The Abstraction Layer (v.1.5)

The Abstraction Layer (AL) is a library to interface a DTN application with the Bundle Protocol independently of the actual Bundle Protocol (BP) implementation. By decoupling the application code from the BP implementation, it is possible to reuse the same application code in different DTN environments, with significant advantages in terms of application portability, maintenance and interoperability. A possible drawback is the dependence of the AL on more than one BP implementation. At present the AL supports DTN2, ION, and IBR-DTN BP implementations.
The AL consists of two elements:

- the AL Types;
- the AL API.

Note that, as the present AL version has been created to support the DTNperf_3 application, only the Types and the functions necessary for this purpose have been "abstracted". Other DTN applications could require the abstraction of others elements as well, thus extending the potentiality of the present version. The API documentation refers to AL ver: 1.5, July 2016.
Credits: Carlo Caini (Academic supervisor carlo.caini@unibo.it), Anna D'Amico (author of ION and DTN2 support), Davide Pallotti (author of IBR-DTN support davide.pallotti@studio.unibo.it), Michele Rodolfi (author of DTN2 support, michirod@gmail.com).

Table of contents

Abstraction Layer Types

The AL Types are an abstraction of DTN2, ION, and IBR-DTN types. They are defined in file "al_bp_types.h".
The types are divided into four groups: general types, registration EID types, bundle types, status report types.
In the table below the correspondence between AL, DTN2, ION, and IBR-DTN types is presented. An empty cell means that there is not any correspondence.
Note that, as the AL, the DTN2 API, and the ION API are written in C while the IBR-DTN API is written in C++ and object-oriented, the correspondence presented in this document between the IBR-DTN API and the other ones is only approximate.

Abstraction Layer	DTN2	ION	IBR-DTN
General Types			
al_bp_handle_t int *	dtn_handle_t int*	BpSAP bpsap_st *{ VEndpoint* vpoint; MetaEid endpointMetaEid; sm_SemId recvSemaphore; }	dtn::api::Client
al_bp_endpoint_id_t {char uri[AL_BP_MAX_ENDPOINT_ID]}	dtn_endpoint_id_t {char uri[DTN_MAX_ENDPOINT_ID]}	char *	dtn::data::EID

al_bp_timeval_t
u32_t
	dtn_timeval_t
	u_int
		DtnTime
		{
		unsigned long seconds;
		unsigned long nanosec;
		}
			dtn::data::Number

al_bp_timestamp_t
{
u32_t secs;
u32_t seqno;
}
	dtn_timestamp_t
	{
	u_hyper secs;
	u_hyper seqno;
	}
		BpTimestamp
		{
		unsigned long seconds;
		unsigned long count;
		}
			dtn::data::Timestamp
			dtn::data::Number

al_bp_error_t { BP_SUCCESS BP_ERRBASE; BP_ENOBPI; BP_EINVAL; BP_ENULLPNTR; BP_EUNREG; BP_ECONNECT; BP_ETIMEOUT; BP_ESIZE; BP_ENOTFOUND; BP_EINTERNAL; BP_EBUSY; BP_ENOSPACE; BP_ENOTIMPL; BP_EATTACH; BP_EBUILDEID BP_EOPEN; BP_EREG; BP_EPARSEEID; BP_ESEND; BP_ERECV; BP_ERECVINT;}			
Registration EID Types			
al_bp_reg_token_t u32_t	dtn_reg_token_t u_hyper		
al_bp_reg_id_t u32_t	dtn_reg_id_t u_int		

al_bp_reg_info_t
{ al_bp_endpoint_id_t endpoint;
al_bp_reg_id_t regid;
u32_t flags;
u32_t replay_flags;
al_bp_timeval_t expiration;
boolean_t init_passive;
al_bp_reg_token_t reg_token;
struct {
u32_t script_len;
char *script_val;} script;
}
	dtn_reg_info_t
	{ dtn_endpoint_id_t endpoint;
	dtn_reg_id_t regid;
	u_int flags;
	u_int replay_flags;
	dtn_timeval_t expiration;
	bool_t init_passive;
	dtn_reg_token_t reg_token;
	struct {
	u_int script_len;
	char *script_val;} script;
	}		

al_bp_reg_flags_t { BP_REG_DROP = 1, BP_REG_DEFER = 2, BP_REG_EXEC = 3, BP_SESSION_CUSTODY = 4, BP_SESSION_PUBLISH = 8, BP_SESSION_SUBSCRIBE = 16, BP_DELIVERY_ACKS = 32 }	dtn_reg_flags_t { DTN_REG_DROP = 1, DTN_REG_DEFER = 2, DTN_REG_EXEC = 3, DTN_SESSION_CUSTODY = 4, DTN_SESSION_PUBLISH = 8, DTN_SESSION_SUBSCRIBE = 16, DTN_DELIVERY_ACKS = 32 }	BpRecvRule {DiscardBundle, EnqueueBundle, }	
Bundle Types			

al_bp_bundle_delivery_opts_t
{
BP_DOPTS_NONE = 0,
BP_DOPTS_CUSTODY = 1,
BP_DOPTS_DELIVERY_RCPT = 2,
BP_DOPTS_RECEIVE_RCPT = 4,
BP_DOPTS_FORWARD_RCPT = 8,
BP_DOPTS_CUSTODY_RCPT = 16,
BP_DOPTS_DELETE_RCPT = 32,
BP_DOPTS_SINGLETON_DEST = 64,
BP_DOPTS_MULTINODE_DEST = 128,
BP_DOPTS_DO_NOT_FRAGMENT = 256,
}
	dtn_bundle_delivery_opts_t
	{
	DOPTS_NONE = 0,
	DOPTS_CUSTODY = 1,
	DOPTS_DELIVERY_RCPT = 2,
	DOPTS_RECEIVE_RCPT = 4,
	DOPTS_FORWARD_RCPT = 8,
	DOPTS_CUSTODY_RCPT = 16,
	DOPTS_DELETE_RCPT = 32,
	DOPTS_SINGLETON_DEST = 64,
	DOPTS_MULTINODE_DEST = 128,
	DOPTS_DO_NOT_FRAGMENT = 256,
	}
		int
		{
		BP_DELIVERED_RPT;
		BP_RECEIVED_RPT;
		BP_FORWARDED_RPT;
		BP_CUSTODY_RPT;
		BP_DELETED_RPT;
		}
			dtn::data::PrimaryBlock::FLAGS
			{
			FRAGMENT = 0x01,
			APPDATA_IS_ADMRECORD = 0x02,
			DONT_FRAGMENT = 0x04,
			CUSTODY_REQUESTED = 0x08,
			DESTINATION_IS_SINGLETON = 0x10,
			ACKOFAPP_REQUESTED = 0x20,
			REQUEST_REPORT_OF_BUNDLE_RECEPTION = 0x4000,
			REQUEST_REPORT_OF_CUSTODY_ACCEPTANCE = 0x8000,
			REQUEST_REPORT_OF_BUNDLE_FORWARDING = 0x10000,
			REQUEST_REPORT_OF_BUNDLE_DELIVERY = 0x20000,
			REQUEST_REPORT_OF_BUNDLE_DELETION = 0x40000
			}

al_bp_bundle_priority_t
{
al_bp_bundle_priority_enum priority
{
BP_PRIORITY_BULK = 0,
BP_PRIORITY_NORMAL = 1,
BP_PRIORITY_EXPEDITED = 2,
BP_PRIORITY_RESERVED = 3,
}
u32_t ordinal;
}
	dtn_bundle_priority_t
	{
	COS_BULK = 0,
	COS_NORMAL = 1,
	COS_EXPEDITED = 2,
	COS_RESERVED = 3,
	}
		int
		{
		BP_BULK_PRIORITY (0)
		BP_STD_PRIORITY (1)
		BP_EXPEDITED_PRIORITY (2)
		}
			dtn::data::PrimaryBlock::PRIORITY
			{
			PRIO_LOW = 0,
			PRIO_MEDIUM = 1,
			PRIO_HIGH = 2
			}

al_bp_extension_block_t {
u32_t type;
u32_t flags;
struct {
u32_t data_len;
char *data_val;
} data;
}
			dtn::data::ExtensionBlock
			{
			block_t blocktype;
			Bitset<ProcFlags> _procflags;
			ibrcommon::BLOB::Reference
			_blobref;
			}

al_bp_bundle_spec_t
{
al_bp_endpoint_id_t source;
al_bp_endpoint_id_t dest;
al_bp_endpoint_id_t replyto;
al_bp_bundle_priority_t priority;
al_bp_bundle_delivery_opts_t dopts;
al_bp_timeval_t expiration;
al_bp_timestamp_t creation_ts;
al_bp_reg_id_t delivery_regid;
struct {
u32_t blocks_len;
al_bp_extension_block_t
*blocks_val;
} blocks;
struct {
u32_t metadata_len;
al_bp_extension_block_t
*metadata_val;
} metadata;
boolean_t unreliable;
boolean_t critical;
u32_t flow_label;
}
	dtn_bundle_spec_t
	{
	dtn_endpoint_id_t source;
	dtn_endpoint_id_t dest;
	dtn_endpoint_id_t replyto;
	dtn_bundle_priority_t priority;
	int dopts;
	dtn_timeval_t expiration;
	dtn_timestamp_t creation_ts;
	dtn_reg_id_t delivery_regid;
	dtn_sequence_id_t sequence_id;
	dtn_sequence_id_t obsoletes_id;
	struct {
	u_int blocks_len;
	dtn_extension_block_t
	*blocks_val;
	} blocks;
	struct {
	u_int metadata_len;
	dtn_extension_block_t
	*metadata_val;
	} metadata;
	}
			dtn::data::Bundle
			{
			EID source;
			Timestamp timestamp;
			Number sequencenumber;
			Number fragmentoffset;
			Bitset<FLAGS> procflags;
			Number lifetime;
			Number appdatalength;
			EID destination;
			EID reportto;
			EID custodian;
			block_list _blocks;
			}

al_bp_bundle_payload_location_t
{
BP_PAYLOAD_FILE = 0,
BP_PAYLOAD_MEM = 1,
BP_PAYLOAD_TEMP_FILE = 2,
}
	dtn_bundle_payload_location_t
	{
	DTN_PAYLOAD_FILE = 0,
	DTN_PAYLOAD_MEM = 1,
	DTN_PAYLOAD_TEMP_FILE = 2,
	}		

al_bp_bundle_id_t
{
al_bp_endpoint_id_t source;
al_bp_timestamp_t creation_ts;
u32_t frag_offset;
u32_t orig_length;
}
	dtn_bundle_id_t
	{
	dtn_endpoint_id_t source;
	dtn_timestamp_t creation_ts;
	u_int frag_offset;
	u_int orig_length;
	}
		BundleId
		{
		EndpointId source;
		BpTimestamp creationTime;
		unsigned long fragmentOffset;
		}
			dtn::data::BundleID
			{
			EID source;
			Timestamp timestamp;
			Number sequencenumber;
			Number fragmentoffset;
			}

al_bp_bundle_payload_t
{
al_bp_bundle_payload_location_t location;
struct
{
u32_t filename_len;
char *filename_val;
} filename;
struct
{
u32_t buf_len;
char *buf_val;
} buf;
al_bp_bundle_status_report_t *status_report;
}
	dtn_bundle_payload_t
	{
	dtn_bundle_payload_location_t location;
	struct {
	u_int filename_len;
	char *filename_val;
	} filename;
	struct {
	u_int buf_len;
	char *buf_val;
	} buf;
	dtn_bundle_status_report_t *status_report;
	}
		Payload
		{
		unsigned long length;
		Object content;
		}
			dtn::data::PayloadBlock
			{
			block_t blocktype;
			Bitset<ProcFlags> _procflags;
			ibrcommon::BLOB::Reference _blobref;
			}

al_bp_bundle_object_t
{
al_bp_bundle_id_t * id;
al_bp_bundle_spec_t * spec;
al_bp_bundle_payload_t * payload;
}			

Status Report Types			

al_bp_status_report_reason_t
{
BP_SR_REASON_NO_ADDTL_INFO = 0x00,
BP_SR_REASON_LIFETIME_EXPIRED = 0x01,
BP_SR_REASON_FORWARDED_UNIDIR_LINK = 0x02,
BP_SR_REASON_TRANSMISSION_CANCELLED = 0x03,
BP_SR_REASON_DEPLETED_STORAGE = 0x04,
BP_SR_REASON_ENDPOINT_ID_UNINTELLIGIBLE = 0x05,
BP_SR_REASON_NO_ROUTE_TO_DEST = 0x06,
BP_SR_REASON_NO_TIMELY_CONTACT = 0x07,
BP_SR_REASON_BLOCK_UNINTELLIGIBLE = 0x08,
}
	dtn_status_report_reason_t
	{
	REASON_NO_ADDTL_INFO = 0x00,
	REASON_LIFETIME_EXPIRED = 0x01,
	REASON_FORWARDED_UNIDIR_LINK = 0x02,
	REASON_TRANSMISSION_CANCELLED = 0x03,
	REASON_DEPLETED_STORAGE = 0x04,
	REASON_ENDPOINT_ID_UNINTELLIGIBLE = 0x05,
	REASON_NO_ROUTE_TO_DEST = 0x06,
	REASON_NO_TIMELY_CONTACT = 0x07,
	REASON_BLOCK_UNINTELLIGIBLE = 0x08,
	}
		BpSrReason
		{
		SrLifetimeExpired = 1,
		SrUnidirectionalLink,
		SrCanceled,
		SrDepletedStorage,
		SrDestinationUnintelligible,
		SrNoKnownRoute,
		SrNoTimelyContact,
		SrBlockUnintelligible
		}
			dtn::data::StatusReportBlock::TYPE
			{
			NO_ADDITIONAL_INFORMATION = 0x00,
			LIFETIME_EXPIRED = 0x01,
			FORWARDED_OVER_UNIDIRECTIONAL_LINK = 0x02,
			TRANSMISSION_CANCELED = 0x03,
			DEPLETED_STORAGE = 0x04,
			DESTINATION_ENDPOINT_ID_UNINTELLIGIBLE = 0x05,
			NO_KNOWN_ROUTE_TO_DESTINATION_FROM_HERE = 0x06,
			NO_TIMELY_CONTACT_WITH_NEXT_NODE_ON_ROUTE = 0x07,
			BLOCK_UNINTELLIGIBLE = 0x08
			}

al_bp_status_report_flags_t
{
BP_STATUS_RECEIVED = 0x01,
BP_STATUS_CUSTODY_ACCEPTED = 0x02,
BP_STATUS_FORWARDED = 0x04,
BP_STATUS_DELIVERED = 0x08,
BP_STATUS_DELETED = 0x10,
BP_STATUS_ACKED_BY_APP = 0x20,
}
	dtn_status_report_flags_t
	{
	STATUS_RECEIVED = 0x01,
	STATUS_CUSTODY_ACCEPTED = 0x02,
	STATUS_FORWARDED = 0x04,
	STATUS_DELIVERED = 0x08,
	STATUS_DELETED = 0x10,
	STATUS_ACKED_BY_APP = 0x20,
	}
		int
		{
		BP_STATUS_RECEIVE 0
		BP_STATUS_ACCEPT 1
		BP_STATUS_FORWARD 2
		BP_STATUS_DELIVER 3
		BP_STATUS_DELETE 4
		BP_STATUS_STATS 5
		}
			dtn::data::StatusReportBlock::TYPE
			{
			RECEIPT_OF_BUNDLE = 0x01,
			CUSTODY_ACCEPTANCE_OF_BUNDLE = 0x02,
			FORWARDING_OF_BUNDLE = 0x04,
			DELIVERY_OF_BUNDLE = 0x08,
			DELETION_OF_BUNDLE = 0x10
			}

al_bp_bundle_status_report_t	dtn_bundle_status_report_t	BpStatusRpt	dtn::data::StatusReportBlock

{
al_bp_bundle_id_t bundle_id;
al_bp_status_report_reason_t reason;
al_bp_status_report_flags_t flags;
al_bp_timestamp_t receipt_ts;
al_bp_timestamp_t custody_ts;
al_bp_timestamp_t forwarding_ts;
al_bp_timestamp_t delivery_ts;
al_bp_timestamp_t deletion_ts;
al_bp_timestamp_t ack_by_app_ts;
}
	{
	dtn_bundle_id_t bundle_id;
	dtn_status_report_reason_t reason;
	dtn_status_report_flags_t flags;
	dtn_timestamp_t receipt_ts;
	dtn_timestamp_t custody_ts;
	dtn_timestamp_t forwarding_ts;
	dtn_timestamp_t delivery_ts;
	dtn_timestamp_t deletion_ts;
	dtn_timestamp_t ack_by_app_ts;
	}
		{
		BpTimestamp creationTime;
		unsigned long fragmentOffset;
		unsigned long fragmentLength;
		char *sourceEid;
		unsigned char isFragment;
		unsigned char flags;
		BpSrReason reasonCode;
		DtnTime receiptTime;
		DtnTime acceptanceTime;
		DtnTime forwardTime;
		DtnTime deliveryTime;
		DtnTime deletionTime;
		}
			{
			char status;
			char reasoncode;
			DTNTime timeof_receipt;
			DTNTime timeof_custodyaccept;
			DTNTime timeof_forwarding;
			DTNTime timeof_delivery;
			DTNTime timeof_deletion;
			BundleID bundleid;
			}

Abstraction Layer API

The AL API aims to decouple the application code from the API of a specific BP implementation. The scheme below summarizes the use of the most important AL functions.

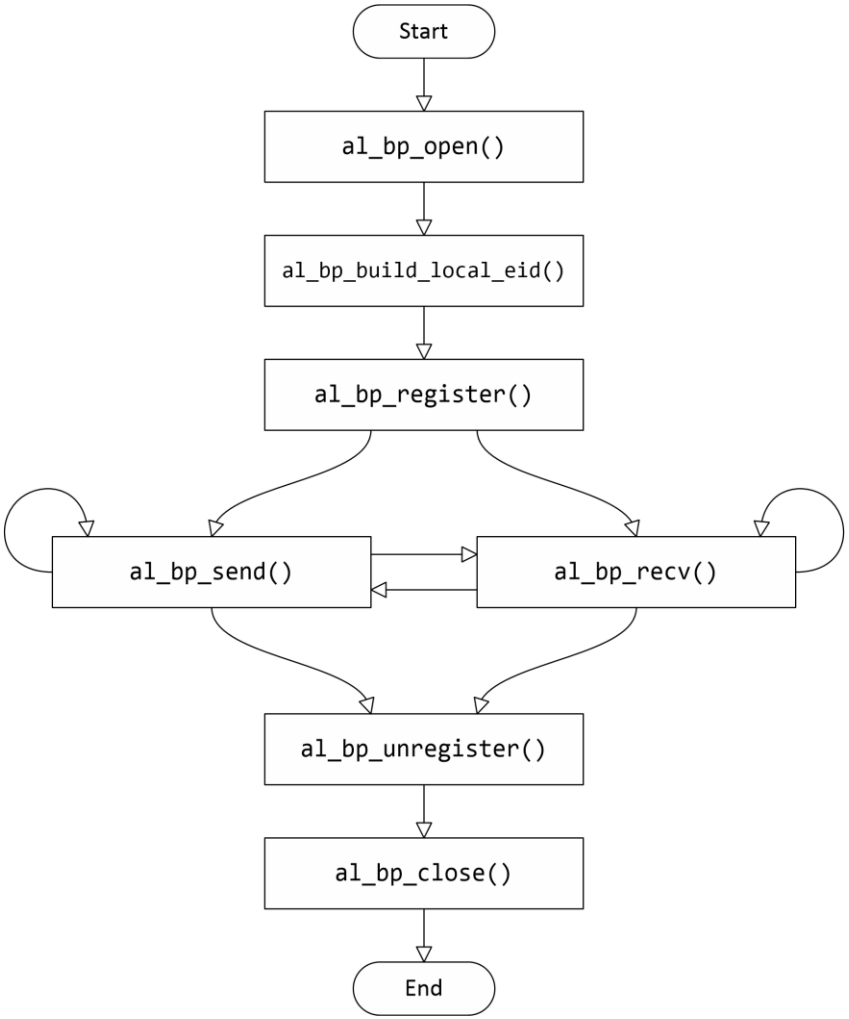

Figure 1. Typical flow of most significant AL functions.

The AL API is defined in file “al_bp_api.h”. Every AL function calls the corresponding function of the specific BP implementation. APIs of DTN2, ION, and IBR-DTN are called by specific functions declared in files “al_bp_dtn.h”, “al_bp_ion.h”, and “al_bp_ibr.h”.
The AL functions are divided into three groups: principal functions, utility functions and high level functions.
In the table below the correspondence between AL, DTN2, ION and IBR-DTN APIs is presented for the principal functions and utility functions. High level functions are not listed in the table because they do not correspond to any DTN2, ION, or IBR-DTN function. In fact, they have been designed to manage errors and to have a better control of the bundle as an object.

Abstraction Layer	DTN2	ION	IBR-DTN
Principal functions			
al_bp_open(al_bp_handle_t* handle)	dtn_open(dtn_handle_t* handle)	bp_attach( )	ibrcommon::vaddress::vaddress(const std::string &address, const int port) ibrcommon::tcpsocket::tcpsocket(const ibrcommon::vaddress &destination) ibrcommon::socketstream::socketstream(ibrcommon::socket *sock)
al_bp_open_with_ip(char *daemon_api_IP, int daemon_api_port, al_bp_handle_t* handle)	dtn_open_with_IP(char *daemon_api_IP, int daemon_api_port, dtn_handle_t* handle)		ibrcommon::vaddress::vaddress(const std::string &address, const int port) ibrcommon::tcpsocket::tcpsocket(const ibrcommon::vaddress &destination) ibrcommon::socketstream::socketstream(ibrcommon::socket *sock)
al_bp_errno(al_bp_handle_t handle)	dtn_errno(dtn_handle_t handle)	system_error_msg( )	

al_bp_build_local_eid(al_bp_handle_t handle, al_bp_endpoint_id_t* local_eid, const char* service_tag, al_bp_scheme_t type)	dtn_build_local_eid(dtn_handle_t handle, dtn_endpoint_id_t* local_eid, const char* service_tag)		
al_bp_register(al_bp_handle_t * handle, al_bp_reg_info_t * reginfo, al_bp_reg_id_t * newregid)	dtn_register(dtn_handle_t handle, dtn_reg_info_t* reginfo, dtn_reg_id_t* newregid)	addEndpoint(char *endpointName, BpRecvRule recvAction, char *recvScript) bp_open(char * eid, BpSAP * ionptr)	dtn::api::Client::Client(const std::string &app, ibrcommon::socketstream &stream)
al_bp_unregister(al_bp_handle_t handle, al_bp_reg_id_t regid, al_bp_endpoint_id_t eid)	dtn_unregister(dtn_handle_t handle, dtn_reg_id_t regid)	removeEndpoint(char *endpointName)	dtn::api::Client::~Client()
al_bp_find_registration(al_bp_handle_t handle, al_bp_endpoint_id_t * eid, al_bp_reg_id_t * newregid)	dtn_find_registration(dtn_handle_t handle, dtn_endpoint_id_t* eid, dtn_reg_id_t* newregid)	findEndpoint(char *schemeName, char *nss, VScheme *vscheme, VEndpoint **vpoint, PsmAddress *elt)	

al_bp_send(al_bp_handle_t handle, al_bp_reg_id_t regid, al_bp_bundle_spec_t* spec, al_bp_bundle_payload_t* payload, al_bp_bundle_id_t* id)	dtn_send(dtn_handle_t handle, dtn_reg_id_t regid, dtn_bundle_spec_t* spec, dtn_bundle_payload_t* payload, dtn_bundle_id_t* id)	bp_send(BpSAP sap, int mode, char * destEid, char * reportToEid, int lifespan, int classOfService, BpCustodySwitch custodySwitch, unsigned char srrFlags, int ackRequested, BpExtendedCOS* extendedCOS, Object adu, Object *newBundle)	void dtn::api::Client::operator<<(const dtn::data::Bundle &b)
al_bp_recv(al_bp_handle_t handle, al_bp_bundle_spec_t* spec, al_bp_bundle_payload_location_t location, al_bp_bundle_payload_t* payload, al_bp_timeval_t timeout)	dtn_recv(dtn_handle_t handle, dtn_bundle_spec_t* spec, dtn_bundle_payload_location_t location, dtn_bundle_payload_t* payload, dtn_timeval_t timeout)	bp_receive(BpSAP sap, BpDelivery *dlvBuffer, int timeoutSeconds)	dtn::data::Bundle dtn::api::Client::getBundle(const dtn::data::Timeout timeout = 0)
al_bp_close(al_bp_handle_t handle)	dtn_close(dtn_handle_t handle)	bp_close(BpSAP ionptr)	ibrcommon::socketstream::close()
Utility functions			
al_bp_get_implementation( )			
void al_bp_copy_eid(al_bp_endpoint_id_t* dst, al_bp_endpoint_id_t* src)	void dtn_copy_eid(dtn_endpoint_id_t* dst, dtn_endpoint_id_t* src)		
al_bp_error_t al_bp_parse_eid_string(al_bp_endpoint_id_t* eid, const char* str)	int dtn_parse_eid_string(dtn_endpoint_id_t* eid, const char* str)	int parseEidString(char *eidString, MetaEid *metaEid, VScheme **scheme, PsmAddress *schemeElt)	dtn::data::EID::EID(const std::string &value)

al_bp_error_t al_bp_get_none_endpoint(al_bp_endpoint_id_t *eid_none)			
al_bp_error_t al_bp_set_payload(al_bp_bundle_payload_t* payload, al_bp_bundle_payload_location_t location, char* val, int len)	int dtn_set_payload(dtn_bundle_payload_t* payload, dtn_bundle_payload_location_t location, char* val, int len)		
void al_bp_free_payload(al_bp_bundle_payload_t* payload)	int dtn_free_payload(dtn_bundle_payload_t* payload)	zco_destroy_file_ref(Sdr sdr, Object fileRef)	
void al_bp_free_extension_blocks(al_bp_bundle_spec_t* spec)			
void al_bp_free_metadata_blocks(al_bp_bundle_spec_t* spec)			
const char* al_bp_status_report_reason_to_str(al_bp_status_report_reason_t err)	const char* dtn_status_report_reason_to_str(dtn_status_report_reason_t err*)*		
char* al_bp_strerror(int err)			

Below we provide the reader with some basic information about the most important AL functions, by pointing out the differences in case they run on top of DTN2, ION, or IBR-DTN BP implementations.

al_bp_open
al_bp_error_t al_bp_open(al_bp_handle_t handle)*

Opens the connection between the application and the BP daemon.
In DTN2 and IBR-DTN the function also initializes the handle.

al_bp_build_local_eid
al_bp_error_t al_bp_build_local_eid(al_bp_handle_t handle, al_bp_endpoint_id_t local_eid, const char* service_tag, al_bp_scheme_t type);*

Creates the local EID.
In DTN2 and IBR-DTN the local EID is retrieved from the handle.
In ION the local eid is built with specific rules, depending on the *type* value, (*CBHE or DTN).*
if CBHE, the "ipn" scheme is used and the local EID will be **ipn:<own_number>:<own_pid>**
if DTN the "dtn" scheme is used and the local EID will be **dtn://<local_hostname>/<service_tag>**.

al_bp_register
*al_bp_error_t al_bp_register(al_bp_handle_t * handle, al_bp_reg_info_t* reginfo, al_bp_reg_id_t* newregid)*

Registers the local EID to the BP daemon. In ION it also calls the API *bp_open()* that initializes the handle and allows the application to start sending and receiving bundles.

High Level functions

High Level functions aim to manage the bundle as an object with "get" and "set" functions for almost every bundle parameter.

al_bp_bundle_send
*al_bp_error_t al_bp_bundle_send(al_bp_handle_t handle, al_bp_reg_id_t regid, al_bp_bundle_object_t * bundle_object)*

Sends the bundle object.

al_bp_bundle_receive
al_bp_error_t al_bp_bundle_receive(al_bp_handle_t handle, al_bp_bundle_object_t bundle_object, al_bp_bundle_payload_location_t

payload_location, al_bp_timeval_t timeout)

Receives a bundle object.

al_bp_bundle_create
*al_bp_error_t al_bp_bundle_create(al_bp_bundle_object_t * bundle_object)*

Creates an empty bundle object.

al_bp_bundle_free
*al_bp_error_t al_bp_bundle_free(al_bp_bundle_object_t * bundle_object)*

Deletes the bundle object from memory.

al_bp_bundle_get_id
*al_bp_error_t al_bp_bundle_get_id(al_bp_bundle_object_t bundle_object, al_bp_bundle_id_t ** bundle_id)*

Retrieves the bundle Id from the bundle object.

al_bp_bundle_set_payload_location
*al_bp_error_t al_bp_bundle_set_payload_location(al_bp_bundle_object_t * bundle_object, al_bp_bundle_payload_location_t location)*

Sets the bundle payload location: either memory or file.

al_bp_bundle_get_payload_location
*al_bp_error_t al_bp_bundle_get_payload_location(al_bp_bundle_object_t bundle_object, al_bp_bundle_payload_location_t * location)*

Returns the bundle payload location.

al_bp_bundle_get_payload_size
*al_bp_error_t al_bp_bundle_get_payload_size(al_bp_bundle_object_t bundle_object, u32_t * size)*

Returns the bundle payload size.

al_bp_bundle_get_payload_file
*al_bp_error_t al_bp_bundle_get_payload_file(al_bp_bundle_object_t bundle_object, char_t ** filename, u32_t * filename_len)*

Returns the value of the payload if it is saved in a file.

bp_bundle_get_payload_mem
*al_bp_error_t al_bp_bundle_get_payload_mem(al_bp_bundle_object_t bundle_object, char ** buf, u32_t * buf_len)*

Returns the value of the payload if it is stored in memory.

al_bp_bundle_set_payload_file
*al_bp_error_t al_bp_bundle_set_payload_file(al_bp_bundle_object_t * bundle_object, char_t * filename, u32_t filename_len)*

Sets the value of the payload if it is saved in a file.

al_bp_bundle_set_payload_mem
*al_bp_error_t al_bp_bundle_set_payload_mem(al_bp_bundle_object_t * bundle_object, * buf, u32_t buf_len)*

Sets the value of the payload if it is saved in memory.

al_bp_bundle_get_source
*al_bp_error_t al_bp_bundle_get_source(al_bp_bundle_object_t bundle_object, al_bp_endpoint_id_t * source)*

Returns the bundle's source EID.

al_bp_bundle_set_source
*al_bp_error_t al_bp_bundle_set_source(al_bp_bundle_object_t * bundle_object, al_bp_endpoint_id_t source)*

Sets the bundle's source EID.

al_bp_bundle_get_dest
*al_bp_error_t al_bp_bundle_get_dest(al_bp_bundle_object_t bundle_object, al_bp_endpoint_id_t * dest)*

Returns the bundle's destination EID.

al_bp_bundle_set_dest
*al_bp_error_t al_bp_bundle_set_dest(al_bp_bundle_object_t * bundle_object, al_bp_endpoint_id_t dest)*

Sets the bundle's destination EID.

al_bp_bundle_get_replyto
*al_bp_error_t al_bp_bundle_get_replyto(al_bp_bundle_object_t bundle_object, al_bp_endpoint_id_t * replyto)*

Returns the status report's destination EID.

al_bp_bundle_set_replyto
*al_bp_error_t al_bp_bundle_set_replyto(al_bp_bundle_object_t * bundle_object, al_bp_endpoint_id_t replyto)*

Sets the status report's destination EID.

al_bp_bundle_get_priority
*al_bp_error_t al_bp_bundle_get_priority(al_bp_bundle_object_t bundle_object, al_bp_bundle_priority_t * priority)*

Returns the bundle's priority.

al_bp_bundle_set_priority
*al_bp_error_t al_bp_bundle_set_priority(al_bp_bundle_object_t * bundle_object, al_bp_bundle_priority_t priority)*

Sets the bundle's priority.

al_bp_bundle_get_expiration
*al_bp_error_t al_bp_bundle_get_expiration(al_bp_bundle_object_t bundle_object, al_bp_timeval_t * exp)*

Return the bundle's expiration time.

al_bp_bundle_set_expiration
*al_bp_error_t al_bp_bundle_set_expiration(al_bp_bundle_object_t * bundle_object, al_bp_timeval_t exp)*

Sets the bundle's expiration time.

al_bp_bundle_get_creation_timestamp
*al_bp_error_t al_bp_bundle_get_creation_timestamp(al_bp_bundle_object_t bundle_object, al_bp_timestamp_t * ts)*

Returns the bundle's creation timestamp.

al_bp_bundle_set_creation_timestamp
*al_bp_error_t al_bp_bundle_set_creation_timestamp(al_bp_bundle_object_t * bundle_object, al_bp_timestamp_t ts)*

Sets the bundle's creation timestamp.

al_bp_bundle_get_delivery_opts
*al_bp_error_t al_bp_bundle_get_delivery_opts(al_bp_bundle_object_t bundle_object, al_bp_bundle_delivery_opts_t * dopts)*

Returns the bundle's delivery options.

al_bp_bundle_set_delivery_opts
*al_bp_error_t al_bp_bundle_set_delivery_opts(al_bp_bundle_object_t * bundle_object, al_bp_bundle_delivery_opts_t dopts)*

Sets the bundle's delivery options.

al_bp_bundle_get_status_report
*al_bp_error_t al_bp_bundle_get_status_report(al_bp_bundle_object_t bundle_object, al_bp_bundle_status_report_t ** status_report)*

Returns the bundle's status report.

File and API structure

The organization of AL files is the following:

- dtnperf/al_bp/src: contains the declaration files and the implementation of the interface, in al_bp_api.c;
- dtnperf/al_bp/src/bp_implementations: contains the interfaces to DTN2, ION, and IBR-DTN APIs (al_bp_dtn.c, al_bp_ion.c, al_bp_ibr.cpp, etc)

From the application, which uses the al_bp API, to the API provided by the specific BP implementation, we have a chain of intermediate calls.
Note that the AL is compiled for a BP implementation if the path to this implementation directory is provided as a parameter after the "make" command. Multiple choices are allowed, so that AL can be compiled for whatever combination of BP implementations. The most relevant cases are for one implementation only, or for all implementations, but also all possible couples of BP implementations are allowed. The (sole) BP implementation that is on is determined (although multiple BP implementations can be installed, only one BP daemon can be active at a given instant) at run time.
Let us explain this with an example, referring to al_bp_send (see figure below).

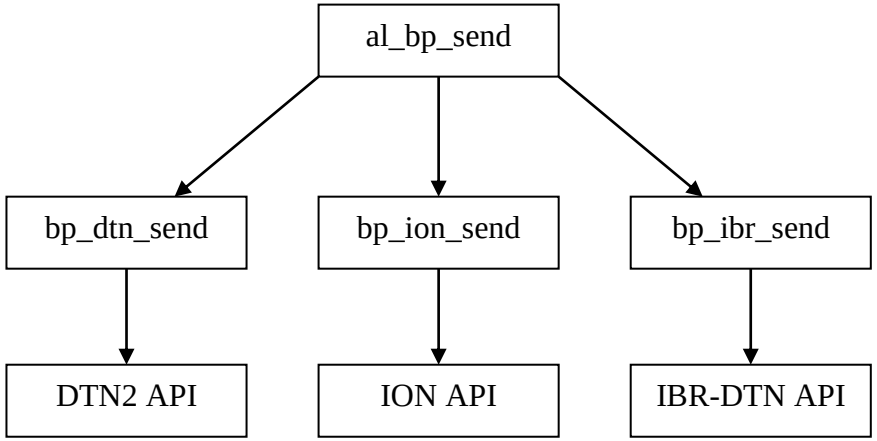

Figure 2. Example of relationship between the AL functions and the BP implementation API.

al_bp_send is defined in al_bp_api.c and is called by the application. It just contains a switch to the BP implementations.

- If DTN2 daemon is on,
    - al_bp_send (in al_bp_api.c) calls bp_dtn_send (in al_bp_dtn.c). Note that in al_bp_dtn_send.c there is a real implementation of the API, to be used when the AL is compiled (at least) for DTN2, and a dummy implementation that does not call the API of DTN2, to avoid compilation errors (e.g. due to lack of DTN2 libraries) otherwise.
    - bp_dtn_send then calls the DTN2 API.

- If ION is on,
    - al_bp_send (in al_bp_api.c) calls bp_ion_send (in al_bp_ion.c). To avoid compilation errors, in al_bp_ion.c there are both a real and a dummy implementation of the API, as before.
    - bp_ion_send then calls the ION API.
- If IBR-DTN is on,
    - al_bp_send (in al_bp_api.c) calls bp_ibr_send (in al_bp_ibr.cpp). To avoid compilation errors, in al_bp_ibr.cpp there are both a real and a dummy implementation of the API, as before.
    - bp_ibr_send then calls the IBR-DTN API.

Type conversions are in files “al_bp_dtn_conversions.c” and “al_bp_ion_conversions.c”.
For instance, the prefix “al_ion” means that the function takes a BP abstract type and returns an ION type, so the conversion is AL → ION, while the prefix “ion_al” means that the function takes an ION type and returns a BP abstract type, so the conversion is ION → AL.
There are no dedicated functions to convert between AL and IBR-DTN types. That is due to the lack of correspondence between most of the AL and IBR-DTN types, which results in conversions being performed directly inside the bp_ibr functions, when needed.