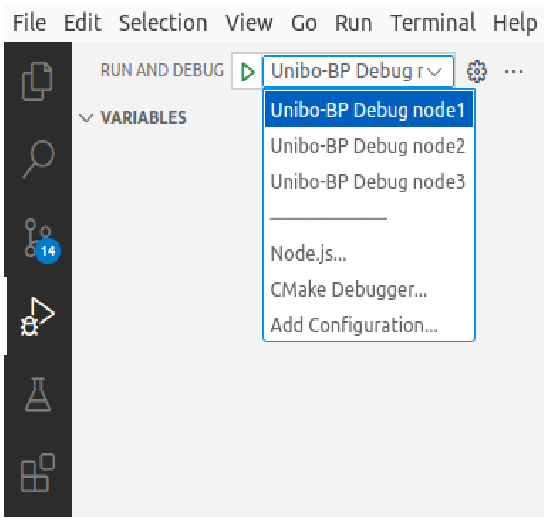


Figure 9 - Debug configurations panel

3.3.2.1 *launch.json (debug configuration file)*

In this file we need to fill many fields, the most important of which are the program we want to launch and its arguments. We assumed that we want to launch (debug) the BP Agent of node 1; the section corresponding to this debug configuration can be filled as below:

```
{
"name": "Unibo-BP Debug node1", //name of the command
"type": "cppdbg",
"request": "launch",
"program": //path of the program to launch
"${workspaceFolder}/build-ide/ninja/debug/Unibo-BP/bin/Debug/unibo-bp",
"args": [   //program arguments
      "start",
      "--set-storage-size",
      "50000000",
      "--dtn-admin",
      "dtn://node1.dtn/",
      "--ipn-admin",
      "ipn:1.0"
],
"stopAtEntry": false,
"cwd": "${workspaceFolder}/test-dev/sc-2-hops/node1",
"environment": [],
"externalConsole": false,
"MIMode": "gdb",
"setupCommands": [
      {
            "description": "Enable pretty-printing for gdb",
            "text": "-enable-pretty-printing",
            "ignoreFailures": true
      }
],
"preLaunchTask": "Unibo-BP stop sc-2-hops"
},
```

Analogous configurations for node 2 and node 3 can be added by copying the previous one and modifying as necessary. Each section must end with a curly bracket "}", followed by a comma (if it is not the last section).

The file above tells VS Code to start the BP Agent program (daemon) from the directory specified in "*cwd*" (*/test-dev/sc-2-hops/node1*). A pre-launch task called "Unibo-BP stop sc-2-hops" is executed before. This task stops the nodes of the scenario possibly active and must be performed before starting a new node only for security reasons.

3.3.2.2 tasks.json (pre-launch operations)

The actual actions performed in the "*preLaunchTask*" must be specified in the *tasks.json* file, which much be created from scratch by the user in the *.vscode* directory. As we want all nodes to be stopped before starting the debug of node 1, we will execute *stop-debug-node.sh* (whose code is reported in the next chapter) which will stop all nodes as shown below.

```
{
"label": "Unibo-BP stop sc-2-hops", //preLaunchTask name
"type": "shell",
"command": "${workspaceFolder}/test-dev/stop-debug-node.sh",
"args": ["sc-2-hops"],
"options": {
      "cwd": "${workspaceFolder}/test-dev/"
      }
}
```

It is essential to insert in the "*label*" field the same name reported as preLaunchTask in the *launch.json* file.

3.3.3 Debug execution

The following scripts start executables named "*unibo-bp**". Unless additional user action is taken, these executables are the last ones to be installed on the system. For example, if a change is made to unibo-bp and it is not installed on the system, we will find ourselves in a situation where the debugging node is running with the last change made while the other nodes are not. There are several alternatives to get around this problem but the easiest one to remember is obviously to re-install Unibo-BP into the system after each change you want to test (`make distclean` && `make init` && `make` && `sudo make install` && `sudo ldconfig`).

3.3.3.1 Start debug

The operations to perform are listed below:

1. From "Run and Debug" select the preferred debug configuration and click the play button.
2. if everything is ok, it should appear on the top of the page a box containing some buttons, as shown in the image below (Figure 10).

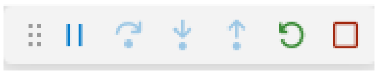


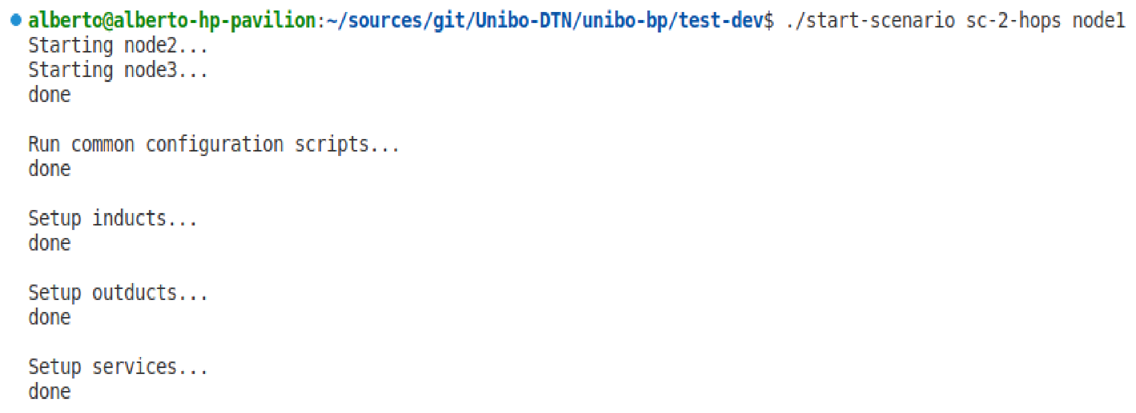
Figure 10 - Debug box

3. at this point, only one node is running (node 1 for example); to start the others open a bash terminal (even directly in VS Code), go to the *test-dev* directory and launch the following command, which starts all the BP Agents except the one started in the IDE, e.g. node 1, and then it configures all nodes, including the one started inside the IDE:

```
./start-scenario sc-2-hops node1
```

Before of starting the script, it is very important that the node inside VS Code is effectively started, which means that no breakpoint is blocking its execution.

If the procedure is successful, the output in the bash will be like the following (Figure 11):



```
alberto@alberto-hp-pavilion:~/sources/git/Unibo-DTN/unibo-bp/test-dev$ ./start-scenario sc-2-hops node1
Starting node2...
Starting node3...
done

Run common configuration scripts...
done

Setup inducts...
done

Setup outducts...
done

Setup services...
done
```

Figure 11 - Starting of the nodes in the Unibo-BP's debug procedure

3.3.3.2 Stop debug

To stop the Scenario, in the bash shell call the "*stop-debug-node.sh*" script to stop the entire debug session:

```
./stop-debug-node.sh sc-2-hops
```

And it is realized in this way:

```
#!/bin/bash
if [$# -ne 1] ; then
      echo "Usage: $0 <test-name>" 1>&2
      exit 1
fi
./stop-scenario $1
sleep 5
./force-stop-scenario $1
./cleanup-scenario $1
exit 0
```

If no breakpoint stops the node, we will see that the session will be normally closed (so you don't have to click the stop button in the box). There is a problem if you stop the debug in a brutal way: the other nodes of the Scenario will remain active, for example if immediately after the stop you want to start another Scenario, the TCP/UDP ports will conflict with each other.

This script tries to stop all nodes (including the one already closed) by calling in succession the script files *stop-scenario*, *force-stop-scenario* and *cleanup-scenario* which were already descripted above. The 5 seconds sleep above *force-stop-scenario* is inserted because lets the nodes stop in a gracefully way, before managing corner cases with *force-stop-scenario*. The latter and cleanup-scenario are necessary, but their aim is to manage situations (caused by bugs) in which nodes are not correctly terminated.

4 UNIFIED API DEBUGGING IN VISUAL STUDIO CODE

When dealing with the development of the Metadata bundle extension, it was necessary to simultaneously debug both the Unified API functions devoted to Metadata, and the new classes introduced in Unibo-BP. Therefore, it was necessary to have two IDE instances, one for Unified API and one for Unibo-BP. The former could be an instance of either Eclipse or Visual studio code, as Unified API is written in C and does not use CMake, indifferently. However, for commonality, we preferred to use the same IDE for both. Below we will explain how to debug Unified API alone and then with Unibo-BP, in parallel on two Visual Studio Code instances.

4.1 IMPORT OF THE UNIFIED API PROJECT

First, it is necessary to create a directory named "***dtnsuite***" which, as you can understand, it will contain the applications that leverage the Unified API (here, as an example, there will be only "*dtnperf*") and Unified API itself. At this point, you have to create another directory called "***unified_api***" in which you must clone the Unified API GitLab project using the command "`git clone`" followed by the link found in the GitLab project's web page. Finally, you have to copy the "Makefile_dtnsuite" in the dtnsuite directory and rename it as "***Makefile***". This file is crucial because it selects for which application make the build and other operations.

The import in VS Code of Unified API is like that of Unibo-BP. From the Explorer icon select the entire *dtnsuite* directory, which contains the Makefile used to build the Unified API library and the DTNsuite applications based on it, including the "***test_program***" file used later. After this step, your VS Code window should be like Figure 12:

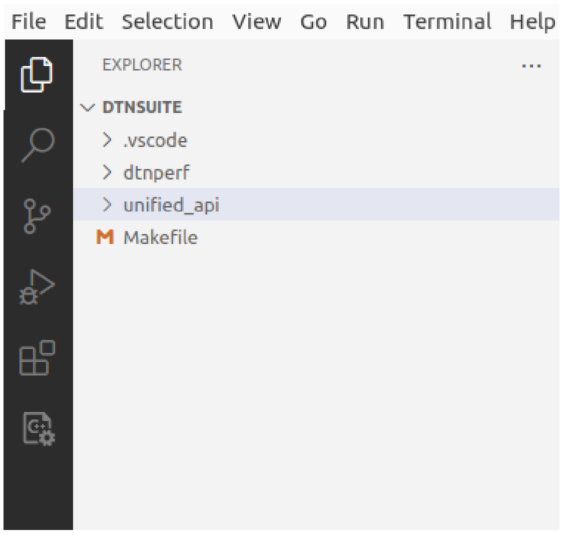


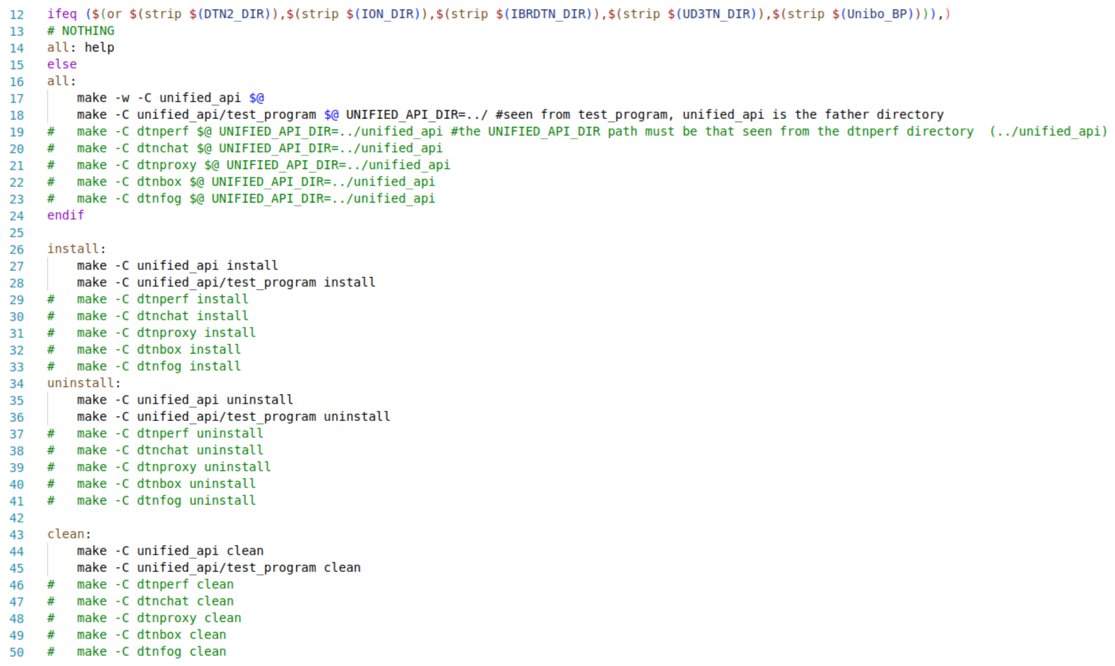
Figure 12 - Explorer section after importing the dtnsuite directory

In this case, for simplicity, the *dtnsuite* directory contains only the *dtnperf* and *unified_api* directories, as we omitted the other applications of the DTNsuite. The important thing is that there is also the DTNsuite Makefile because by modifying it, it is possible to set which application(s) must be built.

4.2 DTNSUITE'S MAKEFILE STRUCTURE: SELECTING "UNIFIED API" AND "*TEST_PROGRAM*"

As you can see from the Figure 13, this file has various configuration like "*all*" and "*install*". DTNsuite's Makefile calls the Makefiles specific to DTNsuite components, thus by commenting or decommenting lines in the component lists, we can easily select which components have to be built or installed, etc. The only compulsory component is obviously Unified API, as all other applications are built on top of it.

As Unified API is not an application but a library, to debug it we need an application. To this end, it is convenient to build only the *test_program* included in the Unified API package, which is much simpler than other DTNsuite applications. Although simple, it allows the developer to test all the functions provided by the Unified API. Note that it requires only a single node, as it simply sends a few bundles to itself.



```
ifeq ($(or $(strip $(DTN2_DIR)),$(strip $(ION_DIR)),$(strip $(IBRDTN_DIR)),$(strip $(UD3TN_DIR)),$(strip $(Unibo_BP)))),)
# NOTHING
all: help
else
all:
	make -w -C unified_api $@
	make -C unified_api/test_program $@ UNIFIED_API_DIR=../ #seen from test_program, unified_api is the father directory
#	make -C dtnperf $@ UNIFIED_API_DIR=../unified_api #the UNIFIED_API_DIR path must be that seen from the dtnperf directory  (../unified_api)
#	make -C dtnchat $@ UNIFIED_API_DIR=../unified_api
#	make -C dtnproxy $@ UNIFIED_API_DIR=../unified_api
#	make -C dtnbox $@ UNIFIED_API_DIR=../unified_api
#	make -C dtnfog $@ UNIFIED_API_DIR=../unified_api
endif

install:
	make -C unified_api install
	make -C unified_api/test_program install
#	make -C dtnperf install
#	make -C dtnchat install
#	make -C dtnproxy install
#	make -C dtnbox install
#	make -C dtnfog install
uninstall:
	make -C unified_api uninstall
	make -C unified_api/test_program uninstall
#	make -C dtnperf uninstall
#	make -C dtnchat uninstall
#	make -C dtnproxy uninstall
#	make -C dtnbox uninstall
#	make -C dtnfog uninstall

clean:
	make -C unified_api clean
	make -C unified_api/test_program clean
#	make -C dtnperf clean
#	make -C dtnchat clean
#	make -C dtnproxy clean
#	make -C dtnbox clean
#	make -C dtnfog clean
```

Figure 13 - dtnsuite Makefile structure

4.3 Visual Studio Code settings

The debug is made by executing the *test_program* in VS Code IDE, as said. Its code consists of a single and very simple .c file (called "*main.c*") which must be built it. To this end, you need the same steps as for Unibo-BP, with the differences pointed out below.

4.3.1 The "*settings.json*" file

In the "`makefile.configurations`" field of this file, you must enter the Makefile configuration that will be used to compile Unified API (you can also insert other configurations for other programs, but this does not matter). In particular, it is very important to set two parameters: "UNIBO_BP=1", which makes Unified API be compiled for Unibo-BP (or better, also for Unibo-BP, as DTNsuite is designed to create executable able to run on a variety of different BP implementations), and "DEBUG=1", which adds debug symbols and remove any optimizations (they could alter the order of code line execution), as it is always required dealing with debug.

As you may have noticed, this "*settings.json*" file is not configured in this way in Unibo-BP, because that project does not use the Makefile to execute the build, but "ninja", so it is not necessary to be configured. This file is associated directly with the Makefile settings, so, since Unified API is directly built and executed using it, this file must be modified with the user's settings.

```
{
      “makefile.configurations”: [
            {
                  “name”: “Debug Unified API”,
                  “makeArgs”: [“UNIBO_BP=1”, “DEBUG=1”]
            }
      ],
      “makefile.makefilePath”: “/home/alberto/sources/git/dtnsuite”
}
```

Last but not least, you have to add the line that makes VS Code conscious of what is the Makefile to consider, i.e. the general Makefile in the *dtnsuite* directory.

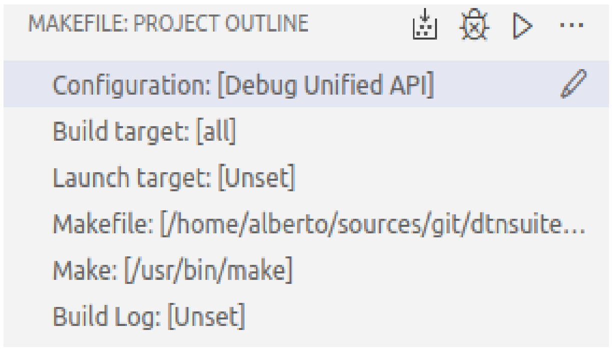
4.3.2 The “*launch.json*” file

This file, as for Unibo-BP, is automatically created by VS Code when you add a new debug configuration in the Run and Debug section. By clicking on “Add configuration”, this file is opened, and you can insert a debug configuration for your application. In this case, the program that has to be debugged is the *test_program* of Unified API, so this section can be filled in this way:

```
{

"name": "Debug Unified API - test_program",

"type": "cppdbg",

"request": "launch",

"program":
"/home/alberto/sources/git/dtnsuite/unified_api/test_program/test_progra
m_vUNIBOBP",

"args": [

      "--unreliable",

      "--mb-type", "1",

      "--mb-string", "giallo",

      "--debug=1"
],

"stopAtEntry": true,

// the first path is for testing with also debugging Unibo-BP, the
second is for using the current installed version

//"cwd": "/home/alberto/sources/git/Unibo-DTN/unibo-bp/test-dev/sc-2-
hops/node1/"
"cwd": "/home/alberto/Unibo-BP",

"environment": [],

"externalConsole": false,

"MIMode": "gdb",

"setupCommands": [

      {

            "description": "Enable pretty-printing for gdb",

            "text": "-enable-pretty-printing",

            "ignoreFailures": true

      }

  ]

}
```

Note that the field "*args*" contains the input parameters to be passed to the *test_program*, which are the same as those that would be inserted by calling the program form the command line prompt. Most likely they will be frequently changed to match the diverse debug purposes. In the case presented, it is noteworthy the request of inserting a metadata block of type "1" containing the "yellow" string. The parameter "debug=1" is necessary to enable the print of metadata fields in some Unified API functions, which is useful to check the correct metadata writing/reading on the spot, from the console window of the IDE.

4.4 BUILDING THE *TEST_PROGRAM* FOR DEBUG AND EXECUTION


MAKEFILE: PROJECT OUTLINE

Configuration: [Debug Unified API]
Build target: [all]
Launch target: [Unset]
Makefile: [/home/alberto/sources/git/dtnsuite...
Make: [/usr/bin/make]
Build Log: [Unset]

Figure 14 - Makefile properties

First of all, you have to build the *test_program*. In the Makefile section (the last of the left column in VS Code), you have to select the configuration dedicated to Unified API, which can be called as you prefer, as you can see above in the previous section when talking about the "*settings.json*" file (in this case is called "Debug Unified API"). This configuration is in fact created at the same time as the *settings.json* configuration, in which you can add as many configurations as you like. So, select "all" as a build target, which creates the executable version of the test program; select "clean" in advance if you want to remove old files, to build from scratch. The Makefile used is, as already specified above, the general one which includes all the DTNsuite applications. Your VS Code Makefile section, at the end, will be like the one in the Figure 14.

At this point, you are ready to build the program by clicking the first button next to "Makefile: project outline" line. If everything until now is correct, in the *test_program* directory you will find an executable called "***test_program_vUNIBOBP***" (if compiled for Unibo-BP, as assumed here).

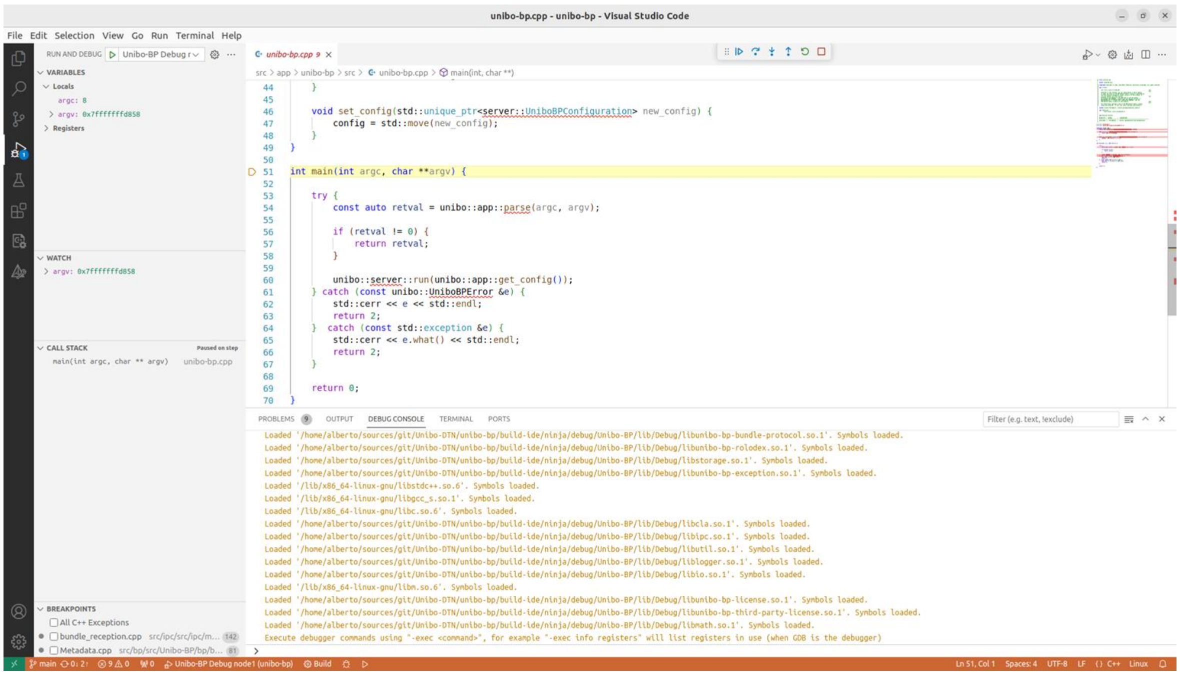
Now, from the Run and Debug section, after selecting the appropriate debug configuration for your executable, you can start the debug by clicking the play button. It is possible to insert some breakpoints to the steps of sending and receiving bundles, to better understand how the application works. A very important thing to keep absolutely in mind is that Unibo-BP requires to be activated before the execution of the debug here in Unified API, otherwise an error will be displayed, and the debug stopped.

4.5 SIMULTANEOUS DEBUG OF THE *TEST_PROGRAM* AND OF A UNIBO-BP BPA

To the simultaneous debug you need two different instances of Visual Studio Code, one for the *test_program* of Unified API and one for Unibo-BP. We chose Visual Studio Code for both for the reasons mentioned above.

4.5.1 Unibo-BP

First, in the VS code instance for Unibo-BP start the BP Agent. For this purpose, go to the Run and Debug section of VS Code and select the debug configuration associated to node 1 (or any another node you may prefer); the output will look as that in Figure 15.


unibo-bp.cpp - unibo-bp - Visual Studio Code
File Edit Selection View Go Run Terminal Help
RUN AND DEBUG
Unibo-BP Debug r
unibo-bp.cpp 9
src > app > unibo-bp > src > unibo-bp.cpp > main(int, char **)
VARIABLES
Locals
argc: 8
argv: 0x7fffffffd858
Registers
WATCH
argv: 0x7fffffffd858
CALL STACK
Paused on step
main(int argc, char ** argv) unibo-bp.cpp
BREAKPOINTS
All C++ Exceptions
bundle_reception.cpp src/ipc/src/ipc/m... 142
Metadata.cpp src/bp/src/Unibo-BP/bp/b... 81
PROBLEMS 9
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
main
Unibo-BP Debug node1 (unibo-bp)
Build

```cpp
    }

    void set_config(std::unique_ptr<server::UniboBPConfiguration> new_config) {
        config = std::move(new_config);
    }
}

int main(int argc, char **argv) {

    try {
        const auto retval = unibo::app::parse(argc, argv);

        if (retval != 0) {
            return retval;
        }

        unibo::server::run(unibo::app::get_config());
    } catch (const unibo::UniboBPError &e) {
        std::cerr << e << std::endl;
        return 2;
    }  catch (const std::exception &e) {
        std::cerr << e.what() << std::endl;
        return 2;
    }

    return 0;
}
```

```
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libunibo-bp-bundle-protocol.so.1'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libunibo-bp-rolodex.so.1'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libstorage.so.1'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libunibo-bp-exception.so.1'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libstdc++.so.6'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libgcc_s.so.1'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libc.so.6'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libcla.so.1'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libipc.so.1'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libutil.so.1'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/liblogger.so.1'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libio.so.1'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libm.so.6'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libunibo-bp-license.so.1'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libunibo-bp-third-party-license.so.1'. Symbols loaded.
Loaded '/home/alberto/sources/git/Unibo-DTN/unibo-bp/build-ide/ninja/debug/Unibo-BP/lib/Debug/libmath.so.1'. Symbols loaded.
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers in use (when GDB is the debugger)
```

Figure 15 - Starting of the debug in Unibo-BP

Then click play inside the debug box to activate all the threads in the call stack. You are ready to effectively activate the selected node (node 1 in this case) the same way as explained before, when talking about the debug in Unibo-BP. At this point, set breakpoints in specific parts of the code, when convenient, e.g. where the encode and the decode of the Metadata extension are performed, to check if the setting of its components (metadata type, metadata length and metadata string) are correct in Unibo-BP.

4.5.2 Unified API

First, modify the "`cwd`" path in the *launch.json* file of Unified API, so that it corresponds to the directory of the BP Agent instance started by the VS Code dedicated to Unibo-BP. For example, the new path is the following on my machine:

```
"cwd": "/home/alberto/sources/git/Unibo-DTN/unibo-bp/test-dev/sc-2-
hops/node1/"
```

Then, start the debug of the *test_program* of Unified API. As for the debug of Unibo-BP, you need to conveniently set a few breakpoints. For the Metadata extension, for sure in the "*send*" and in the "*receive*" functions, where metadata are inserted and read, respectively. Note that on to the IDE devoted to Unified API, you can debug the code of *test_program*, of the Unified API functions, and of the public Unibo-BP APIs in C, but you could not debug neither the internal APIs in C++ nor any other component of Unibo-BP. This is why an independent instance of VS code, devoted to Unibo-BP, is necessary.

5 UNIBO-LTP INTERFACE TO UNIBO-BP

Unibo-LTP [Unibo-LTP] is an implementation of **Multicolor LTP**, an experimental version of ION presented in [Bisacchi 2022]. Unibo-LTP was initially devised as an ION [ION] plug-in but with the building of Unibo-BP it was necessary to include it in its ecosystem. This has been achieved by building a new interface to Unibo-BP in the Unibo-LTP code and by developing a few enhancements in the Unibo-BP code itself. Of great importance for the Unibo-BP interface are the incorporation of the novel "**Orange**" LTP color, which is added to the standard "Red" and "Green" colors, denoting reliable and unreliable services, respectively, and of LTP session timers.

Multicolor LTP assumes that a block can solely be **monochromatic**, signifying it may comprise either a Red or a Green part but not both, as allowed by the standard [RFC5326]. Should the block be Red, the transfer is conducted reliably, entailing the retransmission of lost LTP segments. Conversely, if Green, the block is dispatched as best effort, devoid of any LTP segment retransmission. Orange introduces an extra "**notified**" service. In the event of an Orange block reaching its destination unscathed, a positive reception is relayed to the LTP sender, akin to the Red scenario. By contrast, if any LTP segment is loss, the block is entirely discarded, a failure LTP segment is relayed to the LTP sender, which subsequently notifies the BP that the bundle contained in the LTP block was not delivered. At this stage, BP can determine whether to initiate a bundle retransmission, implying a new LTP session, or not. RFC delegates this decision to the implementation [RFC9171] and both ION and Unibo-BP prefer to resend the bundle.

ION implementation of LTP by default use RED blocks (reliable service) for bundles, with Green (unreliable) reserved to bundles whose "best effort" (alias "unreliable") ECOS [Draft ECOS] extension flag is set. The decision is bundle by bundle, as each bundle possesses its individual ECOS settings. The original version of Unibo-LTP was partially different, because in Multicolor LTP the concept of default link color was introduced, for the reasons explained in [Bisacchi 2022]. Therefore, the policy was that by default LTP used the link color (Red, Green or Orange) indicated in Unibo-LTP settings; this default was however overridden by selecting Green when the best effort ECOS flag was set, as in ION. This way it was possible to use the Orange color, by setting it as link color when convenient, and at the same time to preserve the ION behavior in response to the best effort ECOS flag. By the way, the change of the default LTP color in response to a characteristic of the bundle to be transferred by LTP, can be seen as a first preliminary mapping between QoS at bundle layer (the best effort ECOS flag) and LTP QoS (the color used).

5.1 Modification introduced

One of the aims of the thesis, was to extend this embryonic mapping to something much more elaborated. To this end, the Unibo-LTP interface to Unibo-BP was modified to convey not only the unreliable flag (akin to the ION interface), but also all other ECOS parameters (multiple flags and additional details like priority) [Draft ECOS], along with the count of bundle retransmissions. This wealth of information on bundle QoS now enables Unibo-LTP to establish a **more precise mapping** with the LTP color. Currently, two ECOS flags, “best effort” and “reliable”, in conjunction with the retransmission count, are used to override the default LTP color. This may pave the way to more elaborated mapping in the future [Caini 2024].

A second extension was not related to QoS mapping, but to **timers**. The Unibo-BP interface offers a straightforward resolution to the issue encountered in Red sessions. In space scenarios, there is a potential occurrence where the bundle(s) within a Red block may become “stale”, meaning their lifetime is exceeded, especially during the retransmission of missing LTP segments. In such instances, the utilization of LTP Red would result in the inefficient consumption of time and bandwidth resources, persisting in completing a block whose content (the bundle) is destined to be discarded upon reception. Our interface addresses this challenge by conveying the bundle's lifetime information to Unibo-LTP, allowing for the adjustment of the Red session's lifetime (an extension of Multicolor LTP). Consequently, Red sessions can be promptly terminated once their content becomes obsolete, thereby preventing the wasteful expenditure of resources [Caini 2024].

5.2 THE LTP INTERFACE TO UNIBO-BP

The LTP interface to Unibo-BP is contained in the "***UniboBP-BundleProtocol_upper_protocol.c***" file, compiled only if Unibo-LTP is built for Unibo-BP. The original Unibo-LTP interface with Unibo-BP passed the bundle and only the "unreliable" ECOS flag, used to select the LTP Green color (if set). The new interface passes the bundle and ancillary information grouped under the "**bundle metadata**" term, including: all ECOS flags, the bundle deadline, the number of times the bundle has been retransmitted. The new interface mainly differs from the original one because of the introduction of a bundle metadata structure in pre-existing code. The post-thesis interface is described below. As already said in the title of this chapter, it is contained in the "*UniboBP-BundleProtocol_upper_protocol.c*" file of Unibo-LTP. Being an interface, this file is compiled and executed only when Unibo-LTP is compiled for Unibo-BP, as an alternative to ION.

The Unibo-LTP interface receives from Unibo-BP a bundle (when allowed by BP flow control), it enqueues it in an LTP queue, from which is extracted when allowed by the LTP flow control. It is based on 3 functions:

- ***onOutboundPdu()***: it encapsulates a bundle released by Unibo-BP and some ancillary information, called "bundle metadata", in the *transmission* structure and inserts it in the Unibo-LTP queue (called *peer_data*).
- ***canReceiveDataFromUpperProtocol()***: which reads from the queue when Unibo-LTP allows do to that.
- ***passedDataFromUpperProtocol()***: it creates the *dataToSend* structure containing the pointer to the serialized bundle and the metadata information. *dataToSend* is the only structure visible to the Unibo-LTP core.

Note that ***Outbound*** means that the bundle transfer is from Unibo-BP to Unibo-LTP, while ***Inbound*** the opposite, i.e. from Unibo-LTP to Unibo-BP.

5.2.1 The "*onOutboundPdu()*" function

This function is called when Unibo-LTP is notified by Unibo-BP that a bundle is ready to be transmitted. The bundle PDU is serialized in a file, whose file descriptor is saved in a Unibo-LTP structure called “***transmission***” (of *UniboBPTransmission* type), which is composed by the following fields:

```
typedef struct {
    uint64_t link_id;
    uint64_t transmission_id;


    int fd; // file descriptor of the file containing the bundle
    uint64_t length; //lenght of the file containing the bundle


    LTPBundleMetadata metadata;
} UniboBPTransmission;
```

Below, the structure “***LTPBundleMetadata***”, which is contained in the previous one, is shown:

```
typedef struct {
    LTPEcos ecos; //PARSED ECOS of the bundle
    struct timeval deadline; // deadline to complete the LTP Tx session
    uint64_t reTx_counter; // number of bundle retransmissions
} LTPBundleMetadata;
```

The *LTPBundleMetadata* groups all ancillary information passed by Unibo-BP to Unibo-LTP: all ECOS parameters, the deadline to complete a reliable (i.e. LTP Red) transfer (normally the bundle deadline) and the number of times the bundle has been retransmitted. All these parameters will be made available inside the core of Unibo-LTP inside the “***dataToSend***” structure, as it will be shown later. Now, for the sake of completeness, all ***LTPEcos*** values, organized in different sub-structures, are shown below in top-bottom order:

```
typedef struct {
      LTPEcos_fields fields;
      LTPEcos_priority priority;

      uint32_t QoS_tag;

} LTPEcos;
```

```
typedef struct {

    bool critical;

    bool bestEffort;

    bool qosTagIsPresent;

    bool reliable;

    bool bssp;

    bool bibe;

    bool bibeCustody;

} LTPEcos_fields;
```

```
typedef struct {

      ECOSCardinalPriority cardinal;

      int ordinal;

} LTPEcos_priority;
```

```
typedef enum ECOSCardinalPriority {

    ECOSCardinalPriority_bulk      = 0,

    ECOSCardinalPriority_normal    = 1,

    ECOSCardinalPriority_expedited = 2,

    ECOSCardinalPriority_unknown   = 3,

} ECOSCardinalPriority;
```

The bundle deadline and the number of retransmissions, are inserted into that structure by directly using the Unibo-BP API's functions. ECOS flags, however, are inserted in two steps. First, the unparsed flags (an integer) are obtained by means of the "*unibo_bp_ecos_clone()*"; then they are parsed into the individual fields of the *LTPEcos* structure by means of the new function "*parseECOS()*".

```
UniboBPECOS temp_ecos;

    UniboBPError error =
unibo_bp_ecos_clone(unibo_bp_cla_outbound_pdu_get_ecos(pdu),
&temp_ecos);

    if (error != UniboBP_NoError) {

       handleUniboBPError();

    }

    parseECOS(temp_ecos, &transmission.metadata.ecos);
```

The structure *transmission*, once completed, is added to the Unibo-LTP queue (implemented as a list) which contains the bundles already released for transmission by Unibo-BP flow control (based on contact start and stop times and nominal data speed, as in ION); they will be later extracted from the queue when allowed by Unibo-LTP flow control (e.g. when allowed by the max number of concurrent Tx session constraint).

5.2.2 The "*canReceiveDataFromUpperProtocol()*" function

This function, called by *_mainGetterFromUpperProctol* thread, removes one *transmission* from the LTP queue when permitted by LTP flow control. When allowed by the corresponding mutex, the function reaches the following line:

```
UniboBPTransmission* transmission = list_pop_front(&peer_data-
>transmissions, NULL);
```

The *transmission* structure, once popped from the queue, is copied into *specificData->current_pdu*. The last step is to perform the unlock of the resource.

5.2.3 The "*passedDataFromUpperProtocol()*" function

It is called when Unibo-LTP is ready to send a new block (i.e. to start a new session) and it is waiting for a bundle to be passed from the *canReceiveFromUpperProtocol()* function:

```
sem_wait(&receivedData->semIsDataPresent);
```

receivedData points to *specificData*, that contains the field "*current_pdu*", a *transmission* type structure, i.e. the type used to insert a serialized bundle with its metadata:

```
UniboBPReceivedData* specificData = receivedData->bundleOpaque;

UniboBPTransmission* transmission = specificData->current_pdu;
```

This structure is used to create the bigger "***dataToSend"*** structure (whose type is *UpperProtocolDataToSend*), the only one seen by the LTP core. This structure is formed by various parameters like the buffer with its length, but the most important one is the *metadata* field, which is directly copied from the transmission structure (the two metadata fields have the same type):

```
dataToSend->metadata=transmission->metadata;
```

For the sake of completeness, the *UpperProtocolDataToSend* structure is reported below:

```
typedef struct {

      char*                               buffer;

      long long int                       bufferLength;

      void  (*handlerSessionTerminated)(unsigned int,
LTPSessionResultCode);

      unsigned int                        dataNumber;

      LTPBundleMetadata metadata;

} UpperProtocolDataToSend;
```

After that, the *transmission* struct is freed and the waiting threads are unlocked in the *canReceiveDataFromUpperProtocol()* function. Metadata fields are now available outside of the Unibo-LTP interface to Unibo-BP in “*dataToSend.metadata*”. Currently, they are used in the “*spanBlock.c*” file as shown in next section.

5.3 USE OF THE NEW “BUNDLE METADATA” IN THE LTP CORE

The bundle metadata are used in the “***spanBlock.c***” file, which does not belong to the Unibo-BP interface but to the core of Unibo-LTP and thus its instructions are always compiled and executed, by contrast to Unibo-LTP interfaces to Unibo-BP and ION, which are alternative, i.e. compiled only on request.

This file was modified in order to take advantage of the new metadata information passed by the Unibo-BP interface in the *dataToSend* structure. As this code, however, must work even under ION, the interface to ION has been modified in order to provide the same metadata sub-structure (modifications are not reported in this thesis, as out of the scope). The difference is that in ION the only significative metadata field is the best effort ECOS flags. Note that other ECOS parameters are impossible to extract from ION API, which proves the advantage for research of Unibo-BP, and, more generally, to have full control of all components of DTN code, which is at the basis of the Unibo-DTN project.

By using “***dataToSend.metadata***”, it is possible to do 2 important operations:

- To Associate an LTP color to the ECOS fields, possibly keeping in mind of the re-Tx number. This can be seen as a form of QoS mapping between bundle and LTP protocols.
- Set the LTP Red session timer at the deadline specified in the metadata.

5.3.1 QoS mapping

The first operation is entirely made by the “*handleNewBlockToSend()*” function. If the default LTP link color is Green, the “***sessionColor***” is green as well: this is a safety measure as the default link could have been set to Green to cope with a unidirectional link, which would physically prevent any form of reverse LTP signaling, i.e. the use of any colors but Green. Otherwise, the default link color can be overridden by metadata settings, as shown in the Table 1. The two last lines require a brief explanation. Orange is used only for first transmission, while if the same bundle is retransmitted, it is retransmitted in a Red session. This policy is in order to prevent possible consecutive losses of the same bundle when send as Orange, which may happen if the presence of a high Packet Erasure Rate and large bundles [Caini_2024].

When compiled for ION, the best effort ECOS flag is the only one which can be set to 1, all others, reliable and number of re-Tx are always zero. As a result, only the first and third row are meaningful. In other words, when the default link is red, in ION all bundles are sent as Red, unless the best-effort flag is not set to 1. This is the normal ION behavior, except that Unibo-BP has the concept of default link color, which ION lacks. See [Bisacchi_2022].

"best effort" ECOS flag	**"reliable" ECOS flag**	**Number of reTx**	**LTP Color**
0	0		Default
1	0		Green
0	1		Red
1	1	0	Orange
1	1	>0	Red

Table 1 - Mapping of bundle QoS to LTP colors (when the default link color is not Green) (from [Caini_2024])

5.3.2 Set a session timer

The second operation starts in "*handleNewBlockToSend()*" where the residual life of the bundle to be transmitted by the current session (here called "*msleft*") is calculated:

```
msLeft = 1000*dataToSend.metadata.deadline.tv_sec-nowInMillis();

if (msLeft<0) msLeft=0;
```

At the end of "*handleNewBlockToSend()*" is called "*createNewTxSession()*", to which "msleft" is passed as last parameter, where the actual setting of the Red session timer is made. For this reason, inside *createNewTxSession()* the passed "msleft" is renamed as "***redSessionTime***". Now, if *sessionColor* is equal to "Red", a session timer is set to the minimum between the desired "redSessionTime" and the max allowed duration of a

Red Tx session (a parameter of Multicolor LTP). The code is the following. A few lines have been omitted as not essential for the comprehension of current topic:

```
      if(sessionColor == Red){ //Start red Tx session timer and save
timerID in session->TxSession.timerCancelTxSession

            if (MAX_TX_RED_SESSION_TIME<redSessionTime)
redSessionTime=MAX_TX_RED_SESSION_TIME;

few omitted lines…

session->TxSession.timerCancelTXSession = startTimer(redSessionTime, 1,
timerHandlerSessionTimeoutNotif , CS_notif);

      }
```

If the red Tx session is not completed before in the due time (redSessionTime), the session is closed. This prevent LTP from wasting time in trying to deliver bundles destined to be immediately discarded because their lifetime has elapsed, which was what we wanted to avoid. Note that in the presence of bundle aggregation (a feature not implemented in Unibo-BP), the timer should be set to the minimum lifetime of bundle aggregated into the same block, thus making less effective the solution implemented.

6 Adding the Metadata extension block to Unibo-BP

In this chapter, the adding of the Metadata extension [RFC6258] to Unibo-BP [Unibo-BP] and to the Unified API interface to Unibo-BP is presented.

6.1 Background information

A bundle is composed by two compulsory blocks, the Primary block and the Payload block, and possibly by several Extension blocks. The Extension and the Payload blocks follow the Canonical Bundle Block format. Extensions are of different types, some of them are defined in [RFC9171], such as “Previous Node”, “Bundle Age”, “Hop Count”, other defined in RFC drafts, as “ECOS”, other designed by Unibo in support of Unibo-CGR, as “CGR Route” and “Geographical Route”. The **Metadata** extension was originally defined for BPv6 in [RFC6258] but never updated to BPv7. In this thesis I have designed an updated version, essentially by moving from the SDNV to the CBOR format some numerical values, and inserted it in Unibo-BP.

The Metadata extension is used to carry additional information referring to the bundle payload, e.g. the subject of a photo contained in the bundle payload. This is why is called “***metadata***”. These metadata are usually inserted by the source not by the BP agent, which differentiates this extension form the others. As the RFC 6258 says, this extension has been designed to support a variety of metadata types and formats, thus it must contain a field called “*metadata type*”. Indeed, the only restriction imposed on metadata is that metadata type and format be predefined and registered if public, so that the specific format can be parsed and processed by DTN nodes and applications that support the type.

As also specified at the beginning of the thesis, one of the main goals of this thesis was to add the Metadata extension to Unibo-BP; this required two orders of modifications:

- Inside Unibo-BP code, including its public API (Unibo-BP block in dark blue) in Figure 3.
- In the Unified-API interface to Unibo-BP (Unibo-BP block in pale blue, called “*bp_unibo*”) in Figure 3.

6.1.1 Metadata extension format

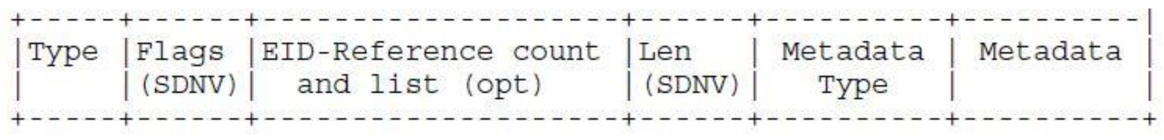
Extensions blocks, and thus the Metadata extension block, follows the Canonical Bundle Block format, defined for BPv7 in [RFC9171]. The main difference with BPv6 is the use of Concise Binary Object Representation (CBOR) encoding [RFC8949] instead of the SDNV (Self Delimiting Numerical Values) [RFC5050]). The Canonical Bundle Block format for BPv7 requires the following fields:

- Block-type code (1 byte),
- CRC-type code (1 byte),
- Block number (1 byte),
- Block Processing Control flags,
- Block-type-specific data (in CBOR the length is added when encoded),
- CRC (if the CRC-type is different from 0).

A Metadata extension has type **0x08** and the block-type-specific data consists of two fields:

- **Metadata Type**: an integer that indicates which kind of metadata are in the metadata field; in RFC 6258 only the metadata type "**URI**" has been defined, with value 1; other types of metadata can be defined in the future. Pay attention to not confuse the metadata extension type (always 0x08), which defines an extension as a metadata extension, with the metadata type (an integer, equal to 1 for the only type defined so far), which defines which kind of metadata are in the metadata field of the metadata extension.
- **Metadata**: contains the metadata, formatted according to the metadata type. For the Metadata type URI, in the absence of an RFC for BPv7 to be used as a normative reference, we decided that in Unibo-BP this field consists of two sub-fields: an integer representing the length (*Metadata length* in the following) of the string containing the URI, and the string itself (*Metadata string* in the following), mimicking the internal representation used by ION. You can observe it in Figure 17.

The structure of the Metadata extension, as defined in RFC 6258 for BPv6, is shown in Figure 16. Note the presence of "*EID-Reference count*" and "*Len*" fields, derived by the BPv6 format for Canonical Bundle Blocks. The latter represented the **aggregate length** of next blocks, i.e. Metadata Type and Metadata. Both of them have been dropped in BPv7. For BPv7 it is necessary to replace the SDNV format with CBOR and to add the CRC-type. If the CRC type is more than 0, the CRC is calculated by the BPA and added when the extension is serialized.



```
+-----+------+--------------------+------+----------+----------|
|Type |Flags |EID-Reference count |Len   | Metadata | Metadata |
|     |(SDNV)|  and list (opt)    |(SDNV)|   Type   |          |
+-----+------+--------------------+------+----------+----------+
```

Figure 16 - Metadata extension format in BPv6 (from [RFC6258])

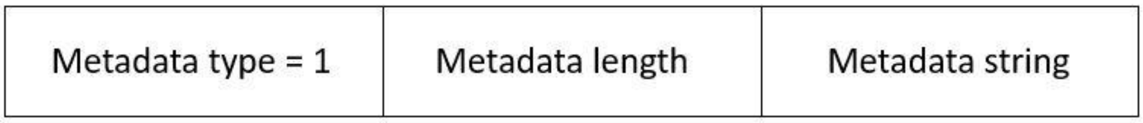


Figure 17 - Block-type-specific data format used for the type URI, as implemented in Unibo-BP

6.2 Metadata extension block implementation in Unibo-BP

6.2.1 Extension block class: syntax and explanation

In Unibo-BP extensions are implemented as described below.

The ***ExtensionBlock*** class in Unibo-BP defines a set of virtual functions that must be overridden by derived classes to implement specific behaviors.

The ***ExtensionBlockOutputMask*** type is a bitmask indicating the information within the extension block that has been modified by the function. Depending on the activated bits, specific operations are executed on the extension block. These operations include, for example, the re-serialization of block-type-specific data when the contents of the extension block have been modified.

The function below, invoked during the insertion of the extension block into a bundle, is currently triggered solely at the bundle creation, with the following syntax:

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

Potential future policies might necessitate the insertion of specific extension blocks into bundles received from other nodes, leading to various subcases, for example when the bundle is inserted into the BP queue of the neighbor to which it is to be sent, or when the bundle is extracted from the BP queue because it can be sent to the neighbor.

6.2.2 New files added to create the Metadata extension

Unibo-BP code is composed by many directories, clearly organized. Extensions are in the "***bp/bundle/extension***" directory. To add the new Metadata extension, I created the classes (.cpp and .h files) "***MetadataBlock***" and "***Metadata***" in the *extension* and in the *bundle* directory, respectively. Then I added a new "***Metadata***" class in the directory "***api/src/Unibo-BP***"; this class contains wrapper of C++ functions, to be used by other programs written in C, like the Unified API functions. To avoid any possible misunderstanding with the other classes added, the last class will be referred as "***MetadataAPI***" in the next sections.

In the following, I will describe these classes.

6.2.2.1 The "MetadataBlock" class

This class is "derived" from the *ExtensionBlock* class. As that, it contains three functions that override the homonymous one in the *ExtensionBlock* class: "***create()***", "***serialize_block_type_specific_data_content()***" and the "***deserialize_block_type_ specific_data_content()***".

The "*create()*" function also set the block-processing control flag to "*block must be replicated in every fragment*". The other two, code and decode the *block-type-specific data* in CBOR format by using the encode/decode functions defined in the "*Metadata*" class.

6.2.2.2 The "Metadata" class

This class contains the functions that operate on the *block-type-specific data* of a Metadata block of type URI, such as those to get and set the *metadata type*, *metadata length* and *metadata string* (see Figure 17), and the functions used to encode/decode each item in CBOR format.

In the “*Metadata.h*” file the “***MetadataType***” is defined as an 8-bit integer (*uint8_t*) enumerative, which at present consist of only two elements:

```
enum class MetadataType : std::uint8_t {

        unknown                  = 0x00,

        uri                      = 0x01

};
```

The “***metadata_length***” is defined as a 64-bit unsigned integer and the “***metadata_string***” as a string.

In the header file, there are also the prototypes of the functions used to set and the triple metadata type, length, and string. As an example, hereafter there are the prototypes of the get/set functions of the metadata type:

```
void set_metadata_type(MetadataType type_) noexcept { this-
>metadata_type = type_; }

[[nodiscard]] MetadataType get_metadata_type() const noexcept { return
this->metadata_type; }
```

The related “*Metadata.cpp*” file, instead, defines the encode/decode functions. It is worth noting that the “*encoder.encode_header_fixed_array(3)*” string means that the array which forms the *block-type-specific data* for the URI type is composed by 3 elements. The other lines use the encoder/decoder of the appropriate type to obtain the requested value, i.e. “*uint*” for the metadata type and metadata length, and “*string*” for the metadata string:

```
void Metadata::encode(io::CborEncoder &encoder) const {

        encoder.encode_header_fixed_array(3);

        encoder.encode_uint(math::to_underlying(metadata_type));

        encoder.encode_uint(metadata_length);

        encoder.encode_string(metadata_string);

}
```

The “*decode*” operations are shown below. It is worth noting that they are “unpacked” (i.e. divided into smaller operations) to facilitate the debug of values assumed by the variables.

```
void Metadata::decode(io::CborDecoder &decoder) {

        [[maybe_unused]] auto should_be_3 =
decoder.decode_header_fixed_array();

        uint64_t metadata_type_64 = decoder.decode_uint();

        MetadataType metadata_type = to_metadata_type(metadata_type_64);

        set_metadata_type(metadata_type);


        uint64_t length = decoder.decode_uint();

        set_metadata_length(length);


        std::string metadata_string = decoder.decode_string();

        set_metadata_string(metadata_string);

}
```

6.2.3 Minor modifications to Unibo-BP’s pre-existent files

A few minor modifications were necessary to pre-existent files (either .cpp or .h or both). First, let us briefly recall modifications to files in the “***bp/bundle***” directory:

- The “***BlockTypeCode***” files: the .h file contains the collections of block type codes processable by Unibo-BP. The metadata block has code 8 [RFC6258], so in the enumerative class *BlockTypeCode* we added a new line with this code, and the “***BlockTypeCode vector***” has to be incremented from 8 to 9 because of the adding of the new value. The related .cpp file, instead, has to be modified by adding in every function a line corresponding to the metadata where other extensions were listed.

- The “***Bundle.h***” file: this is the Bundle class. The struct “***BundleCache***” was modified (according to the other elements) by adding the new extension Metadata.
- The “***ExtensionBlock.cpp***” file: In the “***spawn()***” function new case which regards the metadata extension was added: now when a new block type code coincides with the metadata one, a new block will be created.
- The “***ExtensionManager.cpp***” file: This file is crucial. If the “**creation rule**” for the metadata extension is not set (or, worse, it is set as false), the metadata block will never be created even if requested. Therefore, the creation rule for metadata must be set as true, which, on the other hand, it means that the metadata block will always be created even if not requested (so it will be empty).

Then, the modifications regarding files contained in the “***ipc/src/message/user***” directory:

- The “***bundle_transmission.cpp***” and “***bundle_reception.cpp***” files: since the metadata block has to be processed by other applications other than Unibo-BP, the metadata block “***encode***” and “***decode***” operations have to be added. The former was inserted into the “*encode_payload_bundle_send()*” and the “*encode_payload_bundle_receive()*” functions; the latter into the “*decode_payload_bundle_send()*” and the “*decode_payload_bundle_receive()*” functions.

6.2.4 Unibo-BP Metadata API

External BP applications establish connections with Unibo-BP BPA through a public API that consists of public functions with prototypes following the C syntax, to allow programs written in C to seamlessly integrate and utilize them. The public functions within the API act as wrappers for the C++ internal methods. To add the metadata extension to the API, it was necessary to add the new “***MetadataAPI***” class, and to modify other files.

6.2.4.1 The "MetadataAPI" class

This class is used to manage the passage of metadata between the Unibo-BP core and Unibo-BP applications or the functions of the Unified API library.

In the header file, contained in the "***include/Unibo-BP***" directory as all header files of the API classes, a clone of the *MetadataType* enumerative defined in the *Metadata* class (described above) is defined. It is named ***UniboBPMetadataType*** and it belongs to the API, i.e. it is the only known by the applications:

```
typedef enum UniboBPMetadataType {

    UniboBPMetadataType_unknown                     = 0x00,

    UniboBPMetadataType_uri                         = 0x01

} UniboBPMetadataType;
```

The other two fields of *block-type-specific data*, i.e. *metadata_length* and *metadata_string*, are defined as an integer (uint64_t) and as a string, respectively.

Then, there are the declarations of the functions which create, destroy, and clone the metadata block and, last, of the functions which set and get the triple metadata type, metadata length and metadata string. This .h file is copied in the include directory of the system when the command "*sudo make install*" is executed, after the usual "*make*". This because Unibo-BP provides dynamic instead of static libraries.

All these functions are defined in the .cpp file in "***api/src/Unibo-BP***", as all .cpp files of API classes. They use "***get_c()***" and "***get_cxx()***" conversion functions (from C to C++ and vice versa) defined for code clarity in a separate file, called "***conversion***", in the same directory of "*MetadataAPI.cpp"*

For example, regarding the metadata type, in the .cpp file we have the "***unibo_bp_metadata_set_type()***", which means that the metadata type passes from the API to Unibo-BP itself. In particular, the "*set_metadata_type()*" function is the one defined in the "*Metadata*" class defined above and it is used to manage the volatile data.

```
void unibo_bp_metadata_set_type(UniboBPMetadata metadata,
UniboBPMetadataType type) {
    if (!metadata) return;

    get_cxx(metadata)->set_metadata_type(get_cxx(type));

}
```

The dual operation is performed by the "***unibo_bp_metadata_get_type()***" function, which returns to the application (i.e. Unified API) the metadata type set before.

```
UniboBPMetadataType unibo_bp_metadata_get_type(ConstUniboBPMetadata
metadata) {

    if (!metadata) return UniboBPMetadataType_unknown;

    return get_c(get_cxx(metadata)->get_metadata_type());

}
```

6.2.4.2 The "InboundBundle" and "OutboundBundle" classes

The ***InboundBundle*** class defines the "*get*" functions to read the fields (metadata type, metadata length, and metadata string) of the received bundles; the ***OutboundBundle*** the corresponding "*set*" functions for the bundles to be sent.

As an example, the function "*unibo_bp_inbound_bundle_metadata_get_type()*", is reported below. Note that to get the metadata type, the function "*get_metadata_type()*" which belongs to the "*Metadata*" class defined in Unibo-BP is used (*bundle.cache.metadata.get_metadata_type()*).

```
UniboBPMetadataType
unibo_bp_inbound_bundle_metadata_get_type(ConstUniboBPInboundBundle
bundle) {

    if (!bundle) return UniboBPMetadataType_unknown;

    try {

        return get_c(get_cxx(bundle)-
>bundle.cache.metadata.get_metadata_type());

    } catch (const std::exception& ignore) {

        return UniboBPMetadataType_unknown;

    }

}
```

Analogously, in the “*OutboundBundle.cpp*” file, for the dual function we have:

```
void unibo_bp_outbound_bundle_metadata_set_type(UniboBPOutboundBundle
bundle, UniboBPMetadataType type) {

    if (!bundle) return;

    try {

        get_cxx(bundle)-
>bundle.cache.metadata.set_metadata_type(get_cxx(type));

    } catch (const std::exception& ignore) {}

}
```

6.2.4.3 Minor modifications to the “BlockTypeCode” API class

Here only a line has been added to the .cpp file, to indicate that there could be another case for the new metadata block (as made in the previous Unibo-BP extension files).

6.3 Modification to the Unified API interface to Unibo-BP

In Unified API, the structure that represents a bundle is quite complex and it would be lengthy to provide all details. What matters is that we have a nested structure for extensions, consisting of two fields: one is the *block-type-specific data,* and the other is its length (i.e. the “*Len*” field of Canonical bundle blocks in BPv6, see Figure 16).

The user can insert from the terminal two options: “***–mb-type***” and “***–mb-string***”. Let us assume for clarity that the user sets “1” for the type and “yellow” for the string. These two options are then concatenated into a string, such as “*1/yellow*” and this string is saved in the *block-type-specific data* field of the extension structure. When the bundle is sent, it is necessary to parse/copy this field into the variables used by a specific implementation and vice versa. Details on how this process is performed are provided in the next subsections.

6.3.1 The “*unibo-bp*” directory structure in Unified API

The Unified API interface to Unibo-BP belong to the third layer (“*bp*”) of the Unified API and as that it is included in the Unified API code (see Figure 3). All the code is contained in the “*unibo-bp*” directory which is organized as shown in Figure 18:

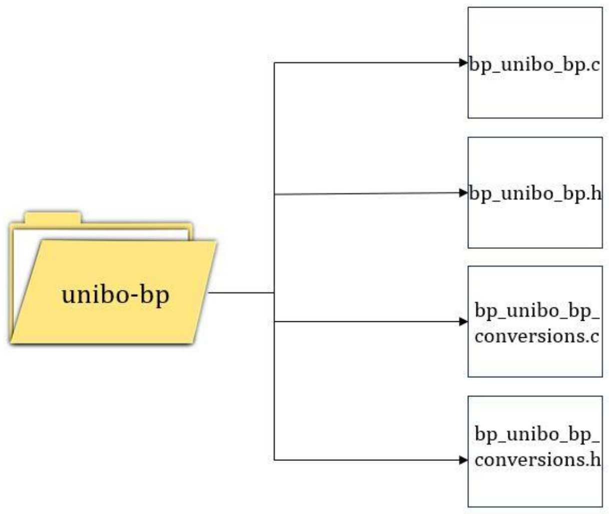


Figure 18 - The Unified API "unibo-bp" directory structure

The "*unibo-bp*" folder, as the majority of the other "*bp*" folders, includes a pair of files for type conversion (*bp_unibo_bp_conversion.c/h*) and another pair for API functions (*bp_unibo_bp.c/h*). Conversion functions have the prefix "***al_to_unibo_bp***" when converting from Unified API types to Unibo-BP types, and "***al_from_unibo_bp***" when performing the reverse conversion.

6.3.2 The "*bp_unibo_bp_conversions*" files

Metadata blocks are a particular case of extension blocks, and thus they in Unified API they use the "***al_types_extension_block***" structure that is a sort of union of the Canonical Bundle Block format in BPv6 and BPv7, as shown below:

```
typedef struct al_types_extension_block {
      uint8_t block_type_code;//RFC 5050 and RFC9171 require 1B
      uint64_t block_processing_control_flags;//RFC9171 requires 8B
      uint8_t crc_type; //only bpv7;
      struct {
            uint64_t block_type_specific_data_len;
            char *block_type_specific_data;
      } block_data;
} al_types_extension_block;
```

The “union” of both formats is due to the fact that Unified API is compatible with both BPv6 and v7.

The field “***block_type_specific_data***” is the only one that really matters. As said, when a bundle is sent it contains the concatenation of metadata_type and metadata_string, e.g. “1/yellow”. It is therefore necessary to have a conversion from this string to the three corresponding variables used by Unibo-BP, i.e. metadata_type, metadata_length, and metadata_string. In more general terms, we need to convert the *block-type-specific data* from the Unified API to the Unibo-BP format.

In “*bp_unibo_bp_conversions.h*” we have the function prototype of “*al_to_unibo_bp_extensions_metadata()*”, which carry out this conversion:

```
UniboBPMetadataType al_to_unibo_bp_extensions_metadata(uint32_t
extension_number, al_types_extension_block *extension_blocks,
UniboBPMetadataType metadata_type, uint64_t metadata_length, char
*metadata_string);
```

The function above takes as input the following arguments:

- ***extension_number***: an integer, whose name is a bit ambiguous. It means the “**number of extension blocks of a bundle in Unified API**”; In practice, it will always be either 0 or 1. A higher would be incompatible with the Unibo-BP implementation of metadata developed in this thesis.
- ***extension_blocks***: the Unified API structure we want to convert.

And provide in outputs the triple used by Unibo-BP (the last 3 arguments). The function is described below.

6.3.2.1 The “al_to_unibo_bp_extensions_metadata()” function

It starts by declaring a character array (i.e. a string) named “***backup_specific_data[]***”, used to store a copy of the ***block-type-specific data*** (e.g. “1/yellow”). To separate the different fields it is necessary to “**tokenize**” the string by using the “*strtok()*” function with “/” as a delimiter. Each time this function is used, a new token (field) is extracted from the original string. First, we obtain the metadata type in string format (“1”), which is converted into an integer by the "*atoi()*" function, and finally assigned to the "***UniboBPMetadataType***" variable.

Then it is extracted the metadata string (yellow) and its length calculated. They are assigned to "***metadata_length***" and "***metadata_string***" both of Unibo-BP.

6.3.2.2 The "al_from_unibo_bp_extensions_metadata()" function

This function performs the dual operation: from the Unibo-BP triple metadata_type, metadata_length and metadata_string, it is obtained Unified API type *block-type-specific data* (e.g. 1/yellow). The result is obtained, by concatenating the metadata_type and metadata_lenght (and the "/" delimiter) by means of the "*strcat()*" function.

6.3.3 The "*bp_unibo_bp.c*" file

This file contains several functions, including the send and the receive functions that are the only of interest here. In these functions, we added the check for the presence of the metadata extension; if the result is positive the conversion functions described in the previous section are called. The addition was easy, and the interested reader can directly inspect the code.

7 CONCLUSIONS

This thesis had two main purposes, both related to Unibo-BP, the Unibo implementation of Bundle Protocol version 7 (BPv7, RFC 9171): first, to improve the interface between Unibo-LTP, the Unibo implementation of the LTP protocol and Unibo-BP; second, to add the Metadata bundle extension to Unibo-BP and to make this extension available to the Unified API library.

A pre-requisite to fulfill both aims, was the ability of debugging Unibo-BP, Unibo-LTP and Unified API under the same IDE. The choice was in favor of Visual Studio Code, for the better handling of CMake used to build Unibo-BP, written in C++ 20 (Unibo-LTP and Unified API, by contrast are written in C). Two chapters of the thesis, third and fourth, are devoted to detailed instructions on the use of this tool for the mentioned aims. The hope is that they can be of help to future students destined to work with the Unibo-BP ecosystem.

The fifth chapter is focused on the first aim. In it is shown how, thanks to the improvements introduced in the thesis, it is possible to map the QoS seen at BP with that offered by LTP, a novel feature that could pave the way to future research.

The sixth chapter is devoted to the second aim and is divided into two parts. The former deals with the introduction of metadata extension in Unibo-BP and in its API; the latter with its use in Unified API. It is worth noting that the new Metadata extension is not fully compliant with the only specification about bundle metadata available so far, i.e. RFC 6258, because this was released in 2011, when the current BP version was 6. This required us to introduce some original changes in the extension format to make it compatible with BPv7 and thus with Unibo-BP.

Once inserted the Metadata extension in Unibo-BP, the support to the Metadata was also added to Unified API, an abstraction layer that makes DTN applications based on it independent of the specific BP implementation. The support of metadata was already present in Unified API upper layers, but it was necessary to introduce it in the Unified API interface to Unibo-BP, at the lowest layer.

All modifications were debugged and tested before being incorporated in development branches of the three projects, hosted by the GitLab platform, from which they can be downloaded.

In conclusion, this thesis was the first to introduce modifications and improvements to Unibo-BP (and related software), but hopefully not the last; the hope is that it could pave the way to a series of future additions, aimed to keep the University of Bologna at the forefront of DTN research.

References

[µD3TN]	µD3TN web site: https://d3tn.com/ud3tn.html; code: https://d3tn.com/ud3tn.html
[Alessi_2019]	N.Alessi, "*Hierarchical Inter-Regional Routing Algorithm for Interplanetary Networks*". Master's thesis in Computer Science Engineering, University of Bologna 2019, https://amslaurea.unibo.it/17468/.
[B-DTN7]	L. Baumgärtner, J. Höchst and T. Meuser, "*B-DTN7: Browser-based Disruption-tolerant Networking via Bundle Protocol 7*", in Proc. of ICT-DM conference, Paris, France, 2019, pp. 1-8, doi: 10.1109/ICT-DM47966.2019.9032944. https://github.com/dtn7/dtn7-rs
[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., vol.40, no.6, pp. 428- 444, Nov.-Dec.2022. doi:10.1002/sat.1421.
[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.
[Caini_2013]	C. Caini, A. d'Amico and M. Rodolfi, "*DTNperf_3: a Further Enhanced Tool for Delay-/Disruption- Tolerant Networking Performance Evaluation*", in Proc. of IEEE Globecom 2013, Atlanta, USA, December 2013, pp. 3009 - 3015., Web site: https://gitlab.com/dtnsuite/dtnperf
[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.
[Caini_2023]	C. Caini and L. Persampieri, "*Unibo-BP: a new Bundle Protocol Implementation*", in Proc. of IEEE Wisee 2023, Aveiro, Portugal, Sept. 2023
[Caini_2024]	C. Caini and L. Persampieri, "*Design and features of Unibo-BP, the Unibo implementation of the DTN Bundle Protocol*", IEEE Journal of Radio Frequency Identification, accepted for publication, 2024.

[CCSDS_BPV6]	CCSDS 734.2-B-1 "*CCSDS Bundle Protocol Specification*", CCSDS Blue Book, Issue 1, Sept. 2015.
[CCSDS_LTP]	CCSDS 734.1-B-1, "*Licklider Transmission Protocol (LTP) for CCSDS*". recommended standard, Blue Book, 2015.
[CCSDS_SABR]	CCSDS 734.3-B-1, "*Schedule-Aware Bundle Routing*", recommended standard, Blue Book, July 2019.
[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/>.
[Draft_ECOS]	S. Burleigh, F. Templin, "*Bundle Protocol Extended Class of Service (ECOS)*", Internet draft, May 2021. Work in progress.
[DTN]	V. Cerf, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss "*Delay-Tolerant Networking Architecture*", Internet RFC 4838, Apr. 2007.
[DTN_IEEE]	A. Schlesinger, B. M. Willman, L. Pitts, S. R. Davidson and W. A. Pohlchuck, "*Delay/Disruption Tolerant Networking for the International Space Station (ISS)*", 2017 IEEE Aerospace Conference, Big Sky, MT, 2017, pp. 1-14. doi: 10.1109/AERO.2017.7943857.
[DTN_Req]	CCSDS 734.0-G-3 "*Rationale, Scenarios, and Requirements for DTN in Space.*" CCSDS Green Book, Issue 3, Jul. 2014.
[DTNME]	DTNME (DTN Marshall Enterprise) code: https://github.com/nasa/DTNME
[DTNsuite]	DTNsuite website: https://gitlab.com/dtnsuite
[ECLSA]	ECLSA code: https://gitlab.com/unibo-dtn/ECLSA.
[HDTN]	R. Dudukovich, B. LaFuente, A. Hylton, B. Tomko and J. Follo, "*A Distributed Approach to High-Rate Delay Tolerant Networking Within a Virtualized Environment*", in the Proc. of 2021 IEEE CCAAW, 2021, pp. 1-5, doi: 10.1109/CCAAW50069.2021.9527297. HDTN code: https://github.com/nasa/HDTN

[IBR-DTN] S. Schildt, J. Morgenroth, W.-B. Pottner, L. Wolf, "*IBR-DTN: A lightweight, modular and highly portable Bundle Protocol implementation*", in Electronic Communications of the EASST, vol. 37, pp. 1-11, Jan. 2011. Code: https://github.com/ibrdtn/ibrdtn

[ION] S. Burleigh, "*Interplanetary Overlay Network: An Implementation of the DTN Bundle Protocol*", in the Proc. of 4th IEEE Consumer Commun. and Networking Conference, 2007, pp. 222-226, doi: 10.1109/CCNC.2007.51. Code: http://sourceforge.net/projects/ion-dtn/.

[IRR] O. De Jonckère and J. A. Fraire, "*Inter-Regional Routing in Interplanetary Networks with Shortcuts and Contact Passageways*", in Proc. of WiSEE conference, Aveiro, Portugal, 2023, pp. 87-92, doi: 10.1109/WiSEE58383.2023.10289358.

[Persampieri_2023] L.Persampieri, "*Unibo-BP: an innovative free software implementation of Bundle Protocol Version 7 (RFC 9171)*", Master's thesis in Computer Science Engineering, University of Bologna, Feb. 2023, https://amslaurea.unibo.it/27740/

[RFC5050] K. Scott, S. Burleigh, "*Bundle Protocol Specification*", Internet RFC 5050, Nov. 2007.

[RFC5325] M. Ramadas, S. Burleigh and S. Farrell, "*Licklider Transmission Protocol – Motivation*", Internet RFC 5325, Sept. 2008.

[RFC5326] M. Ramadas, S. Burleigh and S. Farrell, "*Licklider Transmission Protocol – Specification*", Internet RFC 5326, Sept. 2008.

[RFC6258] S. Symington, "*Delay-Tolerant Networking Metadata Extension Block*", RFC 6258, May 2011.

[RFC7242] M. Demmer, J. Ott, S. Perreault "*Delay-Tolerant Networking TCP Convergence-Layer Protocol*", RFC 7242, June 2014

[RFC8949] C. Bormann, P. Hoffman "*Concise Binary Object Representation (CBOR)*", RFC 8949, Dec. 2020.

[RFC9171] S. Burleigh, K. Fall, E. Birrane, "*Bundle Protocol Version 7*", Internet RFC 9171, Jan. 2022.

[RFC9172]	E. Birrane, and K. McKeever, "*Bundle Protocol Security (BPSec)*", RFC 9172, Jan. 2022.
[RingRoad]	M. Feldmann, J. A. Fraire, F. Walter and S. C. Burleigh, "*Ring Road Networks: Access for Anyone*", in IEEE Communications Magazine, vol. 60, no. 4, pp. 38-44, April 2022, doi: 10.1109/MCOM.001.2100835.
[Unibo-BP]	Unibo-BP code: https://gitlab.com/unibo-dtn/unibo-bp
[Unibo-CGR]	Unibo-CGR web site: https://gitlab.com/unibo-dtn/unibo-cgr.
[Unibo-DTN]	Unibo-DTN code: https://gitlab.com/unibo-dtn.
[Unibo-LTP]	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*" in Proc. Of ASMS/SPSC conference, Graz, Austria, 2022, pp. 1-6, doi: 10.1109/ASMS/SPSC55670.2022.9914734.
[Virtualbricks]	P. Apollonio, C. Caini, M. Giusti and D. Lacamera, "*Virtualbricks for DTN satellite communications research and education*", in Proc. of PSATS 2014, Genoa, Italy, July 2014, pp. 1-14.
[Wireshark]	Wireshark web site: https://www.wireshark.org/
[Zappacosta_2023]	C. Caini, T. de Cola, A. Shrestha and A. Zappacosta, "*LTP Performance on Near Earth Optical Links*", in IEEE Transactions on Aerospace and Electronic Systems, Early access Oct. 2023 doi: 10.1109/TAES.2023.3322392