ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCHOOL OF ENGINEERING AND ARCHITECTURE

Department of Computer Science and Engineering

Two-years Master's degree in computer engineering

MASTER'S THESIS
in
Protocols and architectures for space networks

A Unified Application Programming Interface for Bundle Protocol Applications

CANDIDATE
Silvia Lanzoni

SUPERVISOR
Prof. Carlo Caini

COSUPERVISORS
Dr. Keith Scott
Andrea Bisacchi

Academic Year 2020/2021

Session V

To my parents,
always with me.

Index

Abstract

Future space missions to the Moon and Mars should rely on the DTN architecture and its associated Bundle Protocol (BP) to cope with the challenges faced by interplanetary links, such as long delays and disruptions. This has led to new BP implementations or new releases, incorporating version 7 of the Bundle Protocol, recently standardized by IETF. Although the multiplicity of implementations is positive per se, this has exacerbated an already existing fragmentation problem, as every implementation has its own Application Programming Interface (API) and its own set of bundle options supported, which is a significant obstacle to the development of third-party applications.

To alleviate this problem, a new "abstracted" API for bundle protocol applications, called the Unified API, has been developed during this thesis, and it is presented here.

The Unified API is a complete refactoring of the old "Abstraction Layer". As with the Abstraction Layer, the Unified API aims to decouple a DTN application from the BP implementations, with great advantages in terms of portability, maintenance and interoperability. However, the Unified API is more ambitious, and it has the goal to also present a new API, now much closer to the familiar UDP socket and, more generally, to facilitate the development of new DTN applications as much as possible.

To realize this, the whole structure has been organized into three layers, of which only the top one is seen by applications. This layer consists of four blocks, denoted by different function prefixes. The most important is the "Socket" block, developed with the help of Andrea Bisacchi, which implements the UDP-like socket. The other blocks provide the programmer with high-level tools to easily create and modify a bundle, to parse the input string given by the user, to implement conditional print statements, and many others, as detailed in this document.

The Unified API is compatible with most BP implementations (DTN2/DTNME, ION, IBR-DTN, µD3TN). As with the BP implementations, it is released as free software and it is going to replace its predecessors in the next release of the DTNsuite, the Unibo collection of DTN applications included in ION, the DTN implementation by NASA's Jet Propulsion Laboratory (JPL).

The hope is that the Unified API will encourage third-party programmers to develop new multiplatform applications, thus contributing to the success of DTN networking.

1 INTRODUCTION

1.1 DTN ARCHITECTURE

The Delay-/Disruption-Tolerant Networking (DTN) architecture has been designed to overcome the limits of the TCP/IP protocol suite. Internet protocols are based on the following assumptions, which cannot be ignored:

- Short Round Trip Times (RTT)
- End-to-end connectivity
- Low error rates
- Symmetric link rates

If at least one of these assumptions is not verified, the network is said to be “challenged”, as TCP/IP protocols face serious difficulties in providing satisfactory performance, if any. These “challenged networks” can be found in space (interplanetary networks, LEO satellite networks) but also in a few peculiar terrestrial environments (military tactical networks, underwater networks, disaster response networks, communications in remote areas).

The DTN architecture [RFC4838] solves the problems posed by challenged networks by introducing a new communication layer called “Bundle Layer” between Application and Transport. This new layer is usually implemented on the end nodes and on some selected nodes.

To understand how the introduction of the Bundle layer can overcome TCP/IP limits, let us consider Figure 1, which could represent an interplanetary communication between two end nodes (e.g., one on Earth and one on Mars), through two relays. The end-to-end path between the two end nodes cannot be taken for granted as links are intermittent (because of planet rotations and spacecraft motion), thus each DTN node must be able to store the data to be sent for possibly long intervals (store phase), should the link to the next node be temporarily unavailable, before transmitting it (forward phase). This approach, typical of DTN, is called store and forward.

At the Bundle layer we have the “Bundle Protocol” (BP), whose data units are called “Bundles”. The BP was first specified by IRTF (Internet Research Task Force) in [RFC5050] (version 6, or “bpv6”) and very recently standardized by IETF (Internet Engineering Task Force) (version 7, “bpv7”). BP interfaces with the underlying protocol, usually at Transport layer, are called Convergence Layer Adapters (CLAs). As we can see from the figure, the role of Transport is redefined, as it is no longer end-to-end, but works between two consecutive DTN nodes (i.e., on one DTN hop). Among them we can have a variety of non DTN nodes, represented by the “Network” clouds in the figure. These non DTN nodes are not reported as transparent to the BP.

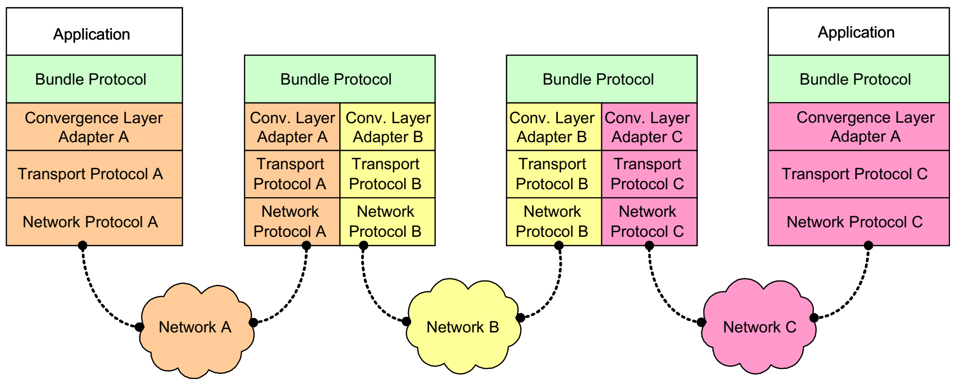


Figure 1. The Bundle Protocol location within the standard protocols stack

The key points of the protocol are:

- Storage at all nodes
- Redefinition of the Transport role
- Late binding of overlay network endpoint identifiers to a constituent Internet address
- Optional custodial transfer (bpv6 only)
- Reactive and proactive fragmentation
- Optional bundle tracking by means of Bundle Status Reports (BSRs)

The interested reader is referred to [RFC5050] and [RFC9171] for a comprehensive treatment.

More information about DTN can be found here: [DTN FOR ISS], [DTN: EVOLVING INTERPLANTERY INTERNET], [DTN VIRTUALIZED ENVIROMENT], [MARS-EARTH], [RATIONALE FOR DTN IN SPACE].

1.2 BP IMPLEMENTATIONS

The BP was designed to reduce as much as possible the bias towards the different architectures. For this reason, the protocol can be realized in different ways [RFC5050] and [RFC9171]:

- As an application server
- Through peer application nodes
- Through Sensor network nodes
- Through a dedicated bundle router

For our purposes, the most interesting realization is the first one where there is a single bundle protocol application server based on a single bundle node running as a demon process on each computer. The demon process includes all functions of the bundle protocol agent, all convergence layer adapters, and both the administrative and application-specific elements of the application agent [RFC 5050]. The applications can communicate with the demon using specific APIs.

The most important available DTN implementations are ION, DTN2, IBR-DTN and µD3TN.

ION (Interplanetary Overlay Network) [ION] is the implementation developed by NASA JPL and is mainly focused to space communications, for this reason, one of its most important features is the possibility to be efficiently used on embedded systems like planetary rovers and deep-space probes. Reliability, portability, and easy integration with heterogenous underlying communication infrastructure (like specific space communications links) are the major goals of this implementation. For safety reasons, it is based on a peculiar management of the memory, which has some impact on speed and code complexity.

DTN2 [DTN2] is the "reference implementation", its goal was to be a landmark for other implementations, but the official version has not been updated since 2011. NASA's Marshall Space Flight Center (MSFC), in 2016 provided a patch to keep it updated with features from ION, and recently released a fork, called DTNME (DTN2 Marshall Edition) [DTNME], which incorporates the previous patch and adds support for bpv7. DTN2/DTNME is written in C and C++.

IBR-DTN is an implementation developed by the University of Braunschweig [IBRDTN]. It is designed for embedded systems and can be used as a framework for DTN applications. It is written in C++ and can be deployed onto Android devices.

µD3TN is a recent implementation, previously known as µPCN, developed by the D3TN company [µDT3TN]. It was the first including bpv7 (draft version) and has recently been tested in space on LEO satellites.

All the above-mentioned implementations have been released as free software. Each implementation is different from the others, and this means that each implementation has its own specific API, its specific functions, and peculiarities. In particular, their support of bpv6 and bpv7 varies, as well as the support for extensions, such as Extended Class of Service Options (ECOS) [ECOS]. Having so many implementations on the one hand demonstrates a renovated interest in DTN architecture and BP, on the other hand it presents to third party programmers a very fragmented environment, which may represent a severe obstacle to the development of DTN applications.

1.3 The old "Abstraction Layer"

The Abstraction Layer (AL) is a library, written in C, to interface a DTN application with a BP node without worrying about the BP implementation running on the node. Its greatest advantage is the possibility to reuse the same application code on top of different BP implementations, which is obviously convenient in terms of application portability, maintenance, and interoperability. On the other hand, a possible weakness derives from the necessity of continuous maintenance, as the AL dependence on the BP implementation requires that any change in the implementation API be treated in the AL. After about a decade form its first version, a total refactoring of the code was in order. This is exactly the aim of this thesis.

The starting point for this thesis work is version 1.7.0, released in 2020 [AL_1_7_0].

This version supported DTN2, ION, and IBR-DTN, and the selection could be performed at compilation time, or, if compiled for multiple implementations, also at run time, a feature particularly useful in interoperability tests.

The AL functions were divided into three groups: basic, utility, and bundle functions.

The bundle functions aimed to manage the bundle (a C structure with many fields) as an "object", with specific functions for reading/writing each bundle field. As examples of bundle functions, we cite the two functions for reading and writing the bundle source EID (Endpoint Identifier):

```
al_bp_bundle_get_source
al_bp_bundle_set_source
```

Basic and utility functions build a direct abstraction of one or more implementation-specific functions, such as:

```
al_bp_open
al_bp_register
al_bp_send
```

Let us consider the last as an example (Figure 2):

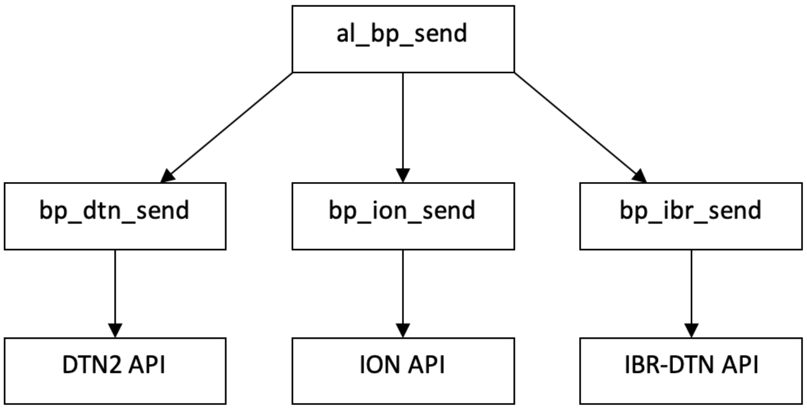


Figure 2. Example of the relationship between an AL function and the BP implementations API. The al_bp_send() calls the bp layer function on the basis of the running BP implementation.

The al_bp_send()s called by the application, and it contains a switch based on the BP implementation. Assuming the choice is performed at run-time, if the ION demon is on, the al_bp_send() will call the bp_ion_send(), which in turn will call the send of the ION_API.

From a file system point of view, the bundle functions were located in the same file as the basic and utility functions and they were simply distinguished by the function prefix, the "al_bp" prefix was used by the basic functions while the “al_bp_bundle” prefix was used by the bundle functions. The functions specific to the BP implementations are instead in separated folders and files.

The AL 1.7.0 includes an extension layer called “ExtB” (B as the initial of his designer Andrea Bisacchi), these functions were in another file called al_bp_extb.c and since it was an extension the related functions left unaltered all the other functions described above. This extension aimed to manage the high complexity of registration and de-registration operations and at the same time to give the user a sort of “UDP socket-like” interface. At the same time, because of compatibility constraints, it had to be kept separated from the rest of the code, which had to be left unaltered.

The weakness of this version is clear by inspection of Figure 3: the layers did not completely overlap, with significant consequences in terms of scalability and portability. Moreover, since the AL components were created or updated by different developers over about a decade since the first release of the AL, there were lots of redefinitions of function/types which were not consistent with the old bpv6 [RFC 5050], not to mention the new bpv7, [RFC9171] finalized during this thesis after not less than 30 drafts, each with its own peculiarities. Finally, the whole structure was not clear, which impaired the use by developers.

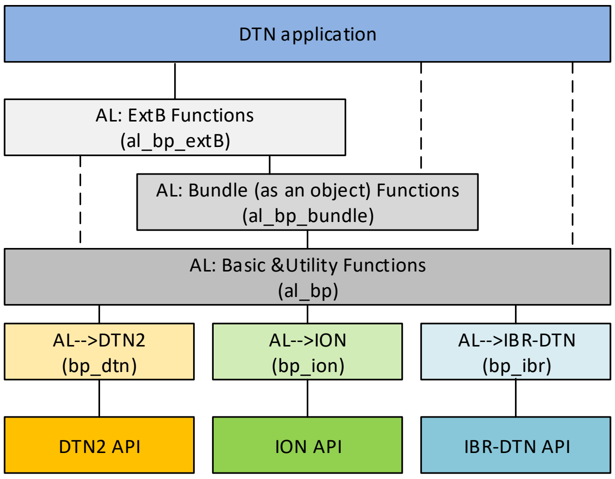

DTN application
AL: ExtB Functions (al_bp_extB)
AL: Bundle (as an object) Functions (al_bp_bundle)
AL: Basic &Utility Functions (al_bp)
AL-->DTN2 (bp_dtn)
AL-->ION (bp_ion)
AL-->IBR-DTN (bp_ibr)
DTN2 API
ION API
IBR-DTN API

Figure 3. The problem of layers in the al_bp v.1.7.0. The layers don’t completely overlap with significant consequences in terms of scalability and portability.

For these and other reasons explained later, a refactoring of the AL was deemed necessary. In the next chapter, we will describe the AL successor, called Unified API.

2 UNIFIED API OVERVIEW

The Unified API maintains the same goal as its predecessor: to prevent code duplication by decoupling the DTN application from the Bundle protocol implementation.

2.1 COMPILATION

Applications based on the Unified API, like DTNperf [DTNPERF] can be compiled for one implementation only or multiple implementations, together with the Unified API, as shown in Figure 4. Currently, the BP implementations supported are DTN2, ION, IBR-DTN and µD3TN. The choice of the implementation (or the supported implementations) is made from the command line at compile time. To allow the coordinated compilation of both DTNperf and the Unified API, we have written a Makefile in the directory DTNsuite, containing both the Unified API and DTNperf as subdirectories. This Makefile calls first the Unified API Makefiles, and then, once the library is compiled, calls the DTNperf Makefile, in an automatic way. For example, if we want to compile the DTNperf application and the Unified API for ION we will use from the DTNsuite directory the command:

```
$ make ION_DIR=<ion_dir_absolute_path>
```

Where <ion_dir> indicates the directory where the ION installation files are located.

If we decide to compile the DTNperf application for multiple implementations, we will write:

```
$ make DTN2_DIR=<dtn2_dir_absolute_path> ION_DIR=<ion_dir_absolute_path>
```

But we must keep in mind that at runtime only one BP implementation can be active at a time. For the user's convenience, postfixes may be added to the application name (e.g. DTNperf_vION denotes an executable compiled for ION only, the same but with vION_vDTN2 for both ION and DTN2, etc.).

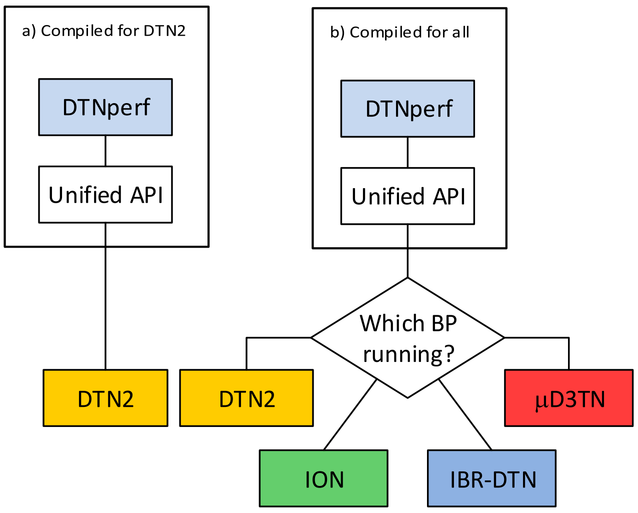


Figure 4 . DTNperf and Unified API compilation flow. DTNperf can be compiled only for one BP implementation only or multiple implementations together with the Unified API.

If we want to run the application compiled for DTN2 but the running implementation is ION, we will receive an error message:

```
$ dtnperf_vDTN2 –server
>Running BP implementation: ION
>Error in al_socket_init: code not compiled for ION
```

If the user adds “DEBUG=1” at compile time, libraries and DTNperf executable will be compiled for the debug and with the code optimizer turned off (to preserve the instruction execution order).

```
$ make ION_DIR=<ion_dir> DEBUG=1
```

To simplify the Makefiles, the support for multiple implementations now relies on the operator ":=", which allows the variables to be “simply expanded” [GNU_MAKEFILE].

The result is shown in this example:

```
ifneq ($(strip $(DTN2_DIR)),)
$(info Making for DTN2)
LIB_NAME :=$(LIB_NAME)_vDTN2
INC := -I$(DTN2_DIR) -I$(DTN2_DIR)/applib/ -I$(DTN2_DIR)/oasys/include
OPT := -DDTN2_IMPLEMENTATION
endif

ifneq ($(strip $(ION_DIR)),)
$(info Making for ION)
LIB_NAME := $(LIB_NAME)_vION
INC := $(INC) -I$(ION_DIR) -I$(ION_DIR)/$(BP)/include -
I$(ION_DIR)/$(BP)/librar>
OPT := $(OPT) -DION_IMPLEMENTATION $(ION_ZCO)
Endif
```

2.2 THE STRUCTURE

The Unified API consists of three layers as shown in Figure 5.

The first top layer is the AL layer, and it contains all the functions that can be directly called by the DTN application, they are divided into four categories:

- “socket”: which contains all the high-level communication functions
- “bundle”: which contains the function that allow to modify the bundle structure
- “types”: which contains the declarations of the Unified API types
- “utilities”: which contains utility functions like “debug print” functions

The first layer will be carefully examined in the next chapter.

The second layer (in dark grey) is called al_bp and it is an intermediate layer, its functions named “al_bp..” have the goal to make the al layer almost independent from the BP implementations. For this reason, the functions are based on a switch that depends on the running BP implementation.

The last layer is the bp one and it contains the interfaces with the supported implementations APIs. Since the interfaces are independent of each other’s, they are represented as different blocks.

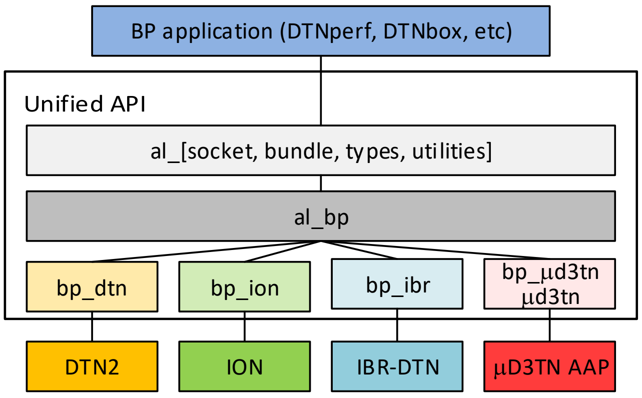

BP application (DTNperf, DTNbox, etc)
Unified API
al_[socket, bundle, types, utilities]
al_bp
bp_dtn
bp_ion
bp_ibr
bp_µd3tn µd3tn
DTN2
ION
IBR-DTN
µD3TN AAP

Figure 5. Unified API layers structure. The first layer (AL) contains all the functions that can be directly called by the application; the intermediate layer (al_bp) aims to make the higher layer independent from the BP implementations; finally, the third layer contains the interface to the BP APIs.

Great attention has been paid in the organization of the code. Layers and categories inside each layer are mirrored by the folder organization, shown in Figure 6. Moreover, each layer, and inside each layer each category, has its own prefix. For example, the socket functions are located in the "al/socket" folder and are denoted by the "al_socket..." prefix, such as al_socket_send(), al_socket_receive, etc. Note that ION files are split into bpv6 and bpv7 folders, as ION can be compiled either for one version or the other, not for both at the same time (ION APIs differ)

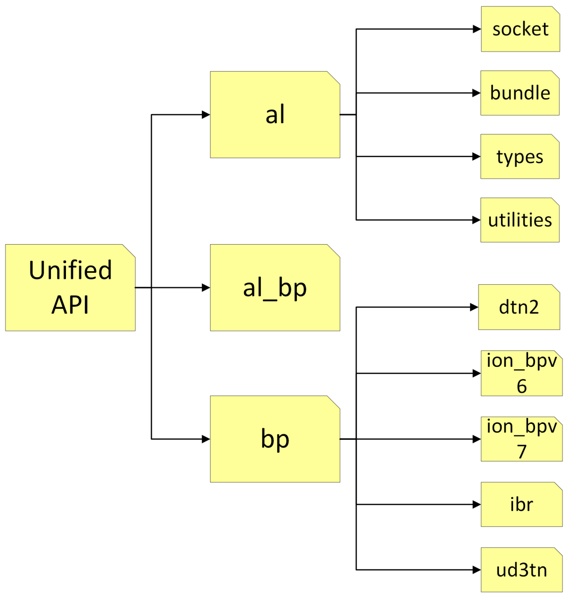


Figure 6. Folder scheme of Unified API. Layers and categories of each layer are mirrored by the folder organization.

2.3 AL API: THE SEND CASE

To understand how the layers work let us consider the "send" function shown in Figure 7. If an application (like the program template shown in the next section) wants to send a bundle, it must call the "al_socket_send()", i.e. a function of the first layer (light grey); this function will do some high-level check and then it will call the al_bp_send (dark grey), which in turns will call the bp_send corresponding to the running BP implementation. In other words, the al_bp_send works as a switch, starting from the knowledge of the BP implementation passed in input, as shown by the excerpt of the code below:

```
      switch (bp_implementation)
      {
      case BP_DTN://DTN2 or DTNME
            return bp_dtn_send(handle, regid, spec, payload);

      case BP_ION:
            return bp_ion_send(handle, regid, spec, payload);

      case BP_IBR:
            return bp_ibr_send(handle, regid, spec, payload);

      case BP_UD3TN:
            return(bp_ud3tn_send(handle, regid, spec, payload));

      default: // bundle protocol implementation not supported
            return BP_ENOBPI;
      }
}
```

Let us assume that the BP implementation running is ION; the al_bp_send (dark grey) will call the bp_ion_send (pale color) which has two main tasks: first, to convert all the Unified API types (declared in al_types.h) to the ION API types; second, to call the bp_send function provided by the ION API (dark color).

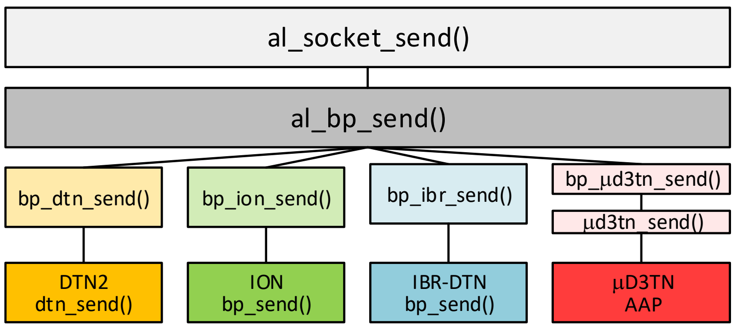


Figure 7. Function layering applied to send function. The al_socket_send() calls the al_bp_send() which calls the bp interface (pale colors) corresponding to the running BP implementation (dark colors).

From this example, it is easy to notice that the modularity and the layering have driven the Unified API design. It is important to notice two important points:

- The first one is related to the scalability of the code, all the future BP implementations that will be added to the Unified API will require only an additional case in the switches of the al_bp layer and the design of proper interface (at bp_layer), but no modify will be made at the first layer
- The second one is related to the development of DTN application, since the first layer remains unaltered and since it is almost independent from the BP implementation, developing a DTN

application became extremely easy. To prove this point in the next chapter we will consider a simple DTN program based on Unified API.

3 Top layer: AL Components

3.1 SOCKET

The socket block is the core of the Unified API as it contains all the functions necessary to the communication. The socket files are shown in Figure 8.

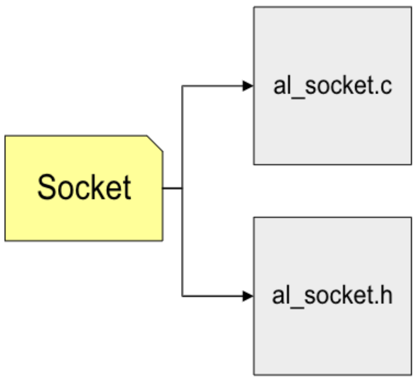


Figure 8. The "al/socket" files structure. The folder contains the al_socket.c file and its header file. This block contains all the functions necessary to implement the UDP-like socket.

The functions of this block, denoted by the "al_socket" prefix, are declared in the "al_socket.h" file. The goals of this layer are:

- to provide the user with an abstract interface to design multiplatform DTN applications.
- to give the user an interface that is similar to the familiar UDP socket, to simplify program writing.

3.1.1 The Registration List

The socket layer uses an internal registration list to keep memory of all the registrations defined as the open connections to the BP (one DTN application can have a single or multiple registrations).

Each registration is represented by a structure of the "al_socket_registration_t" type:

```
typedef struct {
      /**
       * \brief Registration state.
       */
      al_types_reg_info reginfo;
      /**
       * \brief Local enpdoint identifier.
       */
      al_types_endpoint_id local_eid;
      /**
       * \brief Handle object, used in al_bp layer.
       */
      al_types_handle handle;
      /**
       * \brief Mutex for send/receive (not at the same time).
       */
```

```
	pthread_mutex_t mutexDTN2send_receive;
	/**
	 * \brief Registration cookie used in al_bp layer.
	 */
	al_types_reg_id regid;
	/**
	 * \brief Error of layer al_bp.
	 */
	al_bp_error_t error;
	/**
	 * \brief Registration descriptor.
	 */
	al_socket_registration_descriptor reg_des;
} al_socket_registration_t;
```

The structure contains all the elements that characterize a registration. The most important are:

- The registration descriptor which identifies a specific registration; this is an integer inspired by the file descriptor used by UDP and TCP sockets, with the only notable difference that this one is not given by the OS.
- The local EID, which represents the local Endpoint identifier [RFC5050] [RFC9171]

3.1.2 Init and destroy

All DTN applications must start by calling al_socket_init:

```
al_error al_socket_init()
```

Its goal is to initialize the list of BP registrations and discover the BP implementation which is running at the bottom (in this way, all the errors caused by a mismatch between the BP implementations indicated at compilation time and the BP implementation actually running are immediately found). To discover the BP implementation, the al_socket_init calls the al_bp_get_implementation (from the al_bp layer), which looks for the running process specific to each implementation (e.g. BP daemons). This function is the only al_bp function not based on a switch.

The dual of this function, to be called at the end of the DTN program, is the al_socket_destroy:

```
void al_socket_destroy()
```

As the name says, the role of this function is to eliminate the registration list; for this reason, after calling this function the programmer can no longer use any al_socket functions.

3.1.3 Register and Unregister

After calling the al_socket_init() a DTN application must call the al_socket_register()

```
al_error al_socket_register (
al_socket_registration_descriptor *registration_descriptor,
char *dtn_demux_string,
int ipn_service_number,
char force_eid_scheme,
int ipn_node_number_for_DTN2,
char *ip, int port)
```

Thanks to this function the application can register itself to the BP agent either following the “dtn” or the “ipn” scheme (e.g., as dtn://myhost.dtn/ping or ipn:5.2345, respectively). The application needs to provide the desired dtn2 demux (ping) and the ION “service number” (2345) in input, then scheme selection and local EID building will be automatically performed on the basis of the BP implementation running (there is one preferred choice for each implementation). This is an important point to focus on because it is no longer necessary to select the specific EID scheme to be used in registrations (and so removing dependence on the implementation) because this operation is done automatically using the defaults. These defaults can however be overridden by the application by “forcing a specific scheme (force_eid_scheme). Last, this mechanism serendipitously prevents the application user from disguising its real identity with a fake or anonymous source EID.

The way the DTN application and the BP implementation communicate depends on the BP implementation. This is usually transparent to the program, which has simply to call C functions belonging to the implementation API. However, a few implementations allow the program also to communicate with the BP agent by means of a TCP socket, in which case it is also possible to start the BP agent so that it listens to a specific IP and port, as every TCP server. In this case, the application, acting as a TCP client, must register with this IP and port, to establish the TCP connection between the application and the BP agent. Note that if the IP is not localhost, a BP agent can accept connection from DTN applications running on different machine, which may be useful in peculiar circumstances, such as on sensor networks. The al_socket_register allows the user to open the connection the usual way or by using an IP address (IP:port), when the latter is supported by the implementation.

After having inserted the new registration in the list, the al_socket_register returns among the output parameters the registration descriptor that identifies the registration; it will be used by other al_socket functions to send or receive bundles, exactly as the analogous socket descriptor is used by UDP sending and receiving functions.

The dual function of the al_socket_register is the al_socket_unregister():

```
al_error al_socket_unregister(
al_socket_registration_descriptor registration_descriptor)
```

This function removes from the list the registration identified by the registration descriptor passed in input and is the analogous to the close in UDP. If there are multiple registrations all of them must be deregistered before calling the last function which is the al_socket_destroy().

3.1.4 Send and Receive

After the registration we have usually a cycle of sends and receives. These two functions are the core of the communication. Let us start from the al_socket_send:

```
al_error al_socket_send(
al_socket_registration_descriptor registration_descriptor,
al_types_bundle_object bundle,
al_types_endpoint_id destination,
al_types_endpoint_id report_to)
```

The call of the send is simple, the parameters are:

- The registration descriptor which identifies the registration
- The bundle we want to send
- The destination EID
- The report_to EID address which represents the address where to send the status report (eg mission control center in a space application).

The al_socket_receive prototype is:

```
al_error al_socket_receive(
            al_socket_registration_descriptor registration_descriptor,
            al_types_bundle_object *bundle,
            al_types_bundle_payload_location payload_location,
al_types_timeval timeval)
```

This function receives in input the registration descriptor and gives in output the received bundle (a structure containing not only the payload, but all relevant information, such as the source address, etc.); it also presents two other parameters:

- Payload location, which can either be memory or file; the former option has a limit of 50KB payloads
- Timeval which represents a timeout after which the receive is unblocked (if negative it will block forever).

3.1.5 The flow of the al_socket functions

The flow of the al_socket functions, in a typical program, is shown in Figure 9.

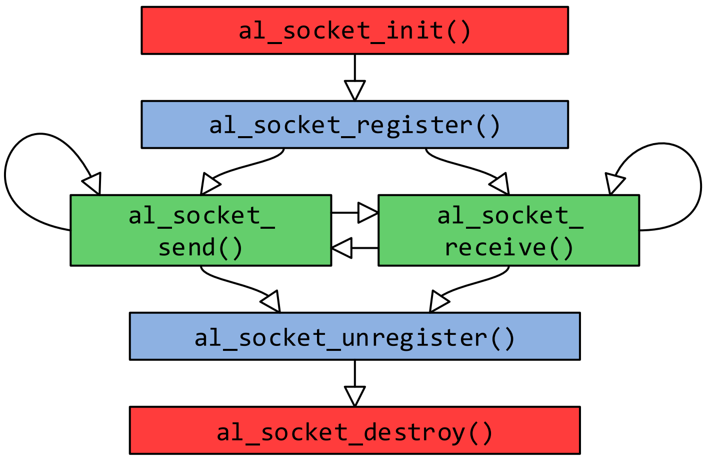


Figure 9 Flow diagram of the most significant al_socket functions. The application must call the al_socket_init(), which initializes the internal registration list and it checks the compatibility, before the al_socket_register(). Then, the application can start calling the al_socket_send() and al_socket_receive() functions. In the end, the application must call the al_socket_unregister() and the al_socket_destroy().

This general flow can also be observed in the program template described in the section The "Test" program

3.1.6 Unified API and UDP socket

As described above, one of the goals of the Unified API is to provide an interface that is similar as possible to UDP sockets. Let us look at the comparison table below:

al_socket functions	UDP socket functions
al error **al_socket_init**()	
al_error **al_socket_register** (al_socket_registration_descriptor *registration_descriptor, char *dtn_demux_string, int ipn_service_number, char force_eid_scheme, int ipn_node_number_for_DTN2, char *ip, int port)	int **socket**(int domain, int type, int protocol) int **bind**(int sockfd, const struct sockaddr *addr, socklen_t addrlen)

al_socket functions	UDP socket functions
al_error **al_socket_send**(al_socket_registration_descriptor registration_descriptor, al_types_bundle_object bundle, al_types_endpoint_id destination, al_types_endpoint_id reply_to)	ssize_t **sendto**(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen)
al_error **al_socket_receive**(al_socket_registration_descriptor registration_descriptor, al_types_bundle_object *bundle, al_types_bundle_payload_location payload_location, al_types_timeval timeval)	ssize_t **recvfrom**(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen)
al_error **al_socket_unregister**(al_socket_registration_descriptor registration_descriptor)	int **close**(int fd)
void **al_socket_destroy**()	

We can notice that the flow is the same, except for the al_socket_init and the al_socket_destroy which in the UDP case are managed directly by the OS. There are some unavoidable differences caused by the different characteristics of the bundle and the UDP protocols. For example, UDP lacks ”report to” addresses, the socket descriptor list is managed by the OS, while the registration description list must be created and updated by our software, which does that in a way that is transparent to the user, but the necessity of inserting the al_socket_init and al_socket_destroy functions at the beginning and at the ending of the program. Anyway, we tried hard to push commonality as much as possible.

3.2 BUNDLE

The Bundle block contains two inner blocks, the first contains all the functions that allow callers to create and modify a single bundle, the second one contains all the functions related to the parsing of the options given by the user. The functions of this block are denoted by the “al_bundle” prefix. The folder structure is shown in the Figure 10 :

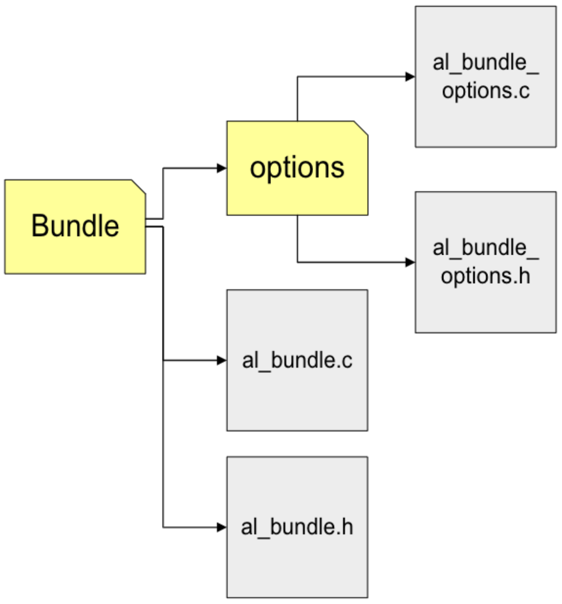


Figure 10.The “al/bundle” folder structure. The folder contains the al_bundle.c file and its header which contains the functions related to the creation and manipulation of the bundle. There is also an internal folder called options which contains the parser component, capable of parsing an input given by the user.

3.2.1 al_bundle.c and al_bundle.h

The bundle is what we want to send or receive, thus is the most important parameter of the sending and receiving functions. Each bundle is represented in the Unified API by an “al_types_bundle” structure (defined in al_types.h), which will be examined in detail in the section TYPES. Let us anticipate here that its fields include the pointer to the bundle payload, bundle options, addresses, and other ancillary data (e.g., metadata information).

The bundle functions aim to manage the bundle structure (as a sort of “object”) and have been designed to help the user to create, destroy and change the parameters of the bundle. The most important ones are examined below.

The al_bundle_create function:

```
al_error al_bundle_create(al_types_bundle_object *bundle_object)
```

Which creates an empty bundle object by dynamically allocating memory for all its fields.

The al_bundle_free function:

```
al_error al_bundle_free(al_types_bundle_object *bundle_object)
```

Which deallocates the memory allocated with al_bundle_create.

The other functions have the goal to set or get a single field of the bundle structure. An example of this kind of functions are:

```
al_error al_bundle_get_dest(al_types_bundle_object bundle_object,
            al_types_endpoint_id *dest)
```

and the dual function:

```
al_error al_bundle_set_dest(al_types_bundle_object *bundle_object,
            al_types_endpoint_id dest)
```

The bundle folder contains also all the elements and the function related to the parsing of the bundle options, which will be treated below.

3.2.2 Options

As mentioned in the introduction, a series of problems led to a makeover of the old al_bp. One of these was related to the parsing of bundle options, a real mess. In brief, not only does each BP implementation its own subset of supported bundle options, but there is also a different subset for bpv6 or bpv7. One main difference is related to the support of ECOS [ECOS], an interesting set of additional options promoted by NASA, which however is only supported by ION and DTNME. Other examples can be given by the "do not fragment" option which can be set in DTNME but not in ION, and in the same way, the custodial transfer, and the cardinal priority, which are supported by bpv6 only. Finally, µD3Tn, being quite recent, does not allow options to be set differently on individual bundles but only at BP daemon start, for all bundles.

All the mentioned limitations and peculiarities of each BP implementations are clearly impossible to be tackled by a third-party programmer; in fact, they are most likely known only to BP implementation developers (maybe not always…) and very few others. To be clear: the maintainers of BP implementation A (in the best case) are aware of its peculiarities, but not at all of those of implementation B, C and D, and vice versa.

The entire problem is resumed in the following table:

Table I Bundle options and DTNsuite options compatibility

Option	Meaning	DTNME	ION bpv6	ION bpv7	IBR-DTN$_{V6}$	µD3TN
V	BP version	✖				⊗
N	do not fragm.	✖			✖	
C	custody *	✖	✖		✖	
r	BSR reception	✖	✖	✖	✖	⊗
c	BSR custody *	✖	✖		✖	⊗
f	BSR forward.	✖	✖	✖	✖	⊗
deletion	BSR deletion	✖	✖	✖	✖	⊗
ack_req_by_ app	ack req. by app				✖	
l	lifetime	✖	✖	✖	✖	⊗
p	priority*	✖	✖		✖	
ordinal	Ordinal prior.	✖$_e$	✖	✖$_e$	✖	
unreliable	Unreliable	✖$_e$	✖	✖$_e$	✖	
critical	Critical	✖$_e$	✖	✖$_e$	✖	
flow	Flow	✖$_e$	✖	✖$_e$	✖	
mb-type	mb-type	✖	✖	✖	✖	
mb-string	mb-string	✖	✖	✖	✖	
irf_trace	irf_trace			✖		
force-eid^	force-eid	✖	✖	✖	✖	
ipn-local	ipn-local	✖				
open-with-ip	open-with-ip	✖	✖	✖	✖	✖
debug	debug	⊖	⊖	⊖	⊖	⊖

LEGEND:

✖ Supported by the BP implementation

✖$_e$ Supported, but in case of BPv7 it is located inside ECOS [ECOS]

⊖ Unified API option, it represents the level of threshold related to the debug print component (for more details consider the section The debug_print folder).

⊗ Supported, but can be set only when the BP daemon starts, i.e. not on individual bundles

* Supported only by BPv6 version

$_{V6}$ means that the BP implementation supports only BPv6

^ Experimental (it triggers the generation of interregional routing information to be sent to the report to node on experimental version of ION, courtesy of Scott Burleigh).

Note that the last four options are all fields of the dtn_suite_options structure.

To allow a prospective programmer to master the chaos, we decided to include in the Unified API a parsing function called "al_bundle_options_parse()" thus moving the burden of bundle options parsing from the application to the Unified API library..

```
al_error al_bundle_options_parse(int argc, char **argv,
            bundle_options_t *bundle_options,
            dtn_suite_options_t *dtn_suite_options,
            application_options_t* residual_options)
```

The arguments are:

- argc and argv, the input parameters given by the user on the command line when the application is launched
- bundle options, the structure defined in the file al_bundle_options.h, which contains all the bundle options that can be set by the user. The list is long, as we have to consider the union of all subsets supported by individual implementations, either for bpv6 or bpv7... Examples of these bundle options are lifetime and cardinal priority.
- dtn_suite_options, a structure defined in a_bundle_options.h that contains a set of application options used by the dtnsuite and potentially useful to other programmers. Examples of these options are the force-eid option, used to override the automatic selection of the registration scheme, or the debug level (which will be explained in the section UTILITIES)
- residual_options, a structure that contains all the options that could not be parsed (i.e., that are neither bundle or DTNsuite options) and that must be parsed by the application, being application-specific, such as "client" or "server" or "monitor" in DTNperf.

The parsing function is necessary quite complex. The first task is the parsing of the input parameters, which is made by using the "getopt" function [GETOPT] inserted in a "while". At each cycle, the getopt parse a new option. Immediately after, our function checks the compatibility of the option parsed with the BP implementation running. Let us consider the example quoted before: the ECOS are supported only by ION and DTNME, for this reason if the user insert as input parameter an ECOS field like "CRITICAL" the function will check if the implementation running is ION or DTNME:

```
case CRITICAL:
                  if (bp_implementation != BP_ION
                            && bp_implementation != BP_DTN) {
                        printf("Parser error, wrong implementation found
\n");
                        return AL_ERROR;
                  }
                  bundle_options->ecos.critical = TRUE;
                  bundle_options->ecos.ecos_enabled = TRUE;
                  break;
```

Only if the BP implementation check is passed, the option will be added in the bundle options structure, otherwise the parsing is stopped, and an error is returned

The entire flow is resumed in the following block diagram:

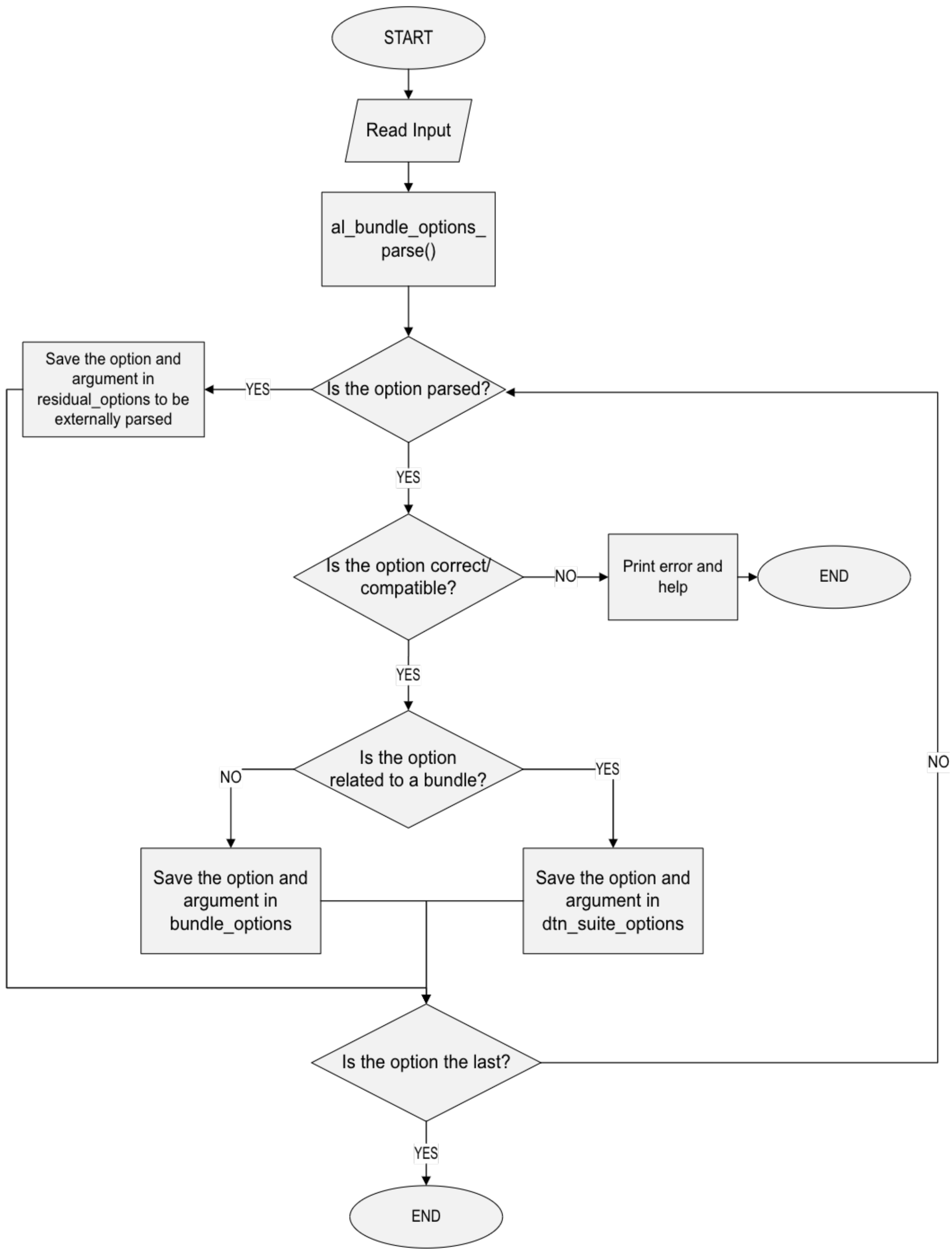

START
Read Input
al_bundle_options_ parse()
Is the option parsed?
YES
Save the option and argument in residual_options to be externally parsed
YES
Is the option correct/ compatible?
NO
Print error and help
END
YES
Is the option related to a bundle?
NO
YES
Save the option and argument in bundle_options
Save the option and argument in dtn_suite_options
NO
Is the option the last?
YES
END

Figure 11. al_bundle_options_parse block diagram. The parse function reads the input, it checks if the option can be recognized if not, it saves the option in the residual_options structure. If the option can be recognized, it checks if the option is supported by the BP protocol running and if not, it prints error and help. If the option is compatible with the BP running protocol it saves the option in the correspondent structure: bundle options if the option is related to the bundle, dtn_suite_options if not. Finally, it checks if the option is the last, if not it restarts the cycle.

The bundle options stored in the structure bundle_options_t can be applied to the bundle structure using the function which overrides the default value:

```
al_error al_bundle_options_set(al_types_bundle_object *bundle,
                bundle_options_t bundle_options)
```

In case of error (e.g., the insertion of a wrong value or an option not supported by the running BP implementation) the function prints a meaningful message and then a help message listing all options with the possible limitations.

The realization of the "option" block has required a considerable effort, but we think it will pay off both in terms of consistency between applications and code maintenance, as any change in bundle options support, once applied to the parse will automatically be available to all applications based on the Unified API in a serendipitous way.

3.3 TYPES

An Abstract API not only requires abstracted functions but also abstracted structures, and these are defined in the Types block. The challenge is that each BP implementation has its own peculiar types, with significant differences. Our aim was to create a set of abstracted types, containing all the types used in the AL layer (the sole layer "seen" by the user), and only these, to clearly define our Unified API. Obviously, to reach this goal we need conversion functions from each al_type to all corresponding BP specific types and vice versa… These functions, being implementation specific, belong to the lowest layer, and we will examine them in the next chapter. In the design of al_types, we tried hard to be as near as possible to the RFCs [RFC9171], [RFC5050], in particular by using only "official" names in the structure fields, to facilitate the user and also to respond to a few BP implementers' concerns recently emerged during CCSDS meetings, who argued for a unambiguous use of names. An example of what we wanted to avoid: the official (in RFC) "return to" EID, becomes "reply to" in DTN2/DTNME, priorities "bulk", "normal", "expedited" are named differently in ION, etc. Therefore, to help the user, we allowed no space for fantasy in our al_types names.

The file structure of Types is shown in the figure below:

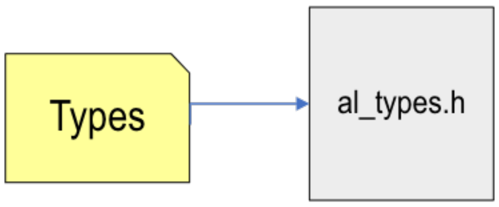


Figure 12.The "al/ types" files structure. The folder contains only a header file which declares the types used in the AL layer in accordance with the RFCs.

Basically, the Types directory contains only the al_types.h header file, including all the types used in the AL layer (“al_types” prefix). The most important is the al_type_bundle_object, which defines the bundle element:

```
typedef struct al_types_bundle_object {
      al_types_bundle_spec * spec;
      al_types_bundle_payload * payload;
} al_types_bundle_object;
```

The bundle object consists of two structures: spec and payload. Let us start from the former:

```
typedef struct al_types_bundle_spec {
      al_types_reg_id delivery_regid;
      uint8_t bp_version;
      al_types_bundle_processing_control_flags bundle_proc_ctrl_flags;
      al_types_bundle_priority_enum cardinal;
      al_types_endpoint_id destination;
      al_types_endpoint_id source;
      al_types_endpoint_id report_to;
      al_types_creation_timestamp creation_ts;
      al_types_timeval lifetime;
      al_types_bundle_ecos_t ecos;
      struct {
                  uint32_t extension_number;
                  al_types_extension_block *extension_blocks;
            } extensions;
unsigned char irf_trace;
} al_types_bundle_spec;
```

It represents the bundle features as indicated in BPV7 RFC [RFC9170], with a few additions (e.g. ECOS [ECOS], version number, etc.). Note that the first parameter, the delivery regid, belongs to the Unified API socket.

Moving to the bundle payload, we have:

```
/**
 * \brief Type definition for a bundle payload
 */
typedef struct al_types_bundle_payload { //al_bundle_payload_t
      al_types_bundle_payload_location location;
      struct {
            uint32_t filename_len;
            char *filename_val;
      } filename;
      struct {
            uint32_t buf_crc;
            uint32_t buf_len;
            char *buf_val;
      } buf;
      al_types_bundle_status_report *status_report;
} al_types_bundle_payload;
```

Here we have two options to save the payload (expressed by the al_types_bundle_location), a file or a buffer. In the former case, expressed by the structure “filename” the payload structure contains the filename while in the second case, “buf”, the structure contains the buffer field.

The last field is the status_report pointer; when we receive a bundle status report, it will point to the al_types_bundle_status_report structure containing the parsed fields of the received status report.

The bundle status report structure is reported below (see [RFC9170]:

```
typedef struct al_types_bundle_status_report {
      al_types_bundle_id bundle_id;
      al_types_status_report_reason reason;
      al_types_status_report_flags flags;
      uint32_t reception_ts;
      uint32_t custody_ts;
      uint32_t forwarding_ts;
      uint32_t delivery_ts;
      uint32_t deletion_ts;
      uint32_t ack_by_app_ts;
} al_types_bundle_status_report;
```

3.4 UTILITIES

This last block of the top layer contains a few auxiliary functions useful to develop DTN applications, such as those of the DTNsuite. They are denoted by the “al_utilities” prefix.

The file structure of this block is the following:

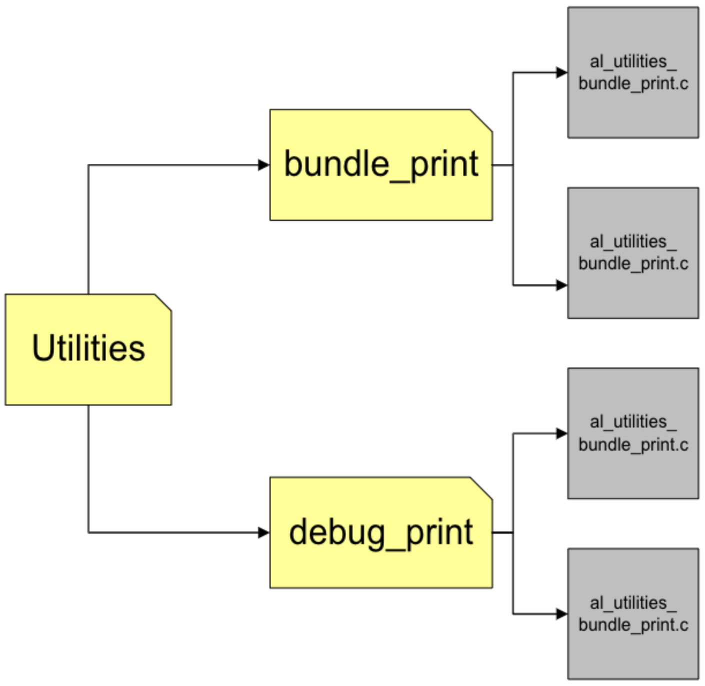

al_utilities_ bundle_print.c
bundle_print
al_utilities_ bundle_print.c
Utilities
al_utilities_ bundle_print.c
debug_print
al_utilities_ bundle_print.c

Figure 13. The “al/utilities” files structure. The folder contains two internal folders: the bundle print folder which contains the utility print out functions and the debug print folder which gives the user the opportunity to choose the print level.

3.4.1 The bundle_print folder

The first folder contains the al_utilities_bundle_print functions, designed to print complex bundle structures. They can either print on a file or on the standard output.

The al_utilities_bundle_print_bundle_object is shown as an example:

```
void al_utilities_bundle_print_bundle_object(al_types_bundle_object
bundle_object, const char* name, size_t indent, FILE* stream)
{
      fprintf(stream, "%s%s %s:\n", al_utilities_bundle_print_tabs(indent),
"al_types_bundle_object", name);
      al_utilities_bundle_print_bundle_spec(*bundle_object.spec, "*spec",
indent+1, stream);
      al_utilities_bundle_print_bundle_payload(*bundle_object.payload,
"*payload", indent+1, stream);
}
```

3.4.2 The debug_print folder

The second folder contains the debug_print functions. In the development and use of complex programs, it may be useful to distinguish the level of the prints at runtime without recompiling the code. For this reason, we created the al_utilties_debug_print which gives the opportunity to the user to choose the level of the print that want to see.

This conditional print is based on a threshold, called debug level, the print should be performed only if the debug level at run time is equal to or higher than the threshold. For this reason, only the debug prints of level 0 are always printed.

The core of this mechanism is the function al_utilities_debug_print:

```
void al_utilities_debug_print(debug_level threshold_level, const char
*string, ...) {
      if (is_initialized && debug_struct.debug >= level) {
            va_list lp;
            va_start(lp, string);
            al_utilities_debug_print_logger_va_list(stdout, string, lp);
            // print log
            if (debug_struct.create_log)

      al_utilities_debug_print_logger_va_list(debug_struct.log_fp, string,
lp);
            va_end(lp);
      }
}
```

This function prints the message given in input but only if the debug level chosen at program start (the global variable debug_struct.debug) is greater than, or equal to, the threshold level passed as first parameter. The string passed as a second parameter coincides with the string usually passed to the C printf. An example may help. Let us suppose we want to print a control message with a debug level 2, we will write:

```
al_utilities_debug_print(DEBUG_L1,"Creating the bundle..\n");
```

The print will be actually performed only if the debug level chosen at start time is 1 or 2, as the threshold is 1.

To use the debug_print, we have first to start the debug logger, through the use of the function:

```
Int al_utilities_debug_print_debugger_init(int debug, boolean_t create_log,
char *log_filename)
```

And at the end of debug print we have to call the dual function:

```
int al_utilities_debug_print_debugger_destroy()
```

The debug print module is obviously independent of DTN protocols and all other Unified API functions; historically, it was developed for DTNperf, but then, we decided to move it in the Unified API to make it available to all DTN applications.

3.5 ERRORS MANAGEMENT (THE DEFINITIONS.H FILE)

In the AL layer there is also a simple header file which contains a few elements common to all blocks. The most important is the enumerative al_error, which represents the possible outcomes of the AL functions:

```
typedef enum al_error
{
	/**
		* \brief Successful function outcome
	*/
	AL_SUCCESS=0,
	/**
		* \brief Fail function outcome
	*/
	AL_ERROR=1,
	/**
		* \brief General warning function outcome
	*/
	AL_WARNING=2,
	/**
		* \brief Warning the receiver is not indicated or is dtn:none
(al_socket_send() specific error).
	*/
	AL_WARNING_RECEIVER=3,
	/**
		* \brief Warning, timed out, (al_socket_receive() specific
error).
	*/
	AL_WARNING_TIMEOUT=4,
	/**
		* \brief Warning, reception interrupted, (al_socket_receive()
specific error).
	*/
	AL_WARNING_RECEPINTER=5
} al_error;
```

The rationale of this enum is to have a unique outcome structure common to all AL layer fuctions. Actually, we started considering only a binary outcome: failure or success. The cause of the failure would be communicated to the user through a simple print. During the redesign of the old code, however, we noticed that a few corner cases required a more complex treatment. For example, the timeout caused by the al_socket_receive is more a warning than an error, and for this reason it had to be threaten in a different way. For this reason, we extended our initial idea to encompass possible warnings, either specific (like in the timeout case) or generic. More details about this can be found in the DTN applications code [DTNPERF][DTNSUITE].

4 THE BOTTOM LAYER: INTERFACES TO IMPLEMENTATION APIS

The bottom layer of the Unified API, the “bp” layer, has the task to directly interface our code with the implementation specific APIs. As mentioned in the introduction, the BP implementations supported by the Unified API are ION [ION], DTN2 [DTN2], IBR-DTN [IBR_DTN] and µD3TN [µD3TN].

Each implementation has its own folder as shown in the Figure 6. The file structure is generally the same as that of the DTN2 folder, shown in Figure 14. Since the goals of this layer are the conversion of the types and the call of the API functions, generally each folder has a pair of files file used for the type conversion (bp_dtn_conversion.c/h) and another used for the call of the API functions (bp_dtn.c/h).:

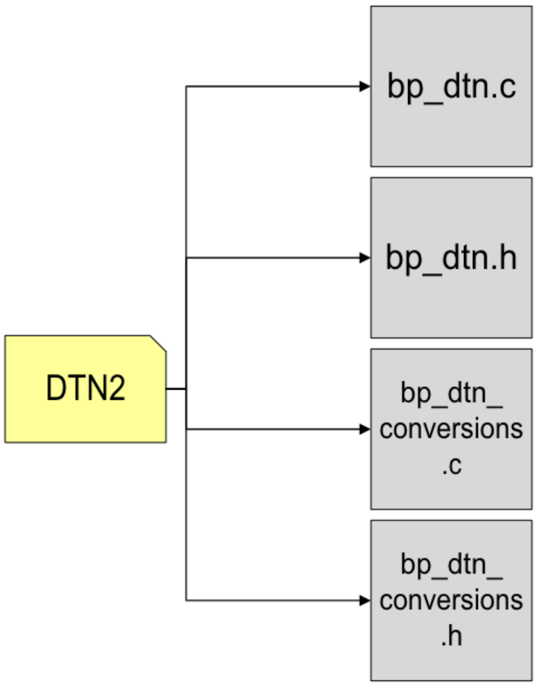


Figure 14 The "bp/dtn2" files structure. The folder contains the files (bp_dtn_conversions.h/c) related to the conversion functions from the AL layer to the DTN2 implementation (and viceversa) and the files (bp_dtn.h/c) related to the communication functions.

The functions are denoted by the “bp” prefix; in particular, the conversion functions are denoted by “bp_to” if they convert the Unified API type to the BP implementation type, “bp_from” vice versa.

4.1 A PRACTICAL EXAMPLE: THE DTN2 SEND

Let us consider the bp_dtn_send, whose code is reported below:

```
al_bp_error_t bp_dtn_send(al_types_handle handle,
                          al_types_reg_id regid,
                          al_types_bundle_spec* spec,
                          al_types_bundle_payload* payload)
{
	al_types_bundle_id id;
	id.source=spec->source;
	id.creation_ts.time=0;
	id.creation_ts.seqno=0;
	id.frag_offset=0;
	id.frag_length=0;
	dtn_handle_t dtn_handle = bp_to_dtn_handle(handle);
	dtn_reg_id_t dtn_regid = bp_to_dtn_reg_id(regid);
	dtn_bundle_spec_t dtn_spec = bp_to_dtn_bundle_spec(*spec);
	dtn_bundle_payload_t dtn_payload =
bp_to_dtn_bundle_payload(*payload);
	dtn_bundle_id_t dtn_id = bp_to_dtn_bundle_id(id);

	int result = dtn_send(dtn_handle, dtn_regid, & dtn_spec, &
dtn_payload, & dtn_id);

	handle = bp_from_dtn_handle(dtn_handle);
	regid = bp_from_dtn_reg_id(dtn_regid);
	*spec = bp_from_dtn_bundle_spec(dtn_spec);
	*payload = bp_from_dtn_bundle_payload(dtn_payload);
	spec->creation_ts = bp_from_dtn_timestamp(dtn_id.creation_ts);
	return bp_dtn_error(result);
}
```

4.1.1 The conversion function from BP to DTN2

The bp_dtn_send must call the dtn_send, declared in DTN2 (note that DTN2 API function start with the “dtn” prefix, actually quite ambiguous in a DTN world; it would have been preferable a more specific identifier, such as “dtn2”) as:

```
int dtn_send(dtn_handle, dtn_regid, & dtn_spec, & dtn_payload, & dtn_id);
```

Before calling the dtn_send, we need to call a few conversion functions from the abstracted types to the DTN2 types:

```
dtn_handle_t dtn_handle = bp_to_dtn_handle(handle);
dtn_reg_id_t dtn_regid = bp_to_dtn_reg_id(regid);
dtn_bundle_spec_t dtn_spec = bp_to_dtn_bundle_spec(*spec);
dtn_bundle_payload_t dtn_payload = bp_to_dtn_bundle_payload(*payload);
dtn_bundle_id_t dtn_id = bp_to_dtn_bundle_id(id);
```

If we consider for example the bp_to_dtn_bundle_spec conversion function, we have:

```
dtn_bundle_spec_t bp_to_dtn_bundle_spec(al_types_bundle_spec bundle_spec)
{
dtn_bundle_spec_t dtn_bundle_spec;
int i;
al_types_extension_block dtn_bundle_block;
memset(&dtn_bundle_spec, 0, sizeof(dtn_bundle_spec));
memset(&dtn_bundle_block, 0, sizeof(dtn_bundle_block));
dtn_bundle_spec.source = bp_to_dtn_endpoint_id(bundle_spec.source);
dtn_bundle_spec.dest = bp_to_dtn_endpoint_id(bundle_spec.destination);
dtn_bundle_spec.replyto = bp_to_dtn_endpoint_id(bundle_spec.report_to);
dtn_bundle_spec.priority = bundle_spec.cardinal;
dtn_bundle_spec.dopts =
bp_to_dtn_bundle_delivery_opts(bundle_spec.bundle_proc_ctrl_flags);
dtn_bundle_spec.expiration = bp_to_dtn_timeval(bundle_spec.lifetime);
dtn_bundle_spec.creation_ts = bp_to_dtn_timestamp(bundle_spec.creation_ts);
dtn_bundle_spec.delivery_regid =
bp_to_dtn_reg_id(bundle_spec.delivery_regid);
[…]
return dtn_bundle_spec;
}
```

The bp_to_dtn_bundle_spec calls the single field conversion function and then builds the bundle spec of DTN2 type. To better understand, let us take as an example a simple field conversion function like the bp_to_dtn_endpoint_id:

```
dtn_endpoint_id_t bp_to_dtn_endpoint_id(al_types_endpoint_id endpoint_id)
{
      dtn_endpoint_id_t dtn_eid;
      strncpy(dtn_eid.uri, endpoint_id.uri, DTN_MAX_ENDPOINT_ID);
      return dtn_eid;
}
```

The function is minimal because the only operation done is the string copy of our endpoint URI in the DTN2 EID structure.

4.1.2 The conversion functions from DTN2 to BP

Once we have called the dtn_send functions we need to call the conversion functions from DTN2 to our bp layer to obtain the data returned by the BP protocol implementation:

```
*spec = bp_from_dtn_bundle_spec(dtn_spec);
*payload = bp_from_dtn_bundle_payload(dtn_payload);
spec->creation_ts = bp_from_dtn_timestamp(dtn_id.creation_ts);
```

The functions are the dual of that working in the opposite direction. For example, we report below the “bp_from_dtn_timestamp”:

```
al_types_creation_timestamp bp_from_dtn_timestamp(dtn_timestamp_t
timestamp)
{
      al_types_creation_timestamp bp_timestamp;
      bp_timestamp.time = timestamp.secs;
      bp_timestamp.seqno = timestamp.seqno;
      return bp_timestamp;
}
```

4.2 The IBR-DTN case

All the bp folders have the same structure as that shown in Figure 14, but IBR-DTN (Figure 15), which contains only two files “bp_ibr.cpp” written in C++ and its own header file “bp_ibr.h”. In this case type conversion functions were not written because IBR-DTN does not declare types similar to that of other implementation, and thus to the abstracted ones. Conversions are made directly before calling the API function when necessary. As an example, consider the portion of the “bp_ibr_send” shown below:

bundle.source = EID(std::string(spec->source.uri));

bundle.destination = EID(std::string(spec->destination.uri));

bundle.reportto = EID(std::string(spec->report_to.uri));

For more details, the interested reader is referred to [PALLOTTI_2016].

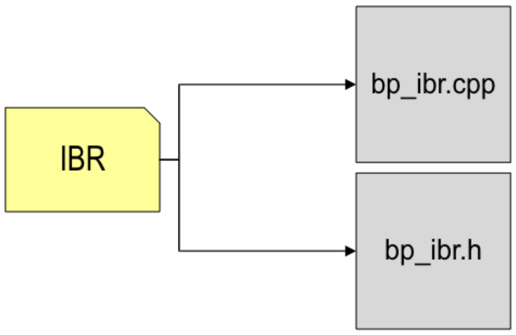


Figure 15, The "bp/ibr" folder structure. The IBR case is different from the other case because the folder doesn't contain the conversion functions file which are instead inside the communication ones.

4.3 THE µD3TN CASE

Support for the µD3TN BP implementation was has been inserted in the Unified API, thanks to one of the correlators, Dr. Keith Scott, The MITRE Corporation, who wrote the corresponding bp layer interface and passed it to Unibo, which finalized and extended it (e.g. by adding support for the "ipn" scheme registration).

The bp/ud3tn folder follows the same general structure shown in Figure 14, with the notable exception of the addition of a file named "bp_ud3tn_types.h" which contains the types used to work with µd3tn. The µD3TN API is different from the others in that the µD3TN is in Python, which required an additional shim to construct C API desgiend after the DTN2 API. That shim could then be integrated into the Unified API. For clarity, we will reconsider the case of the send, shown in Figure 7. We can now notice an asymmetry caused by the presence of a shim layer to implement the "ud3tn_send" on top of µD3TN AAP (Application Agent Protocol, the only interface presently offered by this implementation to C programs).

The code of the bp_ud3tn_send, similar to that for the DTN2 interface, is shown below:

```
al_bp_error_t bp_ud3tn_send(al_types_handle handle,
                             al_types_reg_id regid,
                             al_types_bundle_spec* spec,
                             al_types_bundle_payload* payload)
{
      ud3tn_handle_t ud3tn_handle = bp_to_ud3tn_handle(handle);
      ud3tn_reg_id_t ud3tn_regid = bp_to_ud3tn_reg_id(regid);
      ud3tn_bundle_spec_t ud3tn_spec = bp_to_ud3tn_bundle_spec(*spec);
      ud3tn_bundle_payload_t ud3tn_payload =
bp_to_ud3tn_bundle_payload(*payload);
      int result = ud3tn_send(ud3tn_handle, ud3tn_regid, & ud3tn_spec, &
ud3tn_payload);
      handle = bp_from_ud3tn_handle(ud3tn_handle);
      // regid = bp_from_ud3tn_reg_id(ud3tn_regid);
      *spec = bp_from_ud3tn_bundle_spec(ud3tn_spec);
```

```
	*payload = bp_from_ud3tn_bundle_payload(ud3tn_payload);
	//*id = bp_from_ud3tn_bundle_id(ud3tn_id);
	return bp_ud3tn_error(result);
}
```

The function has the same structure as the functions seen previously, the only difference is that in this case the "ud3tn_send" function does not belong to the presently non-existant µD3TN C API, but to the shim-layer. Note that this function calls the "aap_serialize" which is internal to the µD3TN AAP.

```
int ud3tn_send(ud3tn_handle_t handle,
                              ud3tn_reg_id_t regID,
                              ud3tn_bundle_spec_t* spec,
                              ud3tn_bundle_payload_t* payload)
{
	struct aap_message msg;
	msg.type = AAP_MESSAGE_SENDBUNDLE;
	msg.eid_length = strlen(spec->dest.uri);
	msg.eid =  (char *) malloc(strlen(spec->dest.uri)+1);
	strcpy(msg.eid, spec->dest.uri);
#ifdef PRINT_ON
	printf("in ud3tn_send mallocing %d for payload\n", payload-
>buf.buf_len);
#endif
	msg.payload = (uint8_t *) malloc(payload-
>buf.buf_len*sizeof(uint8_t));
	msg.payload_length = payload->buf.buf_len;
	memcpy(msg.payload, payload->buf.buf_val, payload->buf.buf_len);
	// verify message correctness
	aap_serialize(&msg, smoopy, (void *) ((intptr_t)handle->socket), 1);
//sl TOCHANGE
	aap_message_clear(&msg);
	msg = ud3tn_receive_aap(handle, -1);
#ifdef PRINT_ON
	printf("after send received msg of type %d (expected 5--
SENDCONFIRM)\n", msg.type);
#endif
	return UD3TN_SUCCESS;
}
```

If this skim layer is moved into µD3TN, as it seems probable, the bp block structure will become symmetric.

5 TESTING AND DOCUMENTATION

5.1 THE TESTING

This thesis' aim was a complete refactoring and a significant extension of the old Abstraction Layer; as this code, although obsolete, was stable, we decided to carry out the development of the Unified API gradually, by small incremental steps. In order not to introduce regressions, at each step we tested the correct behavior of the Unified API by debugging in the Eclipse IDE a simple program test (showed in the next section).

More complex texts were carried out by using an experimental version of DTNperf in particular when dealing with multiplatform issues.

Sometimes it was also necessary to compare the behavior of our code with the basic tools provided by the different implementations to distinguish the origin of unexpected results.

Only in particular cases like the metadata management one, we relied on Wireshark dissector [WIRESHARK] whose latest versions can dissect both bpv6 and bpv7 bundles.

Finally, we also used the Valgrind analyses tool [VALGRIND] to catch memory leaks and other errors.

5.1.1 The "Test" program

The Unified API test program has a twofold aim: firstly, it should allow the developer to check any code modifications; secondly, it should represent a simple template for prospective programmers wanting to use the Unified API. For these reasons, it deserves to be described in more details.

The test program aims to send a bundle to itself. This choice is dictated by the convenience of operating on a single DTN node, as a first step towards more complex tests. To this regard, it is worth noting that all implementations, but ION allow a DTN node to send bundles to itself without adding specific configuration instructions, while ION requires an "outduct", a "plan", a "range" and a "contact" to itself. It is therefore recommended to prospective user working with ION to perform a test with ION basic tools to check the configuration before using our test program.

The functions mentioned below have been explained in the 3.1 section (Socket), thus we will limit ourselves to report here only what strictly necessary to clarify the program test.

Since the Socket layer uses an internal registration list to keep memory of all the registrations, the program must start by calling the al_socket_init() which creates the registration list and performs compatibility checks between the executable and the BP implementation running below:

```
int main(int argc, char** argv) {
al_socket_init();
```

After this, we have to call the “al_socket_register()”, which will register the application to the Bundle Protocol; we pass the “test” demux for the “dtn” scheme and the “3” service number for the “ipn” one; the automatic choice between them is not overridden “N”, thus it will performed by our function at run time; after this, we pass the ipn node number for DTN2, which we assume to be “100”; lastly, we pass the parameters “none” and “0” as we do not want to open a TCP connection with the BP daemon:

```
al_socket_registration_descriptor rd;
al_socket_register(&rd, "test", 3, 'N', 100, "none", 0);
```

Since our goal is to send a bundle to ourselves, we must obtain the local eid associated to our registration descriptor:

```
al_types_endpoint_id source = al_socket_get_local_eid(rd);
```

Now, we have to create the bundle to be sent:

```
al_types_bundle_object bundle_out;
al_bundle_create(&bundle_out);
```

And then to set the bundle options to their default values:

```
al_bundle_options_set(&bundle_out, bundle_options);
```

The next step is to set the payload fields:

```
bundle_out.payload->location = BP_PAYLOAD_MEM;
      char* string_payload = "hello";
      bundle_out.payload->buf.buf_len = strlen(string_payload) + 1;
      bundle_out.payload->buf.buf_val = malloc(sizeof(char) *
(bundle_out.payload->buf.buf_len + 1));
      strcpy(bundle_out.payload->buf.buf_val, string_payload);
```

At this stage the bundle can be sent to ourselves, i.e. to the local eid obtained before; the “report to” address is set to none, as we do not want status reports to be issued:

```
al_types_endpoint_id dest;
dest = source;
al_socket_send(rd, bundle_out, dest, none);
```

Now we must prepare a bundle structure to receive the bundle:

```
al_bundle_create(&bundle_in);
```

And simply call the al_socket_receive:

```
al_error result = al_socket_receive(rd, &bundle_in, BP_PAYLOAD_MEM, 20);
```

Once the bundle is received, the program terminates by calling the al_socket_unregister which deletes from the registration list, the registration identified by the registration descriptor in input:

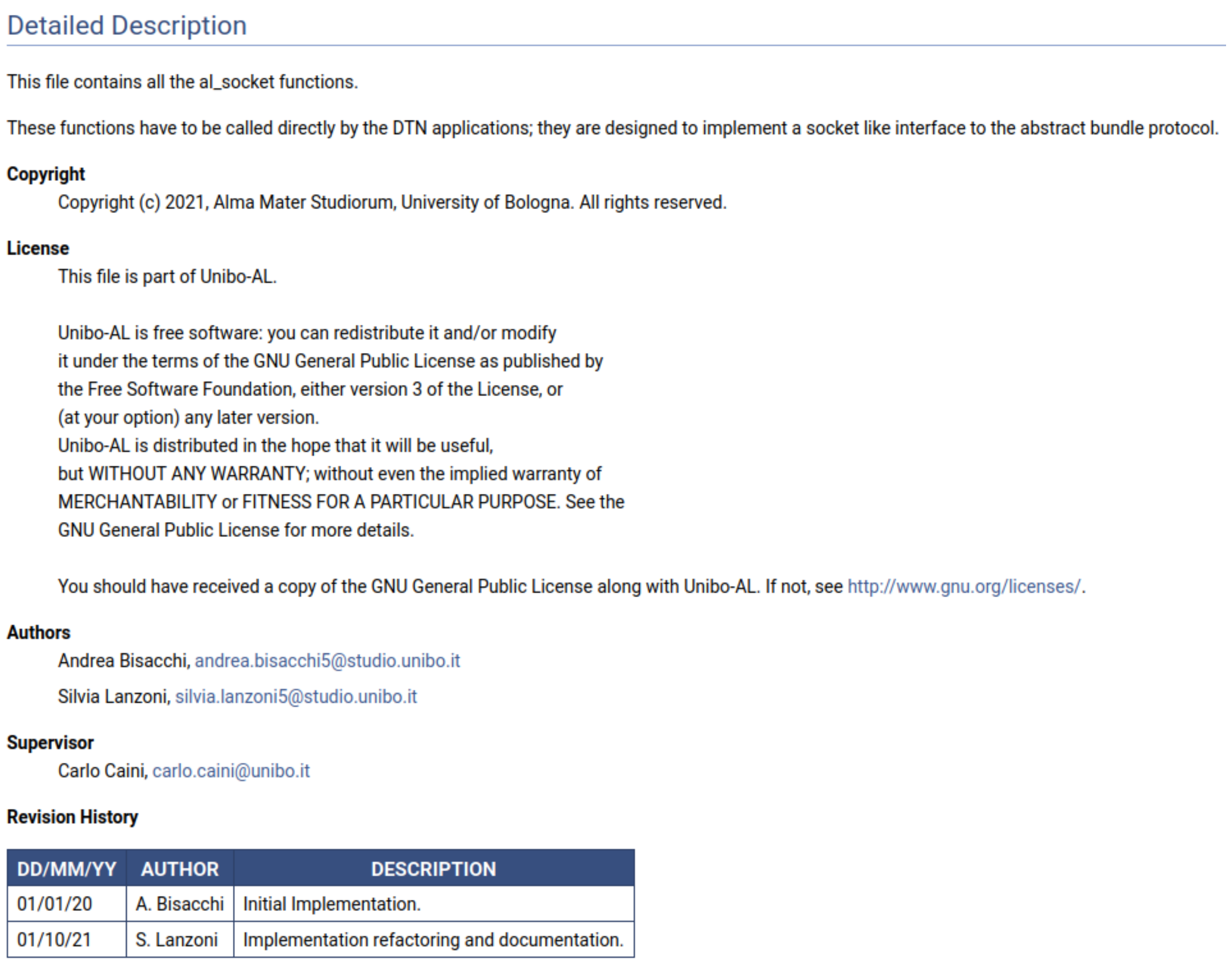
```
al_socket_unregister(rd);
```

And then the al_socket_destroy(), which eliminates the registration list:

```
al_socket_destroy();
return 0;
}
```

5.2 THE DOCUMENTATION

The Unified API code has an html documentation generated automatically by doxygen [DOXYGEN]. To this end, each file starts with a doxygen header, which states the code purpose, the authors and the license. An example of the file documentation produced is shown below, referring to the al_socket.c file.



Detailed Description

This file contains all the al_socket functions.

These functions have to be called directly by the DTN applications; they are designed to implement a socket like interface to the abstract bundle protocol.

Copyright
Copyright (c) 2021, Alma Mater Studiorum, University of Bologna. All rights reserved.

License
This file is part of Unibo-AL.

Unibo-AL is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.
Unibo-AL is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Unibo-AL. If not, see http://www.gnu.org/licenses/.

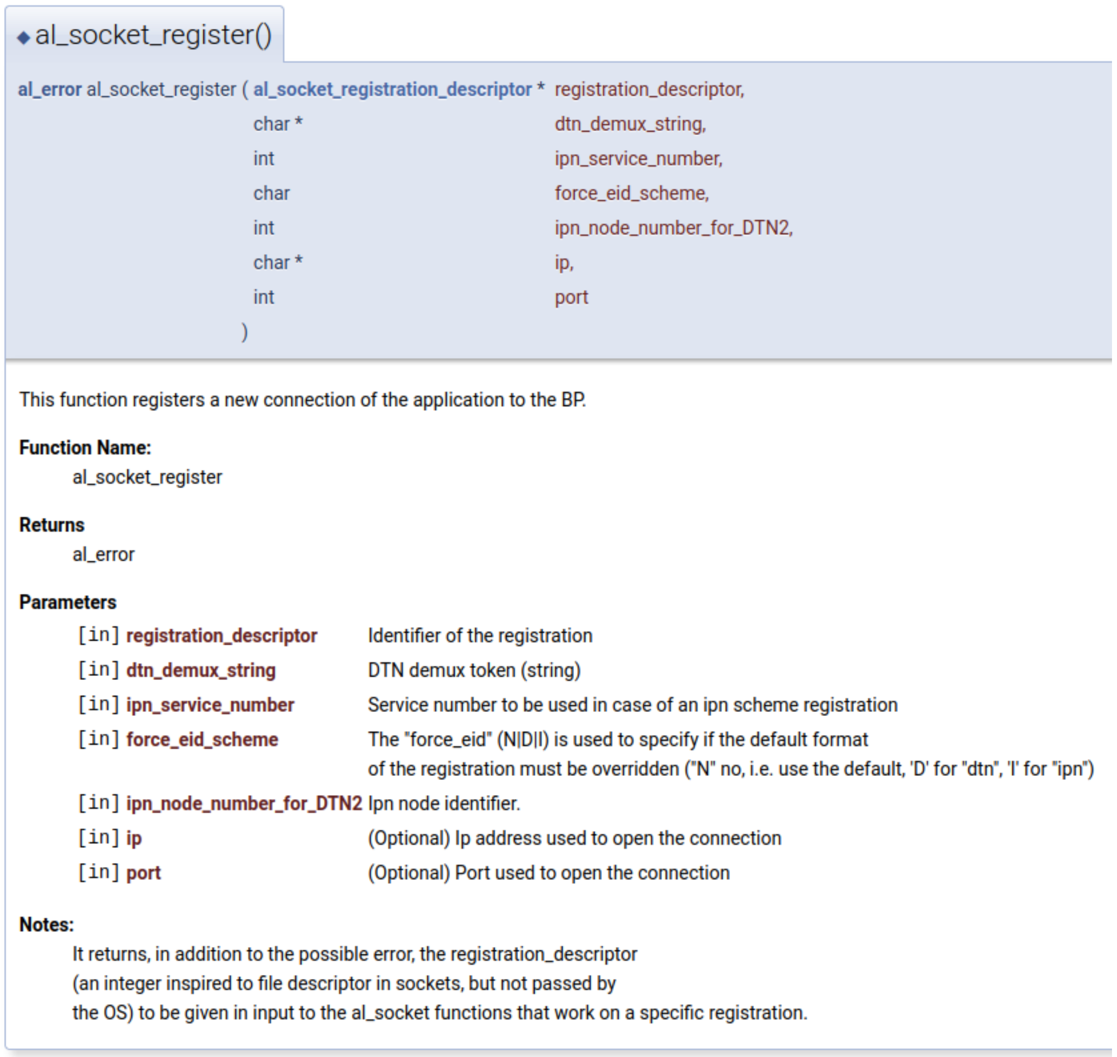
Authors
Andrea Bisacchi, andrea.bisacchi5@studio.unibo.it
Silvia Lanzoni, silvia.lanzoni5@studio.unibo.it

Supervisor
Carlo Caini, carlo.caini@unibo.it

Revision History

DD/MM/YY	AUTHOR	DESCRIPTION
01/01/20	A. Bisacchi	Initial Implementation.
01/10/21	S. Lanzoni	Implementation refactoring and documentation.

Because of their importance, also AL functions have a doxygen hedaer. The result is shown below, with reference to the al_socket_register():



◆ al_socket_register()

```
al_error al_socket_register ( al_socket_registration_descriptor *  registration_descriptor,
                              char *                               dtn_demux_string,
                              int                                  ipn_service_number,
                              char                                 force_eid_scheme,
                              int                                  ipn_node_number_for_DTN2,
                              char *                               ip,
                              int                                  port
                            )
```

This function registers a new connection of the application to the BP.

Function Name:

al_socket_register

Returns

al_error

Parameters

[in]	**registration_descriptor**	Identifier of the registration
[in]	**dtn_demux_string**	DTN demux token (string)
[in]	**ipn_service_number**	Service number to be used in case of an ipn scheme registration
[in]	**force_eid_scheme**	The "force_eid" (N|D|I) is used to specify if the default format of the registration must be overridden ("N" no, i.e. use the default, 'D' for "dtn", 'I' for "ipn")
[in]	**ipn_node_number_for_DTN2**	Ipn node identifier.
[in]	**ip**	(Optional) Ip address used to open the connection
[in]	**port**	(Optional) Port used to open the connection

Notes:

It returns, in addition to the possible error, the registration_descriptor
(an integer inspired to file descriptor in sockets, but not passed by
the OS) to be given in input to the al_socket functions that work on a specific registration.

Other functions, being less important, are generally not documented.

6 DTNsuite applications

The original aim of the Unified API predecessor was to make interoperable the DTNperf application on both ION and DTN2. After a few years, when other applications were developed (DTNbox, DTNfox, DTNproxy), we decided to group both the Abstraction Layer and all applications in an umbrella project, DTNsuite, which was released as free software and almost immediately included in the “contrib” section of the official ION release, thanks to the support of ION maintainers.
All DTNsuite components are released as free software under the GPLv3 license; the updated versions will be passed to ION maintainers as soon as sufficiently stable.
Since all the applications of the DTNsuite are based on the Unified API the changes made have been adapted and reported to each application.
The DTNsuite structure is resumed in the Figure 16:

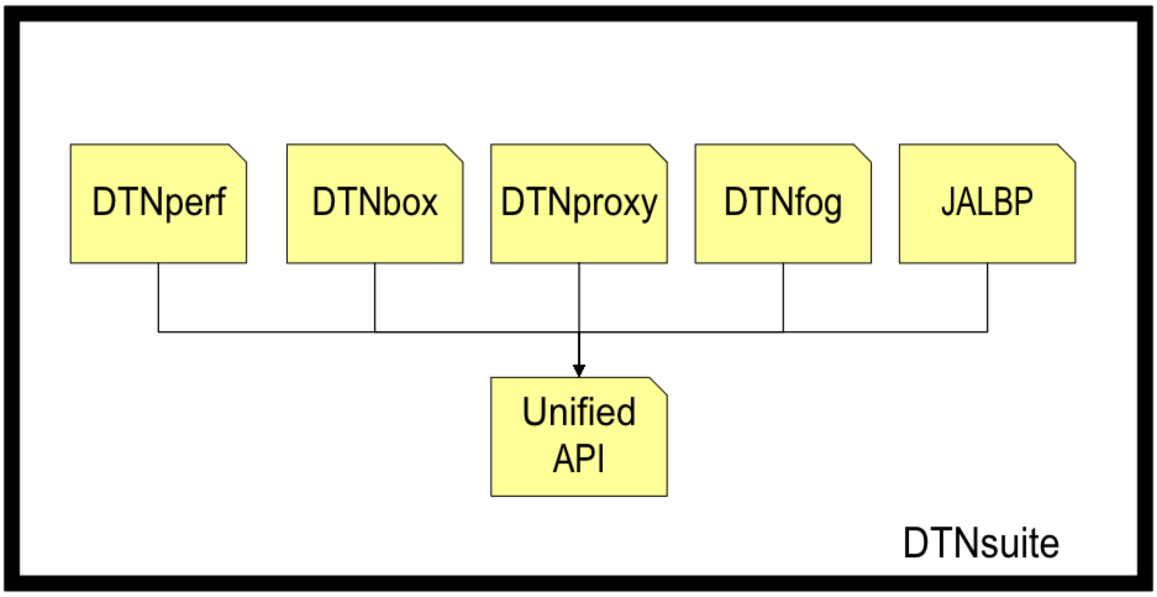


Figure 16 The DTNsuite structure. It comprehends DTNperf, DTNbox, DTNproxy, DTNfog and JALBP.

In the next sections we will summarize all the current applications.

6.1 DTNperf

DTNperf [DTNPERF] is a tool for performance evaluation in Bundle Protocol DTN environments, named after the IPerf tool. The latest major version released is the third, which, dating back to 2013, also requires a complete refactoring. This was going on during the thesis, with the client and the server almost completed by other students at theisis’end. The new version, which will take full advantage of all features of the Unified API, will be released as DTNperf_4 once completed, hopefully next July.
DTNperf has three modes: client, server and monitor, running respectively on the source, the destination and the report to node. The client creates and sends bundles, the server receives them

and acknowledge them if requested by the client, and finally the monitor collects the status reports in a .csv file to be further elaborated by a spreadsheet.

DTNperf has three transmission modes:

- Time: in this modality, bundles with a dummy payload are sent until the wanted interval has elapsed.
- Data: this modality is similar to the previous one, except that the sending process stops after the wanted amount of data has been generated and sent.
- File: in this modality the bundle payload is no more dummy, but contains a file, or segment of a file, if the file is larger than a bundle payload (minus the DTNperf header, few tens of bytes).

And two congestion control methods:

- Window based: bundle generation is ruled by a window, which limits the number of bundles in flight, similarly to the TCP congestion control window, but its dimension is fixed and not dynamic. Bundles sent in window based mode must be acknowledged by bundle ACKs sent by the server.
- Rate based: bundle generation is rate based (in bundles per second or even bit/per second). Bundles sent this way are not acknowledged.

The options of DTNperf client, the most complex component, are listed in its help shown below:

```
USAGE:
-d <dest_eid> <[-T <s> | -D <num> | -F <filename>]> <[-W <size> | -R
<rate>]> [options]
options:
-d, --destination <eid>     Destination EID (i.e. server EID).
-T, --time <seconds>        Time-mode: seconds of transmission.
-D, --data <num[B|k|M]>     Data-mode: amount data to transmit; B = Byte, k
= kByte, M = MByte. Default 'M' (MB). According to the SI and the IEEE
standards 1 MB=10^6 bytes
-F, --file <filename>       File-mode: file to transfer
-W, --window <size>         Window-based congestion control: max number of
bundles in flight).
-R, --rate <rate[k|M|b]>    Rate-based congestion control: Tx rate. k =
kbit/s, M = Mbit/s, b = bundle/s. Default is kb/s
-P, --payload <size[B|k|M]> Bundle payload size; B = Byte, k = kByte, M =
MByte. Default= 'k'. According to the SI and the IEEE standards 1 MB=10^6
bytes.
-m, --monitor <eid>         External monitor EID.
--help                      This help.
```

6.2 DTNBOX

DTNbox is an application for peer-to-peer directory synchronization between DTN nodes. By contrast to Internet applications, DTNbox does not rely on a server node, which would be impractical in challenged networks, due to long delays and intermittent connectivity. DTNbox's rationale, design and potential uses are fully described in [DTNBOX] and [DTNBOX MARS TO EARTH]. It is the most complex application of DTNsuite.

6.3 DTNPROXY

DTNproxy [DTNPROXY] is a simple DTN application to transfer files from TCP/IP and DTN nodes and vice versa.

6.4 DTNFOG

DTNfog [DTNFOG] aims to implement a fog architecture in a DTN environment. The DTNfog node receives a .tar file containing a command and a data file from a node of lower hierarchical level. If the DTNfog node is able to execute the command on the data file it gives the response back, otherwise it forwards the request to the upper DTNfog node in the hierarchical structure, and so on. Archive files can be transferred either by TCP or by Bundle Protocol. DTNfog has been designed for challenged networks, including a future Interplanetary Internet

6.5 JALBP

The Java Abstraction Layer (JAL) library is a Java wrapping of the Abstraction Layer Bundle Protocol (ALBP) [JALBP]. JAL library uses JNI (Java Native Interface) methods to interface the ALBP library written in C with Java applications. It has been designed by the correlator Andrea Bisacchi to extend to Java programmers the advantages of the ALBP. It has been updated to support the Unified API instead of the ALBP in this thesis.

7 Conclusions

The purpose of this thesis was a complete refactoring of the old al_bp. The new version, called Unified API, aims to make a DTN applications intrinsically multiplatform by decoupling them from the BP implementation, as al_bp did. However, this work also has the ambitious aim of facilitating DTN programming as much as possible, to encourage third-party programmers to write new applications, possibly more complex than the usual basic tools that are released with BP implementations.

To this end, the Unified API is based on three well separated layers, of which only the top one is seen by the application. This layer can be divided into four blocks, of which the most important is the “socket”. It is based on the initial work of the correlator Andrea Bisacchi (“extB”) and it contains a totally renovated interface to send and receive bundles designed to offer the programmer an API as close as possible to the familiar UDP socket interface.

Moreover, a significant effort has been devoted to moving the most critical aspects of DTN programming from the application to library functions. For example, the “al_socket_register”, inside the socket block, handles the two alternative EID schemes (most likely the trickiest aspect of BP programming) in a way almost transparent to applications. A second example of great importance is the introduction to the library of a powerful parsing function, able to manage compatibility between options passed by the caller and the running BP implementation, a task that may appear simple but in fact is a real challenge given the great fragmentation of options supported by the different implementations, as shown in the thesis. A third example is the introduction of a conditional printing module, which allows for easy management of verbosity at run time.

For user and maintainer convenience, the internal structure of folders, files and functions adopts a carefully designed prefix mechanism that allows an immediate correspondence between each function and its position in the Unified API structure. We have also included a simple test program, which can be used as a template in writing new applications. These features are the most evident, but improvements have been carried out at all layers.

The hope is that the Unified API will encourage third-party programmers to develop new multiplatform applications, thus contributing to the success of DTN networking.

ACKNOWLEDGEMENTS

I would first like to thank my thesis co-supervisors Dr. Keith Scott and Andrea Bisacchi for their help.

I would also like to thank Alessandro for his endless love and support. In the same way, I would like to thank Simone and Lucia who have always believed in me, and Laura who, from the beginning, has treated me like a daughter.

Finally, I would like to thank all the people which made these five years of university lighter: Giulia for always being at my side, Mattia for all the fun and the laughers we have had in these years, the guys of Gamers Arena for the endless weekends of fun (and work) between the exams and last, but not least, Elisa and Marzia who have been like sisters to me.

Thank you to all of you.

Silvia Lanzoni

REFERENCES

[AL_1_7_0] https://gitlab.com/dtnsuite/al_bp/-/blob/v1.7.0/TheAbstractionLayer_v1.7.0.pdf

[DOXYGEN] Doxygen web site: https://www.doxygen.nl/index.html

[DTN FOR ISS] A. Schlesinger, B. M. Willman, L. Pitts, S. R. Davidson and W. A. Pohlchuck, “Delay/Disruption Tolerant Networking for the International Space Station (ISS),” 2017 IEEE Aerospace Conference, Big Sky, MT, 2017, pp. 1-14. doi: 10.1109/AERO.2017.7943857.

[DTN VIRTUALIZED ENVIROMENT] R. Dudukovich, B. LaFuente, A. Hylton, B. Tomko and J. Follo, "A Distributed Approach to High-Rate Delay Tolerant Networking Within a Virtualized Environment," in the Proc. of 2021 IEEE CCAAW, 2021, pp. 1-5, doi: 10.1109/CCAAW50069.2021.9527297. HDTN code: https://github.com/nasa/HDTN

[DTN: EVOLVING INTERPLANTERY INTERNET] V. Cerf , A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss “Delay-Tolerant Network Architecture: The Evolving Interplanetary Internet,” Internet Draft, Aug. 2002.

[DTN2] code https://sourceforge.net/projects/dtn/ and manual http://dtn.sourceforge.net/DTN2/doc/manual/

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

[DTNBOX MARS TO EARTH] M. Bertolazzi, C. Caini, “Mars to Earth Data Downloading: a Directory Synchronization Approach”, Future Internet, Vol.11, No.8, pp.1-14, Aug. 2019; doi: 10.3390/fi11080173; Open access.

[DTNME] DTNME (DTN Marshall Enterprise) code: https://github.com/nasa/DTNME

[DTNPERF] C. Caini, A. d’Amico and M. Rodolfi, “DTNperf_3: a Further Enhanced Tool for Delay-/Disruption- Tolerant Networking Performance Evaluation”, in Proc. of IEEE Globecom 2013, Atlanta, USA, December 2013, pp. 3009 - 3015.

[DTNSUITE] DTNsuite experimental code: https://gitlab.com/dtnsuite

[ECOS] S. Burleigh, F. Templin, “Bundle Protocol Extended Class of Service (ECOS)”, Internet draft, May 2021. Work in progress. https://datatracker.ietf.org/doc/draft-burleigh-dtn-ecos/ Accessed on: Nov. 4, 2021.

[GETOPT] man page site: https://man7.org/linux/man-pages/man3/getopt.3.html

[GNU_MAKEFILE] gnu make documentation: https://www.gnu.org/software/make/manual/make.html

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

[ION] S. Burleigh, “Interplanetary Overlay Network (ION) – Design and Operation”, https://sourceforge.net/projects/ion-dtn/files/ion- 3.3.1.tar.gz/download, Jet Propulsion Laboratory, 2012

[MARS-EARTH] N. Alessi, C. Caini, T. de Cola, S. Martin, J. Pierce Mayer, “DTN Performance Analysis of Multi-Asset Mars Earth Communications”, International Journal of Satellite Commun. and Networking, No.xx, pp.1-16, July 2019; doi: 10.1002/sat.1326; Open access.

[PALLOTTI_2016] D. Pallotti, "Estensione dell’abstraction layer di DTNperf alle API di IBR-DTN", Tesi di Laurea in Ingegneria Informatica, Università di Bologna, 2016.

[RATIONALE FOR DTN IN SPACE] CCSDS 734.0-G-3 “Rationale, Scenarios, and Requirements for DTN in Space.” CCSDS Green Book, Issue 3, Jul. 2014. https://public.ccsds.org/Pubs/734x0g1e1.pdf

[RFC 9171] S. Burleigh, K. Fall, E. Birrane, “Bundle Protocol Version 7”, Internet RFC 9171, Jan. 2022.

[RFC4838] V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss, “Delay-Tolerant Networking Architecture”, Internet RFC 4838, April 2007, https://tools.ietf.org/html/rfc4838

[RFC5050] K. Scott, S. Burleigh, “Bundle Protocol Specification”, Internet RFC 5050, Nov. 2007.K. Scott, S. Burleigh, “Bundle Protocol Specification”, Internet RFC 5050, Nov. 2007, https://tools.ietf.org/html/rfc5050

[VALGRIND] Valgrind web site: https://valgrind.org/

[WIRESHARK] Wireshark web site: https://www.wireshark.org/

[µD3TN] web site: https://d3tn.com/ud3tn.html;