

ALMA MATER STUDIORUM - UNIVERSITY OF BOLOGNA

Department of Computer Science and Engineering

Master's degree in Computer Engineering

MASTER'S THESIS

In

Protocols and Architectures for Space Networks M

Implementation of CCSDS protocols for space networks.

Candidate:

Danilo Cavallini

Supervisor:

Prof. Carlo Caini

Co-supervisors:

Ph.D. Felix Flentge

Camillo Malnati

Academic Year 2024/2025

ABSTRACT

The advent of the space age has created a growing need for robust communication infrastructures capable of operating reliably in the harsh conditions of interplanetary space. Since the early 2000s, significant research has focused on developing software infrastructures and network architectures suited for such environments. This effort led to the development of the Delay-/Disruption-Tolerant Networking (DTN) architecture, centered around the Bundle Protocol (BP). Within this context, strong standardization is essential. Therefore, numerous space agencies collaborate through the Consultative Committee for Space Data Systems (CCSDS), an international body that develops standards to ensure interoperability and cross-support among space agencies and commercial entities. This thesis presents the design, development, and integration of several DTN networking standard components within the European Space Agency's (ESA) implementation of the Bundle Protocol, known as ESA BP version 3 (ESA BPv3), in Java. This work was carried out during a six-month internship at ESA's European Space Operations Centre (ESOC) in Darmstadt, Germany. The primary objective was to extend ESA BPv3 with new functionalities and new supporting libraries to enhance its flexibility, modularity, and compliance with CCSDS and IETF (Internet Engineering Task Force) standards for space communication. Four key contributions were developed: the File Convergence Layer Element (CLE), the CCSDS Generic Packetiser (GP), the DTN Anycast ("iac") scheme, and the CADU and CLTU CLEs. The new software significantly enhances ESA BPv3's capability to integrate both terrestrial and space communication protocols within a unified DTN framework. This work aims to upgrade the support of DTN architectures for space communication, foster interoperability among agencies such as ESA and NASA, and pave the way for more resilient, adaptable interplanetary networking systems.

CONTENTS

Abstract.....	I
Contents.....	III
1 Introduction	6
1.1 Challenged Networks.....	6
1.2 The DTN Architecture.....	6
1.3 Convergence Layer and Convergence Layer Adapters	7
1.4 An overview of the work done	9
1.4.1 Structure of the thesis.....	10
2 ESA BP v3	11
2.1 Bundle Node.....	11
2.2 ESA Implementation	12
2.3 Application Agent.....	13
2.4 Convergence Layers	14
3 ESA BP v3 building and configuration	17
3.1 Building the ESA BP v3 project.....	17
3.2 Node configuration.....	18
3.3 Configuration files.....	19
3.3.1 Node configuration	20
3.3.2 Convergence Layer configuration	20
3.3.3 Routing Configuration	23
3.3.4 Daemon File	25
3.3.5 GRPC Test Client (Optional)	25
3.4 Start a Bundle Node and a CLI	26

4	The “File CLE”	28
4.1	Sender Part	29
4.1.1	Sender Sequence Flow	29
4.2	Receiver Part	30
4.3	Configuration	32
5	CCSDS Generic Packetiser	34
5.1	CCSDS Space Data Link Protocols	36
5.2	CCSDS Packets	37
5.2.1	Space Packet Protocol	37
5.2.2	Encapsulation Packet Protocol	38
5.3	Configuration Overview	39
5.4	Final System Design	42
5.5	Functionalities	44
5.5.1	CCSDS Packet Creation	44
5.5.2	CCSDS Frame Creation	45
5.5.3	CCSDS Packet Extraction	46
5.6	Performances	48
5.7	The GP CLE	50
6	DTN Anycast	51
6.1	Usages of DTN anycast	52
6.2	Ipn Scheme Structure	55
6.3	The “iac” scheme	56
6.3.1	Group Number	57
6.4	Routing	58
6.4.1	Contact Graph Routing (CGR)	58
6.4.2	Proposed Possible Solution	62

6.4.3	Examples of DTN Anycast Contact Graph Routing	63
6.4.4	Second Possible Solution	74
7	The CADU and CLTU CLEs.....	75
7.1	Channels Coding Structure	76
7.2	CADU CLE	78
7.2.1	TM Synchronization and Channel Coding.....	78
7.2.2	Design and Implementation	80
7.3	CLTU CLE	84
7.3.1	TC Synchronization and Channel Coding	84
7.3.2	Design And Implementation.....	87
8	Conclusions.....	91
	Bibliography	93

1 INTRODUCTION

1.1 Challenged Networks

Challenged networks represent communication environments that are significantly different from the assumptions of traditional Internet architectures. These networks are typically affected by long propagation delays, intermittent or asymmetric connectivity, high bit error rates, and limited bandwidth resources. For example, the interplanetary and deep-space environment.

Conventional end-to-end protocols such as TCP/IP assume relatively stable links and continuous connectivity between sender and receiver. However, in challenged environments, end-to-end paths may never exist at a single point in time, making these traditional IP protocols ineffective. As a result, new approaches are required to provide reliable and efficient communication despite long disconnections and unpredictable link availability.

To address these limitations, research efforts pioneered by the CCSDS (Consultative Committee for Space Data Systems) and the IETF (Internet Engineering Task Force) have led to the definition of DTN (Delay/Disruption Tolerant Networking) [RFC4838] and all the DTN space-related protocols [CCSDS_PROTOCOLS].

1.2 The DTN Architecture

The Delay/Disruption Tolerant Networking (DTN) architecture provides a general-purpose framework for ensuring reliable data exchange across networks characterized by frequent disconnections or high latency, formally described in the RFC4838 [RFC4838] ("Delay-Tolerant Networking Architecture").

The cornerstone of this architecture is the **Bundle Protocol (BP)**, currently standardized as Bundle Protocol v6 in [RFC9171] and CCSDS 734.2-B-1 [CCSDS_BP].

DTN's layered structure integrates with existing transport or convergence-layer protocols such as TCP, UDP, and LTP (Licklider Transmission Protocol) [RFC5326] by introducing a new layer above the IP transport layer and below the application layer,

the bundle protocol. BP introduces the **store-carry-and-forward** paradigm, where intermediate nodes (called bundle nodes) store data, known in BP as bundles, persistently and forward them when links become available.

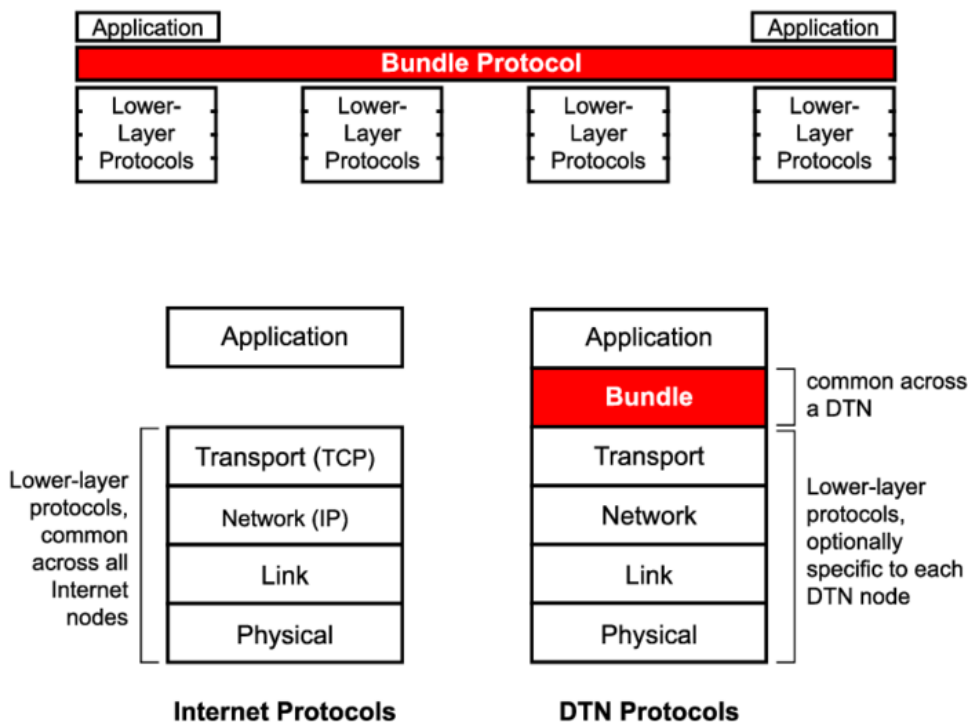


Figure 1.1 – Bundle Protocol Architecture

As depicted in Figure 1.1 [BP_ARC] between these nodes, is it possible that some hops do not support BP. In fact, the sending of a bundle can pass through non-compliant BP nodes, such as a common IP terrestrial router.

1.3 Convergence Layer and Convergence Layer Adapters

The Convergence Layer (CL) represents the set of communication protocols and mechanisms that operate beneath the Bundle Layer in the Delay/Disruption Tolerant Networking (DTN) architecture. As we can see from Figure 1.1, its primary role is to provide reliable data transmission between Bundle Protocol (BP) agents, regardless of the underlying network technologies or transport mechanisms used, by abstracting the complexity and heterogeneity of lower-layer protocols.

To achieve this abstraction, each implementation of the Convergence Layer provides a Convergence Layer Adapter (CLA). This software interface enables the Bundle Protocol to interact consistently with different transport protocols. The CLA manages connection setup, data transfer, segmentation, acknowledgment, and retransmission, depending on the characteristics of the transport medium.

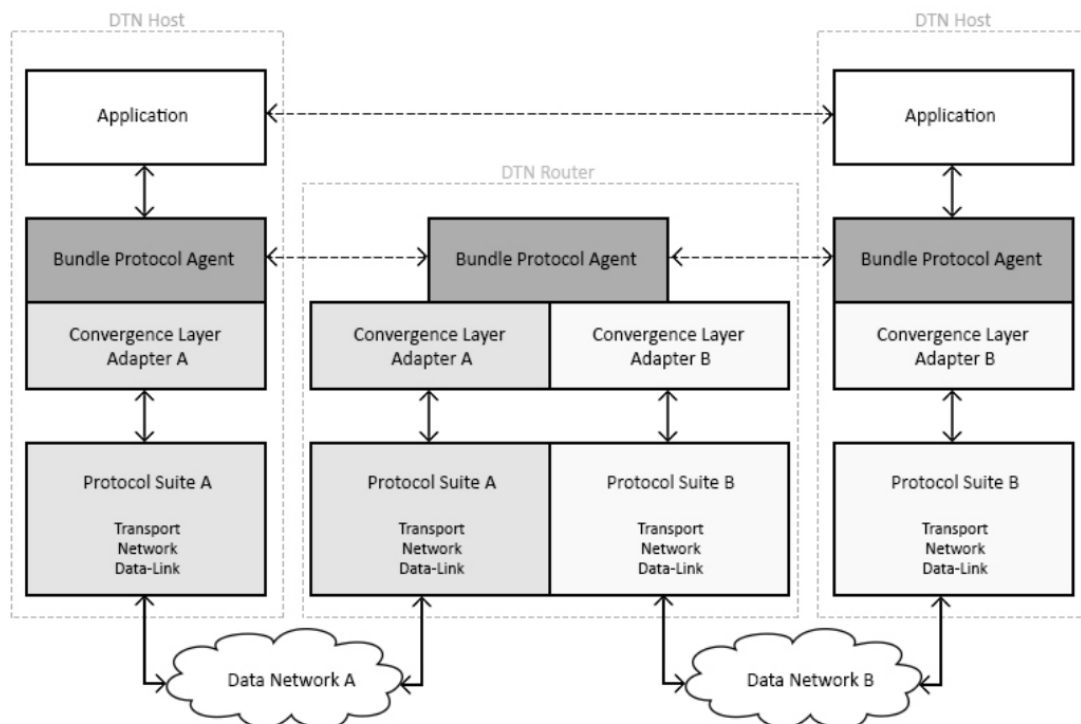


Figure 1.2 - BP and its convergence layer communicating with the lower layers, from [Pettinato_2018]

Among the most commonly used CLAs are the TCP Convergence Layer [RFC_9174] and the LTP Convergence Layer [RFC_5326]. The reader can note that the “A” of adapter was omitted in the official name of these adapters. This is the result of some inconsistencies in the first RFCs.

These are the most common, but as we can see from my work, we can stack other protocols as a Convergence Layer. As mentioned before, the Convergence Layer

represents an abstraction of the underlying protocols. These protocols can be any, ranging from standard IP protocols to CCSDS space-specific protocols.

1.4 An overview of the work done

I completed a six-month internship at the European Space Agency (ESA) in Darmstadt, Germany, where my primary responsibilities were the implementation of space communication protocols to be used in the DTN architecture, complementing the Bundle Protocol (BP) and in particular the ESA implementation that is being developed at Darmstadt (in the following, ESA-BP). Most of my work was focused on the convergence layer part of the ESA BP implementation, although I also contributed to several other system components. During the internship, I carried out four main tasks, described below.

File CLE Implementation: The first task involved the design and the implementation of the “File” Convergence Layer Element (CLE), which enables the transmission of bundles through regular files rather than Internet-based protocols. This provides an alternative method for forwarding single bundles in tests, compared to existing TCP and LTP convergence layers, useful to bypass security firewalls.

Generic Packetiser: The second task focused on developing a Generic Packetiser library capable of handling both packets and frames defined by the Space Data Link Protocols of the Consultative Committee for Space Data Systems (CCSDS) [CCSDS_PROTOCOLS].

DTN Anycast (“iac”) Protocol: The third task was the design, standardization, and implementation of an Anycast-like protocol for DTN, known as Interplanetary Anycast Communication (“iac”). The objective was to contribute to the publication of a formal Request for Comments (RFC) within the DTN Working Group and to implement an experimental version of the protocol within the ESA-BP framework.

CADU and CLTU CLEs: The fourth task comprised the implementation of two additional CLEs, CADU (Channel Access Data Unit) and CLTU (Communications Link Transmission Unit). These protocols operate beneath the CCSDS Frame layer, effectively representing the space equivalent of the data link layer in the TCP/IP model.

They provide key functionalities such as error-control coding and decoding, transfer frame delimiting and synchronization, and bit transition generation and removal.

1.4.1 Structure of the thesis

This thesis is organized as follows. Section 2 reviews the Bundle Protocol (BP) [RFC9171] and the ESA BP version 3 implementation, with particular focus on the convergence layer. Section 3 explains how to configure an ESA bundle node and client for testing purposes. Section 4 describes the File CLElement, detailing its motivation, design, implementation, and integration within ESA BP. Section 5 discusses the Generic Packetiser (GP), its role within the CCSDS protocol stack, and the improvements introduced to the GP library. Section 6 introduces the concept of DTN Anycast, presenting its purpose, formal specification, and experimental implementation within ESA BP. Section 7 provides an overview of the CADU and CLTU protocols, while Section 8 concludes the thesis by summarizing the main findings.

2 ESA BP v3

2.1 Bundle Node

A Bundle Protocol node (or BP node) is the fundamental building block of a Delay/Disruption Tolerant Network (DTN). It is responsible for creating, forwarding, receiving, processing, and storing bundles, which are the basic data units exchanged in DTN communications.

Structurally, a BP node can be viewed as a layered and modular system, composed of several functional components that interact to ensure reliable, delay-tolerant data delivery across heterogeneous networks.

It consists of three primary components:

- **Application Agent (AA):** it represents the upper layer of the node. It is responsible for processing and consuming application data that will be transmitted as bundles:
 - When sending data, the AA encapsulates application messages into service data units (SDUs) that are handed to the Bundle Protocol Agent (BPA).
 - When receiving data, the AA extracts the payload from incoming bundles and delivers it to the appropriate application service.
- **Bundle Protocol Agent (BPA):** The BPA is the core component of the node, **implementing the Bundle Protocol itself** (BPv7, as standardized in [RFC9171]), Its main responsibilities include, bundle creation and parsing (constructing primary and canonical headers, payload blocks, etc.), routing and forwarding bundles based on endpoint identifiers (EIDs), bundle storage for persistent custody and retransmission, processing of administrative records such as status reports and custody signals, etc.
- **Convergence Layer Adapters (CLAs):** The CLAs form the interface between the Bundle Protocol Agent and the lower-layer communication protocols, each

CLA provides a consistent API for the BPA to send and receive bundles, abstracting the details of the transport mechanism in use.

These are the three main components, among all the supporting subsystems for storage, routing, custody, and administration, all of them specific in [RFC9171].

2.2 ESA Implementation

ESA Bundle Protocol v3 is a new implementation of the Bundle Protocol v7 developed by ESA (European Space Agency) [ESABP_SDD]. This implementation is completely written in [JAVA], it supports both BPv7 and BPsec [RFC_9172]. It comes with native implementations of TCPCLv3, UDPCL [RFC_9174] and LTPCL.

The structure of the Bundle Node adopted by ESA follow the one described before, the main components are:

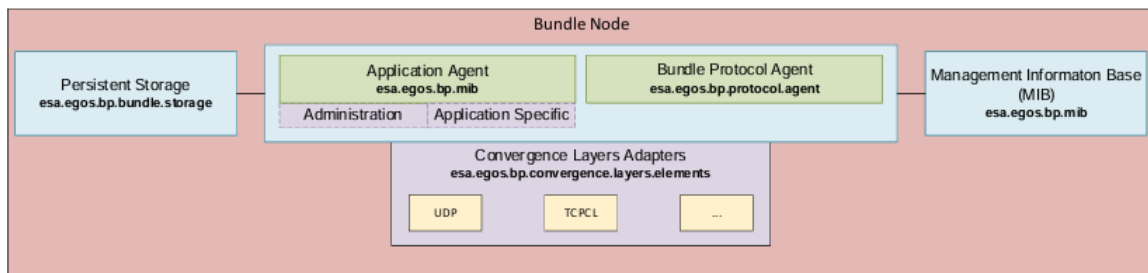


Figure 2.1 - Esa BP Structure, from [ESABP_SDD]

As we can see from Figure 2.1 the main components are:

- Application Agent
- Bundle Protocol Agent
- Convergence Layer Adapter(s)
- **Management Information Base (MIB)**: this component is used to configure all the settings in the local bundle node.
- **Persistent Storage**: used to store the bundles that are waiting to be sent.

The first three components hold the same meaning and are intended to be the implementation of the components described in the [RFC9171].

2.3 Application Agent

The Application Agent (AA) is the BN component that handles the interaction between the Client applications and the Bundle Protocol Agent. [ESABP_SDD]

The AA receives the users' requests through the Daemon and then forwards them to BPA. Moreover, it receives bundle indications and reports from BPA and forwards them to the designated resource/service.

The Daemon is a process that exposes the BP interface to outer clients. The clients can connect and register with a bundle node and subsequently use it through communication with this Daemon. The Daemon in ESA BPv3 is a Java gRPC ([GRPC]) interface, allowing any client to connect to BP using a gRPC client that implements the same procedures (by having the same proto file) to the internal Daemon.

As shown in Figure 2.1, the AA is composed of two elements: an application-specific element and an administrative element. The application-specific element constructs, requests transmission, accepts delivery, and processes application data units (ADUs). The administrative-specific element constructs request of transmissions of administrative records.

The Bundle Protocol Agent (BPA) is the core component of the Bundle Node, as it implements the functionalities of the Bundle Protocol. The key activities of the BPA are:

- Handle user requests: register, deregister and change local endpoint connection, send bundles, cancel bundle requests, and send poll bundles.
- Local Bundle delivery
- Bundle Handling: Forwarding, Reception, Compressed Custody, Deletion

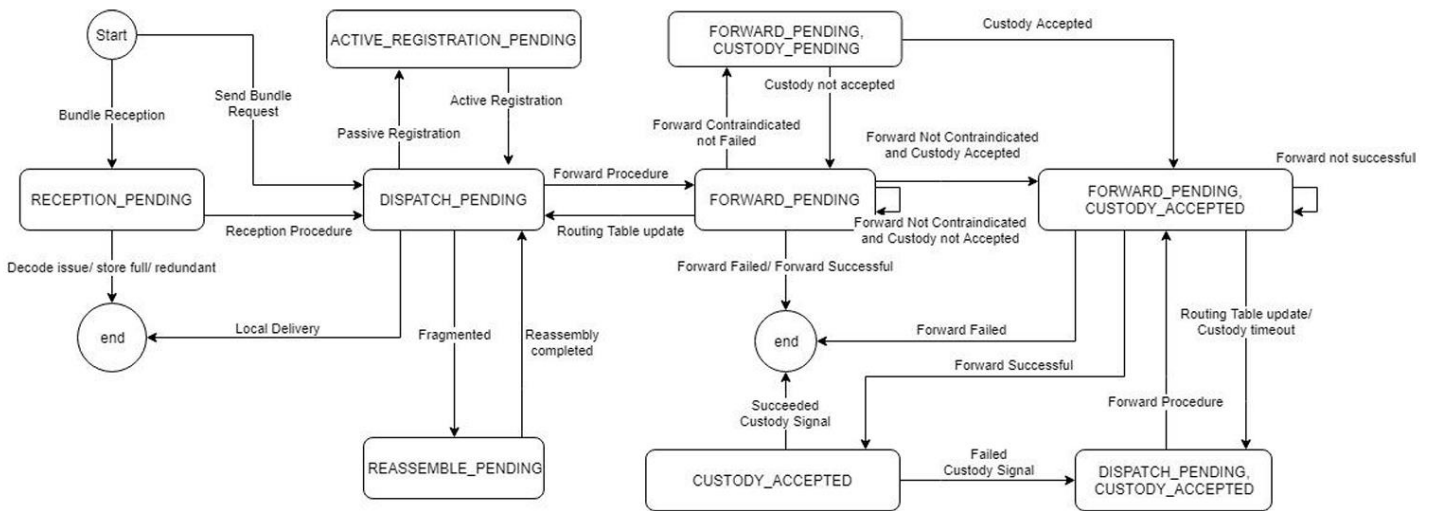


Figure 2.2 - Bundle Constraints Policies, from [ESABP_SDD]

The bundle management within the BPA is achieved by attaching one or more constraints to the bundle, which indicates the status of computation advancement, forwarding, or reception of a bundle. Then there are queues and threads associated with each constraint, each thread is responsible for doing a part of the processing of the bundle, when done, it changes the constraint of the bundle and goes on until the specific action is completed. The name and status corresponding to each constraint are reported in [RFC9171].

2.4 Convergence Layers

As the Bundle Protocol operates as an overlay network over Transport layers, in the DTN architecture the end-to-end scope of Transport is redefined, being limited to the leg between two adjacent DTN nodes (one DTN hop) [Caini_2011]., An important consequence is that it allows for different Transport protocols to be used on the different DTN hops along the path from source to destination. In this context, to allow for proper transport of Bundle Protocol data units (bundles) over the various Transport protocols, Convergence Layer Adaptors (CLAs) are required. In ESA BP the data can go through a single CLE (Convergence Layer Element) or through a stack of CLEs before it is sent as a bundle.

In the Bundle Protocol, Convergence Layer Adapters (CLAs) are designed primarily as interfaces to the lower-layer protocols. However, in the ESA BP implementation, CLAs exhibit a more complex structure, as they incorporate the entire lower protocol stack through the use of CLEs (Convergence Layer Elements), enabling it to have more control over these.

CLEs are modular units that in ESA BP represent a specific protocol in the convergence layer. This method enables ESABPv3 to define an interchangeable stack of protocols (the list of CLEs) for each CLA, so for each link to another bundle node, we can define a custom stack of communication protocols. This ensures that ESABP is highly customizable, and the addons of other protocols, defined as CLEs, are simple and modular. This is a key point, as most of my work revolved around implementing space communication protocols, and the way I have done it is through the implementation of CLEs.

For example, if we want to define in ESABP a TCP convergence layer, we should define a TCP CLA that is composed by only one TCP_CLE, while in the case of LTP, a CLA composed by one LTP_CLE and one UDP_CLE element.

As shown in Figure 2.3, CLAs will sequentially pass the bundle to each CLE in their stack, with the final CLE in the sending stack expected to be the sending CLE (e.g., UDP/TCPCL), and the first CLE in the receiving stack expected to be the receiving CLE (e.g., UDP/TCPCL). The result of the processing of the bundle after a CLE is a byte stream that is called “bundle data” in ESABP, it represents a stage of processing of the bundle. Each CLE when will receive the “bundle data” from the others, it will process it without knowing what it is exactly, as it is a semi-processed bundle.

In this thesis, we will call “bundle data block” a single bundle data chunk processed by the sending or receiving part of a CLE, meanwhile we will address multiple bundle data blocks as “bundle data”.

The CLEs at the bottom of the stack, are called “bottom stack” CLEs, this is important as usually bottom stack CLEs doesn’t only process the bundle but also effectively send

it or receive it through a real connection or whatever is defined.

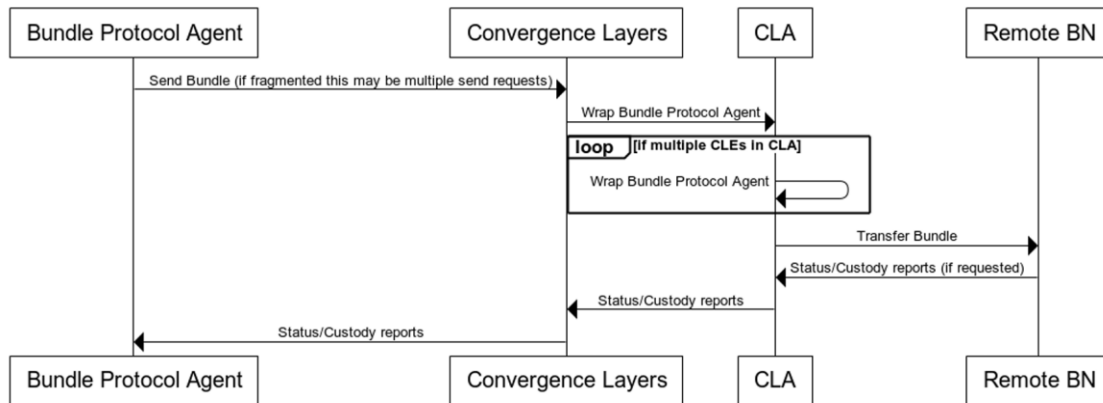


Figure 2.3 - Flow diagram of a bundle sending in ESABP, from [ESABP]

The CLEs in a stack do not communicate directly with one another; rather all interactions are carried out by the CLAdapter. This allows the CLAdapter to manage the flow and be aware of any issues. If one element becomes disconnected the CLA becomes disconnected. For example, TCP_CLE are connected when a connection has been established whereas UDP_CLE are connectionless and therefore connected when initialized. [ESABP_SDD]

3 ESA BP v3 BUILDING AND CONFIGURATION

This chapter, which references [ESA_BP_CIG], aims to describe the procedure for building the code and configuring it.

3.1 Building the ESA BP v3 project

As said in Section 2.2, the project uses Java 11 [JAVA], with Maven as a package manager [MAVEN]. Development machines at ESA are based on the Red Hat enterprise 9 distribution of Linux, and although ESA does not guarantee that the code works on other environments, systems running Ubuntu-like distributions should have no problem as well.

After downloading the project, we first move into the “src” directory at the first level of the repository by means of the following command

```
cd src
```

Then we compile the project with Maven:

```
mvn clean install -Dmaven.test.skip=true
```

The meaning of the commands above is the following:

- The **clean** command cleans the maven registry (.m2 registry) on the local machine, so that we are sure to start a fresh compilation process.
- The **install** command creates a dependency tree, based on the project configuration pom.xml on all the sub projects under the root pom.xml (the root POM) and downloads/compiles all the needed components in the registry called .m2 under the user's folder. The pom.xml is the configuration file the Maven uses to build the project.
- The *-Dmaven.test.skip* option is used to skip test during Maven compilation as it take a considerable amount of time to compile test.

After that, a “target” directory is created, containing the compiled version of the project. To install the code on other machines, we can export this directory as it is.

The “target” directory also contains the following important scripts:

- **GenCerts.sh:** this script will generate all the certificate and keys used by ESA BP both for BPsec and the gRPC secure client.
- **StartBN.sh:** this script will start a bundle node with the configuration given as input
- **StartCLIgrpc.sh:** this script will start a simple gRPC client to communicate with the local ESA BP bundle node

It also contains two important sub directories:

- **bpa:** this is the folder that contains all the information and configuration about the bundle node, bpa, convergence layer, and daemon. All details in Section **Errore. L'origine riferimento non è stata trovata..**
- **cli:** this folder contains the configurations for the gRPC client that we can use to send commands or test the local ESA BP bundle node.

After building we also need to **generate the certificates**, otherwise the bundle agent will not even start; to this end, in the terminal previously opened type:

```
./GenCerts.sh
```

We are now ready to configure the ESA BP v3 bundle node.

3.2 Node configuration

The new version of ESA-BP, still in development, introduces several changes compared to previous releases, particularly in the configuration methodology. Whereas earlier versions relied on XML-based configuration, version 3 adopts the YAML format [YAML], offering improved readability and flexibility.

To configure a basic DTN node, four primary components must be defined:

- Node configuration file (NODE.yml): specifies the fundamental parameters of the bundle node.
- Convergence layer configuration file (CL.yml): defines the interfaces and protocols used for communication.
- Daemon configuration file (DAEMON.yml): manages communication between the bundle node and external processes.
- Routing configuration file (NHT.csv): outlines the routing logic and forwarding rules for bundle delivery.

3.3 Configuration files

In ESA BP the configuration of the Bundle Node is partitioned into many files, following the code structure, instead of being centralized in a unique file, which may be practical when programming, but an obstacle to new users.

The configuration start with a directory, that is given when you compile the project with maven, that is usually called “bpa”.

From the directory “bpa” there are two main subdirectories:

- **etc** - which contains all the configurations that are “static”
- **var** - which contains all the configurations that are “dynamic”

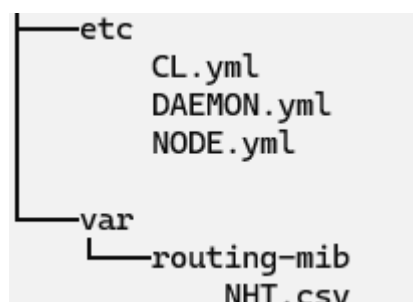
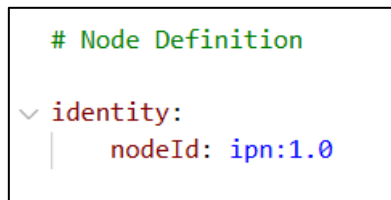


Figure 3.1 - ESA BP configuration tree

We will limit the treatment to the most important files shown in Figure 3.1, to help new users to deploy a simple ESA BP bundle node. Before configuration, however, we need to build the ESA BP project.

3.3.1 Node configuration

To configure the bundle node itself, we have a YAML file called **NODE.yml** under the directory “etc”. This file contains information about the node state, such as node identity and bundle processing information. In our case, the only part of the file that needs to be modified is “node id”.



```
# Node Definition
identity:
  nodeId: ipn:1.0
```

Figure 3.2 - NODE.yml

This value is the node's EID, and we can set it to any valid “ipn” address [RFC_9758]. ESA BP v3 can only deploy bundle nodes whose EID is “ipn”; we must also remember that the node ID must always have the service number 0, as it represents the administrative element of the node. In the same NODE.yml file, you can configure various components of the bundle node, like the bundle storage, bibe, the internal handler's timeout, blocks, etc, all the configurable parameters are written in [ESABP_CIG].

3.3.2 Convergence Layer configuration

To configure the convergence layers, we use a YAML file called CL.yml in the “etc” directory. In this file, we can put configurations for both CLAs and CLEs of the bundle node. In this section, I will explain how to set up a simple TCPCL in ESA BP v3.

As previously described in Section 2.4, each CLElement represents one or more protocols in the Convergence Layer stack; e.g., in ESA BP, a single TCPCL element corresponds to an instance of the TCPCL protocol and to the underlying TCP socket. In ESA BP, a TCPCL element can be either an INITIATOR or a RESPONDER: the Initiator starts the connection (CL client), while the Responder listens for incoming connections (CL server).

First, we configure a CLA for the TCP CL:

```

# convergence layer adapters
adapters:
  - id: ESA_OPSLAN
    endpointId: ipn:2.0
    activeOnStartup: true
    poll: 1000
    flush: 1000
    maxQueueSize: 20
    maxBundleSize: 150
    bibe: false
    retransmissionLimit: 1
    elements:
      - TCPCL_E12

```

Figure 3.3 - CL.yml – TCP CLA

Under the YAML field “adapters,” we can put a list of YAML fields that represent a single CLA, as we can see from the figure. The most important parameters here to set are:

- **id:** represents an identificatory name of the CLA inside ESA BP
- **endpointId:** represents the target EID (Endpoint ID) [RFC_9171] of the CLA, to which bundle node the CLA should communicate.
- **activeOnStartup:** tells the BPA whether the CLA should be active as soon as the bundle node is deployed; the CLA can be either ACTIVE or DEACTIVATED (as stated in Section 2.4), i.e., whether the CLA connection is on or off.
- **elements:** in this list, we include all the CLEs for the configured CLAs, effectively creating a stack of protocols within the CLAs, as mentioned in 2.4. The order of the CLEs is important: when you send the bundle, it will first go into the top CLE, then iterate down the list until the bottom one is reached.
- Other less important parameters are listed and described in [ESABP_CIG]

In the example in the Figure 3.3 we put a single CLE under the CLA called “TCPCL_E12”; the name that we can put is arbitrary, but in this example the name means TCPCL_E (E for Element) from node ipn:1.0 to ipn:2.0 (12).

Now we configure the CLE TCPCL_E12; to configure the CLEs there is a section in the same file below the adapter that is called “elements”:

```
- id: TCPCL_E12
  status: B
  implClassName: esa.egos.bp.convergence.layers.elements.CElementTcpcl
  singleton: false
  element-specific:
    role: "INITIATOR"
    ip: 127.0.0.1:4552
    port: 4552
    magic: dtn!
    version: 3
    ackSeg: true
    reFrag: true
    refCap: true
    reqLen: true
    keepAlive: 10
    maxFragSize: 60000
    maxDataLength: 120
    localReconDelay: 10
    reconDelay: 5
```

Figure 3.4 - CL.yml – TCP CLE

From Figure 3.4, the most important parameters to outline a CLE in ESA BP are:

- **status:** which are the directions that the CLE shall communicate: “S” for send only, “R” for receive only, “B” for both.
- **implClassName:** which is the class path of the Java class to be used as CLE, in this case the class dedicated to the TCP CLE is CElementTcpcl as we can notice from the Figure 3.4.
- **element-specific:** this field contains a list of parameters for the specific CLE (in this example the TCP one), these sub fields depend on the specific CLE, in case of TCPCL the most important ones are:
 - **role:** *INITIATOR* or *RESPONDER*, tells whether the CLE is a TCP client or server.

- **ip:** *<ip:port>* to which IP address and port, the client TCP should connect to.
- **port:** the port where the TCP server should listen on, if this CLE is a TCP Responder CLE the address is always localhost, so we always open the server on the local machine.
- Other parameters can be found at [RFC_9174] and more detailed in [ESABP_CIG]

After configuring the CLAs and the CLEs, the convergence layers of the bundle node is successfully configured. Remember that all of these configurations are static, meaning that at runtime we can't add/remove any CL from the running ESA BP bundle node but only activate or deactivate CLA to simulate a DTN link that is down.

3.3.3 Routing Configuration

Currently, ESA BP supports only static routing as a directly configurable option. In Delay/Disruption Tolerant Networking (DTN), routing presents a significant challenge, as discussed in [Caini_2011]. To enable more advanced network solutions, protocols such as Contact Graph Routing (CGR) and Schedule-Aware Bundle Routing (SABR) [CCSDS_SABR] are more appropriate, as they support dynamic route computation in DTN environments where connectivity changes are planned and scheduled.

Although the ESA BP library does not natively implement these protocols, it allows external applications to modify bundle routing through the dynamic routing file. However, this mechanism alone is insufficient to fully support the SABR protocol, as SABR also considers bundle-specific parameters, such as Time To Live (TTL), bundle size, and other attributes, in addition to contact plan information.

As said, routing is defined through a CSV dynamic file named **NHT.csv** (Next Hop Table) under the directory "var/routing-mib", which specifies all bundle nodes that are reachable from the local node. Each entry in this table is associated with a particular CLA, as configured in the system, thereby determining the communication path for bundle transmission.

```
# priority, destination, adapter, hopcount
# TCP route to ipn:2.0
1 ipn:2.0 ESA_OPSLAN 1
```

Figure 3.5 - NHT.csv

Figure 3.5 illustrates an example of a routing configuration entry: ***1 ipn:2.0 ESA_OPSLAN 1***.

Each line in the NHT.csv file consists of four fields, listed in order from left to right as follows:

1. **Priority** (1): Indicates the priority level for using this specific routing entry over others. For instance, if multiple routes exist to reach the same destination bundle node (same EID), the system will first attempt to use the entry with the highest priority. If that route becomes deactivated, the node will automatically fall back to the next available entry in descending priority order.
2. **Destination** (ipn:2.0): Specifies the EID (Endpoint ID) of the target bundle node to be reached. This defines the destination for the bundle transmission.
3. **Adapter** (ESA_OPSLAN): Identifies the Convergence Layer Adapter (CLA) that the local bundle node will use to forward bundles to the specified destination. The value entered here must correspond to the id field defined in the CL.yml configuration file (see Section 3.2.2).
4. **Hop Count** (1): Defines the number of intermediate bundle nodes (hops) between the local and destination nodes. In the Bundle Protocol (BP), the Hop Count Block [RFC9171] limits the maximum number of hops a bundle can traverse before being discarded.

Once these three configuration files are properly defined, communication between bundle nodes in ESA BP becomes possible. This file is located under the “var” directory, this means that this file is configurable at runtime, so we can change the routing dynamically, depending on our needs.

The final step in the setup process involves configuring the Daemon file, which completes the bundle node configuration.

3.3.4 Daemon File

As described in Section 2.3, The daemon is a component that permits outer BP clients to connect to the bundle node and send commands to it. In ESA BP v3 the configuration of the daemon is a file called **DAEMON.yml** located under the directory “etc”.

```
grpc.address: localhost
grpc.port: 5671
grpc.secure.channel: true
grpc.certificate.folder: bpa/etc/cert
grpc.server.certificate.name: server-cert.pem
grpc.server.key.name: server-key.pem
```

Figure 3.6 - DAEMON.yml

From Figure 3.6, the only important configuration is the “grpc.port”, as this is the port to which the gRPC clients will connect to command the bundle node.

3.3.5 GRPC Test Client (Optional)

As previously said, we can use a gRPC test client already built by ESA to communicate and send commands to the just deployed bundle node. To do so, there is another directory in the deployment directory “target” called “cli”, there lies all the configuration for the ESA BP CLI.

Inside the directory there is one main file that is responsible for the configuration of the cli which is the CLI.yml file:

```
grpc.address: localhost
grpc.port: 5671
grpc.certificate.folder: cli/etc/cert
grpc.client.certificate.name: client-cert.pem
grpc.client.key.name: client-key.pem
grpc.client.ca.root.name: ca-cert.pem
grpc.client.auth.token: test_read
grpc.client.secure.channel: true
grpc.client.logging.properties: cli/etc/logging/logging.properties
grpc.client.notifications.logging.path: cli/etc/logging/
client.service.number: 1
```

Figure 3.7 - CLI.yml

In this configuration file, the primary parameters to be defined are as follows:

1. **grpc.address**: Specifies the **IP address** on which the bundle node and the **gRPC server** are listening. This value is typically set to localhost; however, any valid IP address may be used, provided that a gRPC server is actively listening at the specified endpoint.
2. **grpc.port**: Defines the **port number** to which the gRPC client should connect. This value must correspond to the port configured during the bundle node deployment, as specified in the DAEMON.yml file.
3. **client.service.number**: Indicates the **service number** used when connecting to the bundle node. As discussed in Section 3.3.1, the ESA BP v3 bundle node accepts only IPN addresses [RFC9758] as valid node identifiers. Each time a new service or resource registers with the bundle node, it must provide a unique integer service number, which allows the node to distinguish between multiple remote services. This mechanism is further detailed in Section 3.5 of [RFC9758].

3.4 Start a Bundle Node and a CLI

To start the bundle node, we need to use the “StartBN.sh” script as follow:

```
./StartBN.sh <bpa configuration directory path>
```

For example, if the directory is called “bpa” and is in the same folder (as default):

```
./StartBN.sh bpa
```

If we want to start the CLI we need to use the “StartCLMgrpc.sh” instead:

```
./StartCLMgrpc.sh <cli configuration directory path>
```

For example, if the directory is called “cli” and is in the same folder (as default):

```
./StartCLMgrpc.sh cli
```

After the cli is running we can impose any type of command to it, all described in [], for example if we want to send a bundle to another node the command to use is

```
Send -d=<eid> -adu="adu_message" ....
```

Eid is the destination endpoint id, so the bundle node that we want to send the bundle to, and adu is the message that we want to send to the recipient, in string form. All the commands are listed and described deeply in [ESABP_SUM].

4 THE “FILE CLE”

If NASA and ESA need to conduct an **interoperability test**, a link between these two agencies should be established to carry out formal tests, this are classified test that are carried out seasonally between these two (and more) agencies to ensure compatibility between the different implementation of the Bundle Protocol. This, however, would require a significant effort due to security problems. A much simpler alternative is provided by the “File CLE”, which allows one agency to send a serialized bundle as a file (for example, via email) to the other.

As discussed in Section 2.4, the ESA Bundle Protocol (BP) includes several Convergence Layer Entities (CLEs), such as CLE TCP, CLE UDP, and CLE LTP, each corresponding to real Internet transport protocols. In addition to these, the **File CLE** was introduced, whose primary purpose is to enable the transmission and reception of bundles through regular files rather than via network-based protocols. My contribution to this work involved the design and implementation of the File CLE within the ESA BP version 3.

This CLE can be useful for communication and **testing purposes** (as the example just mentioned), as any operating system and machine can read standard filesystem files, which removes possible interoperability problems related to the use of different OS (e.g. endianness, etc.).

The primary purpose of the SENDER part of the CLE file is to aggregate upper-layer CLE data (e.g. bundle) into files. An important aspect is that a single file can aggregate more than one bundle, so a mechanism to track bundle data is needed to recover them correctly in the RECEIVER part.

Notice that this CLE is not just handling bundle data. However, it effectively writes on the file system, making it permanent; is designed to be a bottom stack CLE because it cannot forward bundle data to any other CLEs but will directly write them as files (the same applies to TCP or UDP, which write directly to the socket streams).

As there is no real connection between the source and the destination endpoint, the CLE will always be in a "CONNECTED" state; the only case when the CLE will not start is when it has wrong configurations, such as bad values or non-existent I/O directories.

4.1 Sender Part

The primary purpose of the SENDER part of this CLE is to **aggregate upper-layer bundle data blocks into files**. An important aspect is that a single file can aggregate more than one bundle data block, so we will need to craft a mechanism to recover them correctly in the RECEIVER part.

The bundle data block needs to be written in the file as if they were on the network because the primary purpose of this CLE is for testing and interoperability checks, so they will be written pure, as binary streams, without any processing.

4.1.1 Sender Sequence Flow

The bundle data block, already processed by the upper layers, arrives at the CLE File through the method called `doSend(BundleData bd)`. CLE now check if a file has already been opened (so we can aggregate this bundle data block with it) by checking a timeout that tells if there are open files and whether adding this bundle data block does not exceed the "fileSize" + "fileTolerance" bytes limit.

When a new file needs to be created, we check if the "outputDir" exists. If, for some unexpected reason, the "outputDir" does not exist, a reconnection sequence is launched. The algorithm will repeatedly check the existence of the output directory with an exponential delay, and after a fixed number of attempts, it will deactivate the CLE.

This is done to permit the recreation of the directory if, for some reason, it got deleted. The newly created file containing the bundles will be named following with the timestamp of their creation.

After the bundle data is successfully written to the file, it is closed if it reaches its maximum size; if not, a timeout will start to ensure that the file is closed after "fileTimeout" milliseconds.

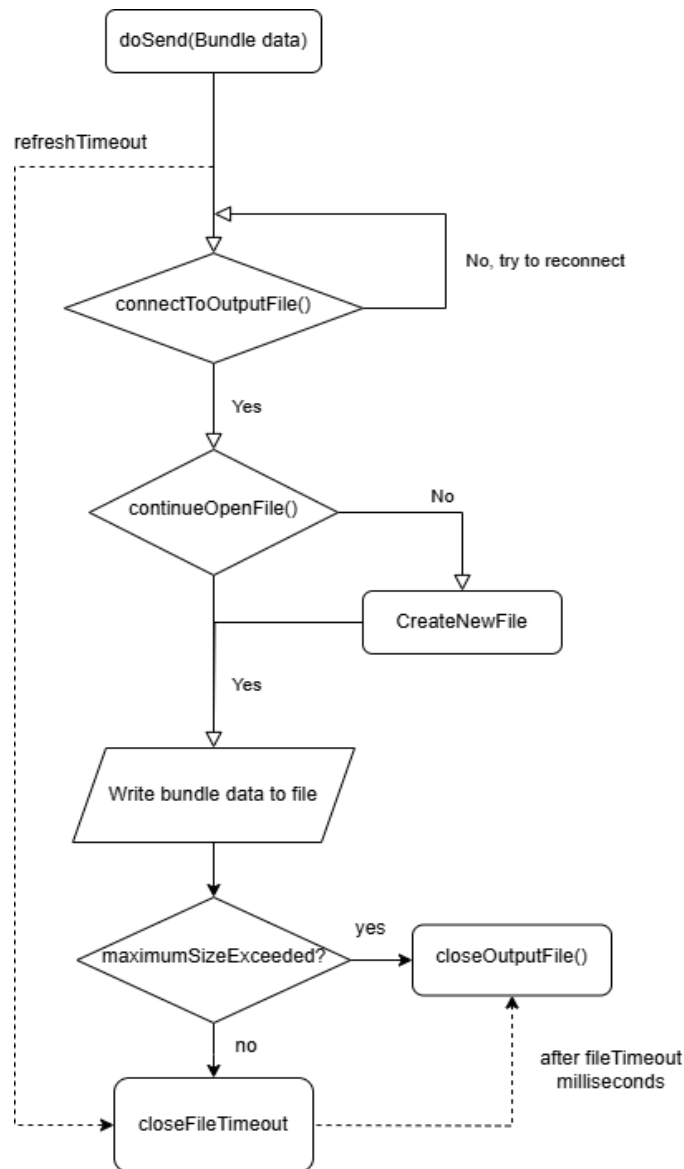


Figure 4.1 - File CLE Sender Part

4.2 Receiver Part

The receiver part is relegated to a `FileWatcher` [JAVA] thread that continues to watch the input directory for incoming files. This setup ensures it does not interfere with the sender part, allowing them to run in parallel. The `FileWatcher` monitors the “inputDir” using the `Watcher-Service` class provided by `Java.nio` package [JAVA], this ensures that the listening mechanism is passive, as the class uses file system primitives to catch file creation events.

The File CLE defines an abstract class called **AbstractStreamParser** that publishes a method called `parse(byte [] data)`. This method takes as input the stream of bytes coming from the bundle file and the identifier of the upper-layer CLEs. With this information, depending on the specific CLE, the input byte stream will be parsed according to the protocol of the upper-layer CLE.

The **StreamParser** has been modelled as an abstract class so that if, in the future, if new CLEs must be supported by the file CLE, the new parsing methods can be just added as separate classes without hindering the entire code structure (Dependency Inversion Principle). This parsing process continues until the whole file is read.

Once a file is completely processed, it is moved to the “processedDir” to ensure that it cannot be read multiple times. As with the sender part, if the “inputDir” is not found, a reconnection mechanism triggers in this part as well.

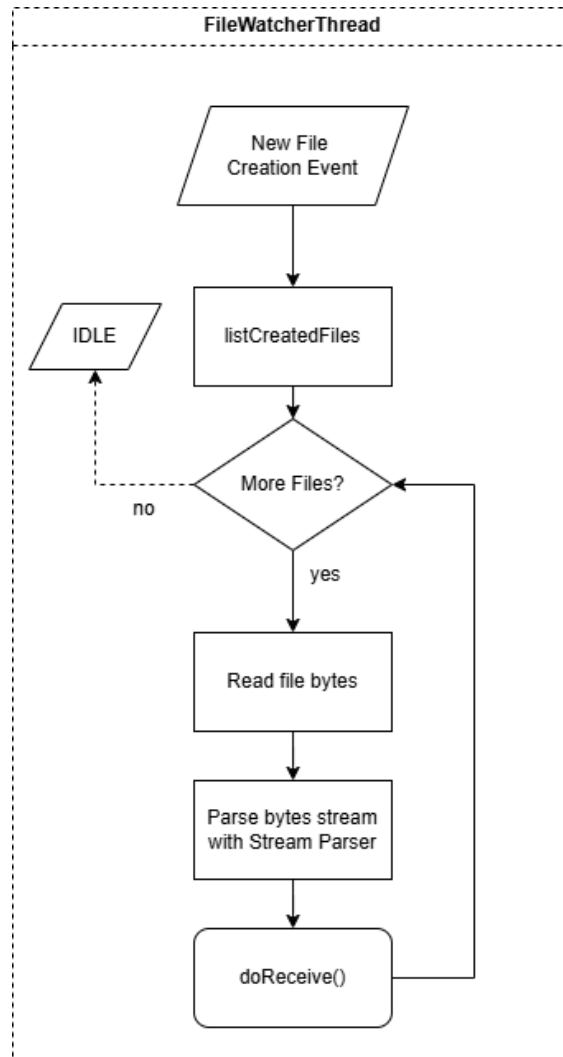


Figure 4.2 - File CLE Receiver Part

4.3 Configuration

The CLE File parameters definitions are the following:

Variable name	Description	Applicable values	Mandatory
outputDir	The output directory path to which the sender part of the CLE will create the files containing bundle data.	STRING(PATH)	Yes
inputDir	The input directory from which the receiving part of the CLE will read files	STRING(PATH)	Yes

processedDir	The directory where the receiving part of the CLE will move files once processed, defaults to inputDir/ processed/	STRING(PATH)	No
fileSize	Maximum size for a single output file; can be exceeded by a maximum of fileTolerance bytes when a new bundle of data block arrives.	LONG	Yes
fileTolerance	Maximum tolerance for incoming bundle data block that tells (with fileSize) if new data must be inserted in a new file because it is too big or can be aggregated in the current open file.	INT	Yes
fileTimeout	Milliseconds, after which the current open file, where bundles are aggregated, is closed	INT	Yes
upperCle	The id of the CLE above this, used to parse the file bundle data	STRING	Yes

Note: outputDir, inputDir, and processedDir must be all valid paths for the current file system path naming; if they are invalid, the CLE will not open, and eventually, it will disconnect.

5 CCSDS GENERIC PACKETISER

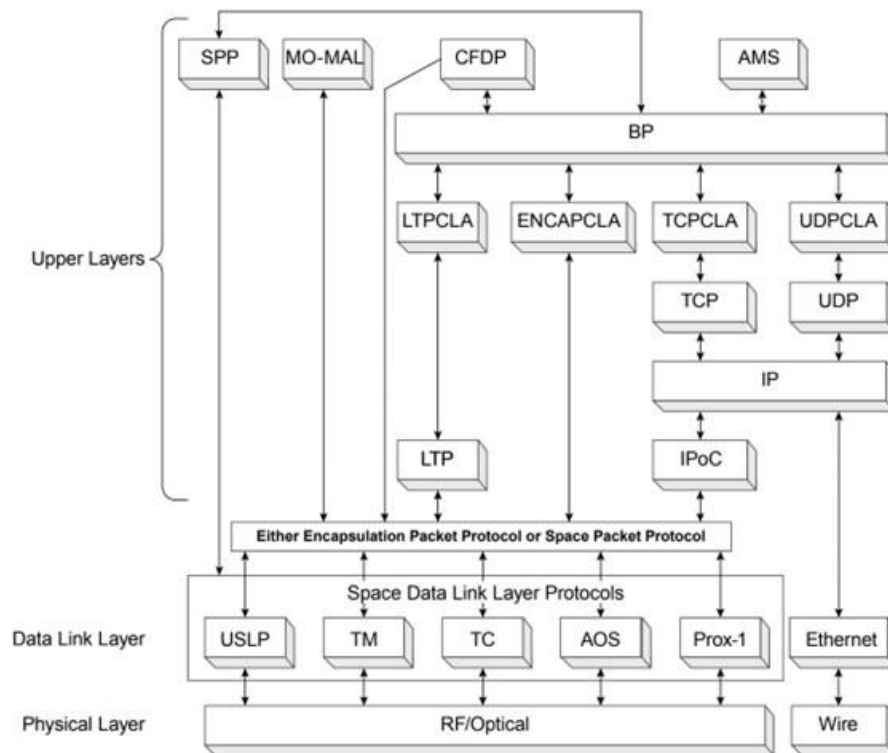


Figure 5.1 - CCSDS Space Protocols Stack, from [CCSDS_PROTOCOLS]

Figure 5.1 represents all the DTN-related protocols that the CCSDS has defined to communicate in space. The main problem in these diagram lies in the interaction between CCSDS Frames (at Space Data Link Layer) and CCSDS Packets, as these two components are **tightly coupled**, even if they should not be, so the main challenge is to design a library or a model that can handle both of them without being tied to the specific implementation.

To overcome this problem, a library called **CCSDS Generic Packetiser (GP)** was designed, whose goal is to provide a uniform and flexible method for encapsulating and transporting different data types over space communication links. GP operates at the level of the Data Link Layer, so it handles both the Space Data Link Layer Protocols (Frames) and the Packet Protocols [CCSDS_PROTOCOLS].

ESA GP (Generic Packetiser) is a software component (library) designed to handle the encapsulation and extraction of CCSDS Packets and Frames. It provides modular functionality for processing **AOS, TM, and TC** space data-link protocol, as well as **Space Packet protocols**.

At the time the task was assigned to me, there was already an implementation of the ESA Generic Packetiser, who supported the following core operations:

- **Creation of Space Packets** from user-provided data (usually bundle data).
- **Creation of CCSDS AOS/TM/TC Frames** from user data (usually bundle data).

My task was to extend the GP by adding the following capabilities to the already existing library.

- Add CCSDS **Encapsulation Packet protocol** to the ESA GP implementation.
- Add independent CCSDS Packet extraction/creation to ESA GP.
- The library **MUST** have good **performances**.
- It must be backward compatible.
- Create a **GP CLE** that uses the GP library to transform bundles into CCSDS Frames or Packet.

The main problem of previous implementation was that the Space Packet Protocol (SPP) was **tightly coupled** to frame implementation and most of the code, resulting in a lack of generalization and flexibility. The core packet interface (IPacket) was specifically bound to SPP, which **constrained** other components and the overall system architecture, making the integration of the Encapsulation Packet Protocol (EPP) practically unfeasible.

To enable this extension, a **major restructuring** of all packet-related modules was required. Also, during this process, the readability, modularity, and extensibility of the codebase were significantly improved.

Another key design challenge was to ensure that the new system would be future proof, that is, resilient to future changes and extensible enough to support any current or forthcoming CCSDS packet or frame types.

The subsequent sections present an introduction to the Space Data Link Protocol, followed by an overview of the CCSDS frame and packet structures. Thereafter, from Section 5.4 onward a detailed examination of the proposed implementation is provided, with particular emphasis on the design and functionality of the generic packetiser.

5.1 CCSDS Space Data Link Protocols

The Consultative Committee for Space Data Systems (CCSDS) defines a comprehensive framework for the formatting, transmission, and management of space mission data through the use of standardized packets and frames. At the core of this framework lie the **CCSDS Space Data Link Protocols**, which enable robust and interoperable communication between spacecraft and ground systems. These protocols establish well-defined standards for data formatting, synchronization, transmission, and verification across space communication links, thereby ensuring reliable, efficient, and internationally compatible data exchange among space missions.

Operating at the data link layer of the space communications architecture, the CCSDS family of frame protocols comprises the **Telemetry** (TM) [CCSDS_TM], **Telecommand** (TC) [CCSDS_TC], and **Advanced Orbiting Systems** (AOS) [CCSDS_AOS] protocols, each tailored to specific operational needs. The TM protocol governs the downlink of spacecraft-generated data, including critical scientific observations and operational telemetry transmitted to Earth. The TC protocol manages the uplink of commands from ground control to the spacecraft, enabling mission operations and system control. The AOS protocol supports more complex and data-intensive missions, offering capabilities for both real-time and packet-based communications while accommodating multiple virtual channels and higher data rates.

5.2 CCSDS Packets

Within this architectural framework, CCSDS packets constitute a **standardized data** structure designed to ensure consistent and interoperable communication between spacecraft and ground systems. Each packet functions as a self-contained unit of information, encapsulating telemetry, telecommand, or scientific data within a flexible and rigorously defined format.

These packets are subsequently organized and conveyed within CCSDS transfer frames, which incorporate additional mechanisms for synchronization, sequencing, and error detection, thereby enhancing the reliability and integrity of data transmission across space communication links.

Here I introduce the two main CCSDS Packet protocols used by ESA GP.

5.2.1 Space Packet Protocol

The CCSDS blue book **CCSDS 133.0-B-2** [CCSDS_SPP] specifies the structure and the usage of the **Space Packet Protocol (SPP)**.

The format of the primary header of a Space Packet:

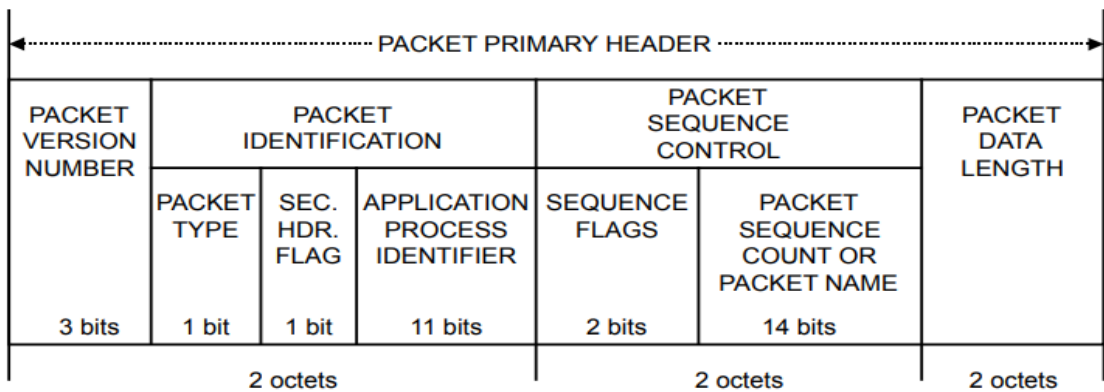


Figure 5.2 - Space Packet Primary Header, from [CCSDS_SPP]

As shown in Figure 5.2 it is composed of a Primary Header, an optional Secondary Header, and a Packet Data Field.

The **Primary Header** (always present) includes key control information:

- **Version Number** identifies the packet format version, present in every CCSDS packet.
- **Packet Identification (PID):** the main purpose of field is to identify the source or destination application process (APID) to separate different data streams.
- **Packet Sequence Control (PSC),**
- **Packet Length** field specifies the total length of the packet data field.

The **Secondary Header** is optional and highly mission-dependent; it often contains time stamps, routing tags, or contextual information required by the application.

The **Packet Data Field** follows, containing the actual user data, which could be instrument measurements, spacecraft status information, or command parameters.

5.2.2 Encapsulation Packet Protocol

The CCSDS blue book **CCSDS 133.1-B-3** [CCSDS_EPP] specifies the structure and the usage of the **Encapsulation Packet Protocol (EPP)**.

The EPP is used to transfer protocol data units recognized by CCSDS that **are not directly transferable** by the Space Data Link Protocols over an applicable ground-to-space, space-to-ground, or space-to-space communications link.

Data units that can be directly transferred by the Space Data Link Protocols have a Packet Version Number (PVN) authorized by CCSDS. The main purpose of the EPP is to provide a mechanism to transfer protocol data units **without an authorized PVN** over a space link.

It has a much simpler structure than the Space Packet:

ENCAPSULATION PACKET HEADER						
PACKET VERSION NUMBER 3 bits	ENCAPSULATION PROTOCOL ID 3 bits	LENGTH OF LENGTH 2 bits	USER DEFINED FIELD 0 or 4 bits	ENCAPSULATION PROTOCOL ID EXTENSION 0 or 4 bits	CCSDS DEFINED FIELD 0 or 2 octets	PACKET LENGTH 0 to 4 octets
'111'	'XXX'	'00'	0 bits	0 bits	0 octets	0 octets
'111'	'XXX'	'01'	0 bits	0 bits	0 octets	1 octet
'111'	'XXX'	'10'	4 bits	4 bits	0 octets	2 octets
'111'	'XXX'	'11'	4 bits	4 bits	2 octets	4 octets

Figure 5.3 - Encapsulation Packet Header, from [CCSDS_EPP]

As shown in Figure 5.3 is composed of a **Packet Header** and a **Packet Data Field**.

The Primary Header (always present) includes:

- **Version Number** identifies the packet format version, present in every CCSDS Packet.
- **Encapsulation Protocol Id (EPI)** used to identify the protocol whose data unit is encapsulated within the Encapsulation Packet.
- **Packet Length/Length of Length** field specifies the total length of the packet data field.

The header length is automatically inferred from the length of the PDU inserted in the EPP.

5.3 Configuration Overview

The current Packetiser software is called "Generic" because it is intended to be used by multiple projects with minimal adjustments, as it can handle all types of frames and packets. However, to provide the necessary flexibility and extendibility, certain specific **assumptions** and constraints about an **"abstract" Generic Frame, Packet**.

The GP is regarded as a **black box**; it receives a specific configuration/s for input data and output data. The input configurations define what is expected for the GP to receive, and the output configuration defines what is expected for the GP to return.

If the receiving data does not match the input configuration, then specific Exceptions are thrown. Therefore, the user must know exactly what is supposed to be delivered to the GP and what the output should be. Both frames and packets must be correctly configured by the user for sending, receiving, or both. The user can use a Builder class provided by the library to configure the Generic Packetiser easily.

These are all the configurations:

Configuration Frame Receiver/Sender	CCSDS Frame type	AOS	TM	TC
Frame type		Required	Required	Required
Virtual Channel (VC)		Required	Required	Required
Spacecraft ID (SCID)		Required	Required	Required
Frame Length		Required	Required	Not defined
Frame Error Control Field (FECW)		Optional	Optional	Optional
Frame Error Control Field (FECW) Usage		Optional	Optional	Optional
Frame Secondary Header		Not defined	Optional	Not defined
Extended VC Frame Count		Not defined	Optional	Not defined
Frame Header Error Control (FHEC)		Optional	Not defined	Not defined
Frame Header Error Control (FHEC) Usage		Optional	Not defined	Not defined
Operational Control Field (OCF)		Optional	Optional	Not defined
Transfer Frame Insert Zone presence (TFIZ) & length		Optional	Not defined	Not defined
VC Frame Cyclic Count		Optional	Not defined	Not defined
Segmentation		Not defined	Not defined	Optional
Idle Frame Status		Optional	Optional	Not defined

Figure 5.4 - Frame configuration parameters for SENDING or RECEIVING types. In the **red** are the optional configurations that are not yet supported by GP. In the **green** are the optional configuration that are supported by GP, from [ESA_GP]

Configuration packet Receiver/Sender	CCSDS Space Packet type	SPP	EPP
CcsdsPacketType		Required	Required
Packet type		Required	Not defined
APID		Required	Not defined
Ancillary Data Field length		Optional	Not defined
Time Code Field length		Optional	Not defined
Packet error control word (PECW)		Optional	Not defined
Packet error control word (PECW) Usage		Optional	Not defined
Required for monitoring		Optional	Not defined
Encapsulation Protocol ID (EPI)		Not defined	Required
User Data Field		Not defined	Optional
Encapsulation Protocol ID Extension Field		Not defined	Optional (Required if EPI field is set to EPI_EXT)

Figure 5.5 - Space Packet configuration parameters for SENDING or RECEIVING types, from [ESA_GP]

An important aspect of the GP is that both receiving frames and packets can have **multiple configurations**, allowing a GP to handle multiple incoming packets and frames.

The various configurations are indexed in a storage with specific IDs as indexes:

- **CCSDS Frames:** the **GCVID** (Global Virtual Channel ID) serves as the main index, identifying a single mission-defined channel ID for all frames.
- **CCSDS Packets:** a specific key is used as an index, as the CCSDS Blue Books do not define a unique key for the packets, so for Space Packets, the **APID** (Application Process ID) is used as the index, and for Encapsulation Packet, the **EPI** (Encapsulation Protocol ID) is used as the index (as show in Figure 5.6).

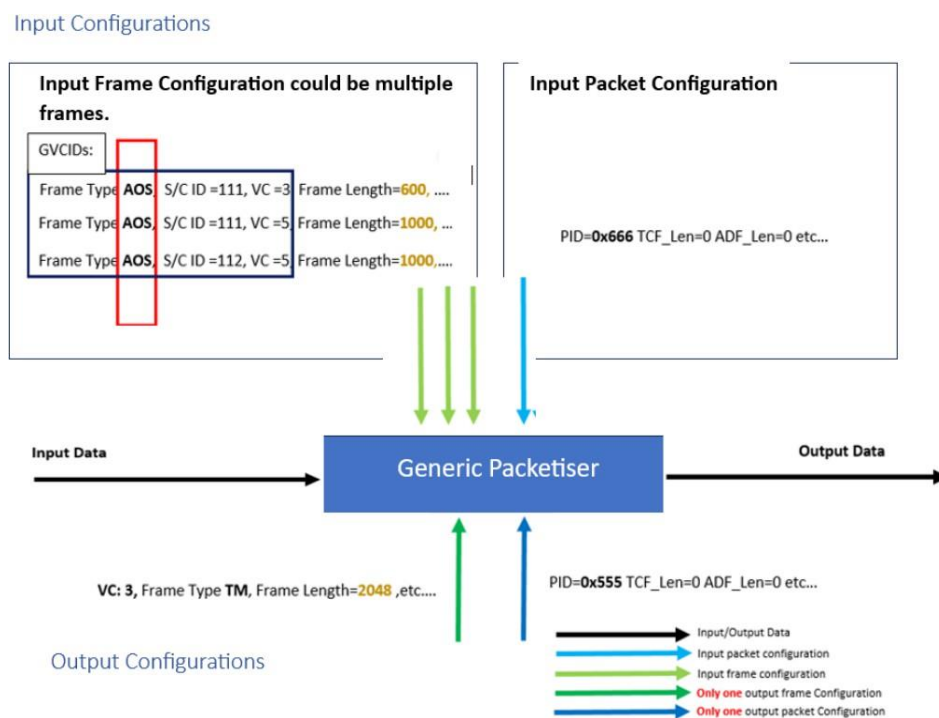


Figure 5.6 - GP Black Box Input and Output, from [ESA_GP]

5.4 Final System Design

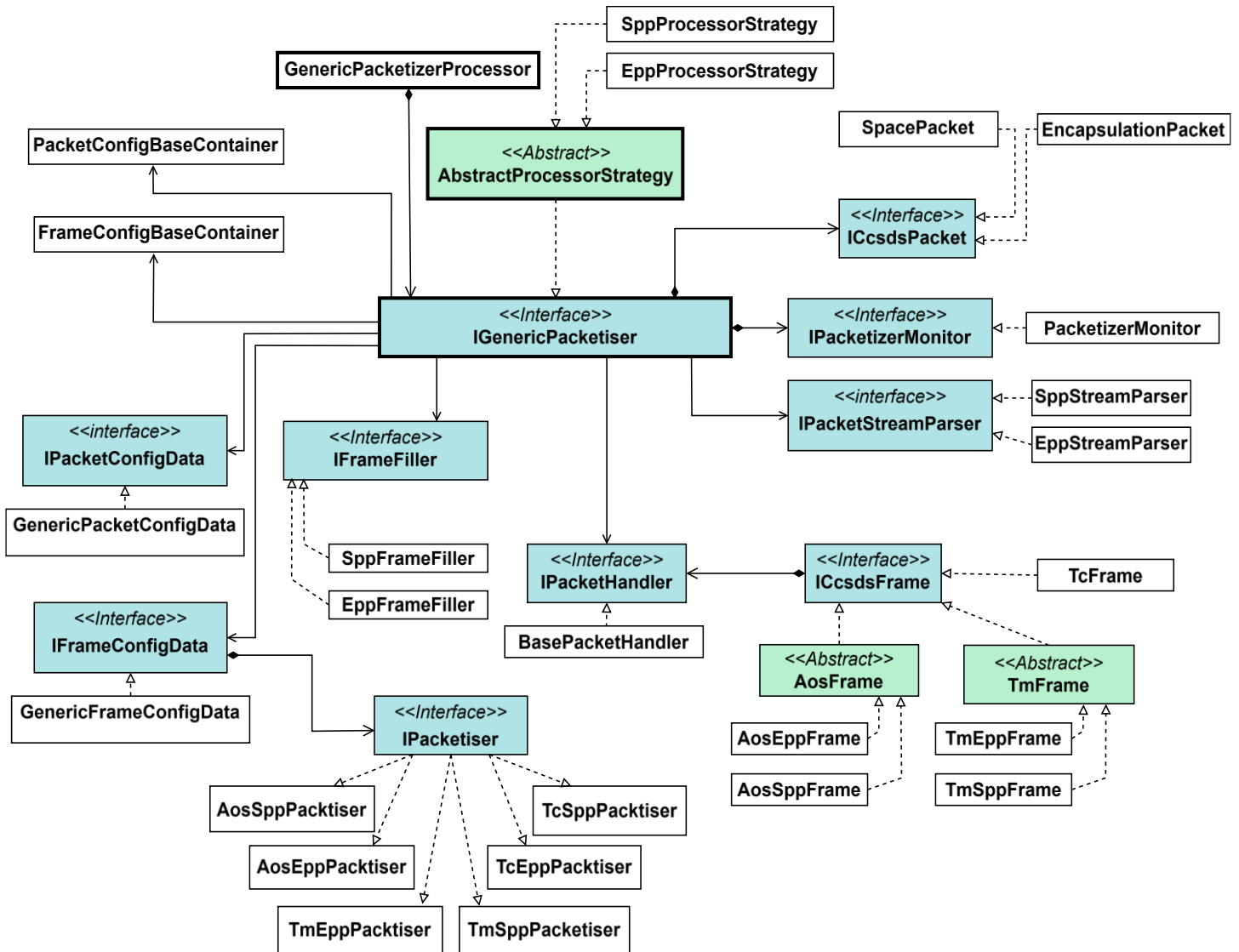


Figure 5.7 - Generic Packetiser UML, from [ESAGP]

Figure 5.7 presents the new Generic Packetiser UML after all the refactoring and the addition of EPP; the new design is cleaner and more robust than the previous one. The overall work on this library has changed its functionalities and structure so much that the ESA GP library went from version 1.2 to 2.0. So, I will now introduce the design and the structure of the new ESA GP 2.0.

As shown in Figure 5.7, The main entry point of using the GP is the interface **IGenericPacketiser**, which holds the main function for frame and packet extraction/creation, this UML depicts a simplified version of the integral system.

The following classes are important:

- **IGenericPacketiser:** as said, this interface holds the main functionality of the GP and is the entry point for the users of this library, its functionalities will be explained in Section 5.5.
- **GenericPacketiserProcessor:** main concrete class of the GP, responsible for implementing the IGenericPacketiser interface, inside handling the processors for send/receive part and for both the implementation of CCSDS Packet (exist a processor of EPP and for SPP), to do so, uses the strategy design pattern to switch implementation depending on the configuration
- **ICcsdsPacket:** contains all the most important methods for CCSDS Packets, such as the: creation and validation of a CCSDS Packet and a way to retrieve its structure. From the various CCSDS blue books about packets and frames we can outline the common behaviour of a "General CCSDS Packet", so a packet must have a PVN (Packet Version Number), being a variable-length, octet-aligned data units, a packet-length usable by the end user and every packet must implement an "idle" one (used as filling/dummy).
- **Space Packet, Encapsulation Packet:** they implement the standards defined in their respective CCSDS Blue Book [CCSDS_SPP] [CCSDS_EPP].
- **ICcsdsFrame** contains all the most important methods for CCSDS Frame in the GP such as the: creation and validation of a CCSDS Frame and insertion of a CCSDS Packet inside a Frame.
- **AosFrame, TmFrame, TcFrame:** they implement the standards defined in their respective CCSDS Blue Book, Aos and Tm frames are similar while Tc frames are different from the other two.
- **GenericFrameConfigData, GenericPacketConfigData:** these classes hold the configuration for frames and packets, also they handle the retrieval of a specific configuration from the multiple ones defined in the receiving part.

- **IPacketHandler**: class that is mainly used for handling a list of CCSDS Packets. This is used in both sending and receiving parts. To send frames, a list of valid packets must be given as input. In this case, the PacketHandler will store all the given packets in a list and slice them to spread them over multiple frames according to the specific frame type and length. When receiving frames, incomplete segments of packets can be found inside the frames. To reassemble them, the PacketHandler will buffer the received packet segments (array of bytes), and when a full packet is formed, it is returned to the user.
- **IPacketizer** and its implementations are used in the steps of extracting the Specific CCSDS Packets they implement from the incoming CCSDS frame. This requires having a IPacketizer for every type of packet and frame because different frames have different headers and methods for computing their PDUs, as well as the packets that lie inside them.
- **IFrameFiller**: this class is used to fill incomplete frame (when the length of the packet doesn't match the length of the frame), by adding "idle" CCSDS packets to the frame, depending on the configured CCSDS packet type. As said in the respective CCSDS Frames blue books [CCSDS_TM] [CCSDS_AOS]

5.5 Functionalities

5.5.1 CCSDS Packet Creation

Packet creation begins with a byte array containing user data and results in a CCSDS-compliant packet:

```
public ICsdsPacket createPacketV2(final byte[] userData) throws
    InvalidParameterException;
```

The implementation of this method is straight forward, it just create a packet with the method `initNewPacket()` of the CCSDS Packet, however an important constraint here is the size of the payload data, depends on the configured CCSDS Packet, the maximum size for Space Packets is 65.536 bytes (as defined in [CCSDS_SPP]), the maximum size for

Encapsulation Packets is 2.100.000.000 bytes (which is less than the standard [CCSDS_EPP], which is 4,294,967,287) .

5.5.2 CCSDS Frame Creation

The two key methods to enable frame construction are:

```
public int createFramesFromUserData(final byte[] userData, boolean
    completeFrameFlag, Queue<byte[]> frames) throws InvalidParameterException;

public int createFramesFromPackets(final byte[] packets, boolean
    completeFrameFlag, Queue<byte[]> frames) throws InvalidParameterException;
```

The sequence diagram of creating the CCSDS Frames is illustrated in Figure 5.8.

The red method inside the CcsdsFrame is an algorithm of inserting a Packet into a CCSDS Frame. The algorithm differs between frame types but shares many similarities with AOS and TM.

The following steps are done to create a CCSDS Frame from the Packets:

- **Data Preparation:** The GP will take every packet given as input and concatenate it together, using the previously mentioned IPacketHandler, to prepare them for spreading over multiple frames. The IPacketHandler is also used as a repository of keeping indexes and offsets of packets for the main algorithm processing.
- **Frame Packing:** CCSDS Packets are inserted into AOS/TM frames (2048 bytes) or TC frames (1024 bytes) using **insertPacket()** method from the IPacketHandler. TC frames are different from AOS/TM, mainly because they are variable-length frames, instead of the AOS/TM Frames, which are fixed-length mission-specific, so the handle of TC Frames is simpler and does not need a filling mechanism if they are not complete.

- **Frame Filling:** The **completeFrameFlag** parameter, used during frame creation, controls whether the Generic Packetiser returns the final frame in its current state. If this flag is set to:
 - **False:** if the last frame is incomplete, the frame is held and completed during the subsequent invocations of the **createFramesFromPackets** method.
 - **True:** the remaining space in the last frame is immediately filled with idle packet(s). The type of idle packet inserted depends on the configured CCSDS Packet Type, and their creation is handled by the **IFrameFiller** component, which ensures the frame is padded with appropriate CCSDS Idle Packets.

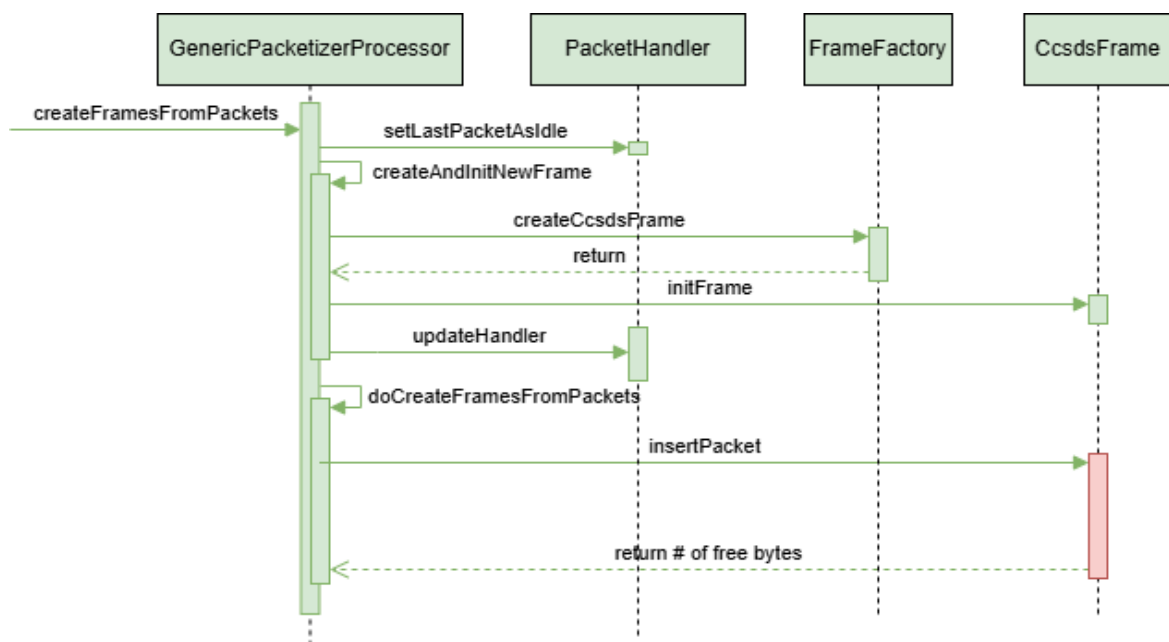


Figure 5.8 - CCSDS Frames Creation Sequence Diagram, from [ESAGP]

5.5.3 CCSDS Packet Extraction

```
public List<ICcsdsPacket> extractPacketsFromFrames(byte[] frames)
```

This method returns a list of complete packets, extracted from a concatenated byte array of CCSDS frames.

The configured receiving settings determine the type of CCSDS Packet extracted. Based on the specified CCSDS Packet Type, the Generic Packetiser dynamically selects the appropriate extraction method. Accordingly, both the ICcsdsFrame (responsible for processing incoming frames) and the IPacketHandler (which manages the packet bytes within the frame's PDU) are instantiated based on the configuration - whether Space Packet Protocol (SPP) or Encapsulation Packet Protocol (EPP). This ensures proper handling of packets, including those that span multiple frames.

The algorithm for extracting CCSDS Packets from AOS and TM frames is similar; however, it differs significantly from the approach used for TC frames. This divergence derives from structural differences between the frame types.

The sequence diagram illustrating the extraction of CCSDS Packets from CCSDS Frames is shown in Figure 5.9. As previously noted, the CcsdsFrame class may represent an AosFrame, TmFrame, or TcFrame. The red method is responsible for validating incoming CCSDS Frames. The Packetiser class serves as a general interface, substituting specific implementations such as AosPacketiser, TmPacketiser, or TcPacketiser, depending on the frame and packet type. Each frame type is processed by its corresponding packetiser, which handles the extraction of CCSDS Packets accordingly.

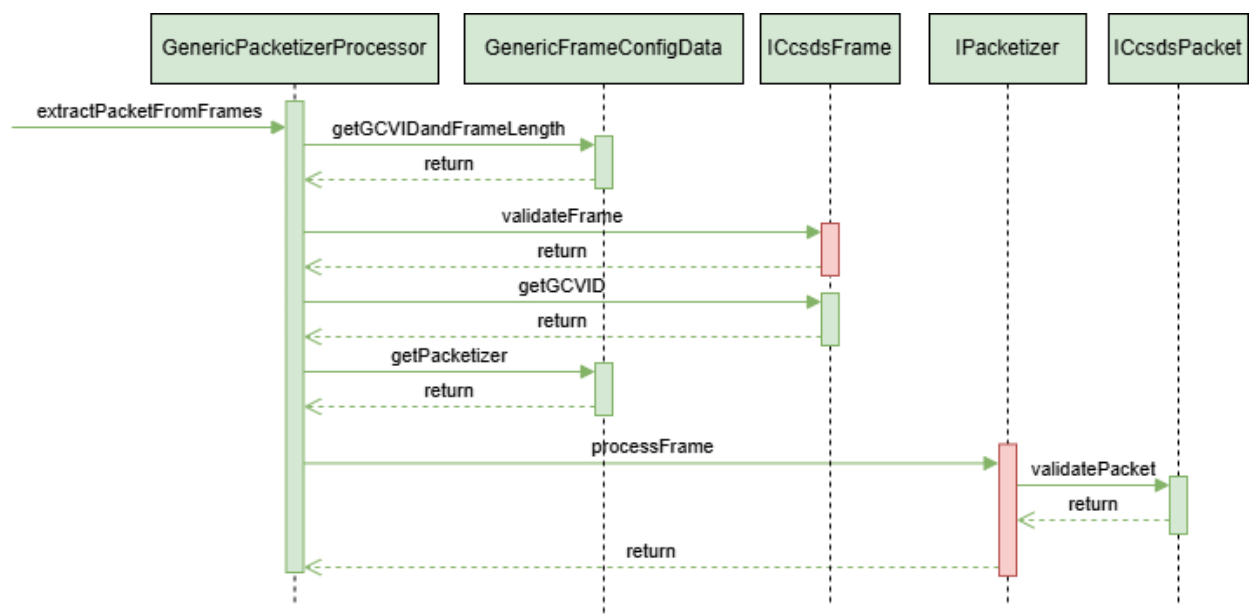


Figure 5.9 - CCSDS Packet Extraction Sequence Diagram, from [ESAGP]

The following observations apply to the algorithm of extracting the CCSDS Packets from the CCSDS Frames:

- **Frame Extraction:** The input configuration plays a crucial role, as the Generic Packetiser can process multiple frame types simultaneously. It can handle mixed frames containing different **Global Virtual Channel Identifier (GVCID)** values and extract the corresponding CCSDS Packets. During processing, the GVCID is derived from each incoming frame and matched against the predefined input configurations to ensure correct handling.
- **Packet Extraction:** The same happens for the Packet Configuration, GP can receive mixed packets with different "keys" values and return them to the end user (as shown in Figure 5.6)
 - For **Space Packets**, the "key" is the **APID** (Application Protocol ID), which specifies an ID used to identify the data contents, data source, and/or data user within a given enterprise.
 - For **Encapsulation Packets**, the "key" is the **EPI** (Encapsulation Protocol Identifier), which specifies the type of PDU inside the Encapsulation Packet.

I will not go into the details of how to extract AOS, TM and TC Frames, as it would overcomplicate the explanation.

5.6 Performances

Another important constraint given to me for this project is **performance**. The generic packetiser must generate CCSDS Frames and Packets as quickly as possible to keep up with inter-satellite and space link performances. The GP can be used in every type of space-related project. However, the ESA BP implementation aims to achieve approximately 250 Mbit/s, which I will use as the minimum threshold for performances.

The benchmarks for the send part involve creating a vast number (hundreds of thousands) of small-sized CCSDS Packets, which will be fed into the frame creation mechanism to generate even more frames.

For the receiving part, we decode all the frames that were just created and benchmark the decoding speed. The benchmarks for the send part consist of creating a vast number (hundreds of thousands) of small-sized CCSDS packets, which are then fed into the frame creation mechanism to generate even more frames. For the receiving part, we decode all the frames that were just created and benchmark the decoding speed.

The main **bottleneck** for this algorithm is the **frame creation part**, and the optional error flags that can be set for both packet and frames (FECW, PECW, ECW stands for error control word; only SPP has PECW), so the tests will be run multiple times to see how the GP behaves with more complex constraints.

Space Packet Protocol				
	Speed		N. Decoded	
Options	Send	Receiving	Packets	Frames
FECW/PECW Off	4.5 Gbit/s	5.94 Gbit/s	92.000/sec	364.000/sec
FECW On	0.9 Gbit/s	1.7 Gbit/s	27.000/sec	105.000/sec
PECW On	1.7 Gbit/s	1.7 Gbit/s	27.000/sec	105.000/sec
FECW/PECW On	0.66 Gbit/s	1 Gbit/s	16.000/sec	61.000/sec

Encapsulation Packet Protocol				
	Speed		N. Decoded	
Options	Send	Receiving	Packets	Frames
FECW Off	5.8 Gbit/s	6.3 Gbit/s	787/sec	390.000/sec
FECW On	0.9 Gbit/s	1.72 Gbit/s	215/sec	106.000/sec

As we can see from the results, the GP implementation meets the 250Mbit/s minimum threshold and even surpasses it significantly (3x-18x times). This GP implementation leverages bitwise operations and memory tricks to achieve such high speed. There are

no indications of how many packets or frames per second are sent, as it is always a fixed number (200,000 packets of 8,000 bytes, totalling 1.6 billion bytes + frame headers).

From the results, we see that Encapsulation Packet Protocol is generally faster than Space Packet Protocol. This is because the Space Packet primary header is more complex than the Encapsulation Packet, making it slower. Moreover, the EPP can contain a much larger PDU, with a limit of 2 billion bytes, compared to the 65535-byte limit of the SPP.

However, there are many fewer EPP Packets sent per second compared to SPP, because, as just said, the EPP can contain a much larger PDU. The difference is not that noticeable, as the main bottleneck is frame processing, as we are slicing these large packets into 1k-2k frames each time.

5.7 The GP CLE

I omitted the implementation of the Generic Packetiser as CLE in the ESA BP, because the implementation was trivial, I used the GP library that I worked on, linking and using it inside the ESA BP project as a standalone CLE. The configuration of the library is also straightforward, I did it by using the builder classes provided by the library, from inside the CLE, and configuring it in the ESA BP system by using the Java adapter pattern, just as any other user would do.

6 DTN ANYCAST

In the Bundle Protocol, bundle nodes are uniquely identified by an Endpoint Identifier (EID). To transmit a bundle from one node to another, the destination EID must be specified.

In DTNs the concept EID referring to endpoint containing more than one node, such as anycast or multicast EIDs, was already introduced in Section 3.4 of [RFC_4838], one of the fundamental RFCs in DTN. The rationale of unicast, anycast or multicast EIDs is defined in DTN within the RFC, although, for the time being, only unicast schemes are implemented and used within the BP.

According to the current specification of the Bundle Protocol [RFC_9171], two primary EID schemes are defined: the “**ipn**” **scheme** [RFC_9758] and the “**dtm**” **scheme** [RFC_9171].

Although these schemes differ in structures, both adhere to a **unicast routing model**, meaning they support only one-to-one communication between nodes. However, in conventional IP networks, anycast and multicast communications enable **one-to-many** or **one-to-any** transmission patterns. The introduction of DTN multi-destination schemes could prove beneficial in DTN, as explained in following sections.

I formally defined the **DTN anycast scheme** within the Bundle Protocol, with a dedicated Internet-Draft, the scheme is referred to as **Interplanetary Anycast Communication (“iac”)** and conforms to the DTN anycast EID specifications outlined in [RFC_4838]. The objective of this scheme is to integrate IP anycast functionality into the Bundle Protocol framework, enabling the delivery of a bundle to a single DTN node that is a member of a designated DTN anycast group, as illustrated in Figure 6.1.

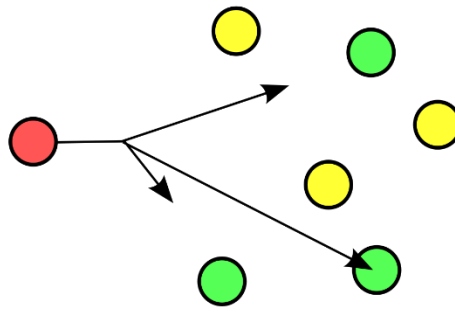


Figure 6.1 - Anycast, from [ANYCAST]

The ultimate objective of my work, after identifying and analysing the potential applications of the DTN anycast scheme, was to develop the **Internet-Draft** previously mentioned (which is available at [IETF_draft_IAC]), in view of a possible RFC, formally specifying this scheme. This draft was also discussed in public seasonal meetings of the CCSDS organization, and more recently at the half-yearly IETF (Internet Engineering Task Force) meeting.

The second part of my work involved implementing an experimental version of the “iac” scheme within **ESA BPv3**. The anycast functionality was successfully integrated into the ESA BPv3 framework and validated through a series of tests conducted in a controlled environment consisting of two or three bundle nodes, each registered to different anycast groups, together with their respective clients (see Section 3 for ESA BPv3 node configuration). The tests focused on transmitting bundles with different “iac” EIDs and verifying that their behaviour aligned with the expected DTN anycast semantics, which was confirmed to be the case.

6.1 Usages of DTN anycast

The rationale for introducing an anycast scheme is clearly stated in [RFC_4838], section 3.4. However, a few application classes, which are largely in common to terrestrial Internet counterparts, are worth citing.

A primary class of use cases involves CDN-like architectures. In traditional IP networks, anycast addressing is employed so that multiple CDN servers share a common address; client requests are automatically routed to the nearest or best server, ensuring efficiency and reduced latency.

In DTN, a similar concept applies. However, determining the “nearest” node is inherently more complex due to intermittent connectivity, variable delays, and dynamic topology changes. For this reason, the proposed Internet-Draft ([IETF_draft_IAC]) does not prescribe a specific routing policy. Instead, it allows each organization or mission to define its own criteria for selecting the optimal destination node (e.g. hop count), thereby providing flexibility and adaptability across diverse interplanetary communication scenarios. Although in latter Sections, I will describe a possible integration of DTN anycast into the modern and more precise DTN routing mechanism.

As a potential future application, **DTN Anycast** could be employed in military aviation networks, for instance, to allow a private military airplane to connect to the nearest CDN server providing a streaming or content download service. This would enable in-flight entertainment systems to operate efficiently over DTN links, dynamically selecting the optimal service node as the aircraft moves.

As previously discussed, any **CDN-like architecture** can benefit from the “iac” scheme approach. I will now list the most significant ones:

- **Key and Certificate Distribution:** In DTN environments, cryptographic keys and certificates play a crucial role in protecting bundle transmissions from unauthorized access or tampering. The **Bundle Protocol Security (BPsec)** specification [RFC 9172] defines the mechanisms used to ensure confidentiality, authenticity, and integrity of data in association with the BP. Within this context, the **Interplanetary Anycast Communication (“iac”)** scheme can enhance the efficiency and robustness of security credential dissemination, enabling bundle nodes to retrieve keys or certificates from any available and trusted distribution point within a designated group.
- **DTN BGP and DNS:** DTN currently lacks direct counterparts to the **Border Gateway Protocol (BGP)** [RFC 4271] and the **Domain Name System (DNS)** [RFC 1035], though their introduction would be highly beneficial, particularly a DTN-adapted BGP system. A BGP-like protocol in DTN could function by automatically exchanging updated **contact plans** among nearby bundle nodes.

In the Bundle Protocol, a contact plan defines predetermined time intervals (contacts) during which a node A can send data to a node B with high probability of reception; note that in space, by contrast to terrestrial networks, contacts are unidirectional (the time at which node A can send to A is different from the time at which B can send to A. In this future protocol, the “**iac**” scheme would naturally be integrated, as anycast is used also in IP-based BGP and DNS to achieve robustness and efficiency.

- **Content Delivery Networks (CDNs):** As previously mentioned, CDN-like services represent another prime application of “iac”, as this allows data requests to be routed dynamically to the nearest or most available service node within the DTN network.

Among potential applications, several are tied to specific usages of interest to one space agency. One specific use case for the “iac” scheme was in relation to the BIBE protocol [IETF_draft_BIBE]. Within ESA BP, there was a need to address “**any**” next bundle node rather than a specific one. However, this behaviour could not be achieved using the existing “ipn” or “dtn” naming schemes, as both require explicitly defined destination EIDs.

The DTN Anycast (“iac”) scheme enables precisely this functionality. It allows the definition of an “any” next hop using a special **group number**, which will be discussed in later sections. This mechanism was necessary because, in the intended BIBE workflow, a bundle would be transmitted and, at each hop, unwrapped, inspected (for integrity verification or other processing), and then re-wrapped and forwarded. The “iac” scheme thus provides the flexibility required for dynamic intermediate node selection within this process.

In the following sections, I will first briefly introduce the “ipn” scheme [RFC_9758], as the “iac” scheme was heavily inspired by the ipn scheme, and subsequently, I will do an overture of the “iac” scheme and its structure, containing a distilled version of what is written in [RFC_IAC].

6.2 Ipn Scheme Structure

The “ipn” scheme is a standardized unicast addressing format in use in BP implementations from the DTN origins. It was formally defined years later by the CCSDS and its first IETF definition is in [RFC_6260]. Then it was officially cited by BPv7 specifications [RFC_9171]. It has been recently augmented in [RFC_9758], by introducing the allocator identifier, to allow for a decentralized management of node EIDs.

This latest version follows the general structure [RFC_9758]:

ipn:[<allocator-identifier>.<node-number>.<service-number>

The fields are:

- **Allocator Identifier(optional)**: indicates the entity responsible for assigning Node Numbers to individual resource nodes, maintaining uniqueness whilst avoiding the need for a single registry for all assigned Node Numbers (it looks like what done for MAC addresses). An Allocator is any organization that wishes to assign Node Numbers for use with the 'ipn' URI scheme and has the facilities and governance to manage a public registry of assigned Node Numbers
- **Node Number**: identifies a node that hosts a resource in the context of an Allocator.
- **Service Number**: is an identifier to distinguish between DTN applications running on a given node (it is used to demultiplex bundles, the same way Internet ports do in Internet). Inside the service numbers, there are also the “well-known service numbers” (the similarity with TCP and UDP “well-know ports” is evident) which are pre-agreed default Service Number, so that unless overridden by explicit configuration, such services can be sensibly assumed to be operating on the well-known Service Number on a particular node.

6.3 The “iac” scheme

The creation of the “iac” scheme and its related Internet-draft were inspired by the “ipn” scheme [RFC_9171]. The “iac” scheme follow the general structure:

iac:<allocator-identifier>.<group-number>.<service-number>

The first part “iac” is called the **scheme name**, and the rest of the string “<allocator-identifier>.<group-number>.<service-number>” is called the scheme specific part. This is the standard format for endpoint IDs within the bundle protocol; the specifications for this rule are in section 4.2.5.1 of [RFC_9171].

In the cited section, it is also written that each scheme that can be used to form a BP endpoint ID must be added to the “Bundle Protocol URI Scheme Types” SANA registry [BP_URI_REG], which is a registry that contains a list of verified URI scheme types for the bundle protocol. This value must be put in the first item of the CBOR [RFC_8949] encoding of the EID comprising two items:

- The first item of the array SHALL be the **code number** (type) identifying the endpoint ID's URI scheme.
- The second item of the array SHALL be the applicable CBOR encoding of the **scheme-specific part** of the EID, so this depends on the specific scheme.

The code number “3” has been requested in the “iac” Internet-draft [IETF_draft_IAC] in accordance with the SANA procedure in Section 3 of [RFC_8126], as the values “1” and “2” are already taken by the “ipn” and “dtn” schemes, respectively. Along these schemes, a DTN multicast scheme is currently being formalized by ION (Interplanetary Overlay Network), yet to be standardized.

In the following section I will only explain the “iac” group numbers, as the allocator identifier and the service number are intended to have the same meaning and structure as the ones specified respectively in Section 3.2 and 3.5 of [RFC_9758], the only difference is that the allocator identifier in “iac” is mandatory, this was done to make simpler the decoding mechanism of the scheme.

6.3.1 Group Number

In Section 2.4 of [RFC_IAC] we formally define what an “iac” group number is; a **DTN anycast group** is a non-singleton Bundle Protocol (BP) endpoint identified by a URI conforming to the “iac” scheme. Basically, the group is a collection of bundle nodes that share a single anycast group. A DTN node joins an IAC group by registering with the corresponding endpoint and leaves it by unregistering.

A registered node must deliver bundles only to a single endpoint of the destination anycast group indicated as destination address in the bundle’s primary header. If the destination group differs from the node’s registration, the bundle must be forwarded to another IAC-capable node or, if unavailable, discarded.

These groups can be range from agency-defined groups for a specific service or resource, or a BP well-known group. Well-known groups are collections of DTN nodes that are publicly specified and widely adopted, so they have a predefined meaning or behaviour. To have this a SANA registry for the well-known groups has been requested called “Bundle Protocol Well-Known Group Number”.

Between these well-known group numbers, some have been already set, as they have special well-known meanings:

- The group number “0” represents the null “iac” endpoint. No nodes are associated with the null “iac” URI, and any destination identified by such an endpoint is, by definition, unreachable. The inclusion of this special null group number is intentional, as it is preferable for each URI scheme to define its own null Endpoint Identifier. In the Bundle Protocol, the EID “dtn:none” is used as a universal null EID for both the “ipn” and “dtn” schemes. This approach is undesirable and not suggested, as it could introduce inconsistencies or performance issues during EID decoding, as this particular EID is not compliant to both “ipn” and “dtn” schemes.
- The group number “1” corresponds to the “any next hop” group. When an “iac” EID is formed using group number 1, the bundle is delivered to any nearest IAC-compliant DTN node. This group effectively acts as a wildcard within the “iac”

scheme, encompassing all possible “iac” groups simultaneously. By default, every DTN node compliant with the “iac” specifications must be registered to the “any next hop” (1) group, ensuring that any bundle addressed to this anycast EID is always delivered to the node.

No other group numbers are yet defined within the *Well-Known Group Number Registry*. Consequently, any organization or individual may **request the assignment** of a **group number** within the available range for a specific purpose. For instance, a recent use case thought by ESA involved reserving a group number for BIBE nodes. In this context, any DTN node registered under the BIBE group number (e.g., group number 2) would be recognized as a BIBE-capable DTN node. Thus, when a bundle is sent with that specific IAC group number, it will be routed to the nearest BIBE-compliant node.

6.4 Routing

6.4.1 Contact Graph Routing (CGR)

As said in Section 3.3.3, the routing problem in DTN network is much more complex than in ordinary terrestrial networks. The interested reader can find general overview in [Caini_2011]. In challenged networks, as said in Section 1.1, the network is composed of channels that are often intermittent, with a fundamental difference between terrestrial and space challenged networks.

The main difficulty arises from the fact that, unlike in Internet, where routes can be computed using up-to-date and we have a nearly complete knowledge of the network topology provided by IP routing protocols, DTN routing cannot rely on such immediacy. Due to the inherent characteristics of DTNs, information about topology changes cannot be assumed to propagate quickly or reliably enough to be consistently available for route computation.

By the very nature of DTNs, it cannot be assumed that information about topology changes is propagated to the relevant network entities quickly enough to be reliably used for route computation. [Burleigh_2016]

In DTN networks we can rely on CGR (Contact Graph Routing) with its latest standardized version called SABR (Schedule-aware Bundle Routing) by CCSDS [CCSDS_SABR] to route bundles in a deterministic scheduled connectivity environment. SABR is designed for use in networks in which changes in connectivity are planned and scheduled, rather than predicted, discovered, or contemporaneously asserted.

In DTN networks in which changes in connectivity are scheduled, a global 'contact plan' of all such events may be distributed in advance to all DTN nodes, enabling each node to have a theoretically accurate understanding of connectivity in the network at any specified moment.

A DTN network contact constitutes a unidirectional transmission from some identified node to some other identified node, characterized by transmission time, reception time, and volume, the maximum amount of data that can be transferred during the contact, given by the difference between contact start time and contact stop time, multiplied by the transmission data rate. [Burleigh_2016]

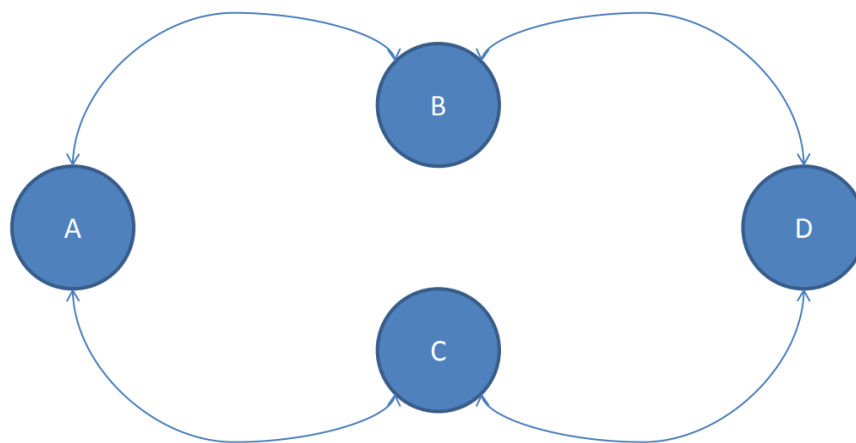


Figure 6.2 - Network Topology Example, from [Burleigh_2016]

Sender	Recvr	From	Until	Range (light seconds)
A	B	1000	1100	1
A	C	1100	1200	30
B	D	1400	1500	120
C	D	1500	1600	90

Figure 3-3: Contact Plan Example: Range Intervals

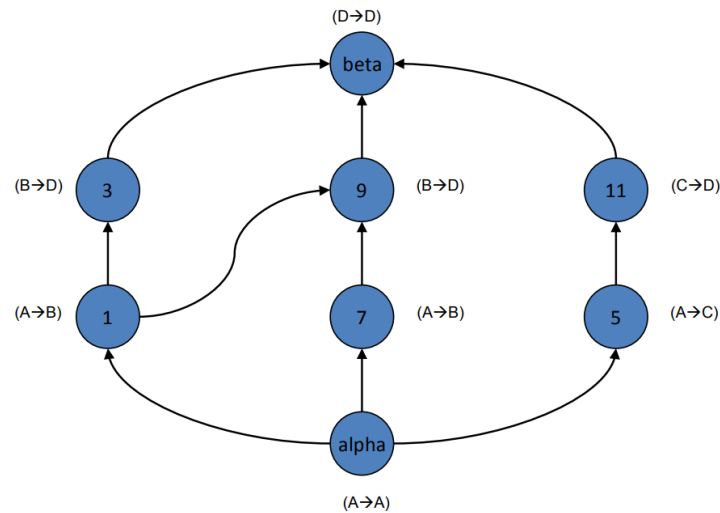


Figure 6.3 - Contact Graph, from [CCSDS_SABR]

Internet routing can be viewed as analogous to planning a road trip: links represent highway segments forming the **arcs** of a graph, while hosts and routers correspond to towns and highway interchanges, serving as the **vertices**. Each arc is associated with a traversal cost, and the objective is to determine the route with the **lowest total cost** between two vertices in the graph.

Contact graph routing is instead like booking airline flights for a business trip. A single airline flight constitutes transit from some identified airport to some other identified airport, characterized by departure time, arrival time, and the number of passengers the aircraft can carry.

As we can see from Figure 6.3, CGR uses a structure called “contact graph” to make routing decisions, because as just said, the **graph nodes correspond to contacts between nodes**, not to nodes.

This is crucial, as CGR objective is to select, for each bundle, a **sequence of contacts** (not nodes) that results in the earliest final arrival time, regardless of which nodes are on the route.

It is also important to point out that, in the graph, these contacts are communication between two BPAs of two bundle nodes that are identified by '**unicast**' EID. In Figure 6.3, the contact between node A and node B could correspond to a communication between a DTN node with EID such as '*ipn:2.1*' to '*dtn://server/a*' (arbitrary chosen), multi-destination protocol such as anycast/multicast are incompatible with the current specification of CGR/SABR, as stated in Section 2.2 of [CCSDS_SABR] .

The terminal vertexes are auto-contacts for both source and destination node, done to account propagation delays in both bundle sending and delivery. As previously discussed, the objective of CGR/SABR is to send a bundle from 'alpha' to 'beta' (from Figure 6.3) following the fastest path (series of contacts). To achieve this, shortest-path algorithms are applied to the contact graph to determine the most efficient route for bundle delivery; in CGR, Dijkstra's algorithm is used, while in SABR, Yen's algorithm is used [YEN].

Dijkstra's algorithm has proven to be correct, as it always identifies the optimal shortest path between nodes [DIJKSTRA]. At each iteration, the algorithm selects a graph node V that has not yet been finalized and possesses the smallest tentative distance from the set of finalized graph nodes. The node V is then marked as finalized, and the algorithm updates the distances to each of its unvisited neighbouring nodes if a shorter path is found through V .

This process ensures that, at the moment a graph node V is finalized, the algorithm has determined the minimum possible distance to that node, this applies for all graph nodes not only the destination ones. This property is particularly relevant for the subsequent discussion on the adaptation of CGR/SABR algorithm for DTN anycast routing.

SABR let the implementer free of choosing the algorithm to compute the shortest path but cites in a note the possible use of Yen's, the algorithm that is operationally used in

ION. Yen's algorithm, computes the K loop-less shortest paths instead of one as Dijkstra, however, Yen's algorithm internally relies on Dijkstra's algorithm to determine each shortest path, thereby preserving the same correctness guarantees.

6.4.2 Proposed Possible Solution

The goal is to integrate DTN anycast into the current implementation of CGR. The first component to adapt is the contact graph itself, because as said in previous sections, the graph only contains contacts between nodes defined by a unicast EID. The add-ons is pretty much trivial, as we just need to introduce a new variable to each node that contains its **list of registered group numbers**, therefore adding the knowledge of the "iac" anycast groups number into the contact graph.

While traditional CGR routing selects a single unicast destination as the terminal vertex in the contact graph, DTN anycast introduces multiple possible destination endpoints. Therefore, for each possible anycast destination, an auto-loop contact, must be added to the contact graph, this ensures that the propagation delay of the delivery of a bundle is considered at each destination node.

The second component requiring adaptation is the route computation algorithm itself. Specifically, the Dijkstra or Yen algorithm must be slightly modified. In the original CGR Dijkstra algorithm, the process terminates once the destination node is marked as *finalized*. My intuition was to apply Dijkstra's algorithm to the anycast-compliant contact graph and observe which of the feasible anycast destination node is reached first.

As discussed in the previous section, Dijkstra's algorithm always explores the closest graph nodes to those already finalized, and each time a new graph node is finalized, the shortest path to that new node is determined by the currently selected arcs to reach it. Based on this principle, the desired behaviour emerges when applying the modified version of Dijkstra's algorithm.

If one of the potential anycast destination bundle nodes is **finalized** by the Dijkstra algorithm during expansion, it means that this destination node was examined before any other destination node. Dijkstra finalizes graph nodes in order of increasing

distance, therefore, according to its **correctness property** [DIJKSTRA], the newly finalized destination node is indeed the closest to the source. By finalizing this graph node, Dijkstra has already determined the shortest sequence of contacts leading to it, allowing the algorithm to pre-emptively terminate, without further exploration, saving up computational cost.

6.4.3 Examples of DTN Anycast Contact Graph Routing

In this subsection, we present two examples. These examples are based on the Dijkstra, however, the examples remain fully applicable to the Yen's algorithm, since it internally relies on Dijkstra's algorithm for the first and the secondary shortest paths computation.

6.4.3.1 Example 1

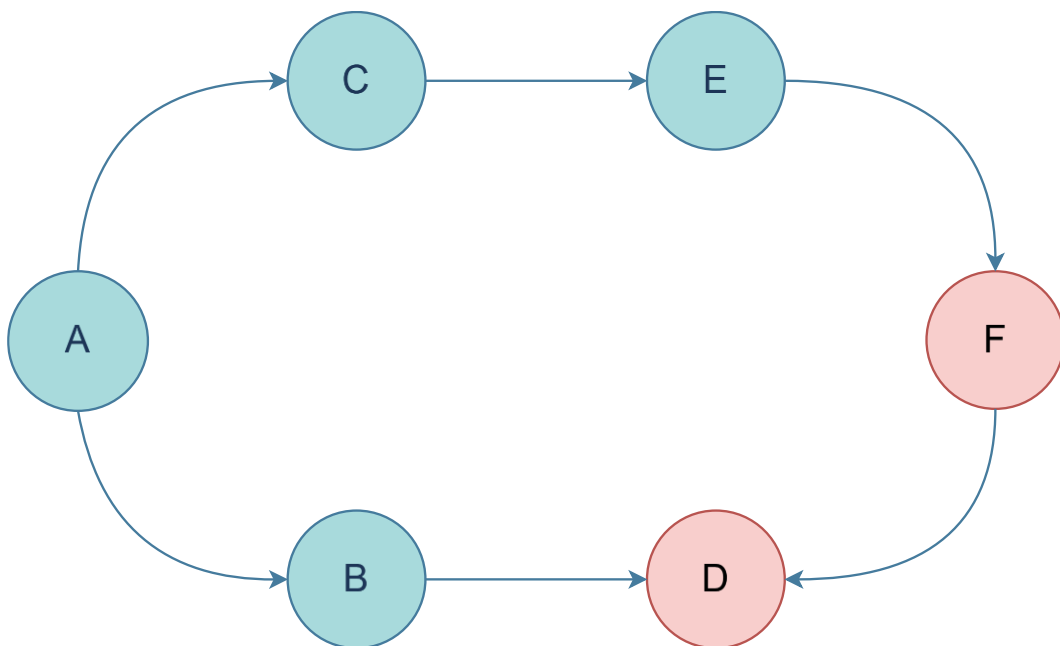


Figure 6.4 - Example 1 - Network Topology

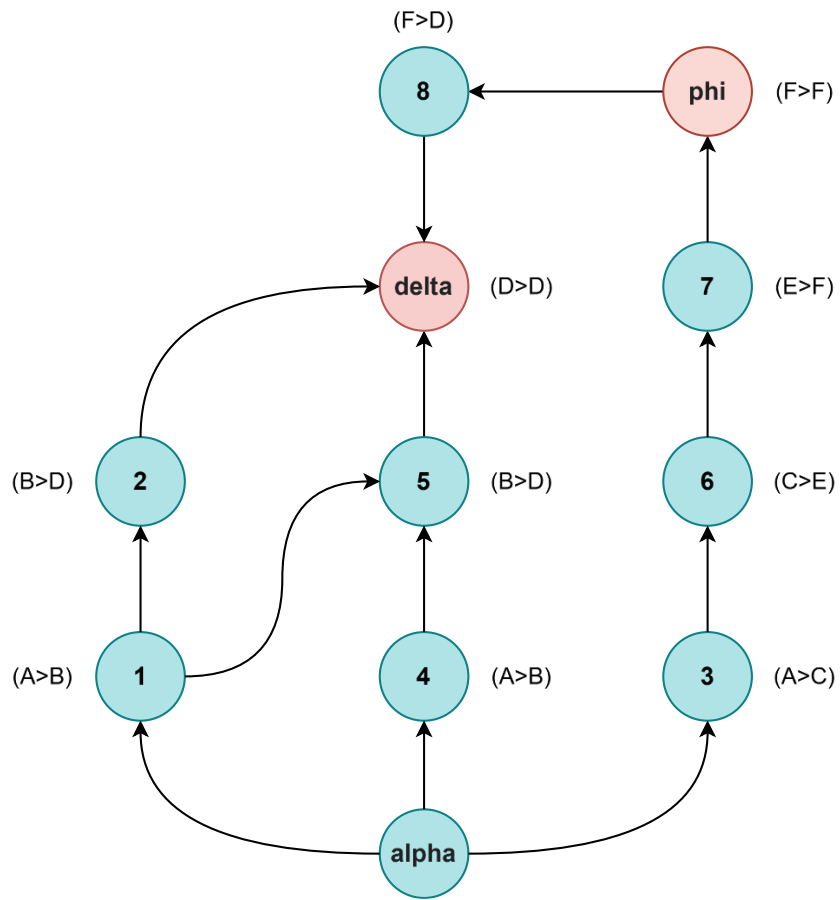


Figure 6.5 - Example 1 - Contact Graph

Contact	Sender	Receiver	From	Until	Rate (Kbps)	Propagation Delay (ms)
1	A	B	1000	1100	1000	0
2	B	D	1200	1300	1000	0
3	A	C	1100	1200	1000	0
4	A	B	1300	1400	1000	0
5	B	D	1400	1500	1000	0
6	C	E	1200	1300	1000	0
7	E	F	1500	1600	1000	0
8	F	D	1600	1700	1000	0

Table 6.1 - Example 1 - contact plan for contact graph in Figure 6.5

Node	Endpoint ID	Anycast Groups
A	lpn:1.0	[]
B	lpn:2.0	[]
C	lpn:3.0	[3]
D	lpn:4.0	[2,3]
E	lpn:5.0	[3]
F	lpn:6.0	[2]

Table 6.2 - Example 1 - Nodes details

We take as an example the contact graph shown in Figure 6.5, which illustrates a modified contact graph containing DTN anycast compliant bundle nodes, as detailed in Table 6.1 and Table 6.2.

The objective is to transmit a **DTN anycast bundle** with the “iac” EID “**iac:0.2.1**”. This implies that any DTN node registered to group number 2 is a valid recipient. As indicated in Table 6.2, **nodes D** and **F** meet this condition, as both are registered to the DTN anycast group 2. This means that in the final contact graph we will have two auto-loop contacts, one for each destination, the contacts “delta” and “phi” respectively.

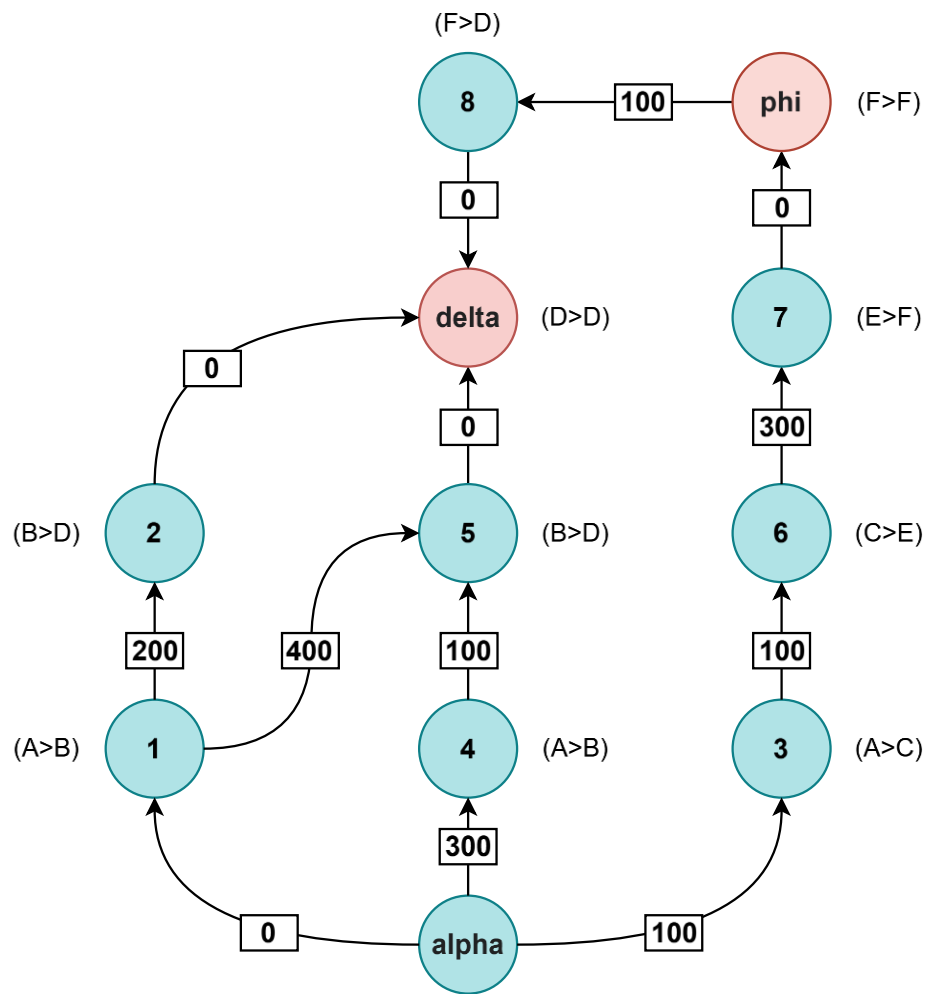


Figure 6.6 - Example 1 - Contact Graph with arc costs (retention time + propagation delay)

Dijkstra algorithm requires distances on the arcs to compute the shortest path, each arc cost in the contact graph is a sum of: the **retention time** of the bundle on a specific node until the next contact plan is ready, the **propagation delay** for each contact that delays the bundle forwarding and the time taken to transmit the bundle given its size and the transmission rate of the link (not taken into consideration into this example).

Figure 6.6, shows the example contact graph with the arc costs of the bundle specified on each arc. In this example we will assume that:

- The size of the bundle is negligible, so the transmission speed is negligible, only the propagation delays and retention times are considered.
- There is no traffic on the network.

- The starting time of the simulation is at time “1000”.

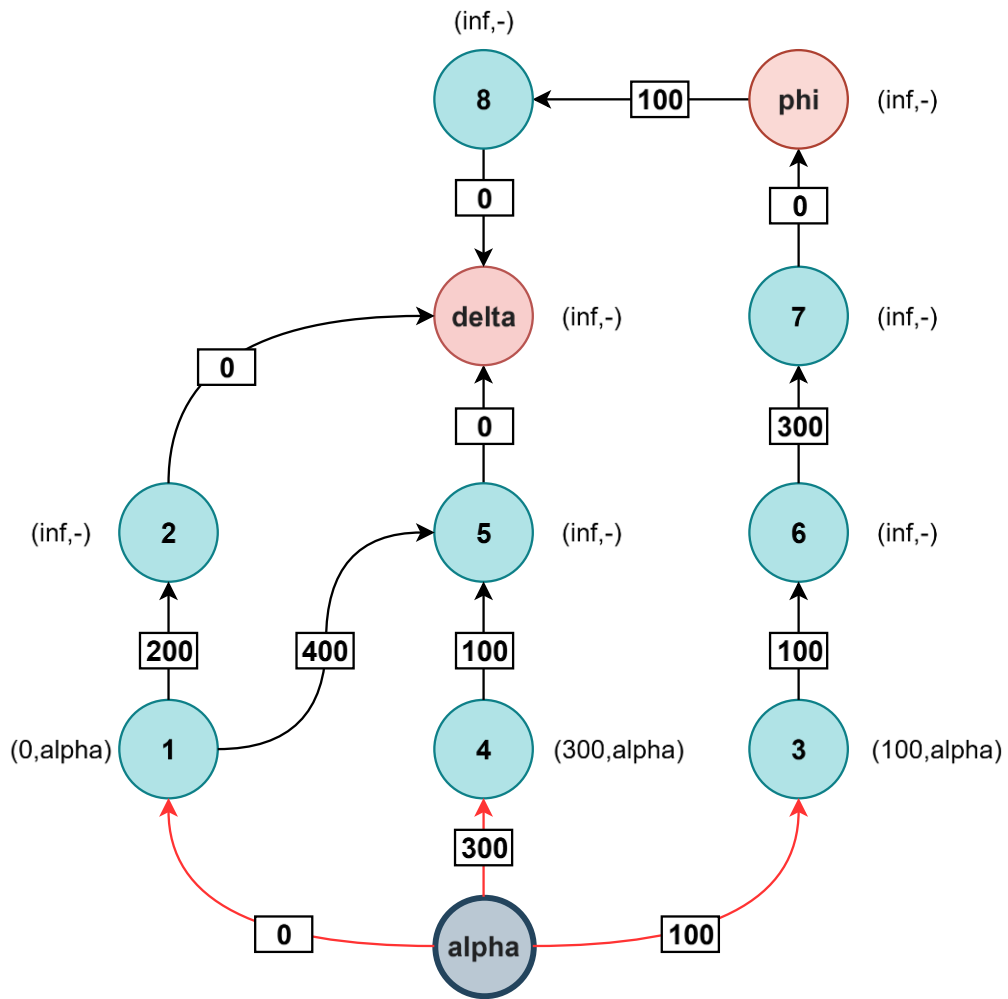


Figure 6.7 - Example 1 - Dijkstra execution step 1

Figure 6.7 shows the first step of the execution of the Dijkstra algorithm on the example contact graph shown in Figure 6.6. In the figure we start from the source node ‘alpha’, Dijkstra at each iteration must choose the closest node from the ‘finalized’ ones. In the figure, the finalized nodes are outlined and coloured grey, the possible arc choices at each iteration are coloured red.

Beside each node there’s the standard Dijkstra notation (L, P) , where L is the “length” i.e. the total shortest distance from the source node and P is the previous graph node in the shortest path.

In this first iteration, as we can see from figure Dijkstra has 3 possibilities: expand node 1 with cost 0, expand node 4 with cost 400, and expand node 3 with cost 100. Dijkstra expand the node 1 first as it is the lowest cost one.

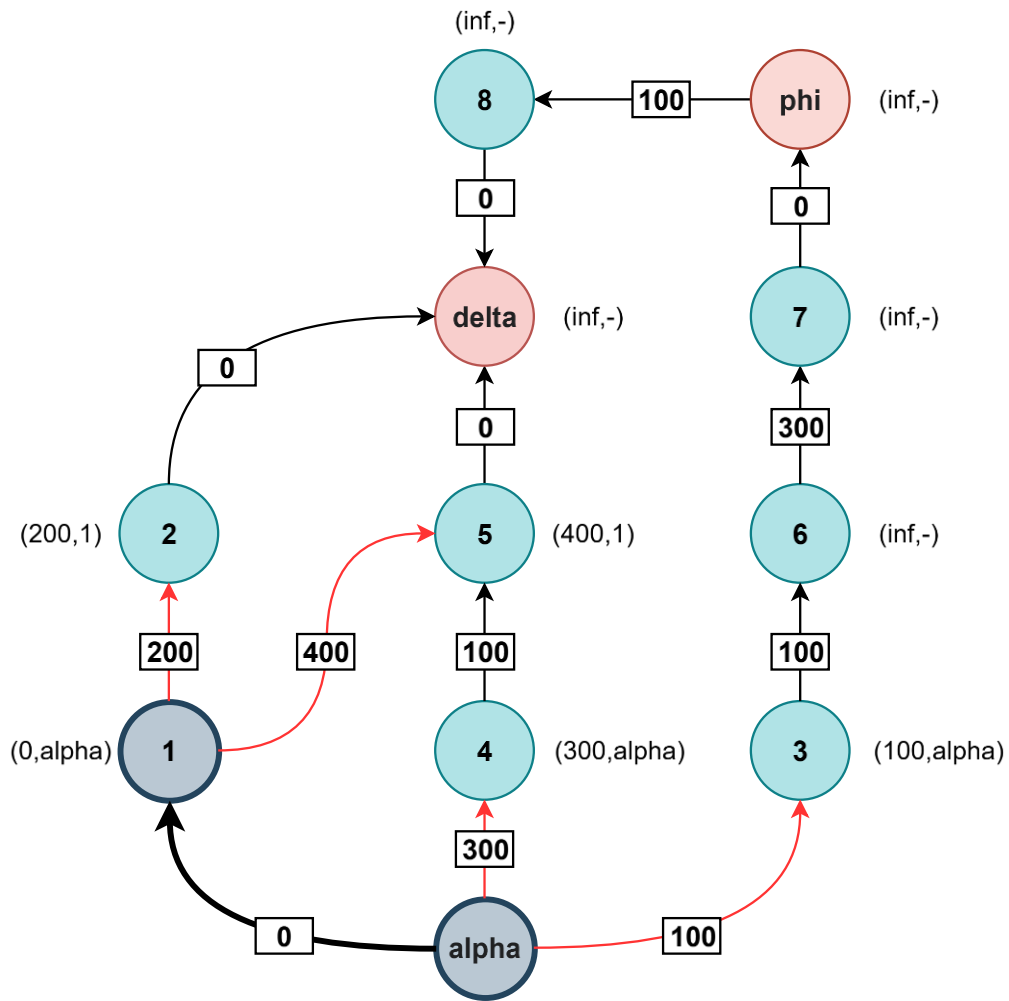


Figure 6.8 - Example 1 - Dijkstra execution step 2

Figure 6.8 shows the step 2 of the execution of Dijkstra, previous step finalized graph node (contact) 1, so now we compute all the possible new expansion cost from the outgoing arcs of node 1, and we end up with a total of 4 possible expansion paths at this point:

- con 2 - distance 200 (200+0).
- node 5 - distance 400 (400+0).

- node 4 - distance 300.
- node 3 - distance 100.

So, Dijkstra will expand the graph node 3 at this iteration.

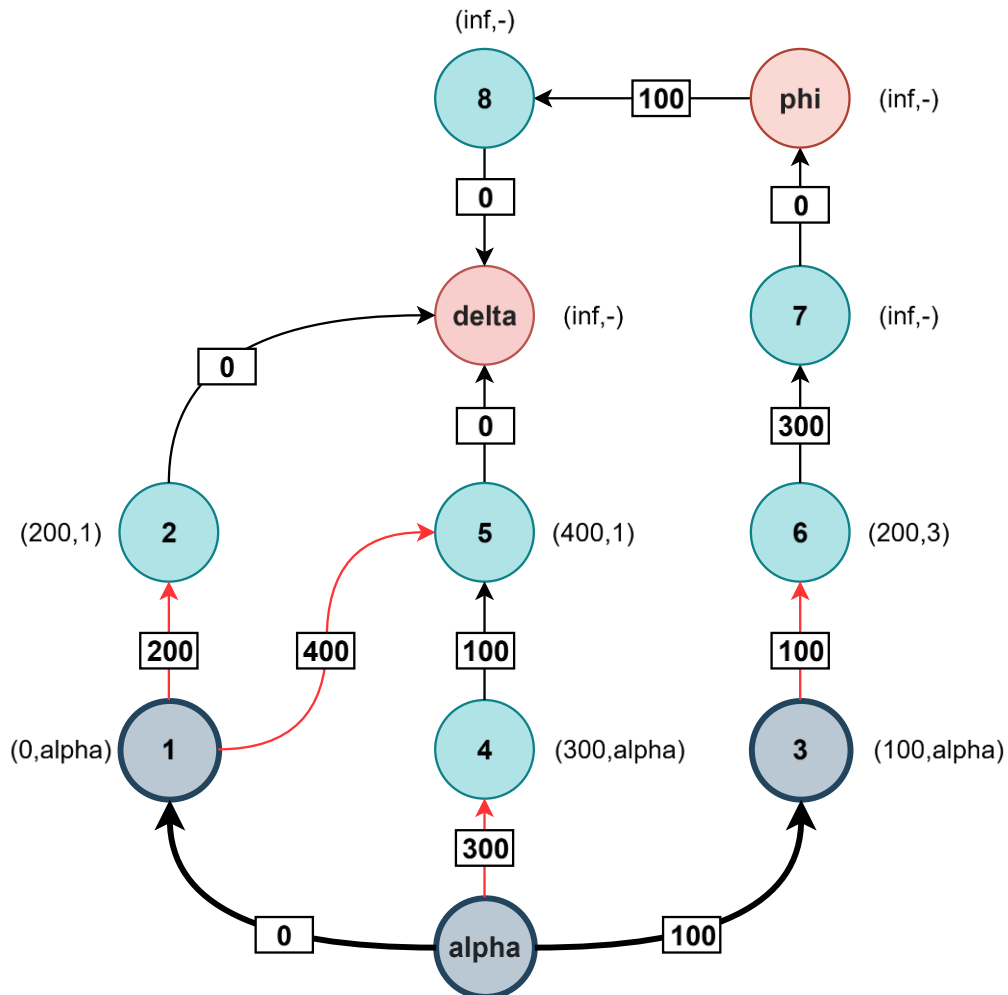


Figure 6.9 - Example 1 - Dijkstra execution step 3

Figure 6.9 shows the step 3 of the execution of Dijkstra, the graph node 3 has been expanded at the previous iteration. After computing all the possible new path costs, Dijkstra encounter a tie at this step, between the graph node 6 and node 2, as they both have cost 200.

It is important to point out that in tie cases, whatever path we choose, the path found by Dijkstra will still be the optimal one, holding true by its correctness property, the choice is purely based on our personal needs.

In tie cases the sub-comparison methods are used, in CGR the sub-comparison methods were: hop counts, but both contacts are distant two hops from the source one, propagation delay, but both contacts have a propagation delay of 0, and ultimately the contact number, therefore the contact 2 is selected.

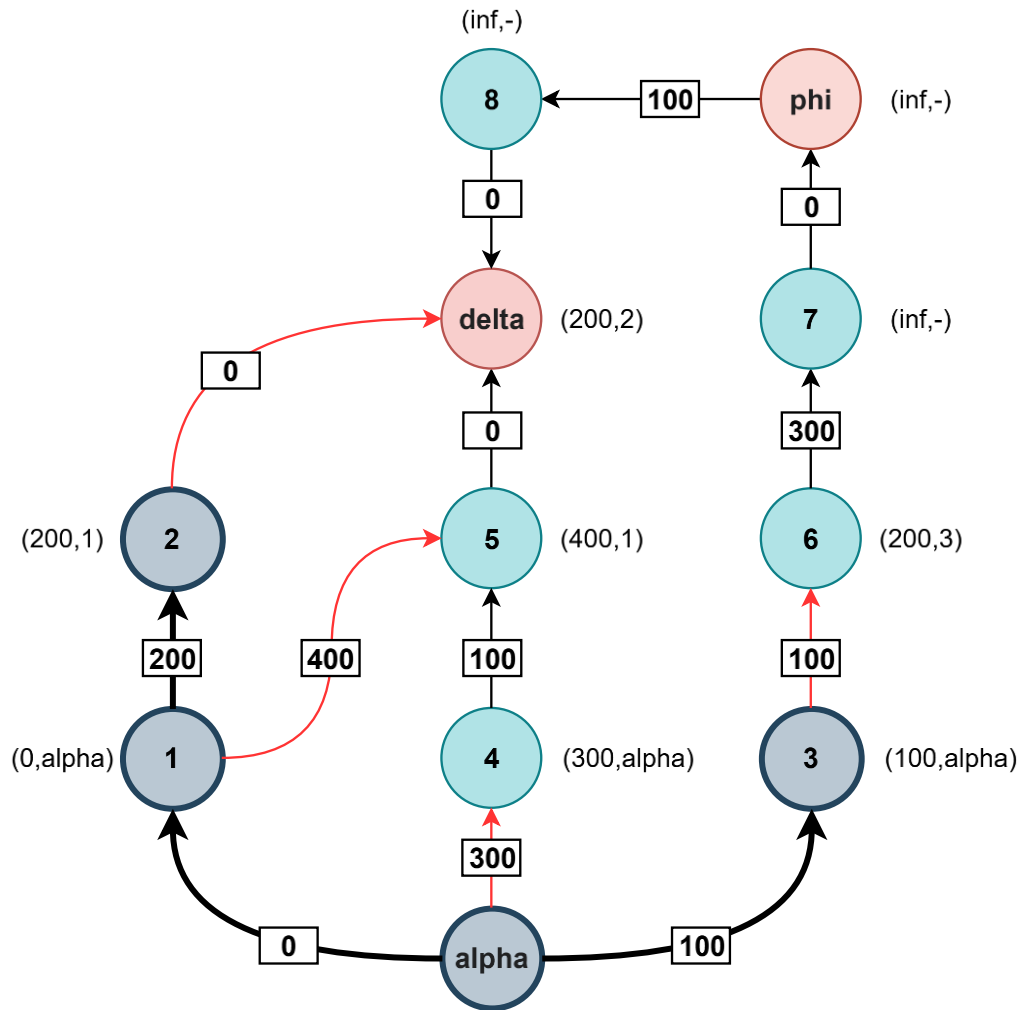


Figure 6.10 - Example 1 - Dijkstra execution step 4

Figure 6.10 shows the step 3 of the execution of Dijkstra, at this iteration Dijkstra will choose to expand the contact delta, as there is still a tie between node 6 and node delta with a cost of 200, but for the reasons said before, the contact delta will be expanded.

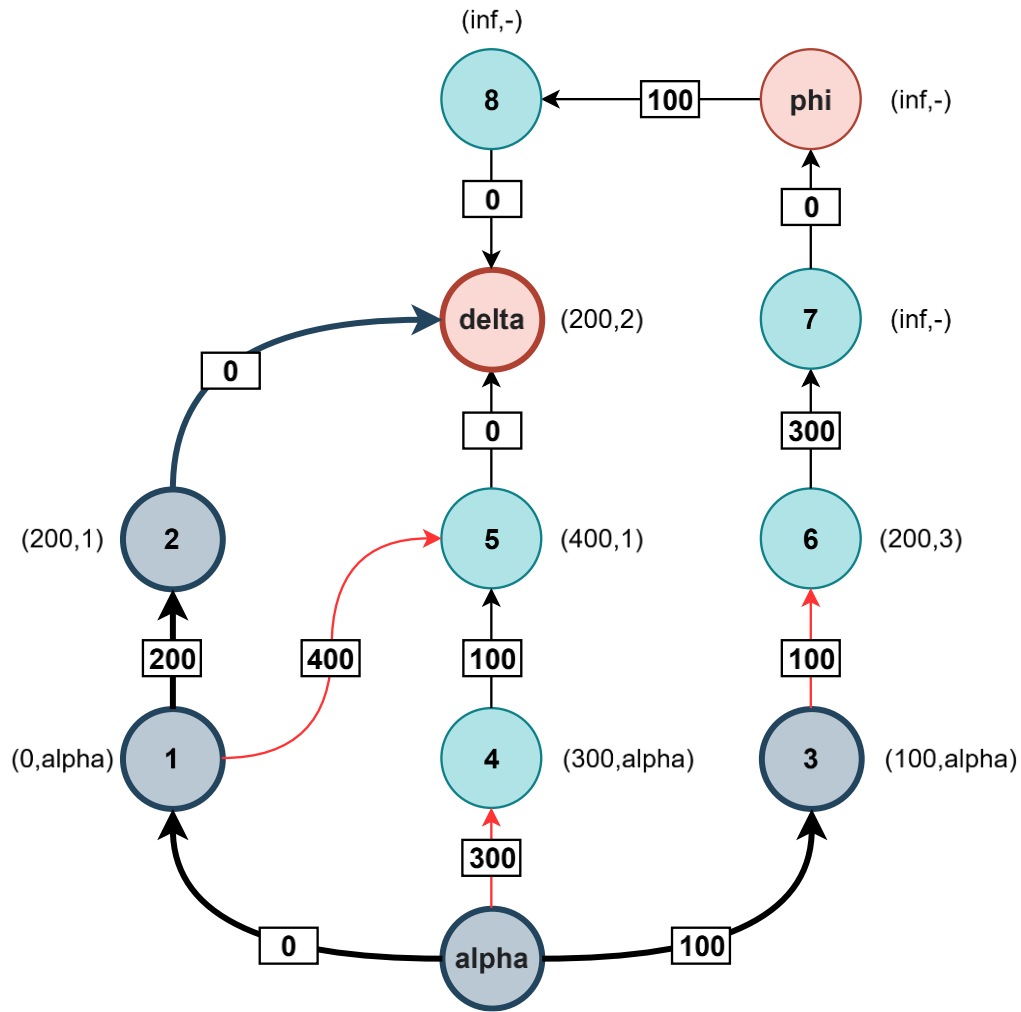


Figure 6.11 - Example 1 - Dijkstra execution step 5, end

Figure 6.11 shows the end of the execution of the Dijkstra algorithm on the anycast contact graph, as by expanding graph nodes we found a destination, therefore the contact 'delta' and so the bundle node D is the closest one to the source. The shortest path found by Dijkstra is the **contact series "alpha -> 1 -> 2 -> delta"** outlined grey in figure, this means that the anycast bundle will go through the DTN nodes A -> B -> D (see network topology in Figure 6.4).

If another valid anycast destination would have been encountered during the shortest path computation, that node would have been identified as the closest one. This outcome is consistent with the **correctness** property of the Dijkstra algorithm, as discussed in Section 6.4.1.

6.4.3.2 Example 2

Contact	Sender	Receiver	From	Until	Rate (Kbps)	Propagation Delay (ms)
1	A	B	1100	1200	1000	0
2	B	D	1400	1500	1000	0
3	A	C	1000	1100	1000	0
4	A	B	1300	1400	1000	0
5	B	D	1700	1800	1000	0
6	C	E	1100	1200	1000	0
7	E	F	1300	1400	1000	0
8	F	D	1600	1700	1000	0

Table 6.3 - Example 2 - contact plan for contact graph in Figure 6.5

In this example we will use the same contact graph as the one show in Figure 6.5, but the contact plan will be different, the new contacts are shown in Table 6.3.

The objective is still the same as Section 6.4.3, transmit a **DTN anycast bundle** with the “iac” EID “**iac:0.2.1**”. The possible recipients, with this new contact plan, are still nodes D and F.

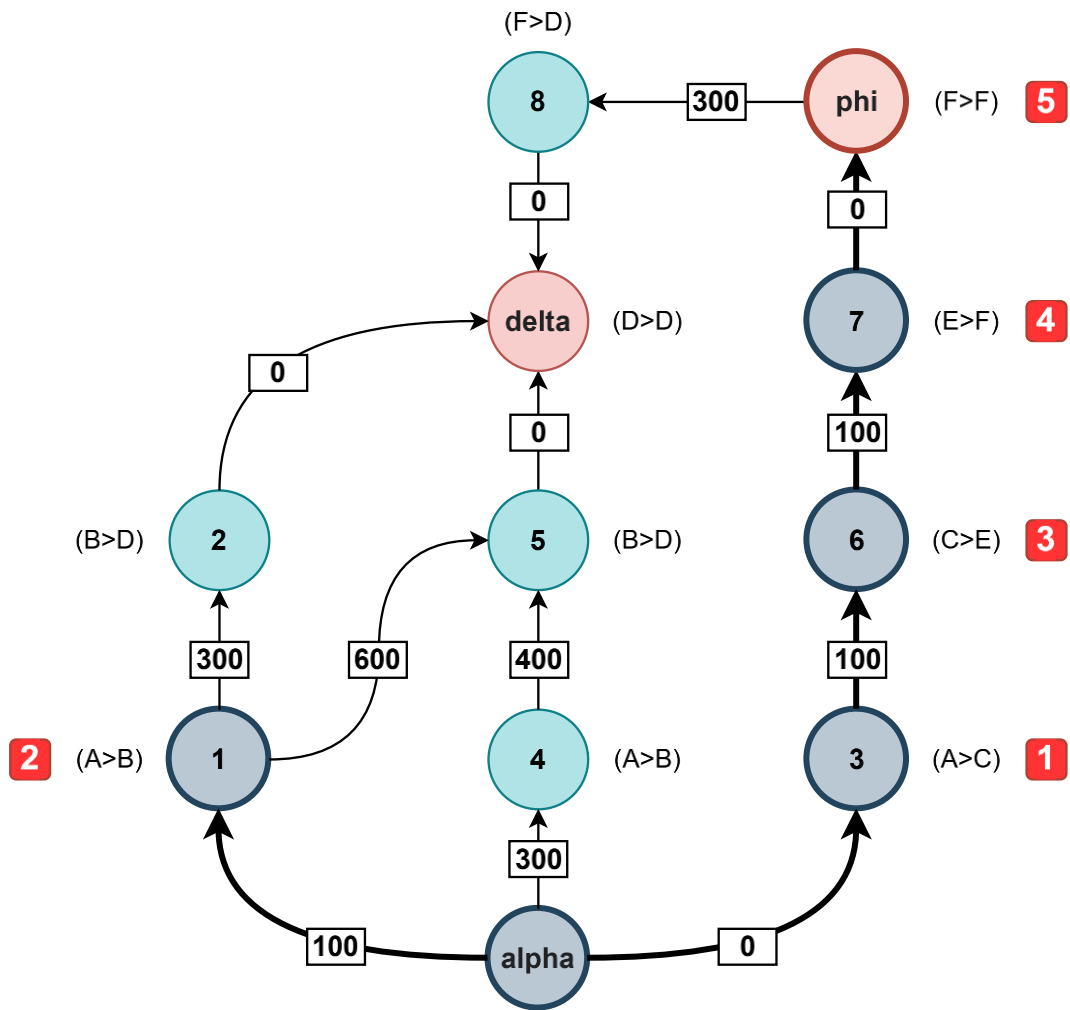


Figure 6.12 - Contact Graph showing execution of Dijkstra algorithm and retention times, shows example 1 of CGR of an anycast bundle

Figure 6.12, shows the evaluation of the Dijkstra algorithm on this new example. This time the algorithm first checks and finalizes the contact 'phi', therefore selecting destination node F first. When this contact is finalized, the Dijkstra algorithm is stopped, as this is a valid recipient and the outcome is the **series of contacts "alpha -> 3 -> 6 -> 7 -> phi"**, that it is equivalent to the series of nodes A -> C -> E -> F.

As previously discussed, Dijkstra's algorithm finalizes graph nodes in order of increasing distance from the source. Therefore, if the algorithm expands a secondary destination node first, it indicates that this secondary destination node is in fact the closest destination to the source. This behaviour is guaranteed by the correctness property of the Dijkstra algorithm [DIJKSTRA].

6.4.4 Second Possible Solution

Another possible solution to the integration of DTN anycast Contact Graph Routing involves a naiver solution, that is easier to integrate into current implementations of CGR/SABR, but at the cost of computational speed.

The solution is, for each possible anycast destination node, execute the Dijkstra or Yen's algorithm to get the shortest paths to each destination node and then compare their distances to get the true closest one.

This algorithm is simple and easily introducible into implementation of CGR although, as said before, it is computationally heavier than the one proposed in Section 6.4.2. The related computational problems are two:

- It computes **N shortest paths instead of one** where N is the number of possible DTN anycast destination nodes (N paths for Dijkstra; in the case of Yen's algorithm, its $K \times N$), which increases the route computation time by a constant factor (actually more, as each of these are increasingly heavier) from the solution proposed in Section 6.4.2. In the current SABR implementation routes are recalculated only when the DTN network topology changes, thereby reducing overall computational load, the resulting overhead remains a relevant consideration.
- Potential DTN anycast destination nodes may be located far from the source node, for instance, in a scenario involving a globally (or universal) distributed group of DTN CDN servers, computing a path to such distant anycast destinations would require substantial processing time, as the Dijkstra algorithm prioritizes **correctness over speed**. Consequently, in these cases, Dijkstra route computation could significantly degrade performance.

7 THE CADU AND CLTU CLEs

As discussed in Section 5.1, CCSDS established a comprehensive framework for the formatting, transmission, and management of space mission data using standardized packets and frames. The layer defined by CCSDS, known as the Space Data Link Layer, serves as the space communications “counterpart” of the Data Link Layer in the OSI reference model, which uses CCSDS frames as data unit.

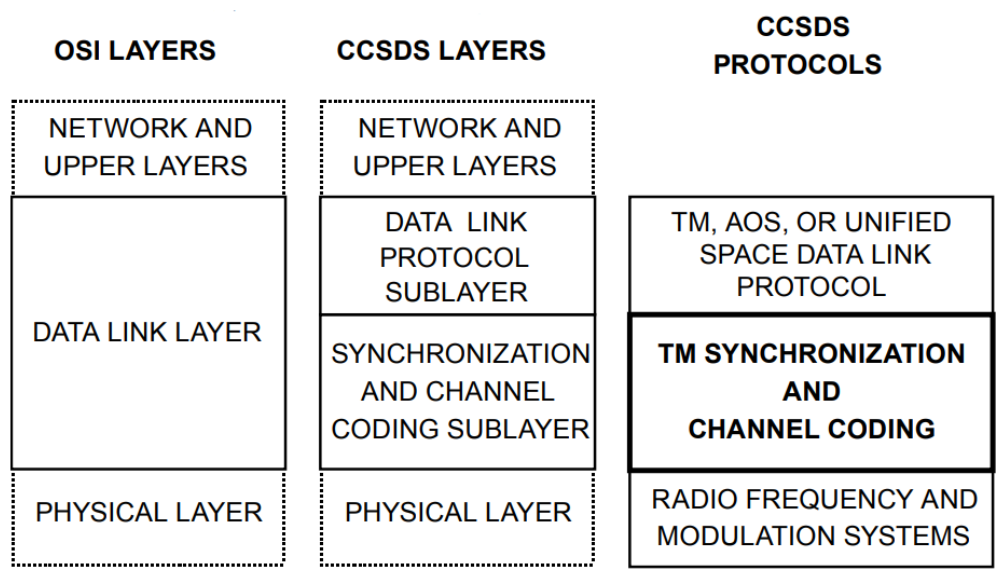


Figure 7.1 - CCSDS Data Link layer division, from [CCSDS_CADU]

As we can see from Figure 7.1, the CCSDS Space Data Link Layer defined by CCSDS is further divided into two sublayers:

1. **Data Link Protocol Sublayer:** provide functions for transferring data using the protocol data unit called the Transfer Frame, described in Section 5.1.
2. **Synchronization and Channel Coding Sublayer:** provides additional functions necessary for transferring Transfer Frames over a space link. These functions are error-control coding/decoding, Transfer Frame delimiting/synchronizing, and bit transition generation/removal. This layer is more a **physical layer** than a **data link layer**, in fact, the standard inputs of this layer are transfer frames, but in theory, we can put any byte stream, as the internal mechanism of this sublayer uses sub-protocols that encode and decode

a generic sequence of bit, without handling the header or values of any transfer frames in specific.

My work was to **design** and **implement** a system that could integrate the Synchronization and Channel Coding Sublayer in the current implementation of ESA Bundle Protocol v3, including both the sublayers for TM and TC frames.

The solution was to implement these two protocols as CLEs (described in Section 2.4) within the ESA BP framework, namely CADU CLE (Channel Access Data Unit) (for TM frames [CCSDS_TM]) and CLTU CLE (Communications Link Transmission Unit) (for TC frames [CCSDS_TC]), leaving the implementation open and prone to future updates and expansion.

A part of the work was also to provide a possibly fast implementation of the sub-coding methods internal to each of these two sublayers, such as Reed Solomon (RS) [Solomon_1960] [CCSDS_TM], and Bose–Chaudhuri–Hocquenghem (BCH) coding [Bose_1960] [CCSDS_TC], that led to a fast bitwise Java implementation of these two coding, within the ESA BP library.

7.1 Channels Coding Structure

As mentioned in the previous Section, the Synchronization and Channel Coding Sublayers exist for both TM and TC frames, adding functionality for error-control, coding/decoding Transfer Frame for both protocols. The specification of these two protocols is well-defined in their respective blue book [CCSDS_CADU] [CCSDS_CLTU].

These two protocols hold similarities in their structure and their behaviours, so I will first introduce the general structure they use to then move on more fine-grained details about the internal coding mechanisms they use and how they work, for each of them.

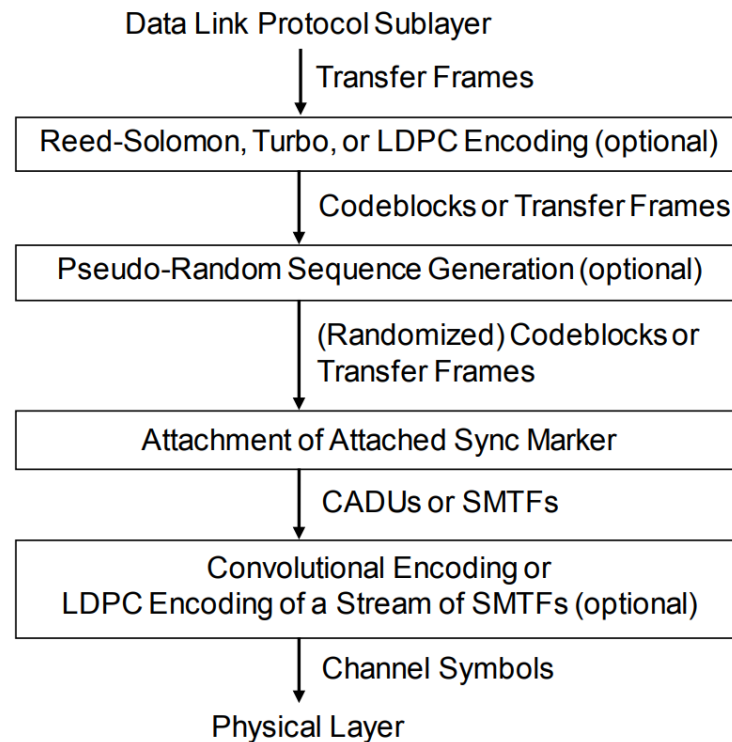


Figure 7.2 – TM Synchronization Channel Coding Sublayer, from [CCSDS_CADU]

As we can see from Figure 7.2, the CCSDS Channel Coding Sublayer has a layered structure, the input of the protocols stack are Transfer Frames (TC or TM) and the output are the channel symbols for the physical layer.

The input Transfer Frames are subsequently processed through a **series of layers** which are coding layers, each serving a specific function, such as error correction, synchronization, or framing. Additionally, some of these layers may segment the original Transfer Frame into smaller data units, necessitating a reconstruction mechanism at the receiving end, which receives only a contiguous stream of bits.

Bit delimitation within this symbol stream is achieved through specialized data units known as CADU (Channel Access Data Unit) or CLTU (Communications Link Transmission Unit). Both introduce distinctive **synchronization markers** into the symbol stream, enabling the receiver to identify frame boundaries, correctly segment the data, and process the frames in sequence.

This is why the name of the CLEs is CADU and CLTU, as it is the “protocol data unit” of each of the Synchronization and Channel Coding layers, even if the name is not

semantically correct, it emphasizes what the two CLE handles, giving them a meaningful name.

Most of the sub-coding in the TM and TC channel coding stack are well-known telecommunication standards or CCSDS defined protocols, most of these are also represented by a logic or block schematic diagram that depicts their logic-gate structure. As said in previous section, in theory, this allows for an **electrical equivalent** implementation of the entire channel coding stack, as these codings handles sequence of bits, not Transfer Frames directly.

However, in ESA's implementation, this approach was intentionally avoided. Instead, the channel coding components were integrated directly within the ESA BP stack to simplify configuration, even at the cost of some computational efficiency.

7.2 CADU CLE

7.2.1 TM Synchronization and Channel Coding

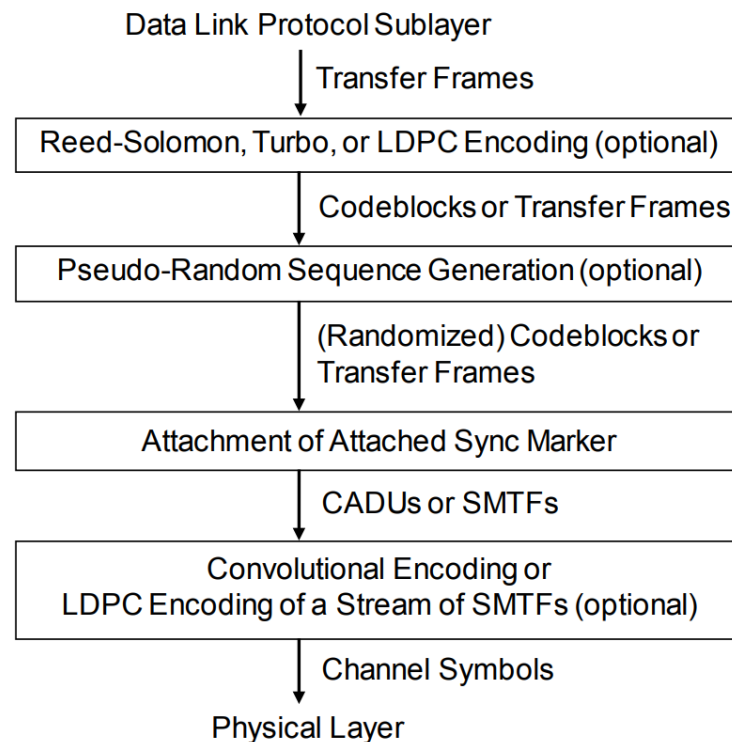


Figure 7.3 – TM Synchronization and Channel Coding, from [CCSDS_CADU]

The Synchronization and Channel Coding Sublayer provides three main functions, each one given by a layer of the coding stack (as we can see from Figure 7.3, from top to bottom), for transferring Transfer Frames over a space link:

- a) **Error-control coding, including frame validation:** The first layer in the stack is the most important one, as it implements error-control codes. To do so, coding mechanism such as Reed-Solomon [Solomon_1960], Turbo [Berrou_1993], or LDPC (Low-Density Parity-Check) [Gallager_1962] codes are used. These codes have been standardized by the CCSDS for both telemetry and telecommand application in [CCSDS_CADU]. Each of this coding mechanism computes a codeword that is added to the Transfer Frame, as shown in Figure 7.4, providing error detection or correction. Additionally, on the decoding side of the mechanism, some coding can determine (with very high probability) whether a frame is valid before returning it to the upper layers; this can be done using RS (Reed-Solomon) and LDPC decoders.

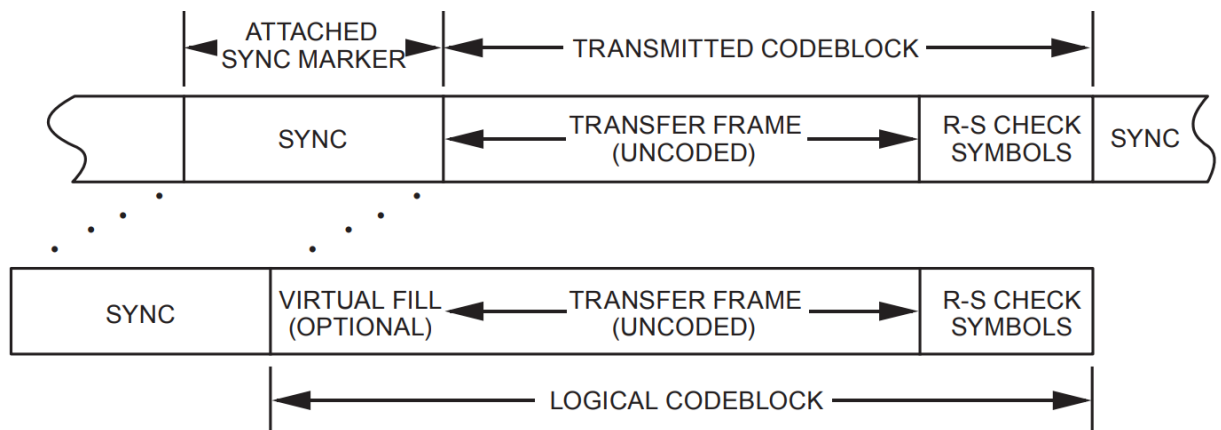


Figure 7.4 – RS Encoded with ASM Transfer Frame, from [CCSDS_CADU]

- b) **Synchronization:** Frames or code blocks synchronization is necessary for proper decoding of Reed-Solomon, Turbo, and LDPC codewords, and subsequent processing of the Transfer Frames. Furthermore, it is necessary to synchronize the pseudo-random generator if used. The [CCSDS_CADU] specifies a method for synchronizing Transfer Frames using an Attached Sync Marker

(ASM), as shown in the third layer of Figure 7.3. The ASM attached to the coded Transfer Frames depends on the configured error-control code.

- c) **Pseudo-Randomizing:** For the receiver system to work correctly, every data capture system at the receiving end requires that the data stream be sufficiently random and that the incoming signal have sufficient transition density to allow proper synchronization of the decoder. Pseudo-Randomizer is a component defined in Section 10 of [CCSDS_CADU] and is the preferred method to ensure sufficient randomness for all combinations of CCSDS-recommended modulation and coding schemes.

These are the three main functions of the TM Synchronization and Channel Coding. However, as we can see from Figure 7.3, not all layers are mandatory; in fact, only the ASM attaching layer is mandatory, the presence or absence of some sub-protocols is managed for a given physical channel (i.e., must be known a priori for the given channel by the sender and the receiver).

7.2.2 Design and Implementation

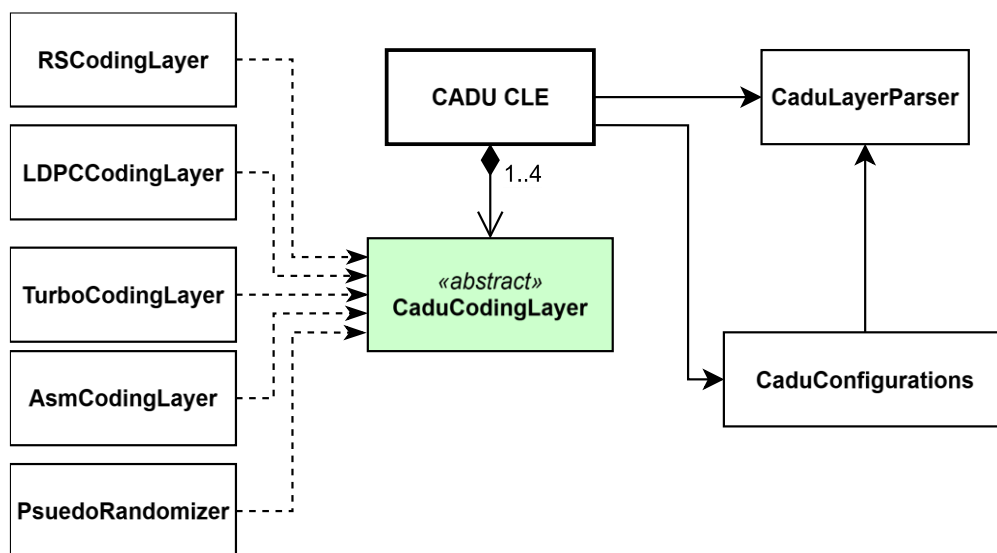


Figure 7.5 – CLTU CLE UML

As shown in Figure 7.5, the design and implementation of the CADU CLE inside the ESA BP library reflects its layered architecture, where each coding layer is implemented as a modular component that can be flexibly integrated into the TM channel coding stack

according to the client's needs. This modular structure enhances the code's extensibility and reusability, allowing new coding layers to be added easily in the future by simply creating a new CaduCodingLayer module.

The classes in the figure are:

- **CADU CLE:** The Convergence Layer Element component itself, containing all the other classes and usable inside an ESA BP CLA, as described in Section 2.4. Usually the CADU CLE will be placed directly beneath a Transfer Frame CLE in the ESA BP CLA configuration, as this CLE represents its Synchronization and Channel Coding layer, for instance, a possible CLA configuration could be composed of a Generic Packetiser (Section 5) CLE and, below, a CADU CLE. Indeed, an integration of the CADU and CLTU mechanisms directly into the Generic Packetiser could be desired.
- **CaduCodingLayer:** A Java abstract class that represents a generic coding layer within the CADU CLE; as described in Section 7.2.1 every layer adds a coding to the original Transfer Frame or to the PDU of the previous coding layer, in both encoding and decoding. This is why this abstract class exposes two main abstract functions

```
public abstract List<byte[]> encode(byte[] byteStream) throws CodingException;
```

- The **encode()** method processes the input byte stream, which initially consist of the byte stream representation of the Transfer Frame. It then applies the specific encoding operation for the given layer and output the encoded data. The method returns a list of byte streams, as the Coding Layer could also slice the original Transfer Frame into shorter PDUs (as said in Section 7.2.1).

```
public abstract List<byte[]> decode(byte[] byteStream) throws CodingException;
```

- The **decode()** method processes the input byte stream, which initially consists of the channel symbols provided by the physical layer. It then applies the specific decoding operations for the given layer and outputs

the decoded data. The method returns a list of byte streams, because, as mentioned earlier, the encoding process may fragment the original Transfer Frame into smaller PDUs. Consequently, the method may need to buffer incoming bytes if they result insufficient to reconstruct a complete layer-specific PDU, meaning that a layer can return zero or one PDU per call.

- **ReedSolomon, Turbo, LDPC, Asm, ..., CodingLayers:** The concrete classes for the coding layers implement their relative encoding and decoding mechanism. At the time of the task assignment, the required coding layers were only the Reed-Solomon and ASM layers for CADU. As said in previous sections, developing an optimized, fast bitwise-based implementation of Reed-Solomon was also part of my work; this efficient version is now available throughout the ESA BP framework. Additional coding layers can be easily integrated in the future, as the modular design of the system supports straightforward extension and integration.
- **CaduLayerParser:** This component parses the configured layers by the client for the CADU CLE; a list of coding layers with their relative managed parameters (specified in [CCSDS_CADU]), if present, must be provided by the user, within the configuration file specified in Section 3.3.2.

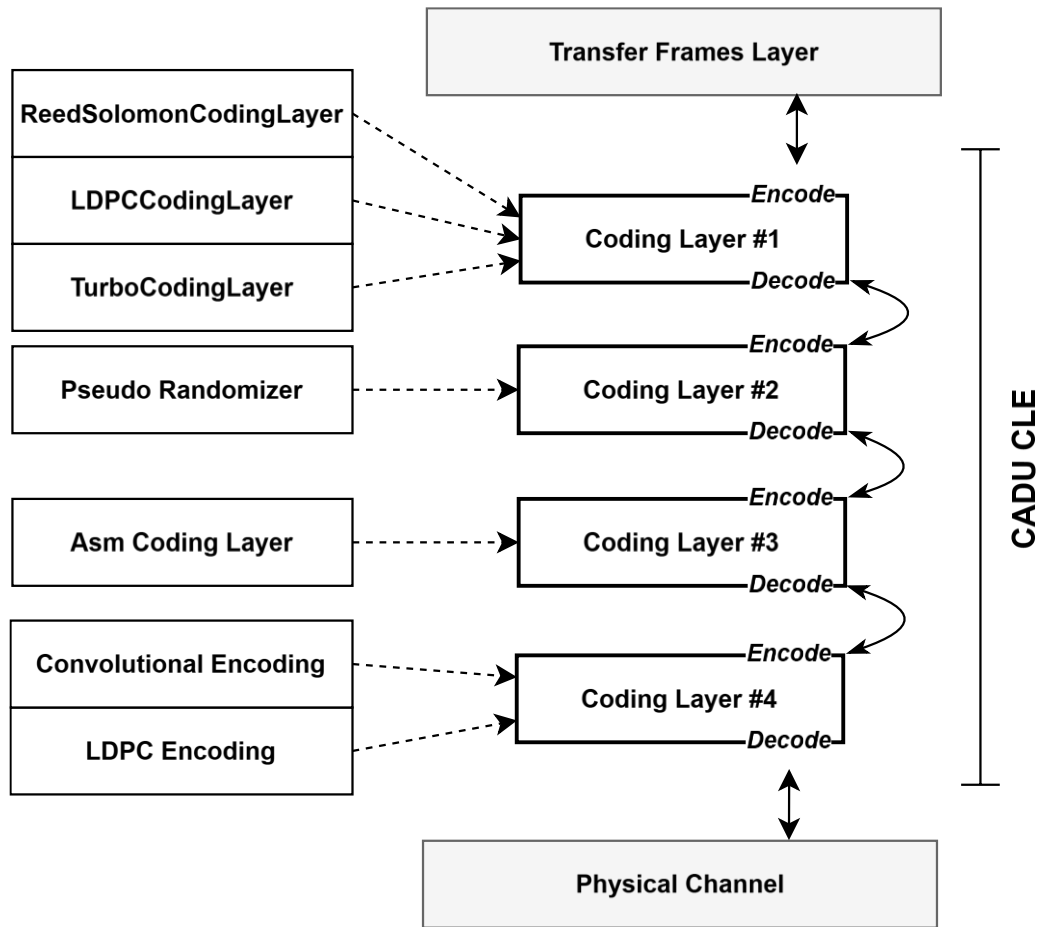


Figure 7.6 – Functional Structure of the CADU CLE

The logical structure of the CADU CLE is illustrated in Figure 7.6. It includes four configurable TM Channel Coding Layers, reflecting the Channel Coding subprotocol stack structure shown in Figure 7.3. To each layer the user can assign a specific coding component (RS, LDPC, TURBO,...) corresponding to its level in the stack, as each level supports a defined range of subprotocols.

Once configured, encoding a Transfer Frame involves sequentially passing it through the *encode()* methods of the four (or less, if not all configured) coding layers, from the first down to the fourth, ultimately producing the channel symbols for the physical layer. Decoding follows the reverse process, using the *decode()* methods starting from the physical layer and proceeding upward until valid Transfer Frames are reconstructed at the top of the stack.

7.3 CLTU CLE

7.3.1 TC Synchronization and Channel Coding

The TC Synchronization and Channel Coding Sublayer provides functions necessary for transferring Transfer Frames over a space link. These functions are error-control coding/decoding, codeword delimiting/synchronizing, and bit transition generation/removal. The architecture holds similarities with the TM Synchronization and Channel Coding Sublayer explained in Section 7.2.1, the overall structure and protocol is simpler, as the structure only depends on the selected error-control code, (as we can see from Figure 7.7 and Figure 7.8) and, within this architecture, there are fewer subprotocols configurable.

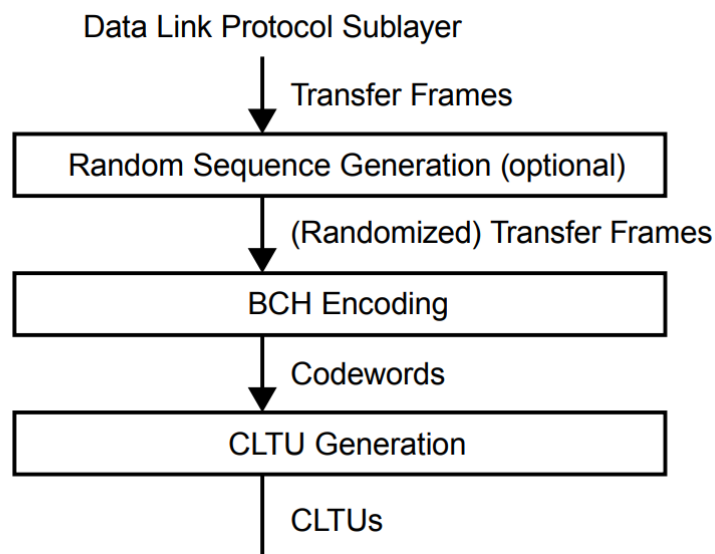


Figure 7.7 - Internal Organization of the Sublayer when BCH Coding is Used, from [CCSDS_CLTU]

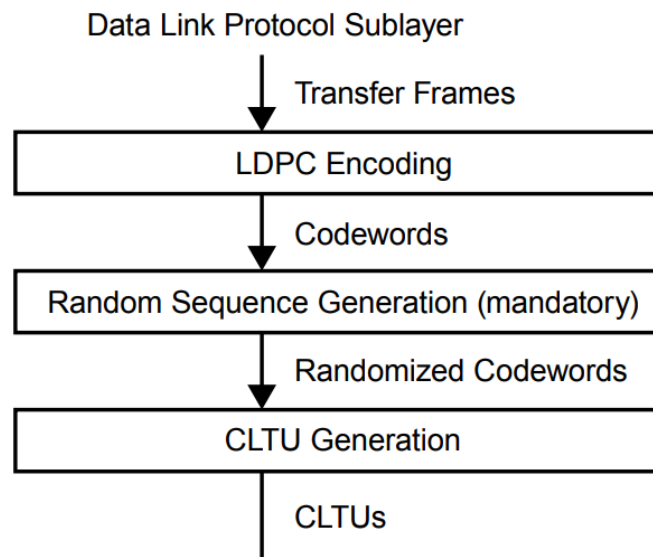


Figure 7.8 - Internal Organization of the Sublayer when LDPC Coding is Used, from [CCSDS_CLTU]

The TC Synchronization and Channel Coding Sublayer provides the following four functions for transferring Transfer Frames over a space link:

- a) **error-control coding:** The TC Sublayer defines two error-control coding methods. One uses a modified **BCH** (Bose–Chaudhuri–Hocquenghem) [Bose_1960] code (as described in Section 3 of [CCSDS_CLTU]); the other uses **LDPC** (Low-Density Parity-Check) [Gallager_1962] codes. The modified BCH code may be decoded either in an error-detecting mode or in an error-correcting mode, depending on mission requirements. The LDPC codes are typically decoded from soft symbols and operate at a lower Signal to Noise Ratio. These two error-control methods define two different Channel Coding stacks, in fact, the BCH coding stack is shown in Figure 7.7, while the LDPC coding stack is illustrated in Figure 7.8.
- b) **synchronization:** A method for synchronizing BCH or LDPC codewords is required, this is done using data units called Communications Link Transmission Units (CLTU), which consists of a Start Sequence, BCH or LDPC codewords, and a Tail Sequence (optional for LDPC)
- c) pseudo-randomizing, repeated transmissions are optional in the TC Frames Channel Coding stack.

The CCSDS specifications also include a section on Physical Layer Operations (PLOP), which, for evident reasons, cannot be represented at an informational or software level. Within ESA, these operations were not required and are therefore excluded from both the following explanation and the code implementation, for the time being.

7.3.1.1 Receiving Part

The sending part of TC Synchronization and Channel Sublayer is identical to the TM counterpart, and I will omit it from the explanation, however the receiving part is different. In the receiving part the CLTU processor is treated in the specifications like a state machine.

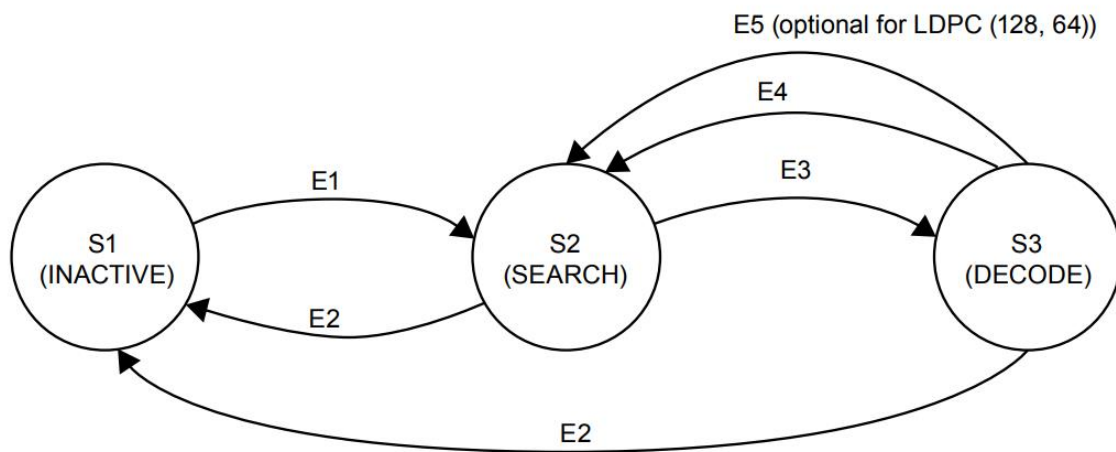


Figure 7.9 - CLTU Reception State Diagram (Receiving End), from [CCSDS_CLTU]

There are two important states:

- **S2, SEARCH:** The CLTU receptor is searching, bit by bit, the incoming bit stream, actively looking for the Start Sequence bit pattern that we put at the start of each BCH/LDPC codeword at the sending end.
- **S3, DECODE:** Codewords are decoded from the incoming bit stream, the ones free of errors or corrected, are transferred to the sublayer above (usually a TC frame data link layer). In this state the CLTU receptor also look for the Tail Sequence, so that it knows when the sequence of codeword will end.

The main events are:

- **E3, START SEQUENCE FOUND:** The Start Sequence pattern has been detected
- **E4, CODEWORD REJECTION:** The decoder has indicated uncorrected errors. No data from this decoding attempt is transferred to the sublayer above.
- **E5, TAIL SEQUENCE FOUND:** The Tail Sequence has been detected, signalling the end of a CLTU. The search for the Start Sequence resumes at the end of the Tail Sequence.

In the schema, there are also the active or inactive states, their explanation was skipped as this state is given by the physical operations mentioned earlier, therefore they are omitted in both explanation and implementation.

7.3.2 Design And Implementation

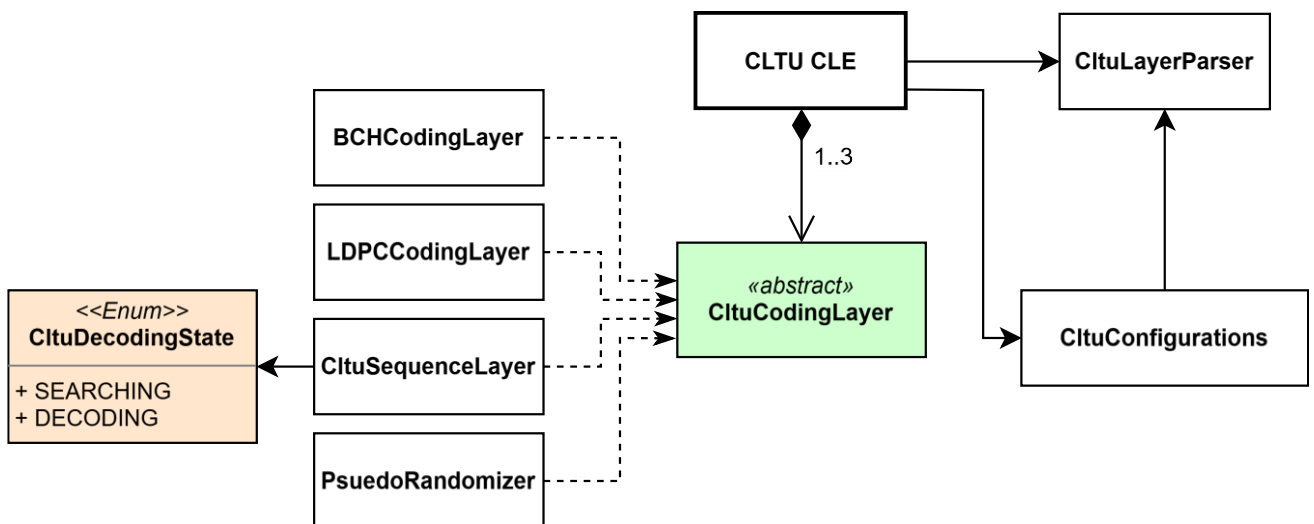


Figure 7.10 - CLTU CLE UML

The design of the CLTU CLE is shown in Figure 7.10, the design and implementation of the CLTU CLE reflects its layered architecture, where each coding layer is implemented as a modular component that can be flexibly integrated into the TC channel coding stack according to the client's needs.

The components within the CLTU CLE, hold the same meaning and purpose as the CADU CLE in Section 7.2.2, so I will not explain in detail the similar ones.

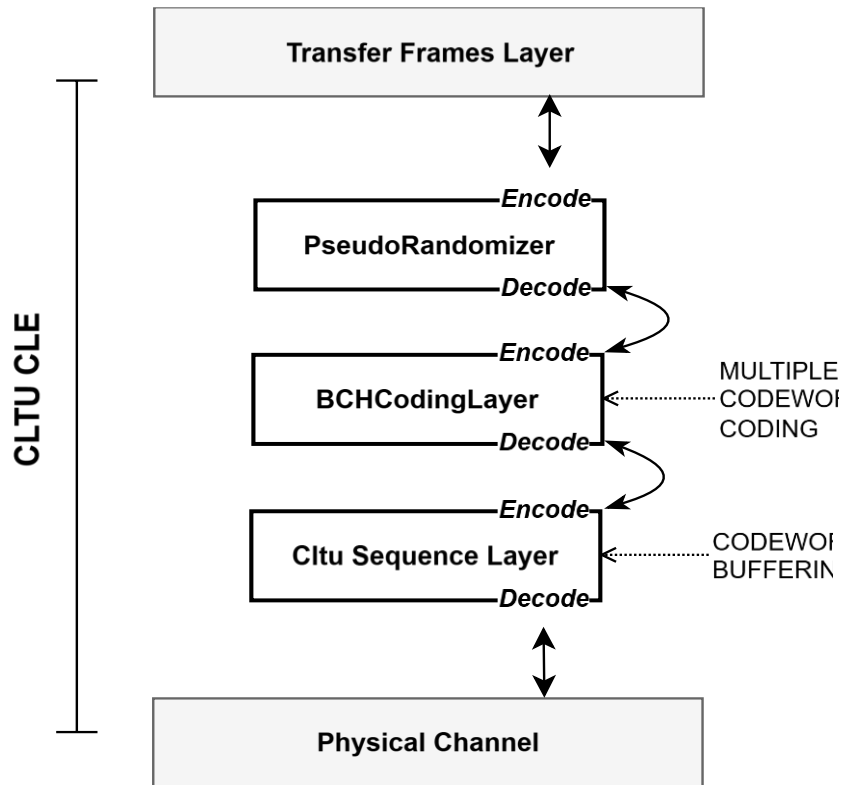


Figure 7.11 - CLTU CLE Coding Flow Diagram (BCH)

The main difference in the structure and functionality is in CLTU Sequence Layer, in both encoding and decoding part. The Figure 7.11 illustrate an example of CLTU CLE coding stack with BCH error code configured. In the Figure:

- Encoding part:
 - First, the Transfer Frames bits are randomized using the pseudo randomizer (this is optional), this doesn't change the length of the byte stream of the original Transfer Frames, just changes the values

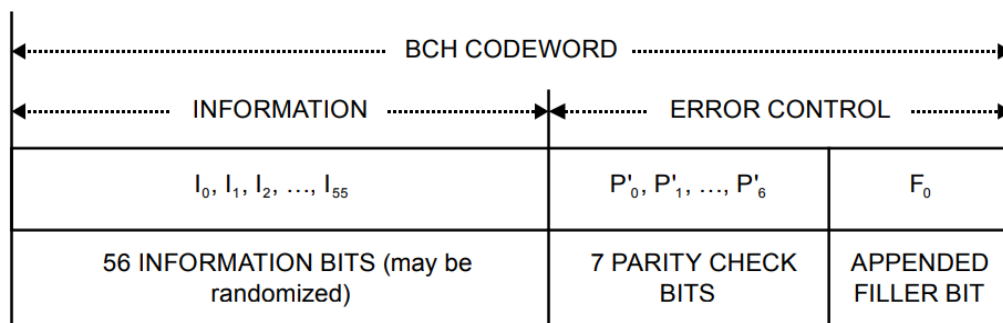


Figure 7.12 - BCH Codeword

- Secondly, the new bit stream is encoded using the BCH error-control code. The CLTU employs a modified version of BCH with a (63,56) code, as described in Section 3 of [CCSDS_CLTU]. As shown in Figure 7.12, for every 56 information bits, BCH generates 7 parity-check bits, which are appended to the data along with a fill bit. Consequently, the BCH coding layer divides the input byte stream into 56-bit segments and adds 8 bits to each, resulting in a new byte stream composed of a sequence of 64-bit BCH codewords.
- Last, the CLTU sequence layer will append to the codewords sequence a BCH-specific initial and a tail sequence.
- Decoding part:
 - First, the bit stream coming from the physical channel, will be the input in the decoding method of the CltuSequenceLayer, this, as previously described, act as a state machine, and has two different behaviours depending on its state:
 - **SEARCHING STATE:** The CltuSequenceLayer continuously scans the input bit stream for a valid BCH initial sequence. Once detected, it indicates that a sequence of N BCH codewords will follow. At this point, the component transitions to the DECODING state.
 - **DECODING STATE:** In this state, the CltuSequenceLayer acts as a buffer, storing each BCH codeword until the BCH tail sequence is detected in the bit stream. When the tail sequence is found, all buffered BCH codewords are passed to the upper layer, and the component returns to the SEARCHING state.
 - Any unrecognized symbol encountered in either state is discarded.

- Secondly, the sequence of N BCH codewords is passed to the BCHCodingLayer decoder. The decoder processes each codeword, verifying the validity of its parity check bits. Codewords that pass this verification are forwarded to the upper layer, ultimately reconstructing a complete and valid Transfer Frame, provided that no pseudo-randomization step is required.

8 CONCLUSIONS

This thesis presented a set of developments and design contributions carried out during a six-month internship at the European Space Agency (ESA), with the aim of extending and improving the ESA Bundle Protocol version 3 (ESA BPv3) implementation, by enhancing its modularity, interoperability, and compliance with DTN and CCSDS standards. The work covered the definition and implementation of four main elements: the *File Convergence Layer Element (File CLE)*, the *Generic Packetiser (GP)*, the *Interplanetary Anycast Communication (IaC)* scheme, and the *CADU* and *CLTU* convergence layer elements. All of them are written in Java, the language of ESA-BPv3.

The *File CLE* introduced a practical method for exchanging bundles via standard filesystems, enabling simplified interoperability and testing across different systems and agencies. The *Generic Packetiser* provided a unified framework for handling CCSDS packets and frames, improving the flexibility and scalability of ESA BP by integrating new CCSDS recommended standards. The DTN Anycast (*IaC*) scheme extended the Bundle Protocol with anycast capabilities, standardizing the anycast communication paradigm for DTNs, allowing the delivery of a bundle to one of multiple eligible destinations. Finally, the *CADU* and *CLTU* CLE implemented the lower-level channel coding mechanisms defined by CCSDS (Synchronization and Channel Coding Sublayer), which provide the necessary functionalities for transferring Transfer Frames over a space link and offer a modular structure that facilitates the addition of new coding layers in future developments.

As a whole, these contributions strengthen ESA BPv3 as a research and operational platform for Delay/Disruption Tolerant Networking and CCSDS-compliant communication systems. As a computer engineer, one of the main goals was to enforce software modularity, to facilitate easier maintenance, future proof implementations, faster integration of new protocols, and improved testing capabilities.

Future developments could focus on integrating anycast into specialized DTN routing protocols such as CGR/SABR, expanding IaC testing to more complex DTN networks,

and optimizing the performance of the implemented coding layers. Overall, the hope is that this work will contribute to the design of flexible, interoperable, and resilient communication frameworks for future space missions.

BIBLIOGRAPHY

-
- [ANYCAST] Public, <https://commons.wikimedia.org/w/index.php?curid=909336>
- [Berrou_1993] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo codes," in *Proc. IEEE Int. Conf. on Communications (ICC '93)*, vol. 2, Geneva, Switzerland, pp. 1064–1070, May 1993.
- [Bose_1960] R. C. Bose and D. K. Ray-Chaudhuri, "On a class of error correcting binary group codes", *Information and Control*, 3, pp.68–79, 1960.
- [BP_AR] Department of electrical and computer engineering software and application development sector accurate estimation of end-to-end delivery delay in space internets: protocol design and implementation - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/Bundle-Protocol-overlay-position-within-the-DTN-protocol-stack-and-comparison-with-the_fig2_360485728 [accessed 16 Oct 2025]
- [Burleigh_2016] S. Burleigh, C. Caini, J. J. Messina and M. Rodolfi, "Toward a unified routing framework for delay-tolerant networking," 2016 IEEE International Conference on Wireless for Space and Extreme Environments (WiSEE), Aachen, Germany, 2016, pp. 82-86, doi: 10.1109/WiSEE.2016.7877309.
- [Caini_2011] C. Caini, H. Cruickshank, S. Farrell, M. Marchese, "Delay- and Disruption-Tolerant Networking (DTN): An Alternative Solution for Future Satellite Networking Applications," *Proceedings of the IEEE*, vol.99, no.11, pp.1980,1997, Nov. 2011, DOI: [10.1109/JPROC.2011.2158378](https://doi.org/10.1109/JPROC.2011.2158378)
- [CCSDS_AOS] Consultative Committee for Space Data Systems. *AOS Space Data Link Protocol* (CCSDS 732.0-B-4, Blue Book). Washington, D.C.: CCSDS Secretariat, October 2021. [Online]. Available: <https://ccsds.org/Pubs/732x0b4.pdf>
- [CCSDS_BP] Consultative Committee for Space Data Systems. *CCSDS Bundle Protocol Specification* (CCSDS 734.2-B-1). Washington, D.C.: CCSDS Secretariat, September 2015. [Online]. Available: <https://ccsds.org/Pubs/734x2b1.pdf>
- [CCSDS_CADU] Consultative Committee for Space Data Systems. *Tm Synchronization and Channel Coding* (CCSDS 131.0-B-5, Blue Book). Washington, D.C.: CCSDS Secretariat, September 2023. [Online]. Available: <https://ccsds.org/Pubs/131x0b5.pdf>

[CCSDS_CLTU]	Consultative Committee for Space Data Systems. <i>Tc Synchronization and Channel Coding</i> (CCSDS 231.0-B-4, Blue Book). Washington, D.C.: CCSDS Secretariat, July 2021. [Online]. Available: https://ccsds.org/Pubs/231x0b4e1.pdf
[CCSDS_EPP]	Consultative Committee for Space Data Systems, <i>Encapsulation Packet Protocol</i> , CCSDS 133.1-B-3, Blue Book, Washington, D.C., May 2020. [Online]. Available: https://ccsds.org/publications/bluebooks/entry/3261/?utm_source=chatgpt.com
[CCSDS_PROTOCOL S]	Consultative Committee for Space Data Systems. <i>Overview of space communications protocols</i> (CCSDS 130.0-G-4, Green Book). Washington, D.C.: CCSDS Secretariat, April 2023. [Online]. Available: https://ccsds.org/Pubs/130x0g4e1.pdf
[CCSDS_SABR]	Consultative Committee for Space Data Systems. <i>Schedule-aware bundle routing</i> (CCSDS 734.3-B-1, Blue Book). Washington, D.C.: CCSDS Secretariat, July 2019. [Online]. Available: https://ccsds.org/Pubs/734x3b1.pdf
[CCSDS_SPP]	Consultative Committee for Space Data Systems, <i>Space Packet Protocol</i> , CCSDS 133.0-B-2, Blue Book, Washington, D.C., June 2020. [Online]. Available: https://ccsds.org/publications/bluebooks/entry/3264/?utm_source=chatgpt.com
[CCSDS_TC]	Consultative Committee for Space Data Systems. <i>TC Space Data Link Protocol</i> (CCSDS 232.0-B-4, Blue Book). Washington, D.C.: CCSDS Secretariat, October 2021. [Online]. Available: https://ccsds.org/Pubs/232x0b4e1c1.pdf
[CCSDS_TM]	Consultative Committee for Space Data Systems. <i>TM Space Data Link Protocol</i> (CCSDS 132.0-B-3, Blue Book). Washington, D.C.: CCSDS Secretariat, October 2021. [Online]. Available: https://ccsds.org/Pubs/132x0b3.pdf
[DIJKSTRA]	<u>Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2022) [1990]. "22". Introduction to Algorithms (4th ed.). MIT Press and McGraw-Hill. pp. 622–623. ISBN 0-262-04630-X</u>
[ESA_GP]	ESA Generic Packetiser version 2 Software Design Documentation (Internal Document, Reserved for ESA internal use only)
[ESABP_CIG]	ESA Bundle Protocol version 3 – Configuration Installation Guide (Internal Document, Reserved for ESA internal use only)
[ESABP_SDD]	ESA Bundle Protocol version 3 - Software Design Document (Internal Document, Reserved for ESA internal use only)
[ESABP_SUM]	ESA Bundle Protocol version 3 – Software User Manual (Internal Document, Reserved for ESA internal use only)

- [Gallager_1962] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.
- [GRPC] Google remote procedure call documentation for java [Online], Available: <https://grpc.github.io/grpc-java/javadoc/>
- [IETF_draft_BIBE] Burleigh, S., Montilla, A., Deaton, J., Caini, C., *Bundle-in-Bundle Encapsulation* [White paper], IETF Internet-Draft, March 2025, [Online], Available: <https://datatracker.ietf.org/doc/draft-ietf-dtn-bibect/>
- [IETF_draft_IAC] Cavallini, D., Flentge, F., DTN Interplanetary Anycast Communication Scheme. IETF Internet-Draft, September 2025, [Online], Available: <https://datatracker.ietf.org/doc/draft-cavallini-dtn-iac/>
- [JAVA] Java 11 Official Documentation [Online], Available: <https://docs.oracle.com/en/java/javase/11/docs/api/index.html>
- [MAVEN] Maven official site with documentation [Online], Available: <https://maven.apache.org/index.html>
- [Pettinato_2018] Master thesis on "Prototipazione di una architettura di tipo Delay-Tolerant Network operante su tecnologie di livello fisico eterogenee", from Carlo Pettinato 2018, [Online]. Available: <https://webthesis.biblio.polito.it/9509/1/tesi.pdf>
- [RFC_4838] Cerf, V., Burleigh, S., Hooke, A., Torgerson, L., Durst, R., Scott, K., Fall, K., and H. Weiss, "*Delay-Tolerant Networking Architecture*", RFC 4838, DOI 10.17487/RFC4838, April 2007, <<https://www.rfc-editor.org/info/rfc4838>>.
- [RFC_5326] Ramadas, M., Burleigh, S., and S. Farrell, "*Licklider Transmission Protocol - Specification*", RFC 5326, DOI 10.17487/RFC5326, September 2008, <<https://www.rfc-editor.org/info/rfc5326>>.
- [RFC_6260] Burleigh, S., "Compressed Bundle Header Encoding (CBHE)", RFC 6260, DOI 10.17487/RFC6260, May 2011, <<https://www.rfc-editor.org/info/rfc6260>>.
- [RFC_8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC_9171] Burleigh, S., Fall, K., Birrane, E. III, "*Bundle Protocol Version 7*", RFC 9171, DOI 10.17487/RFC9171, January 2022, <<https://www.rfc-editor.org/info/rfc9171>>.
- [RFC_9174] Sipos, B., Demmer, M., Ott, J., and S. Perreault, "Delay-Tolerant Networking TCP Convergence-Layer Protocol Version 4", RFC 9174, DOI 10.17487/RFC9174, January 2022, <<https://www.rfc-editor.org/info/rfc9174>>.

- [RFC_9758] Taylor, R. and E. Birrane III, "Updates to the 'ipn' URI Scheme", RFC 9758, DOI 10.17487/RFC9758, May 2025, <<https://www.rfc-editor.org/info/rfc9758>>.
- [Solomon_1960] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960. <https://epubs.siam.org/doi/10.1137/0108018>
- [YAML] Ben-Kiki, O., Evans, C., & Ingerson, B. (2009). *YAML Ain't Markup Language (YAML™) Version 1.2* [Specification]. YAML Language. Retrieved from <https://yaml.org/>
- [YEN] Jin Y. Yen. "Finding the K Shortest Loopless Paths in a Network." *Management Science* 17, no. 11 (July 1971): 712–716.
-