POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Prototipazione di una architettura di tipo Delay-Tolerant Network operante su tecnologie di livello fisico eterogenee

Relatore
prof. Fulvio Risso

Supervisore:
dott. Gabriele Castellano

Candidato
Carlo Pettinato

Dicembre 2018

Ringraziamenti

Un particolare ringraziamento va al prof. Fulvio Risso, per aver accettato l'incarico di relatore, e per il supporto fornitomi durante l'intero percorso da tesista. Desidero inoltre ringraziare Gabriele Castellano, per i consigli e l'aiuto pratico ricevuto durante lo sviluppo del progetto di tesi.

Il lavoro realizzato in questa tesi rientra all'interno di un progetto più ampio, svolto in collaborazione con l'azienda Tierra S.p.A, che offre soluzioni telematiche in ambito agricolo, delle costruzioni, del rilievo e dell'industria in generale. Pertanto, vorrei rivolgere un sentito ringraziamento all'azienda, ed in particolare a Riccardo Loti, per avermi dato la possibilità di prendere parte al progetto, e per la grande disponibilità dimostratami.

Infine, non per importanza, ci tengo a ringraziare tutti coloro che mi hanno sostenuto durante questi anni universitari. In particolare, vorrei ringraziare la mia famiglia, i miei amici ed Arianna, in quanto, è soprattutto grazie a loro che è stato possibile raggiungere questo traguardo.

Indice

Capitolo 1

Introduzione

La sempre crescente diffusione di dispositivi mobili, in concomitanza con la rapida evoluzione delle tecnologie di comunicazione wireless, introducono una serie di nuove sfide dal punto di vista del *networking*. E' sempre più comune imbattersi in contesti in cui le entità coinvolte nella comunicazione sono in continuo movimento e caratterizzate, nella maggior parte dei casi, da risorse hardware limitate. In tali scenari, identificabili con il nome di "*challenged networks*", non è possibile fare affidamento ai paradigmi di comunicazione tradizionali. Le comunicazioni in Internet, ad esempio, si fondano sull'assunzione secondo cui, in ogni istante, è garantita l'esistenza di almeno un percorso end-to-end tra la sorgente e la destinazione del traffico. Questo non è assolutamente auspicabile nel contesto di una *challenged network*, che è invece caratterizzata da continue interruzioni della connettività, e quindi, un percorso completo tra sorgente e destinazione potrebbe non esistere mai. Oltre che alla mobilità dei dispositivi, la connettività intermittente può essere dovuta a fattori intrinseci dell'ambiente in cui la *challenged network* è collocata, come la presenza di ostacoli che si interpongono tra i dispositivi atti a comunicare, oppure il verificarsi di fenomeni atmosferici ed ambientali avversi. La connettività intermittente porta con sé una serie di ulteriori problemi, come partizionamento della rete, ritardi lunghi e/o variabili, ed alto tasso di perdite, che rendono la comunicazione ancora più complessa. Gli scenari soggetti a tali problematiche sono molteplici e spaziano dall'ambito militare a quello interplanetario, dalle reti di sensori alle aree non dotate di alcuna infrastruttura di telecomunicazione, quali villaggi e zone rurali. Le comunicazioni interplanetarie, ad esempio, sono caratterizzate da lunghi ritardi di propagazione e da significative interruzioni della visibilità tra le entità coinvolte, le quali si muovono continuamente lungo percorsi orbitali. Le reti di sensori o *Wireless Sensor Network (WSN)*, invece, coinvolgono dispositivi tipicamente mobili e dotati di limitate risorse computazionali e di memorizzazione, per cui, spesso, alcuni link di comunicazione sono intenzionalmente spenti al fine di risparmiare energia. Oltre alle problematiche proprie dei suddetti scenari, è opportuno considerare che gran parte delle applicazioni esistenti suppongono di operare su reti connesse, caratterizzate da ritardi piccoli o perlomeno trascurabili. Poiché la modifica di tali applicazioni richiederebbe uno sforzo materialmente insostenibile, date le assunzioni precedenti, l'approccio più semplice è quello di introdurre un framework comune a tutti i nodi della rete, una sorta di interfaccia che permetta di far fronte alle esigenze tipiche delle *challenged network*.

Il paradigma del *Delay-Tolerant Networking* rappresenta una potenziale soluzione al problema, atta a garantire l'interoperabilità all'interno e tra diverse *challenged network*, mediante la definizione di un'astrazione rispetto ai protocolli sottostanti. Le architetture DTN mirano alla creazione di una rete "overlay", capace di operare sugli stack protocollari esistenti, all'interno dei contesti applicativi più svariati. Nel caso di Internet, ad esempio, una DTN potrebbe operare sulla suite TCP/IP, mentre, nel caso di reti di sensori, potrebbe favorire l'interconnessione di dispositivi che utilizzano la tecnologia Bluetooth, piuttosto che protocolli di comunicazione non ancora standardizzati. A tal scopo, è necessario estendere lo stack di rete dei dispositivi coinvolti nella comunicazione, e quindi, in altre parole, introdurre un nuovo livello protocollare: il *Bundle Layer*. Tale strato, comune a tutti i nodi partecipanti alla rete DTN, in combinazione con il *Bundle Protocol*, definisce le modalità e il formato dei messaggi con cui i essi possono comunicare tra loro. Il *Delay-Tolerant Networking*, pertanto, risponde all'esigenza di garantire l'interoperabilità tra reti eterogenee, ognuna caratterizzata dalle proprie assunzioni e dalle proprie architetture protocollari. Tuttavia, il suo obiettivo principale è quello di garantire, con una buona probabilità, che un pacchetto giunga da sorgente a destinazione, nonostante la mancanza di un percorso completo tra esse. Il perseguimento di tale obiettivo è raggiunto mediante l'utilizzo di un meccanismo asincrono di inoltro dei messaggi, che utilizza un approccio molto simile a quello adottato per la posta elettronica, noto con il nome di *store-and-forward message switching*. Secondo quest'approccio, un messaggio viene mantenuto localmente fin quando non risulta possibile consegnarlo direttamente a destinazione, oppure inoltrarlo a qualche altro nodo intermedio, ritenuto un potenziale next-hop verso la destinazione.

1.1 Obiettivo della tesi

Date le considerazioni precedenti, lo sviluppo di questa tesi è orientato alla prototipazione di un'architettura di rete robusta, basata sul paradigma del *Delay Tolerant Networking (DTN)*, che consenta ai nodi coinvolti di poter operare su tecnologie di trasmissione eterogenee, in maniera del tutto trasparente. Tutto ciò si traduce nella capacità, da parte di un nodo partecipante alla rete DTN, di poter usufruire dei servizi e delle primitive di comunicazione fornite dal *Bundle Protocol*, prescindendo dalle specifiche tecnologie e dai protocolli utilizzati nei livelli sottostanti. L'utilizzo del paradigma DTN, e in particolare, dell'approccio *store-and-forward*, consente di sfruttare la mobilità dei dispositivi, e di conseguenza, i contatti opportunistici che vengono a crearsi tra essi, al fine di fronteggiare i partizionamenti della rete e permettere ai messaggi di giungere a destinazione. L'architettura proposta è rivolta principalmente a quegli scenari applicativi che coinvolgono dispositivi mobili con risorse limitate, e di conseguenza, soggetti a vincoli di banda e di consumo energetico. La soluzione, pertanto, vuole ottenere l'interoperabilità tra le più comuni tecnologie di comunicazione wireless esistenti in tali contesti, quali WiFi, Bluetooth/BLE e IEEE 802.15.4, il tutto con il minimo impatto possibile sulle risorse disponibili. Quello che si desidera ottenere è che uno stesso dispositivo sia in grado, in un dato momento, di comunicare con dispositivi vicini diversi utilizzando tecnologie differenti, come WiFi e Bluetooth/BLE, ad esempio. Al dispositivo, il fatto che una trasmissione dati avvenga su una tecnologia piuttosto che un'altra, deve risultare

del tutto indifferente, in quanto esso deve semplicemente preoccuparsi di ricevere e trasmettere dati, secondo le modalità previste dal *Bundle Protocol*.

L'elaborato è strutturato come segue:

- **Capitolo 2:** introduce il paradigma del *Delay-Tolerant Networking* e il *Bundle Protocol*, e successivamente, svolge una breve panoramica sugli algoritmi di routing utilizzati in ambito DTN.
- **Capitolo 3:** descrive l'architettura protocollare dello standard Bluetooth, sia nella sua accezione classica che nella versione a basso consumo energetico (BLE).
- **Capitolo 4:** presenta l'architettura software di IBR-DTN, l'implementazione open-source del *Bundle Protocol* utilizzata per lo sviluppo di questa tesi.
- **Capitolo 5:** descrive l'implementazione software realizzata allo scopo di fornire ad IBR-DTN il supporto alla tecnologia Bluetooth.
- **Capitolo 6:** illustra le prove sperimentali eseguite allo scopo di validare il funzionamento dell'implementazione realizzata, descritta nel 5
- **Capitolo 7:** espone le conclusioni ed alcune idee di progetti futuri che potranno utilizzare tale tesi come base di partenza.
- **Appendice A:** contiene le istruzioni per la compilazione, la configurazione e l'utilizzo del software IBR-DTN, presentato nel Capitolo 4.
- **Appendice B:** descrive un lavoro preliminare allo sviluppo della tesi, volto a validare il funzionamento del *Bundle Protocol* in un ambiente virtuale, realizzato allo scopo di emulare uno scenario applicativo reale.

Capitolo 2

Il paradigma del Delay-Tolerant Networking

Il seguente capitolo introduce il concetto del *Delay-Tolerant Networking* e mostra una serie di scenari in cui tale paradigma di comunicazione può essere applicato. La trattazione prosegue con la descrizione del *Bundle Protocol*, un protocollo sperimentale che definisce il modo con cui le entità coinvolte nei suddetti scenari possono comunicare. Infine, il capitolo effettua una breve panoramica sul problema del routing in ambito DTN, illustrando alcuni dei principali algoritmi sviluppati a tale scopo. Quanto espresso nel capitolo è frutto di una collaborazione con lo studente Giacomo Ranieri, il cui progetto di tesi presenta argomenti affini, e fa riferimento a [1–6].

2.1 Delay-Tolerant Networking

2.1.1 Introduzione

Le *Delay Tolerant Networks (DTNs)* definiscono un'architettura end-to-end capace di fornire connettività nelle cosiddette *"challenged networks"*, ovvero ambienti particolarmente sfidanti dal punto di vista della comunicazione, in cui non è possibile fare fede alle assunzioni su cui si fondano i protocolli Internet tradizionali. La suite dei protocolli Internet, infatti, è concepita per operare su collegamenti pressoché stabili e su architetture di rete con determinate caratteristiche:

- Esistenza di un percorso end-to-end tra sorgente e destinazione per l'intera durata di una sessione di comunicazione;
- Ritardi bassi e fissi;
- Basso tasso di perdite;
- Velocità di trasmissione simmetriche;
- Presenza di un'infrastruttura di comunicazione.

Tali assunzioni non sono assolutamente ipotizzabili nel contesto di una "challenged network", che è invece caratterizzata da connettività intermittente, ritardi lunghi e/o variabili, alto tasso di errori e asimmetria nelle trasmissioni. Il *Delay-Tolerant Networking* si propone di realizzare architetture di rete che possano operare in tali contesti, utilizzando un approccio di tipo *store-and-forward*, secondo cui un messaggio può essere memorizzato per molto tempo, almeno fino a quando risulti possibile inoltrato a qualcun altro.

Il concetto di DTN nasce nell'ambito delle comunicazioni interplanetarie, ma attualmente trova moltissime applicazioni in ambito commerciale, scientifico, militare e di servizio pubblico. Oggigiorno, infatti, è sempre più comune scontrarsi con scenari applicativi in cui i dispositivi che devono comunicare sono in movimento e operano a potenza limitata. A causa della mobilità dei nodi, i collegamenti tra essi sono spesso ostruiti dalla presenza di ostacoli, e inoltre, in alcuni casi, i collegamenti fisici vengono intenzionalmente spenti al fine di preservare energia. Questi fenomeni sono alla base della connettività intermittente, che ha come conseguenza naturale quella del partizionamento della rete. In tali scenari, la comunicazione mediante i protocolli TCP/IP causerebbe un numero significativo di dati persi. Ad esempio, nel caso di un pacchetto che non possa essere inoltrato immediatamente, il TCP assume il congestionamento della rete, scarta il pacchetto e prova a ritrasmetterlo, abbassando gradualmente la velocità di ritrasmissione, fino a chiudere la sessione nel caso di intermittenza troppo elevata.

Come anticipato, per far fronte alle problematiche delle "challenged networks" e trarre beneficio dai contatti tra i nodi, le DTN utilizzano la tecnica dello *store-and-forward message switching*. Secondo questo paradigma, concettualmente simile al meccanismo utilizzato per la posta elettronica, interi messaggi o frammenti di essi sono spostati dallo storage di un nodo a quello di un altro, lungo un percorso che potenzialmente conduce alla destinazione. Quando un nodo riceve un pacchetto, esso viene inoltrato immediatamente se possibile, oppure memorizzato localmente per essere trasmesso in futuro. Per questo motivo, ogni "router DTN" deve disporre di un supporto che permetta di memorizzare i messaggi per un tempo indefinito (un hard disk, ad esempio), garantendo così la persistenza dell'informazione. Questo approccio risulta molto diverso rispetto a quanto accade nei router IP, che utilizzano dei buffer di memoria per accodare i pacchetti in attesa di essere inoltrati, garantendone una persistenza dell'ordine dei millisecondi. Nel caso delle DTN, è necessario che lo storage sia persistente poiché alcuni link di comunicazione potrebbero essere non disponibili per lunghi periodi di tempo, nelle situazioni in cui venga richiesta la ritrasmissione di un messaggio oppure nel caso di un nodo che trasmetta e/o riceva i dati molto più velocemente di un suo vicino.

2.1.2 Scenari

La seguente sezione illustra una serie di scenari, caratterizzati da fattori che rendono la comunicazione problematica, in cui l'utilizzo del paradigma del *Delay-Tolerant Networking* risulta più che plausibile. Innanzitutto, parlando di contesti soggetti a connettività intermittente, è opportuno fare distinzione tra i contatti pianificati e quelli opportunistici (figura 2.1). Lo scenario tipico dei contatti pianificati o

scheduled è quello dello spazio, in cui i nodi si muovono su percorsi orbitali predicibili, tanto che è possibile prevedere o ricevere gli instanti in cui essi occuperanno le loro future posizioni, e di conseguenza, organizzare le sessioni di comunicazione. Per contatti opportunistici, invece, si intendono i contatti tra un trasmettitore e un ricevitore in istanti non programmati. E' il caso di persone, veicoli, aerei o satelliti, che potrebbero voler scambiare informazioni quando risultato in linea di vista e quindi abbastanza vicini da poter comunicare usando la loro potenza, seppure limitata.

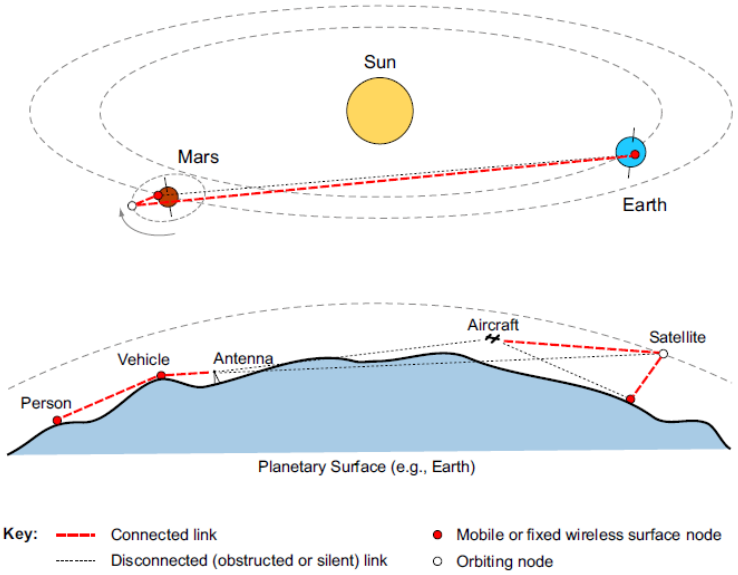


Figura 2.1: Esempi di contatti pianificati (comunicazioni interplanetarie) e opportunistici (comunicazioni sulla superficie terrestre) [5]

Comunicazioni spaziali e interplanetarie

Il *Delay-Tolerant Networking* nasce come una soluzione alla comunicazione tra stazioni terrestri e veicoli spaziali. Le comunicazioni nello spazio sono spesso caratterizzate da un'interruzione della visibilità tra le entità coinvolte nella comunicazione. Inoltre, i lunghi tempi di propagazione del segnale rendono inefficienti le classiche soluzioni adottate per la comunicazione. Tuttavia, le comunicazioni spaziali sono predicibili, almeno in gran parte dei casi. Questa peculiarità semplifica, di gran lunga, gli algoritmi di routing dei messaggi, che devono essere capaci di calcolare un percorso ottimale verso le destinazioni, garantendo un utilizzo opportuno della banda. L'obiettivo qui è quello di far fronte ai lunghi e variabili ritardi di propagazione, che possono oscillare tra i secondi e le ore.

Comunicazioni militari

A differenza delle comunicazioni spaziali, gli scenari militari, che in gran parte dei casi risultano caratterizzati da una topologia connessa, sono soggetti ad interruzioni imprevedibili, a causa di condizioni meteorologiche, interferenze, mobilità dei nodi,

distruzione delle infrastrutture o guasti. In tali contesti, la comunicazione è basata su tecnologie cablate o wireless, con bassi ritardi di propagazione. Quello che si mira ad ottenere, è una soluzione che fornisca affidabilità delle comunicazioni, nel contesto di reti in cui i collegamenti sono spesso non funzionanti e non esiste una connettività continua end-to-end (tra sorgente e destinazione).

Accesso ad Internet in aree rurali

Il *Delay-Tolerant Networking* può risultare una buona soluzione in villaggi o aree rurali, in cui si vuole fornire accesso ad Internet a prezzi accessibili. In tali scenari, infatti, l'accesso a Internet tradizionale non è presente, perché troppo costoso o materialmente irrealizzabile. Alcune soluzioni presenti in campo, prevedono l'uso di sistemi di trasporto, che tramite collegamenti wireless, portano i dati dalle aree connesse a quelle rurali (e viceversa). Il problema qui non è legato ai collegamenti, spesso predicibili, bensì a ragioni economiche. Infatti, l'hardware necessario a fornire connettività deve essere economico, ma allo stesso tempo, resistente alle condizioni atmosferiche avverse.

Reti di sensori

Le reti di sensori, o *Wireless Sensor Networks (WSNs)*, costituiscono uno scenario sfidante dal punto di vista energetico, fortemente compatibile con il paradigma DTN. Tipicamente, i sensori sono alimentati a batteria, e di conseguenza, dotati di risorse computazionali e di memorizzazione limitate. Nel caso delle WSN statiche, i nodi sono solitamente collocati all'interno di un'area di contatto, e finché la topologia rimane invariata, è possibile calcolare i percorsi ottimali verso destinazione. Il paradigma dello *store-and-forward* si presta bene a tali scenari, in quanto consente di risparmiare energia, sospendendo i nodi, e ritardando l'inoltro dei dati al momento in cui essi entrano in contatto. Nel caso di scenari dinamici, il paradigma DTN permette di far fronte alla continua evoluzione della topologia, causata dalla scomparsa e/o rottura dei nodi.

Rete mobile

Un'architettura DTN è concepita come rete overlay, in cui i nodi possono usufruire della connettività offerta da altri e comunicare utilizzando diverse tecnologie di rete wireless. Esistono scenari in cui, sebbene ci sia una buona copertura di rete mobile, le interruzioni sono abbastanza frequenti. Inoltre, in alcuni ambienti, il segnale della rete cellulare è troppo debole, persino per effettuare una semplice chiamata telefonica. Le DTN, tramite un meccanismo di inoltro multi-hop fra i nodi, consentono di migliorare l'esperienza degli utenti che utilizzano la rete mobile, nonostante, in alcuni momenti, si verifichi la completa assenza di connessione.

2.2 Bundle Protocol

L'architettura DTN realizza una rete "overlay" al di sopra dei protocolli di comunicazione esistenti. A tal scopo, prevede l'introduzione di un nuovo livello di astrazione, il *bundle layer*, che estende lo stack di rete dei nodi partecipanti alla DTN, ponendosi tra il livello applicativo e il livello trasporto (vedi figura 2.2). L'obiettivo principale di questo layer è quello di rendere i programmi applicativi agnostici rispetto ai livelli protocollari sottostanti, favorendo la creazione di reti eterogenee. Due nodi che vogliono instaurare una comunicazione, dunque, devono interagire con il *bundle layer*, senza preoccuparsi della natura dei protocolli utilizzati nei livelli inferiori. Infatti, il *bundle layer* costituisce un vero e proprio strato di *internetworking*, incaricato dell'instradamento dei messaggi applicativi da sorgente a destinazione. Il protocollo corrispondente al *bundle layer* è il *Bundle Protocol (BP)*, sviluppato all'interno del *Delay Tolerant Networking Research Group (DTNRG)* dell'IRTF, e specificato nella RFC 5050 [1].

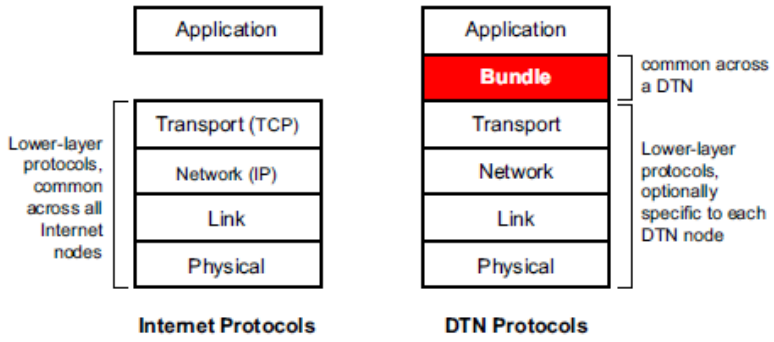


Figura 2.2: Confronto fra uno stack Internet (a sinistra) e uno stack DTN (a destra) [5]

2.2.1 Architettura

Mentre il *Delay Tolerant Networking*, fondamentalmente, descrive un meccanismo basato sul paradigma *store-and-forward*, il *Bundle Protocol* definisce il formato dei messaggi (PDU) scambiati tra le entità partecipanti alle comunicazioni di una DTN. In particolare, i messaggi previsti dal BP sono detti *bundle*, mentre i *bundle node* o semplicemente nodi, sono le entità capaci di ricevere e trasmettere tali messaggi.

Secondo le specifiche del Bundle Protocol, un *bundle node* è concettualmente costituito da tre componenti fondamentali:

- *Bundle Protocol Agent (BPA)*: fornisce i servizi del BP. Il modo con cui tali servizi sono offerti dipende dall'implementazione. Il BPA, infatti, può essere implementato in hardware, come libreria condivisa tra più nodi su una singola macchina, come processo demone con cui i nodi su una o più macchine possono interagire, tramite meccanismi di comunicazione tra processi o comunicazione di rete (tramite socket, ad esempio).

- *Convergence Layer Adapter (CLA)*: invia e riceve bundle per conto del BPA, sfruttando i servizi offerti da un qualche protocollo di comunicazione (TCP o UDP, ad esempio). Anche in questo caso, il modo in cui il CLA gestisce la trasmissione dei bundle, dipende dall'implementazione adottata.

- *Application Agent (AA)*: utilizza i servizi del BP per comunicare. L'AA è generalmente composto da due elementi, uno amministrativo e uno applicativo. L'elemento amministrativo, tipicamente integrato nell'implementazione del BPA, costruisce e richiede la trasmissione di record amministrativi (*status report* e segnali di custodia 2.2.6, ad esempio) e processa bundle amministrativi ricevuti dal nodo. L'elemento applicativo, invece, costruisce, trasmette e processa i dati applicativi veri e propri, e può essere implementato in software o in hardware. Un nodo che ha esclusivamente funzionalità di "router" (*DTN Router*), può non disporre di alcun elemento applicativo. La comunicazione tra l'elemento applicativo dell'AA e il BPA avviene tramite l'interfaccia di servizio esposta da quest'ultimo.

I principali servizi che un BPA dovrebbe fornire all'AA di un nodo, sono i seguenti:

- registrazione del nodo ad un endpoint;
- terminazione della registrazione;
- trasmissione di un bundle ad uno specifico endpoint;
- annullamento della trasmissione;
- consegna di un bundle ricevuto.

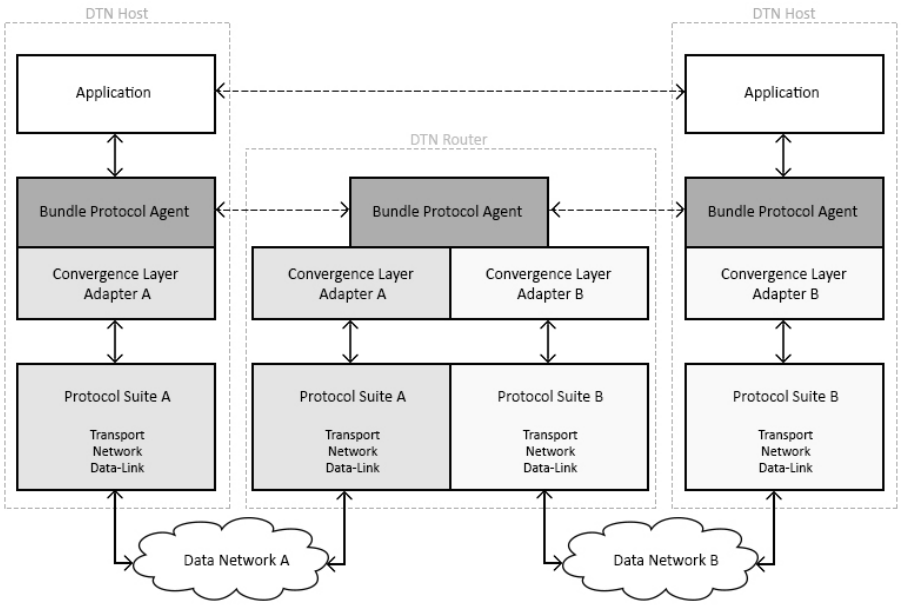


Figura 2.3: DTN Bundle Protocol su una rete eterogenea

La figura 2.3 mostra un esempio di architettura di rete eterogenea, all'interno della quale i nodi comunicano tra loro tramite l'utilizzo del Bundle Protocol. Come accennato in precedenza, il BPA in esecuzione su un nodo accede allo stack protocollare sottostante tramite un CLA, uno strato che consente di mappare le funzionalità native dei protocolli dello stack in un insieme di primitive di comunicazione, comuni a tutti i nodi partecipanti alla DTN. Lo scenario rappresentato in figura include due nodi *DTN Host*, che supportano due diverse suite di protocolli sottostanti (*Protocol Suite A* e *Protocol Suite B*), ed un nodo *DTN Router*, che le supporta entrambe. A differenza del *DTN Router*, che è privo di elemento applicativo, i *DTN Host* eseguono una o più applicazioni, che interagiscono con il BPA locale per trasmettere e/o ricevere bundle. La presenza del *DTN Router* permette alle applicazioni presenti sui *DTN Host* di poter comunicare senza dover tener conto dei protocolli effettivamente adottati per realizzare la trasmissione dati.

2.2.2 Incapsulamento

Il Bundle Protocol estende la gerarchia dell'incapsulamento realizzata dai protocolli Internet, semplicemente incapsulandoli, senza alterarne i dati. La figura 2.4 mostra un esempio di incapsulamento dei protocolli TCP/IP.

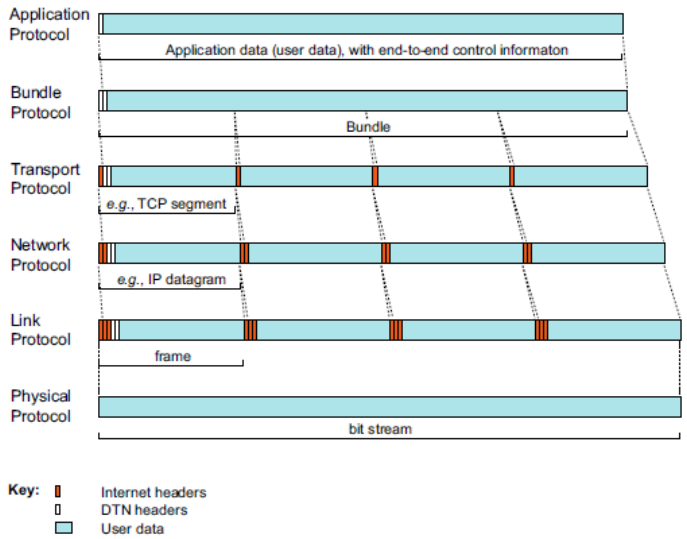


Figura 2.4: Incapsulamento dei protocolli TCP/IP nel Bundle Protocol [5]

Nel caso di bundle troppo grandi, il *bundle layer* dovrebbe in grado di suddividere i messaggi in più frammenti, in maniera abbastanza simile a come il livello IP frammenta i pacchetti. In caso di frammentazione, è compito del nodo destinazione quello di riassemblare i frammenti nell'ordine corretto, in modo da ottenere il bundle originario. Per un trattazione più dettagliata della frammentazione prevista dal Bundle Protocol, si rimanda alla Sezione 2.2.5.

2.2.3 Indirizzamento

La sorgente e la destinazione di un bundle sono identificati da un *Endpoint Identifier (EID)*. Ogni EID è conforme al formato *Uniform Resource Identifier (URI)*, ed è composto da due parti: *<scheme-name>:<scheme-specific part (SSP)>*. La lunghezza di entrambi i campi non deve eccedere i 1023 bytes. Gli schemi di rappresentazione proposti per l'EID sono molteplici, ma tipicamente, sono sempre caratterizzati da uno *scheme-specific part* suddiviso in due porzioni: la prima indicante il nodo, la seconda il *demux-token*, ovvero una singola applicazione. Uno degli schemi più diffusi è quello identificato dalla stringa `dtn`, che assume la forma `dtn://node/demux-token`. Mentre la presenza del `node` è obbligatoria, il `demux-token` può anche non esserci, come nel caso di bundle amministrativi diretti al BPA del nodo. Un EID tipicamente identifica un singolo nodo partecipante alla DTN (o meglio, una singola applicazione in esecuzione su un nodo), ma può anche rappresentare gruppi di nodi. Nel primo caso di caso si parla di EID *singleton*, nel secondo di EID *multicast*.

In combinazione con la rappresentazione degli EID, il Bundle Protocol introduce un approccio noto come *"late binding"*. La procedura di interpretare lo SSP di un EID, allo scopo di trasferire a destinazione il messaggio ad esso associato, è detta *"binding"*. Nel contesto delle reti basate su IP, si potrebbe pensare al concetto di "binding" come la risoluzione di una URI in un host. Si tratta quindi di una procedura svolta prima che il traffico venga inviato verso destinazione. Nel caso di una DTN, invece, l'EID non viene risolto una solo volta dal mittente, ma è reinterpretato ad ogni hop lungo il percorso verso la destinazione. L'uso di un approccio di questo tipo è indispensabile, per il verificarsi di casi in cui il tempo di validità di un "binding" è inferiore al tempo necessario affinché il pacchetto giunga a destinazione.

2.2.4 Formato di un bundle

Come accennato in precedenza, il Bundle Protocol incapsula i dati applicativi all'interno di messaggi di dimensione arbitraria, detti *bundle*. Un singolo bundle è costituito dalla concatenazione di due o più blocchi. Il primo blocco della sequenza, il *primary block*, contiene informazioni equivalenti a quelle di un'intestazione IP, necessarie all'instradamento del bundle verso destinazione. Il *primary block* deve essere unico e deve essere seguito da un solo blocco di payload. Inoltre, oltre al blocco di payload, il *primary block* può essere seguito da una serie di *extension block* per supportare le estensioni del protocollo, come il *Bundle Security Protocol (BSP)*. La maggior parte dei campi di un bundle hanno lunghezza variabile e utilizzano una notazione compatta detta *Self-Delimiting Numerical Values (SDNVs)*[7].

Bundle Primary Block

La figura 2.5 mostra il formato completo del *primary block* di un bundle. Il primo byte del *primary block* indica la versione del bundle, la quale assume valore fisso 0x06.

Version	Bundle processing ctrl flags (*)
Block length (*)	
Destination scheme offset (*)	Destination SSP offset (*)
Source scheme offset (*)	Source SSP offset (*)
Report-to scheme offset (*)	Report-to SSP offset (*)
Custodian scheme offset (*)	Custodian SSP offset (*)
Creation Time-stamp time (*)	
Creation Time-stamp sequence number (*)	
Life-time (*)	
Dictionary length (*)	
Dictionary byte array (variable)	
Optional: Fragment offset (*)	
Optional: Total application data unit length (*)	

(*) marked fields are encoded as SDNV

Figura 2.5: Formato del *primary block* di un bundle [6]

I *Bundle Processing Control Flags* costituiscono una stringa di bit utili al processamento del bundle, e sono suddivisi in tre categorie. I bit 0-6 specificano informazioni generali sul bundle. Essi indicano, ad esempio, se si tratta di un frammento di bundle, se il bundle è regolare o amministrativo, se l'EID destinazione è *singleton* o meno. Inoltre, tramite l'abilitazione di alcuni di questi bit, è possibile richiedere gli acknoledgment e/o il trasferimento in custodia (2.2.6) per il bundle in esame. I bit 7-13 definiscono la classe di servizio per il bundle, e di conseguenza, costituiscono informazioni utili per il suo instradamento. Ogni bundle è caratterizzato da una specifica priorità, che consente di discriminare il traffico e influenzare l'ordine con cui i bundle generati dalla stessa sorgente vengano schedulati. In base alla priorità, i bundle possono appartenere a tre diverse categorie: *bulk*, *normal*, *expedited* (priorità maggiore). Infine, i bit 14-20 permettono di richiedere degli *status report* per il bundle, ovvero dei messaggi che notificano il verificarsi di eventi legati alla vita del bundle stesso. Uno *status report* può essere richiesto in caso di ricezione, inoltro, consegna, cancellazione e accettazione della custodia di un bundle.

Il campo successivo, all'interno del *primary block*, è il *Block Length*, che indica il numero di byte rimanenti del blocco. Questa informazione può essere utile qualora si desideri saltare al blocco successivo del bundle, senza dover effettuare il parsing di tutti i campi del blocco corrente.

Dopo il *Block Length*, il *primary block* include i riferimenti a quattro diversi EID (*destination*, *source*, *report-to*, *custodian*), ognuno dei quali è codificato tramite una coppia di offset: uno per lo schema, l'altro per la SSP. Tali offset non sono altro che puntatori alle stringhe, rappresentanti gli EID, memorizzate all'interno del dizionario, una parte del *primary block* contenente un insieme di stringhe concatenate. Oltre a sorgente e destinazione, troviamo il *custodian EID*, che identifica l'ultimo nodo ad aver accettato la custodia del bundle, e il *report-to EID*, che indica il nodo a cui inviare gli *status report* per eventi che coinvolgono il bundle in esame. Nel caso di un EID non settato, il riferimento ad esso contiene il valore "dtn:none". Poiché gli EID costituiscono la maggior parte dei byte di overhead dovuti al Bundle

Protocol, il dizionario rappresenta un meccanismo per ridurre la quantità di spazio necessario alla loro memorizzazione. Ad esempio, nel caso in cui gli EID *source* e *report-to* coincidano, compariranno due riferimenti a tali EID, ma un'unica stringa all'interno del dizionario. Il dizionario è codificato con il campo *Dictionary Length*, più una serie di stringhe separate da un simbolo di terminazione, memorizzate all'interno del *Dictionary byte array*.

Un'altra informazione significativa del *primary block* è il *Creation Timestamp*, a sua volta composto da due elementi: *Creation Timestamp time* e *Creation Timestamp sequence number*. Il primo *Creation Timestamp time* indica il tempo di creazione del bundle, inteso come l'istante in cui il BPA riceve la richiesta di trasmissione, espresso come il numero di secondi trascorsi dall'inizio dell'anno 2000 nel fuso orario UTC. Il *Creation Timestamp sequence number*, invece, rappresenta l'ultimo valore di un numero positivo crescente gestito dal BPA sorgente e che può essere resettato ogni qualvolta il tempo corrente avanza di un secondo. Il *Creation Timestamp*, insieme al campo *EID source* identificano univocamente un bundle. Nel caso di un bundle frammentato, è necessario considerare il *Fragment offset* e la lunghezza del payload per distinguere i singoli frammenti.

L'ultimo campo degno di nota è il *Lifetime*, che rappresenta il tempo di vita del bundle, espresso come offset rispetto al tempo di creazione. L'uso del campo *Lifetime* consente di eliminare i bundle in eccesso all'interno della rete, in quanto, ogni volta che un nodo riceve un bundle che ha terminato il suo tempo di vita, lo scarta. Poiché sia il *Creation Timestamp* che il *Lifetime* utilizzano il tempo reale, è necessario che i nodi partecipanti alla DTN siano sincronizzati, seppure in maniera grossolana.

Altri blocchi

Block type	Block processing ctrl flags (*)
EID-reference Count (*)	
Ref. Scheme 1 (*)	Ref. SSP 1 (*)
Ref. Scheme 2 (*)	Ref. SSP 2 (*)
Block length (*)	
Block body data (variable)	

(*) marked fields are encoded as SDNV

Figura 2.6: Formato di un blocco successivo al *primary block* [6]

I blocchi successivi al *primary block* di un bundle, possono essere, o blocchi di payload o *extension block*. Tuttavia, se in ogni bundle ci deve essere obbligatoriamente un unico blocco di payload, è possibile che ci siano più *extension block*, inseribili opzionalmente, all'interno del bundle, in un ordine arbitrario. La figura 2.6 mostra la struttura dei blocchi che seguono il *primary block* all'interno un bundle. Il primo byte è costituito dal campo *Block Type*, che identifica il tipo di blocco. Nel caso di un blocco di payload, tale campo assume valore `0x01`, mentre presenta valori più alti nel caso di *extension block*. Dopo il *Block Type*, troviamo i *Block Processing Control Flags*, una maschera di bit che fornisce indicazioni su come il

blocco debba essere trattato. In particolare, oltre ad indicare che si tratti dell'ultimo blocco presente all'interno di un bundle, tali flag specificano, ad esempio, che il blocco debba essere replicato in ogni frammento associato al bundle. Inoltre, forniscono indicazioni su come comportarsi nel caso di un blocco non supportato, e dunque non processabile dal BPA. In particolare, le opzioni possibili sono: generare uno *status report*, scartare il blocco, o eliminare l'intero bundle. Nel caso di un *extension block*, è possibile che all'interno del blocco sia presente una lista di riferimenti a degli EID specifici dell'estensione (es. *Bundle Security Protocol*). Infine, il campo *Block Length* contiene la dimensione del *Block body data*, che, a sua volta, contiene i dati del blocco (tali dati saranno specifico dell'applicazione nel caso di un blocco di payload, specifici dell'estensione nel caso di un *extension block*.

2.2.5 Frammentazione

Poiché il payload di un bundle può avere una dimensione abbastanza arbitraria e potenzialmente grande, è piuttosto ragionevole pensare di suddividere un bundle in più frammenti. Il meccanismo della frammentazione ha come obiettivo quello di migliorare l'efficienza dei trasferimenti di bundle, evitando la ritrasmissione di bundle trasferiti parzialmente a causa di interruzioni della connettività. Il Bundle Protocol definisce due approcci per effettuare la frammentazione. La frammentazione *proattiva* prevede che un blocco dati sia suddiviso in frammenti, ognuno dei quali trasferito all'interno di un bundle indipendente. E' responsabilità della destinazione finale quella di riassemblare i frammenti, al fine di ottenere il blocco dati originale. La frammentazione proattiva risulta sensata nel momento in cui i volumi di dati scambiati tra i nodi sono noti o predicibili a priori. La frammentazione *reattiva*, invece, non prevede che un bundle venga suddiviso prima di essere trasmesso, ma interviene a seguito del non completo trasferimento di un bundle verso un nodo. Il nodo ricevitore trasforma il bundle parzialmente ricevuto in frammento. Il trasmettitore, una volta noto il numero di bundle ricevuti dal ricevitore, trasmetterà ad esso i restanti dati sotto forma di frammenti durante i futuri contatti, o direttamente, o passando per nodi intermedi. Il *primary block* dei frammenti di uno stesso bundle è uguale, eccetto che per i campi *Total application data unit length* e *Fragment Offset*, che indicano, rispettivamente, la lunghezza e l'offset del singolo frammento rispetto al bundle originario, secondo un meccanismo simile a quello utilizzato in IP. Inoltre, nel caso di un frammento di bundle, è abilitato un opportuno bit dei *Bundle Processing Control Flags*. Secondo quanto presente all'interno delle specifiche del Bundle Protocol, l'implementazione della frammentazione è opzionale, almeno in parte. Infatti, non è detto che un nodo debba essere in grado di generare frammenti, ma deve essere sicuramente capace di riassemblarli nell'ordine corretto, in modo da poter consegnare il contenuto del bundle originario alle applicazioni. Inoltre, sebbene di solito i frammenti sono riassemblati a destinazione, è teoricamente possibile che essi vengano riassemblati da un nodo intermedio, situato lungo il percorso verso la destinazione.

2.2.6 Affidabilità delle trasmissioni

Le architetture DTN supportano meccanismi di ritrasmissione di dati persi e/o corrotti, sia a livello dei protocolli di trasporto utilizzati per la trasmissione, che a livello di Bundle Protocol. Tuttavia, poiché le DTN presentano tipicamente un'eterogeneità nei protocolli di trasporto adottati dai nodi, è opportuno che l'affidabilità delle trasmissioni sia implementata a livello di Bundle Protocol, mediante un meccanismo di ritrasmissione da nodo a nodo, noto come *trasferimento in custodia*. Di base, quando il custode corrente di un bundle deve inoltrarlo, effettua una richiesta di trasferimento in custodia e fa partire un timer di ritrasmissione. Se il BPA del nodo ricevente decide di accettare la custodia, invia un messaggio di acknoledgement al mittente. Se non viene ricevuto alcun acknoledgement prima della scadenza del timer, il bundle viene ritrasmesso. Il valore del timer di ritrasmissione può essere distribuito ai nodi insieme alle informazioni di routing o calcolato localmente dai nodi stessi, secondo la loro esperienza passata. Il custode corrente di un bundle rappresenta, quindi, il nodo responsabile di mantenere il bundle in memoria persistente finché esso non viene ricevuto da un nuovo custode. Non è assolutamente detto che un nodo della DTN debba obbligatoriamente offrire il servizio di trasferimento in custodia. Un nodo potrebbe, ad esempio, rifiutare una richiesta di trasferimento in custodia per la mancanza di risorse disponibili, per una questione di policy e/o implementativa. Tuttavia, in un contesto in cui si voglia minimizzare il numero di perdite, sarebbe opportuno che tutti i nodi utilizzassero il trasferimento in custodia, a patto che esistano le risorse di storage necessarie e che la frequenza di generazione dei bundle non superi quella di consegna, oltre che la capacità di buffering della rete. Dunque, il meccanismo di trasferimento in custodia, combinato con l'utilizzo di storage persistente sui nodi intermedi, consente di delegare la responsabilità di trasferimenti affidabili a porzioni della rete, piuttosto che al mittente del bundle. Purtroppo, questo non è sufficiente a garantire l'affidabilità delle trasmissioni, ma solo a migliorarla. Un ulteriore passo può essere compiuto utilizzando il *return receipt*, un messaggio che conferma, al mittente, l'avvenuta consegna di un bundle a destinazione. Tuttavia, un'eccessiva quantità di bundle o frammenti di essi rischia di consumare le risorse di storage disponibili, congestionando la DTN. In caso di congestionamento, un nodo può adottare diverse strategie: eliminare dallo storage le copie di bundle che hanno terminato il loro tempo di vita, trasferire dei bundle ad altri, non accettare bundle con trasferimento in custodia, piuttosto che bundle regolari, eliminare bundle non scaduti, anche se il nodo ne è il custode. L'utilizzo di quest'ultima opzione è assolutamente sconsigliato, poiché chiaramente contraddittoria rispetto ai principi cardine delle DTN.

2.3 Routing nel Delay-Tolerant Networking

Nel contesto del *Delay-Tolerant Networking*, con il concetto di routing si intende la capacità di trasferire, o meglio, di instradare dati da una sorgente ad una destinazione. Tuttavia, il problema del routing nelle DTN è tutt'altro che analogo al problema del routing nel caso di reti IP, in cui si presuppone di operare su una

topologia costantemente connessa. Infatti, mentre nelle reti tradizionali è garantita, in ogni istante, l'esistenza di almeno un percorso end-to-end tra sorgente e destinazione, in una DTN un percorso completo potrebbe non esserci mai.

I protocolli di routing utilizzati nelle DTN devono tenere in considerazione una serie di aspetti. Innanzitutto, a seconda del contesto applicativo, è necessario capire se si è in presenza di contatti predicibili o opportunistici. In secondo luogo, è opportuno valutare se i nodi della rete sono mobili e in tal caso, se è possibile sfruttare la loro mobilità per l'instradamento del traffico. Infine, è di fondamentale importanza considerare la disponibilità di risorse all'interno della rete. Ad esempio, molti nodi, come i cellulari, dispongono di limitate capacità di archiviazione e di durata della batteria. I protocolli di routing possono utilizzare tutte queste informazioni per determinare il modo migliore con cui i messaggi devono essere instradati e memorizzati, evitando di sovraccaricare le risorse disponibili.

2.3.1 Classificazione

I protocolli di routing DTN possono essere suddivisi in due macro-categorie, a seconda del fatto che essi creino o meno delle repliche dei messaggi. I protocolli che non generano alcuna copia dei messaggi sono detti *forwarding-based*, gli altri, invece, sono considerati *replication-based*. Ogni approccio, chiaramente, presenta i propri vantaggi e svantaggi, e la soluzione più appropriata dipende soprattutto dallo scenario applicativo. I protocolli *forwarding-based* sono meno impattanti da un punto delle risorse, in quanto, in ogni istante, è presente una sola copia di un certo messaggio all'interno della rete. Di conseguenza, quando il messaggio arriva a destinazione, nessun altro nodo può possederne una copia. L'utilizzo di questo approccio elimina la necessità, da parte della destinazione, di notificare alla rete che il messaggio è stato ricevuto e che le copie "in sospeso" possono essere eliminate. Come svantaggio, i protocolli *forwarding-based* sono caratterizzati da una bassa probabilità di consegna dei messaggi. Dall'altra parte, i protocolli *replication-based* garantiscono una più alta percentuale di messaggi consegnati, oltre che un minor tempo necessario affinché essi giungano a destinazione. Tale comportamento è dovuto al fatto che esistono più copie di un singolo messaggio, ma è sufficiente che solo una di esse raggiunga la destinazione. Tuttavia, l'utilizzo delle risorse della rete è nettamente più stressante.

2.3.2 Principali algoritmi di routing

In questa sezione sono illustrati alcuni esempi di algoritmi di routing utilizzati in ambito DTN. Si tratta, chiaramente, di una trattazione non esaustiva rispetto alla grande quantità di algoritmi presenti in letteratura, ma è volta semplicemente a fornire un'idea del funzionamento di alcuni tra i principali algoritmi di routing DTN.

Direct Delivery

Il Direct Delivery rappresenta il più semplice ed intuitivo algoritmo di routing utilizzato in ambito DTN. Esso prevede che un nodo sorgente mantenga localmente un messaggio fino al contatto diretto con la destinazione. Di conseguenza, in ogni istante, esiste un'unica replica per ogni messaggio all'interno della rete. Il *Direct Delivery* non necessita di alcuna conoscenza topologica, ma richiede l'esistenza di un percorso diretto verso destinazione. Il punto debole di questo algoritmo è dovuto al fatto che, qualora la sorgente e la destinazione di un messaggio non entrino mai in contatto, il messaggio non sarà mai consegnato.

Routing epidemico

Il *routing epidemico* può essere considerato come una sorta di *flooding* migliorato, in quanto i nodi generano repliche di messaggi e le trasmettono, durante i contatti, esclusivamente a quelli che non possiedono alcuna copia del messaggio. Di base, ogni nodo possiede localmente una hash table, che tiene traccia di tutti i messaggi memorizzati su esso. Ad ogni contatto, i nodi si scambiano i cosiddetti "summary vector", dei vettori di bit che indicano i messaggi indicizzati all'interno della hash table locale. Alla ricezione di tale vettore, un nodo viene a conoscenza delle copie di messaggi possedute dal suo vicino ma non in suo possesso, e di conseguenza, ne richiede la trasmissione. Terminata questa fase di negoziazione, i nodi si scambiano solamente i messaggi richiesti. In linea teorica, considerando una dimensione infinita dei buffer di memoria, il routing epidemico consente di ottenere la più alta percentuale di consegna e il più basso tempo medio di consegna dei messaggi. Il difetto principale di questo algoritmo sta nel fatto che esso non cerca, in alcun modo, di eliminare le copie di messaggi che risultano inutili a migliorare la probabilità di consegna. Si tratta dunque di una strategia efficace nel caso di contatti opportunistici puramente casuali. Tuttavia, negli scenari reali, gli incontri opportunistici tra i nodi sono spesso governati da un qualche pattern, e di conseguenza, non sono totalmente casuali.

Spray and Wait

L'algoritmo *Spray and Wait* tenta di ottenere delle probabilità di consegna dei messaggi tipiche dei protocolli *replication-based*, sfruttando i benefici nell'utilizzo delle risorse dei protocolli *forwarding-based*. Di base, l'algoritmo consiste in due fasi: "*spray*" e "*wait*". Durante la fase di *spray*, quando un nodo genera un messaggio, ad esso viene associato un numero `L`, che indica il numero massimo di copie di tale messaggio che possono essere presenti nella rete. In questa fase, la sorgente del messaggio è responsabile di trasmettere una copia del messaggio a `L` nodi intermedi. Quando un nodo intermedio riceve la copia, entra nella fase di *wait*, durante la quale esso mantiene il messaggio finché non avviene un contatto diretto con la destinazione, e quindi, la consegna.

PRoPHET

Il *Probabilistic Routing Protocol using History of Encounters and Transitivity (PRoPHET)* è basato su un algoritmo che mira a sfruttare la non-casualità degli incontri, tenendo traccia della probabilità che la consegna di un messaggio avvenga con successo, considerando tutte le destinazioni note all'interno della DTN (*delivery predictabilities*). Secondo questo algoritmo, durante un incontro opportunistico, un nodo replica un messaggio e lo trasmette ad vicino solo se quest'ultimo possiede una maggiore probabilità (rispetto a lui stesso) di farlo giungere a destinazione. Il calcolo delle *delivery predictability* avviene mediante l'utilizzo di un algoritmo adattativo, basato sull'assunzione che, se due nodi hanno avuto un contatto opportunistico di recente, è molto probabile che essi si incontrino nuovamente in futuro. Ogni nodo M memorizza le *delivery predictability* $P(M, D)$ per ognuna delle destinazioni note D. Ad ogni contatto opportunistico, tali valori vengono ricalcolati, secondo tre regole fondamentali:

1. Quando un nodo M incontra un altro nodo E, la *delivery predictability* per E viene incrementata:

   $$P(M, E)_{new} = P(M, E)_{old} + (1 - P(M, E)_{old}) L_{encounter}$$

   dove $L_{encounter} \in [0,1]$ è una costante di inizializzazione.

2. Le probabilità per tutte le destinazioni D vengono "invecchiate":

   $$P(M, D)_{new} = P(M, D)_{old} \gamma^K,$$

   dove $\gamma \in [0,1]$ è una costante di invecchiamento e K sono le unità di tempo trascorse dall'ultima operazione di invecchiamento.

3. M ed E si scambiano le *delivery predictability* e la proprietà transitiva viene utilizzata per aggiornare le probabilità delle destinazioni D, per cui E ha il valore $P(E, D)$:

   $$P(M, D)_{new} = P(M, D)_{old} + (1 - P(M, D)_{old}) \cdot P(M, E) \cdot P(E, D) \cdot \beta,$$

   dove $\beta \in [0,1]$ è una costante di ridimensionamento.

Capitolo 3

Bluetooth

All'interno di questo capitolo vengono illustrate le principali caratteristiche dello standard Bluetooth, dapprima nella sua accezione classica, nota come Bluetooth BR/EDR o Bluetooth "classico", e successivamente nella sua versione a basso consumo energetico, meglio conosciuta come Bluetooth Low Energy (BLE). La parte finale del capitolo fornisce una breve descrizione del package BlueZ, lo stack Bluetooth ufficiale di Linux, utilizzato per la realizzazione di questo progetto di tesi.

Il contenuto del capitolo fa riferimento a [8–11].

3.1 Bluetooth

Bluetooth è uno standard di trasmissione wireless economico e a basso impatto energetico, pensato per piccole reti ad-hoc, note come *Wireless Personal Area Networks (WPANs)*. Bluetooth fornisce un meccanismo standard per la trasmissione di dati e traffico voce nel campo delle frequenze radio (RF) a corto raggio. E' attualmente integrato in dispositivi di natura diversa, quali computer, cellulari, tastiere, cuffie, stampanti, automobili, dispositivi medici e molto altro.

3.1.1 Storia dello standard

La prima versione dello standard Bluetooth è stata sviluppata da Ericsson nel luglio 1999 e in seguito formalizzata dalla Bluetooth Special Interest Group (SIG). La versione immediatamente successiva, la 1.1, si distingue dalla precedente per la risoluzione di alcuni errori emersi durante l'utilizzo della tecnologia e per un miglioramento delle prestazioni. Le specifiche 1.1 e 1.2 supportano una velocità di trasmissione massima di 1 Mbps, nota come *Basic Rate (BR)*. La specifica Bluetooth 2.0 ha introdotto il cosiddetto *Enhanced Data Rate (EDR)*, che supporta fino a 3 Mbps di velocità trasmissiva e circa 2,1 Mbps di throughput dati. Un ulteriore incremento della velocità di trasmissione è stato ottenuto con la specifica 3.0 + HS, meglio nota come Bluetooth *High Speed (HS)*, in cui è possibile raggiungere al massimo 24 Mbps. Un altro passo significativo è stato fatto con la specifica 4.0,

che ha introdotto *Bluetooth Low Energy (LE)*, una variante del Bluetooth "classico" pensata per ridurre i consumi energetici e rivolta principalmente a dispositivi alimentati a batteria, quali cellulari o sensori. Per una trattazione più dettagliata delle caratteristiche di Bluetooth LE, si consiglia di consultare la Sezione 3.2. La specifica più recente dello standard Bluetooth, la 5.0, è caratterizzata da una maggiore efficienza nell'uso dei canali di trasmissione, una migliore individuazione ed eliminazione delle interferenze, oltre che da un aumento della capacità trasmissiva a parità di consumo energetico rispetto alle versioni precedenti. E' stata concepita per rivolgersi alle nuove frontiere della connettività, che vanno dalla domotica alle applicazioni in ambito industriale. Tutte le specifiche sinora elencate sono retrocompatibili, in quanto permettono ai dispositivi conformi alle specifiche più recenti di supportare velocità trasmissive più basse, caratteristiche delle specifiche precedenti. Inoltre, i dispositivi Bluetooth conformi alle specifiche dalla 4.0 in poi possono operare in *dual mode*, supportando simultaneamente BR/EDR/HS e LE.

3.1.2 Caratteristiche tecniche

Bluetooth opera nella banda radio ISM a 2.4 GHz. Le trasmissioni si basano sul meccanismo di *Frequency Hopping Spread Spectrum (FHSS)*, che ha come principale obiettivo quello di ridurre il fenomeno delle collisioni. A tal scopo, le trasmissioni tra dispositivi Bluetooth sono spalmate su diverse frequenze. La banda è suddivisa in 79 canali radio (numerati da 0 a 78), ognuno dei quali corrispondente ad una sequenza predeterminata (in modo pseudo-casuale) di diverse frequenze, detta *sequenza di hopping*. Ogni trasmissione dati rimane su una certa frequenza per un breve periodo di tempo, in modo che, in caso di interferenza, il pacchetto dati venga ritrasmesso su una frequenza differente, potenzialmente libera da interferenze.

La tecnologia Bluetooth consente ai dispositivi di controllare il livello fisico per regolare la potenza emessa. Un parametro fondamentale in tal senso è il *Received Signal Strength Indicator (RSSI)*, utilizzato nel contesto delle tecnologie radio per misurare il livello di potenza ricevuto da un'antenna. Si tratta di un valore espresso in percentuale (più alto è il valore, più forte è il segnale), che non ha alcun legame con i parametri fisici, poiché il suo significato è stabilito dal costruttore del chip. All'interno di una rete Bluetooth, un dispositivo può leggere il valore di RSSI e richiedere agli altri dispositivi di diminuire o aumentare la potenza di trasmissione del segnale, allo scopo di ridurre il consumo energetico o consentire la rilevazione del segnale all'interno di un certo raggio di preferenza. Il parametro RSSI può essere anche utilizzato da un dispositivo ricevitore per stimare la distanza rispetto al trasmettitore, considerandone l'andamento temporale piuttosto che il valore assoluto, così da accorgersi che la sorgente del segnale si stia allontanando o avvicinando.

A seconda della distanza che i dispositivi BR/EDR possono ricoprire, si distinguono tre diverse classi. La tabella sottostante mostra il livello massimo di potenza trasmissiva e il conseguente raggio di copertura massimo di ognuna di esse.

Classe	**Potenza max**	**Raggio d'azione**
Classe 1	100 mW (20 dBm)	fino a 100 m
Classe 2	2.5 mW (4 dBm)	fino a 10 m
Classe 3	1 mW (0 dBm)	fino a 1 m

La tecnologia Bluetooth permette di realizzare reti ad-hoc, in cui dispositivi collocati in una stessa area possono comunicare senza la la presenza di un'infrastruttura di rete. Di base, quando un dispositivo Bluetooth entra nel raggio d'azione di un altro dispositivo, questi vanno a formare una rete locale, detta *piconet*. I dispositivi appartenenti alla stessa piconet possono comunicare tra loro utilizzando lo stesso canale fisico, dopo essersi sincronizzati ad un clock e ad una sequenza di hopping condivisi. I dispositivi all'interno di una piconet possono assumere due ruoli: master e slave. Il master si occupa di tutti gli aspetti riguardanti la sincronizzazione dei clock degli altri dispositivi (slave) e della sequenza di hopping. Gli slave sono le unità della piconet sincronizzate dal master. Un dispositivo Bluetooth può essere slave di più piconet, ma master solo di una. Una piconet BR/EDR può includere fino a 7 slave attivi e 255 inattivi. Una piconet LE supporta un numero illimitato di slave. Grazie alla tecnica adottata per l'accesso al canale, nota come *Time Division Multiplexing (TDM)*, un dispositivo può operare contemporaneamente come slave in una piconet e come master in un'altra piconet, favorendo la creazione di una catena di reti. Tale catena, detta *scatternet*, permette ad un vasto numero di dispositivi differenti di comunicare all'interno di area geografica estesa e in una topologia che evolve nel tempo. Il concetto di scatternet è tipico di Bluetooth BR/EDR e non è supportato dal Bluetooth LE.

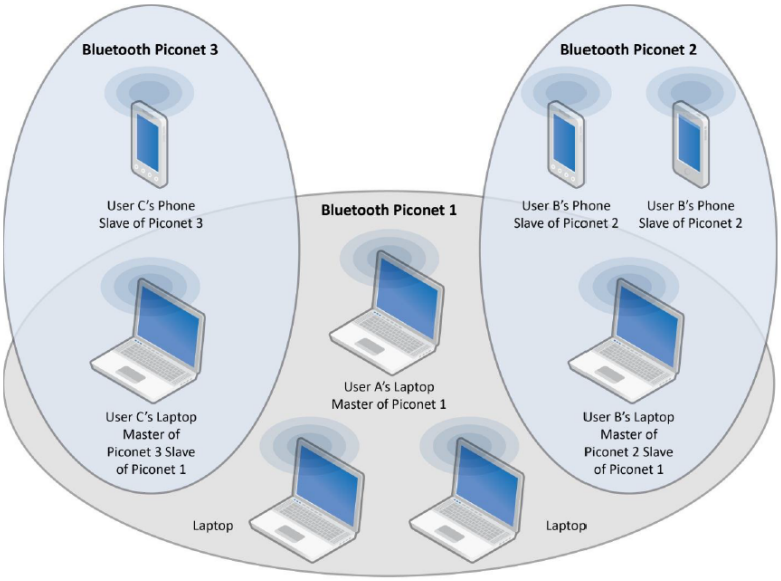


Figura 3.1: Esempio di scatternet Bluetooth (catena di piconet) [8]

Da un punto di vista operativo, un dispositivo Bluetooth può trovarsi sostanzialmente in due stati: standby o connessione. Un dispositivo in standby non è connesso con altri dispositivi e non partecipa in alcun modo alle attività di una piconet. Durante questo periodo il dispositivo risparmia energia, rimanendo in ascolto di messaggi provenienti dal master senza assorbire picchi di potenza alla pari dei dispositivi attivi.

3.1.3 Indirizzamento

Ogni chip Bluetooth è identificato da un indirizzo di 48 bit globalmente univoco, equivalente ad un indirizzo MAC Ethernet. I 24 bit più significativi costituiscono l'Organizationally Unique Identifier (OUI), assegnato dall'IEEE per identificare il costruttore del dispositivo. I restanti 24 bit sono stabiliti dal costruttore stesso per garantire l'univocità dell'indirizzo. A differenza di quanto accade nelle reti tradizionali, l'indirizzo Bluetooth è utilizzato a tutti i livelli dello stack protocollare. Nel caso di una rete Ethernet, ad esempio, l'indirizzo MAC smette di essere utilizzato una volta che si passa a livello rete, dove tipicamente si lavora con gli indirizzi IP. Il principio di comunicazione rimane lo stesso, in quanto è necessario conoscere l'indirizzo di un dispositivo per poter comunicare con esso. Oltre all'indirizzo, un dispositivo Bluetooth può essere identificato da un nome human-readable, solitamente configurabile dall'utente in maniera arbitraria. Il nome può essere non univoco, in quanto la comunicazione è comunque basata sugli indirizzi.

3.1.4 Architettura dello stack protocollare Bluetooth

Così come il modello OSI, Bluetooth specifica la sua struttura protocollare utilizzando un approccio a livelli. La figura 3.2 mostra i moduli fondamentali che compongono l'architettura Bluetooth BR/EDR (Bluetooth "classico"). Seppur mostrando una serie di analogie con il modello OSI, i livelli dello stack Bluetooth non coincidono perfettamente con quelli OSI. Inoltre, l'architettura Bluetooth presenta una serie di protocolli pensati appositamente per specifiche applicazioni (telefonia, audio, ...). Tuttavia, indipendentemente dalla natura di un'applicazione, i livelli più bassi dello stack, che nel modello OSI chiameremmo livello fisico e data-link, sono utilizzati in qualsiasi caso. Data la complessità dell'architettura protocollare Bluetooth, questa sezione si limita ad illustrare le principali caratteristiche di alcuni strati dello stack, tralasciando i dettagli relativi ai moduli di poco interesse per lo sviluppo di questa tesi.

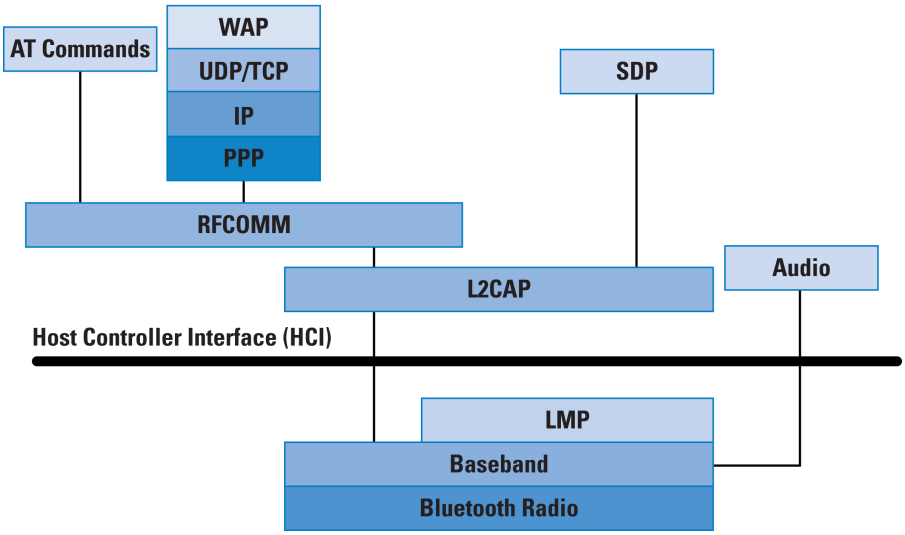


Figura 3.2: Stack protocollare Bluetooth [12]

I livelli più bassi dell'architettura Bluetooth (Radio, Baseband e LMP) costituiscono il cosiddetto *Controller*, ovvero l'hardware e le primitive Bluetooth, mentre i protocolli dei livelli superiori costituiscono l'*Host*, tipicamente implementato in software. In realtà, il fatto che un livello protocollare sia implementato in un posto piuttosto che l'altro dipende fortemente dalla tipologia di sistema preso in considerazione. In un sistema "hosted', ad esempio, i livelli più bassi dello stack sono implementati nel Controller, quelli più alti nell'Host. In un modello "embedded", invece, l'intero stack si trova nel Controller, mentre l'Host esegue una singola applicazione utente. Esistono poi i cosiddetti sistemi "fully embedded", come ad esempio le cuffie audio, dove sia lo stack che l'applicazione risiedono nel Controller. Di seguito vengono illustrate le principali funzionalità dei singoli livelli.

Livelli Radio, Baseband e LMP

Il livello più basso dello stack protocollare Bluetooth è il livello *Radio*, ovvero il livello fisico, che definisce i requisiti nel dominio della radiofrequenza (frequenze, canali, caratteristiche di trasmettitore/ricevitore).

Immediatamente sopra è situato il livello *Baseband*, che implementa i meccanismi di accesso al mezzo fisico e le procedure per cui due o più dispositivi in una piconet possano instaurare un collegamento e scambiarsi dei dati a livello fisico. Il suo funzionamento è basato sulle procedure di *Inquiry* e *Paging*, necessarie alla sincronizzazione e connessione dei dispositivi. L'Inquiry è la procedura adottata dai dispositivi Bluetooth per la scoperta dei vicini, il *Paging*, invece, è il processo di creazione della connessione tra due dispositivi appartenenti alla stessa piconet. Il livello Baseband offre due tipi di collegamento a livello fisico: *Asynchronous ConnectionLess (ACL)* e *Synchronous Connection-Oriented (SCO)*. ACL è un collegamento point-to-multipoint tra il master e gli slave di una di una piconet, che permette di trasportare pacchetti generici da/verso i livelli LMP o L2CAP, e supporta una velocità di trasmissione massima di 721 kbps. SCO, invece, modella un collegamento simmetrico punto-punto tra un master e un singolo slave della piconet e consente l'invio di inviare dati a 64 kbps. Offre un servizio sincrono e connection-oriented, rivolto a quelle comunicazioni che richiedono una banda dedicata tra due dispositivi, come ad esempio, le trasmissioni audio. Se nel caso di errore in un pacchetto ACL, tipicamente viene effettuata una ritrasmissione, i pacchetti SCO non sono mai ritrasmessi.

Il *Link Manager Protocol (LMP)* è il livello incaricato di gestire le connessioni Bluetooth, sfruttando le primitive esposte dal livello Baseband. E' responsabile di creare e configurare i collegamenti, monitorarne lo stato e distruggerli in caso di errore o di ricezione dell'apposito comando dai livelli superiori. LMP implementa i meccanismi di sicurezza, quali pairing tra i dispositivi, l'autenticazione e la cifratura. Per maggiori informazioni sulla sicurezza in Bluetooth, si rimanda alla sezione 3.3.

Host Controller Inteface (HCI)

La *Host Controller Interface (HCI)* non è un vero e proprio livello, bensì un'interfaccia tra la parte tipicamente software (Host) e la parte hardware (Controller)

dell'architettura Bluetooth. Implementa un'interfaccia basata su socket che permette di instaurare una connessione diretta con l'adapter Bluetooth locale. In tal senso, l'interfaccia HCI fornisce un meccanismo uniforme per accedere alle funzionalità dell'hardware Bluetooth, tramite l'uso di comandi ed eventi che fluiscono tra l'Host e il Controller. I comandi sono incapsulati all'interno di pacchetti *HCI Command Packet*, il cui formato è mostrato in figura 3.3. Esistono diverse categorie di comandi, a seconda dello scopo e dei moduli coinvolti nella loro esecuzione. Ad esempio, i pacchetti di tipo *HCI Link Command* forniscono all'Host la capacità di controllare le connessioni verso altri dispositivi Bluetooth. Tali comandi coinvolgono il Link Manager al fine di scambiare comandi LMP con i dispositivi remoti. Ognuno dei comandi HCI impiega una diversa quantità di tempo per essere completato. Tuttavia, al completamento di ogni comando viene scatenato un particolare evento di notifica all'Host.

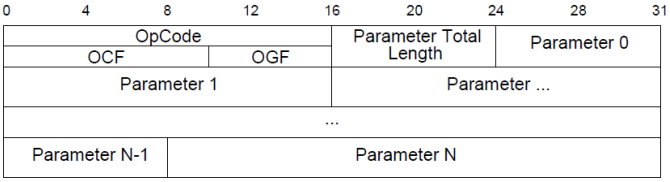


Figura 3.3: Formato di un pacchetto HCI Command Packet [9]

La gestione degli eventi è basata su notifiche asincrone utilizzate dal Controller per avvisare l'Host che qualcosa è accaduto sull'adapter. Ogni evento è racchiuso all'interno di un pacchetto di tipo *HCI Event Packet*, che contiene un identificativo dell'evento ed una serie di parametri ad esso associati (figura 3.4). Un esempio di evento è il *Connection Complete Event*, con cui il Controller notifica l'Host che una nuova connessione è stata stabilita. Tale pacchetto contiene una serie di parametri, tra cui un identificativo (handle) della connessione, l'indirizzo del dispositivo connesso, il tipo di collegamento (SCO/ACL). La lista completa di comandi ed eventi, con relativa indicazione del supporto da parte dei diversi tipi di Controller fisici, è consultabile all'interno delle specifiche Bluetooth [9].

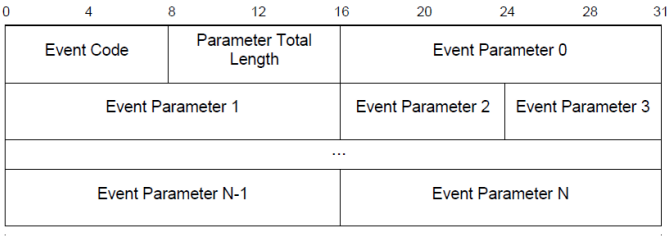


Figura 3.4: Formato di un pacchetto HCI Event Packet [9]

Il funzionamento della HCI dipende da una serie di componenti software di basso livello, come mostra la figura 3.5. In particolare, il *Firmware HCI*, collocato nell Controller, implementa i comandi per l'hardware Bluetooth accedendo ai comandi della Baseband e del LMP, ai registri hardware di stato, controllo e degli eventi. Il *Driver HCI* è invece un componente presente sull'Host, che permette ai livelli superiori dello stack di ricevere notifiche asincrone sugli eventi riguardanti la

HCI. Tra il driver e il firmware si pone l'*Host Controller Transport Layer*, che ha come obiettivo principale quello di garantire la trasparenza, intesa come capacità di trasferire dati senza avere l'effettiva conoscenza dei dati stessi. L'Host Controller Transport Layer può essere realizzato utilizzando diversi standard di trasporto, ognuno dei quali utilizza una diversa interfaccia hardware per trasferire lo stesso comando, evento o pacchetto dati. I più comuni sono USB e UART.

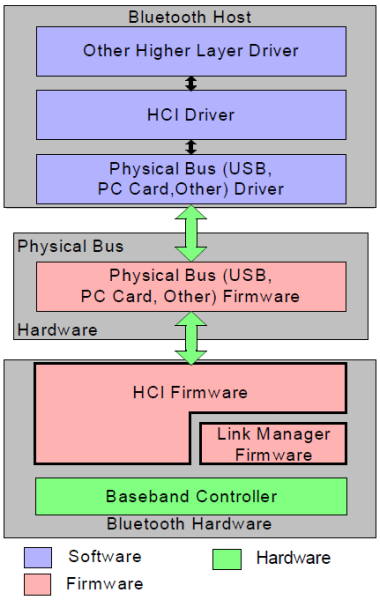


Figura 3.5: Architettura dei componenti software di basso livello

L2CAP

Il livello *Logical Link Control Adaptation Protocol (L2CAP)* fornisce un'interfaccia verso le applicazioni che utilizzano collegamenti ACL. L2CAP è incaricato di effettuare le operazioni di multiplexing/demultiplexing dei dati. In particolare, nel caso di trasmissione, L2CAP riceve i dati dai livelli superiori, li raggruppa in trame e li manda verso il livello Baseband. Nel caso di ricezione, invece, accetta trame dal livello Baseband, ne estrae i dati e li consegna al protocollo di livello superiore appropriato. Le tecniche di multiplexing/demultiplexing permettono la trasmissione di più trame su un singolo canale condiviso. Tuttavia, non bisogna confondere il canale logico L2CAP con il canale fisico di trasmissione. Un canale logico L2CAP è un canale che unisce virtualmente i livelli L2CAP residenti su due dispositivi Bluetooth differenti e può trasportare i dati per conto dei livelli superiori dello stack. L2CAP è inoltre responsabile della segmentazione e riassemblaggio dei dati. Poiché i pacchetti provenienti dai livelli superiori devono essere opportunamente ridimensionati per poter attraversare l'hardware, L2CAP si occupa di creare pacchetti con un payload conforme alle specifiche dell'interfaccia HCI. In tal modo, i protocolli degli strati superiori possono generare dati in maniera arbitraria, senza doversi preoccupare della loro futura trasmissione. Di base, L2CAP lavora su trame con payload configurabile fino ad un massimo di 64 kB e con MTU minima supportata di 48 byte. L2CAP può operare sia in modalità base, senza offrire alcun meccanismo di affidabilità, sia realizzando ritrasmissione e/o controllo di flusso.

RFCOMM

Radio Frequency Communication (RFCOMM) è un protocollo di trasporto, operante su L2CAP, nato per emulare la comunicazione su porte seriali, allo scopo di favorire la migrazione delle applicazioni esistenti alla nascita di Bluetooth. Il protocollo RFCOMM consente la creazione di più connessioni simultanee tra dispositivi, affidando a L2CAP il compito di multiplarle su una singola connessione. RFCOMM, inoltre, si affida al livello Baseband di Bluetooth per garantire la consegna in ordine dei byte trasmessi sul canale di comunicazione.

RFCOMM implementa un protocollo di trasmissione *connection-oriented* affidabile, in maniera simile al TCP nella suite di protocolli Internet. E' concepito principalmente per quelle applicazioni che vogliono instaurare connessioni punto-punto su cui scambiare flussi di byte con una serie di garanzie di affidabilità. Molte applicazioni Bluetooth sono basate sul protocollo RFCOMM, in quanto esso è supportato e disponibile in gran parte dei sistemi operativi. Inoltre, dal punto di vista dello sviluppo software, scrivere applicazioni che operino su TCP piuttosto che su RFCOMM è davvero molto simile. La differenza sostanziale sta nella scelta dei numeri di porta. Nel protocollo RFCOMM non si parla propriamente di porte, ma di canali, anche se concettualmente i due termini hanno lo stesso significato. Tuttavia, mentre TCP permette di aprire fino a 65535 porte sulla stessa macchina, RFCOMM supporta l'utilizzo di un massimo di 30 canali. In RFCOMM, ogni canale è identificato da un *Data Link Connection Identifier (DLCI)*.

La comunicazione RFCOMM avviene mediante lo scambio di trame, incapsulate all'interno del payload dei messaggi L2CAP. Ne esistono fondamentalmente cinque tipologie:

- *SABM - Start Asynchronous Balanced Mode* (creazione di una connessione)
- *UA - Unnumbered Acknowledgement* (risposta di accettazione di una connessione)
- *DISC - Disconnect* (chiusura di una connessione)
- *DM - Disconnected Mode* (risposta di rifiuto di una connessione)
- *UIH - Unnumbered Information with Header check*

SABM, UA, DM e DISC rappresentano trame di controllo di "basso livello". Invece, le trame UIH trasmesse sul canale con DLCI=0 sono utilizzate per inviare messaggi di controllo, quelle con DLCI diverso da 0 per la trasmissione dati.

SDP

Il *Service Discovery Protocol (SDP)* implementa un meccanismo per la registrazione, pubblicazione, annuncio e scoperta dei servizi da parte di un dispositivo Bluetooth. Permette di scoprire i servizi offerti da altri dispositivi e ottenere i parametri necessari per usufruire di tali servizi. Ogni servizio è identificato univocamente da una costante numerica chiamata UUID (Universally Unique Identifier).

I dispositivi descrivono l'interfaccia dei servizi che supportano, elencando anche i protocolli che erogano ciascun servizio. SDP, inoltre, consente la definizione di servizi non nativi di Bluetooth.

Altri protocolli

Rientrano in questa sezione i cosiddetti *adopted protocols*, ovvero protocolli definiti da altre organizzazioni di standardizzazione e successivamente incorporati all'architettura Bluetooth. Alcuni esempi sono PPP (Point-to-Point Protocol), TCP/UDP-IP, OBEX (protocollo per lo scambio di oggetti, simile ad HTTP) e WAP. Tali protocolli sono concepiti per operare sullo standard RFCOMM, e favoriscono il riutilizzo delle applicazioni esistenti, senza la necessità di riscrivere il software da zero.

3.2 Bluetooth Low Energy

Il Bluetooth Low Energy (BLE), noto anche come Bluetooth Smart, è una tecnologia di comunicazione wireless progettata e commercializzata da Bluetooth Special Interest Group (SIG) allo scopo di ridurre il consumo energetico, la complessità e i costi del Bluetooth classico, mantenendo però un raggio d'azione simile. Il BLE è stato introdotto con la specifica Bluetooth 4.0 (2010) e ha avuto una rapida diffusione grazie alla presenza sempre maggiore di dispositivi con limitate risorse hardware, quali smartphone o tablet, o più in generale di sistemi embeddded.

3.2.1 Panoramica del Bluetooth Low Energy

Per ottenere un significativo risparmio da un punto di vista energetico, le scelte progettuali alla base di Bluetooth Low Energy sono mirate a semplificare drasticamente l'implementazione dell'architettura protocollare Bluetooth. Tale semplificazione si traduce in una serie di differenze tra Bluetooh classico e BLE, che rendono le due tecnologie incompatibili. Tuttavia, il Bluetooth LE e quello classico presentano alcuni tratti comuni. Innanzitutto, il BLE mantiene la stessa separazione tra Host e Controller fatta dal Bluetooth classico. Inoltre, da un punto di vista del livello fisico, anche il BLE opera a 2.4 GHz nella banda ISM. Tuttavia, BLE utilizza un sistema di modulazione più semplice, un numero e una distribuzione differente dei canali, oltre che un proprio formato di dati e una propria modalità di gestione delle connessioni e del flusso dati. Il *Link Layer (LL)* di BLE è l'equivalente di LMP in Bluetooth classico, ma più semplice. La sicurezza, che in Bluetooth classico è annidata nel LMP, in BLE è implementata da un componente dedicato, detto *Security Manager*. Sopra la HCI e L2CAP, comuni al Bluetooth classico, BLE definisce dei suoi livelli protocollari. L'*Attribute Protocol (ATT)* consente ai dispositivi di esporre degli attributi, ovvero dei dati accessibili atomicamente in lettura/scrittura. Il *Generic Access Profile (GAP)* definisce il modo con cui due dispositivi sono capaci di scoprirsi a vicenda e stabilire delle connessioni. Infine, il *Generic Attribute Profile (GATT)*, fornisce una descrizione dei servizi in BLE. Definisce come gli attributi ATT possono essere raggruppati per formare dei servizi.

Il Bluetooth LE definisce quattro ruoli per i dispositivi:

- *Advertiser*: genera periodicamente messaggi di advertising per annunciare la sua presenza, ma non accetta connessioni in ingresso.
- *Observer*: è il duale di un Advertiser. Rimane in ascolto passivo di dispositivi BLE nel suo raggio d'azione e processa i messaggi di advertising che riceve.
- *Peripheral*: è un dispositivo caratterizzato da basso consumo, assimilabile ad uno "slave" del Bluetooth classico.
- *Central*: è responsabile di iniziare e gestire la comunicazione, in modo paragonabile ad un "master" del Bluetooth classico.

3.2.2 Bluetooth Low Energy Discovery

Una delle novità più significative di Bluetooth LE risiede nel meccanismo di scoperta dei dispositivi. Infatti, mentre nel Bluetooth classico la scoperta è basata sulla procedura di Inquiry, in cui il dispositivo che effettua la scansione manda un messaggio in broadcast e i dispositivi *discoverable* rispondono, in Bluetooth LE, i dispositivi emettono periodicamente dei messaggi di advertising per annunciare la propria presenza. Tali pacchetti contengono tutte le informazioni necessarie per identificare il mittente e instaurare una connessione. La seguente sezione descrive dettagliatamente le procedure di *advertising* e *Scanning*, necessarie all'espletamento della discovery in Bluetooth LE.

Bluetooth LE advertising

Nel contesto delle tecnologie wireless, si sente spesso parlare di *beaconing*, inteso come meccanismo di diffusione di piccoli frammenti di informazione. Nel Bluetooth Low Energy questo concetto è messo in atto tramite il meccanismo di advertising, che permette ad un dispositivo Bluetooth di diffondere informazioni senza dover instaurare alcuna connessione, oppure di farsi scoprire da altri dispositivi nel suo raggio d'azione, il tutto con un minimo consumo di energia. I messaggi di advertising sono inviati all'interno di eventi di advertising, che posso essere suddivisi in due macro-categorie: diretti e indiretti. L'advertising indiretto è inviato in broadcast e permette di scoprire dispositivi non ancora noti, quello diretto è indirizzato ad uno specifico dispositivo (scanner) al fine di instaurare una connessione.

La figura 3.6 illustra il diagramma di funzionamento di un dispositivo *advertiser*, mostrando i parametri principali della procedura di advertising. Innazitutto, è possibile notare come soltanto 3 dei 40 canali utilizzati da Bluetooth LE siano dedicati all'advertising: 37, 38 e 39. Quando un dispositivo entra in modalità advertising, invia periodicamente pacchetti di advertising su ognuno dei 3 canali, uno dopo l'altro. In teoria, è possibile selezionare solamente uno o due canali, ma in pratica è fortemente sconsigliato, poiché, sebbene trasmettere gli advertisement su meno canali permetta di risparmiare energia, si rischia di generare il fenomeno dell'interferenza. Infatti, nel caso in cui venga selezionato un unico canale e questo

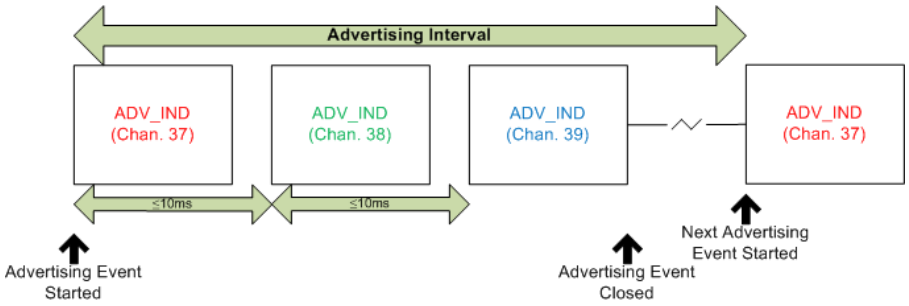


Figura 3.6: Diagramma temporale della procedura di advertising [10]

risulti occupato, il dispositivo non potrà funzionare correttamente. La figura L'intervallo di tempo che intercorre tra due eventi di advertising è detto *advertising Interval* ed è costituito da una componente fissa e una pseudo-casuale. La componente fissa è configurabile via software con valori che vanno da 20 ms a 10.24 s, a multipli di 0.625 ms . E' importante scegliere accuratamente la durata di tale intervallo, cercando il giusto compresso tra consumo energetico e latenza, intesa come periodo di tempo necessario affinché un advertisement venga rilevato. La componente pseudo-casuale è un ritardo tra 0 e 10 ms inserito automaticamente dall'adapter al fine di ridurre la probabilità di collisione tra gli advertisement di dispositivi differenti, incrementando cosi' la robustezza di Bluetooth.

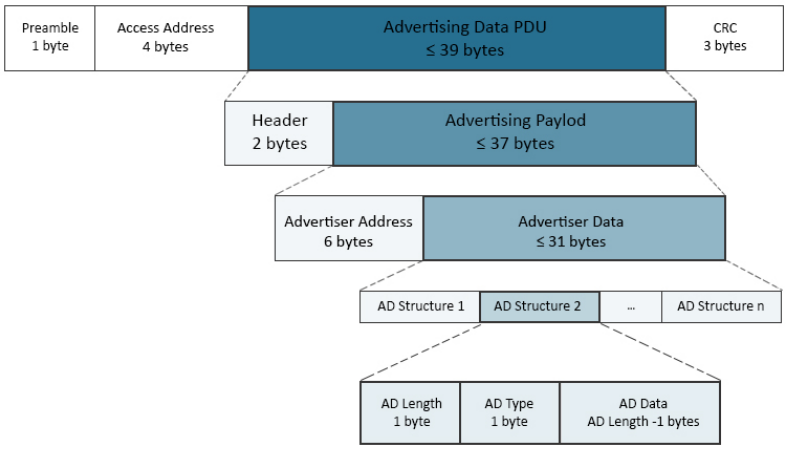


Figura 3.7: Formato di un messaggio di advertising

Come si può notare in figura 3.7, un pacchetto di advertising è composto essenzialmente da 4 parti: *Preamble* (1 byte), *Access Address* (4 byte), *advertising Data PDU* (≤39 byte) e *CRC* (3 byte). Nel caso dei messaggi di advertising, l'*Access Address*, così come il *Preamble*, assumono un valore prefissato. L'*advertising Data PDU* è composto da una parte di *Header* (2 byte) e una di *Payload* (≤37 byte). Il *Payload*, a sua volta, contiene l'indirizzo MAC Bluetooth del dispositivo advertiser (6 byte) ed una serie di dati definiti dall'utente (≤31 byte). La parte dati è costituita da una sequenza di *AD structure*, ognuna delle quali prevede un campo *AD Length* (1 byte) e un campo *AD Type* (1 byte), che indicano rispettivamente la lunghezza e il tipo di informazioni contenute nei restanti byte, che invece costituiscono l'*AD Data*. Alcuni esempi comuni di *AD Type* sono *Complete Local Name* (`0x09`), *Incomplete List of 16-bit Service Class UUID* (`0x02`), *Service Data* (`0x16`)

e *Manufacturer Specific Data* (`0xFF`). La lista completa dei valori possibili e il loro significato sono specificati nel *Bluetooth Core Specification Supplement (CSS)* [9]. Un grande vantaggio per gli sviluppatori risiede nella possibilità di personalizzare l'AD Data dei messaggi di advertising a seconda delle proprie esigenze.

Bluetooth LE Scanning

Il processo duale all'advertising è lo scanning, meccanismo che permette ad un dispositivo di rimanere in ascolto di messaggi di advertising provenienti da altri dispositivi nel suo vicinato. Per questo motivo, il processo di scanning è noto anche con il nome di *device discovery*. Lo scanning può essere attivo o passivo. In entrambi i casi il dispositivo è capace di ricevere dati di advertising, ma nel caso di uno scanner attivo, a fronte della ricezione di un advertising, questo può richiedere informazioni aggiuntive al dispositivo advertiser.

La figura 3.8 mostra i parametri fondamentali coinvolti nel processo di scanning. La *Scan Window* indica il periodo di tempo in cui il dispositivo rimane in ascolto su un singolo canale. Lo *Scan Interval* rappresenta l'intervallo di tempo tra l'inizio di due *Scan Window* consecutive. Entrambi i parametri possono assumere i valori da 10 ms a 10.24 s. Il tempo totale per cui il dispositivo rimane in stato di scanning è detto *Scan Duration*. I canali su cui il dispositivo si mette in ascolto sono gli stessi utilizzati per l'advertising, e il loro ordine non è modificabile.

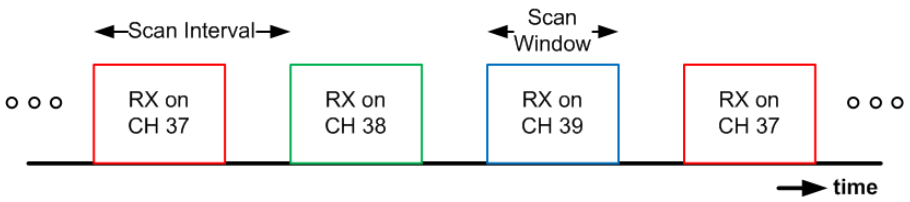


Figura 3.8: Diagramma temporale della procedura di Scanning [10]

3.3 Sicurezza in Bluetooth

In ogni tecnologia di rete wireless, e di conseguenza anche in Bluetooth, la sicurezza costituisce un aspetto da non sottovalutare. Questa sezione offre una breve descrizione del principale strumento di sicurezza offerto dal Bluetooth moderno, il *Secure Simple Pairing (SSP)*. I concetti in seguito esposti trovano riscontro sia nel Bluetooth classico che nella sua variante Low Energy.

Parlando di sicurezza in Bluetooth, non si può non menzionare il pairing, il meccanismo utilizzato dai dispositivi Bluetooth per realizzare comunicazioni sicure. Il processo di pairing è finalizzato alla creazione di un segreto condiviso tra due dispositivi, la *Link Key*. La Link Key è una chiave usata dai dispositivi per autenticarsi e, anche se non direttamente, per cifrare/decifrare i dati che essi scambiano. Prima della specifica Bluetooth 2.1, il pairing dei dispositivi avveniva secondo una procedura meglio nota come *PIN/Legacy Pairing*. Secondo tale meccanismo, i due

dispositivi coinvolti dovevano derivare la Link Key nel momento in cui veniva inserito un codice PIN su uno o ciascuno di essi, a seconda della configurazione e dalla tipologia dei dispositivi. La principale fonte di insicurezza di questo meccanismo era dovuta al fatto che il PIN fosse trasmesso in chiaro e utilizzato direttamente per la derivazione della Link Key. Dunque, una volta noto il codice PIN, un attaccante avrebbe potuto derivare la Link Key in maniera piuttosto semplice. A causa dei suoi problemi di sicurezza, il meccanismo di PIN/Legacy Pairing era soggetto ad una serie di attacchi noti. Per questo motivo, la specifica Bluetooth 2.1 + EDR ha introdotto il *Secure Simple Pairing (SSP)*, un meccanismo volto a semplificare il pairing lato utente e ad aumentare il livello di sicurezza rispetto al metodo precedente.

Il Secure Simple Pairing (SSP) è stato introdotto in Bluetooth 2.1 e successivamente esteso ai dispositivi Bluetooth LE con la specifica 4.0. Il Bluetooth SSP è basato sulla crittografia a curva ellitica, e in particolare sull'algoritmo *Elliptic-Curve Diffie-Hellman (ECDH)* con curva P-192. La specifica Bluetooth 4.1 ha introdotto la funzionalità *Secure Connection*, che aggiunge un ulteriore livello di sicurezza grazie all'utilizzo di una curva ellittica P-256. Il principale punto di forza del SSP è quello di non trasmettere in chiaro alcuna informazione critica ai fini della sicurezza. Infatti, a differenza del PIN/Legacy Pairing, il SSP non utilizza nessun codice PIN per la derivazione della Link Key, bensì dei numeri pseudo-casuali molto grandi. La figura 3.9 ne illustra le fasi in maniera dettagliata.

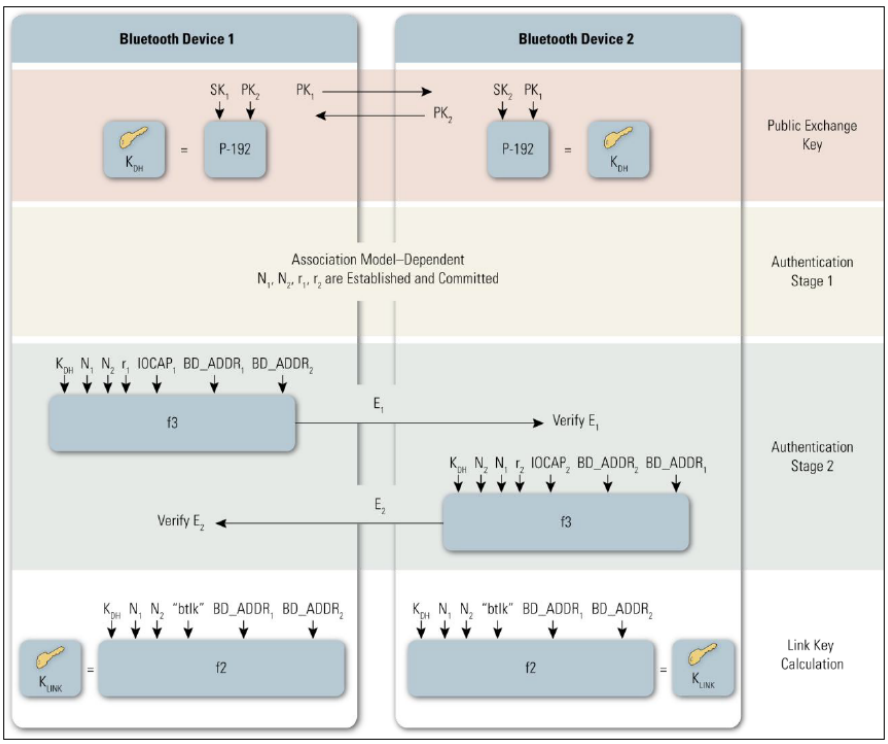


Figura 3.9: Generazione della Link Key nel processo di Secure Simple Pairing (SSP) [8]

Prima di tutto, i dispositivi coinvolti nella procedura di SSP devono possedere una chiave pubblica e una chiave privata. Durante la fase iniziale, i dispositivi

scambiano le proprie capacità di I/O, al fine di determinare il metodo di associazione da adottare. Il SSP definisce quattro diversi metodi:

- *Numeric Comparison* è ideale per i dispositivi dotati di capacità di output. Sui display dei dispositivi coinvolti viene mostrato un numero a 6 cifre e viene chiesto all'utente di confermare che i numeri coincidano.
- *Passkey Entry* è adatto ai dispositivi con sole capacità di input, come le tastiere, in quanto richiede l'inserimento di un numero di 6 cifre.
- *Just Works* è pensato per quelle situazioni in cui almeno uno dei dispositivi non ha alcuna capacità di I/O. Richiede di accettare la connessione senza effettuare alcun confronto. Non fornisce protezione da attacchi di tipo MITM.
- *Out of Band (OOB)* è basato sull'utilizzo di un ulteriore canale wireless o wired (es. NFC) per lo scambio delle informazioni crittografiche. Ad esempio, nel caso di NFC, per effettuare il pairing, è sufficiente mettere i dispositivi a contatto e confermare tramite la pressione di un tasto.

Sia nel caso di Numeric Comparison, che di Passkey Entry, il numero visualizzato o introdotto da tastiera non viene utilizzato per il calcolo dalla Link Key. Di conseguenza, qualora un attaccante passivo riuscisse a risalire a tale numero, questo risulterebbe del tutto inutile.

La seconda fase consiste nello scambio (in chiaro) delle chiavi pubbliche, che dovranno essere generate ogni volta che sarà necessario eseguire un pairing. In Bluetooth BR/EDR la generazione delle chiavi avviene nel Controller, in Bluetooth LE tale compito è affidato ad un componente detto *Security Manager*. Le chiavi pubbliche sono utilizzate per generare un segreto condiviso tra i due dispositivi, un numero casuale di 192-bit noto come *Diffie-Hellman Key (DHKey)*. La fase successiva della procedura è l'autenticazione dei dispositivi, a sua volta suddivisa in due parti. La prima dipende dal metodo di associazione adottato e consiste nello scambio di alcune informazioni utili alla seconda parte, durante la quale i dispositivi calcolano un valore comune combinando tali informazioni con la DHKey. Il valore ottenuto dovrà essere confermato da ambe le parti affinché l'autenticazione si concluda con successo. La procedura di SSP termina con il calcolo della Link Key, che sarà la stessa in entrambi i dispositivi. Quest'ultima potrà essere utilizzata sia per l'autenticazione che per derivare la *Encryption Key*, necessaria alla cifratura/decifratura dei dati in transito tra i due dispositivi. Terminato il processo di pairing, il dispositivo che ha iniziato la procedura diventerà master della piconet, il dispositivo "scoperto" diventerà slave, e potrà iniziare il flusso dati.

3.4 Bluetooth in Linux: BlueZ

Questa sezione illustra brevemente il package *BlueZ*, che implementa lo stack protocollare Bluetooth nei sistemi Linux, e fornisce agli sviluppatori di software dei moduli di libreria per poter interagire con esso.

Attualmente presente di default nelle versioni recenti del kernel, BlueZ rappresenta di fatto lo stack Bluetooth ufficiale di Linux. Si tratta di un progetto open source, distribuito con licenza GPL (GNU General Public License). Lo stack BlueZ è costituito da alcuni moduli nel kernel e alcuni moduli nello spazio utente. I moduli annidati nel kernel sono responsabili di fornire i driver per l'interfacciamento con l'hardware, l'implementazione dei livelli protocollari più bassi e le funzionalità di sicurezza. Per quanto riguardo lo spazio utente, invece, il componente centrale è *bluetoothd*, un processo demone che implementa le funzionalità principali da esporre alle applicazioni, nascondendo i dettagli di basso livello allo sviluppatore.

BlueZ include, inoltre, una libreria scritta in C, *libbluetooth*, che definisce, ad esempio, le strutture dati per la rappresentazione degli indirizzi Bluetooth ed una serie di funzioni utili per poter operare su essi (confronto, copia, conversione, ...). La libreria fornisce, inoltre, una HCI API, che consente di instaurare una connessione via socket con gli adapter Bluetooth locali, al fine di inviare comandi generici e mettersi di ascolto di eventi che coinvolgono gli adapter stessi. La libreria offre anche supporto all'implementazione di socket su RFCOMM, mediante la definizione di strutture dati atte a questo scopo. Oltre a *libbluetooth*, il package BlueZ include una serie di strumenti a linea di comando utili in fase di sviluppo, per il debugging ed il tracing. Tramite questi tool è possibile, ad esempio, interagire con gli adapter Bluetooth locali, abilitare/disabilitare funzionalità specifiche, aprire e ricevere connessioni, e molto altro.

Non esistendo alcuna documentazione ufficiale a riguardo, quanto realizzato nel progetto di tesi è frutto dell'ispezione del codice sorgente relativo alla libreria e ai tool integrati all'interno di BlueZ. Infine, una piccola nota riguardante il supporto di Bluetooth LE. In pratica, per poter usufruire delle funzionalità di BLE in Linux, è necessario installare una versione di BlueZ uguale o successiva alla 5.1, oltre che a disporre di un'interfaccia hardware compatibile con le specifiche Bluetooth 4.0 o superiori.

Capitolo 4

Architettura di IBR-DTN

All'interno di questo capitolo vengono illustrati i moduli essenziali dell'architettura di IBR-DTN, l'implementazione del Bundle Protocol [1] utilizzata per lo sviluppo di questa tesi. Il contenuto del capitolo fa riferimento, oltre che ad un'ispezione del codice sorgente di IBR-DTN, a [6, 13]. Per una guida dettagliata all'installazione e l'utilizzo del software IBR-DTN, si rimanda all'appendice A.

Realizzata all'interno dell'Università Tecnica di Braunschweig (Germania), IBR-DTN è un'implementazione open-source leggera, efficiente e modulare, concepita principalmente per poter essere eseguita su dispositivi embedded con limitate risorse hardware, ma più in generale su sistemi Linux standard. Oltre alla versione per i sistemi operativi classici, sviluppata in C++, ne esiste un'implementazione per piattaforme Android. Di base, il Bundle Protocol Agent (BPA) è implementato come processo demone ed espone una API basata su socket, con cui le applicazioni possono interagire per usufruire delle funzionalità del Bundle Protocol.

4.1 Componenti fondamentali

I componenti dell'architettura IBR-DTN si suddividono in due categorie: statici e dinamici. I componenti dinamici forniscono le funzionalità definite da una specifica interfaccia, ma il modo in cui lo fanno dipende dall'implementazione. Alcuni di essi devono essere obbligatoriamente presenti, in quanto realizzano funzionalità di base, come lo storage, altri invece sono opzionali (i *convergence layer*, ad esempio). Al contrario, i componenti statici devono essere eseguiti sempre, e non sono pensati per essere rimpiazzati da implementazioni differenti.

I componenti di IBR-DTN sono suddivisibili in 5 gruppi, a seconda delle funzionalità che implementano: core, storage, network, routing e API (figura 4.1).

4.1.1 Core

Il componente principale del core è il modulo `Bundle Core`, il quale, a seconda della configurazione, istanzia e gestisce i vari moduli che compongono l'architettura, permettendo ad essi di comunicare tra loro. L'interazione tra i diversi moduli

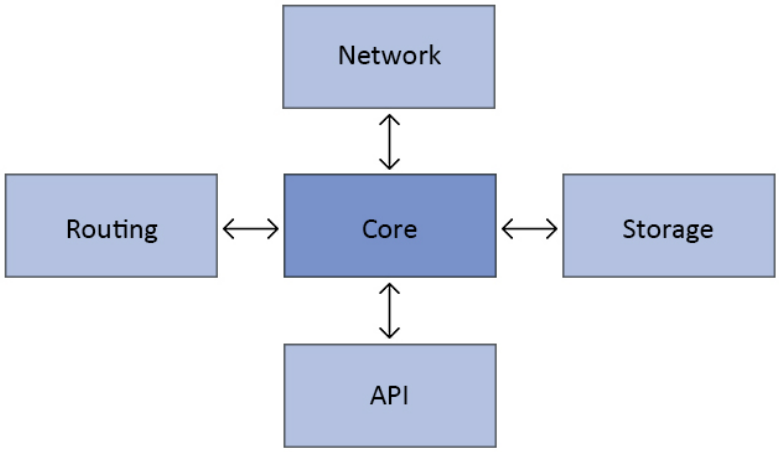


Figura 4.1: Visione di alto livello dell'architettura IBR-DTN

può avvenire o tramite chiamate sincrone, oppure mediante un meccanismo basato su eventi. In tal senso, all'interno del core IBR-DTN, è presente il modulo `Event Switch`, incaricato di affidare la gestione dei singoli eventi ai componenti interessati. Oltre a ciò, il core offre il modulo `Wall Clock`, che genera un evento al secondo per fornire un clock centralizzato a tutti i componenti che necessitano di una sincronizzazione temporale.

4.1.2 Storage

Poiché le architetture DTN sono basate sul paradigma *store-and-forward*, un qualsiasi *bundle node* deve essere capace di memorizzare bundle per un certo periodo di tempo. Per questo motivo, l'architettura IBR-DTN include un sottosistema di storage, atto a gestire la memorizzazione dei bundle. Lo storage è un componente obbligatorio ma dinamico, in quanto può essere implementato in modi differenti. Attualmente, l'implementazione di IBR-DTN la memorizzazione di bundle in memoria RAM, su disco (file-system) e su basi di dati SQLite. Tuttavia, a prescindere dall'implementazione, un modulo di storage deve fornire le primitive per la lettura, cancellazione e memorizzazione dei bundle. Per ognuna di queste operazioni, un bundle è identificato da un identificativo univoco. Inoltre, qualora non fosse possibile identificare univocamente un bundle, un modulo di storage, può, facoltativamente, offrire un meccanismo per selezionare i bundle che fanno match con un certo criterio.

4.1.3 Network

Un *bundle node* che non possa essere interconnesso con altri nodi è piuttosto inutile. I componenti appartenenti al gruppo network implementano le funzionalità di comunicazione, ovvero le diverse implementazioni di *convergence layer* e di *discovery agent*, i moduli che realizzano la scoperta del vicinato. Ciascun *convergence layer* deve fornire un'interfaccia per la ricezione e l'invio di bundle su una specifica tecnologia di rete. Il loro scopo è quello di definire un'astrazione rispetto alla

rete sottostante, nascondendo alle applicazioni i dettagli della comunicazione. I *convergence layer* sono componenti dinamici opzionali, poiché la loro esecuzione deve essere esplicitamente abilitata in configurazione. Il modulo incaricato della loro gestione è il `Connection Manager`. L'implementazione attuale di IBR-DTN supporta i seguenti *convergence layer* e *discovery agent*:

- *TCP/IP convergence layer* (RFC 7242 [14])
- *TLS extension for TCP convergence layer*
- *UDP/IP convergence layer* (draft-irtf-dtnrg-udp-clayer-00 [15])
- *HTTP convergence layer*
- *IEEE 802.15.4 LoWPAN convergence layer*
- *Generic datagram convergence layer* con supporto a IEEE 802.15.4 e UDP
- *IP neighbor discovery* (draft-irtf-dtnrg-ipnd-01 [16])

4.1.4 Routing

I componenti di routing sono incaricati di accodare i bundle che devono essere trasferiti ad altri nodi, nel momento in cui si verifica un nuovo contatto. Il componente principale è il `Base Router`, che si occupa di gestire diversi moduli di routing. Ognuno di essi implementa uno specifico algoritmo di routing DTN (statico, epidemico e PRoPHET) e può essere agganciato al `Base Router` come una sorta di plugin. Tutti i moduli di routing sono notificati dai moduli di scoperta del vicinato, a seguito della comparsa/scomparsa di un nuovo vicino, e dal sistema di storage, nel momento in cui un nuovo bundle giunge al processo demone. In quest'ultimo caso, il modulo di routing che si riterrà responsabile dell'inoltro del bundle, contatterà il `Connection Manager` per attivare il convergence layer opportuno a realizzare il trasferimento. Oltre ai moduli che implementato gli algoritmi di routing, questo gruppo contiene una serie di componenti per gestire le ritrasmissioni, per scambiare i *summary vector* e altre informazioni di routing tra i nodi, e algoritmi per lo scheduling dei bundle.

4.1.5 Application Programming Interface (API)

L'ultima categoria di componenti include i componenti relativi alla API. La API fornisce un'interfaccia con cui le applicazioni possono creare e cancellare registrazioni, oltre che ricevere e trasmettere bundle. Oltre a queste funzionalità, la API fornisce una serie di funzioni di gestione, che permettono di ottenere informazioni sui vicini del nodo locale o sui bundle memorizzati su esso. Di default la API è disponibile alla porta TCP 4550 in formato testuale e binario. Per una panoramica completa sulle funzionalità esposte dalla API di IBR-DTN, si faccia riferimento alla documentazione [17].

La figura 4.2 riassume quanto detto finora e illustra, in maniera dettagliata, i moduli principali che compongono l'architettura IBR-DTN e le interazioni tra essi.

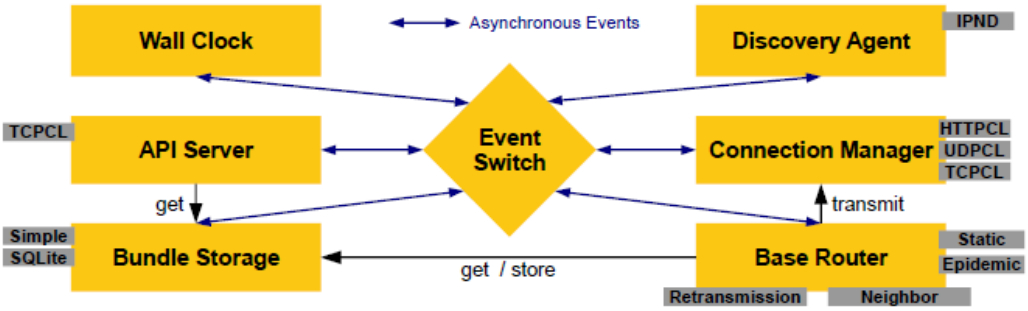


Figura 4.2: Componenti fondamentali dell'architettura IBR-DTN [13]

4.2 Struttura del codice

Uno degli aspetti caratteristici di IBR-DTN è sicuramente la portabilità del codice, ottenuta tenendo conto di due aspetti essenziali: evitare librerie esterne per implementare le funzionalità di base e fornire una versione minimale del software che possa essere compilata senza librerie aggiuntive. Qualora si vogliano integrare delle funzionalità extra, che richiedono l'uso di librerie specifiche, è preferibile che esse siano ampiamente diffuse e supportate dai sistemi operativi, o in alternativa, è opportuno sviluppare un livello di astrazione per quelle funzionalità platform-dependent.

Da un punto di vista del codice, la suite IBR-DTN è organizzato in diversi package, come mostra la figura 4.3.

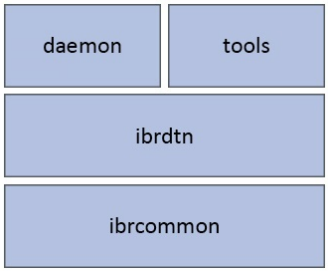


Figura 4.3: Struttura del codice in IBR-DTN

- `ibrcommon`: libreria che implementa le primitive per gestione di file, comunicazione in rete, concorrenza e sincronizzazione, monitoraggio dei collegamenti, logging ed altro. Le funzionalità fornite da tale libreria non sono direttamente legate al Delay-Tolerant Networking e definiscono un'astrazione rispetto alla piattaforma di esecuzione. Per questo motivo, un pezzo di codice che utilizza la libreria `ibrcommon`, può essere portato da una piattaforma ad un'altra in maniera abbastanza indolore, a patto che la piattaforma includa `ibrcommon`.

- `ibrdtn`: libreria, basata su `ibrcommon`, che implementa gli aspetti strettamente legati al Delay-Tolerant Networking, definendo algoritmi e strutture dati. In particolare, essa contiene moduli che implementano il parsing, il processamento e la serializzazione/deserializzazione delle strutture dati definite dalle specifiche del Bundle Protocol. Inoltre, la libreria definisce le funzionalità da esporre alle applicazioni che si collegano all'API di IBR-DTN tramite socket.

- `daemon`: eseguibile che implementa tutte le funzionalità di un *bundle node*, includendo tutte le funzionalità dei moduli precedentemente descritti.

- `tools`: un insieme di applicazioni a linea di comando che interagiscono con il *bundle node*, sfruttando le funzionalità da esso esposte.

4.3 Entità

Il software IBR-DTN è caratterizzato da una grande quantità di entità, ovvero strutture dati generate e processate dai vari componenti. Ogni entità è modellata mediante un oggetto, istanza di una specifica classe. Questa sezione illustra le principali entità coinvolte all'interno di IBR-DTN.

4.3.1 Event

Gran parte delle interazioni tra i moduli IBR-DTN avvengono tramite un meccanismo ad eventi. Gli eventi possono essere di diverso tipo, a seconda di ciò che li ha scatenati, ma tutti sono derivati dall'entità `Event`.

4.3.2 EID

L'entità `EID` modella un endpoint identifier e lo rende confrontabile con altri. `EID` contiene dei metodi specifici per estrarre e confrontare le parti che compongono l'identificativo.

4.3.3 BLOB

I bundle ricevuti da un *bundle node* sono memorizzati, inoltrati e potenzialmente cancellati. Poiché i bundle possono essere piuttosto grandi e ogni bundle può essere costituito da un numero potenzialmente infinito di blocchi, è necessario un meccanismo che permetta di memorizzare i dati dei blocchi nello storage. L'interfaccia per operare su tali dati deve essere generica e trasparente al fatto che essi si trovino in memoria, piuttosto che su disco. L'entità `BLOB` gestisce grandi quantità di dati in maniera efficiente, senza tener conto che essi siano memorizzati su un supporto persistente o volatile. Essa garantisce, inoltre, la protezione sugli accessi concorrenti.

4.3.4 BundleID

Come indicato all'interno di RFC 5050 [1], un bundle è identificato univocamente da EID sorgente, timestamp di creazione e numero di sequenza. Nel caso di un frammento, va considerato anche l'offset. Un entità di tipo `BundleID` racchiude tutte le suddette informazioni all'interno di un identificatore univoco, che può essere utilizzato per referenziare un bundle o un frammento di esso.

4.3.5 Bundle

L'entità `Bundle` contiene i riferimenti a tutti i dati che compongono un bundle. Un'istanza di `Bundle` è identificata da un `BundleID` e permette l'accesso al *primary block* e alle estensioni allegate al bundle.

4.3.6 Block

Un'entità di tipo `Block` rappresenta un blocco di payload per un entità di tipo `Bundle`.

4.3.7 Node

Un'entità `Node` rappresenta un nodo remoto scoperto o connesso. E' identificata dal corrispondente `EID` e include una serie di informazioni utili al trasferimento di bundle verso il nodo.

4.3.8 Registration

L'entità `Registration` è utilizzata per salvare lo stato delle applicazioni. Più precisamente, un'applicazione, interagendo con la API, può creare un istanza di `Registration` e associarvi uno o più entità `EID`. Se la destinazione di un bundle ricevuto o memorizzato sul nodo coincide con uno degli EID, il bundle è passato all'applicazione che aveva effettuato la registrazione.

4.4 Gestione degli eventi

Come già detto in precedenza, nella maggior parte dei casi, i componenti di IBR-DTN interagiscono tra loro tramite un meccanismo basato su eventi. Il sistema degli eventi è una parte del core e consente la concorrenza e l'isolamento tra thread differenti. Il sistema accetta gli eventi scatenati e li inoltra a tutti quei componenti che hanno dichiarato di essere interessati a questi tipi di eventi. Il componente `Event Dispatcher` gestisce la sottoscrizione da parte dei componenti ad un singolo tipo di evento. Gli eventi scatenati possono inoltrati immediatamente a tutti i componenti sottoscritti, oppure inseriti all'interno di una coda, in modo da essere consegnati successivamente. Più precisamente, nel primo caso, il sistema rimane

bloccato finché tutti i componenti non completano il processamento dell'evento. Nel secondo caso, invece, gli eventi sono sempre passati al modulo `Event Switch` e messi in una coda. Nel momento opportuno, `Event Switch` preleverà un'istanza di evento dalla coda e chiamerà `Event Dispatcher` per distribuirlo ai vari componenti. La decisione sul processare immediatamente l'evento o meno è presa da `Event Dispatcher`. La figura 4.4. mostra un esempio di processamento di un evento all'interno dell'architettura IBR-DTN. Nel caso dell'esempio, l'evento è scatenato da `Component 1`, il quale chiama `Event Dispatcher` per generare una nuova istanza di `Event`. Il `Component 2` rappresenta un modulo che ha effettuato una sottoscrizione per l'evento.

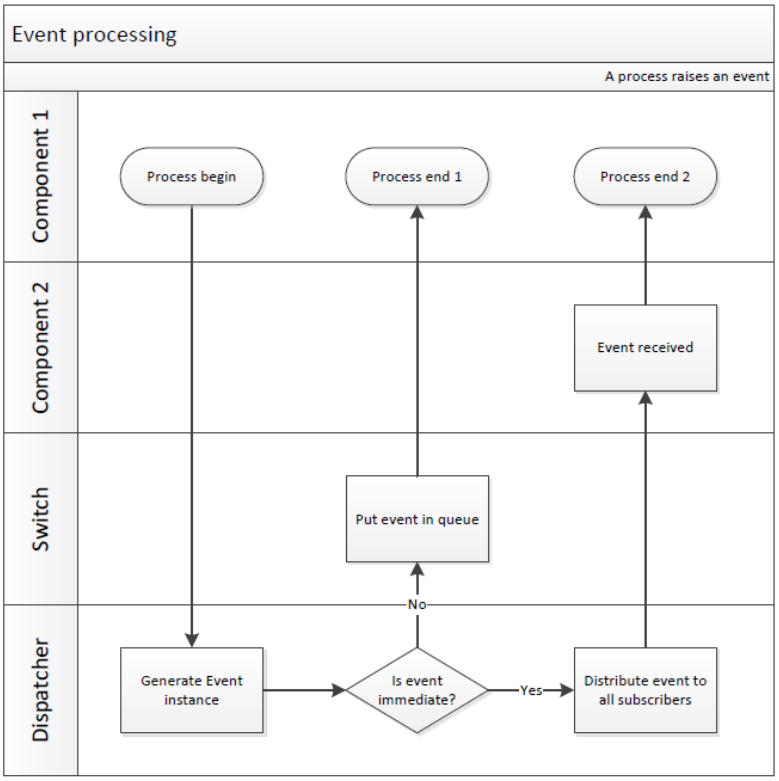


Figura 4.4: Esempio di generazione e gestione di un `Event` da parte del core IBR-DTN [6]

L'architettura IBR-DTN include diversi tipi di eventi, ognuno dei quali è scatenato al verificarsi di una certa condizione e coinvolge componenti differenti. Di seguito ne vengono riportati alcuni esempi:

- `BundleEvent`: scatenato quando un bundle viene ricevuto, memorizzato, inoltrato, cancellato o consegnato;
- `BundleExpiredEvent`: scatenato nel momento in cui un bundle memorizzato nello storage oltrepassa il suo *lifetime*;
- `BundleReceivedEvent`: scatenato quando un bundle viene ricevuto tramite la API o un *convergence layer*, allo scopo di passare tale bundle al componente incaricato del suo processamento;

- `ConnectionEvent`: segnala che una connessione è iniziata o terminata, tenendo traccia del `Node` coinvolto e dello stato della connessione;
- `GlobalEvent`: notifica cambiamenti di stato all'interno del core, indicando lo stato del *bundle node* (in spegnimento, sospeso, idle, ...);
- `NodeEvent`: segnala i cambiamenti di stato di un entità di tipo `Node` (nuovo nodo disponibile nel vicinato, nodo scomparso), oppure l'aggiunta/rimozione di dati da un `Node`.
- `QueueBundleEvent`: indica che un bundle è stato memorizzato nello storage ed è pronto per essere instradato.
- `TransferCompleteEvent`: notifica che la trasmissione di un bundle è stata completata, e tiene un riferimento al bundle e all'EID del nodo destinazione.

4.5 Concorrenza

Come descritto in precedenza, l'architettura IBR-DTN è partizionata in diversi componenti. Ognuno di essi può essere eseguito in un thread a sé stante o può soltanto reagire al verificarsi di eventi. Nel primo caso si parla di componente indipendente, nel secondo di componente integrato (effettua esclusivamente il processamento di eventi). L'utilizzo del multi-threading porta, da un lato, una serie di vantaggi dal punto di vista della scalabilità, tuttavia, dall'altro lato, introduce una serie di problemi legati all'accesso concorrente ai dati. All'interno di IBR-DTN, l'accesso ai dati è protetto mediante *mutex*, una versione semplificata dei semafori, che garantisce che un solo processo alla volta entri in una sezione critica. Qualora un secondo processo voglia entrare nella stessa regione critica, rimarrà bloccato fino al rilascio del *mutex* da parte dell'altro processo.

Capitolo 5

IBR-DTN su reti eterogenee

5.1 Obiettivo

Lo sviluppo di questa tesi si propone di creare il prototipo di un'architettura di rete che, sfruttando le caratteristiche del *Delay Tolerant Networking*, permetta di interconnettere dispositivi che utilizzano tecnologie di rete di natura eterogenea. Più precisamente, si desidera che due dispositivi possano scambiarsi dei dati, incapsulandoli all'interno di bundle, senza dover tener conto dell'effettiva tecnologia di rete con cui avviene la trasmissione. Lo scenario tipico è quello dell'*Internet of Things (IoT)*, in cui i dispositivi, solitamente mobili, devono poter comunicare tra loro utilizzando tecnologie di tipo wireless, senza la necessità di un'infrastruttura di rete vera e propria. Tra le tecnologie di comunicazione wireless maggiormente diffuse, troviamo WiFi (IEEE 802.11), LoWPAN (IEEE 802.15.4), Bluetooth e BLE (IEEE 802.15.1). Poiché le tecnologie wireless sono caratterizzate da interruzioni della connettività e ritardi molto lunghi, in questo tipo di scenari applicativi, è fortemente sconsigliato l'uso dei paradigmi di comunicazione tradizionali. La soluzione proposta cerca di ovviare a tale problema mediante il paradigma del *Delay Tolerant Networking*, e di conseguenza, del Bundle Protocol. Un qualunque dispositivo, per poter agire da *bundle node*, ovvero, essere in grado di ricevere e trasmettere bundle, necessita di un Bundle Protocol Agent (BPA), uno strato software che estende lo stack di rete del dispositivo, consentendo ad esso di offrire i servizi previsti dal Bundle Protocol. L'implementazione di BPA utilizzata per lo sviluppo di questa tesi è IBR-DTN, le cui caratteristiche sono state descritte nel Capitolo 4. Il software IBR-DTN offre supporto nativo alla suite di protocolli Internet (TCP/IP, UDP/IP), operanti su Ethernet o WiFi, allo standard IEEE 802.15.4 , ma non a Bluetooth. Per questa ragione, si è deciso di estenderne il codice, implementando uno strato software che consenta il supporto alla tecnologia Bluetooth. Il contenuto di questo capitolo è suddiviso in due sezioni: la prima offre una panoramica sugli aspetti generali da considerare nel momento in cui si vuole estendere il software IBR-DTN, la seconda descrive in dettaglio l'architettura implementata per il raggiungimento dell'obiettivo preposto.

5.2 Estensione del software IBR-DTN

L'architettura di IBR-DTN è contraddistinta da un'organizzazione fortemente modulare, in cui ognuno dei componenti è incaricato di svolgere uno specifico task. Uno dei principali vantaggi della modularità del codice è quello di permettere agli sviluppatori di estendere il software in maniera piuttosto semplice e poco invasiva.

5.2.1 Creazione e integrazione di un nuovo componente

La suite IBR-DTN offre supporto ai progettisti software tramite la definizione di una serie di classi interfaccia, ognuna pensata per modellare il comportamento di uno specifico modulo. Perciò, ogni qualvolta si desideri aggiungere un nuovo modulo all'architettura IBR-DTN, è opportuno che esso implementi la classe interfaccia definita per la categoria di moduli a cui esso appartiene (*convergence layer* o *routing module*, ad esempio), e che quindi fornisca l'implementazione dei metodi definiti dall'interfaccia stessa. Una volta creato un nuovo modulo, questo può interagire con il resto dei componenti presenti nell'architettura secondo due modalità: mediante chiamate sincrone o tramite il meccanismo basato su eventi, illustrato nella Sezione 4.4. In particolare, il modulo può scatenare eventi specifici al verificarsi di certe condizioni, allo scopo di "svegliare" i componenti interessati al suddetto evento e fornire ad essi tutte le informazioni necessarie alla sua gestione. A sua volta, il modulo può sottoscrivere il proprio interesse ad essere notificato nel momento in cui si verifica un certo evento, ed implementare una propria logica di gestione dello stesso. Dal punto di vista operativo, invece, il modulo può essere di due tipi: *integrated*, come modulo che rimane in ascolto (passivo) di specifici eventi e reagisce di conseguenza, oppure come componente *indipendent*, eseguito in un thread a sé stante. In quest'ultimo caso, solitamente, il modulo esegue una routine principale, in modo ciclico fino alla sua terminazione. Il software IBR-DTN supporta la realizzazione di entrambe le modalità operative, mettendo a disposizione apposite classi interfaccia, `IndipendentComponent` e `IntegratedComponent`, entrambe derivate dalla classe base `Component`. Infine, qualora il modulo creato necessiti di caricare una particolare configurazione, bisognerà estendere la parte di codice in cui sono configurati i diversi componenti di IBR-DTN, includendovi la configurazione dei parametri del nuovo modulo.

5.2.2 Implementazione del supporto ad uno stack protocollare

Dopo aver effettuato una breve panoramica su cosa implichi, in generale, estendere il software IBR-DTN tramite l'aggiunta di nuovi componenti, la trattazione sposta l'attenzione sugli aspetti progettuali e implementativi legati allo sviluppo della tesi. Come accennato in precedenza, il lavoro svolto è stato finalizzato a fornire, alla suite IBR-DTN, il supporto allo standard Bluetooth come tecnologia di rete. Affinché IBR-DTN possa supportare Bluetooth, o più in generale, un qualsiasi standard di comunicazione, è necessaria la realizzazione e l'integrazione di alcuni componenti software, che possano interagire in maniera trasparente con gli altri moduli già

presenti all'interno dell'architettura. La seguente sezione si propone di illustrare gli aspetti generali che si devono considerare qualora si desideri interfacciare il software IBR-DTN con una nuova tecnologia di comunicazione.

In generale, per permettere a IBR-DTN di operare su un particolare stack protocollare, è necessario implementare almeno due moduli fondamentali:

- Un *convergence layer*, che offra un servizio di trasmissione dei bundle, permettendone l'imbustamento nei protocolli di trasporto offerti dallo stack in esame;
- Un *discovery agent*, che realizzi un meccanismo di scoperta del vicinato, sfruttando i protocolli e le procedure fornite dallo stack protocollare.

In teoria, dovrebbe essere lo stesso *convergence layer* a permettere la scoperta dei nodi vicini, nel momento in cui questi entrano nel raggio d'azione del nodo locale. Tuttavia, ad eccezione di alcuni casi, le funzionalità di trasmissione e di scoperta del vicinato sono inglobate in moduli separati, in un certo senso indipendenti tra loro. Infatti, la scelta di mantenere le funzionalità separate, oltre a favorire una migliore organizzazione del codice, consente di sostituire l'implementazione del meccanismo di *neighbor discovery* senza dover apportare modifiche al corrispondente *convergence layer* (e viceversa).

Convergence Layer

Lo scopo fondamentale di un *convergence layer* è quello di incapsulare i dettagli del protocollo di trasporto utilizzato per la trasmissione dei bundle. Nel momento in cui il core DTN desidera inviare un bundle, passa i dati del bundle e le informazioni sul nodo destinazione all'opportuno *convergence layer*, tramite una coda. I dettagli su come avvenga la trasmissione del bundle sono gestiti internamente al *convergence layer*, in quanto fortemente legati al protocollo di trasporto adottato. Come conseguenza naturale, ogni *convergence layer* può essere caratterizzato da una complessità differente, che dipende dal protocollo di trasporto che esso implementa. Tuttavia, in generale, ogni *convergence layer* dovrebbe fornire almeno le seguenti funzionalità:

- instaurazione di una connessione;
- trasferimento dati;
- chiusura di una connessione.

In realtà, l'instaurazione e la chiusura di una connessione sono funzionalità necessarie solo nel caso di un *convergence layer* di tipo connection-oriented. Quello che, invece, deve essere sempre garantito, è un meccanismo per il trasferimento dei dati, o meglio, dei bundle. Oltre alla trasmissione, lo stesso *convergence layer* dovrebbe occuparsi della ricezione dei bundle. Più precisamente, esso dovrebbe agire da server, mettendosi in ascolto di bundle provenienti da altri nodi, o dall'API,

e una volta ricevuti, trasferirli al core DTN per il loro processamento. Qualora fossero necessari meccanismi di frammentazione/riassemblaggio dei bundle, questi devono essere implementati a livello di convergence layer, in modo da nasconderne i dettagli al core. Inoltre, piuttosto che un unico *convergence layer*, è plausibile pensare ad un *convergence layer* per ogni servizio di trasmissione che si vuole offrire, a prescindere dal fatto che ognuno di essi utilizzerà gli stessi protocolli di più basso livello per realizzare la trasmissione. In molti casi, infatti, può essere conveniente fornire diversi servizi di trasporto di bundle, a seconda delle esigenze di affidabilità e/o semplicità che si vogliono soddisfare. Ad esempio, nel caso del protocollo IP, la trasmissione dei bundle può essere veicolata su TCP o UDP, in base alla tipologia di servizio che si vuole offrire. Tuttavia, indipendentemente dalla scelta di utilizzare un protocollo rispetto che l'altro, i bundle in ingresso e in uscita transiteranno sicuramente dallo strato IP e quelli inferiori.

Discovery Agent

Per quanto concerne l'implementazione di un *discovery agent*, è opportuno che esso possa funzionare indipendentemente dal protocollo di trasporto adottato per la trasmissione dei bundle. Per comprendere meglio quanto enunciato, si prenda come esempio il modulo `IP Neighbor Discovery (IPND) Agent`, integrato nella suite IBR-DTN per fornire un meccanismo di scoperta del vicinato, basato su IP, a tutti quei nodi che utilizzano la suite di protocolli Internet (TCP/IP, UDP/IP). Il funzionamento di tale modulo è basato sull'invio periodico di *beacon*, dei messaggi molto piccoli utilizzati dai nodi per annunciare la propria presenza e scambiarsi tutte le informazioni necessarie alla comunicazione. I *beacon* non sono altro che datagrammi UDP inviati ad un indirizzo IP multicast noto (o comunque specificabile in configurazione). Ogni nodo annuncia il proprio EID all'interno di tali messaggi, in modo da permettere, ai nodi che lo riceveranno, di creare un mapping univoco tra l'EID e l'indirizzo IP del nodo mittente (ricavato dall'intestazione IP del pacchetto). L'esecuzione del modulo `IPND Agent` è assolutamente indipendente dal protocollo di trasporto adottato per la trasmissione dei bundle, purché quest'ultimo si appoggi su IP. Ne consegue che, a prescindere dal fatto che un nodo possa trasmettere bundle su TCP/IP, piuttosto che UDP/IP, potrà sfruttare il modulo `IPND Agent` per la scoperta del vicinato. L'unico requisito necessario al funzionamento di tale modulo è che il nodo disponga di almeno un'interfaccia con un indirizzo IP ad essa associato. Tuttavia, il nodo dovrà specificare, all'interno dei *beacon* di scoperta, il protocollo che utilizza per il trasporto dei bundle e la relativa informazione di servizio, ovvero la porta alla quale è in ascolto. Queste informazioni, insieme a EID e indirizzo IP, costituiscono quanto necessario all'instaurazione di una comunicazione tra i nodi.

Nel caso di uno stack protocollare non basato su IP, non è possibile adottare `IP Neighbor Discovery (IPND) Agent` per la scoperta del vicinato, perlomeno senza effettuare alcuna modifica. In realtà, esistono due opzioni differenti. La prima prevede che il *convergence layer* che implementa il protocollo di trasporto, fornisca un'interfaccia per la trasmissione di messaggi in multicast, permettendo così di utilizzare `IPND Agent` sullo stack protocollare in esame. La seconda opzione consiste nell'implementazione di un meccanismo ad-hoc che, utilizzando i protocolli

e le procedure offerti dallo stack protocollare, realizzi un meccanismo per lo scambio delle informazioni necessarie alla comunicazione fra i nodi.

5.3 IBR-DTN su Bluetooth

Il contenuto di questa sezione fornisce una descrizione dettagliata dell'architettura realizzata allo scopo di integrare, all'interno della suite IBR-DTN, il supporto alla tecnologia Bluetooth.

5.3.1 Requisiti di sistema

La soluzione realizzata è concepita per sistemi Linux, preferibilmente dotati di una versione piuttosto aggiornata del kernel. Lo sviluppo è basato sull'utilizzo di BlueZ (Sezione 3.4), lo stack Bluetooth ufficiale di Linux, e in particolar modo sulla libreria *libbluetooth*, da esso fornita. Sebbene il package BlueZ sia integrato, di default, nelle versioni recenti del kernel Linux, la libreria utente deve essere installata esplicitamente, digitando il seguente comando:

```
$ sudo apt-get install libbluetooth-dev
```

Complessivamente, i requisiti di sistema, sia hardware che software, necessari per poter eseguire la soluzione implementata, sono i seguenti:

- interfaccia fisica Bluetooth conforme alle specifiche 4.0 o superiori, capace di operare in "dual mode" (BR/EDR + LE);
- versione del package BlueZ pari o superiore alla 5.1;
- libreria utente *libbluetooth-dev*.

Il supporto alla tecnologia Bluetooth da parte di IBR-DTN è stato reso facoltativo, andando incontro ad uno dei suoi principi fondamentali, secondo cui deve esistere una versione minimale del software, che fornisca esclusivamente le funzionalità di base di un *bundle node*. Tutte le funzionalità accessorie, che tipicamente richiedono l'utilizzo di librerie esterne, devono essere opzionali. Di conseguenza, affinché vengano compilati correttamente tutti i moduli che offrono supporto a Bluetooth, durante la compilazione dei sorgenti, come parametro del comando `configure`, deve essere specificato il flag `--with-bluetooth`. Più precisamente, invocando il comando `configure --with-bluetooth`, viene controllata la presenza, all'interno del sistema, della libreria *libbluetooth-dev*. Qualora essa non sia presente, il processo di compilazione ignorerà tutti i file relativi a Bluetooth. Supponendo, invece, che la compilazione e l'installazione dei componenti di supporto a Bluetooth sia andata a buon fine, sarà necessario abilitare in modo esplicito la loro esecuzione, modificando un'apposita sezione nel file di configurazione del processo demone, come sarà mostrato più avanti nella Sezione 5.3.6.

5.3.2 Architettura

L'espletamento del supporto alla tecnologia Bluetooth da parte di IBR-DTN, ha richiesto la creazione e l'integrazione di alcuni componenti, o più precisamente, di alcune classi software:

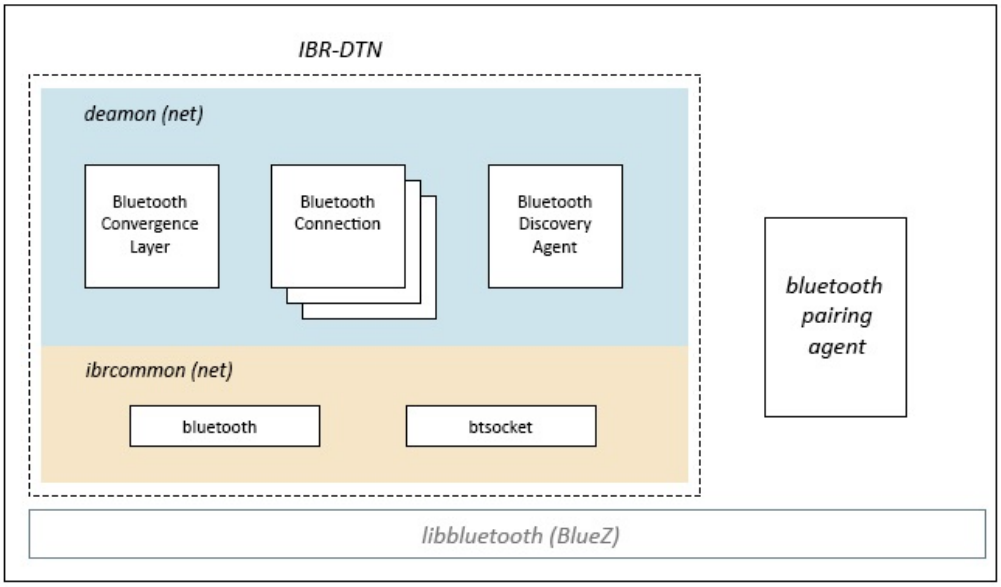


Figura 5.1: Moduli dell'architettura IBR-DTN su Bluetooth

- `bluetooth`: libreria di funzioni di controllo dell'adapter Bluetooth locale, che sfruttano la HCI API offerta dallo stack BlueZ;
- `btsocket`: libreria di funzioni utili alla comunicazione mediante socket su Bluetooth;
- `Bluetooth Convergence Layer`: implementa un server Bluetooth (RFCOMM), che gestisce le connessioni in ingresso;
- `Bluetooth Connection`: modella una connessione Bluetooth RFCOMM tra due nodi, gestendo sia la ricezione che l'invio di bundle;
- `Bluetooth Discovery Agent`: realizza la scoperta del vicinato, allo scopo di rilevare la presenza di altri dispositivi Bluetooth, partecipanti alla DTN, situati nel raggio d'azione del nodo locale.

Da un punto di vista organizzativo, le classi `bluetooth` e `btsocket` sono state integrate all'interno della libreria `ibrcommon`, in quanto implementano funzionalità non direttamente attinenti con il Delay-Tolerant Networking. Al contrario, le classi `Bluetooth Convergence Layer`, `Bluetooth Connection` e `Bluetooth Discovery Agent`, che realizzano funzionalità di *networking* per un *bundle node*, sono state incluse all'interno del package `daemon`.

Oltre alle suddette classi, è stato utilizzato un ulteriore componente, il `bluetooth pairing agent`, concepito come modulo esterno indipendente. Il suo unico scopo è quello di automatizzare e semplificare le procedure di pairing tra i dispositivi Bluetooth, rendendo praticamente inutile l'intervento umano. I dettagli sul funzionamento dei singoli moduli saranno descritti nelle sezioni successive.

5.3.3 Trasmissione di bundle su Bluetooth

Allo scopo di offrire, alla suite IBR-DTN, un meccanismo di trasmissione di bundle sulla tecnologia Bluetooth, sono state implementate due classi software, `Bluetooth Convergence Layer` e `Bluetooth Connection`. In dettaglio, si è voluto realizzare un meccanismo di trasmissione e ricezione di bundle su Bluetooth, che incapsulasse i dettagli relativi all'hardware e al protocollo di trasporto utilizzato. Come protocollo di trasporto, è stato scelto RFCOMM, nato per emulare comunicazioni seriali sullo standard Bluetooth. Il protocollo RFCOMM offre un servizio *connection-oriented* e *stream-based* affidabile, in maniera piuttosto simile al protocollo TCP per la suite dei protocolli Internet. La comunicazione avviene mediante socket, sfruttando le primitive di comunicazione implementate all'interno della libreria `btsocket`. Date le forti analogie tra RFCOMM e TCP, l'implementazione delle classi `Bluetooth Convergence Layer` e `Bluetooth Connection` prende spunto da quanto presente, all'interno di IBR-DTN, per fornire comunicazione su TCP/IP. L'attuale implementazione non supporta la trasmissione di dati su Bluetooth attraverso un protocollo basato su *datagram* e non affidabile.

Bluetooth Convergence Layer

La classe `Bluetooth Convergence Layer` implementa un server Bluetooth RFCOMM, in attesa di connessioni ingresso. Tale componente è istanziato dal modulo `ConnectionManager`, invocato all'interno del processo demone (`NativeDaemon`) per configurare e mettere in esecuzione i singoli *convergence layer*. Il modulo `Bluetooth Convergence Layer`, una volta istanziato, riceve dal `ConnectionManager` due parametri, caricati dal file di configurazione (se presente): l'interfaccia Bluetooth e il canale RFCOMM su cui rimanere in ascolto. Questi parametri sono passati tramite chiamata sincrona del metodo `add()`, che a sua volta invoca il metodo `listen()` di `Bluetooth Convergence Layer`. Quest'ultimo metodo, dopo aver memorizzato internamente le informazioni su interfaccia e canale, istanzia un server RFCOMM, che dovrà restare in ascolto sull'indirizzo MAC Bluetooth associato all'interfaccia e sul canale.

Allo scopo di operare come *convergence layer*, la classe `Bluetooth Convergence Layer` implementa l'interfaccia `ConvergenceLayer`, i cui metodi sono illustrati in figura 5.2.

Un modulo che desideri agire da *convergence layer*, deve fornire, come minimo, l'implementazione dei metodi `getDiscoveryProtocol()` e `queue()`. Il metodo `getDiscoveryProtocol()` ritorna un identificativo del protocollo implementato dal *convergence layer*. Come si vedrà più avanti, la fase di scoperta del vicinato mira ad ottenere una serie di informazioni sui vicini, tra cui l'identificativo del

```
class ConvergenceLayer
{
        public:

        // distruttore virtuale
        virtual ~ConvergenceLayer() = 0;

        virtual dtn::core::Node::Protocol getDiscoveryProtocol()
           const = 0;

        virtual void queue(const dtn::core::Node &n, const
           dtn::net::BundleTransfer &job) = 0;

        virtual void open(const dtn::core::Node&) {};

        virtual void resetStats();

        virtual void getStats(ConvergenceLayer::stats_data &data)
           const;
};
```

Figura 5.2: Interfaccia `ConvergenceLayer` di IBR-DTN.

protocollo con cui è possibile comunicare con essi. Ogni volta che i moduli del core DTN desiderano avviare il trasferimento di un bundle verso un nodo vicino, confrontano l'identificativo di protocollo ottenuto dalla fase di discovery con quello ritornato dal metodo `getDiscoveryProtocol()`, al fine di attivare il *convergence layer* opportuno per il trasferimento. Più precisamente, per attivare il trasferimento, viene invocato il metodo `queue()` del *convergence layer* selezionato, che inserisce il bundle in attesa di essere trasferito all'interno di una coda. Il metodo riceve come parametri un oggetto di tipo `Node`, che identifica il nodo destinatario del bundle, e un oggetto di tipo *BundleTransfer*, che incapsula un EID e un riferimento al bundle da trasferire. Nel caso del `Bluetooth Convergence Layer`, il metodo `queue()` è implementato come segue:

- controlla se esiste una connessione aperta verso il nodo passatogli come parametro, scorrendo una lista di oggetti di tipo `Bluetooth Connection`, che tiene traccia di tutte le connessioni Bluetooth RFCOMM aperte verso altri nodi;
- in caso positivo, il bundle da trasferire viene messo in una coda associata alla connessione aperta;
- in caso negativo, viene creata una nuova connessione Bluetooth RFCOMM verso il nodo, istanziando un oggetto di tipo `Bluetooth Connection`. A fronte della creazione della nuova connessione, viene scatenato un evento di tipo

`ConnectionEvent::CONNECTION_SETUP`, allo scopo di notificare gli altri moduli di IBR-DTN che esiste una nuova connessione. Infine, la nuova connessione viene memorizzata all'interno della lista delle connessioni, e il bundle da trasferire viene messo all'interno della coda associata alla connessione stessa.

Il metodo `open()` dovrebbe permette di aprire una connessione verso un nodo remoto, anche in assenza di dati trasferire. Il suo funzionamento è molto simile a quello del metodo `queue()`, con la differenza che in questo caso non viene passato alcun bundle da mettere in coda. Infine, i metodi accessori `getStats()` e `resetStats()` consentono, rispettivamente, di ottenere e resettare le statistiche relative al *convergence layer*, ovvero il numero di bundle lo attraversano in ingresso e/o uscita.

Il modulo `Bluetooth Convergence Layer` è stato concepito come componente *indipendent*, da eseguire in un thread a sè stante. Come tutti i componenti di questo tipo, che implementano l'interfaccia `IndipendentComponent`, il modulo deve implementare i metodi `componentUp()`, `componentDown()` e `componentRun()`. I metodi `componentUp()` e `componentDown()` sono invocati, rispettivamente, poco dopo che il modulo è partito e immediatamente prima che esso termini. `componentUp()` inizializza ed esegue il `Bluetooth Discovery Agent`, e fa partire l'ascolto sul socket istanziato in precedenza (tramite il metodo `listen()`. In maniera duale, `componentDown()` termina l'esecuzione del `Bluetooth Discovery Agent`, chiude il socket e forza la chiusura di tutte le connessioni attive. Il metodo `componentRun()` contiene la routine principale eseguita dal thread, racchiusa all'interno di un loop. In particolare, il thread di `Bluetooth Convergence Layer` rimane in attesa di connessione Bluetooth RFCOMM ingresso, provenienti da altri dispositivi Bluetooth partecipanti alla DTN. Nel momento in cui giunge una richiesta di connessione da parte di un nodo client, vengono recuperati l'indirizzo MAC Bluetooth e il canale RFCOMM associati alla richiesta, in modo da costruire l'istanza di `Node` associata al nodo in esame. A questo punto, viene creata una nuova connessione Bluetooth RFCOMM, istanziando un oggetto `Bluetooth Connection`, a cui vengono passati l'oggetto `Node` e il socket su cui poter comunicare con il nodo.

Bluetooth Connection

La classe `Bluetooth Connection` modella una connessione Bluetooth RFCOMM verso un nodo remoto, partecipante alla DTN. Ogni istanza di questa classe gestisce una connessione verso un nodo specifico, occupandosi sia della trasmissione, che della ricezione di bundle sullo stream di comunicazione.

Di base, ogni connessione tra due *bundle node* che utilizzino Bluetooth RFCOMM inizia con uno scambio di `StreamContactHeader`, una struttura dati contenente l'EID del nodo mittente ed una serie di flag che abilitano o meno delle funzionalità opzionali per il trasferimento dati. Nel caso specifico di `Bluetooth Connection`, tali flag forzano la richiesta di *acknoledgement*, con cui il nodo locale chiede che sia confermata la ricezione dei segmenti di dati, da parte del destinatario. Oltre agli *acknoledgement*, il nodo può richiedere di abilitare la frammentazione, a patto che la sua abilitazione sia richiesta in configurazione del demone.

Come accennato nella sezione precedente, ogni volta che il modulo `Bluetooth Convergence Layer` riceve una richiesta di connessione da parte di un nodo, viene creata un'istanza di `Bluetooth Connection`, che riceve una serie di parametri, tra cui un riferimento al nodo remoto e il socket RFCOMM aperto per poter comunicare con esso. L'oggetto così creato, eseguito come thread *detached*, è incaricato di gestire la ricezione dei bundle in arrivo sul socket.

All'interno di IBR-DTN, i bundle sono gestiti tramite oggetti, incapsulati in BLOB. Poiché due *bundle node* non possono scambiare direttamente oggetti tra di loro, i bundle devono essere codificati in una rappresentazione binaria, secondo un processo noto come serializzazione. IBR-DTN fornisce delle classi che realizzano le funzionalità di serializzazione e deserializzazione, che permettono di leggere e scrivere bundle su degli stream in maniera molto semplice.

Dunque, la parte di ricezione di `Bluetooth Connection` riceve dati in ingresso sulla connessione e provvede alla loro deserializzazione, in modo da ottenere il bundle originario (o un frammento di esso, nel caso in cui sia stata eseguita la frammentazione). Dopo aver eseguito dei controlli minimali sui campi del bundle ottenuto, quest'ultimo attraversa delle tabelle di filtraggio. Questa operazione risulta sensata solamente nel caso in cui siano abilitate le estensioni del Bundle Security Protocol (BSP), che permettono di effettuare controlli di integrità ed autenticazione a livello di Bundle Protocol. Infatti, nel momento in cui le estensioni del BSP sono disabilitate, tutti i bundle in ingresso sono accettati automaticamente. In caso contrario, a seconda dell'esito del filtraggio, il bundle ricevuto può essere scartato o accettato. In caso di accettazione, il bundle viene iniettato nel `BundleCore`, il quale, dopo averlo memorizzato nel modulo di storage, scatenerà un opportuno evento (`BUNDLE_RECEIVED`) per attivare i moduli di routing, incaricati del suo inoltro.

Dall'altro lato, la trasmissione dei bundle è gestita tramite un thread separato, detto `sender` thread, istanziato da `Bluetooth Connection` stessa. Tale thread rimane bloccato sulla coda dei bundle in attesa di essere trasferiti. Nel momento in cui un bundle deve essere inviato, dalla coda ne viene prelevato il riferimento, mentre il bundle vero e proprio viene recuperato dal modulo di storage. A questo punto, valgono le stesse considerazioni sulle tabelle di filtraggio fatte per la parte di ricezione. Supponendo che il bundle da trasferire sia stato accettato, questo viene serializzato e trasmesso verso il nodo remoto sul flusso di comunicazione. Nel caso in cui sia abilitata la frammentazione, durante questa fase non viene inviato l'intero bundle, ma i frammenti di esso, uno per volta, tenendo traccia dell'offset rispetto al bundle originale.

Oltre al `sender` thread, ogni `Bluetooth Connection` istanzia un ulteriore thread, detto `keepalive sender` thread. Questo thread, ad intervalli regolari e piuttosto lunghi (o comunque configurabili), trasmette un messaggio di KEEPALIVE sullo stream di comunicazione, un messaggio di qualche byte inviato allo scopo di tenere sotto controllo lo stato del collegamento ed evitare che la connessione cada, nel caso in cui non avvenga alcun trasferimento dati per molto tempo. Poiché l'instaurazione di una connessione RFCOMM su Bluetooth è un'operazione molto onerosa, l'utilizzo del `keepaliveSender` thread evita di dover creare una nuova connessione ogni volta che non vengono trasferiti dati per tanto tempo, con conseguente caduta della connessione.

5.3.4 Scoperta dei propri vicini

Poiché lo standard Bluetooth definisce un intero stack protocollare, diverso dalla suite di protocolli Internet, e di conseguenza non basato su IP, non si è potuto utilizzare il modulo `IPND Agent` per la scoperta del vicinato. Di conseguenza, si è realizzato un meccanismo ad-hoc basato su Bluetooth, che sfrutta le procedure di adverting e scanning offerte da Bluetooth Low Energy per permettere ai nodi di annunciare la propria presenza e scambiare tutte le informazioni utili alla comunicazione. Tale meccanismo è stato implementato all'interno della classe `Bluetooth Discovery Agent`, integrata alla suite IBR-DTN per permettere la scoperta del vicinato di un nodo utilizzando la tecnologia Bluetooth. Nel contesto di Bluetooth, o più in generale delle tecnolgie wireless, con "vicinato di un nodo" si intende l'insieme dei dispositivi collocati all'interno del raggio d'azione del nodo stesso. Tale concetto si traduce, in un'architettura di tipo Delay-Tolerant Network, come l'insieme dei nodi, partecipanti alla DTN, raggiungibili direttamente dal nodo di riferimento. Come accennato in precedenza, il funzionamento del `Bluetooth Discovery Agent` è basato sul meccanismo di scoperta dei dispositivi nativo del Bluetooth LE, a sua volta fondato sulle procedure di advertising e scanning.

La scelta di adottare dei meccanismi nativi della tecnologia Bluetooth LE è stata influenzata da una serie di fattori. In primo luogo, è necessario considerare che il demone IBR-DTN è concepito per dispositivi con limitate risorse hardware, soprattutto dal punto di vista della durata della batteria. Per di più, è auspicabile che il modulo di scoperta del vicinato rimanga in esecuzione per un tempo indefinito, potenzialmente molto lungo, durante il quale esso consuma cicli di CPU. L'utilizzo del Bluetooth LE si presta particolarmente bene a soddisfare l'esigenza di risparmio energetico di una vasta gamma di dispositivi. Inoltre, l'utilizzo di procedure native dello standard rende la soluzione decisamente più semplice, in quanto evita di dover definire un protocollo di comunicazione ad-hoc per permettere ad un nodo di annunciarsi ai suoi vicini. Un'ulteriore motivazione per cui si è optato all'utilizzo di Bluetooth LE risiede nella possibilità, offerta ai progettisti software, di personalizzare il contenuto del payload dei messaggi di advertising. Per lo scopo di questa tesi, è stato definito un formato personalizzato per i messaggi di advertising, atto a contenere tutte le informazioni che i nodi devono scambiare per poter trasmettere bundle su Bluetooth. Qui di seguito ne vengono illustrati i dettagli.

Formato dei messaggi

Come accenato in precedenza, al fine di espletare il meccanismo di scoperta dei vicini, il `Bluetooth Discovery Agent` genera dei messaggi di advertising con un payload personalizzato. Per una trattazione dettagliata sul formato previsto dallo standard Bluetooth LE per i messaggi di advertising, si rimanda alla Sezione 3.2.2.

Nonostante lo standard Bluetooth LE offra la possibilità di personalizzare il contenuto del payload dei messaggi di advertising, affinché il loro formato sia conforme alle specifiche, è richiesto che il primo e il secondo byte del payload indichino, rispettivamente, la lunghezza (`Length`) e il tipo (`Type`) di messaggio. Il campo `Length` indica la lunghezza totale del payload, escluso 1 byte relativo al campo `Length` stesso. Il campo `Type` indica il tipo di informazioni contenute nel payload

e può assumere una serie di valori predefiniti, la cui lista completa e il rispettivo significato è reperibile all'interno delle specifiche Bluetooth. Nell'implementazione del `Bluetooth Discovery Agent`, il campo `Type` assume il valore `0x16 (Service Data)`, per indicare che i messaggi di advertising contengono informazioni associate ad un particolare servizio. Quali siano le informazioni effettivamente contenute, dipende dal servizio specifico. Nel nostro caso, il servizio che si va ad annunciare è quello di "IBR-DTN over Bluetooth", inteso come la possibilità, da parte di un nodo che esegue il demone IBR-DTN, di inoltrare bundle sulla tecnologia Bluetooth. I dati associati al servizio racchiudono quanto necessario all'instaurazione di una connessione Bluetooth tra nodi partecipanti alla DTN, su cui si andrà a veicolare la trasmissione dei bundle. Ovviamente, "IBR-DTN over Bluetooth" è solo un nome adottato per convenzione, non corrispondente a nessuno dei servizi definiti e supportati dallo standard Bluetooth. I due byte immediatamente successivi al campo `Type` rappresentano il `Service UUID16`, un valore esadecimale su 16-bit che identifica il servizio annunciato. A tal proposito, il servizio "IBR-DTN over Bluetooth" è identificato da un valore predeterminato (`0x19EA`), generato in modo pseudo-casuale. Dunque, ogni nodo IBR-DTN che fornisce il servizio di trasmissione di bundle su Bluetooth, dovrà annunciare tale servizio, all'interno dei messaggi di advertising, utilizzando il valore `0x19EA`. In questo modo, alla ricezione di un messaggio di advertising, un nodo sarà capace di capire che esso proviene da un altro nodo partecipante alla DTN, e che questo offre connettività Bluetooth. I restanti byte del payload del messaggio corrispondono ai dati associati al servizio. In particolare, il campo `Channel` indica il canale RFCOMM al quale il nodo *advertiser*, o meglio, il `Bluetooth Convergence Layer` in esecuzione su esso, si trova in attesa di connessioni Bluetooth (via RFCOMM). Poiché la soluzione proposta offre esclusivamente la trasmissione di bundle su Bluetooth via RFCOMM, non è necessario discriminare, all'interno del messaggio di advertsing, che il campo *Channel* rappresenti un'informazione relativa al servizio di trasmissione su RFCOMM, piuttosto che su un altro servizio di trasporto. Per questo motivo, nel momento in cui esista più di un servizio di trasmissione di bundle su Bluetooth, è opportuno che il campo `Channel` sia preceduto da un identificativo del servizio di trasmissione. Gli ultimi byte del payload dei messaggi di advertising contengono l'EID del nodo *advertiser*. Un EID, come descritto nella Sezione 2.2.3, si compone, a sua volta, di due parti: *EID scheme* e *EID scheme-specific part (ssp)*. Per poter garantire un maggior numero di byte al campo *EID ssp*, piuttosto che includere l'intero *EID scheme* all'interno dei messaggi di advertising, si è preferito codificare tale informazione con un numero esadecimale, incluso all'interno del campo `Encoded EID scheme`. La versione attuale del `Bluetooth Discovery Agent` supporta esclusivamente schema DTN ("dtn://"), codificato con il valore esadecimale `0x01`, ma è possibile, in maniera piuttosto semplice, estendere il codice per offrire supporto ad altri schemi. I byte successivi a `Encoded EID scheme` costituiscono il campo `EID scheme-specific part`, una stringa, che insieme allo schema, identifica univocamente il nodo *advertiser* all'interno della DTN.

Con riferimento a quanto detto finora, la figura 5.3 mostra un esempio di payload di un messaggio di advertising, generato da un nodo, avente EID `dtn://neighbor1`, che offre connettività Bluetooth via RFCOMM ed è in ascolto sul canale 10 (`0x0A`).

1	1	2	1	1	9
LENGTH 0x0E	**TYPE** 0x16 (Service Data)	**SERVICE UUID16** 0x19EA (IBR-DTN over Bluetooth)	**CHANNEL** 0x0A	**ENCODED** **EID scheme** 0x01 (dtn://)	**EID scheme-specific part** neighbor1

Figura 5.3: Esempio del payload di un messaggio di advertising generato dal modulo `Bluetooth Discovery Agent`

La trattazione, a questo punto, prosegue descrivendo la configurazione adottata per le procedure di advertising e scanning, analizzando i principali parametri coinvolti e le ragioni alla base della scelta dei valori per tali parametri. Per maggiori dettagli sul significato dei parametri, si faccia riferimento alla Sezione 3.2.2.

Configurazione della procedura di advertising

I principali parametri coinvolti nella procedura di advertising sono:

- *Advertising Interval*: rappresenta l'intervallo di tempo che intercorre tra la generazione di due messaggi di advertising consecutivi. Il suo valore va espresso in millisecondi e deve essere multiplo di 0,625 ms. Per questo motivo, nel caso in cui si voglia utilizzare un *Advertising Interval* di n ms, sarà necessario dividere il valore di n per 0,625. Nel nostro caso, ad esempio, il parametro assume il valore 2048, che corrisponde, in millisecondi, al valore 1280. La scelta del valore per il parametro *Advertising Interval* dipende fortemente contesto applicativo, ma, in generale, è opportuno garantire un giusto compromesso tra consumo di risorse (in termini di carico di CPU, e conseguentemente, di consumo energetico) e latenza. Il valore di 1280 ms è frutto di una serie di prove sperimentali, da cui è emerso come valori dell'ordine del secondo conferiscano una discreta reattività al meccanismo di scoperta, senza impattare pesantemente sulle risorse.

- *Channel Map*: indica su quali canali, dei tre che Bluetooth LE dedica all'advertising (37, 38, 39), si vuole che vengano trasmessi i messaggi di advertising. In pratica, non è altro che una maschera di bit, in cui il valore di ogni bit (considerando i tre meno significativi) abilita/disabilita l'uso di un canale. Ad esempio, per abilitare tutti e tre i canali, è necessario assegnare a *Channel Map* il valore `0x0111` (7), per abilitare esclusivamente i canali 37 e 39, il valore `0x0101` (5). Sebbene la scelta di un set di ridotto di canali permetta di risparmiare energia, nella soluzione realizzata, per una questione di robustezza rispetto al fenomeno dell'interferenza, si è preferito abilitare l'uso di tutti e tre.

- *Advertising Type*: definisce il tipo di messaggio di advertising. Detto in maniera grossolana, tale parametro distingue il caso di advertising diretto (unicast) da quello indiretto (broadcast). Per lo scopo del `Bluetooth Discovery Agent`, che fondamentalmente si propone di diffondere informazioni ai nodi collocati nel vicinato, è stato utilizzato il tipo ADV_IND ("Advertising Indications"), che consente la trasmissione di messaggi in broadcast. I messaggi di advertising di tipo ADV_IND supportano un *Advertising Interval* con valore tra 20 ms e 10,24 s.

Configurazione della procedura di scanning

Per quanto riguarda la procedura di scanning, i parametri significativi sono:

- *Scan Window*: indica l'intervallo di tempo durante il quale si resta in ascolto su un singolo canale.
- *Scan Interval*: rappresenta l'intervallo di tempo tra due *Scan Window* consecutive. Lo *Scan Interval*, così come la *Scan Window*, possono assumere valori tra 10 ms e 10,24 ms, a multipli di 0,625 ms. Pertanto, per impostare correttamente il valore di tali parametri, è necessario eseguire la stessa operazione di divisione descritta in precedenza per l'*Advertising Interval*. Nel nostro caso, entrambi sia la *Scan Window* che lo *Scan Interval* sono stati impostati al valore 160, corrispondente a 100 ms. Tuttavia, più che considerar la scelta del valore in sé, è opportuno notare che si è forzato lo stesso valore per entrambi i parametri. In questo modo, ad eccezione dei piccoli intervalli temporali necessari a passare da un canale ad un altro (gli stessi canali utilizzati per l'advertising), è possibile rimanere in ascolto di messaggi di advertising durante tutto il tempo, andando incontro all'esigenza di reattività del protocollo di scoperta.
- *Scan Type*: definisce il tipo di procedura di scanning. Lo scanning, infatti, può essere attivo o passivo. Nel primo caso, alla ricezione di un messaggio di advertising, sarà generato un opportuno messaggio di risposta. Nel secondo, il dispositivo rimarrà semplicemente in ascolto, senza generare alcuna risposta. Per il nostro scopo, volendo mantenere una certa semplicità nel protocollo di scoperta, uno scanning di tipo passivo è più che ragionevole.
- *Scan Policy*: indica la politica di scanning da adottare. Nel nostro caso, si è mantenuto il comportamento di default, secondo cui la *Scan Policy* accetta tutti i messaggi di advertising in ingresso. Tuttavia, esiste un'altra opzione, secondo cui è possibile accettare i messaggi solo se provenienti da un set ridotto di nodi, i quali devono essere specificati all'interno di una lista detta "whitelist".
- *Duplicate Filter*: indica se filtrare o meno i messaggi duplicati. Di default, il filtraggio è abilitato. Tuttavia, in alcune applicazioni, può risultare utile processare più messaggi di advertising provenienti dallo stesso vicino. Ad esempio, il modulo `BluetoothDiscovery Agent` deve poter ricevere più messaggi di advertising da uno stesso vicino, per accorgersi che esso continua ad essere in attività. Per questo motivo, il parametro *Duplicate Filter* è stato disabilitato.

Implementazione di `Bluetooth Discovery Agent`

Dopo aver discusso ampiamente il formato definito per i messaggi di advertising e la configurazione delle procedure di scanning e advertising, la trattazione prosegue con una panoramica sul funzionamento del modulo`Bluetooth Discovery Agent`,

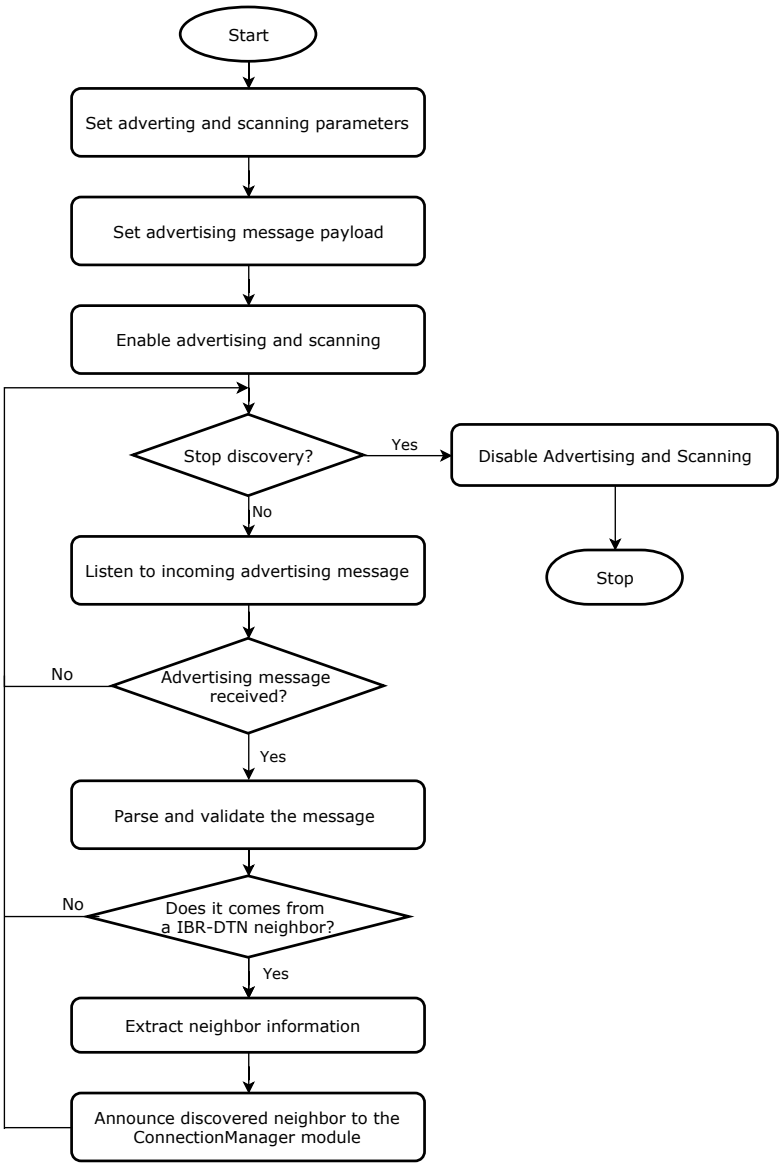


Figura 5.4: Diagramma di flusso del funzionamento del modulo `Bluetooth Discovery Agent`

ponendo particolare attenzione agli aspetti implementativi legati ad esso. Quanto verrà annunciato in questa sezione, anche se in maniera semplificata, è riassunto visivamente nella figura 5.4.

Innanzitutto, la classe `Bluetooth Discovery Agent` implementa l'interfaccia `IndipendentComponent`, al fine di essere eseguito come thread indipendente. Il suo ciclo di vita è gestito completamente dal modulo `Bluetooth Convergence Layer`, che ne effettua l'istanziazione, l'avvio, la terminazione e la distruzione. Il modulo, inoltre, è associato ad uno specifico adapter Bluetooth, lo stesso a cui è associato il suddetto convergence layer.

All'avvio, dopo aver configurato opportunamente i parametri e preparato i dati da includere all'interno dei messaggi di advertising, il `Bluetooth Discovery Agent` fa partire sia la procedura di scanning che quella di advertising. In questo modo, il modulo potrà annunciare la sua presenza ai nodi nel suo raggio d'azione e, al

contempo, rimanere in ascolto di eventuali messaggi provenienti da altri nodi. La procedura di advertising, così come lo scanning, sono caratterizzate da una serie di parametri, che devono essere impostati prima che le procedure vengano avviate. A livello pratico, la configurazione dei parametri, l'avvio e la terminazione delle procedure, può essere effettuata interagendo con la HCI API fornita dalla libreria Bluetooth di BlueZ, che permette, in generale, di inviare dei comandi all'adapter Bluetooth locale, astraendo l'applicazione dai dettagli di basso livello.

Nella routine principale, il modulo `Bluetooth Discovery Agent` si mette in ascolto di messaggi di advertising provenienti dai nodi, partecipanti alla DTN, situati nel suo raggio d'azione. Più precisamente, il modulo rimane in ascolto di eventi che riguardano l'interfaccia HCI, e in particolare di quelli di tipo EVT_LE_ADVERTISING_REPORT, che notificano la ricezione di messaggi di advertising sull'adapter Bluetooth locale, fornendo il contenuto di tali messaggi al livello applicativo. In questo modo, per ogni messaggio di advertising ricevuto sull'interfaccia, `Bluetooth Discovery Agent` ne effettua il parsing, e dopo aver verificato la correttezza del formato, estrae i dati utili alla comunicazione. In particolare, esso estrae l'indirizzo MAC Bluetooth del mittente dall'intestazione del messaggio, mentre le restanti informazioni necessarie all'instaurazione di una connessione, sono reperite dal payload. Accertato che il mittente del messaggio di advertising appartenga alla DTN (controllando che il valore del campo `Service UUID16` coincida con quello definito per il servizio "IBR-DTN over Bluetooth"), il `Bluetooth Discovery Agent` estrae, rispettivamente, il canale RFCOMM e l'EID dal payload del messaggio. In realtà, poiché il campo `EID scheme` è codificato, durante la fase di parsing viene effettuata la decodifica del valore, e il risultato ottenuto viene concatenato con il campo `EID scheme-specific part` per ricostruire EID originario completo.

Una volta ricavati l'indirizzo MAC Bluetooth e il canale RFCOMM su cui il modulo `Bluetooth Convergence Layer` del nodo remoto è in ascolto, oltre che il suo EID completo, il nodo locale disporrà di tutte le informazioni essenziali alla creazione di una connessione con il nuovo vicino. Per concludere la fase di scoperta, le informazioni relative al nodo devono essere memorizzate e notificate agli altri componenti. In IBR-DTN, le informazioni riguardanti un nodo sono modellate tramite la classe `Node`. Ogni oggetto della classe `Node` è identificato, oltre che dal suo EID, da una URI, che racchiude le informazioni per poter comunicare con il nodo e alcune informazioni di stato. Più precisamente, la URI include il protocollo da utilizzare per la comunicazione con il nodo e una stringa di coppie chiave-valore, che nel caso di `Bluetooth Discovery Agent` assume la forma *bdaddr=<address>;channel=<channel>;*. I parametri *<address>* e *<channel>* sono sostituiti con le informazioni estrapolate in precedenza. La URI include, inoltre, un attributo `expiryTime`, che rappresenta la validità temporale dell'informazione, intesa come l'istante di tempo, in relazione a quello corrente, fino a cui le informazioni relative al nodo potranno essere considerate aggiornate. Di default, le informazioni di un nodo hanno una validità temporale pari a tre volte il valore scelto per il parametro *Advertising Interval*. Questa scelta può essere interpretata, seppur grossolanamente, nel seguente modo: qualora un nodo non riceva, per tre o più volte consecutive, un messaggio di advertising da parte di un suo vicino, quest'ultimo è da ritenersi non più raggiungibile. Infine, l'oggetto `Node` deve essere

annunciato al `ConnectionManager`, che provvederà a inserirlo nel *neighbor database*, aggiornando così il vicinato del nodo locale. A questo punto, qualora il suddetto nodo risulti già presente all'interno del vicinato, il `ConnectionManager` provvederà solamente ad aggiornare le informazioni ad esso associate (il valore del parametro `expiryTime` sarà posticipato). Invece, nel caso si tratti di un vicino del tutto nuovo, quest'ultimo sarà inserito nel vicinato del nodo locale, e la sua presenza sarà notificata tramite la generazione dell'evento `NodeEvent::NODE_AVAILABLE`. Sarà lo stesso `ConnectionManager` a dover controllare periodicamente il valore dell'`expiryTime`, di tutti i nodi appartenenti al vicinato del nodo, al fine di rilevare la scadenza delle informazioni ad essi associate. Qualora il valore dell'`expiryTime` di un certo nodo risulti precedente rispetto all'istante di tempo corrente, il nodo sarà dichiarato come non più disponibile (anche in questo caso, generando un apposito evento, `NodeEvent::NODE_UNAVAILABLE`).

5.3.5 Pairing dei dispostivi

Affinché due dispositivi Bluetooth siano capaci di scambiarsi dei dati, è necessario che essi effettuino, preliminarmente all'instaurazione della connessione, il processo di *pairing*. Tale procedura, che ha come obiettivo la generazione di una chiave condivisa, è stata ampiamente trattata nella Sezione 3.3. Per lo sviluppo di questa tesi, il pairing dei dispositivi, basato sul meccanismo del *Secure Simple Pairing (SSP)*, è gestito tramite il `bluetooth pairing agent`, uno script Python presente come applicazione di test all'interno di BlueZ, opportunamente modificato per conseguire il nostro scopo. Lo script, concepito per essere eseguito esternamente al processo demone di IBR-DTN, consente infatti di realizzare il pairing tra i dispositivi in maniera totalmente automatica. Può essere avviato nel seguente modo:

```
$ bt-pairing-agent -i hci0 -c KeyboardOnly
```

Il parametro `-i` indica l'interfaccia Bluetooth a cui associare il pairing agent. Se tale parametro è omesso, il pairing agent viene associato alla prima interfaccia disponibile. Il parametro `-c` deve essere seguito dalle capacità di I/O del dispositivo. In questo caso, ad esempio, `KeyboardOnly` indica che il dispositivo locale possiede esclusivamente capacità di input. Di conseguenza, la procedura di pairing potrà l'introduzione di una passkey di sei cifre oppure la conferma mediante la pressione di un tasto. All'arrivo di una richiesta di pairing al dispositivo locale, lo script intercetta la richiesta e, a seconda della tipologia di associazione, risponde opportunamente. Per esempio, nel caso in cui giunga una richiesta di passkey, lo script risponde con un valore fisso. Affinché tutto funzioni correttamente, lo stesso script deve essere eseguito su tutti i dispositivi che eseguono il demone IBR-DTN e che desiderano comunicare su Bluetooth. Il meccanismo implementato, che ha il solo obiettivo di semplificare e automatizzare il pairing tra i dispositivi, espone ad una serie di problematiche di sicurezza, ritenuta tuttavia un aspetto marginale del progetto di tesi. Essendo lo script esterno alla suite IBR-DTN, è decisamente semplice avviarlo dichiarando una capacità di I/O diversa da `KeyboardOnly`, realizzando l'accoppiamento i dispositivi secondo altri meccanismi, che richiedono l'intervento umano, ad esempio.

5.3.6 Configurazione

Al fine di avviare il software IBR-DTN con il supporto a Bluetooth, è necessario ritoccare, all'interno del file di configurazione del processo demone, la parte relativa alle istanze di *convergence layer*. Qui di seguito è illustrato un esempio di configurazione del `Bluetooth Convergence Layer`:

```
# a list (seperated by spaces) of names for convergence layer
    instances.
net_interfaces = bt0

...

# configuration for a convergence layer named bt0
net_bt0_type = bluetooth
net_bt0_interface = hci0
net_bt0_port = 10
```

Con la configurazione qui sopra, si richiede che venga istanziato un *convergence layer*, di nome "bt0", che possa comunicare su Bluetooth e che rimanga in ascolto sull'interfaccia *hci0* e sul canale RFCOMM numero 10. Come si può notare, piuttosto che "channel", viene utilizzata la dicitura "port" per indicare il canale RFCOMM. Non si tratta di un errore, bensì di una scelta dettata dalla volontà di mantenere lo stesso formato per la configurazione di tutti i *convergence layer*, e di conseguenza di non modificare il parsing del file di configurazione. Una volta impostati i parametri di configurazione, per abilitare l'esecuzione del *convergence layer*, è sufficiente che il suo nome compaia in `net_interfaces`, una lista che contiene i nomi dei *convergence layer* da istanziare.

Capitolo 6

Validazione sui dispositivi fisici

Il seguente capitolo descrive lo svolgimento di alcune prove sperimentali, effettuate allo scopo di validare e valutare le prestazioni dell'architettura implementata in questo progetto di tesi, di cui si è ampiamente discusso nel capitolo 5.

6.1 Banco di prova

Per eseguire le prove di validazione, sono stati utilizzati due *Raspberry Pi Model B* con le seguenti caratteristiche:

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
- 1GB RAM
- BCM43438 wireless LAN integrato
- Bluetooth 4.1 (BLE) integrato
- S.O. Raspbian GNU/Linux 9 (stretch)

I dispositivi, inoltre, sono equipaggiati con il software IBR-DTN, e in particolare con la sua versione estesa per offrire supporto alla tecnologia Bluetooth. Entrambi i dispositivi sono configurati per eseguire il modulo *Bluetooth Convergence Layer*, per cui, la comunicazione tra essi avviene utilizzando il protocollo RFCOMM sulla tecnologia Bluetooth. In tutti gli esperimenti, l'inoltro dei bundle è basato su rotte statiche (a livello DTN), incluse all'interno del file di configurazione del demone IBR-DTN presente sui dispositivi.

Prima di cominciare con la descrizione degli esperimenti svolti, è opportuno fare una precisazione. Essendo Bluetooth una tecnologia di tipo wireless, potenzialmente instabile, i risultati ottenuti nelle prove sperimentali sono influenzati da una serie di fattori, come le interferenze o la quantità di traffico istantaneo ricevuto dall'interfaccia fisica (in molti casi corrispondente a traffico generato da dispositivi esterni, non facenti parte del banco di prova). Questo implica che replicando le prove sperimentali più volte, si ottengono risultati potenzialmente diversi tra loro.

6.2 Instaurazione di una connessione

L'instaurazione di una connessione RFCOMM è un'operazione piuttosto onerosa, in quanto richiede lo scambio di una serie di informazioni tra le entità coinvolte, prima che il flusso dati possa iniziare. La figura 6.1 mostra una cattura effettuata su Wireshark del traffico scambiato tra due dispositivi (precedentemente autenticati tramite la procedura di pairing) al fine di instaurare una connessione RFCOMM su Bluetooth. La cattura è stata effettutata sul dispositivo che riceve la richiesta di connessione.

Source	Destination	Protocol	Length	Info
controller	host	HCI_EVT	13	Rcvd Connect Request
host	controller	HCI_CMD	11	Sent Accept Connection Request
controller	host	HCI_EVT	7	Rcvd Command Status (Accept Connection Request)
controller	host	HCI_EVT	11	Rcvd Role Change
controller	host	HCI_EVT	14	Rcvd Connect Complete
host	controller	HCI_CMD	6	Sent Read Remote Supported Features
controller	host	HCI_EVT	6	Rcvd Max Slots Change
controller	host	HCI_EVT	7	Rcvd Command Status (Read Remote Supported Features)
controller	host	HCI_EVT	14	Rcvd Read Remote Supported Features
host	controller	HCI_CMD	7	Sent Read Remote Extended Features
controller	host	HCI_EVT	7	Rcvd Command Status (Read Remote Extended Features)
controller	host	HCI_EVT	16	Rcvd Read Remote Extended Features Complete
host	controller	HCI_CMD	14	Sent Remote Name Request
localhost ()	Raspberr_97:78:73 (raspberrypi1)	L2CAP	15	Sent Information Request (Extended Features Mask)
controller	host	HCI_EVT	7	Rcvd Command Status (Remote Name Request)
controller	host	HCI_EVT	258	Rcvd Remote Name Request Complete
controller	host	HCI_EVT	9	Rcvd Link Key Request
host	controller	HCI_CMD	26	Sent Link Key Request Reply
controller	host	HCI_EVT	13	Rcvd Command Complete (Link Key Request Reply)
controller	host	HCI_EVT	7	Rcvd Encryption Change
host	controller	HCI_CMD	6	Sent Read Encryption Key Size
controller	host	HCI_EVT	10	Rcvd Command Complete (Read Encryption Key Size)
Raspberr_97:78:73 (ras…	localhost ()	L2CAP	15	Rcvd Information Request (Extended Features Mask)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	L2CAP	21	Sent Information Response (Extended Features Mask, Success)
Raspberr_97:78:73 (ras…	localhost ()	L2CAP	21	Rcvd Information Response (Extended Features Mask, Success)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	L2CAP	15	Sent Information Request (Fixed Channels Supported)
controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
Raspberr_97:78:73 (ras…	localhost ()	L2CAP	15	Rcvd Information Request (Fixed Channels Supported)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	L2CAP	25	Sent Information Response (Fixed Channels Supported, Success)
Raspberr_97:78:73 (ras…	localhost ()	L2CAP	25	Rcvd Information Response (Fixed Channels Supported, Success)
controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
Raspberr_97:78:73 (ras…	localhost ()	L2CAP	17	Rcvd Connection Request (RFCOMM, SCID: 0x0040)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	L2CAP	21	Sent Connection Response - Success (SCID: 0x0040, DCID: 0x0040)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	L2CAP	32	Sent Configure Request (DCID: 0x0040)
controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
Raspberr_97:78:73 (ras…	localhost ()	L2CAP	32	Rcvd Configure Request (DCID: 0x0040)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	L2CAP	23	Sent Configure Response - Success (SCID: 0x0040)
Raspberr_97:78:73 (ras…	localhost ()	L2CAP	23	Rcvd Configure Response - Success (SCID: 0x0040)
Raspberr_97:78:73 (ras…	localhost ()	RFCOMM	13	Rcvd SABM Channel=0
localhost ()	Raspberr_97:78:73 (raspberrypi1)	RFCOMM	13	Sent UA Channel=0
controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
Raspberr_97:78:73 (ras…	localhost ()	RFCOMM	23	Rcvd UIH Channel=0 -> 10 MPX_CTRL DLC Parameter Negotiation (PN)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	RFCOMM	23	Sent UIH Channel=0 -> 10 MPX_CTRL DLC Parameter Negotiation (PN)
Raspberr_97:78:73 (ras…	localhost ()	RFCOMM	13	Rcvd SABM Channel=10 (Unknown)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	RFCOMM	13	Sent UA Channel=10
localhost ()	Raspberr_97:78:73 (raspberrypi1)	RFCOMM	17	Sent UIH Channel=0 -> 10 MPX_CTRL Modem Status Command (MSC)
controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
Raspberr_97:78:73 (ras…	localhost ()	RFCOMM	17	Rcvd UIH Channel=0 -> 10 MPX_CTRL Modem Status Command (MSC)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	RFCOMM	17	Sent UIH Channel=0 -> 10 MPX_CTRL Modem Status Command (MSC)
Raspberr_97:78:73 (ras…	localhost ()	RFCOMM	17	Rcvd UIH Channel=0 -> 10 MPX_CTRL Modem Status Command (MSC)
localhost ()	Raspberr_97:78:73 (raspberrypi1)	RFCOMM	14	Sent UIH Channel=10 UID
localhost ()	Raspberr_97:78:73 (raspberrypi1)	RFCOMM	40	Sent UIH Channel=10
controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
Raspberr_97:78:73 (ras…	localhost ()	RFCOMM	14	Rcvd UIH Channel=10 UID
controller	host	HCI_EVT	8	Rcvd Number of Completed Packets
Raspberr_97:78:73 (ras…	localhost ()	RFCOMM	40	Rcvd UIH Channel=10
Raspberr_97:78:73 (ras…	localhost ()	RFCOMM	165	Rcvd UIH Channel=10
localhost ()	Raspberr_97:78:73 (raspberrypi1)	RFCOMM	16	Sent UIH Channel=10
localhost ()	Raspberr_97:78:73 (raspberrypi1)	RFCOMM	165	Sent UIH Channel=10

1 2 3 4 5

Figura 6.1: Cattura del traffico scambiato tra due dispositivi per instaurare una connessione RFCOMM

Dato che le trame RFCOMM sono imbustate all'interno di pacchetti L2CAP, l'instaurazione di una connessione RFCOMM comincia con la creazione di un canale logico L2CAP tra i due dispositivi, che avviene secondo le procedure previste dal

servizio L2CAP, ed include, tra le tante cose, lo scambio delle *Link Key* calcolate al momento del pairing. Il protocollo RFCOMM è identificato da un *Protocol Service Multiplexer (PSM)* riservato (`0x0003`), che permette ad L2CAP di discriminare il traffico RFCOMM. Per questo motivo, alla ricezione di una trama L2CAP contenente tale valore per il PSM, questa sarà trasmessa al livello RFCOMM per il processamento.

In figura 6.1, sono state evidenziate le parti che coinvolgono lo scambio di trame proprie del protocollo RFCOMM. La prima trama inviata dall'entità che ha iniziato la procedura di connessione è la trama SABM, necessaria ad iniziare l'instaurazione del canale di controllo (Channel = 0). Per accettare l'instaurazione del canale, il dispositivo ricevitore risponde con una trama UA (1). Nel caso in cui volesse rifiutare la connessione, esso risponderebbe inviando una trama DM. La procedura di accettazione è seguita dallo scambio di trame di tipo *Parameter Negotiation (PN)* (2), che concludono la creazione del canale di controllo. A questo punto, il protocollo RFCOMM prevede la creazione di un ulteriore canale, quello dati (Channel = 10), la cui instaurazione segue lo stesso scambio di trame svolto in precedenza (3). Infine, i dispositivi scambiano le trame *Modem Status Commands (MCS)* (4), necessarie al controllo di flusso sul canale appena creato. A questo punto comincia la trasmissione dei dati veri e propri, imbustati all'interno delle trame di tipo UIH (Channel = 10) (5).

Facendo riferimento alla cattura qui sopra, per il calcolo del tempo necessario ad instaurare una connessione RFCOMM, è stato considerato l'intervallo di tempo che intercorre dal momento in cui viene generato l'evento che notifica la richiesta di connessione (`Rcvd Connect Request`), fino alla ricezione del primo pacchetto dati. In particolare, sono state effettuate 10 diverse prove, da cui si è evinto che, nel 99% dei casi, il tempo di instaurazione di una connessione RFCOMM cade nell'intervallo tra i 362,3 e i 378,5 millisecondi. Tali risultati sono stati calcolati considerando dispositivi che hanno svolto la procedura di pairing prima di instaurare la connessione. Invece, nel caso di dispositivi non ancora associati, il tempo di instaurazione della connessione cresce in maniera significativa, in quanto, ai tempi precedenti, è necessario sommare il tempo necessario all'espletamento della procedura di pairing. In quest'ultimo caso, considerando una procedura di *Secure Simple Pairing* 3.3 basata sullo scambio di *passkey*, il tempo di instaurazione della connessione è dell'ordine del secondo, e in particolare, oscilla tra i 1602,2 e i 1691,1 millisecondi.

Dai risultati ottenuti si desume che la procedura necessaria all'instaurazione di una connessione RFCOMM su Bluetooth sia molto onerosa in termini di tempo. A tal proposito, è opportuno sottolineare il fatto che la soluzione realizzata è progettata in modo che, dal momento in cui viene instaurata una connessione, questa viene mantenuta attiva fino ad esplicita chiusura da parte di uno dei due peer coinvolti oppure finché essa cade, a causa di un eccessivo allontanamento dei dispositivi, ad esempio. Tale approccio consente di evitare che ad ogni nuovo trasferimento dati i due dispositivi debbano instaurare nuovamente la connessione, con tutto l'overhead che ne consegue.

6.3 Throughput

Quanto descritto in questa sezione mira a valutare il throughput di rete nella comunicazione tra due dispositivi connessi, al variare della dimensione del segmento utilizzata per la trasmissione e ricezione dei bundle tramite il *Bluetooth Convergence Layer*. Di base, la dimensione di tale segmento indica il numero di byte che si desidera trasmettere in ogni blocco dati. Chiaramente, l'utilizzo di un segmento più grande implica un overhead di processamento più basso. Inoltre, dato che la ricezione di ogni segmento dati viene confermata tramite un segmento di ACK da parte del ricevitore, un segmento più grande implica un minore overhead di processamento dei segmenti di ACK. La figura 6.2 mostra il throughput medio misurato nel trasferimento di un file di dimensione 1MB, al variare della dimensione del segmento dati.

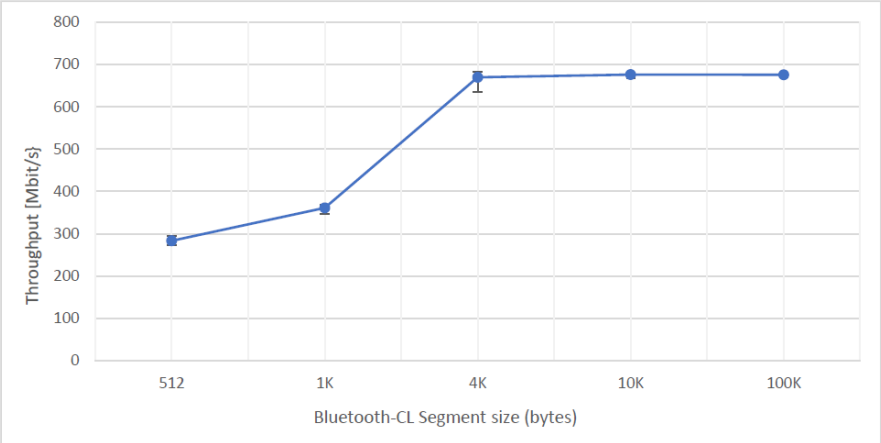


Figura 6.2: Throughput medio al variare della dimensione del segmento dati utilizzato dal *Bluetooth Convergence Layer*

6.4 Ping DTN

Gli esperimenti descritti nella seguente sezione sono stati svolti utilizzando il tool `dtnping` (vedi A.2.2), integrato nella suite IBR-DTN, che implementa un ping a livello di Bundle Protocol. Più precisamente, il tool consente di creare un bundle destinato ad uno specifico EID (`dtn://node-name/echo`) e mettersi in attesa della risposta. Alla ricezione di tale bundle, il nodo destinatario costruisce un bundle con lo stesso payload di quello ricevuto, diretto alla sorgente di quest'ultimo. Le prove sperimentali effettuate, che coinvolgono la trasmissione di 1500 bundle con payload di 64 byte, intendono valutare il *Round Trip Time (RTT)* medio, ovvero l'intervallo di tempo che intercorre tra l'invio di un bundle e la ricezione del bundle di risposta.

Il calcolo del RTT medio è stato effettuato considerando due diverse configurazioni. In un caso, le due Raspberry Pi sono state configurate per trasmettere bundle tramite RFCOMM sulla tecnologia Bluetooth, nell'altro mediante TCP/IP su una

rete WiFi ad Hoc. La tabella seguente mostra i risultati ottenuti in entrambe le configurazioni.

RTT [ms]	MIN	AVG	MAX
Bluetooth via RFCOMM	23,08	52,98	87,52
WiFi ad Hoc via TCP/IP	12,32	32.52	284,74

Durante la trasmissione dati, si è tenuto traccia del carico di CPU medio misurato sui dispositivi. In entrambe le configurazioni realizzate è emerso un carico di CPU medio decisamente basso e pressoché simile, pari a circa 2,6 %. Questo dato è la dimostrazione pratica di uno dei principi cardine di IBR-DTN, il basso impatto sulle risorse dei dispositivi, che la rendono una soluzione particolarmente adeguata per sistemi con risorse hardware limitate.

Capitolo 7

Conclusioni

Questo progetto di tesi ha portato alla prototipazione di un'architettura, basata sul paradigma del Delay-Tolerant Networking, che consente l'interconnessione di reti eterogenee nell'ambito delle challenged network. L'architettura realizzata, utilizzando un approccio di tipo store-and-forward, permette di sfruttare i contatti occasionali che vengono ad instaurarsi tra i nodi della rete, al fine di garantire la consegna a destinazione dei messaggi nonostante il fenomeno della connettività intermittente.

L'obiettivo fondamentale della soluzione proposta, concepita principalmente per sistemi embedded con risorse limitate, è quello di garantire la trasparenza nei confronti delle tecnologie e dei protocolli utilizzati per il trasporto dei dati. In tal senso, essa prevede l'estensione dello stack di rete di ognuno dei nodi partecipanti alla DTN, tramite l'introduzione di un nuovo livello protocollare, il bundle layer. Tale strato, in combinazione con uno o più Convergence Layer Adapter, uno per ogni tecnologia e/o protocollo di comunicazione che si desidera supportare, consente ad un nodo di trasferire dati sfruttando le primitive di comunicazione fornite dal Bundle Protocol, senza che esso debba preoccuparsi di come effettivamente avvenga la trasmissione.

L'implementazione del Bundle Protocol utilizzata per lo sviluppo della tesi è IBR-DTN, un progetto open-source portabile e leggero, pensato per essere esteso in maniera flessibile e poco invasiva. Allo scopo di consentire la trasmissione di dati sulle principali tecnologie di comunicazione esistenti in ambito wireless, è stato necessario integrare, al software IBR-DTN, il supporto allo stack protocollare Bluetooth. Tuttavia, la soluzione realizzata consente esclusivamente la trasmissione su Bluetooth via RFCOMM, che implementa un servizio di trasmissione affidabile, connection-oriented e stream-based. Si potrebbe pensare, in futuro, di offrire diversi servizi di trasmissione di dati su Bluetooth, in modo da soddisfare le esigenze dei diversi contesti applicativi. Inoltre, data la sempre crescente diffusione di nuove tecnologie di comunicazione wireless, in particolar modo in ambito IoT (LoRa, ad esempio), un ulteriore sviluppo futuro potrebbe consistere nell'integrazione, alla soluzione realizzata in questa tesi, del supporto ai nuovi standard.

Appendice A

Guida all'utilizzo di IBR-DTN

Il contenuto di questa appendice rappresenta una guida all'utilizzo di IBR-DTN. In particolare, la prima parte include le istruzioni per l'installazione, la configurazione e l'avvio del software; la seconda, invece, descrive alcuni strumenti utili per muovere i primi passi con IBR-DTN.

A.1 Installazione, configurazione e avvio

La seguente sezione elenca le istruzioni per la compilazione e l'installazione del software IBR-DTN a partire dai sorgenti, per la configurazione e l'avvio del processo demone.

A.1.1 Compilazione e installazione

Come accennato poco fa, le seguenti istruzioni permettono di compilare ed installare la suite IBR-DTN a partire dai sorgenti. Le procedure qui riportate sono valide per distribuzioni Linux, Debian e derivate (Raspbian).

Come prima cosa, è necessario installare le librerie necessarie. Sebbene l'installazione di alcune librerie elencate in seguito sia opzionale, in quanto necessarie soltanto per i moduli opzionali, è consigliabile installarle comunque, onde evitare problemi durante la compilazione.

```
$ apt-get install build-essential libssl-dev zlib1g-dev
   libsqlite3-dev libcurl4-gnutls-dev libdaemon-dev automake
   autoconf pkg-config libtool libcppunit-dev libnl-3-dev
   libnl-cli-3-dev libnl-genl-3-dev libnl-nf-3-dev
   libnl-route-3-dev libarchive-dev
```

Una volta che le librerie sono state installate, si procede con la compilazione e l'installazione vera e propria. Per ottenere i sorgenti, si effettui il `clone` del repository Github ufficiale di IBR-DTN [18], digitando il seguente comando:

```
$ git clone https://github.com/ibrdtn/ibrdtn.git
   ibrdtn-repository
```

A questo, si prosegua con le seguenti istruzioni:

```
$ cd ibrdtn-repository/ibrdtn
$ ./autogen.sh
$ ./configure
$ make
$ sudo make install
$ ldconfig
```

E' possibile includere, alla versione base di IBR-DTN, una serie di moduli opzionali, specificandoli come parametri del comando `configure`, utilizzando la notazione `--with-module`. Ad esempio, il flag `--with-sqlite` aggiunge il supporto a SQLite come meccanismo di storage dei bundle, mentre `--with-openssl` abilita l'utilizzo di TLS.

A.1.2 File di configurazione

Per modificare il comportamento predefinito del demone IBR-DTN è necessario specificare i parametri da utilizzare all'interno di un file di configurazione. Un esempio di configurazione è reperibile al path `ibrdtn/daemon/etc/ibrdtnd.conf`, all'interno del repository utilizzato per l'installazione, o all'indirizzo [19]. Di seguito ne sono illustrate le parti più significative.

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

permette di personalizzare l'EID locale.

```
# define the interface for the API, choose any to bind on all
   interfaces
api_interface = any

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

permette di modificare l'interfaccia di rete e la porta TCP alla quale il demone è in ascolto e può essere contattato tramite API.

```
# 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 del file-system in cui memorizzare i bundle in maniera persistente.

```
# defines the storage module to use
# default is "simple" using memory or disk (depending on storage_path)
# storage strategy. if compiled with sqlite support, you could change
# this to sqlite to use a sql database for bundles.
storage = default
```

definisce il meccanismo di storage da utilizzare.

```
# Specify how often discovery beacons are sent. The default is every 5
    seconds.
discovery_interval = 5
```

indica la cadenza temporale con cui inviare i beacon di discovery.

```
# 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   # with 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   # with port 4556 (default)
```

elenca i converge layer da instanziare con gli opportuni parametri (protocollo, interfaccia di rete, porta).

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

specifica l'algoritmo di routing da utilizzare scegliendo tra le opzioni mostrate nei commenti.

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

abilita/disabilita l'inoltro di bundle da parte del nodo.

```
# forward singleton bundles directly if the destination is a neighbor
routing_prefer_direct = yes
```

abilita/disabilita l'inoltro diretto alla destinazione di un bundle se questa è raggiungibile direttamente.

A.1.3 Avvio

Supponendo di aver completato correttamente l'installazione, è possibile avviare il demone IBR-DTN utilizzando il comando `dtnd`. Tale comando, invocato senza

parametri, avvia il demone utilizzando la configurazione di default. Per avviare il demone con la configurazione specificata all'interno di un file, si invochi il comando `dtnd` con il flag `-c`, seguito dal path del file. E' possibile, inoltre, utilizzare l'opzione `-i` per specificare l'interfaccia di rete alla quale associare il processo demone, oppure `-v` per abilitare la stampa dei messaggi di log. Ad esempio, il comando seguente avvia il processo demone sull'interfaccia *eth0*, con la configurazione specificata all'interno del file *ibrdtnd.conf*:

```
$ dtnd -i eth0 -c ibrdtnd.conf
```

Per un elenco completo delle opzioni disponibili, si invochi il comando `dtnd` con il flag `-h`.

Una volta avviato, il demone IBR-DTN rileverà automaticamente la presenza dei processi demone in esecuzione su nodi direttamente raggiungibili dal nodo locale, grazie all'uso del modulo `IPND Agent`. Simultaneamente, il demone comincerà ad annunciare il suo EID locale, in modo da essere scoperto dagli altri. Secondo la configurazione di default, l'EID locale utilizza lo schema DTN (dtn://) ed assume la forma `dtn://hostname`, dove `hostname` rappresenta il nome della macchina.

A.2 Applicazioni

Il contenuto di questa sezione è utile qualora si vogliano muovere i primi passi con il software IBR-DTN. In particolare, viene fornito un esempio di interazione con la API di IBR-DTN, oltre che la descrizione di alcuni strumenti utili per testare le funzionalità di un *bundle node*.

A.2.1 API

Le applicazioni non integrate nel *bundle node* necessitano di accedere alle primitive da esso esposte. A tal proposito, la libreria `ibrdtn` fornisce delle classi per creare e gestire connessioni verso la API di IBR-DTN. Tipicamente, la API è basata su testo human-readable, ma è possibile utilizzare diversi protocolli e di conseguenza, diversi schemi di codifica. Un modo semplice per instaurare una connessione con la API di IBR-DTN è di utilizzare `telnet`, come mostrato nell'esempio seguente:

```
$ telnet localhost 4550
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
IBR-DTN 0.11.0 (build dfb7402) API 1.0
protocol management
200 SWITCHED TO MANAGEMENT
neighbor list
200 NEIGHBOR LIST
dtn://neighbor1
dtn://neighbor2
```

Di default, il demone IBR-DTN è in ascolto localmente sulla porta 4550. Nel momento in cui viene instaurata una connessione col *bundle node*, la API mostra un messaggio contenente la versione del software e della API. A questo punto, è possibile interagire con la API utilizzando diversi protocolli. Per passare da un protocollo ad un altro, è necessario di digitare il comando `protocol`, seguito dal nome del protocollo. Nel caso qui sopra, il comando `protocol management` permette di accedere alla API di management, che fornisce una serie funzioni di gestione per *il bundle* (visualizzazione del vicinato, visualizzazione/cancellazione dei bundle nello storage, creazione/rimozione di connessioni statiche, ...). Nell'esempio, il comando `neighbor list` visualizza tutti i nodi appartenenti al vicinato del nodo locale.

Ad ogni connessione verso la API, è associato un oggetto di tipo `Registration`, che permette al client di gestire le sottoscrizioni ad un EID, di inviare e ricevere bundle. Di default, la registrazione è volatile, in quanto viene distrutta nel momento in cui la connessione verso la API termina. Tuttavia, il client può evitare questo comportamento, forzando la registrazione ad essere persistente. In tal caso, infatti, la registrazione rimarrà attiva fino ad un'esplicita disconnessione da parte del client. Nel momento in cui il client si connetterà nuovamente alla API, potrà recuperare la registrazione sospesa in precedenza, usando un identificativo univoco.

A.2.2 Strumenti utili

La suite IBR-DTN mette a disposizione una serie di programmi a linea di comando per la gestione e il testing di un *bundle node* . Tali strumenti, inclusi all'interno del package `tools`, permettono di prendere confidenza con l'uso del software IBR-DTN e di avere un'idea di come si scrivano applicazioni che interagiscono con esso. Qui di eseguito ne sono presentati alcuni.

dtnping

Questo tool invia un bundle ad uno specifico EID destinazione e si mette in attesa di una risposta, contenente lo stesso payload del messaggio inviato. Il tool, inoltre, tiene traccia del RTT per ogni trasmissione, ovvero il periodo di tempo che intercorre tra l'invio del bundle e la ricezione della risposta. Di default, ogni trasmissione coinvolge 64 byte di payload e ha un *lifetime* di 30 secondi. Di conseguenza, nel caso in cui la risposta alla trasmissione di un bundle non sia ricevuta entro due volte il *lifetime*, il bundle è considerato perso e si passa ad una nuova trasmissione.

dtnsend e dtnrecv

`dtnsend` permette di inviare bundle con un dato payload ad un certo EID destinazione. In combinazione con `dtnrecv`, consente di realizzare piccole applicazioni che effettuano trasferimenti di bundle. In particolare, `dtnrecv` è usato per ricevere il payload di uno o più bundle. L'output viene generato solamente a seguito dell'avvenuta ricezione del bundle. Tramite il parametro `name`, seguito da una stringa, il comando `dtnrecv` effettua la registrazione all'EID del nodo locale, usando come nome per l'applicazione la suddetta stringa.

Appendice B

Validazione del DTN Bundle Protocol in ambiente virtualizzato

Il contenuto di questa appendice espone un lavoro preliminare allo sviluppo della tesi, volto a validare il comportamento del DTN Bundle Protocol in contesti applicativi realistici, come, ad esempio, un gruppo di dispositivi che comunicano tra loro tramite connessioni opportunistiche. L'attività svolta è un proseguimento del lavoro di tesi sviluppato da Luigi Maio, in cui viene proposto il prototipo di un sistema di telemetria basato sul paradigma del *Fog Computing* nell'ambito del *Delay-/Disruption-Tolerant Networking*. Al fine di valutare le prestazioni del DTN Bundle Protocol in scenari caratterizzati da un numero significativo di nodi, è stato realizzato un ambiente di simulazione virtuale che emula un sistema di telemetria. Il repository Github [20] contiene il materiale impiegato per effettuare le simulazioni.

B.1 Tecnologie utilizzate

La seguente sezione descrive brevemente il software e gli strumenti tecnologici utilizzati per la realizzazione dell'ambiente di simulazione virtuale.

B.1.1 Docker

Docker [21] è una piattaforma software basata sull'approccio a container. Il container è un'unità standardizzata che racchiude al suo interno tutto il necessario all'esecuzione del software, inclusi codice, librerie, tool di sistema e impostazioni. Un container può essere pensato come una sorta di macchina virtuale "leggera", anche se le due cose sono concettualmente diverse. Infatti, mentre una macchina virtuale emula l'hardware, un container virtualizza un sistema operativo. I container in esecuzione sulla stessa macchina condividono il kernel e l'hardware, con un overhead pressoché simile a quello dei processi eseguiti direttamente sulla macchina. Questo approccio permette un isolamento tra i container analogo a quello tra macchine virtuali, ma ad un costo inferiore.

Docker può essere considerato lo standard de facto per la creazione e il deployment di container tra Docker host. Introduce un alto livello di portabilità, permettendo l'esecuzione di uno stesso container su Docker host differenti, indipendentemente dal sistema operativo sottostante. Docker ha sia una versione open source che una Enterprise. Il componente fondamentale è il *Docker Engine*, che sarebbe l'analogo di un hypervisor per le macchine virtuali. Si occupa di costruire ed eseguire container. Consiste in un processo demone, una REST API e una command-line interface (CLI). Quest'ultima permette agli utenti di interagire con il processo demone tramite l'API.

Una guida completa all'utilizzo di Docker è reperibile nella documentazione ufficiale.

B.1.2 Universal Node

L'orchestratore *Universal Node* è un progetto sviluppato dal *Netgroup* del Politecnico di Torino. Può essere pensato come una sorta di "datacenter in una scatola", e in tal senso fornisce funzionalità simili al cluster OpenStack, ma limitate ad un singolo server. Fondamentalmente, lo Universal Node riceve richieste di servizio e si occupa dell'orchestrazione e allocazione delle risorse di computing e di rete necessarie a realizzare il servizio richiesto, gestendo il ciclo di vita dei container di computing (macchine virtuali, container Docker, ...) e delle primitive di rete (regole OpenFlow, istanziazione di switch virtuali, ...). L'orchestratore riceve i comandi tramite una REST API, espressi mediante il formalismo del *Network Functions Forwarding Graph (NF-FG)*, e si occupa di implementarli sul nodo fisico. Secondo il formalismo NF-FG, un servizio va espresso in termini di *Virtual Network Functions (VNFs)* e interconnessioni tra esse.

Universal Node è costituito da diversi moduli, di cui i principali sono il *compute controller* e il *network controller*. Il network controller, che interagisce con lo switch virtuale (vSwitch), è composto, a sua volta, da due elementi: un *controller OpenFlow* per ogni switch logico istanziato, che permette al traffico di fluire attraverso le porte, e uno *switch manager*, che crea e distrugge gli switch logici, le porte virtuali ed altro. La versione corrente di Universal Node supporta OpenvSwitch (OvS), l'extensible Data Path daemon (xDPd) e l'Ericsson Research Flow Switch (ERFS) come implementazioni dello switch virtuale. Dall'altra parte, il compute controller interagisce con l'ambiente di esecuzione virtuale (l'hypervisor, ad esempio) e gestisce l'intero ciclo di vita delle VNFs (creazione, aggiornamento, distruzione), incluse le operazioni necessarie a collegare una VNF alle porte dello switch virtuale. Attualmente l'orchestratore supporta VNFs implementate sotto forma di macchine virtuali, container Docker, processi DPDK e funzioni native. Nella pratica possono essere disponibili solo alcune implementazioni, a seconda dello switch virtuale scelto.

Per maggiori informazioni sull'istallazione e l'utilizzo di Universal Node, si faccia riferimento alla documentazione presente nel repository Github ufficiale [22].

B.1.3 RabbitMQ

RabbitMQ è un broker di messaggi open-source che implementa l'*Advanced Message Queueing Protocol (AMQP)*, un protocollo applicativo che definisce un meccanismo standard per lo scambio di messaggi. Un broker di messaggi è un modulo che convalida, elabora ed esegue il routing di messaggi tra diverse applicazioni. L'utilizzo di un broker realizza un disaccoppiamento tra le applicazioni, poiché, ponendosi "in mezzo", riduce al minimo la consapevolezza reciproca che le applicazioni dovrebbero avere per scambiare dei messaggi.

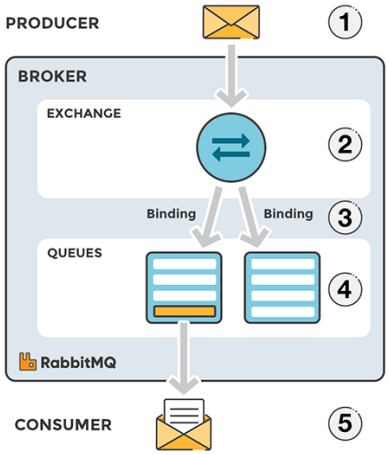


Figura B.1: Flusso dei messaggi in *RabbitMQ*

In uno scenario classico di scambio di messaggi, RabbitMQ definisce, oltre alle entità *Producer*, *Consumer* e *Queue* tipiche del protocollo AMQP, l'elemento *Exchange* (figura B.1). Di base, RabbitMQ consente di definire delle code, a cui le applicazioni possono collegarsi per trasferire messaggi ad esso. Più precisamente, quando un applicativo *Producer* invia un messaggio, questo non passa direttamente alla coda, ma all'oggetto *Exchange*, il quale crea una comunicazione con la coda attraverso l'operazione di *binding*. Il messaggio è mantenuto nella coda fin quando un'applicazione *Consumer* si connette alla coda e preleva il messaggio.

L'utilizzo di un *Exchange* porta alla definizione di un modello di tipo Publish/-Subscribe, secondo cui il messaggio di un *Producer* viene trasmesso a tutte le code sottoscritte ad uno specifico *Exchange* relativo al *Producer* mittente. E' compito delle entità *Consumer* quello di collegarsi alle suddette code, al fine di ottenere il messaggio.

B.1.4 Stack ELK

Lo *Stack ELK* [23] è una soluzione end-to-end che consente cercare, analizzare e visualizzare in tempo reale informazioni provenienti da diverse sorgente di dati. "*ELK*" è l'acronimo di tre progetti open-source: *Elasticsearch*, *Logstash* e *Kibana*. Sebbene ognuno di questi tre progetti sia stato sviluppato indipendentemente dagli altri, essi sono costruiti in modo da poter interoperare, al fine di costituire un unico stack software.

Figura B.2: Componenti dello *Stack ELK*

Logstash consente di raccogliere, analizzare, aggregare ed indicizzare dati (che possono assumere diversi formati) e memorizzarli in un unico modulo di storage. *Elasticsearch* è un motore di ricerca distribuito e scalabile, dotato di interfaccia REST e basato sul formato JSON. *Kibana* è il motore di visualizzazione dei dati di *Elasticsearch*, in quanto consente di accedere ai dati presenti nello storage tramite una dashboard personalizzabile.

B.1.5 IBR-DTN

IBR-DTN è un'implementazione open-source del DTN Bundle Protocol, leggera e portabile ,scelta per validare il funzionamento del protocollo nell'ambiente di simulazione virtuale. Per una trattazione dettagliata sull'architettura software di IBR-DTN, si rimanda al Capitolo 4.

B.2 Prove sperimentali in ambiente virtualizzato

Il contenuto di questa sezione descrive, in primo luogo, l'ambiente di simulazione virtuale costruito per emulare il sistema di telemetria e, successivamente, mostra i risultati ottenuti dalle prove sperimentali eseguite nel suddetto ambiente.

B.2.1 Ambiente di simulazione

Le simulazioni sono state eseguite su una macchina fisica con le seguenti caratteristiche:

- CPU: Intel Core i5-3450S CPU @ 2.80GHz x 4
- RAM: 8 GB
- OS: Ubuntu 14.04 LTS 64-bit

Architettura

L'architettura proposta per emulare il sistema di telemetria, prevede uno scambio di messaggi basato sul protocollo MQTT, che è basato, a sua volta, sul paradigma Publish/Subscribe. L'idea è quella di distribuire i messaggi nella rete sfruttando i contatti occasionali tra i nodi, durante i quali, essi instaurano delle connessioni e possono scambiare i dati. L'architettura include due tipi di nodi: *sensing edge device* e *fog node*. Un nodo *sensing edge device* emula il comportamento di un dispositivo

equipaggiato con dei sensori, che genera dati casuali e li imbusta, in modo strutturato, all'interno di messaggi MQTT. Un *fog node*, invece, rappresenta un nodo di Fog computing, che rimane in attesa di messaggi MQTT provenienti dai sensori e li inoltra direttamente ad un server RabbitMQ, il quale, implementa un broker di messaggi MQTT. L'obiettivo, dunque, è quello di convogliare i messaggi generati dai *sensing edge device* al *fog node*, sfruttando le funzionalità del *Delay-Tolerant Networking* e conseguentemente, del Bundle Protocol. E' opportuno evidenziare come l'architettura realizzata garantisca la trasparenza nei confronti delle applicazioni, che inoltrano messaggi secondo il protocollo MQTT, senza rendersi conto di essere su una DTN. Tale approccio permette alle applicazioni concepite nativamente per utilizzare il protocollo MQTT, di continuare ad operare come hanno sempre fatto, senza la necessità di apportare delle modifiche ad esse per consentire l'imbustamento dei messaggi all'interno di bundle. Infatti, l'incapsulamento e il decapsulamento dei messaggi MQTT in bundle viene svolto da due *gateway*, dei componenti software "trasparenti", esterni alle applicazioni.

I nodi della rete e i collegamenti tra essi sono stati virtualizzati tramite il sistema di orchestrazione Universal Node, che permette di istanziare un grafo in cui i vertici sono costituiti da VNF e gli archi sono configurabili dinamicamente tramite regole OpenFlow. La creazione e l'aggiornamento del suddetto grafo sono effettuati tramite uno script, che inietta delle regole nell'orchestratore interagendo con la API esposta da quest'ultimo. In tal modo si è potuto emulare il comportamento di una topologia dinamica, in cui i nodi si muovono nel tempo e i contatti tra essi sono opportunistici. Lo Universal Node è configurato in modo da utilizzare Docker come compute controller e OpenvSwitch come network controller. Le VNF che modellano i nodi della rete sono quindi immagini Docker, contenenti al loro interno tutto il software necessario ad emulare il comportamento di dispositivi fisici.

Allo scopo di usufruire delle funzionalità del Delay-Tolerant Networking e del Bundle Protocol, tutti i nodi della rete contengono nella loro immagine Docker il BPA IBR-DTN. Per di più, i nodi di tipo *sensing edge device* contengono un gateway MQTT-to-DTN e un'applicazione che genera messaggi MQTT con dati casuali ad intervalli regolari (ogni 5 secondi). I messaggi MQTT, che tale applicazione crede di inviare al server RabbitMQ remoto, sono intercettati localmente dal gateway (in ascolto sulla stessa porta del server RabbitMQ), che li incapsula all'interno di bundle e li inoltra sulla DTN. Tali bundle sono destinati al gateway DTN-to-MQTT in esecuzione sul *fog node*, il quale, è identificato da un EID predefinito. La consegna a destinazione dei bundle potrà avvenire o per contatto diretto con essa, oppure attraverso una serie di nodi intermedi (di tipo *sensing edge device*) che si prenderanno carico del loro inoltro. Nel momento in cui un bundle è ricevuto dal gateway DTN-to-MQTT in esecuzione sul *fog node*, esso provvede ad estrarre il messaggio MQTT originario e ad inviarlo al server RabbitMQ, istanziato come container Docker indipendente e collegato direttamente al *fog node*. La raccolta, l'aggregazione e la visualizzazione dei messaggi memorizzati in RabbitMQ, avviene mediante l'utilizzo dello stack ELK, istanziato anch'esso come container Docker a sé stante, e collegato direttamente al container RabbitMQ. In particolare, lo stack ELK è configurato per collegarsi ad una coda interna a RabbitMQ, in modo da consumare i dati presenti in esso. Quanto espresso finora è riassunto visivamente nella figura B.3.

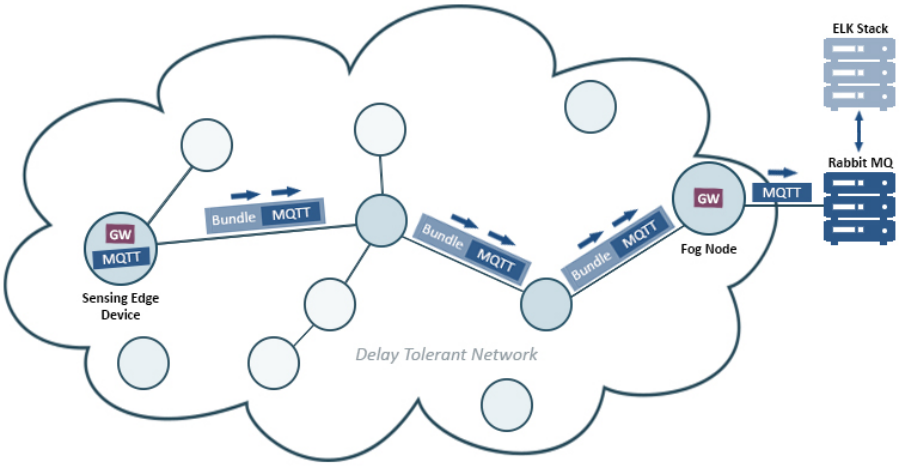


Figura B.3: Visione concettuale dell'ambiente di simulazione

Configurazione IBR-DTN

Il demone IBR-DTN in esecuzione sui nodi della rete è stato configurato in modo da utilizzare il *TCP Convergence Layer*, allo scopo di trasmettere i bundle sullo stack TCP/IP. Per quanto concerne il routing, invece, si è scelto di adottare l'algoritmo *epidemico*, che garantisce un'alta probabilità di consegna dei bundle (oltre che un basso tempo di consegna), ma impatta in maniera significativa sulle risorse della rete. Essendo, infatti, nient'altro che un *flooding* migliorato, l'algoritmo di routing epidemico genera una grande quantità di copie di bundle nella rete e, di conseguenza, richiede una quantità significativa di risorse in termini storage. Pertanto, l'utilizzo di questo algoritmo consente di "stressare" le risorse della rete e di valutarne le prestazioni a fronte della generazione di una grande quantità di traffico. Infine, data la breve durata dei contatti opportunistici tra i nodi durante le simulazioni, si è deciso di impostare un *discovery interval* (intervallo di tempo tra la generazione di due messaggi di discovery) pari ad un secondo. In tal modo, due nodi, nel momento entrano in contatto, impiegano al più un secondo per scoprirsi a vicenda. Per maggiori dettagli sui parametri di configurazione del demone IBR-DTN, si faccia riferimento alla Sezione A.1.2.

B.2.2 Risultati sperimentali

Il contenuto della seguente sezione mostra i risultati ottenuti dalle prove sperimentali svolte nell'ambiente di simulazione virtuale precedentemente descritto. Gli esperimenti svolti hanno una durata di 30 minuti e comprendono 15 nodi di tipo *sensing edge device* e un *fog node*. Durante le prove sperimentali sono state testate diverse configurazioni, allo scopo di identificare i requisti minimi in termini di connessione tra i dispositivi, che permettano al sistema di operare correttamente. In particolare si è variata la probabilità che esista una connessione tra due nodi (probabilità di incontro) tra i valori di 10%, 25% e 40%, e la durata delle connessioni opportunistiche tra i valori di 1, 2, 4 e 6 secondi.

Percentuale e tempo medio di consegna di bundle

Il primo set di esperimenti realizzati ha come scopo quello di valutare la percentuale di bundle consegnati con successo durante il tempo di simulazione e il tempo che mediamente un bundle necessita per giungere a destinazione. Per ognuna delle combinazioni tra i parametri di simulazione (probabilità di incontro e durata dei contatti), sono state eseguite 10 simulazioni, di cui si è tenuto traccia del valore medio, massimo e minimo dei risultati ottenuti.

Per il calcolo della percentuale di bundle consegnati con successo, sono stati conteggiati i messaggi ricevuti dallo stack ELK, e se ne è fatto il rapporto con il numero di messaggi complessivamente generati dai nodi *sensing edge device* durante la simulazione. Invece, per ottenere il tempo che un singolo bundle impiega mediamente per raggiungere la destinazione, sono state utilizzate le informazioni presenti in esso nel momento in cui è ricevuto dallo stack ELK. Ogni messaggio generato presenta, infatti, un campo "*Timestamp*" che ne indica l'istante di creazione; quando il messaggio è ricevuto dallo stack ELK, il modulo Elastisearch vi inserisce un metacampo "*@timestamp*", indicante l'istante di ricezione. La differenza tra i due valori rappresenta il tempo di consegna del bundle.

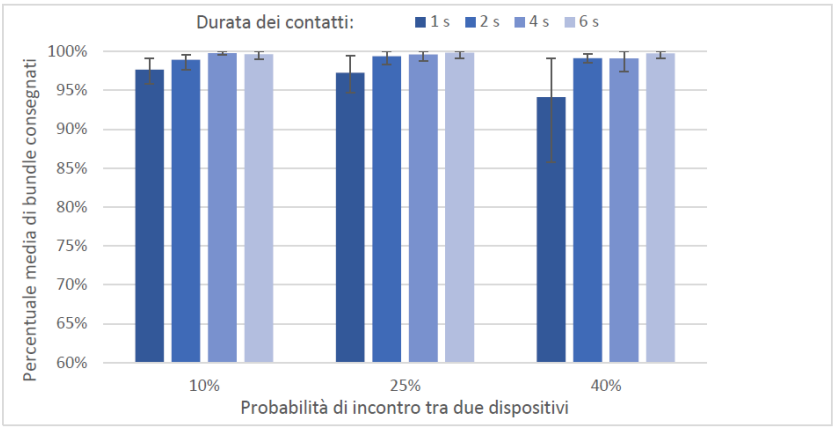


Figura B.4: Percentuale di bundle consegnati durante il tempo di simulazione

La figura B.4 indica la percentuale (media) di bundle consegnati a destinazione durante il tempo di simulazione. Per tutte le simulazioni eseguite, è stato impostato un *lifetime* dei bundle pari a 5 minuti. Dai risultati ottenuti dalle simulazioni, è possibile notare che per connessioni opportunistiche della durata di 1 secondo e per un' alta probabilità di incontro, si ottengono le prestazioni peggiori. Il motivo è da ricercarsi nella troppo scarsa durata dei contatti tra i nodi, tempo durante il quale essi non riescono a scambiarsi tutti i bundle memorizzati al loro interno. Inoltre, all'aumentare della probabilità di contatto tra i nodi, cresce il numero di connessioni che vengono ad instaurarsi, con un maggiore overhead dovuto all'algoritmo di routing epidemico ed un significativo aumento delle copie di bundle in giro per la rete. Infatti, per ognuno dei vicini nell'intorno di un nodo, l'algoritmo epidemico effettua una fase di negoziazione (scambio dei summary vector) e successivamente, lo scambio delle repliche di bundle. Infine, essendo maggiore il numero di copie di bundle ricevute da un nodo, sarà anche maggiore il tempo necessario al loro processamento e memorizzazione. Aumentando la durata delle connessioni

a 2, 4 e 6 secondi, si ottengono risultati visivamente più positivi, con un leggero miglioramento dei valori ottenuti all'aumentare della probabilità di contatto tra i nodi, come si poteva immaginabile intuitivamente. In questi casi, infatti, la quasi totalità dei bundle generati è consegnata con successo (oltre il 98%).

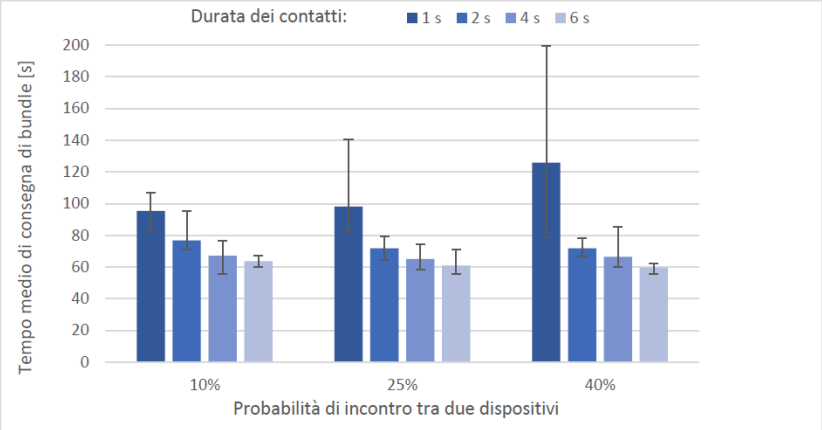
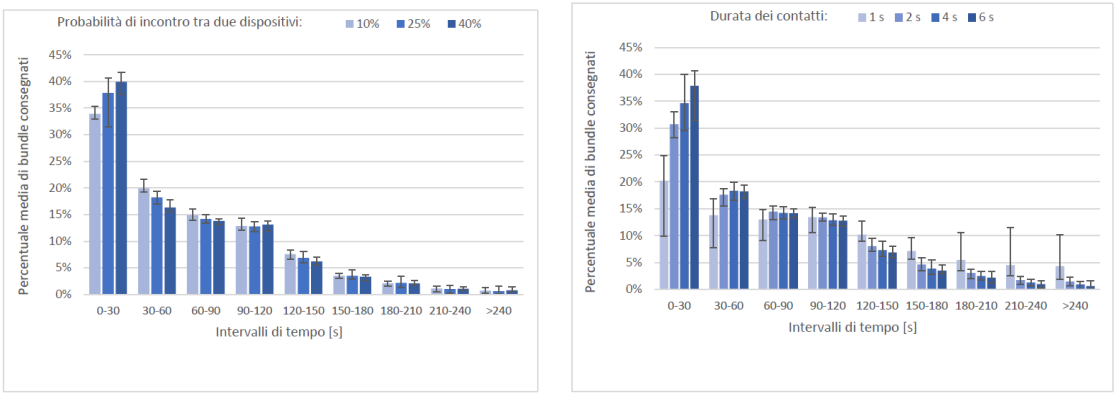


Figura B.5: Tempo medio di consegna di un bundle

La figura B.5 mostra il tempo medio di consegna di un bundle. Anche in questo caso, valgono le stesse considerazioni sull'overhead causato dal routing epidemico fatte in precedenza. E' possibile notare, infatti, un comportamento anomalo del sistema nei casi in cui la durata dei contatti è pari a 1 secondo, dove il tempo medio di consegna cresce al crescere della probabilità di contatto, fino a superare i 2 minuti nel caso peggiore. Risultati sperimentalmente più sensati si ottengono, invece, nei restanti casi, in cui il tempo medio di consegna oscilla tra i 60 e gli 80 secondi all'incirca.

Dalle simulazioni effettuate, è emerso che i messaggi generati negli ultimi istanti di simulazione, tranne che per qualche caso fortunato, difficilmente giungono a destinazione entro la fine del tempo di simulazione. Al fine di ottenere dei risultati statistici sensati, i messaggi generati negli ultimi 5 minuti sono stati esclusi dal calcolo dei dati di interesse. Ne consegue che i messaggi generati negli istanti poco precedenti agli ultimi 5 minuti di simulazione, o raggiungeranno la destinazione nell'arco dei 5 minuti successivi, oppure non la raggiungeranno mai, in quanto terminerà il loro tempo di vita.

Le figure B.6(a) e B.6(b) mostrano la frequenza di distribuzione dei bundle consegnati nei diversi intervalli temporali, rispettivamente variando la probabilità di incontro tra i nodi e mantenendo invariata la durata dei contatti a 6 secondi nel primo caso, variando la durata dei contatti e fissando la probabilità di incontro al 25% nel secondo. Dalla figura B.6(a), emergono risultati interessanti per quasi tutte le configurazioni create, come ad esempio, che circa un terzo dei bundle generati sono consegnati nell'arco di 30 secondi, mentre la fetta maggiore di essi necessita di al più 3 minuti per raggiungere la destinazione. Aumentando la probabilità di contatto tra i nodi, cresce visibilmente la percentuale di bundle recapitati entro 30 secondi. La figura B.6(b) mostra come per durate dei contatti inferiori, decresce anche la percentuale di bundle consegnata nei primi 30 secondi, ma non cambia il fatto che quasi la totalità dei bundle sia recapitata in non più di 3 minuti. Fa



(a) (b)

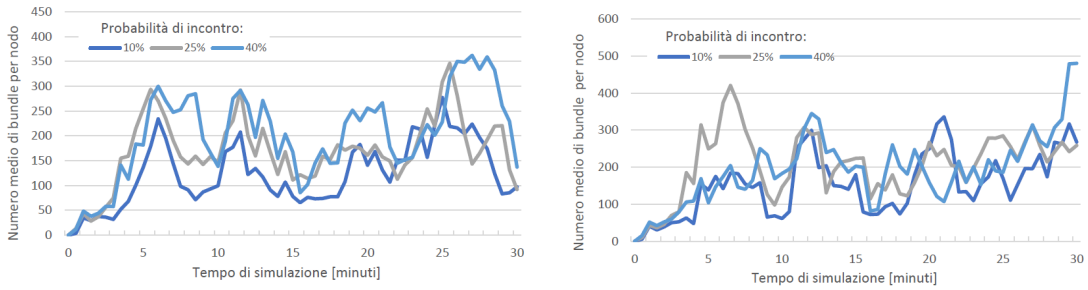
Figura B.6: Distribuzione dei bundle consegnati negli intervalli di tempo, variando la probabilità di incontro tra due dispositivi (a) e la durata dei contatti opportunistici (b)

eccezione il caso in cui la durata dei contatti è impostata a 1 secondo, in cui la distribuzione dei bundle tra gli intervalli di tempo è più uniforme rispetto ai casi precedenti.

Requisiti di storage

La seconda parte degli esperimenti svolti si propone valutare i requisiti in termini di storage che un dispositivo deve possedere per poter usufruire delle funzionalità del Bundle Protocol, oltre che l'overhead dovuto all'utilizzo dell'algoritmo di routing epidemico. Lo script di simulazione utilizzato per il calcolo delle statistiche relative a percentuale di bundle consegnati e tempo di consegna è stato leggermente modificato al fine di poter misurare il numero di repliche di bundle memorizzate sui nodi della rete durante le simulazioni. I bundle memorizzati su ciascun nodo corrispondono sia a bundle generati dal nodo stesso, che a repliche di bundle da inoltrare per conto di altri nodi. Periodicamente, un thread istanziato dallo script di simulazione interroga tutti i nodi della rete, in modo da rilevare il numero di copie di bundle memorizzate in ognuno di essi. Tali repliche sono reperibili nella directory del filesystem specificata nel file di configurazione del demone IBR-DTN (*storage path*). Durante la fase di conteggio, i nodi vengono isolati tra loro (eliminando tutti i collegamenti del grafo) e viene esplicitamente interrotta la generazione dei dati, al fine di evitare che alcuni bundle vengano inoltrati durante la misurazione, ottenendo, di conseguenza, un risultato potenzialmente errato. Prima di cominciare con il conteggio, viene tenuta traccia del grafo dei collegamenti corrente, in modo da ripristinarlo a misurazione terminata.

La figure B.7(a) e B.7(b) mostrano il numero medio di bundle memorizzati su un singolo nodo durante una simulazione, al variare della probabilità di incontro tra due nodi, impostando un *lifetime* di bundle pari a 5 e 10 minuti rispettivamente. Dal grafico in figura B.7(a) si può notare che, per una probabilità di contatto pari al 10%, il numero di bundle mediamente presenti su un nodo, dopo una fase iniziale di transitorio, oscilla tra i 100 e i 250 circa. Aumentando la probabilità di contatto,



(a) (b)

Figura B.7: Numero medio di bundle memorizzati su un nodo durante il tempo di simulazione, per diverse probabilità di incontro tra i dispositivi, con un *lifetime* di bundle pari a 5 minuti (a) e 10 minuti (b)

come già detto in precedenza, cresce significativamente il numero di copie di bundle in giro per la rete e di conseguenza, il numero di repliche di bundle memorizzate sui nodi. Tuttavia, quando il funzionamento del sistema è a regime, il numero di bundle non cresce mai in modo indefinito, ma rimane limitato all'interno di un intervallo di valori. Tale andamento è un requisito fondamentale per poter dimensionare opportunamente le risorse di storage dei dispositivi che compongono il sistema. La figura B.7(b) mostra come, aumentando il *lifetime* di bundle da 5 a 10 minuti, il numero di bundle mediamente presenti su un nodo cresce significativamente. In questo caso, infatti, per una probabilità di incontro pari al 10%, il numero di bundle mediamente memorizzati su un nodo va da 100 a ben oltre 300. Per di più, rispetto al caso precedente, si può notare come il sistema necessiti di più tempo per stabilizzarsi ed assumere valori compresi all'interno di un intervallo definito. Tale comportamento è giustificato dal fatto che, nonostante un bundle giunga a destinazione, i nodi della rete continueranno a mantenere una replica di esso fino alla scadenza del suo *lifetime*.

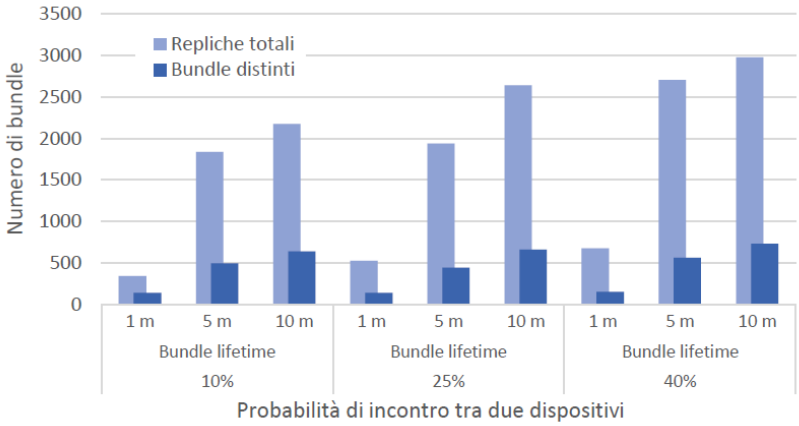


Figura B.8: Overhead dovuto allo storage

La figura B.8 si propone di valutare l'overhead medio dovuto allo storage dei bundle, al variare della probabilità di incontro tra i dispositivi (10%, 25%, 40%) e del *lifetime* di bundle (1, 5, 10 minuti). La figura mostra la differenza tra il

numero medio di bundle distinti rispetto al numero totale di repliche di bundle mediamente presenti all'interno della rete. E' evidente come il numero di bundle (sia distinti, che le repliche totali) aumenti in maniera significativa al crescere del *lifetime* e della probabilità di incontro, come conseguenza dell'utilizzo dell'algoritmo di routing epidemico.

Poiché, mediamente, soltanto il 55% dei bundle sono consegnati nell'arco di un minuto (come mostrato in figura B.6(a)), un *lifetime* molto piccolo rischia di generare un numero eccessivo di cancellazioni di bundle non ancora consegnati. Tuttavia, poiché circa il 95% dei bundle sono consegnati entro 3 minuti dalla loro generazione, l'utilizzo di un *lifetime* pari a 5 minuti risulta piuttosto ragionevole in uno scenario caratterizzato da un numero significativo di dispositivi. E' altresì importante sottolineare come la taratura del *lifetime*, così come quella di altri parametri, è fortemente dipendente dal contesto applicativo.

Bibliografia

[1] K. Scott e S. Burleigh. *Bundle Protocol Specification*. RFC 5050. IETF, 2007. URL: https://tools.ietf.org/html/rfc5050.

[2] V. Cerf et al. *Delay-Tolerant Networking Architecture*. RFC 4838. http://www.rfc-editor.org/rfc/rfc4838.txt. RFC Editor, 2007. URL: http://www.rfc-editor.org/rfc/rfc4838.txt.

[3] Luigi Maio. «Scenari applicativi nell'ambito del paradigma Fog Computing in presenza di reti partizionate e non affidabili». Tesi di Laurea Magistrale. Politecnico di Torino, 2017.

[4] K. Fall e S. Farrell. «DTN: An Architectural Retrospective». In: *IEEE J.Sel. A. Commun.* 26.5 (giu. 2008), pp. 828–836. ISSN: 0733-8716. DOI: 10.1109/JSAC.2008.080609. URL: https://doi.org/10.1109/JSAC.2008.080609.

[5] Forrest Warthman. *Delay and Disruption Tolerant Networks (DTNs): A Tutorial*. Lug. 2012. URL: http://ipnsig.org/wp-content/uploads/2012/07/DTN_Tutorial_v2.04.pdf.

[6] Johannes Morgenroth. «Event-driven Software-Architecture for Delay- and Distruption-Tolerant Networking». Tesi di Dottorato di Ricerca. Università Tecnica di Braunschweig, 2015. URL: http://www.digibib.tu-bs.de/?docid=00061364.

[7] W. Eddy e E. Davies. *Using Self-Delimiting Numeric Values in Protocols*. RFC 6256. IETF, mag. 2011. URL: https://tools.ietf.org/html/rfc6256.

[8] K. Scarfone J. Padgette L. Chen. «Guide to Bluetooth Security». In: (2017). URL: https://doi.org/10.6028/NIST.SP.800-121r2.

[9] *Bluetooth Core Specification v 5.0*. Rapp. tecn. 2016. URL: https://www.bluetooth.com/specifications/bluetooth-core-specification.

[10] *Bluetooth Low Energy Scanning and Adveritising*. URL: http://dev.ti.com/tirex/content/simplelink_academy_cc2640r2sdk_1_12_01_16/modules/ble_scan_adv_basic/ble_scan_adv_basic.html.

[11] Larry Rudolph Albert S. Huang. *Bluetooth Essentials for Programmers*. Cambridge University Press, 2007.

[12] *Bluetooth Stack*. URL: https://www.tutorialspoint.com/wireless_security/wireless_security_bluetooth_stack.htm.

[13] W.B. Pöttner S. Schildt J. Morgenroth e L. Wolf. «IBR-DTN: A lightweight, modular and highly portable Bundle Protocol implementation». In: *Electronic Communications of the EASST, Volume 37: Kommunikation in Verteilten Systemen 2011* (2011). DOI: http://dx.doi.org/10.14279/tuj.eceasst.37.512.544.

[14] M. Demmer, J. Ott e S. Perreault. *Delay-Tolerant Networking TCP Convergence-Layer Protocol*. RFC 7242. RFC Editor, 2014.

[15] Hans Kruse e Shawn Ostermann. *UDP Convergence Layers for the DTN Bundle and LTP Protocols*. Internet-Draft draft-irtf-dtnrg-udp-clayer-00. IETF Secretariat, 2008. URL: http://www.ietf.org/internet-drafts/draft-irtf-dtnrg-udp-clayer-00.txt.

[16] D. Ellard e D. Brown. *DTN IP Neighbor Discovery (IPND)*. Internet-Draft draft-irtf-dtnrg-ipnd-01. IETF, 2010. URL: https://tools.ietf.org/html/draft-irtf-dtnrg-ipnd-01.

[17] Johannes Morgenroth. *IBR-DTN API*. URL: https://github.com/ibrdtn/ibrdtn/wiki/API.

[18] *IBR-DTN public Github repository*. URL: https://github.com/ibrdtn.

[19] Johannes Morgenroth. *IBR-DTN configuration file example*. URL: https://github.com/ibrdtn/ibrdtn/blob/master/ibrdtn/daemon/etc/ibrdtnd.conf.

[20] *Fog Over DTN public Github repository*. URL: https://github.com/netgroup-polito/fog-over-dtn.

[21] *Docker Website*. URL: https://www.docker.com/.

[22] *Universal Node public Github repository*. URL: https://github.com/netgroup-polito/un-orchestrator.

[23] *ELK Stack*. URL: https://www.elastic.co/elk-stack.