ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea in Ingegneria Informatica

TESI DI LAUREA

in

Fondamenti di Telecomunicazioni T

Estensione dell'Abstraction Layer di DTNperf alle API di IBR-DTN

CANDIDATO
Davide Pallotti

RELATORE
Prof. Ing. Carlo Caini

Anno Accademico 2015/16

Sessione I

PREMESSA

Sono dette “challenged networks” quelle reti in cui lunghi ritardi, partizionamenti frequenti, interruzioni dei canali di trasmissione, elevati tassi di errore e di perdita o altre problematiche analoghe non consentono l’impiego dei classici protocolli di comunicazione di Internet, in particolare la suite TCP/IP. Il primo scenario di interesse a presentare queste caratteristiche è stato l’Interplanetary Networking, ma le stesse criticità si riscontrano anche in reti molto diverse, quali reti di sensori, reti militari, reti senza fili terrestri, reti di satelliti, ambienti estremi e isolati. Il Delay-/Disruption-Tolerant Networking, o DTN, propone una soluzione per il trasferimento di dati attraverso queste tipologie di reti.

L’architettura DTN, descritta nella RFC 4838, prevede l’introduzione, tra i livelli di trasporto e applicativo, del cosiddetto “bundle layer”, che si occupa di veicolare messaggi, denominati bundle, secondo l’approccio store-and-forward. Il funzionamento è analogo alla modalità di consegna dei pacchi postali: ogni nodo DTN conserva persistentemente un bundle fino a quando non vede presentarsi l’opportunità di inoltrarlo al nodo successivo nel percorso verso la destinazione. Il protocollo impiegato all’interno del bundle layer è il Bundle Protocol, definito dalla RFC 5050.

Le principali implementazioni del Bundle Protocol sono tre: DTN2, l’implementazione di riferimento destinata a rappresentare fedelmente le caratteristiche delle DTN; ION, sviluppata da NASA-JPL (Jet Propulsion Laboratory) e maggiormente orientata alle comunicazioni spaziali; IBR-DTN, realizzata dall’Università Tecnica di Braunschweig e rivolta principalmente ai dispositivi mobili ed embedded. Ciascuna di esse offre API che consentono la scrittura di applicazioni che possano comunicare con il Bundle Protocol per inviare e ricevere bundle. Inoltre, poiché il protocollo di riferimento è lo stesso, tali implementazioni sono, o dovrebbero essere, interoperabili.

DTNperf è uno strumento applicativo progettato per la valutazione delle prestazioni in un’architettura DTN, in particolare per la misurazione del goodput e la raccolta delle informazioni relative al transito dei bundle. Una sessione di test con DTNperf coinvolge tre entità: un client, che genera e invia bundle; un server, che attende la ricezione di bundle e risponde al client; un monitor, che raccoglie gli “status report” prodotti dal Bundle Protocol riguardanti l’inoltro dei bundle e genera un file di log. La terza e più recente iterazione, denominata DTNperf_3, è compatibile sia con DTN2 che con ION, senza che sia stato necessario realizzare due

versioni distinte del programma. Ciò è possibile grazie all'introduzione di una libreria appositamente sviluppata, denominata "Abstraction Layer", che fornisce un'unica interfaccia per l'interazione con le diverse implementazioni del Bundle Protocol e che solo internamente si occupa di invocare le API specifiche dell'implementazione attiva sulla macchina. L'Abstraction Layer è totalmente indipendente da DTNperf e dunque riutilizzabile in altre applicazioni DTN che debbano poter essere eseguite al di sopra di più implementazioni del Bundle Protocol.

Obiettivo di questa tesi è estendere l'Abstraction Layer affinché supporti anche le API di IBR-DTN, cosicché DTNperf_3 possa essere impiegato indifferentemente su DTN2, ION e IBR-DTN. Il lavoro sarà ripartito su tre fasi. Nella prima fase esploreremo IBR-DTN, preoccupandoci di ottenere un'installazione correttamente configurata e funzionante, sperimentandone gli strumenti di comunicazione e studiandone le API. Nella seconda fase procederemo all'effettiva estensione del codice dell'Abstraction Layer, servendoci delle API analizzate durante la prima fase. Nella terza, infine, effettueremo alcuni test per verificare il corretto funzionamento di DTNperf a seguito delle modifiche apportate, sia in ambiente esclusivamente IBR-DTN, sia mettendo in comunicazione IBR-DTN con DTN2 e ION.

Questo documento espone il lavoro svolto ripercorrendone i passaggi in ordine logico. Il primo capitolo illustra i concetti fondamentali delle DTN e accenna alle tre già citate principali implementazioni del Bundle Protocol. Il secondo capitolo è riservato a IBR-DTN: a una guida all'installazione, alla configurazione e all'uso di base segue una sezione dedicata alla descrizione degli elementi fondamentali delle sue API. Il terzo capitolo fornisce una panoramica di DTNperf_3 e passa poi ad esaminare la struttura dell'Abstraction Layer. Il quarto capitolo costituisce il cuore della tesi, e presenta le estensioni e le modifiche apportate al codice dell'Abstraction Layer, concentrandosi in particolare sui problemi riscontrati e sulle soluzioni proposte. Il quinto capitolo è dedicato ai conseguenti aggiustamenti richiesti da DTNperf stessa e ai test di validazione. Infine, le conclusioni riassumono il nostro operato e ne tratteggiano le prospettive di applicazione.

INDICE

Premessa **3**

Indice **5**

1 Introduzione **7**

1.1 Delay-/Disruption-Tolerant Networking 7

1.1.1 Architettura di rete 8

1.2 Specifiche e implementazioni del Bundle Protocol 11

2 L'implementazione IBR-DTN **13**

2.1 Guida all'installazione 13

2.1.1 Installazione da repository 13

2.1.2 Installazione da sorgenti 14

2.2 Avvio e configurazione 16

2.2.1 File di configurazione 16

2.3 Strumenti di base 18

2.4 API 19

2.4.1 Cenni sulle API testuali 20

2.4.2 API C++ 20

3 Il software DTNperf_3 .... **29**

3.1 Descrizione generale 29

3.1.1 Il client 29

3.1.2 Il server 30

3.1.3 Il monitor 31

3.2 Guida all'uso 31

3.2.1 Il client 31

3.2.2 Il server 33

3.2.3 Il monitor 33

3.2.4 Esempi 34

3.3 L'Abstraction Layer 35

3.3.1 Tipi di base 37

3.3.2 Strutture dati 37

3.3.3 Funzioni e procedure 42

4 Estensione dell'Abstraction Layer a IBR-DTN ... 51
4.1 Utilizzo di una libreria C++ in un programma scritto in C ... 51
4.2 Integrazione dei Makefile ... 54
4.3 Prime estensioni del codice ... 55
4.4 Implementazione delle funzioni di basso livello ... 57
4.4.1 IbrHandle ... 57
4.4.2 bp_ibr_open e bp_ibr_open_with_ip ... 58
4.4.3 bp_ibr_errno ... 58
4.4.4 bp_ibr_register e bp_ibr_find_registration ... 58
4.4.5 bp_ibr_build_local_eid: ottenere l'EID del nodo locale ... 59
4.4.6 bp_ibr_send: restituire l'identità del bundle inviato ... 61
4.4.7 bp_ibr_recv ... 62
4.4.8 bp_ibr_unregister ... 63
4.4.9 bp_ibr_close ... 63
4.4.10 Funzioni di utilità ... 63
5 Modifiche a DTNperf e testing ... 65
5.1 Modifiche a DTNperf ... 65
5.1.1 Estensioni e schemi degli EID ... 65
5.1.2 Validazione degli ACK su IBR-DTN ... 66
5.2 Testing su testbed in Virtualbricks ... 67
5.2.1 Virtualbricks ... 67
5.2.2 Test su IBR-DTN ... 68
5.2.3 Interoperabilità con DTN2 ... 70
5.2.4 Interoperabilità con ION ... 71
6 Conclusioni ... 73
7 Bibliografia ... 75

1 INTRODUZIONE

1.1 Delay-/Disruption-Tolerant Networking

I protocolli di comunicazione utilizzati in Internet, in particolare la suite TCP/IP, si basano su alcuni assunti piuttosto forti riguardanti l'architettura di rete, tra cui:

- per tutta la durata della comunicazione esiste un percorso ininterrotto tra il mittente e il destinatario;
- i ritardi di comunicazione sono contenuti e paragonabili nelle due direzioni;
- gli errori di trasmissione e le perdite di pacchetti sono relativamente ridotti, e la ritrasmissione da parte della sorgente è un metodo efficace per ripararvi, proprio per i brevi ritardi;
- tutti i nodi della rete supportano gli stessi protocolli, quelli della suite TCP/IP.

Chiamiamo "challenged networks" quelle reti in cui almeno una di queste ipotesi non è verificata e in cui, dunque, l'architettura e i protocolli TCP/IP non sono sufficienti. Per supportare la comunicazione attraverso tali reti è stato introdotto l'approccio del *Delay-/Disruption-Tolerant Networking*, brevemente DTN.

L'architettura DTN nasce nell'ambito dell'*Interplanetary Networking* (IPN), scenario caratterizzato da ritardi anche dell'ordine delle decine di minuti, ad esempio sulla distanza Terra-Marte, e da interruzioni della linea di visibilità necessaria alla comunicazione radio, dovute ad esempio al moto dei pianeti e alla rivoluzione dei satelliti intorno al proprio pianeta. Il dominio di applicazione dei primi concetti sviluppati per IPN è stato poi esteso a tutte quelle reti che, seppur strutturalmente molto diverse dall'Internet interplanetario ed eterogenee fra loro, presentano criticità simili: è il caso di reti di sensori utilizzati in ambiente oceanico, reti tattiche militari, reti senza fili terrestri o reti di satelliti con connettività saltuaria o periodica, comunità di persone poste in luoghi isolati o dalle condizioni estreme e, in futuro, reti costituite dagli innumerevoli dispositivi dell'*Internet of Things*.

L'idea di fondo richiama le modalità di consegna dei pacchi postali: la comunicazione si basa, anziché su flussi continui di dati o pacchetti di al più qualche migliaio di byte, su messaggi potenzialmente di grandi dimensioni, che vengono mantenuti dal nodo che ne è in possesso finché non diventa possibile l'inoltro al nodo seguente verso la destinazione, anch'esso in grado di conservare il messaggio fino all'inoltro successivo. Per contro, naturalmente, è drasticamente ridotta la possibilità di realizzare dialoghi applicativi bidirezionali, cioè basati su continue

richieste e risposte, a causa degli stessi vincoli fisici posti dalle reti DTN, quali un ritardo di propagazione ineliminabile.

Punto di riferimento per l'attività di ricerca e standardizzazione è stato l'IRTF (Internet Research Task Force) DTN Research Group [DTNRG]. La maturazione dei risultati ha portato al passaggio da IRTF a IETF (Internet Engineering Task Force), per cui il DTNRG ha cessato la propria attività e l'IETF DTN Working Group ne ha assunto le funzioni [DTNWG]. In parallelo, per le sole applicazioni spaziali, la standardizzazione è svolta anche dal CCSDS (The Consultative Committee for Space Data Systems), ente di standardizzazione che raccoglie tutte le maggiori agenzie spaziali, a partire da NASA ed ESA [CCSDS].

1.1.1 Architettura di rete

1.1.1.1 Il bundle layer

L'architettura DTN, descritta nella [RFC4838], è realizzata aggiungendo alla pila dei protocolli di comunicazione un nuovo strato, il *bundle layer*, al di sopra del livello di trasporto (e al di sotto di quello applicativo). Un'applicazione pensata per le DTN invia messaggi di lunghezza arbitraria detti *Application Data Unit* (ADU), preferibilmente autocontenuti dal punto di vista dell'informazione che recano, che vengono convertiti dal bundle layer in uno o più *bundle*, l'unità dati di base di tale livello.

Il bundle layer si occupa dell'inoltro di bundle da un nodo DTN, o *Bundle Protocol Agent* (BPA), al successivo. Ogni nodo DTN conserva un bundle finché la comunicazione per l'inoltro non diventa possibile, cioè finché non è disponibile un *contatto*. Affinché i bundle in attesa in un nodo sopravvivano a un eventuale riavvio del sistema, è solitamente utilizzata una qualche forma di persistenza su memoria di massa. Questo approccio, diametralmente opposto rispetto al TCP/IP, in cui un pacchetto che non possa essere instradato immediatamente da un router viene scartato, va sotto il nome di *store-and-forward*, memorizza-e-inoltra.

Al di sotto del bundle layer, la comunicazione tra nodi DTN avviene tramite un sottostrato, detto *convergence layer*, che funge da interfaccia verso un protocollo di trasporto, e può essere differente per le diverse coppie di nodi DTN successivi. Si hanno quindi convergence layer per i protocolli di trasporto utilizzati in Internet, come TCP e UDP, nel qual caso il contatto può passare attraverso nodi intermedi che non possiedono il bundle layer nel proprio stack di rete, e per protocolli pensati specificamente per le DTN, come LTP (Licklider Transmission Protocol), impiegato su collegamenti a ritardo elevato, ad esempio attraverso lo spazio. [Farrell]

1.1.1.2 Endpoint Identifier

Un *endpoint* DTN è un insieme di zero o più nodi DTN identificati da un *Endpoint Identifier* (EID). Il caso più comune è quello di un endpoint costituito da un solo nodo, detto *singleton*, ma EID comprendenti più nodi permettono di inviare bundle in multicast.

Gli EID seguono la sintassi degli URI. In particolare, la prima parte indica lo schema, mentre la seconda, la SSP (*scheme-specific part*), è una sequenza di caratteri che segue le regole dello schema. In generale, poi, una porzione della SSP indica il nodo, un'altra, il *demux token*, individua la singola applicazione (analogamente alle porte UDP e TCP).

I due schemi tipicamente usati sono DTN e CBHE. Un EID nello schema DTN ha la forma

`dtn://`*`node/application`*

dove *`node`* e *`application`* sono stringhe qualsiasi, esclusi alcuni caratteri speciali. Ad esempio

`dtn://host2.dtn/ping`

Un EID nello schema CBHE (*Compressed Bundle Header Encoding*), comunemente indicato anche come IPN, ha la forma

`ipn:node.application`

dove *`node`* e *`application`* sono stringhe numeriche. Ad esempio

`ipn:81.2`

Si noti che nello schema CBHE, al contrario di quello DTN, il demux token, numerico, deve sempre essere presente; in particolare, per indicare il bundle agent stesso si utilizza il numero "0".

La richiesta da parte di un'applicazione di ricevere Application Data Unit destinate a un determinato EID è detta *registrazione*.

1.1.1.3 Delivery option e status report

Per ogni bundle è possibile specificare una serie di *delivery option* che richiedono ai nodi DTN per cui il bundle transita di intraprendere determinate azioni. La maggioranza di queste opzioni riguardano l'invio, al nodo mittente o a un altro nodo predisposto, di bundle informativi, detti *bundle status report* (BSR), al fine di monitorare il percorso del bundle, in analogia con quanto accade per un pacco inviato con corriere. Ad esempio, è possibile richiedere l'invio di uno status report quando il bundle è ricevuto dal destinatario, quando giunge a un nodo DTN

intermedio, quando viene inoltrato da un nodo al successivo, quando viene eliminato.

1.1.1.4 Custodia

Le restanti delivery option sono legate alla richiesta, da parte dell'applicazione, del trasferimento del bundle con custodia, che introduce la consegna affidabile a livello di bundle layer tramite timeout e ritrasmissione. Quando il mittente emette un bundle con la richiesta di custodia, dapprima il nodo sorgente e in seguito quelli successivi sono liberi di accettare o meno la richiesta; se un nodo la accetta, diventa il custode del bundle, cioè il nodo che si impegna a far giungere il bundle stesso a destinazione.

Quando il custode corrente invia un bundle al nodo successivo, viene richiesto il trasferimento della custodia (*custody transfer*) e avviato un timer di ritrasmissione. Se il custode corrente riceve dal nodo successivo un *custody signal*, uno speciale bundle amministrativo di accettazione della custodia, prima dello scadere del timer, significa che il nodo successivo ha accettato la custodia: quest'ultimo diventa il nuovo custode corrente e il vecchio custode può rilasciare la custodia e cancellare la propria copia del bundle. Viceversa, il custode corrente ritrasmette il bundle.

Il meccanismo di ritrasmissione realizzato con la custodia non coinvolge necessariamente un singolo hop DTN, in quanto uno o più nodi intermedi possono rifiutare la custodia. In tal caso la ritrasmissione copre più hop DTN, tra due custodi successivi ma non immediatamente adiacenti.

Si noti che se invece l'applicazione non richiede il trasferimento con custodia, il successo della consegna del bundle dipende soltanto dall'affidabilità dei protocolli sottostanti il bundle layer.

1.1.1.5 Struttura a blocchi di un bundle

I bundle hanno un formato definito che li vede costituiti da blocchi, contenenti dati applicativi, informazioni per la consegna o altro.

Il primo blocco di un bundle è sempre il *primary block*, che riporta l'EID del mittente, l'EID del destinatario, l'EID del destinatario degli status report ("report-to EID"), l'eventuale EID del custode corrente, le delivery option, la priorità del bundle, la vita massima del bundle (*lifetime*, oltre la quale potrà essere scartato) e il *creation timestamp*. Il creation timestamp è ottenuto dalla concatenazione del momento della creazione del bundle, con granularità di un secondo, con un *sequence number* che garantisca l'unicità del creation timestamp tra i bundle inviati da uno stesso

mittente: allora il creation timestamp insieme all'EID del mittente costituiscono l'identificativo del bundle.

Il *payload block* contiene il payload, ovvero l'ADU vera e propria, preceduto da informazioni su di esso, come la lunghezza.

Blocchi di altra tipologia, denominati *extension block*, possono essere aggiunti per trasportare ulteriori informazioni. Ad esempio un *security block* permette di applicare il *Bundle Security Protocol* (BSP) per garantire riservatezza, integrità e autenticità del bundle. [Warthman]

Infine, se non contrariamente indicato, nel suo percorso dal mittente al destinatario un bundle può essere diviso in frammenti, che sono a loro volta bundle. In tal caso il primary block è replicato in ogni frammento e riporta la posizione rispetto all'ADU originaria.

1.2 Specifiche e implementazioni del Bundle Protocol

Il protocollo di comunicazione end-to-end utilizzato all'interno del bundle layer per l'inoltro dei bundle tra nodi DTN secondo la modalità store-and-forward è il *Bundle Protocol* (BP). Le specifiche del Bundle Protocol sono definite nella [RFC5050], che descrive con precisione il formato di bundle e blocchi, la trasmissione, l'inoltro, l'elaborazione e la ricezione di bundle, l'invio e il formato degli status report, il trasferimento della custodia.

In quanto strato sovrastante il livello di trasporto e destinato tuttora soltanto a scopi specifici, il Bundle Protocol è normalmente realizzato da un processo, eventualmente demone, che esegue in modalità utente, con cui le applicazioni DTN, per mezzo di apposite API, comunicano attraverso un meccanismo di *Inter-Process Communication* (IPC).

Le principali implementazioni sono tre, ciascuna rivolta a scopi di utilizzo differenti. Di seguito ne vediamo in breve caratteristiche e peculiarità, rimandando un'analisi approfondita di IBR-DTN, di interesse per il nostro lavoro, al capitolo successivo.

DTN2 è l'implementazione di riferimento del Bundle Protocol. Il suo obiettivo è presentare fedelmente le caratteristiche dell'architettura DTN e permettere la sperimentazione, perciò alle performance sono preferite chiarezza, generalità, robustezza, estensibilità e flessibilità, senza però comprometterne la possibilità di essere impiegata nel mondo reale. È scritta prevalentemente in C++ e usa il framework Oasys, ma le API offerte per l'interfacciamento con le applicazioni sono in linguaggio C e basate sulla RPC di Sun. [DTNRGcode]

ION (Interplanetary Overlay Network) è l'implementazione realizzata dal Jet Propulsion Laboratory (JPL) della NASA. È prevedibilmente rivolta alla comunicazione spaziale, nonché alla leggerezza e all'efficienza per l'utilizzo su dispositivi embedded come veicoli spaziali e rover. È interamente scritta in C, così come le API fornite. Al contrario di DTN2 supporta completamente il Bundle Security Protocol. Include inoltre una serie di altri protocolli, quali il protocollo CGR per il routing deterministico, il protocollo IPND per il *discovery* dei nodi e AMS per la distribuzione di messaggi. Il suo difetto principale è la complessità di configurazione. [DTNRGcode]

IBR-DTN è sviluppata all'interno dell'Università Tecnica di Braunschweig. È un'implementazione efficiente, modulare e portabile, pensata specificamente per piattaforme embedded, come sensori, dispositivi mobili e altri sistemi in cui l'energia e le risorse hardware disponibili sono limitate. È scritta in C++, incluse le sue API. [Schildt]

Tutte e tre le implementazioni citate sono software libero. Anche se esse sono le principali, altre implementazioni sono in corso di sviluppo, rivolte a scenari e architetture hardware ancora differenti.

2 L'IMPLEMENTAZIONE IBR-DTN

IBR-DTN è l'implementazione del Bundle Protocol sviluppata dall'Institut für Betriebssysteme und Rechnerverbund (Istituto per i Sistemi Operativi e le Reti di Calcolatori) della Technische Universität Braunschweig (Università Tecnica di Braunschweig), in Germania. È contraddistinta da leggerezza, efficienza ed elevata portabilità per rivolgersi a piattaforme mobili e sistemi embedded per cui i costi devono essere contenuti e in cui l'energia e le risorse hardware sono limitate. In tal senso, oltre all'implementazione per i tradizionali sistemi operativi desktop, è disponibile una versione di IBR-DTN per piattaforma Android. Ulteriore obiettivo di IBR-DTN è aprire all'estensione delle proprie funzionalità in maniera semplice e non invasiva tramite l'aggiunta di moduli software. Infine è offerto il pieno supporto al Bundle Security Protocol.

La versione di IBR-DTN per i sistemi tradizionali è scritta in C++. L'interfaccia con cui le applicazioni possono contattare il demone che realizza il Bundle Protocol è basata su socket, con la possibilità di utilizzare sia socket TCP, che consentono di eseguire l'applicazione DTN su una macchina diversa rispetto a quella su cui è attivo il demone, sia socket di dominio locale, più efficienti quando si opera su una singola macchina. Sono fornite API e librerie che possono essere linkate dalle applicazioni DTN per interfacciarsi in maniera semplice con il demone IBR-DTN. Tuttavia l'utilizzo di tali API diventa complesso in ambienti diversi dal C++. [Schildt]

2.1 Guida all'installazione

Vediamo di seguito la procedura di installazione della suite di IBR-DTN su sistema operativo GNU-Linux, facendo in particolare riferimento alla distribuzione Ubuntu 14.04. Come per la maggioranza del software libero destinato a Linux, l'installazione può avvenire da repository, favorendo la semplicità e sfruttando l'automatizzazione della configurazione, o a partire dai sorgenti, necessaria se si vogliono sviluppare applicazioni utilizzando in maniera versatile le librerie fornite.

2.1.1 Installazione da repository

Per l'installazione da repository è prima di tutto necessario scegliere la versione adatta alla propria distribuzione dalla lista presente all'indirizzo [1]: nel nostro caso utilizziamo "`xUbuntu_14.04`".

[1] http://download.opensuse.org/repositories/home:/j_morgenroth/

In seguito occorre aggiungere la corrispondente chiave PGP al portachiavi di `apt` tramite il comando

```
wget -O - \
http://download.opensuse.org/repositories/home:/j_morgenroth/xUbuntu_14.04/Release.key | \
apt-key add -
```

Successivamente è necessario includere l'URL del repository nella lista dei repository, inserendo nel file `/etc/apt/sources.list` la riga

```
deb http://download.opensuse.org/repositories/home:/j_morgenroth/xUbuntu_14.04 ./
```

A questo punto è sufficiente aggiornare la lista dei pacchetti con

```
apt-get update
```

e installare IBR-DTN con

```
apt-get install ibrdtnd ibrdtn-tools
```

Il demone è denominato `dtnd` (infelice caso di omonimia con il demone di DTN2) ed è installato nella directory `/usr/sbin`, mentre le librerie sono poste in `/usr/lib`.

2.1.1.1 Disabilitare l'avvio automatico del demone

L'installazione da repository predispone l'avvio automatico del demone allo startup, causando conflitti con l'omonimo demone di DTN2 oppure nel caso in cui si voglia avviare `dtnd` manualmente.

Per terminare il demone è sufficiente il comando

```
dtnd -k
```

ma per disabilitarne definitivamente l'avvio è necessario utilizzare gli strumenti di configurazione del sistema operativo. In Ubuntu 14.04, ad esempio, si procede eliminando il link simbolico `/etc/rc2.d/S20ibrdtnd`, ed eventualmente anche `/etc/rc3.d/S20ibrdtnd`, `/etc/rc4.d/S20ibrdtnd` e `/etc/rc5.d/S20ibrdtnd`, rapidamente con il comando

```
rm /etc/rc?.d/S20ibrdtnd
```

In alternativa si può agire brutalmente ma efficacemente sul file `/etc/init.d/ibrdtnd`, che si occupa effettivamente dell'avvio del demone, inserendo

```
exit
```

all'inizio.

2.1.2 Installazione da sorgenti

L'installazione a partire dai sorgenti ha inizio con il download degli stessi. Posizionarsi in una directory di lavoro ed eseguire

```
wget https://www.ibr.cs.tu-bs.de/projects/ibr-dtn/releases/ibrcommon-1.0.1.tar.gz
wget https://www.ibr.cs.tu-bs.de/projects/ibr-dtn/releases/ibrdtn-1.0.1.tar.gz
```

```
wget https://www.ibr.cs.tu-bs.de/projects/ibr-dtn/releases/ibrdtnd-1.0.1.tar.gz
wget https://www.ibr.cs.tu-bs.de/projects/ibr-dtn/releases/ibrdtn-tools-1.0.1.tar.gz
```

Estrarre i quattro archivi tramite

```
tar -xf ibrcommon-1.0.1.tar.gz
tar -xf ibrdtn-1.0.1.tar.gz
tar -xf ibrdtnd-1.0.1.tar.gz
tar -xf ibrdtn-tools-1.0.1.tar.gz
```

Assicurarsi che `pkg-config` sia installato. In caso contrario, installarlo con

```
apt-get install pkg-config
```

Installare `libssl-dev`:

```
apt-get install libssl-dev
```

Entrare nella directory `ibrcommon-1.0.1` ed eseguire

```
./configure --with-openssl
make
make install
```

Passando ulteriori parametri a `configure` è possibile selezionare i componenti da includere, ma l'installazione più basilare, a noi sufficiente, in teoria non ne richiede alcuno. In realtà, a causa di un bug (si veda [2]) risolto da tempo ma non ancora incluso nei sorgenti in versione per il rilascio, si rende necessario aggiungere il supporto al Bundle Security Protocol con l'opzione `--with-openssl`.

Posizionarsi ora nelle directory `ibrdtn-1.0.1`, `ibrdtnd-1.0.1` e `ibrdtn-tools-1.0.1`, avendo cura di seguire questo ordine, e lanciare

```
./configure
make
make install
```

La compilazione di `ibrdtnd-1.0.1` può richiedere diversi minuti.

Questa volta il demone è installato, per impostazione predefinita, nella directory `/usr/local/sbin`, mentre le librerie in `/usr/local/lib`.

Non è finita. Occorre aggiungere la directory `/usr/local/lib` all'elenco dei percorsi consultati nella ricerca di librerie condivise. Inserire nel file `/etc/ld.so.conf` la riga

```
/usr/local/lib
```

[2] https://github.com/ibrdtn/ibrdtn/commit/576212fe41036c29246aaa64a2cafe314a97e2dd

ed eseguire

```
ldconfig
```

L'installazione tramite sorgenti ha il vantaggio di non predisporre l'avvio automatico del demone.

2.2 Avvio e configurazione

Supponendo di aver disabilitato l'esecuzione automatica allo startup o di aver effettuato l'installazione a partire dai sorgenti, l'avvio del demone IBR-DTN si basa sul comando `dtnd`.

La modalità di utilizzo più semplice è nella forma

```
dtnd -i interface [-v]
```

dove l'opzione `-i` specifica l'interfaccia di rete a cui collegarsi per uscire dal nodo, mentre `-v` abilita la stampa di messaggi di log. Ad esempio

```
dtnd -i eth0 -v
```

Eseguendo

```
dtnd -h
```

è possibile visualizzare le altre opzioni disponibili.

Con questa configurazione minimale, il demone IBR-DTN rileva i demoni in esecuzione su altre macchine direttamente raggiungibili mediante un modulo di discovery IPND (attualmente di versione diversa rispetto a quello di ION), che utilizza il protocollo UDP con un indirizzo IP multicast, di default 224.0.0.142, sulla porta 4551. Inoltre, per impostazione predefinita, l'EID locale segue lo schema DTN e ha come SSP il nome della macchina, ovvero è nella forma `dtn://`*`hostname`*.

2.2.1 File di configurazione

Naturalmente è possibile specificare innumerevoli opzioni per modificare il comportamento predefinito del demone. La modalità è simile a quella prevista da DTN2, ovvero è necessario passare a `dtnd` un file di configurazione utilizzando l'opzione `-c`, cioè eseguendo

```
dtnd -c config_file
```

Se l'installazione è avvenuta da repository, un esempio di file di configurazione completo, lo stesso utilizzato dal demone avviato automaticamente, è quello con percorso `/etc/ibrdtn/ibrdtnd.conf`. L'installazione da sorgenti assegna invece allo stesso file il nome `/usr/local/etc/ibrdtnd.conf`. In ogni caso il file è

reperibile all'indirizzo [3]. Di seguito ne riportiamo i passaggi più interessanti insieme ai rispettivi commenti e a brevi spiegazioni.

```
# the local eid of the dtn node
# default is the hostname
local_uri = dtn://node.dtn
```

permette di modificare l'EID locale.

```
# specifies an additional logfile
logfile = /var/log/ibrdtn/ibrdtn.log
```

specifica un file di log.

```
# define the port for the API to bind on
api_port = 4550
```

modifica la porta su cui la socket per la comunicazione con il demone tramite API è in ascolto.

```
# define a folder for temporary storage of bundles
# if this is not defined bundles will processed in memory
blob_path = /tmp
```

specifica una directory in cui mantenere temporaneamente i bundle mentre vengono processati.

```
# define a folder for persistent storage of bundles
# if this is not defined bundles will stored in memory only
storage_path = /var/spool/ibrdtn/bundles
```

specifica la directory in cui i bundle vengono memorizzati in attesa di poter essere inoltrati.

```
# a list (seperated by spaces) of names for convergence layer instances
net_interfaces = lan0 lan1
#
# configuration for a convergence layer named lan0
net_lan0_type = tcp        # we want to use TCP as protocol
net_lan0_interface = eth0  # listen on interface eth0
net_lan0_port = 4556       # listen port 4556 (default)
#
# configuration for a convergence layer named lan1
net_lan1_type = udp        # we want to use UDP as protocol
net_lan1_interface = eth0  # listen on interface eth0
net_lan1_port = 4556       # listen port 4556 (default)
```

elenca i convergence layer e per ognuno il protocollo, l'interfaccia di rete e la porta.

[3] https://github.com/ibrdtn/ibrdtn/blob/master/ibrdtn/daemon/etc/ibrdtnd.conf

```
# routing strategy
# values: default | epidemic | flooding | prophet | none
#
# With "default" the daemon only delivers bundles to neighbors and static
# available nodes. The alternative module "epidemic" spreads all bundles to
# all available neighbors. Flooding works like epidemic, but does not send
# a node's summary vector to neighbors. Prophet forwards based on the
# probability to encounter other nodes (see RFC 6693).
routing = prophet
```

seleziona l'algoritmo di instradamento da utilizzare. Il commento fornisce già un'idea delle loro caratteristiche.

```
# forward bundles to other nodes (yes/no)
routing_forwarding = yes
```

specifica se il nodo deve occuparsi di inoltrare bundle, per impostazione di default, o soltanto di inviarne e riceverne.

```
# static routing rules
# - a rule is a regex pattern
# - format is <target-scheme> <routing-node>
#
# route all bundles for "dtn://*.moon.dtn/*" to dtn://router.dtn
route1 = ^dtn://[[:alpha:]].moon.dtn/[[:alpha:]] dtn://router.dtn
```

elenca regole di routing statiche basate sulla corrispondenza tra l'EID di destinazione e un'espressione regolare.

```
# static connections
# to configure static connections it is important to begin
# with "static1_" and count up ("static2_", "static3_", ...)
#
### node-five.dtn ###
static1_address = 10.0.0.5   # the node has the address 10.0.0.5
static1_port = 4556          # accept bundles on port 4556
static1_uri = dtn://node-five.dtn  # eid of the node is "dtn://node-five.dtn"
static1_proto = tcp          # reachable over TCP
static1_immediately = yes    # connect immediately to this node
static1_global = yes  # this node is only reachable with internet access
```

elenca connessioni statiche con altri nodi DTN. Se si utilizzano soltanto connessioni statiche è possibile disabilitare il modulo di discovery avviando `dtnd` con l'opzione `--nodiscovery`.

2.3 Strumenti di base

Oltre al demone vero e proprio, la suite IBR-DTN fornisce anche una serie di strumenti a linea di comando che permettono di effettuare comunicazioni di base

utilizzando il Bundle Protocol, installati nella stessa directory di `dtnd`. Purtroppo anch'essi, come il demone, sono omonimi dei corrispondenti programmi DTN2, per cui occorre prestare attenzione nel caso entrambe le implementazioni siano presenti sulla macchina; se non altro, le directory di installazione predefinite differiscono tra le due implementazioni, evitando sovrascritture involontarie.

Presentiamo di seguito i tre strumenti più semplici e utili.

`dtnrecv` riceve un bundle e ne scrive il contenuto sullo standard output o sul file specificato. La sintassi di base è

```
dtnrecv --name appname [--file file]
```

Ad esempio

```
dtnrecv --name receiver --file file.txt
```

`dtnsend` consente l'invio di un file, o dello standard input se nessun file è specificato, a un EID. La sintassi di base è

```
dtnsend destination [file]
```

Ad esempio

```
dtnsend dtn://host2/receiver file.txt
```

`dtnping` invia ripetutamente bundle a un EID e per ognuno attende un bundle di risposta con lo stesso payload, mostrando il tempo di trascorso. La sintassi di base è

```
dtnping destination
```

Per il funzionamento di `dtnping` è necessaria un'applicazione DTN che realizzi un servizio di echo: il demone IBR-DTN integra già tale servizio sotto il demux token `echo`, e analogamente per il demone di DTN2 ma con il demux token `ping`. Allora avremo ad esempio

```
dtnping dtn://host2/echo
```

per un destinatario IBR-DTN e

```
dtnping dtn://host2/ping
```

per un destinatario DTN2.

Ognuna di queste applicazioni può essere invocata con l'opzione `-h` o `--help` per visualizzare tutti i possibili parametri e le modalità di utilizzo. [IBR_GH]

2.4 API

IBR-DTN offre due diversi protocolli per la comunicazione con il demone da parte di un'applicazione. Entrambi i protocolli sono disponibili tramite una socket in ascolto per conto del demone; di default si tratta di una socket TCP sulla porta 4550,

ma è anche possibile configurare l'utilizzo di una socket di dominio locale, come una Unix Domain Socket.

2.4.1 Cenni sulle API testuali

Un primo protocollo è puramente testuale e, basato su direttive costituite ciascuna da una o più parole e separate da ritorni a capo, ricorda per modalità di utilizzo la linea di comando. Ad ogni comando inviato dal client corrisponde una risposta del server, che comprende un codice che notifica il successo o il fallimento dell'operazione seguito eventualmente da un invito al client ad immettere ulteriori dati o dalle informazioni richieste restituite dal server.

In quanto testuale, tale protocollo può essere testato con strumenti di comunicazione remota quali `netcat` o `telnet`, mentre più impegnativo è l'utilizzo all'interno delle applicazioni, perché non è fornita una libreria che generi i comandi testuali a partire da chiamate a funzione. Inoltre è alcuni ordini di grandezza più lento rispetto al protocollo binario che vedremo tra poco. Per queste sue caratteristiche, il protocollo testuale è adatto soltanto a fini esplorativi e per uso in ambienti che non supportano le API binarie. [IBR_API]

Non approfondiamo il funzionamento di questo protocollo e i numerosi comandi disponibili perché non interessanti ai fini del nostro lavoro.

2.4.2 API C++

L'altro protocollo segue invece un formato binario. La documentazione relativa è assente, ma ciò è poco importante perché nello sviluppo di applicazioni DTN si farà uso, se possibile, dalle API scritte in C++ che lo implementano.

Le API sono incluse nell'installazione e corrispondono, se questa è avvenuta da sorgenti, al componente individuato da `ibrdtn-1.0.1`. In quanto API C++, sono rilevanti i file di intestazione (.h), da includere nei sorgenti dell'applicazione, e le librerie (.a) verso cui effettuare il linking al momento della creazione del file eseguibile. Le librerie da linkare sono in particolare `libibrcommon.a` e `libibrdtn.a`, ma potrebbe essere richiesta anche la libreria di compressione `libz.a`, normalmente già disponibile dal momento dell'installazione del sistema operativo.

Vediamo di seguito i componenti più importanti di queste API, elencando, senza pretesa di completezza, i metodi e i campi di maggiore interesse.

2.4.2.1 La classe Client

La classe `dtn::api::Client` è definita in `ibrdtn-1.0.1/ibrdtn/api/Client.h` e implementata in `ibrdtn-1.0.1/ibrdtn/api/Client.cpp`. Essa rappresenta il

punto di accesso alle API e permette di comunicare con il demone per inviare e ricevere bundle.

Il costruttore

```
Client(const std::string &app, ibrcommon::socketstream &stream)
```

costruisce un oggetto `Client` dati un demux token, che verrà aggiunto all'EID locale associato al demone, e uno stream di byte, attraverso il quale avviene la comunicazione con la socket esposta dal demone.

Il costruttore

```
Client(const std::string &app, const dtn::data::EID &group,
       ibrcommon::socketstream &stream)
```

aggiunge al precedente la possibilità di ricevere anche bundle destinati a un EID multicast.

Il metodo

```
void connect()
```

avvia la comunicazione con il demone eseguendo l'handshake definito dal protocollo binario. Si noti che la connessione era già stata aperta ancor prima della costruzione dell'oggetto `ibrcommon::socketstream` passato al costruttore.

Il metodo

```
void close()
```

termina la comunicazione con il demone inviando un apposito messaggio definito dal protocollo. Si noti che questo metodo non si occupa della chiusura della connessione, che deve invece avvenire separatamente agendo sull'istanza di `ibrcommon::socketstream` passata al costruttore.

L'operatore

```
void operator<<(const dtn::data::Bundle &b)
```

passa un bundle al demone per l'invio. Indipendentemente dai valori riportati nell'oggetto `dtn::data::Bundle`, il creation timestamp (istante di creazione e sequence number) viene riassegnato dal demone al momento dell'invio. Purtroppo il nuovo creation timestamp non è restituito al chiamante, che non ha quindi modo di conoscere l'identità del bundle inviato: vedremo in seguito che ciò sarà fonte di alcuni problemi. Tale possibilità è invece offerta dalle API testuali.

Il metodo

```
dtn::data::Bundle getBundle(
        const dtn::data::Timeout timeout = 0)
```

riceve un bundle in maniera sincrona, bloccando il thread corrente fino alla ricezione o fino allo scadere del timeout, nel qual caso viene sollevata l'eccezione `dtn::api::ConnectionTimeoutException`. Il timeout è espresso in secondi e un valore nullo implica un'attesa infinita: ciò significa che il minimo timeout impostabile è di un secondo, il che può non essere ideale in alcune circostanze.

Il tipo `dtn::data::Timeout`, definito dalle API stesse insieme a `dtn::data::Length` e `dtn::data::Size` in `ibrdtn-1.0.1/ibrdtn/data/Number.h`, non è altro che, come questi, un nome alternativo più specifico per l'intero senza segno `size_t`.

Notiamo come la classe `dtn::api::Client` non esponga alcun metodo per apprendere l'EID amministrativo del nodo locale. Anche questo sarà un problema che in seguito dovremo gestire e che invece non si presenta facendo uso delle API testuali.

2.4.2.2 *Le classi socketstream, tcpsocket e vaddress*

Come accennato, la classe `ibrcommon::socketstream`, definita in `ibrcommon-1.0.1/ibrcommon/net/socketstream.h` e implementata in `ibrcommon-1.0.1/ibrcommon/net/socketstream.cpp`, mantiene uno stream bidirezionale di byte inviati e ricevuti tramite una socket.

Il suo costruttore

```
socketstream(clientsocket *sock)
```

accetta una socket come puntatore a un'istanza della classe `ibrcommon::clientsocket` o di una sua sottoclasse, quale `ibrcommon::tcpsocket`.

Il metodo di `ibrcommon::socketstream`

```
void close()
```

chiude lo stream e la socket sottostante.

La classe `ibrcommon::tcpsocket`, definita in `ibrcommon-1.0.1/ibrcommon/net/socket.h` e implementata in `ibrcommon-1.0.1/ibrcommon/net/socket.cpp`, realizza ovviamente una socket TCP.

Il suo costruttore

```
tcpsocket(const vaddress &destination)
```

crea una socket e apre la connessione verso l'indirizzo di rete passatogli.

La classe `ibrcommon::vaddress`, definita in `ibrcommon-1.0.1/ibrcommon/net/vaddress.h` e implementata in `ibrcommon-1.0.1/ibrcommon/net/vaddress.cpp`, rappresenta infine in maniera flessibile un generico indirizzo che può essere associato a una socket.

Uno dei suoi costruttori è

```
vaddress(const std::string &address, const int port)
```

che crea un indirizzo di rete a partire da un host, ad esempio un indirizzo IP, e da una porta.

Dunque, per costruire un oggetto `Client` si procede nel verso opposto rispetto a quello seguito nella presentazione delle quattro classi viste finora: si crea un indirizzo di rete `vaddress`, poi una socket `tcpsocket` connessa a tale indirizzo, poi uno stream `socketstream` che incapsula la socket, e infine il `Client`, cui è passato lo stream.

2.4.2.3 La classe EID

La classe `dtn::data::EID` è definita in `ibrdtn-1.0.1/ibrdtn/data/EID.h` e implementata in `ibrdtn-1.0.1/ibrdtn/data/EID.cpp`. Essa rappresenta evidentemente un Endpoint Identifier e permette di ottenerne e impostarne le varie parti.

Il costruttore

```
EID()
```

istanzia un EID vuoto.

Il costruttore

```
EID(const std::string &scheme, const std::string &ssp)
```

istanzia un EID dati lo schema e la SSP.

Il costruttore

```
EID(const std::string &value)
```

istanzia un EID dello schema appropriato a partire da una stringa.

Il costruttore

```
EID(const dtn::data::Number &node,
    const dtn::data::Number &application)
```

istanzia un EID con schema CBHE a partire dal numero associato al nodo DTN e dal numero associato all'applicazione.

`dtn::data::Number` è un tipo di dato numerico definito dalle stesse API, insieme a `dtn::data::Float`, `dtn::data::Integer` e `dtn::data::Timestamp`, in `ibrdtn-1.0.1/ibrdtn/data/Number.h`.

Il metodo

```
std::string getString() const
```

restituisce l'EID sotto forma di stringa.

Il metodo

```
EID getNode() const
```

restituisce l'EID a cui è stato rimosso il demux token.

Il metodo

```
const std::string getScheme() const
```

restituisce lo schema dell'EID.

Il metodo

```
const std::string getSSP() const
```

fornisce la scheme-specific part dell'EID.

Il metodo

```
std::string getHost() const
```

restituisce la porzione della SSP associata al nodo DTN.

Il metodo

```
std::string getApplication() const
```

restituisce la porzione della SSP associata all'applicazione, cioè il demux token.

I metodi

```
void setApplication(const std::string &app)
```

e

```
void setApplication(const dtn::data::Number &app)
```

consentono di modificare il demux token.

2.4.2.4 La classe Bundle

La classe `dtn::data::Bundle` è definita in `ibrdtn-1.0.1/ibrdtn/data/Bundle.h` e implementata in `ibrdtn-1.0.1/ibrdtn/data/Bundle.cpp`. Essa rappresenta un bundle, ricevuto o da inviare, e consente di gestirne il primary block e gli altri blocchi.

`dtn::data::Bundle` eredita pubblicamente dalla classe `dtn::data::PrimaryBlock`, definita in `ibrdtn-1.0.1/ibrdtn/data/PrimaryBlock.h` e implementata in `ibrdtn-1.0.1/ibrdtn/data/PrimaryBlock.cpp`, che a sua volta eredita pubblicamente dalla classe `dtn::data::BundleID`, definita in `ibrdtn-1.0.1/ibrdtn/data/BundleID.h` e implementata in `ibrdtn-1.0.1/ibrdtn/data/BundleID.cpp`. Si noti che queste due relazioni di ereditarietà sono molto discutibili dal punto di vista dell'ingegneria del software, perché viene utilizzata l'ereditarietà di estensione per esprimere un legame di contenimento anziché una relazione "is-a".

I campi

```
dtn::data::EID source
dtn::data::EID destination
dtn::data::EID reportto
dtn::data::EID custodian
```

memorizzano rispettivamente l'EID del mittente, l'EID del destinatario, il report-to EID e l'EID dell'eventuale custode attuale.

I campi

```
dtn::data::Timestamp timestamp
```

e

```
dtn::data::Number sequencenumber
```

rappresentano insieme il creation timestamp.

Il campo

```
dtn::data::Number lifetime
```

memorizza il massimo tempo di vita del bundle.

I metodi

```
bool isFragment() const
```

e

```
void setFragment(bool val)
```

consentono rispettivamente di conoscere se un bundle è un frammento e di impostare il bundle come tale.

Il campo

```
dtn::data::Number fragmentoffset
```

conserva la posizione del frammento rispetto all'ADU originaria.

I metodi

```
PRIORITY getPriority() const
```

e

```
void setPriority(PRIORITY p)
```

rispettivamente restituiscono e modificano la priorità del bundle. `PRIORITY` è l'enumeratore

```
enum PRIORITY { PRIO_LOW = 0, PRIO_MEDIUM = 1, PRIO_HIGH = 2 }
```

I metodi

```
bool get(FLAGS flag) const
```

e

```
void set(FLAGS flag, bool value)
```

permettono di leggere e impostare le delivery option, il trasferimento con custodia e altre flag definite dal Bundle Protocol che influiscono sull'elaborazione e l'inoltro del bundle. `FLAGS` è l'enumeratore che dichiara tali flag.

Il metodo

```
dtn::data::PayloadBlock& push_back(
        ibrcommon::BLOB::Reference &ref)
```

aggiunge il payload al bundle.

La classe `dtn::data::PayloadBlock`, definita in `ibrdtn-1.0.1/ibrdtn/data/PayloadBlock.h` e implementata in `ibrdtn-1.0.1/ibrdtn/data/PayloadBlock.cpp`, eredita da `dtn::data::Block` e rappresenta ovviamente un payload block. Di interesse sono soltanto i metodi

```
ibrcommon::BLOB::Reference getBLOB() const
```

con cui ottenere il contenuto, e

```
Length getLength() const
```

che restituisce la sua dimensione.

Il metodo generico

```
template<class T> const T& find() const
```

cerca un blocco della classe `T` tra i blocchi di un bundle. In particolare, invocato su un'istanza di `Bundle` come `find<dtn::data::PayloadBlock>().getBLOB()`, permette di ottenere il payload block e da questo il suo contenuto.

Il metodo

```
dtn::data::Length getPayloadLength() const
```

restituisce la dimensione in byte del payload.

Sono poi presenti ulteriori metodi, che omettiamo, che consentono l'inserimento, la lettura e la rimozione di extension block, rappresentati dalla classe `dtn::data::ExtensionBlock` che, come `dtn::data::PayloadBlock`, eredita da `dtn::data::Block`.

Infine il costruttore

```
Bundle()
```

costruisce un'istanza di `Bundle` contenente soltanto il primary block e con solo i campi `timestamp` e `sequencenumber` inizializzati con valori realistici. Ricordiamo che al momento dell'invio tali campi sono comunque sovrascritti.

2.4.2.5 La classe BLOB

La classe `ibrcommon::BLOB` è definita in `ibrdtn-1.0.1/ibrdtn/data/BLOB.h` e implementata in `ibrdtn-1.0.1/ibrdtn/data/BLOB.cpp`. Essa rappresenta un BLOB, un *Binary Large Object*, letteralmente un oggetto binario di grandi dimensioni. Il suo utilizzo, principalmente per mantenere il payload di un bundle, è mediato dalla classe `ibrcommon::BLOB::Reference`.

Il metodo

```
BLOB::iostream iostream()
```

restituisce l'istanza di una classe che si comporta esattamente come un puntatore all'oggetto `std::iostream` con cui leggere e scrivere sul BLOB nelle modalità consuete per il trattamento degli stream in C++.

Il metodo statico

```
static ibrcommon::BLOB::Reference create()
```

crea un BLOB vuoto.

Il metodo statico

```
static ibrcommon::BLOB::Reference open(const ibrcommon::File &f)
```

apre un file come un BLOB in sola lettura. Può essere invocato anche passando direttamente una stringa contenente il nome del file, sfruttando i meccanismi di conversione di tipo del C++.

3 IL SOFTWARE DTNPERF_3

DTNperf è uno strumento open-source nato per la valutazione del *goodput* in un'architettura DTN, sul modello del software Iperf utilizzato per testare le prestazioni di una rete nel veicolare dati tramite i protocolli TCP e UDP. Ricordiamo che per goodput si intende la quantità media di dati confermati nell'unità di tempo a livello applicativo, cioè "utili", a differenza del *throughput* che considera il totale dei dati grezzi in transito su un canale, incluse ad esempio le intestazioni dei protocolli di comunicazione e le ritrasmissioni.

La sua terza versione, denominata DTNperf_3, che ha esteso il supporto dal solo DTN2 anche ad ION, presenta un'architettura client-server, tra i quali scorre il flusso dei dati portati dai bundle, a cui è affiancata una terza entità, il monitor, che, in linea con quanto previsto dalle specifiche dell'architettura DTN, si occupa di raccogliere informazioni dagli status report generati dal Bundle Protocol, con cui creare, per ogni sessione di test, un file di log che ne permetta un'analisi approfondita. Questi file di log sono particolarmente importanti nelle reti DTN caratterizzate da intermittenza dei collegamenti, perché per esse la nozione di goodput tende a perdere di significato all'aumentare della frequenza e della durata delle interruzioni. Inoltre essi permettono di effettuare un'analisi "microscopica" che permette di comprendere al meglio i fattori che determinano i risultati macroscopici come il goodput. [Caini]

3.1 Descrizione generale

In quanto utilizzato per la valutazione del goodput e la raccolta degli status report, DTNperf è uno strumento di livello applicativo, che si interfaccia con il demone che realizza il Bundle Protocol. Le entità coinvolte in una sessione di test sono tre, cui corrispondono tre modalità operative di DTNperf: il client crea e invia bundle, il server li riceve ed eventualmente risponde al client, il monitor raccoglie gli status report e genera un file di log. È interessante notare che, sebbene progettate per lavorare insieme, le tre entità possono anche essere usate separatamente, in maniera autonoma: il monitor, ad esempio, può essere impiegato per raccogliere gli status report anche in esperimenti che non coinvolgono il client o il server.

3.1.1 Il client

Il ruolo del client è quello di generare bundle e di inviarli al server: ciò può avvenire secondo tre modalità di trasmissione. In modalità "time" una serie di bundle dal payload privo di significato sono inviati (o meglio, passati al Bundle Protocol) fino allo scadere di un tempo impostato. La modalità "data" è simile alla precedente, ma

l'invio procede fino al raggiungimento di una certa quantità di dati. In modalità "file" è trasmesso un file anziché dati fittizi; si noti che, se la dimensione massima stabilita per i bundle è inferiore alla dimensione del file, il file sarà suddiviso dal client su più bundle.

DTNperf offre due criteri di controllo di congestione, uno *window-based* (basato su una finestra scorrevole) e uno *rate-based*. Il primo prevede, in ogni istante, un tetto al numero di bundle che sono stati inviati ma per cui non si è ancora ricevuta la relativa conferma (*acknowledgment*) da parte del server. Il secondo provoca la generazione di una quantità di traffico costante, espressa in bundle per secondo o in bit per secondo; si noti che nella modalità rate-based non è necessaria alcuna conferma di ricezione da parte del server.

Oltre alla modalità di trasmissione, al controllo di congestione e naturalmente all'EID del server, il client prevede una serie di altre opzioni con cui configurare la sessione di test. Tra queste sono degne di nota: l'EID del monitor, la dimensione del payload dei bundle, la richiesta di trasferimento con custodia, la vita massima dei bundle, la loro priorità, le opzioni dei bundle riguardanti la generazione di specifici status report (gli status report relativi alla cancellazione di un bundle e alla ricezione da parte del destinatario sono sempre richiesti), la possibilità di frammentare i bundle.

3.1.2 Il server

Il server resta in attesa di bundle e, al momento della ricezione, intraprende eventuali azioni determinate dalle opzioni scelte per il client e portate dal payload del bundle stesso in un header applicativo introdotto da DTNperf. In particolare, se (e solo se) il client utilizza il controllo di congestione basato su finestra, il server risponde con la conferma della ricezione nella forma di un apposito bundle ACK, anch'esso definito da DTNperf. Se d'altra parte il client sta inviando un file, il server si occupa di riassemblarlo, gestendo anche la ricezione disordinata dei bundle su cui è stato distribuito. Non sono tuttavia previste ritrasmissioni a livello di DTNperf.

Uno stesso server può gestire la ricezione di bundle da molteplici client, e in particolare il riassemblaggio di più file contemporaneamente, mantenendo per ognuno una sessione di trasferimento che rimane attiva fino a quando il file è stato trasmesso per intero (o fino alla scadenza di un timeout, per evitare che una sessione possa restare incompleta per un tempo illimitato).

Tra le opzioni per il server annoveriamo il lifetime e la priorità degli acknowledgment.

3.1.3 Il monitor

Il monitor ha il compito di ricevere gli status report generati dal Bundle Protocol e di utilizzarli per produrre file di log, uno per ogni client. I log vengono salvati in formato .csv (*comma-separated values*) e permettono una micro-analisi dei test effettuati. Similarmente al server, il monitor può gestire client multipli in contemporanea, per ognuno dei quali viene aperta una sessione dedicata. È comunque possibile, su richiesta, raccogliere tutti gli status report in un unico file, nonché far stampare a video una riga per ogni status report ricevuto, così da riuscire (se il traffico è limitato) a monitorare un esperimento in tempo reale.

Oltre agli status report, il monitor supporta la ricezione di due bundle speciali, STOP e FORCE STOP, definiti da DTNperf come gli ACK, con cui viene segnalato il completamento di una sessione e che causano la terminazione della scrittura del file di log (ancora analogamente al server, è previsto un timeout oltre il quale la sessione viene comunque chiusa, in caso di perdita del bundle STOP). La differenza tra i due bundle risiede nella ragione che ha comportato la fine del test: STOP è inviato quando il client ha concluso l'invio di tutti i bundle previsti e ricevuto gli eventuali ACK relativi, FORCE STOP è generato se il client viene interrotto dal segnale SIGINT, ad esempio tramite Ctrl-C da tastiera.

In realtà, nello scenario più semplice non è richiesto l'avvio esplicito di un monitor. Se infatti, al momento del lancio del client, non è indicato un monitor esterno già in esecuzione, è il client stesso ad avviare un processo monitor figlio, dedicato alla produzione del file di log relativo a quella sola specifica sessione. Il comportamento di tale monitor è invariato rispetto a quanto visto in precedenza, con l'eccezione che, in quanto dedicato a un singolo client, alla terminazione della sessione segue la propria terminazione. [Rodolfi]

3.2 Guida all'uso

Client, server e monitor sono avviati dallo stesso eseguibile, rispettivamente specificando una tra le opzioni `--client`, `--server` o `--monitor`, seguita dalle opzioni specifiche per la modalità scelta.

3.2.1 Il client

La sintassi per l'avvio del client è

```
dtnperf --client required_inputs [options]
```

Gli input richiesti sono:

- l’EID del server, tramite l’opzione
  `-d` *`server_EID`*

  oppure
  `--destination` *`server_EID`*
- una modalità di trasmissione, tra
  - “time”, con l’opzione
    `-T` *`seconds`*

    oppure
    `--time` *`seconds`*
  - “data”, tramite
    `-D` *`bytes`*

    oppure
    `--data` *`bytes`*

    dove i byte sono indicati da un numero seguito, senza spazi, da uno tra i suffissi `B` (byte), `k` (kilobyte) o `M` (megabyte);
  - “file”, con
    `-F` *`filename`*

    oppure
    `--file` *`filename`*
- il controllo di congestione, tra
  - window-based, con l’opzione
    `-W` *`size`*

    oppure
    `--window` *`size`*
  - rate-based, con
    `-R` *`rate`*

    oppure
    `--rate` *`rate`*

    dove *`rate`* è un numero seguito, senza spazi, da uno tra i suffissi `b` (bundle), `k` (kbit/s) o `M` (Mbit/s);

Una lista parziale delle opzioni facoltative è data dalla tabella seguente.

`-m` *`monitor_EID`* `--monitor` *`monitor_EID`*	EID del monitor esterno. Di default viene avviato un monitor interno dedicato.
`-C` `--custody`	Richiede il trasferimento con custodia.

-P *size* --payload *size*	Dimensione in byte del payload dei bundle, indicata da un numero seguito, senza spazi, da uno tra i suffissi `B` (byte), `k` (kilobyte) o `M` (megabyte). Di default è di 50 kB.
-N --nofragment	Richiede di non frammentare i bundle.
-l *seconds* --lifetime *seconds*	Il tempo di vita dei bundle. Di default è di 60 secondi.
-p *value* --priority *value*	La priorità dei bundle, tra `bulk`, `normal`, `expedited` e `reserved`. Di default è normale.
-r --received	Richiede l'invio di uno status report ogni volta che il bundle giunge a un nodo DTN.
-f --forwarded	Richiede l'invio di uno status report ogni volta che il bundle è inoltrato da un nodo DTN al successivo.
-v --verbose	Stampa alcuni messaggi informativi su standard output.
--debug[=*level*]	Stampa messaggi di debug su standard output. Il livello può essere `1` o `2` (2 se non specificato).

3.2.2 Il server

La sintassi per l'avvio del server è

```
dtnperf --server [options]
```

Come si vede, non ci sono opzioni obbligatorie. Alcune tra le opzioni ammesse sono elencate nella tabella che segue.

-l *seconds* --lifetime *seconds*	Il tempo di vita degli ACK. Di default è di 60 secondi.
-p *value* --priority *value*	La priorità degli ACK, tra `bulk`, `normal`, `expedited` e `reserved`. Di default è normale.
-v --verbose	Stampa alcuni messaggi informativi su standard output.
--debug[=*level*]	Stampa messaggi di debug su standard output. Il livello può essere `1` o `2` (2 se non specificato).

3.2.3 Il monitor

La sintassi per l'avvio del monitor esterno è

```
dtnperf --monitor [options]
```

Anche in questo caso non ci sono opzioni obbligatorie. In tabella sono elencate alcune opzioni ammesse.

`--rt-print[=`*filename*`]`	Stampa in tempo reale informazioni sugli status report, su standard output o sul file specificato.
`--oneCSVonly`	Genera un unico file di log per tutti i client.
`-v` `--verbose`	Stampa alcuni messaggi informativi su standard output.
`--debug[=`*level*`]`	Stampa messaggi di debug su standard output. Il livello può essere `1` o `2` (2 se non specificato).

In ogni caso, facendo seguire `--client`, `--server` o `--monitor` dall'opzione `-h` o `--help` si ottiene l'elenco di tutti i parametri e le opzioni disponibili per la modalità scelta.

3.2.4 Esempi

Seguono alcuni esempi di utilizzo delle tre modalità di DTNperf_3, al fine di chiarire la sintassi delle opzioni.

Avviamo il server:

```
dtnperf --server
```

Avviamo il server stampando tutti i messaggi diagnostici, riducendo il tempo di vita degli ACK e aumentandone la priorità:

```
dtnperf --server -l 10 -p expedited -v --debug=2
```

Avviamo un monitor esterno stampando tutti i messaggi diagnostici:

```
dtnperf --monitor -v --debug=2
```

Effettuiamo un ping per 10 secondi verso un server DTNperf con EID nello schema DTN:

```
dtnperf --client -d dtn://vm2.dtn -T 10 -W 1 -v --debug=2
```

Tracciamo il percorso di dieci bundle di 1 MB, inviati a distanza di cinque secondi e trasferiti con custodia verso un server con EID nello schema CBHE, su un monitor con EID nello stesso schema:

```
dtnperf --client -d ipn:2.2000 -D 10M -P 1M -R 0.2b -C -r -f \
        -m ipn:3.1000 -v --debug=2
```

Notiamo che con lo schema CBHE è necessario specificare anche il demux token, che è sempre 2000 per un server e 1000 per un monitor esterno.

Trasferiamo un file in piccoli bundle non frammentabili, aumentando la finestra di congestione, senza mostrare messaggi informativi:

```
dtnperf --client -d dtn://vm2.dtn -F picture.jpg -P 10k -W 10 -N
```

3.3 L’Abstraction Layer

Come abbiamo già avuto modo di descrivere nel paragrafo 1.2, l’invio e la ricezione di bundle da parte di un’applicazione DTN richiede la comunicazione con il demone fornito dall’implementazione del Bundle Protocol in uso, secondo una qualche forma di IPC. Tale interazione non segue una modalità standard, ma è realizzata utilizzando le API fornite dalla specifica implementazione, estremamente differenti da un’implementazione all’altra. Da ciò segue che un’applicazione che faccia uso del Bundle Protocol sia fortemente legata all’implementazione sottostante scelta al momento dello sviluppo, e che un successivo ripensamento riguardo tale scelta richieda la riscrittura di una discreta quantità di codice, per adattarlo alle nuove API.

Nel caso specifico di DTNperf, abbiamo accennato che una delle novità di maggior rilievo della terza versione è il supporto sia a DTN2 che ad ION, anche al fine di metterne a confronto le prestazioni e di studiarne l’interoperabilità. Ma un utilizzo diretto delle API fornite dalle due implementazioni avrebbe richiesto la realizzazione di due differenti versioni di DTNperf, con inaccettabili duplicazioni del codice che avrebbero avuto un grandissimo impatto sulla sua manutenibilità ed estensibilità.

La risposta informatica a questo tipo di situazione è come sempre l’introduzione di un livello di astrazione, che fornisca un’unica interfaccia di alto livello e che solo internamente si occupi di effettuare le giuste chiamate alle funzioni specifiche per l’implementazione del Bundle Protocol attiva sulla macchina.

Il livello di astrazione introdotto a tale scopo in DTNperf consiste in una libreria di funzioni e strutture dati che ha preso semplicemente il nome di *Abstraction Layer* (o “AL”), letteralmente “livello di astrazione”, ma talvolta anche riferito come “API grigie”. Lo sviluppo è iniziato astraendo le API di DTN2 ed è proseguito includendo il supporto per ION, lasciando aperta la porta ad una possibile successiva estensione a IBR-DTN, oggetto di questa tesi, o ad altre implementazioni del Bundle Protocol. L’Abstraction Layer è mantenuto completamente separato dal codice sorgente che realizza la logica di DTNperf, pertanto può essere riutilizzato nello sviluppo di altre applicazioni DTN che vogliano supportare fin dall’inizio multiple implementazioni del Bundle Protocol.

Idealmente, ogni funzione e ogni tipo di dato dell’Abstraction Layer dovrebbe avere un corrispondente nelle API di ogni implementazione. Come accade regolarmente in questi casi, tuttavia, le profonde discrepanze tra le API, sviluppate del tutto indipendentemente le une dalle altre, fanno sì che una simile corrispondenza non

possa sempre essere rispettata. [Caini] Tali discrepanze originano non solo da approcci naturalmente eterogenei intrapresi da differenti team di sviluppo delle API, ma anche più in profondità dalle diverse architetture dei demoni stessi, dalle differenti funzionalità aggiuntive offerte dalle varie implementazioni del Bundle Protocol e dalle loro peculiarità, perfino dai linguaggi di programmazione con cui sono state sviluppate le API: quest'ultima considerazione diventa evidente se pensiamo che le API di IBR-DTN, in quanto scritte in C++, saranno probabilmente orientate agli oggetti, al contrario di quelle di DTN2 e ION, scritte in linguaggio C.

La rilevazione dell'implementazione del Bundle Protocol in esecuzione, verso le cui API inoltrare le chiamate alle funzioni dell'Abstraction Layer, avviene dinamicamente a runtime, esaminando l'elenco dei processi attivi. Tuttavia una prima selezione avviene al momento della compilazione: non è infatti necessario compilare l'Abstraction Layer per tutte le implementazioni previste, bensì è possibile rivolgersi soltanto a quelle di interesse, cosicché non occorra avere sempre a disposizione sorgenti e librerie di tutte le API supportate. Le funzioni che effettuano l'interfacciamento con API escluse dalla compilazione saranno semplicemente funzioni vuote, che restituiscono un codice di errore.

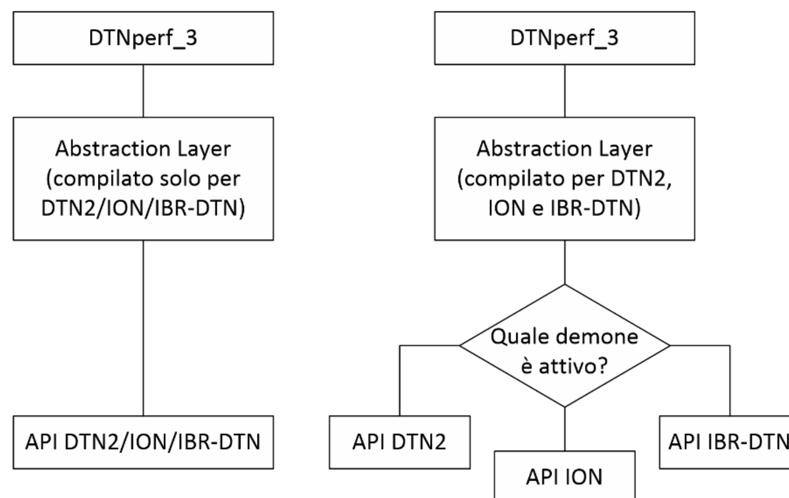

Figura 3-1 – Compilazione selettiva dell'Abstraction Layer e rilevazione a runtime dell'implementazione del Bundle Protocol in esecuzione

Presentiamo di seguito, senza pretesa di completezza, l'interfaccia e il modello dei dati offerti dall'Abstraction Layer, corredando funzioni e strutture dati che li compongono con brevi descrizioni. Maggiori dettagli saranno forniti, se e quando necessari, nell'analisi dell'estensione dell'Abstraction Layer alle API di IBR-DTN, nel prossimo capitolo.

3.3.1 Tipi di base

All'interno del file `types.h` l'Abstraction Layer ridefinisce per comodità alcuni tipi di base, esponendone nomi consistenti tra loro. Ne citiamo solo alcuni come esempio:

```
typedef char char_t;

typedef char* string_t;

typedef const char* cstring_t;

typedef char boolean_t;

typedef uint8_t u8_t;

typedef uint16_t u16_t;

typedef uint32_t u32_t;

typedef uint64_t u64_t;

typedef float float32_t;

typedef double float64_t;
```

3.3.2 Strutture dati

Il file `al_bp_types.h` definisce le strutture dati astratte e altri tipi di dato utilizzati dalle funzioni che costituiscono l'interfaccia dell'Abstraction Layer. Si noti che, poiché l'Abstraction Later nasce come astrazione delle API di DTN2, molte strutture dati e funzioni ricalcano tali API, con le opportune modifiche dettate dalla necessità di supportare più implementazioni.

```
typedef enum {
    BP_NONE = 0,
    BP_DTN,
    BP_ION
} al_bp_implementation_t;
```

individua un'implementazione del Bundle Protocol.

```
typedef enum {
    CBHE_SCHEME = 0,
    DTN_SCHEME,
} al_bp_scheme_t;
```

rappresenta uno schema di EID.

```
#define AL_BP_MAX_ENDPOINT_ID 256
typedef struct {
    char uri[AL_BP_MAX_ENDPOINT_ID];
} al_bp_endpoint_id_t;
```

memorizza un EID.

```
typedef int* al_bp_handle_t;
```

mantiene il punto di accesso per la comunicazione con il Bundle Protocol.

```
typedef u32_t al_bp_reg_token_t;
typedef u32_t al_bp_reg_id_t;
```

identificano una registrazione presso il demone.

```
typedef struct {
    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;
} al_bp_reg_info_t;
```

contiene informazioni su una registrazione.

```
typedef u32_t al_bp_timeval_t;
```

rappresenta una durata temporale.

```
typedef struct {
    u32_t secs;
    u32_t seqno;
} al_bp_timestamp_t;
```

memorizza un creation timestamp.

```
typedef enum {
    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,
} al_bp_bundle_delivery_opts_t;
```

rappresenta una delivery option.

```
typedef enum {
    BP_PRIORITY_BULK = 0,
    BP_PRIORITY_NORMAL = 1,
    BP_PRIORITY_EXPEDITED = 2,
    BP_PRIORITY_RESERVED = 3,
} al_bp_bundle_priority_enum;
typedef struct {
    al_bp_bundle_priority_enum priority;
    u32_t ordinal;
} al_bp_bundle_priority_t;
```

rappresentano la priorità di un bundle.

```
typedef struct {
    u32_t type;
    u32_t flags;
    struct {
        u32_t data_len;
        char* data_val;
    } data;
} al_bp_extension_block_t;
```

memorizza un extension block.

```
typedef struct {
    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;
} al_bp_bundle_spec_t;
```

memorizza un bundle, escluso il payload.

```
typedef enum {
    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,
} al_bp_status_report_reason_t;
```

rappresenta il *reason code* di uno status report.

```
typedef enum {
    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,
} al_bp_status_report_flags_t;
```

rappresenta le *status flag* di uno status report.

```
typedef struct {
    al_bp_endpoint_id_t source;
    al_bp_timestamp_t creation_ts;
    u32_t frag_offset;
    u32_t orig_length;
} al_bp_bundle_id_t;
```

definisce l'identificatore univoco di un bundle.

```
typedef struct {
    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;
} al_bp_bundle_status_report_t;
```
memorizza uno status report.

```
typedef enum {
    BP_PAYLOAD_FILE = 0,
    BP_PAYLOAD_MEM = 1,
    BP_PAYLOAD_TEMP_FILE = 2,
} al_bp_bundle_payload_location_t;
```
rappresenta la tipologia di locazione fisica in cui memorizzare o è memorizzato un payload.

```
typedef struct {
    al_bp_bundle_payload_location_t location;
    struct {
        u32_t filename_len;
        char* filename_val;
    } filename;
    struct {
        uint32_t buf_crc;
        u32_t buf_len;
        char* buf_val;
    } buf;
    al_bp_bundle_status_report_t* status_report;
} al_bp_bundle_payload_t;
```
memorizza un payload o il nome del file in cui è conservato.

```
typedef struct {
    al_bp_bundle_id_t* id;
    al_bp_bundle_spec_t* spec;
    al_bp_bundle_payload_t* payload;
} al_bp_bundle_object_t;
```
memorizza un bundle completo.

```
typedef enum {
    BP_SUCCESS = 0, /* success */
    BP_ERRBASE,     /* base error code */
    BP_ENOBPI,      /* no Bundle Protocol implementation */
    BP_EINVAL,      /* invalid argument */
```

```
    BP_ENULLPNTR,   /* operation on a null pointer */
    BP_EUNREG,      /* error unregistering EID */
    BP_ECONNECT,    /* error connecting to server */
    BP_ETIMEOUT,    /* operation timed out */
    BP_ESIZE,       /* payload/EID too large */
    BP_ENOTFOUND,   /* not found */
    BP_EINTERNAL,   /* internal error */
    BP_EBUSY,       /* registration already in use */
    BP_ENOSPACE,    /* no storage space */
    BP_ENOTIMPL,    /* function not yet implemented */
    BP_EATTACH,     /* error attaching to the BP */
    BP_EBUILDEID,   /* error building a local EID */
    BP_EOPEN,       /* error opening a connection with the BP */
    BP_EREG,        /* error registering an EID */
    BP_EPARSEEID,   /* error parsing a string to an EID */
    BP_ESEND,       /* error sending bundle*/
    BP_ERECV,       /* error receiving bundle */
    BP_ERECVINT     /* reception interrupted */
} al_bp_error_t;
```
rappresenta il codice di errore delle funzioni dell'Abstraction Layer.

3.3.3 Funzioni e procedure

Il file header `al_bp_api.h` e l'unità di compilazione `al_bp_api.c` rispettivamente definiscono e implementano le funzioni esposte dall'Abstraction Layer. Queste funzioni sono a loro volta suddivise in tre categorie, che chiameremo "di basso livello", "di utilità" e di "alto livello", anche se i termini possono in alcuni casi non essere del tutto appropriati.

La chiamata a una funzione "di basso livello" viene convertita dalla funzione stessa nell'invocazione del corretto servizio offerto dalla sottostante implementazione del Bundle Protocol in uso. Ciò avviene in realtà delegando ulteriori funzioni che, specifiche per ogni implementazione, si occupano dell'effettiva invocazione delle relative API: le funzioni specifiche per DTN2 sono definite in `bp_implementations/al_bp_dtn.h` e implementate in `bp_implementations/al_bp_dtn.c`, quelle per ION sono definite in `bp_implementations/al_bp_ion.h` e implementate in `bp_implementations/al_bp_ion.c`. Il corpo delle funzioni "di basso livello" segue sempre lo stesso modello, illustrato dal seguente esempio:

```
al_bp_error_t 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)
{
    // check parameters

    switch (al_bp_get_implementation())
    {
    case BP_DTN:
        return bp_dtn_send(handle, regid, spec, payload, id);
    case BP_ION:
        return bp_ion_send(handle, regid, spec, payload, id);
    default: // cannot find BP implementation
        return BP_ENOBPI;
    }
}
```

Le funzioni "di utilità" operano su alcune strutture dati dell'Abstraction Layer senza comunicare con il Bundle Protocol, ma per vari motivi devono farlo in maniera differente e specifica a seconda dell'implementazione del Bundle Protocol in esecuzione. La loro struttura è dunque simile alle funzioni "di basso livello".

Le funzioni "di alto livello" non hanno corrispondenze nelle API concrete sottostanti, bensì agiscono sulle strutture dati dell'Abstraction Layer, ad esempio ottenendo o modificando un valore, oppure effettuano operazioni non elementari che richiedono la chiamata a più di una funzione "di basso livello". L'implementazione di queste funzioni è unica e indipendente dal demone in esecuzione.

Il diagramma che segue sintetizza la sequenza di utilizzo delle più importanti funzioni dell'Abstraction Layer, in particolare quelle "di basso livello".

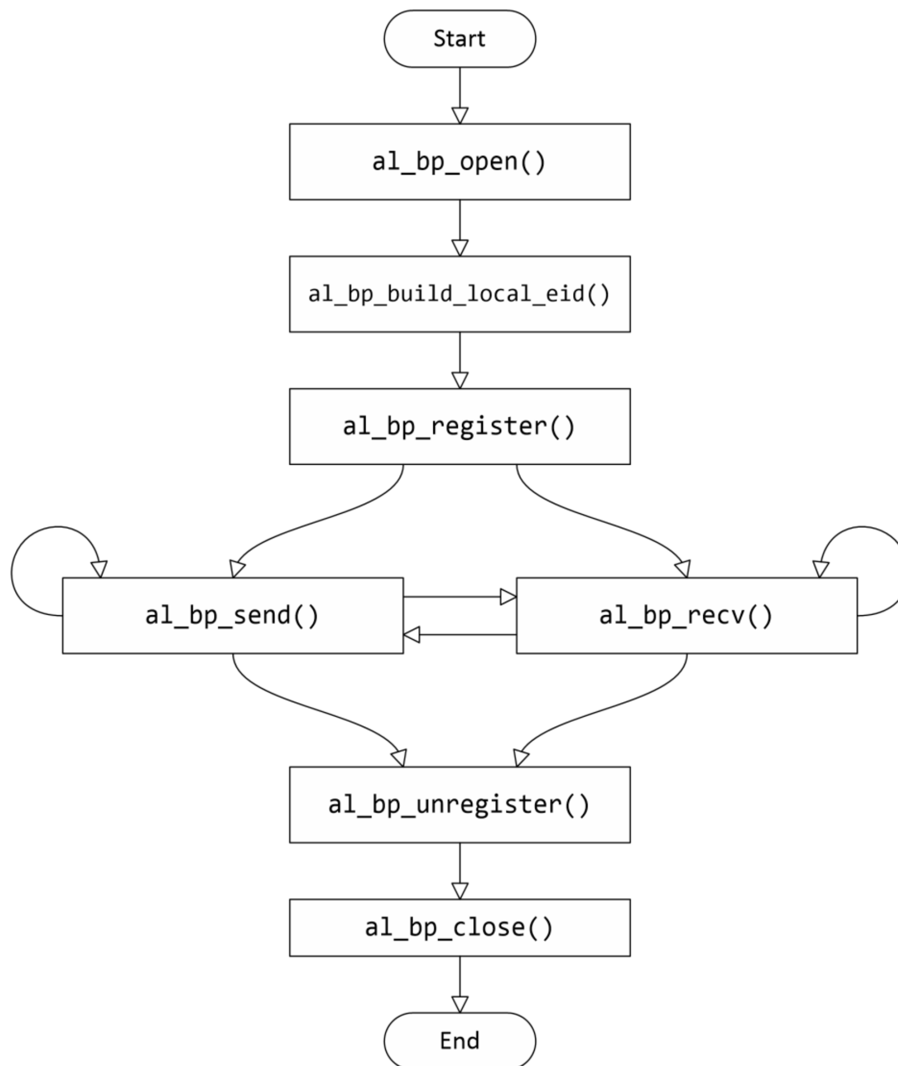

Figura 3-2 – Diagramma di flusso delle principali funzioni dell'Abstraction Layer

3.3.3.1 Funzioni di basso livello

```
al_bp_error_t al_bp_open(al_bp_handle_t* handle)
```

apre una connessione verso il demone sulla macchina locale.

```
al_bp_error_t al_bp_open_with_ip(
        char* daemon_api_IP,
        int daemon_api_port,
        al_bp_handle_t* handle)
```

apre una connessione verso il demone dato un indirizzo IP e una porta.

```
al_bp_error_t al_bp_errno(al_bp_handle_t handle)
```

restituisce il codice di errore associato a una connessione con il demone.

```
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)
```

restituisce l'EID ottenuto dalla concatenazione dell'EID del nodo locale, fornito dal demone, e di un demux token, scelto dall'applicazione.

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

crea una registrazione presso il demone, per ricevere e inviare bundle.

```
al_bp_error_t al_bp_find_registration(
        al_bp_handle_t handle,
        al_bp_endpoint_id_t* eid,
        al_bp_reg_id_t* newregid)
```

ricerca presso il demone una registrazione esistente per un certo EID.

```
al_bp_error_t 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)
```

consegna un bundle al Bundle Protocol per l'invio.

```
al_bp_error_t 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)
```

blocca il thread corrente in attesa della ricezione di un bundle.

```
al_bp_error_t al_bp_unregister(
        al_bp_handle_t handle,
        al_bp_reg_id_t regid,
        al_bp_endpoint_id_t eid)
```

rimuove una registrazione presso il demone.

```
al_bp_error_t al_bp_close(al_bp_handle_t handle)
```

chiude la connessione con il demone.

3.3.3.2 Funzioni di utilità

```
al_bp_error_t al_bp_parse_eid_string(
        al_bp_endpoint_id_t* eid,
        const char* str)
```

effettua il parsing di una stringa per ottenere un EID.

```
void al_bp_copy_eid(
        al_bp_endpoint_id_t* dst,
        al_bp_endpoint_id_t* src)
```

copia un EID.

```
al_bp_error_t al_bp_get_none_endpoint(
        al_bp_endpoint_id_t* eid_none)
```

restituisce l'EID `dtn: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)
```

popola la struttura `al_bp_bundle_payload_t`.

```
void al_bp_free_payload(al_bp_bundle_payload_t* payload)
```

libera la memoria occupata da un payload.

```
void al_bp_free_extension_blocks(al_bp_bundle_spec_t* spec)
void al_bp_free_metadata_blocks(al_bp_bundle_spec_t* spec)
```

liberano la memoria occupata dagli extension block di un bundle.

3.3.3.3 Funzioni di alto livello

```
al_bp_implementation_t al_bp_get_implementation()
```

restituisce l'implementazione del Bundle Protocol in esecuzione sulla macchina.

```
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)
```

invia un bundle, gestendo opportunamente allocazione, inizializzazione e deallocazione della memoria.

```
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)
```
riceve un bundle, gestendo opportunamente la memoria.

```
al_bp_error_t al_bp_bundle_create(
        al_bp_bundle_object_t* bundle_object)
```
alloca e inizializza la memoria per un `al_bp_bundle_object_t`.

```
al_bp_error_t al_bp_bundle_free(
        al_bp_bundle_object_t* bundle_object)
```
libera in profondità la memoria occupata da un `al_bp_bundle_object_t`.

```
al_bp_error_t al_bp_bundle_get_id(
        al_bp_bundle_object_t bundle_object,
        al_bp_bundle_id_t** bundle_id)
al_bp_error_t al_bp_bundle_get_source(
        al_bp_bundle_object_t bundle_object,
        al_bp_endpoint_id_t* source)
al_bp_error_t al_bp_bundle_set_source(
        al_bp_bundle_object_t* bundle_object,
        al_bp_endpoint_id_t source)
al_bp_error_t al_bp_bundle_get_dest(
        al_bp_bundle_object_t bundle_object,
        al_bp_endpoint_id_t* dest)
al_bp_error_t al_bp_bundle_set_dest(
        al_bp_bundle_object_t* bundle_object,
        al_bp_endpoint_id_t dest)
al_bp_error_t al_bp_bundle_get_replyto(
        al_bp_bundle_object_t bundle_object,
        al_bp_endpoint_id_t* replyto)
al_bp_error_t al_bp_bundle_set_replyto(
        al_bp_bundle_object_t* bundle_object,
        al_bp_endpoint_id_t replyto)
al_bp_error_t al_bp_bundle_get_priority(
        al_bp_bundle_object_t bundle_object,
        al_bp_bundle_priority_t* priority)
al_bp_error_t al_bp_bundle_set_priority(
        al_bp_bundle_object_t* bundle_object,
        al_bp_bundle_priority_t priority)
al_bp_error_t al_bp_bundle_get_unreliable(
        al_bp_bundle_object_t bundle_object,
        boolean_t* unreliable)
```

```
al_bp_error_t al_bp_bundle_set_unreliable(
        al_bp_bundle_object_t* bundle_object,
        boolean_t unreliable)
al_bp_error_t al_bp_bundle_get_critical(
        al_bp_bundle_object_t bundle_object,
        boolean_t* critical)
al_bp_error_t al_bp_bundle_set_critical(
        al_bp_bundle_object_t* bundle_object,
        boolean_t critical)
al_bp_error_t al_bp_bundle_get_flow_label(
        al_bp_bundle_object_t bundle_object,
        u32_t* flow_label)
al_bp_error_t al_bp_bundle_set_flow_label(
        al_bp_bundle_object_t* bundle_object,
        u32_t flow_label)
al_bp_error_t al_bp_bundle_get_expiration(
        al_bp_bundle_object_t bundle_object,
        al_bp_timeval_t* exp)
al_bp_error_t al_bp_bundle_set_expiration(
        al_bp_bundle_object_t* bundle_object,
        al_bp_timeval_t exp)
al_bp_error_t al_bp_bundle_get_creation_timestamp(
        al_bp_bundle_object_t bundle_object,
        al_bp_timestamp_t* ts)
al_bp_error_t al_bp_bundle_set_creation_timestamp(
        al_bp_bundle_object_t* bundle_object,
        al_bp_timestamp_t ts)
al_bp_error_t al_bp_bundle_get_delivery_opts(
        al_bp_bundle_object_t bundle_object,
        al_bp_bundle_delivery_opts_t* dopts)
al_bp_error_t al_bp_bundle_set_delivery_opts(
        al_bp_bundle_object_t* bundle_object,
        al_bp_bundle_delivery_opts_t dopts)
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)
```

restituiscono o impostano diversi campi di un bundle.

```
al_bp_error_t al_bp_bundle_get_payload_location(
        al_bp_bundle_object_t bundle_object,
        al_bp_bundle_payload_location_t* 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)
al_bp_error_t al_bp_bundle_get_payload_size(
        al_bp_bundle_object_t bundle_object,
        u32_t* size)
al_bp_error_t al_bp_bundle_get_payload_mem(
        al_bp_bundle_object_t bundle_object,
        char** buf, u32_t* buf_len)
al_bp_error_t al_bp_bundle_set_payload_mem(
        al_bp_bundle_object_t* bundle_object,
        char* buf, u32_t buf_len)
al_bp_error_t al_bp_bundle_get_payload_file(
        al_bp_bundle_object_t bundle_object,
        char_t** filename, u32_t* filename_len)
al_bp_error_t al_bp_bundle_set_payload_file(
        al_bp_bundle_object_t* bundle_object,
        char_t* filename, u32_t filename_len)
```

ottengono o impostano il contenuto di un payload.

4 ESTENSIONE DELL'ABSTRACTION LAYER A IBR-DTN

Nel capitolo 2 abbiamo analizzato l'implementazione IBR-DTN del Bundle Protocol, e ne abbiamo esplorato le API C++ nel paragrafo 2.4.2. Nel capitolo 3 abbiamo presentato DTNperf_3 per la valutazione del goodput in un'architettura DTN, con una panoramica, nel paragrafo 3.3, sull'Abstraction Layer, che permette a DTNperf di supportare in un'unica versione più implementazioni (originariamente DTN2 e ION). L'obiettivo di questa tesi è estendere l'Abstraction Layer a IBR-DTN affinché DTNperf possa essere utilizzato anche su questa implementazione.

4.1 Utilizzo di una libreria C++ in un programma scritto in C

DTNperf, l'Abstraction Layer, le API di DTN2 e le API di ION sono tutti scritti in linguaggio C, e pertanto naturalmente integrabili in uno stesso programma. Le API di IBR-DTN, invece, sono scritte in C++. Se il riutilizzo di codice e librerie scritti C in un programma C++ è, con le dovute accortezze, un procedimento relativamente lineare, la situazione simmetrica, quale la nostra, ovviamente non lo è affatto.

Una prima idea che potrebbe venire in mente è quella di sfruttare la retrocompatibilità del C++ con il C sostituendo l'uso del compilatore C con il compilatore C++, cosicché il codice C già presente continui ad essere compilato e allo stesso tempo possa essere integrato con codice C++. In realtà, eccetto nel caso di programmi molto semplici, ciò non è possibile: per quanto il C++ nasca dal C proponendosi di esserne un'estensione e un miglioramento che mantenga la compatibilità, a causa di alcune differenze semantiche e di nuove caratteristiche introdotte nel corso degli anni nell'uno ma non nell'altro linguaggio, tale compatibilità si è persa. In altre parole, il C non è un sottoinsieme del C++.

Ciò nonostante, il legame molto stretto di parentela tra i due linguaggi rimane, e, insieme al costrutto `extern "C"` introdotto dal C++ proprio in supporto alla compatibilità con il C, può essere sfruttato al nostro scopo. Vediamo come con un esempio minimale.

Supponiamo di avere una libreria scritta in C++ già compilata, costituita dal file di intestazione `print.h`, dal file sorgente `print.cpp`, non necessariamente fornito, e dalla libreria statica `libprint.a`. La libreria fornisce il metodo `void print(std::string str)`, che stampa una stringa C++ su standard output.

`print.h` contiene

```
#ifndef PRINT_H
#define PRINT_H

#include <string>

void print(std::string str);

#endif // PRINT_H
```

`print.cpp` contiene

```
#include "print.h"

#include <iostream>

void print(std::string str) {
    std::cout << str << std::endl;
}
```

Vogliamo utilizzare tale libreria nel nostro programma scritto in C, il cui sorgente è il file `main.c`, ma non possiamo farlo direttamente, ad esempio perché il tipo di dato `std::string` è proprio del C++ e perché il C non supporta i namespace. Introduciamo due ulteriori file, l'header `print_c.h` e il sorgente `print_c.cpp`, che definiscono una funzione `void print_c(char* str)` che presenta un'interfaccia compatibile con il linguaggio C ma il cui corpo potrà contenere codice C++, in particolare la chiamata alla funzione `print` della libreria; notiamo che, poiché il tipo `std::string` non è utilizzabile in C, il parametro passato da `main.c` deve essere di un tipo diverso compatibile con il C, in questo caso `char*`, poi convertito dalla funzione `print_c`.

`print_c.h` conterrà

```
#ifndef PRINT_C_H
#define PRINT_C_H

#ifdef __cplusplus
extern "C" {
#endif

void print_c(char* str);
```

```
#ifdef __cplusplus
}
#endif

#endif // PRINT_C_H
```

`print_c.cpp` conterrà

```
#include "print_c.h"

#include "print.h"

void print_c(char* str) {
    std::string cppstr(str);
    print(cppstr);
}
```

`print_c.cpp` inoltra una chiamata a `print_c` alla funzione della libreria fornita, effettuando anche le opportune conversioni di tipo. È scritto in C++ e il relativo file oggetto `print_c.o` è compilato con

```
g++ -c print_c.cpp
```

Il file su cui concentrare l'attenzione è però l'header `print_c.h`, che definisce l'interfaccia che `main.c` può utilizzare e che deve essere compatibile con il linguaggio C. L'attenzione è ovviamente rivolta al costrutto `extern "C"`, condizionato dall'identificatore `__cplusplus`, definito da ogni compilatore C++. Il punto è che `print_c.h` è utilizzato in due file sorgenti, `print_c.cpp`, compilato con un compilatore C++, e `main.c`, compilato con un compilatore C. Per quanto l'interfaccia definita da `print_c.h` appaia compatibile con il C, nella produzione dei file oggetto il compilatore C++ opera una traduzione dei nomi (detta *name mangling*) necessaria per realizzare l'overloading delle funzioni; poiché il C non supporta l'overloading, il compilatore C non effettua invece il name mangling. Questa differenza di comportamento tra i due compilatori comporta una discordanza tra nomi di funzioni che, al momento del linking, non permette la risoluzione dei riferimenti e fa fallire il processo. `extern "C"` informa il compilatore di adottare, per le funzioni contenute al suo interno, il linking del C, ovvero ne inibisce il name mangling. `#ifdef __cplusplus` semplicemente abilita `extern "C"` solo per il compilatore C++, ovvero solo quando `print_c.h` è incluso in `print_c.cpp`, non in `main.c`.

Alla fine, il sorgente `main.c` del nostro programma sarà

```
#include "print_c.h"

int main(void) {
    // ...

    print_c("Hello World!");

    // ...
}
```

La sua compilazione per ottenere il file oggetto `main.o` è effettuata dal compilatore C, con il comando

```
gcc -c main.c
```

Il linking per produrre l'eseguibile, infine, deve però essere effettuato dal linker C++, a causa delle librerie standard del C++ coinvolte. Ciò non è un problema, perché le incompatibilità di linguaggio sono già state risolte al momento della compilazione. Il comando per il linking è

```
g++ -o main main.o print_c.o libprint.a
```

Trasponendo l'esempio all'Abstraction Layer, le API C++ di IBR-DTN costituiscono la libreria esterna, le funzioni "di basso livello" e "di utilità" dell'Abstraction Layer sono il programma in C, e le funzioni specifiche per IBR-DTN, da realizzare, che le funzioni "di basso livello" e "di utilità" sfrutteranno corrispondono all'interfaccia di conversione tra C e C++. In particolare, le funzioni specifiche per IBR-DTN saranno definite nel file di intestazione `bp_implementations/al_bp_ibr.h` e implementate nel file sorgente `bp_implementations/al_bp_ibr.cpp`.

4.2 Integrazione dei Makefile

La compilazione di DTNperf utilizza lo strumento `make`, istruito tramite appositi file di configurazione, i Makefile. Poiché il codice dell'Abstraction Layer è separato dal nucleo di DTNperf, abbiamo un Makefile per ognuna delle due parti, più un terzo Makefile a coordinare i primi due.

Al comando `make` è necessario passare come parametri le directory in cui ricercare i file header delle API delle implementazioni del Bundle Protocol che si vogliono supportare: le implementazioni per cui la directory non è indicata saranno escluse. La locazione per le API di DTN2 può essere specificata impostando la variabile `DTN2_DIR`, per ION la variabile `ION_DIR`: per prima cosa aggiungiamo allora il

riconoscimento di una terza variabile, `IBRDTN_DIR`, per le API di IBR-DTN. La sintassi definitiva sarà

```
make [DTN2_DIR=dtn2_dir] [ION_DIR=ion_dir] [IBRDTN_DIR=ibrdtn_dir]
```

dove almeno una delle tre variabili deve essere assegnata.

Se la variabile `IBRDTN_DIR` è impostata, predisponiamo il compilatore e il linker di conseguenza: nel Makefile dell'Abstraction Layer abilitiamo la ricerca degli header delle API all'interno del percorso passato, definiamo in compilazione l'identificatore `IBRDTN_IMPLEMENTATION` per poter condizionare il codice al supporto per IBR-DTN, compiliamo con il compilatore C++ il file `bp_implementations/al_bp_ibr.cpp`; nel Makefile di DTNperf sostituiamo il linker C con il linker C++ (per quanto visto nel paragrafo 4.1) e linkiamo le librerie `libibrcommon.a`, `libibrdtn.a` e `libz.a` (come illustrato nel paragrafo 2.4.2).

4.3 Prime estensioni del codice

Sistemata la fase di compilazione, cominciamo ora ad apportare le prime semplici modifiche al codice dell'Abstraction Layer.

Le strutture dati definite nel file `al_bp_types.h` non necessitano, fortunatamente, di alcuna modifica: nell'eventualità, alcuni campi saranno opportunamente convertiti o lasciati vuoti. L'unico tipo che richiede un intervento è l'enumeratore `al_bp_implementation_t`, che individua un'implementazione del Bundle Protocol, cui deve essere aggiunto l'elemento relativo a IBR-DTN: il risultato finale è

```
typedef enum {
    BP_NONE = 0,
    BP_DTN,
    BP_ION,
    BP_IBR
} al_bp_implementation_t;
```

Anche le funzioni "di alto livello", per come le abbiamo definite, non richiedono integrazioni. L'unica eccezione è evidentemente la funzione `al_bp_implementation_t al_bp_get_implementation()`, che restituisce l'implementazione del Bundle Protocol attiva sulla macchina tramite l'enumeratore appena esteso.

Il riconoscimento del demone in esecuzione si basa sul comando `ps` e sulla presenza di un processo attivo dal nome `dtnd` o `rfxclock`, rispettivamente per DTN2 e ION. Sfortunatamente, come abbiamo già avuto modo di osservare, il demone di IBR-DTN e quello di DTN2 sono omonimi, perciò occorre una ulteriore

discriminante tra i due. La scelta è ricaduta sulla directory di installazione, approfittando del fatto che il demone DTN2 è installato di default nel percorso `/usr/bin/dtnd`, mentre il demone IBR-DTN in `/usr/sbin/dtnd` o `/usr/local/sbin/dtnd`, a seconda che l'installazione sia avvenuta da repository o da sorgenti.

Il comando eseguito per il riconoscimento dell'implementazione IBR-DTN, in C con la mediazione della funzione `system`, è quindi il seguente:

```
ps axe | \
grep -e '/usr/sbin/dtnd' -e '/usr/local/sbin/dtnd' | \
grep -v -q 'grep'
```

dove `ps axe` mostra informazioni su tutti i processi, tra cui le variabili d'ambiente, che includono il percorso completo del file eseguibile, `grep -e '/usr/sbin/dtnd' -e '/usr/local/sbin/dtnd'` seleziona solo le righe contenenti uno tra i due percorsi di interesse, `grep -v -q 'grep'` esclude lo stesso processo `grep` ed esce con stato affermativo se è rimasta almeno una riga, quella corrispondente al demone cercato.

L'estensione delle funzioni "di basso livello" e "di utilità", invece, richiede l'esistenza delle funzioni specifiche che si interfacciano con le API di IBR-DTN e la cui implementazione vedremo in dettaglio nel paragrafo 4.4. Tuttavia, una volta che di queste ultime è stata definita almeno l'interfaccia e uno scheletro, è già possibile modificare le funzioni "di basso livello" affinché invochino ognuna, se è attivo il demone IBR-DTN, la controparte specifica. Ad esempio, la funzione `al_bp_send`, già utilizzata a scopo chiarificatore nel paragrafo 3.3.3, diventa

```
al_bp_error_t 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)
{
    // check parameters

    switch (al_bp_get_implementation())
    {
    case BP_DTN:
        return bp_dtn_send(handle, regid, spec, payload, id);
    case BP_ION:
        return bp_ion_send(handle, regid, spec, payload, id);
    case BP_IBR:
        return bp_ibr_send(handle, regid, spec, payload, id);
```

```
        default: // cannot find BP implementation
            return BP_ENOBPI;
        }
}
```

4.4 Implementazione delle funzioni di basso livello

Il lavoro centrale della tesi è ora l'implementazione delle funzioni specifiche per IBR-DTN, che ricordiamo saranno definite in `bp_implementations/al_bp_ibr.h` e implementate in `bp_implementations/al_bp_ibr.cpp`, verso cui le funzioni "di basso livello" e "di utilità" dell'Abstraction Layer inoltrano le loro chiamate.

Questo paragrafo elenca uno per uno i corrispettivi delle funzioni "di basso livello" e ne fornisce uno schema più o meno approfondito di come sono stati realizzati, senza comunque entrare nei dettagli delle righe di codice. Si farà spesso riferimento alle API C++ descritte nel paragrafo 2.4.2. Una sezione è infine riservata a considerazioni generali sulle funzioni "di utilità".

4.4.1 IbrHandle

Tutte le funzioni "di basso livello" accettano tra gli argomenti un parametro `al_bp_handle_t handle`, che rappresenta il punto di accesso per la comunicazione, tramite le rispettive API, con il demone che realizza il Bundle Protocol.

Sfortunatamente, il tipo `al_bp_handle_t`, definito come puntatore a intero da `typedef int* al_bp_handle_t`, non è adeguato alle API di IBR-DTN, accessibili tramite un'istanza di `dtn::api::Client`. D'altra parte, però, sappiamo che in C e C++ tutti i tipi puntatore sono in realtà lo stesso tipo di dato e, previa una conversione, intercambiabili: è allora immediato utilizzare il parametro `handle` come un puntatore a `Client`.

In realtà, una sessione di comunicazione con il demone IBR-DTN richiede di mantenere, oltre all'oggetto `Client`, anche altre informazioni, necessarie in momenti diversi: è il caso dell'indirizzo IP e della porta su cui il demone accetta connessioni, dello stream che incapsula la socket aperta verso il demone, dell'ultimo stato di uscita di una funzione dell'Abstraction Layer, che deve poter essere ottenuto in qualsiasi momento tramite `al_bp_errno`. Il risultato finale è quindi la seguente struttura dati:

```
struct IbrHandle {
    char* daemonIP;
    int daemonPort;
    ibrcommon::socketstream* stream;
```

```
    dtn::api::Client* client;
    al_bp_error_t error;
};
```

Il cast da `al_bp_handle_t handle` sarà semplicemente dato da

```
IbrHandle* ibrHandle = (IbrHandle*) handle;
```

4.4.2 bp_ibr_open e bp_ibr_open_with_ip

La funzione

```
al_bp_error_t bp_ibr_open_with_ip(
        const char* daemon_api_ip,
        int daemon_api_port,
        al_bp_handle_t* handle_p)
```

è la specializzazione della funzione dell'Abstraction Layer `al_bp_open_with_ip` e apre la connessione verso il demone IBR-DTN.

Nell'ordine, viene aperta una `ibrcommon::tcpsocket` verso l'indirizzo IP e la porta specificati, viene creata, gestendo le eventuali eccezioni di connessione, l'istanza di `ibrcommon::socketstream` con cui leggere e scrivere sulla socket, e viene allocato e inizializzato l'`IbrHandle`, restituito come parametro di uscita.

`al_bp_error_t bp_ibr_open(al_bp_handle_t* handle_p)`, chiamata da `al_bp_open`, è semplicemente un caso particolare della funzione precedente, dove la macchina di destinazione è quella locale e la porta è la predefinita 4550. Il corpo sarà quindi banalmente

```
return bp_ibr_open_with_ip("localhost", 4550, handle_p);
```

4.4.3 bp_ibr_errno

La funzione

```
al_bp_error_t bp_ibr_errno(al_bp_handle_t handle)
```

corrispondente di `al_bp_errno`, si limita a restituire il valore contenuto nel campo `al_bp_error_t error` di `IbrHandle`, che mantiene l'ultimo codice di successo o errore restituito da una funzione "di basso livello" in una sessione di comunicazione con il Bundle Protocol. A tal fine, ognuna delle funzioni presentate in questo paragrafo che accetti il parametro `al_bp_handle_t handle` assegna al campo `error` di `IbrHandle` il proprio codice di uscita subito prima di restituirlo.

4.4.4 bp_ibr_register e bp_ibr_find_registration

La funzione

```
al_bp_error_t bp_ibr_register(
        al_bp_handle_t handle,
```

```
          al_bp_reg_info_t* reginfo,
          al_bp_reg_id_t* newregid)
```
corrisponde a `al_bp_register` e avvia la sessione di comunicazione verso il demone, registrando l'applicazione per la ricezione di bundle e abilitandola all'invio.

Poiché la connessione è già stata aperta da `bp_ibr_open`, è sufficiente creare un oggetto `Client` passandogli il demux token estratto dall'EID contenuto nel parametro `reginfo`, chiamarne il metodo `connect()` e memorizzarlo nell'`IbrHandle`. Oltre al demux token, il costruttore di `Client` richiede anche lo stream, aperto in `bp_ibr_open`, attraverso il quale comunicare con il demone: vediamo dunque il perché del campo `ibrcommon::socketstream* stream` in `IbrHandle`. Il parametro di uscita `newregid`, specifico di DTN2, è inutilizzato, e il valore da esso puntato è inizializzato, per completezza, a zero.

La funzione
```
  al_bp_error_t bp_ibr_find_registration(
          al_bp_handle_t handle,
          al_bp_endpoint_id_t* eid,
          al_bp_reg_id_t* newregid)
```
chiamata dalla generica `al_bp_find_registration`, avrebbe il compito di verificare la presenza di una registrazione per un dato EID e, nel caso, restituirne i dettagli. Tuttavia, poiché tale funzionalità non è offerta dalle API di IBR-DTN né, fortunatamente, è di particolare interesse ai nostri scopi, non possiamo far altro che limitarci a fornire sempre una risposta negativa.

4.4.5 bp_ibr_build_local_eid: ottenere l'EID del nodo locale

La funzione
```
  al_bp_error_t bp_ibr_build_local_eid(
          al_bp_handle_t handle,
          al_bp_endpoint_id_t* local_eid,
          const char* service_tag,
          al_bp_scheme_t type)
```
chiamata da `al_bp_build_local_eid`, costruisce un EID apponendo il demux token scelto dall'applicazione all'EID amministrativo del nodo locale. L'EID restituito è utilizzato al momento dell'invocazione di `bp_ibr_register`.

La specifica appare molto semplice, ma purtroppo la classe `Client` non offre alcun servizio per richiedere l'EID locale al demone. Dobbiamo allora trovare un espediente per giungere allo stesso risultato.

L'idea è quella di creare un oggetto `Client` al solo scopo di inviare un bundle a noi stessi, ricevere il bundle, estrarne l'EID del mittente e da questo rimuovere il demux token. La strada sembrerebbe non praticabile a causa della necessità di conoscere proprio l'EID che stiamo cercando, ma IBR-DTN supporta un EID "jolly", `api:me`, che il demone sostituisce con l'EID del mittente al momento dell'invio del bundle. Ciò nonostante, questo approccio non funziona, per una motivazione completamente differente: il demone IBR-DTN non consegna bundle destinati all'applicazione mittente.

Perseveriamo e, anziché spedire un bundle al nostro EID, scegliamo come destinazione un EID multicast qualsiasi, per il quale possiamo abilitarci alla ricezione tramite l'apposito costruttore di `Client`, quello che accetta anche il parametro `const dtn::data::EID &group`: così facendo, EID del mittente ed EID destinatario non coincidono. Ci rendiamo presto conto che l'esito è comunque negativo, per un motivo simile al precedente: se il `Client` registratosi per un EID multicast cui è destinato un bundle è anche il mittente del bundle, il demone non procede alla consegna.

Siamo però sulla buona strada. È sufficiente creare due oggetti `Client`: uno ha il compito esclusivo di inviare un bundle all'EID multicast, l'altro è dedicato solamente alla ricezione per quell'EID, cosicché gli EID non multicast con cui i due `Client` sono registrati siano differenti e il demone sia obbligato a consegnare il bundle. Ricevuto il bundle da parte del secondo `Client`, è sufficiente leggerne il mittente e rimuovere il demux token per ottenere l'EID amministrativo. Poiché il bundle è indirizzato a un EID multicast, esso non viene eliminato al momento della consegna, perché potrebbe essere destinato anche ad altri: per minimizzarne la permanenza presso il demone è sufficiente impostare un lifetime molto breve, ad esempio di un secondo.

Il procedimento appare oneroso, ma deve essere eseguito una sola volta all'avvio dell'applicazione, dopodiché l'EID del nodo locale è memorizzato ed è immediatamente disponibile ad ogni successiva chiamata di `bp_ibr_build_local_eid`. Osserviamo anche che la memorizzazione nell'`IbrHandle` di indirizzo IP e porta del demone, specificati al momento della chiamata a `bp_ibr_open_with_ip`, si è resa necessaria affinché gli stessi, se diversi dalla macchina locale e dalla porta predefinita, siano utilizzati nella creazione dei due `Client`.

Ricavato l'EID amministrativo, la costruzione dell'EID dell'applicazione è solamente una questione di concatenazione di stringhe, cui vanno aggiunti controlli

per alcuni casi particolari e per assicurarsi che la lunghezza finale non superi quella massima supportata dalla struttura `al_bp_endpoint_id_t`.

Accenniamo al fatto che esiste un altro metodo per ottenere l'EID amministrativo, che si serve del protocollo testuale di comunicazione con il demone. Poiché ne siamo venuti a conoscenza dagli sviluppatori di IBR-DTN solo in seguito, l'approccio originario è stato mantenuto.

4.4.6 bp_ibr_send: restituire l'identità del bundle inviato

La funzione

```
al_bp_error_t bp_ibr_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)
```

specializza per IBR-DTN la funzione `al_bp_send` e consegna un bundle al demone per l'invio.

L'implementazione della funzione richiede principalmente di convertire i parametri `spec` e `payload` in un oggetto `dtn::data::Bundle`, che possa essere passato al `Client` invocando l'operatore `<<`. Dapprima vengono assegnati l'EID del mittente, l'EID del destinatario, il report-to EID, la priorità, le delivery option, il lifetime del bundle. Successivamente vengono aggiunti gli extension block, creando, popolando e aggiungendo all'istanza di `Bundle` gli opportuni `dtn::data::ExtensionBlock`. Ad essi segue il payload, il cui contenuto è caricato, tramite la classe `ibrcommon::BLOB`, da una locazione di memoria o da un file in un oggetto `dtn::data::PayloadBlock`, anch'esso poi aggiunto al `Bundle`. A questo punto il bundle è pronto per l'invio. Il parametro `regid` è inutilizzato.

La funzione dovrebbe restituire al chiamante, per mezzo del parametro `id`, l'identità del bundle, e in particolare il creation timestamp (timestamp e sequence number). Sfortunatamente, le API C++ di IBR-DTN non offrono questa funzionalità, perché l'operatore `<<` non ha un valore di ritorno e non modifica il contenuto del proprio parametro `Bundle`. Per ora possiamo solamente limitarci a restituire due valori "plausibili", sfruttando il fatto che un oggetto `Bundle` è inizializzato dal costruttore con il timestamp attuale e con un sequence number crescente univoco a livello di applicazione. Tali valori, tuttavia, non hanno alcun legame con il creation timestamp reale, assegnato dal demone, del bundle inviato: in particolare, costruendo un nuovo oggetto `Bundle` immediatamente dopo l'invocazione dell'operatore `<<`, al solo scopo di generare un creation timestamp, se da una parte

il timestamp così ottenuto sarà probabilmente coincidente con quello reale o al più riporterà il secondo successivo (poiché la granularità dei timestamp del Bundle Protocol è di un secondo, ciò si verifica se lo scatto del secondo è caduto tra un'istruzione e la successiva), dall'altra il sequence number sarà quasi certamente diverso. Vedremo nel paragrafo 5.1.2 le conseguenze di questo problema.

4.4.7 bp_ibr_recv

La funzione

```
al_bp_error_t bp_ibr_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)
```

è la specializzazione di `al_bp_recv`. Essa riceve un bundle in maniera sincrona, sospendendo il thread corrente fino all'avvenuta ricezione o alla scadenza del timeout.

L'implementazione segue lo schema inverso rispetto a `bp_ibr_send`, ovvero popola i parametri `spec` e `payload` a partire da un oggetto della classe `Bundle`. La ricezione del bundle prevede l'utilizzo del metodo `getBundle` di `Client`, eventualmente passando il timeout dell'operazione con il parametro opzionale, e la gestione delle eccezioni relative alla scadenza del timeout e all'interruzione della connessione con il demone.

Incontriamo già un primo problema: secondo la semantica di `al_bp_recv`, che accetta un timeout in millisecondi, un valore nullo indica che la ricezione ha successo solo se un bundle è immediatamente disponibile, mentre il massimo intero rappresentabile dal tipo `al_bp_timeval_t` corrisponde a un timeout infinito; viceversa, `getBundle` accetta un timeout in secondi, e un valore nullo implica un'attesa illimitata. Ciò significa che `al_bp_recv` permetterebbe di specificare un timeout con granularità di un millisecondo e al minimo nullo, ma `getBundle` consente granularità di un secondo e timeout minimo ancora di un secondo. Non possiamo far altro che accontentarci e accettare il fatto che, quando il Bundle Protocol in uso è IBR-DTN, la granularità e il valore minimo del timeout passato a `al_bp_recv` saranno soggetti alle limitazioni di `getBundle`. In ogni caso, non ci sono implicazioni sull'utilizzo usuale che prevede un timeout infinito, se non la necessità di convertire tra le sue due diverse rappresentazioni.

Ottenuta l'istanza di `Bundle`, procediamo assegnando agli opportuni campi del parametro `spec` il creation timestamp, l'EID del mittente, l'EID del destinatario, il

report-to EID, la priorità, le delivery option e gli extension block. In seguito estraiamo il payload, riversandone il contenuto in memoria o su un file, a seconda del valore di `location`, e popolando il parametro `payload` di conseguenza. Infine, se il bundle è uno status report, con l'ausilio della classe `dtn::data::StatusReportBlock` ne ricaviamo dal payload tutte le informazioni, riempiamo con esse una struttura `al_bp_bundle_status_report_t`, e assegniamo quest'ultima al campo `status_report` di `payload`.

4.4.8 bp_ibr_unregister

La funzione

```
al_bp_error_t bp_ibr_unregister(
        al_bp_handle_t handle,
        al_bp_reg_id_t regid,
        al_bp_endpoint_id_t eid)
```

è la corrispondente specifica di `al_bp_unregister`, nonché la duale di `bp_ibr_register`, e rimuove dal demone la registrazione dell'applicazione.

Praticamente, la funzione si limita a reperire l'oggetto `Client` dall'`IbrHandle`, a chiamarne il metodo `close()`, e a deallocarlo. I parametri `regid` e `eid` sono inutilizzati. Notiamo che al termine di questa funzione la socket di comunicazione con il demone è ancora aperta.

4.4.9 bp_ibr_close

La funzione

```
al_bp_error_t bp_ibr_close(al_bp_handle_t handle)
```

corrispondente specifica di `al_bp_close` e duale di `bp_ibr_open`, chiude effettivamente la connessione con il demone.

Prima di tutto, se il `Client` è ancora allocato significa che la funzione `bp_ibr_unregister` non è stata chiamata, e viene quindi chiamata sul momento. Dopodiché viene recuperata la `socketstream` dall'`IbrHandle` e ne viene invocato il metodo `close()`: osserviamo dunque per la seconda volta la necessità di aver memorizzato lo stream. Infine, l'`IbrHandle` viene deallocato insieme ai suoi campi ancora allocati.

4.4.10 Funzioni di utilità

Le funzioni "di utilità", che non comunicano con il demone ma operano su alcune strutture dell'Abstraction Layer, consistono in più o meno complesse manipolazioni della memoria, perciò non indugeremo sui dettagli.

```
al_bp_error_t bp_ibr_parse_eid_string(
        al_bp_endpoint_id_t* eid,
        const char* str)
```

corrispondente di `al_bp_parse_eid_string`, ma invocata anche da `al_bp_get_none_endpoint` passando la stringa `"dtn:none"`, si serve della classe `dtn::data::EID` delle API di IBR-DTN per effettuare il parsing da una stringa a un Endpoint Identifier.

```
void bp_ibr_copy_eid(
        al_bp_endpoint_id_t* dst,
        al_bp_endpoint_id_t* src)
```

corrispondente di `al_bp_copy_eid`, è una banale copia tra stringhe.

```
al_bp_error_t bp_ibr_set_payload(
        al_bp_bundle_payload_t* payload,
        al_bp_bundle_payload_location_t location,
        char* val, int len)
```

corrispondente di `al_bp_set_payload`, e

```
void bp_ibr_free_payload(al_bp_bundle_payload_t* payload)
```

corrispondente di `al_bp_free_payload`, sono costituite da semplici assegnazioni, copie di valori e deallocazioni di memoria.

```
void bp_ibr_free_extension_blocks(al_bp_bundle_spec_t* spec)
```

corrispondente di `al_bp_free_extension_blocks`, e

```
void bp_ibr_free_metadata_blocks(al_bp_bundle_spec_t* spec)
```

corrispondente di `al_bp_free_metadata_blocks`, sono anch'esse semplici deallocazioni.

5 MODIFICHE A DTNPERF E TESTING

5.1 Modifiche a DTNperf

Nonostante l'Abstraction Layer sia stato introdotto precisamente allo scopo di consentire l'esecuzione di DTNperf al di sopra di diverse implementazioni del Bundle Protocol in maniera completamente trasparente, cioè senza che il nucleo del programma faccia distinzione tra le implementazioni, ci sono in realtà delle eccezioni, a causa delle quali l'introduzione del supporto a IBR-DTN ha richiesto alcuni ritocchi al codice di DTNperf. In generale, infatti, occorre considerare che le varie implementazioni, seppur tutte conformi alle specifiche date dalla RFC 5050, presentano ognuna le proprie peculiarità, le più importanti delle quali riassunte in seguito.

5.1.1 Estensioni e schemi degli EID

In primo luogo, alcune implementazioni ammettono delle estensioni: ION, ad esempio, supporta i parametri ECOS, *Extended Class of Service*. Per non limitare le potenzialità di DTNperf, le opzioni di invio comprendono alcune di queste estensioni, ma possono essere specificate solamente se il demone sottostante le implementa: continuando con l'esempio, le opzioni ECOS sono impostabili solo se DTNperf è lanciato su ION. Di conseguenza DTNperf deve essere informata su quale sia l'implementazione in uso per poter effettuare controlli specifici.

Un altro punto critico riguarda i due schemi possibili per gli EID, DTN e CBHE. DTN2, ad esempio, predilige lo schema DTN, ION lo schema CBHE, e il supporto allo schema alternativo è spesso carente: DTN2 supporta lo schema CBHE solo tramite l'applicazione di specifiche patch rilasciate dalla NASA per permettere l'interoperabilità di ION con DTN2, requisito fondamentale per la certificazione in ambito CCSDS [CCSDS]; ION supporta nativamente anche lo schema DTN, ma con alcune limitazioni, come il non poter utilizzare il routing CGR.

In questo ambito risulta complesso anche il problema dell'identità, DTN o CBHE o entrambe, di un nodo: un nodo DTN2 ha sempre e solo un'identità DTN, ma è possibile registrare un'applicazione con un EID nello schema CBHE fornito dall'applicazione stessa; un nodo ION ha sempre obbligatoriamente un'identità CBHE, ma può assumere anche un'identità DTN (forzatamente del tipo `dtn://`*`hostname`*`.dtn`), per di più senza essere consapevole della doppia identità e senza riconoscere che i propri due EID si riferiscono entrambi a se stesso! In IBR-DTN, fortunatamente, la questione è molto più semplice e chiara: entrambi gli

schemi sono consentiti e pienamente supportati, ma un nodo può assumere in ogni momento un solo EID, dello schema prescelto.

Per le motivazioni esposte, allora, in DTNperf lo schema dell'EID locale utilizzato per impostazione predefinita varia con l'implementazione sottostante: per DTN2 e IBR-DTN viene adottato lo schema DTN, mentre per ION lo schema CBHE. È dunque stata necessaria l'introduzione di una terza via, specifica per IBR-DTN, negli if che stabiliscono lo schema da adottare.

Un'ulteriore differenza tra le implementazioni è rappresentata dalla possibilità per un'applicazione di comunicare con un demone in esecuzione su una macchina diversa da quella locale e/o in ascolto su una porta TCP diversa dalla predefinita: tale possibilità è offerta soltanto da DTN2 e IBR-DTN. DTNperf, ad esempio, consente di specificare indirizzo IP e porta del demone tramite le opzioni `--ip-addr` e `--ip-port`, ma queste sono significative (e accettate) solo se l'implementazione del Bundle Protocol attiva è una delle suddette.

5.1.2 Validazione degli ACK su IBR-DTN

La particolarità di IBR-DTN che ha avuto il maggiore impatto sul codice di DTNperf è però l'impossibilità per il mittente di ottenere, tramite le API C++, il timestamp e soprattutto il sequence number dei bundle inviati. La loro conoscenza è necessaria al client di DTNperf quando viene utilizzata la modalità di controllo di congestione window-based, perché per lo scorrimento della finestra non è sufficiente la ricezione di un acknowledgment, ma è anche richiesto che esso riporti al proprio interno uno dei creation timestamp memorizzati in attesa di essere confermati. Questo tipo di "validazione" degli ACK è stato introdotto nella seconda versione di DTNperf per aumentare la robustezza a fronte di arrivi di bundle relativi a test precedenti; a partire dalla terza versione, questo controllo è stato reso superfluo ai fini originari dal demux token dinamico del client, contenente il PID dell'applicazione; ciò nonostante, la validazione è stata mantenuta, perché utile al fine di impedire lo scorrimento erroneo della finestra in caso di duplicazione di un bundle dati o del relativo ACK, evento infrequente ma possibile.

Discutendo l'implementazione della funzione `bp_ibr_send`, nel paragrafo 4.4.6, abbiamo detto che, vista la limitazione delle API di IBR-DTN, ci siamo accontentati della possibilità di restituire come timestamp e sequence number dei valori "plausibili", comunque non allineati con il creation timestamp reale assegnato al bundle dal demone. Ciò comporta la mancata validazione, da parte del client di DTNperf, di tutti o quasi gli acknowledgment.

Per risolvere la questione avremmo potuto rimuovere la validazione degli ACK, completamente oppure soltanto se il demone attivo sul nodo che ospita il client è IBR-DTN. Invece abbiamo optato per un approccio che, sempre solo per IBR-DTN, si limita a rilassare le condizioni per la validazione.

Come abbiamo notato, mentre il sequence number restituito da `bp_ibr_send` è completamente slegato da quello reale, così non è per il timestamp: per come tale funzione è stata implementata, se applicazione e demone dispongono di orologi sincronizzati, il che è sempre vero se sono in esecuzione su una stessa macchina, come di norma, il timestamp restituito coincide, o al più segue di un secondo, il timestamp effettivo. Allora possiamo considerare valido un ACK se esiste almeno un creation timestamp ancora da confermare il cui timestamp è uguale o superiore di un secondo rispetto a quello riportato nell'ACK, senza prendere in considerazione i sequence number. In presenza di più creation timestamp che soddisfano questa condizione si ritiene confermato quello minore, ovvero quello con timestamp minore (ciò è fondamentale affinché tutte le validazioni successive vadano a buon fine) o, in caso di parità, con sequence number minore.

La validazione rigorosa degli ACK è realizzata dalla funzione `is_in_info`, mentre quella specifica per IBR-DTN da `is_in_info_timestamp`; entrambe le funzioni sono definite nell'header `bundle_tools.h` e implementate nel file `bundle_tools.c`.

5.2 Testing su testbed in Virtualbricks

Il testing di DTNperf con le modifiche finora descritte è stato effettuato su un testbed costituito da una rete di macchine virtuali gestita tramite il software Virtualbricks [Apollonio].

5.2.1 Virtualbricks

Virtualbricks è un gestore di testbed di rete virtuali per ambiente GNU/Linux, sviluppato dall'Università di Bologna e ora incluso nelle distribuzioni Debian e Ubuntu. Virtualbricks realizza una piattaforma per la gestione centralizzata e tramite interfaccia grafica di macchine virtuali QEMU (Quick Emulator) e KVM (Kernel-based Virtual Machine), interconnesse per mezzo di dispositivi VDE (Virtual Distributed Ethernet), quali switch, cavi, emulatori di canale e altro.

Virtualbricks nasce proprio con l'intento di semplificare la creazione, la configurazione e la gestione di testbed costituiti da una molteplicità di macchine virtuali interconnesse in un'infrastruttura di rete complessa, anch'essa virtuale, sui quali effettuare ricerche ed esperimenti sulle tecnologie di rete, e in particolare sulle DTN. In questo senso, ad esempio, si rivelano molto utili gli emulatori di canale,

perché permettono di riprodurre canali di comunicazione reali di cui modificare a piacere ritardi, perdite, errori, banda, interruzioni, ecc.

Tra i principali vantaggi di Virtualbricks rispetto ad altre soluzioni annoveriamo il supporto fornito nella costruzione della rete virtuale e la possibilità di esportare e importare un testbed come un unico file archivio, facilitandone notevolmente la migrazione, la condivisione e l'archiviazione.

5.2.2 Test su IBR-DTN

Il layout del semplice testbed da noi utilizzato è illustrato in figura. I nodi "vm1", "vm2" e "vm4" sono macchine virtuali KVM con sistema operativo Debian 8 su cui sono installati DTN2, ION e IBR-DTN, mentre il nodo "vm3" è la macchina host, con sistema operativo Ubuntu 14.04 su cui è installato solo IBR-DTN: il testbed è quindi "ibrido", perché comprende anche l'host, ma naturalmente Virtualbricks si occupa soltanto degli elementi virtuali. Gli EID in schema DTN configurati per IBR-DTN e DTN2 (ove presente) sono quelli mostrati.

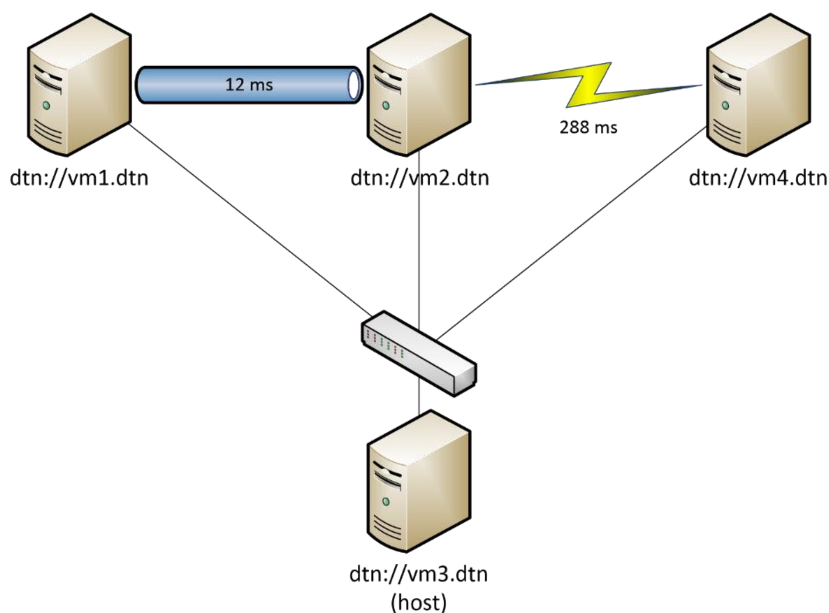

Figura 5-1 – Testbed ibrido (Virtualbricks e macchina host)
per il testing di DTNperf_3 su IBR-DTN e DTN2

Una prima serie di test è stata effettuata compilando DTNperf soltanto per IBR-DTN ed eseguendone client, server ed eventualmente monitor tutti al di sopra dell'implementazione IBR-DTN, con i demoni in comunicazione tramite convergence layer TCP. Lo scopo era quello di verificare il corretto funzionamento di DTNperf in seguito alle modifiche apportate.

Riportiamo soltanto un esempio semplice, in cui il client è posto sulla macchina vm1, il server su vm4 e il monitor sull'host vm3. Il demone sulla macchina vm1 è

configurato con una regola di routing statica che impone a bundle destinati a vm4 di essere inoltrati a vm2, e simmetricamente è configurato il demone su vm4 per bundle destinati a vm1, cosicché i bundle tra client e server non transitino per vm3. Inoltre, le connessioni tra vm1 e vm2 e tra vm2 e vm4 sono configurate staticamente e il modulo di discovery è disabilitato con l'opzione `--nodiscovery`, per garantire che i pacchetti IP delle due connessioni TCP su cui sono veicolati i bundle tra vm1 e vm4 non transitino per lo switch, bypassando gli emulatori di canale.

Server e monitor sono eseguiti senza opzioni particolari con i comandi

```
dtnperf_vIBRDTN --server -v --debug=2
```

e

```
dtnperf_vIBRDTN --monitor -v --debug=2
```

mentre il client con il comando

```
dtnperf_vIBRDTN --client -d dtn://vm4.dtn -m dtn://vm3.dtn \
                -D 200k -W 4 -P 50k -r -f -v --debug=2
```

che prevede l'invio di quattro bundle di 50 kB, uno di seguito all'altro, richiedendo la generazione di tutti gli status report.

Il log prodotto dal monitor, opportunamente riadattato, è:

RX Time	Report Source	Report TS	Rep. SQN	Rep. Type	Bundle Source	Bundle TS	Bndl SQN	DLV	CT	RCV	FWD	DEL
0.0000	dtn://vm1.dtn	520625763	10	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	7			520625763		
0.0067	dtn://vm1.dtn	520625763	17	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	14			520625763		
0.0134	dtn://vm1.dtn	520625763	24	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	21			520625763		
0.0173	dtn://vm1.dtn	520625763	29	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	26			520625763		
0.0700	dtn://vm2.dtn	520625763	1	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	7			520625763		
0.0839	dtn://vm1.dtn	520625763	32	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	7				520625763	
0.1243	dtn://vm2.dtn	520625763	7	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	14			520625763		
0.1406	dtn://vm1.dtn	520625763	34	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	14				520625763	
0.1595	dtn://vm2.dtn	520625763	13	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	21			520625763		
0.1758	dtn://vm1.dtn	520625763	37	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	21				520625763	
0.1936	dtn://vm2.dtn	520625763	19	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	26			520625763		
0.2076	dtn://vm1.dtn	520625763	40	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	26				520625763	
1.0094	dtn://vm4.dtn	520625764	1	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	7			520625764		
1.0197	dtn://vm4.dtn	520625764	7	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	14			520625764		
1.0300	dtn://vm4.dtn	520625764	14	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	7	520625764				
1.0763	dtn://vm4.dtn	520625764	19	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	14	520625764				
1.3205	dtn://vm2.dtn	520625764	0	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	7				520625764	
1.6199	dtn://vm4.dtn	520625765	3	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	21			520625765		
1.6559	dtn://vm4.dtn	520625765	8	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	21	520625765				
1.9270	dtn://vm2.dtn	520625765	2	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	14				520625765	
1.9404	dtn://vm2.dtn	520625765	6	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	21				520625765	
2.8063	dtn://vm4.dtn	520625766	3	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	26			520625766		

2.8507	dtn://vm4.dtn	520625766	8	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	26	520625766				
3.0867	dtn://vm2.dtn	520625766	0	SR	dtn://vm1.dtn/dtnperf:/src_1813	520625763	26				520625766	
3.1578	dtn://vm1.dtn/ dtnperf:/src_1813	520625766	3	STOP								

5.2.3 Interoperabilità con DTN2

Una seconda serie di test ha invece avuto lo scopo di saggiare l'interoperabilità tra IBR-DTN e DTN2 nelle configurazioni più semplici possibili per i due demoni, cioè con ancora convergence layer TCP, routing statico e nessuna opzione di sicurezza, compressione o similari.

Si noti che l'interoperabilità deve essere su due livelli: a livello bundle, ovvero le due implementazioni del Bundle Protocol devono interagire correttamente, e a livello di applicazione, che nel nostro caso significa che istanze di DTNperf su implementazioni differenti devono essere in grado di comunicare tra loro. L'interoperabilità a livello bundle è ovviamente un prerequisito per la seconda, ma, nonostante il comune riferimento alle stesse specifiche della RFC 5050, essa è tutt'altro che banale, e richiede una notevole familiarità con i file di configurazione delle varie implementazioni. Viceversa, l'interoperabilità a livello di applicazione è garantita quasi automaticamente da DTNperf, che ha quindi il pregio di liberare l'utente dalla necessità di utilizzare, su macchine con implementazioni differenti, strumenti differenti, con tutti i problemi che questo comporta.

L'esempio che riportiamo prevede client IBR-DTN su vm1, server DTN2 su vm4 e monitor dedicato, ovvero interno al client; il demone in esecuzione su vm2 è DTN2, mentre la macchina host non è coinvolta.

Il server è eseguito con il consueto comando

```
dtnperf_vDTN2 --server -v --debug=2
```

mentre il client con

```
dtnperf_vIBRDTN --client -d dtn://vm4.dtn \
                -T 10 -P 50k -W 1 -C -v --debug=2
```

che causa l'invio di bundle da 50 kB per 10 secondi, con una finestra di congestione di un solo bundle, richiedendo il trasferimento con custodia.

Segue il log prodotto dal monitor interno, che conferma l'interoperabilità, almeno in condizioni standard, tra DTN2 e IBR-DTN:

RX Time	Report Source	Report TS	Rep. SQN	Rep. Type	Bundle Source	Bundle TS	Bndl SQN	DLV	CT	RCV	FWD	DEL
0.0000	dtn://vm2.dtn	520625967	1	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625967	7		520625967			
1.8144	dtn://vm4.dtn	520625968	2	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625967	7		520625968			
1.8436	dtn://vm4.dtn	520625968	4	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625967	7	520625968				
1.9458	dtn://vm2.dtn	520625969	7	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625968	9		520625969			

3.7610	dtn://vm4.dtn	520625970	7	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625968	9		520625970			
3.7862	dtn://vm4.dtn	520625970	9	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625968	9	520625970				
3.8934	dtn://vm2.dtn	520625971	13	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625970	9		520625971			
5.7099	dtn://vm4.dtn	520625972	12	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625970	9		520625972			
5.7345	dtn://vm4.dtn	520625972	14	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625970	9	520625972				
5.8376	dtn://vm2.dtn	520625973	19	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625972	9		520625973			
7.6512	dtn://vm4.dtn	520625974	17	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625972	9		520625974			
7.6758	dtn://vm4.dtn	520625974	19	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625972	9	520625974				
7.7892	dtn://vm2.dtn	520625975	25	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625974	9		520625975			
9.6057	dtn://vm4.dtn	520625975	22	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625974	9		520625975			
9.6307	dtn://vm4.dtn	520625976	24	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625974	9	520625976				
9.7415	dtn://vm2.dtn	520625977	31	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625976	9		520625977			
11.5563	dtn://vm4.dtn	520625977	27	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625976	9		520625977			
11.5812	dtn://vm4.dtn	520625977	29	SR	dtn://vm1.dtn/dtnperf:/src_1867	520625976	9	520625977				
11.6295	dtn://vm1.dtn/ dtnperf:/src_1867	520625978	9	STOP								

Ulteriori test sono stati effettuati variando i ritardi introdotti dai canali di comunicazione e introducendo un tasso di perdita.

5.2.4 Interoperabilità con ION

Test analoghi sono stati infine condotti sull'implementazione ION e anch'essi si sono conclusi con esito positivo. Nel corso di questi test si è però evidenziata, a livello di convergence layer TCP, un'incompatibilità di IBR-DTN con ION, nel caso in cui il nodo ION utilizzasse lo schema CBHE. Un'analisi del problema ha permesso di ricondurre il bug a una non corretta interpretazione, in IBR-DTN, della sintassi CBHE, che prevede, come accennato nel paragrafo 1.1.1.2, che il demux token sia sempre specificato, utilizzando "0" per indicare il bundle agent stesso (ad esempio `ipn:124.0`). Segnalato il problema, esso è stato rapidamente risolto dai manutentori di IBR-DTN, ma si presti attenzione al fatto che la patch non è ancora stata integrata nella versione di IBR-DTN destinata al rilascio.

6 CONCLUSIONI

Una delle novità della terza versione di DTNperf è stata l'estensione del supporto dal solo DTN2 anche ad ION. A questo scopo è stata introdotta una libreria, l'Abstraction Layer, che espone un'interfaccia comune per l'interazione con le diverse implementazioni del Bundle Protocol e che solo internamente invoca le API relative a DTN2 o ad ION, a seconda di quale sia effettivamente in esecuzione sulla macchina. In questa tesi abbiamo esteso l'Abstraction Layer affinché DTNperf_3 supporti anche una terza implementazione, IBR-DTN.

In un primo momento ci siamo dedicati a studiare IBR-DTN, comprendendone la compilazione e l'installazione, la configurazione e l'avvio, gli strumenti di comunicazione e le modalità di utilizzo di base, virando poi sull'esplorazione e la sperimentazione delle API C++ tramite le quali realizzare applicazioni abilitate all'invio e alla ricezione di bundle.

Successivamente, ottenuta una sufficiente dimestichezza con IBR-DTN e le sue API, abbiamo effettivamente proceduto all'estensione dell'Abstraction Layer, implementando una serie di funzioni che mappano la sua interfaccia alle suddette API. Tra i problemi in cui ci siamo imbattuti in questa fase vale la pena ricordarne due: le API di IBR-DTN non forniscono un servizio per ricavare l'EID del nodo locale, pertanto abbiamo dovuto fare ricorso a un escamotage basato sulla lettura del mittente di un bundle inviato a noi stessi; le stesse API non restituiscono al chiamante l'identità esatta di un bundle inviato, perciò siamo stati costretti a rilassare le condizioni per la validazione, da parte del client di DTNperf, degli acknowledgment provenienti dal server.

Infine abbiamo svolto alcuni test, tramite i quali abbiamo dapprima verificato il corretto funzionamento di DTNperf su IBR-DTN, quindi accertato la piena interoperabilità di istanze diverse di DTNperf operanti su IBR-DTN, DTN2 e ION.

La nuova versione di DTNperf con supporto a IBR-DTN sarà rilasciata come sempre come software libero con licenza GPL. Sarà inoltre fornita alla NASA-JPL perché venga aggiornata la versione di DTNperf già inclusa in ION. Infine sarà proposta agli sviluppatori di IBR-DTN affinché ne valutino l'inclusione nel proprio pacchetto.

7 BIBLIOGRAFIA

[Apollonio] P. Apollonio, C. Caini, M. Giusti, D. Lacamera, "Virtualbricks for DTN satellite communications research and education", in Proc. of PSATS 2014, Genoa, Italy, July 2014, pp. 1-14

[Burleigh] S. Burleigh, A. Hooke, L. Torgerson, K. Fall, V. Cerf, B. Durst, K. Scott, "Delay-tolerant networking: An approach to interplanetary Internet", IEEE Communications Magazine, vol. 41, no. 6, pp. 128-136, June 2003

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

[CCSDS] Sito web del CCSDS: http://public.ccsds.org/default.aspx

[DTNRG] IRTF Delay-Tolerant Networking Research Group (DTNRG), https://irtf.org/dtnrg

[DTNRGcode] Implementazioni del Bundle Protocol sul sito web dell'IRTF DTN Research Group: https://sites.google.com/site/dtnresgroup/home/code

[DTNWG] IETF Delay Tolerant Networking Working Group (DTNWG), https://datatracker.ietf.org/wg/dtnwg/charter/

[Farrell] A. McMahon, S. Farrell, "Delay- and Disruption Tolerant Networking", IEEE Internet Computing, vol. 13, no. 6, pp. 82-87, Nov./Dec. 2009

[IBR_API] IBR-DTN Api Documentation, https://www.ibr.cs.tu-bs.de/projects/ibr-dtn/apidoc/0.12/api.pdf

[IBR_GH] IBR-DTN Wiki su Github, https://github.com/ibrdtn/ibrdtn/wiki

[ION_DOC] 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

[ION_SF] S. Burleigh, "Delay-Tolerant Networking suitable for use in spacecraft", https://sourceforge.net/projects/ion-dtn/

[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, https://tools.ietf.org/html/rfc5050

[Rodolfi] Michele Rodolfi, "DTNperf 3: Software per la valutazione delle prestazioni delle Delay/Disruption Tolerant Network (DTN)", 2011

[Schildt] 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, https://www.ibr.cs.tu-bs.de/papers/schildt-ibrdtn.pdf

[Warthman] Forrest Warthman, "Delay- and Disruption-Tolerant Networks (DTNs): A Tutorial", Ver. 3.2, Sep. 2015, http://www.warthman.com/projects-IRTF-Interplanetary-Internet-IPN-Delay-Tolerant-Networks-DTN-Tutorial.htm