POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Prototipazione di un sistema di comunicazione orientato ai servizi su reti tolleranti ai ritardi

Politecnico di Torino

Relatore
prof. Fulvio Risso

Candidato
Riccardo Mengoli

Supervisore aziendale
Tierra Telematics Design
dott. Calogero Carrabbotta, dott. Riccardo Loti

Dicembre 2022

Sommario

La seguente tesi tratta varie modifiche al codice sorgente di un software di gestione di reti delay-tolerant, con l'obiettivo finale di ottenere un prodotto pronto per il rilascio. Nella prima parte vengono presentati i miglioramenti applicati alla gestione di connessioni via Bluetooth, con il supporto a Bluetooth 5.0 e la rimozione delle dipendenze da programmi esterni. La seconda parte del documento descrive le modifiche al processo di build ed illustra il refactoring del codice sorgente. Il progetto viene aggiornato allo standard C++17 e vengono introdotti design pattern per una migliore organizzazione del codice. Conclude una descrizione sull'uso di strumenti di testing, code coverage e static analysis, al fine di gestire le fasi finali dello sviluppo software.

Indice

Capitolo 1

Introduzione

La sempre crescente diffusione di dispositivi IoT, in concomitanza con la rapida evoluzione delle tecnologie di comunicazione wireless, introducono una serie di nuove sfide dal punto di vista del *networking*. È 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 garantito nel contesto di una *challenged network*, che è invece caratterizzata da continue interruzioni o addirittura assenza di connettività, perciò non è mai assicurato un percorso stabile tra sorgente e destinazione. 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 il partizionamento della rete, ritardi lunghi e/o variabili ed alto tasso di perdite, i quali rendono la comunicazione ancora più complessa.

Gli scenari soggetti a tali problematiche sono molteplici e spaziano dalle reti di sensori in aree non dotate di alcuna infrastruttura di telecomunicazione, quali zone montuose, rurali o sotterranee, fino alle reti in ambito militare ed interplanetario. Le comunicazioni militari sono tipicamente situate in zone dove la connessione non è in alcun modo garantita per ovvi motivi, ma al contempo è richiesta la massima affidabilità della rete, evitando a tutti i costi il partizionamento, e la minima latenza di propagazione delle informazioni possibile. Le comunicazioni interplanetarie 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, in grado di 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 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 può essere inoltrato a qualche altro nodo intermedio ritenuto un potenziale next-hop verso la destinazione.

1.1 Obiettivo della tesi

Lo sviluppo di questa tesi è orientato alla gestione delle parti finali dello sviluppo del software *Æther*, con lo scopo di ottenere un prodotto pronto per il rilascio e per l'uso effettivo nel mondo reale. Æther è un software per la gestione di reti Delay-Tolerant, sviluppato da Tierra Telematics, che consente a diversi nodi di entrare a far parte della stessa DTN e di comunicare tra loro attraverso tecnologie di trasmissione eterogenee ed in maniera del tutto trasparente. Nella prima parte della tesi si è posto come obiettivo il miglioramento del supporto di connessioni Bluetooth, in quanto erano presenti vari problemi di compatibilità e poca facilità d'uso. La seconda parte, invece, riguarda il refactoring del progetto, per ottenere codice più pulito, leggibile, modulare e aggiornato con costrutti introdotti nello standard C++17. Il processo di build è stato rivisto e modernizzato, tagliando parti non necessarie e sostituendo varie istruzioni con equivalenti più leggibili. Per concludere, è stata inserita una parte di testing e produzione di report di code coverage, in modo da poter verificare che il software svolga il proprio compito in maniera corretta.

Capitolo 2

Delay-Tolerant Networks

Il seguente capitolo introduce il concetto di *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 che definisce il modo con cui i nodi della rete possono comunicare.

2.1 Delay-Tolerant Networking

Le *Delay-Tolerant Network* (DTN) 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 piccoli buffer di memoria per accodare i pacchetti in attesa di essere inoltrati, garantendone una persistenza nell'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.

2.1.1 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 istanti 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. È il caso di persone, veicoli, aerei o satelliti, che potrebbero voler scambiare informazioni qualora risultassero abbastanza vicini da poter comunicare utilizzando i propri mezzi di trasmissione.

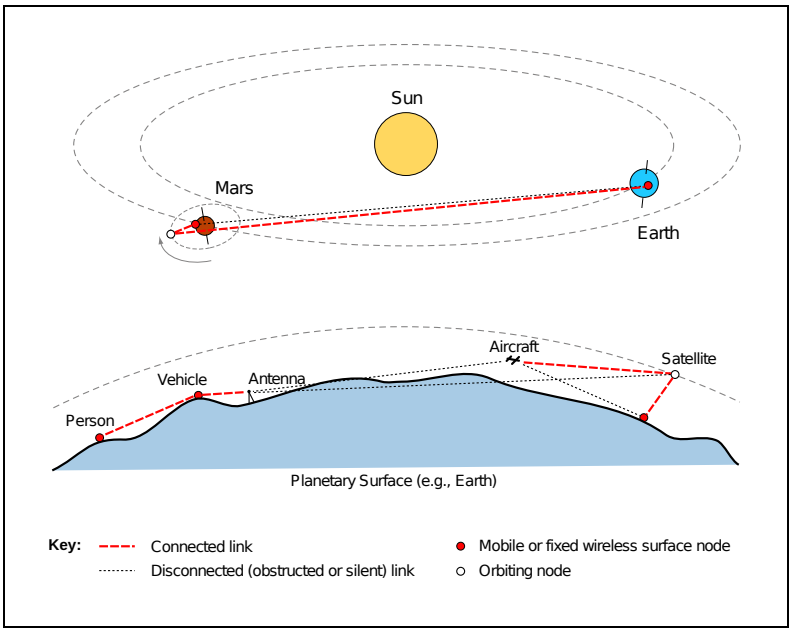


Figura 2.1. Esempi di contatti pianificati (comunicazioni interplanetarie) e opportunistici (comunicazioni sulla superficie terrestre) [1]

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).

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.

2.1.2 Overlay network

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 (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 dovranno 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.

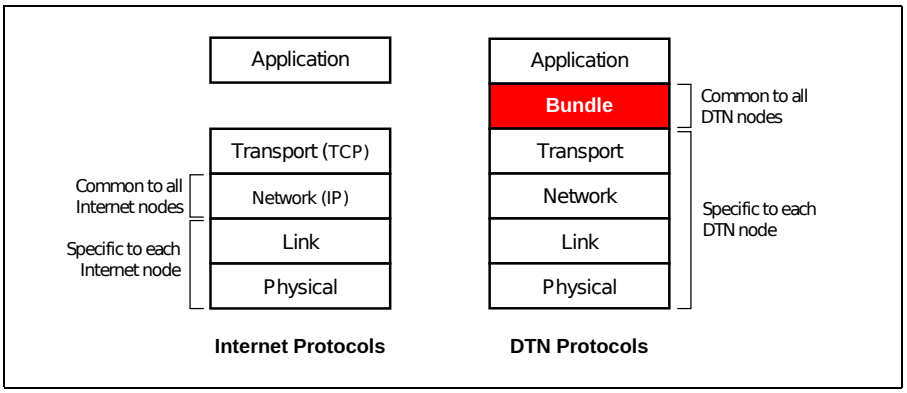


Figura 2.2. Confronto fra lo stack Internet (a sinistra) e lo stack DTN (a destra) [1]

2.2 Bundle Protocol

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 [2]. 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 chiamati *bundle*, mentre i *bundle node* o semplicemente nodi, sono le entità capaci di ricevere e trasmettere tali messaggi.

2.2.1 Architettura

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, 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* 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 deve 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.

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.

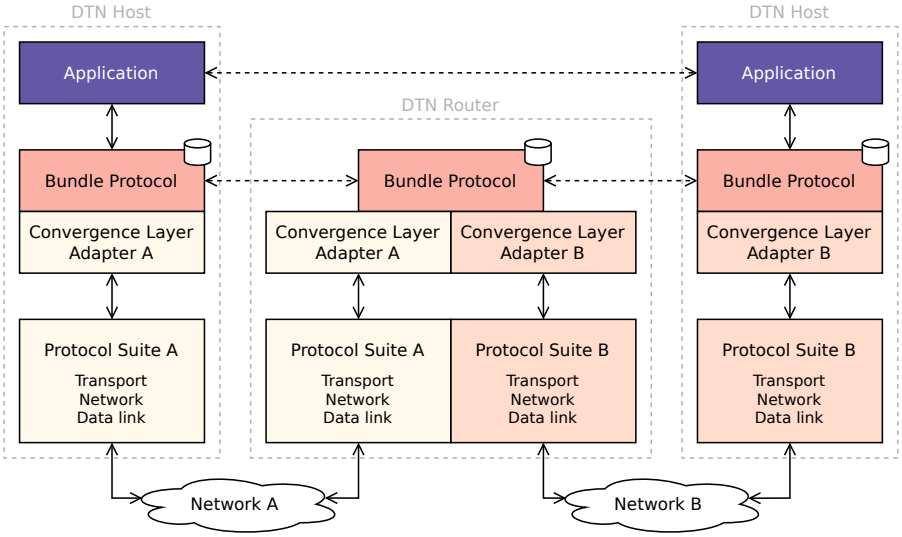


Figura 2.3. DTN Bundle Protocol su una rete eterogenea

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.

Nel caso di bundle troppo grandi, il bundle layer dovrebbe essere in grado di suddividere i messaggi in più frammenti, in maniera abbastanza simile a come il livello IP frammenta i propri 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

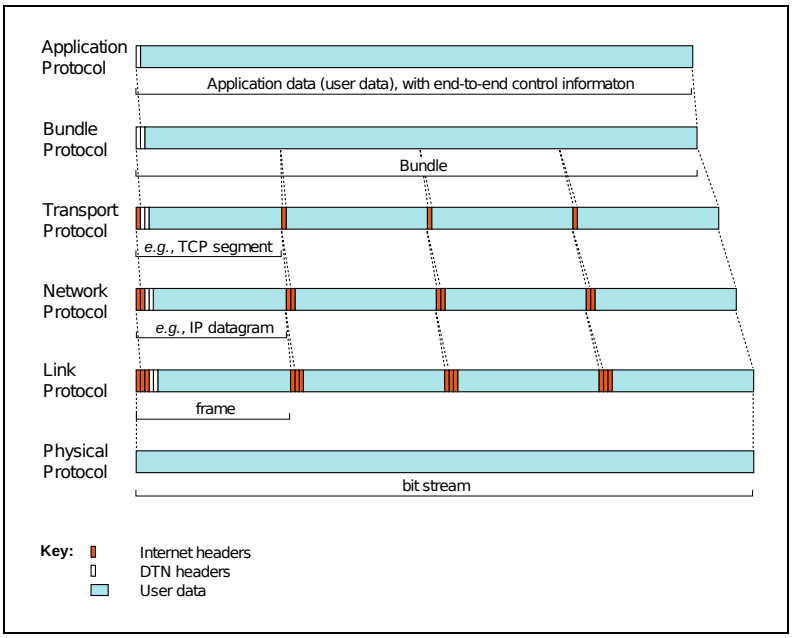


Figura 2.4. Incapsulamento dei protocolli TCP/IP nel Bundle Protocol [1]

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*. Il *binding* consiste nella procedura di interpretazione dello SSP di un EID, allo scopo di trasferire a destinazione il messaggio ad esso associato. Nel contesto delle reti basate su IP, si potrebbe pensare al concetto di binding come la risoluzione di un 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 sola 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 sia 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 chiamati *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) [3]. La maggior parte dei campi di un bundle hanno lunghezza variabile e utilizzano una notazione compatta detta *Self-Delimiting Numerical Values* (SDNVs) [4].

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 Protocol, che tipicamente rappresenta la versione 6 definita nell'RFC 5050.

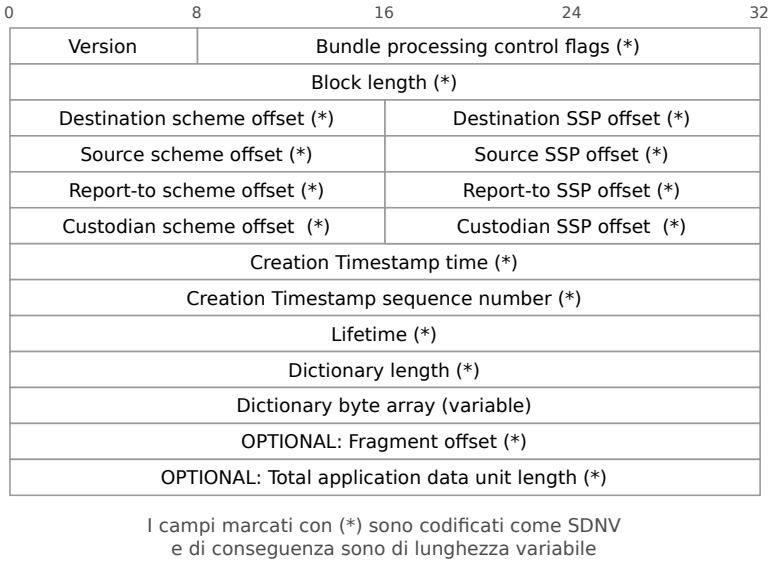


Figura 2.5. Formato del *primary block* di un bundle

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 acknowledgement 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 dei due campi 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. La combinazione di *Creation Timestamp* ed *EID source* identifica 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 esso viene scartato. 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 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 di un potenziale blocco che segue 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 se si tratta dell'ultimo blocco presente all'interno di un bundle, tali flag specificano, ad

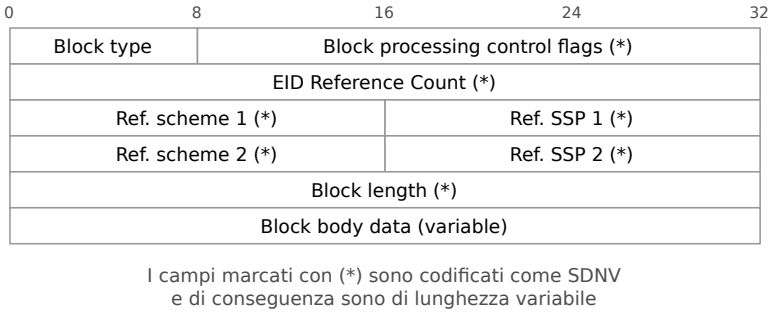


Figura 2.6. Bundle block con due referenze a EID

esempio, se il blocco deve 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. I dati saranno specifici 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, 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. È 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 siano 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 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 o per ragioni implementative. 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.

2.2.7 Bundle Protocol versione 7

La versione 6 del Bundle Protocol, descritta in questo capitolo, viene diffusa nel novembre 2007 all'interno della RFC 5050 [2]. Dalla sua pubblicazione, il Bundle Protocol è stato implementato in diversi linguaggi di programmazione e distribuito su un'ampia varietà di piattaforme informatiche. L'esperienza ricavata dall'uso del Bundle Protocol ha permesso di identificare le opportunità per rendere il protocollo più semplice, più efficace e più facile da usare. Nel Gennaio 2022 nasce così il Bundle Protocol versione 7, pubblicato nell'RFC 9171 [5]. BPv7 introduce cambiamenti significativi rispetto alla versione 6, senza però rivoluzionare il funzionamento del protocollo. Tra le novità possiamo elencare:

- Cambiamento dell'encoding dei valori da SDNV a CBOR;
- Ristrutturazione del *primary block*, rendendolo immutabile;
- Definizione di vari *extension block*.

2.3 Routing

Nel contesto del Delay-Tolerant Networking, con il concetto di routing si intende la capacità 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

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, esisterà 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.

Epidemic routing l'*Epidemic routing* 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 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.

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;
2. Le probabilità per tutte le destinazioni D vengono invecchiate;
3. M ed E si scambiano le *delivery predictability* e la proprietà transitiva viene utilizzata per aggiornare le probabilità delle destinazioni D.

Capitolo 3

Æther

Nel seguente capitolo viene descritta l'architettura di Æther, il software per la gestione di reti delay-tolerant su cui si è concentrato lo sviluppo di questa tesi. Æther, progetto di Tierra Telematics, nasce come estensione di IBR-DTN, software open-source sviluppato nel 2008 all'Università Tecnica di Braunschweig [6, 7]. Il contenuto del capitolo fa riferimento a [8], oltre che ad un'ispezione del codice sorgente di Æther.

IBR-DTN è un'implementazione open-source leggera, efficiente e modulare del Bundle Protocol versione 6, 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.

IBR-DTN rappresenta una delle implementazioni più popolari del Bundle Protocol, insieme a DTN2, ION e µD3TN. Il progetto Æther si prefigge come obiettivo quello di ampliare le funzionalità di IBR-DTN, introducendo nuovi Convergence Layer, nuove interfacce e nuovi algoritmi di routing. A seguire verranno presentati i componenti principali di IBR-DTN, che, non avendo subito variazioni, risultano identici nel progetto Æther.

3.1 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 3.1).

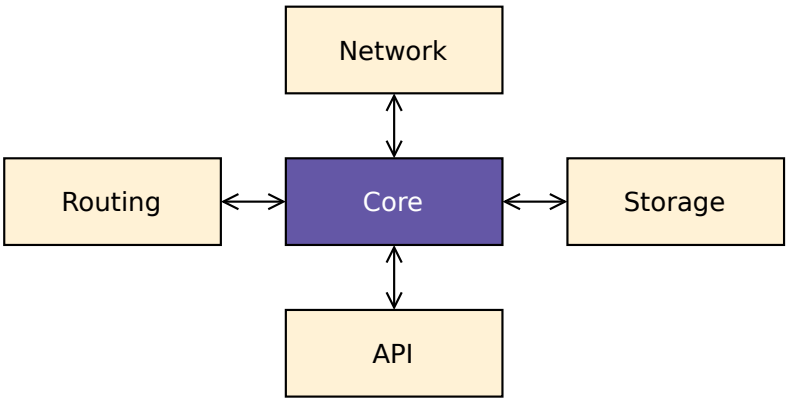


Figura 3.1. Visione di alto livello dell'architettura IBR-DTN

3.1.1 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 può avvenire o tramite chiamate sincrone o 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.

3.1.2 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 consente la memorizzazione di bundle in memoria RAM, su disco (filesystem) e su database 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 corrispondono ad un certo criterio.

3.1.3 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
- TLS extension for TCP convergence layer
- UDP/IP convergence layer
- 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

3.1.4 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.

3.1.5 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.

La figura 3.2 riassume quanto detto finora e illustra, in maniera dettagliata, i moduli principali che compongono l'architettura IBR-DTN e le interazioni tra essi.

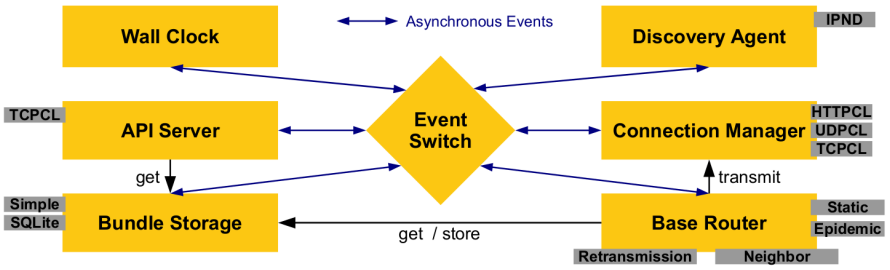


Figura 3.2. Componenti fondamentali dell'architettura IBR-DTN [6]

3.2 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 3.3.

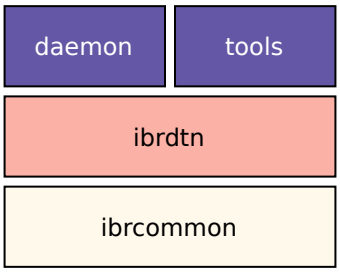


Figura 3.3. Divisione in packages di IBR-DTN

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.

3.3 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.

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`.

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.

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.

BundleID Come indicato all'interno dell'RFC 5050 [2], 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.

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.

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

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

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.

3.4 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`. L'architettura IBR-DTN include diversi tipi di eventi, ognuno dei quali è scatenato al verificarsi di una certa condizione e coinvolge componenti differenti.

3.5 Concorrenza

Ognuno dei componenti dell'architettura IBR-DTN 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*, che garantiscono 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.

3.6 Aggiornamenti in Æther

Æther introduce alcune funzionalità al software IBR-DTN, in particolare:

- Bluetooth Convergence Layer [9];
- V2X Convergence Layer [10];
- Service discovery e Virtual Service proxy [11];

- Algoritmo di routing MaxProp [12].

Per approfondire è possibile consultare i riferimenti specificati, che descrivono le implementazioni dei rispettivi moduli in Æther.

Come già preannunciato nell'introduzione, questa tesi punta a migliorare la gestione del Bluetooth Convergence Layer di Æther. Nel capitolo 4 verrà presentata la tecnologia Bluetooth, comprensiva delle novità di Bluetooth 5.0, mentre nel capitolo 5 saranno descritte le modifiche apportate al codice sorgente. Un secondo obiettivo, ma non meno importante, è il code refactoring e l'aggiornamento del processo di build, con lo scopo di raggiungere gli standard di qualità richiesti da un prodotto finale. Questi cambiamenti saranno presentati nel capitoli 6 e 7.

Capitolo 4

Bluetooth

In questo capitolo sono descritte le caratteristiche della tecnologia Bluetooth, partendo dalle origini con Bluetooth BR/EDR fino ad arrivare alle nuove versioni con Bluetooth Low Energy (BLE) e Bluetooth 5.0.

4.1 Bluetooth

La tecnologia wireless Bluetooth è un sistema di comunicazione a corto raggio, nata con l'obiettivo di sostituire i cavi per il collegamento verso dispositivi portatili o fissi. Le caratteristiche principali di Bluetooth sono la robustezza della comunicazione, il basso costo e il basso consumo energetico. Molte delle funzionalità previste dallo standard Bluetooth sono opzionali e ciò permette di differenziare i prodotti e adattarli al meglio al caso d'uso. Esistono due tecnologie wireless Bluetooth: *Basic Rate* (BR) e *Low Energy* (LE). Entrambi i sistemi includono procedure di device discovery e connection establishment, con lo stesso scopo ma differenti nell'implementazione. Il sistema BR prevede un'estensione opzionale chiamata *Enhanced Data Rate* (EDR), che, come evidenziato dal nome, punta ad aumentare la velocità di trasmissione dei dati. Mentre il primo si ferma ad una velocità massima teorica di 721,2 kbps, il secondo può raggiungere 2,1 Mbps.

4.1.1 Compatibilità

I dispositivi possono adottare una sola delle tecnologie oppure entrambe, in modo tale da supportare la connessione verso quanti più sistemi possibili. I due sistemi di trasmissione non sono compatibili tra di loro, un dispositivo Bluetooth BR/EDR non può comunicare in nessun modo con un dispositivo Bluetooth LE. In figura 4.1 sono mostrati tre diversi tipi di dispositivi:

BR/EDR Supporta solamente la trasmissione wireless con le tecnologie rientranti nella definizione di Bluetooth Classic, ovvero BR ed EDR.

BR/EDR/LE Supporta entrambi i sistemi di comunicazione Bluetooth e può quindi comunicare sia con BR/EDR che con BLE.

BLE Supporta esclusivamente trasmissioni via Bluetooth Low Energy.

La figura 4.1 inoltre illustra i diversi stack protocollari, che verranno approfonditi in seguito.

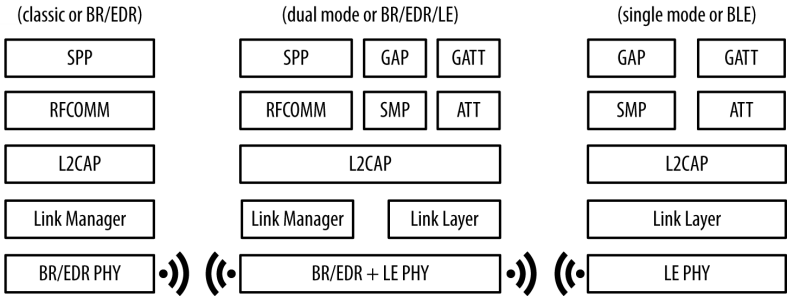


Figura 4.1. Differenti configurazioni di dispositivi che supportano diverse tecnologie Bluetooth [13]

4.1.2 Indirizzamento

Per fare in modo che un dispositivo Bluetooth possa stabilire una comunicazione con un altro dispositivo, è necessario stabilire un modo per identificarlo. Ogni chip Bluetooth prodotto prevede un indirizzo a 48 bit globalmente univoco. Questo indirizzo è equivalente all'indirizzo MAC di Ethernet, tanto che entrambi gli spazi di indirizzi sono gestiti dalla stessa organizzazione, la IEEE. Gli indirizzi vengono assegnati al momento della produzione del dispositivo e sono destinati a rimanere statici per tutta la vita del chip. A differenza dello stack TCP/IP, l'indirizzo Bluetooth è l'unica modalità di indirizzamento di un dispositivo e viene utilizzato in tutti i livelli dello stack protocollare, dai protocolli radio di basso livello ai protocolli applicativi. Gli indirizzi Bluetooth sono frequentemente scritti come 6 coppie di caratteri esadecimali, separati dal simbolo ":", per esempio `2C:41:A1:27:6D:90`. Per una maggiore facilità di utilizzo, solitamente i dispositivi Bluetooth hanno associato un nome user-friendly che viene tipicamente visualizzato al posto dell'indirizzo del dispositivo, anche se il nome non è necessariamente univoco a livello globale. L'utente può così personalizzare il proprio dispositivo per renderlo facilmente identificabile.

4.1.3 Sicurezza

In qualsiasi sistema informatico la sicurezza è importante per garantire le proprietà di riservatezza, integrità e disponibilità delle informazioni. Varie misure di sicurezza sono presenti nello standard Bluetooth sin dalle prime versioni, tuttavia l'uso di algoritmi deboli e meccanismi poco sicuri ha delineato Bluetooth come ambiente non affidabile. Alcuni semplici attacchi alla rete Bluetooth dei primi anni 2000 sono stati denominati bluebugging, bluejacking e bluesnarfing. Il gruppo incaricato allo sviluppo dello standard Bluetooth, cosciente del problema, ha lavorato per aggiornare le misure di sicurezza con nuove procedure e con l'uso di algoritmi approvati da FIPS (standard del governo americano), rendendo Bluetooth moderno un sistema sicuro. Alcuni dettagli sulla sicurezza di Bluetooth BR/EDR verranno presentati nella sezione 4.2.4.

4.1.4 Organizzazione del dispositivo

Ogni dispositivo Bluetooth è costituito da tre blocchi fondamentali:

Application L'applicazione utente che si interfaccia con lo stack Bluetooth.

Host I livelli superiori dello stack protocollare Bluetooth, paragonabili a un sistema operativo.

Controller I livelli inferiori dello stack protocollare Bluetooth, spesso implementati come system on a chip.

La specifica fornisce un protocollo di comunicazione standard tra l'host ed il controller, la *Host Controller Interface* (HCI), per consentire l'interoperabilità tra host e controller di diversi produttori. Host e controller devono comportarsi come container logici indipendenti interagendo attraverso l'interfaccia HCI, ma possono essere implementati fisicamente nello stesso componente. In figura 4.2 vengono mostrate tre diverse configurazioni hardware: la prima viene solitamente utilizzata su dispositivi semplici come sensori, la seconda è la scelta più comune per dispositivi con CPU potenti a disposizione, come smartphone e computer. La terza configurazione è più rara e viene usata in scenari quali l'aggiunta di connettività Bluetooth a un microcontrollore personalizzato.

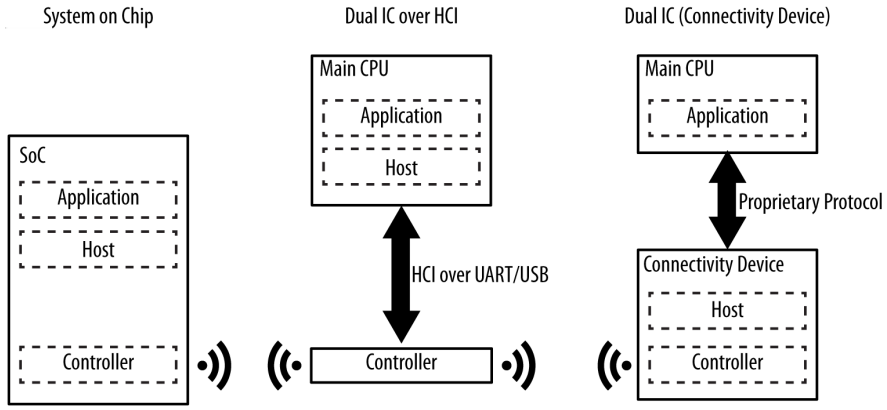


Figura 4.2. Configurazioni hardware dispositivi Bluetooth [13]

4.2 Bluetooth Classic

In questa sezione vengono descritte le principali caratteristiche di Bluetooth BR/EDR, anche chiamato Bluetooth Classic.

4.2.1 Protocol stack

La figura 4.3 mostra alcuni dei moduli che compongono lo stack protocollare di Bluetooth Classic. Per essere conforme alla specifica Bluetooth BR/EDR un dispositivo deve necessariamente soddisfare un insieme di requisiti definiti nei livelli Radio, Baseband e LMP per

quanto riguarda la parte Controller e L2CAP, SDP e GAP per quanto riguarda la parte Host. Nei paragrafi seguenti verranno descritti i livelli utilizzati nel seguente lavoro di tesi, mentre per i restanti è possibile consultare la *Bluetooth Core Specification* [14].

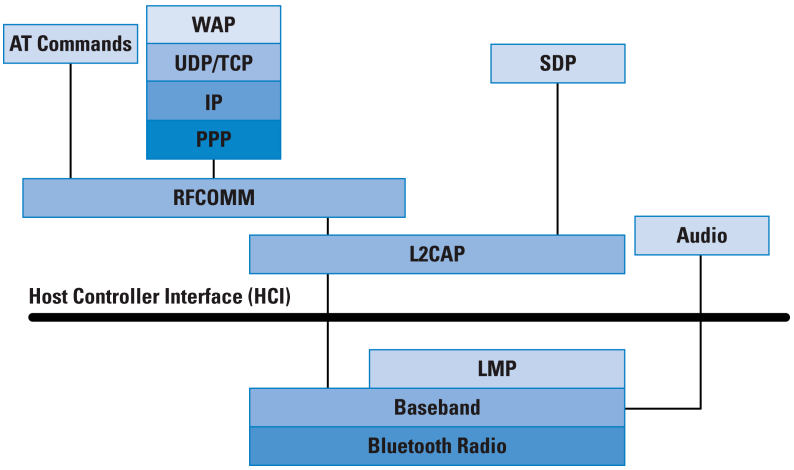


Figura 4.3. Stack protocollare Bluetooth Classic [15]

4.2.2 Radio

Il livello più basso dello stack Bluetooth è rappresentato dalla radio, ovvero tutto l'insieme di specifiche per la trasmissione in radiofrequenza. Bluetooth opera nella banda senza licenza ISM (Industrial Scientific Medical) a 2.4 GHz ed utilizza il meccanismo di *adaptive frequency hopping*, che consiste nel cambio di frequenza ad intervalli regolari al fine di ridurre le interferenze con altri dispositivi che trasmettono nella stesso spettro. Bluetooth Classic suddivide la banda a disposizione in 79 canali radio, ciascuno di dimensione 1 MHz.

4.2.3 Baseband

Il livello Baseband implementa l'accesso al mezzo fisico e definisce numerosi concetti e procedure, tra cui:

- struttura logica dei canali di trasmissione (sincroni, asincroni, unicast, broadcast, connection-oriented, connectionless);
- struttura dei pacchetti;
- procedura di connessione tra dispositivi;
- procedura di *Inquiry*;
- procedura di *Paging*;
- operazioni di error checking ed error correction.

Con Bluetooth Classic è possibile stabilire connessioni point-to-point oppure point-to-multipoint, formando reti definite *piconet*. In una *piconet*, un unico dispositivo Bluetooth ha il ruolo di *Central* ed i restanti avranno ruolo *Peripheral*. Nello stabilire una connessione il dispositivo che invia la richiesta ricoprirà il ruolo di Central mentre la controparte ricoprirà il ruolo di Peripheral, con la possibilità di invertire i ruoli in un secondo momento.

Connessione e device discovery Per stabilire una connessione Bluetooth BR/EDR i dispositivi Central e Peripheral devono trovarsi rispettivamente nello stato di Page e Page Scan. La procedura di connessione viene chiamata Paging e richiede come input solamente l'indirizzo del dispositivo a cui connettersi. Il Central trasmetterà la propria richiesta di connessione su varie frequenze in vari momenti, fino a quando non riceverà una risposta dalla destinazione. La latenza di connessione è determinata da un processo probabilistico, perché la Peripheral può essere in ascolto su uno qualsiasi dei 32 canali a disposizione e solo per un certo periodo di tempo, cambiando periodicamente canale. Per velocizzare i tempi di connessione è possibile sfruttare la procedura di Inquiry, uno scambio di messaggi tra Central e Peripheral che permette al Central di ricavare informazioni di timing della Peripheral. Con tali informazioni il Central è in grado di prevedere la frequenza su cui la Peripheral sarà in ascolto e di conseguenza la richiesta di connessione giungerà a destinazione nel minor tempo possibile. Anche per la procedura di Inquiry sono previsti due stati, Inquiry e Inquiry Scan. I dispositivi in stato di Inquiry trasmettono i messaggi di Inquiry su varie frequenze e i dispositivi in Inquiry Scan rispondono con le proprie informazioni.

4.2.4 LMP

Il *Link Manager Protocol* (LMP) viene utilizzato per controllare e negoziare tutti gli aspetti del collegamento Bluetooth tra due dispositivi. Tra questi sono incluse cinque importanti funzionalità di sicurezza:

Pairing Il processo di creazione di una o più chiavi segrete condivise.

Bonding La memorizzazione delle chiavi create durante il pairing per utilizzarle nelle connessioni successive.

Device authentication La verifica che due dispositivi dispongano delle stesse chiavi.

Encryption Cifratura dei messaggi per ottenere la riservatezza della comunicazione.

Message integrity Protezione dalle manipolazioni dei messaggi.

Nel seguito verranno introdotti alcuni dettagli sulla sicurezza Bluetooth BR/EDR. Per approfondimenti è possibile consultare la *Bluetooth Core Specification* [14][Vol. 1 Part A Section 5 e Vol. 2 Part H] oppure *Guide to Bluetooth Security* [16].

Autenticazione LMP definisce due procedure di autenticazione, *Legacy Authentication* e *Secure Authentication*. Entrambe implementano lo schema di challenge-response basato su una chiave segreta condivisa chiamata *link key*. La link key è un valore di 128 bit alla base di tutta la sicurezza di Bluetooth Classic, utilizzata non solo per l'autenticazione ma anche come parametro per derivare la chiave di cifratura.

Pairing Quando due dispositivi non dispongono di una *link key* comune è necessario crearne una utilizzando la procedura di *Legacy Pairing* o *Secure Simple Pairing*.

Il Legacy Pairing prevedeva l'inserimento di un PIN alfanumerico di 16 caratteri su entrambi i dispositivi in connessione, PIN che veniva poi utilizzato per derivare la Link Key. Per facilitare l'interazione con l'utente, i PIN venivano spesso ridotti a 4 cifre o addirittura a PIN fissi, limitando in modo significativo la sicurezza del collegamento. Un attaccante in grado di ottenere il codice PIN sarebbe stato in grado di avere il controllo completo della connessione.

Secure Simple Pairing (SSP) viene introdotto in Bluetooth 2.1 e si prefigge come obiettivo quello di superare il livello di sicurezza fornito dal Legacy Pairing e di semplificare la procedura di pairing per l'utente, garantendo protezione da intercettazioni e attacchi man-in-the-middle (MITM). SSP definisce quattro possibili modalità di associazione:

Numeric Comparison In questa procedura i dispositivi in connessione calcolano e mostrano a video un numero di sei cifre. Se l'utente conferma che essi coincidono su entrambi i dispositivi allora significa che la connessione è autenticata ed è possibile procedere con la generazione della link key.

Passkey Entry In questa procedura almeno uno dei due dispositivi dispone di funzionalità di input e l'altro dispone di output. Il dispositivo con output genera e mostra un numero di sei cifre che l'utente dovrà inserire nel secondo dispositivo. Se i due valori combaciano allora il pairing ha avuto successo.

Just Works Nel caso in cui almeno uno dei due dispositivi non disponga di alcuna funzionalità di input o output è stato previsto il modello di associazione *Just Works*, che a livello LMP utilizza la procedura di Numeric Comparison senza però mostrare a video il numero generato. In questa modalità non è possibile garantire la protezione da attacchi MITM.

Out Of Band (OOB) In questa modalità viene utilizzato un canale alternativo a Bluetooth per trasmettere i valori crittografici richiesti per il processo di pairing. Un esempio può essere una soluzione con Near Field Communication (NFC).

Se durante lo scambio delle capacità di I/O sono presenti dati OOB da parte di entrambi i dispositivi allora LMP utilizzerà la modalità OOB. In caso contrario, LMP sceglie automaticamente la procedura da utilizzare in base alle proprietà di sicurezza richieste ed alle funzionalità di I/O presenti nel dispositivo.

Secure Simple Pairing è stato aggiornato in Bluetooth 4.1 con l'estensione *Secure Connections*, che prevede l'uso di algoritmi più forti nelle procedure di encryption, authentication e key generation. In particolare l'algoritmo di cifratura E0 è stato sostituito con AES-CCM, che a differenza di E0 permette di garantire l'integrità dei messaggi.

4.2.5 HCI

La *Host Controller Interface* (HCI) definisce un'interfaccia standardizzata attraverso la quale host e controller possono comunicare tra di loro. L'interfaccia HCI è utilizzata sia da Bluetooth BR/EDR che da Bluetooth LE. La specifica definisce i messaggi scambiati con

il nome di *commands* e *events*, i comandi sono inviati dall'host verso il controller mentre gli eventi sono inviati dal controller verso l'host. Un evento può rappresentare una risposta ad un comando, ma può anche essere inviato come messaggio indipendente. L'interfaccia HCI può essere implementata su quattro possibili trasporti fisici: UART, USB, Secure Digital (SD) o Three-wire UART.

4.2.6 L2CAP

Il *Logical Link Control and Adaptation Protocol* (L2CAP) rappresenta il livello più basso della parte Host. Esso opera come protocol multiplexer, ricevendo dati da protocolli di livello superiore e incapsulandoli nel formato standard dei pacchetti Bluetooth (e viceversa). L2CAP gestisce anche la frammentazione e ricombinazione, quel processo che divide pacchetti di grandi dimensioni in pezzi che possano rientrare nella dimensione massima del singolo pacchetto di trasmissione. Per fare un paragone, L2CAP è simile a TCP, in quanto permette ad un'ampia gamma di protocolli di coesistere senza problemi su un singolo collegamento fisico, ciascuno con dimensioni e requisiti di pacchetto diversi. L2CAP è presente sia in Bluetooth Classic che in Bluetooth LE, con due implementazioni diverse per adattarsi al differente livello fisico sottostante.

4.2.7 GAP

Il *Generic Access Profile* (GAP) è un profilo che introduce definizioni, modalità e procedure relative a tutti i livelli Bluetooth obbligatori. GAP descrive sia i livelli Bluetooth Classic che quelli Bluetooth Low Energy.

Discoverability modes GAP richiede che un dispositivo Bluetooth BR/EDR sia in qualsiasi momento in modalità discoverable o in modalità non-discoverable. Quando è attiva la non-discoverable mode il dispositivo non entra nello stato di Inquiry Scan e di conseguenza non riceverà né risponderà ai messaggi di inquiry. Se invece il dispositivo viene reso discoverable, esso sarà in limited discoverable mode oppure in general discoverable mode. La prima modalità viene usata per dispositivi che devono essere rilevabili solo per un periodo limitato di tempo, la seconda invece è una modalità generica in cui un dispositivo risulta rilevabile senza condizioni specifiche.

Connectability modes Per quanto riguarda la connettività, un dispositivo Bluetooth BR/EDR può essere in modalità connectable o non-connectable. La modalità connectable significa che il dispositivo entra periodicamente nello stato di Page Scan. La variazione dei parametri di Page Scan permette di ottenere un compromesso tra consumo energetico e velocità di connessione. La non-connectable mode prevede invece che il dispositivo non entri mai nello stato di Page Scan e di conseguenza non potrà ricevere richieste di connessione.

4.2.8 RFCOMM

RFCOMM è un protocollo di trasporto che offre l'emulazione di porte seriali RS-232 attraverso il protocollo L2CAP. Introdotto nel 2001, RFCOMM è uno dei protocolli Bluetooth più datati, progettato per facilitare l'aggiunta di funzionalità Bluetooth a dispositivi con

porte seriali già esistenti. Esso fornisce all'incirca le stesse garanzie di servizio e affidabilità di TCP, permettendo di stabilire connessioni punto-punto su cui scambiare dati in modo affidabile. Al fine di comunicare con un dispositivo remoto sono necessari un protocollo, un indirizzo e un numero di porta. Nel protocollo RFCOMM le porte sono chiamate canali, tuttavia svolgono esattamente lo stesso ruolo delle più conosciute porte TCP. Le applicazioni server RFCOMM possono registrarsi sui canali da 1 a 30 (0 e 31 riservati), in attesa di ricevere richieste di connessione dai client.

4.3 Bluetooth Low Energy (BLE)

Bluetooth Low Energy, spesso indicato con la sigla BLE, è stato introdotto nella specifica Bluetooth 4.0. Bluetooth LE può apparire a primo impatto come una versione più piccola e maggiormente ottimizzata di Bluetooth BR/EDR, ma in realtà ha origini e obiettivi di progettazione completamente differenti.

4.3.1 Obiettivi

BLE viene originariamente progettato da Nokia con il nome di *Wibree* ed in seguito adtottato dal *Bluetooth Special Interest Group* (Bluetooth SIG). Fin dall'inizio, l'obiettivo era quello di progettare uno standard radio che ottenesse il minor consumo energetico e che fosse ottimizzato per un basso costo, bassa potenza e bassa complessità. Questi intenti sono evidenti nella specifica: BLE si vuole proporre come vero e proprio standard di trasmissione a basso consumo per essere utilizzato nel mondo reale con budget energetici e di silicio limitati. BLE entra nello standard Bluetooth nel dicembre 2009, con il rilascio della versione 4.0 della *Bluetooth Core Specification*, dopo vari anni di scrittura e decisioni su parti controverse. Il primo aggiornamento importante, Bluetooth 4.1, viene rilasciato nel dicembre 2013 ed introduce la standardizzazione di diverse pratiche comuni. Ad oggi, novembre 2022, lo standard Bluetooth è disponibile nella versione 5.3. Tutte le specifiche Bluetooth sono retrocompatibili con le precedenti, in modo da permettere la corretta interoperabilità tra dispositivi che implementano versioni diverse.

4.3.2 Differenze con altri standard wireless

BLE è uno standard giovane eppure ha visto un tasso di adozione estremamente rapido. Il successo è facile da comprendere, poiché legato alla crescita fenomenale di smartphone e tablet. BLE è stato adottato fin dalle origini dai maggiori produttori mondiali come Apple e Samsung e ciò l'ha reso immediatamente popolare. Possiamo descrivere BLE come la tecnologia giusta, con i compromessi giusti al momento giusto. La grande quantità di investimenti in ricerca e sviluppo ha permesso a BLE di diventare una tecnologia matura in pochi anni e ampliare i suoi casi d'uso a sempre più applicazioni. Tra tutte le tecnologie wireless, Bluetooth è quello che meglio soddisfa i requisiti di basso consumo e basso costo. È da evidenziare, tuttavia, che Bluetooth non cerca di essere la soluzione ad ogni necessità di trasferimento dati wireless. Altre tecnologie come WiFi, NFC, Zigbee e GSM continuano ad essere ampiamente utilizzate nei loro specifici contesti.

4.3.3 Limitazioni

Fino all'introduzione di Bluetooth 5.0, la velocità di modulazione della radio BLE era fissata dalla specifica al valore costante di 1 Mbps. Questo definisce il limite teorico del throughput raggiungibile, ma nei casi reali questo valore risulta molto spesso ridotto a causa di una varietà di fattori, tra cui traffico bidirezionale, overhead del protocollo o limitazioni della CPU o della radio. Alcune stime tratte da Townsend et al. [13] mostrano come in casi ottimali sia possibile raggiungere velocità di 125 kbps, traducibili in un throughput di poche decine di KB per secondo. Questi valori sono utili per capire cosa si può e cosa non si può fare con BLE, ed evidenziano chiaramente il perché Bluetooth Classic o WiFi vengano ancora ampiamente utilizzati. Nonostante questi valori possano sembrare ridicoli, essi sottolineano la principale scelta di design di Bluetooth Low Energy: usare poca energia.

Le nuove versioni di Bluetooth hanno portato miglioramenti per quanto riguarda il throughput e a partire da Bluetooth 5.0 è stato anche introdotto un nuovo livello fisico a 2 Mbps. Ciò nonostante le considerazioni ricavate in questa sezione rimangono valide, BLE non nasce per trasferire grandi quantità di dati ma per farlo in modo efficiente.

4.3.4 Protocol stack

La figura 4.4 mostra i livelli dello stack BLE e la loro distribuzione nella parte host e controller. Lo stack Bluetooth LE copre tutti i livelli del modello di riferimento OSI, diversamente da quanto fatto da altre tecnologie wireless che si limitano solo ad una parte di essi. Il vantaggio di questa scelta è che non ci sono dipendenze esterne da altri organismi di standardizzazione che potrebbero rallentare l'evoluzione della tecnologia.

4.3.5 Physical Layer

Il livello fisico definisce tutti gli aspetti della tecnologia Bluetooth relativi all'uso della radio, tra cui gli schemi di modulazione, le bande di frequenza e la divisione in canali. BLE opera nella banda a 2.4 GHz, la stessa di Bluetooth Classic, ma divide il proprio intervallo di frequenze in 40 canali a 2 MHz. È compito del Link Layer decidere come utilizzare al meglio questi canali. Lo schema di modulazione utilizzato si chiama *Gaussian Frequency Shift Keying* (GFSK) e viene utilizzato per codificare i dati digitali ricevuti dai livelli superiori in segnali radio e per decodificarli nel dispositivo ricevente. Il suo funzionamento è semplice: considerando il segnale di riferimento come la frequenza centrale del canale utilizzato, un bit che rappresenta il valore 1 avrà una frequenza leggermente superiore rispetto al segnale di riferimento e viceversa per il valore 0. Per quanto riguarda gli schemi di modulazione sono state definite tre varianti, indicate con il termine PHY:

LE 1M PHY Primo schema di BLE, symbol rate di 1 Msym/s (mega symbols per second) corrispondente a 1 Mbps di data rate, supporto obbligatorio.

LE 2M PHY Schema simile a LE 1M ma con un data rate massimo di 2 Mbps, introdotto in Bluetooth 5.0. Il supporto allo schema è opzionale.

LE Coded PHY Schema con symbol rate a 1 Msym/s nel quale i pacchetti sono soggetti a *Forward Error Correction* (FEC), che permette di aumentare la distanza di trasmissione a costo di una ridotto data rate. Il supporto allo schema è opzionale.

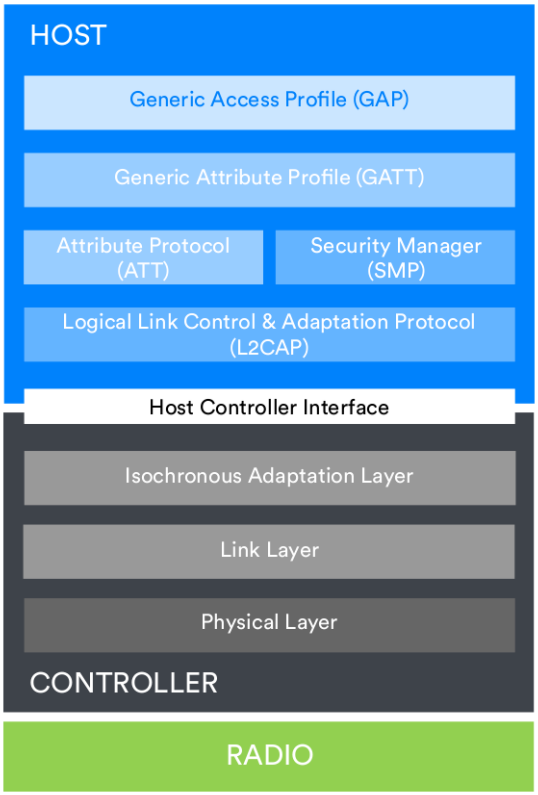


Figura 4.4. Lo stack Bluetooth Low Energy [17]

4.3.6 Link Layer

La specifica del Link Layer è quasi la più corposa delle sezioni Bluetooth LE della *Bluetooth Core Specification*, seconda solo alla sezione relativa alla Host Controller Interface. Questo perché il link layer Bluetooth LE ha molte responsabilità. Tra le tante, esso definisce i tipi di pacchetti, seleziona i canali da utilizzare, supporta comunicazioni connesse o connectionless, point-to-point o one-to-many, gestisce l'advertising e lo scanning e, se richiesto, si occupa di cifrare e decifrare i dati trasmessi.

In figura 4.5 viene rappresentata la struttura di un pacchetto utilizzato con schemi di modulazione non codificati. Il campo PDU può contenere diversi *Protocol Data Unit* (PDU), che contengono al loro interno i dati effettivi dell'applicazione utente.

Il Link Layer è controllato da una macchina a stati, rappresentata in figura 4.6. Nello stato *Standby* il dispositivo non trasmette né riceve pacchetti, *Advertising* trasmette pacchetti di advertising, *Scanning* rimane in ascolto di pacchetti di advertising, *Initiating* risponde ai pacchetti di advertising ricevuti per richiedere una connessione e *Connection* significa che il dispositivo è connesso. I dettagli approfonditi di ogni stato sono disponibili nella specifica Bluetooth [14]. Nello stato *Connection* sono previsti due importanti ruoli, *Central* e *Peripheral*. Un dispositivo che attiva una connessione passa dallo stato *Initiating* a

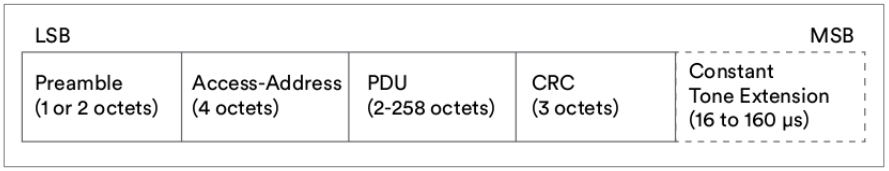


Figura 4.5. Formato dei pacchetti del livello link layer per LE uncoded PHYs [17]

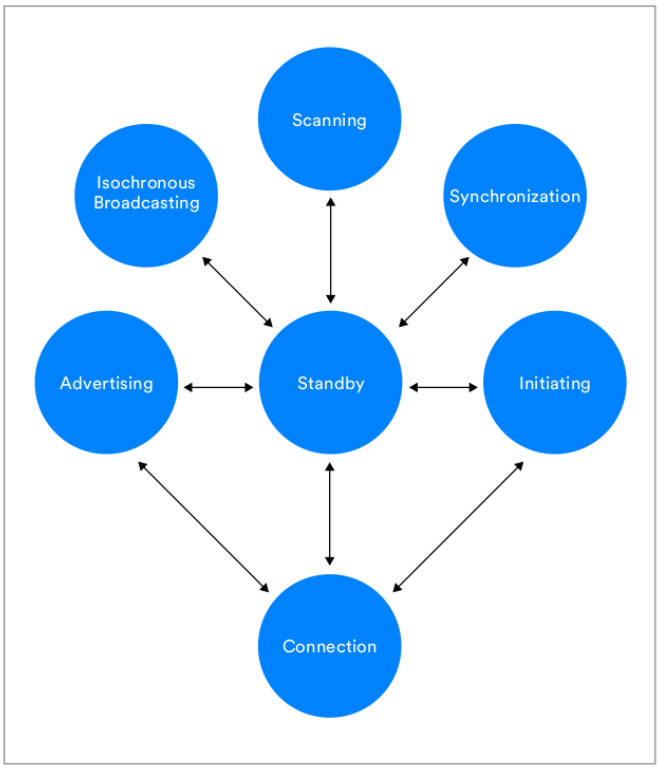


Figura 4.6. Macchina a stati del Link Layer [17]

quello *Connection* e assume il ruolo di *Central*. Un dispositivo che accetta una connessione passa dallo stato *Advertising* a quello *Connection* e assume il ruolo di *Peripheral*. Questa differerenziazione dei ruoli è una scelta di design di Bluetooth LE, che prevede capacità e responsabilità asimmetriche dei dispositivi. In questo modo dispositivi con batterie relativamente grandi come smartphone possano svolgere un lavoro più pesante rispetto a dispositivi funzionanti con batterie a bottone.

Bluetooth LE prevede maggiori possibilità per quanto riguarda gli indirizzi dei dispositivi. Infatti, oltre agli indirizzi già descritti nella sezione 4.1.2, che in BLE vengono chiamati *Public Device Addresses*, sono stati resi disponibili anche indirizzi randomici, chiamati *Random Device Addresses*. Questi indirizzi garantiscono un livello di privacy maggiore, per

esempio riducendo la possibilità di tracking dei dispositivi. I *Random Device Addresses* a loro volta si suddividono in base al valore dei due bit più significativi dell'indirizzo in *Static addresses*, *Non-resolvable Private Addresses* e *Resolvable Private Addresses*.

4.3.7 Host layers

In questa tesi non è stato fatto uso dei livelli Host di Bluetooth LE, di conseguenza non verranno qui descritti. Per approfondire è possibile consultare, come sempre, la *Bluetooth Core Specification* [14] oppure *The Bluetooth Low Energy Primer* [17].

4.4 BLE Advertising

La procedura di *advertising* viene utilizzata dai dispositivi Bluetooth LE di tipo Peripheral per indicare la loro disponibilità nello stabilire una connessione. Dato che l'advertising è una modalità di comunicazione connectionless nella quale i pacchetti sono destinati ad essere ricevuti da qualsiasi dispositivo nel raggio di azione, essa può essere sfruttata anche per diffondere dati in broadcast. Esistono due procedure di advertising, denominate *Legacy Advertising* e *Extended Advertising*.

4.4.1 Legacy Advertising

La procedura di legacy advertising utilizza tipicamente pacchetti di tipo ADV_IND, lunghi 37 ottetti, con 6 ottetti di header e 31 ottetti di payload.[1] Copie identiche dello stesso pacchetto sono trasmesse su un massimo di tre canali dedicati, i *primary advertising channels*, che corrispondono ai canali numerati 37, 38 e 39. La frequenza di trasmissione dei pacchetti di advertising segue il parametro *advInterval*, tuttavia essa viene deliberatamente resa irregolare da un parametro di timing pseudocasuale, definito *advDelay*, con valore in un intervallo da 0 a 10 ms. Lo scheduling di advertising events effettuato in questo modo permette di ridurre le collisioni con advertisement di altri dispositivi ma richiede anche un maggiore tempo in ascolto da parte di quei dispositivi interessati agli advertisement periodici, che dovranno far fronte alla tempistica non prevedibile.

Tipi di PDU La specifica definisce diversi tipi di PDU da poter utilizzare nel legacy advertising. Essi sono brevemente riportati in tabella 4.1. L'ADV_IND è il più comune e rappresenta un advertisement generico verso tutti, che permette di connettersi al dispositivo che l'ha inviato ed anche di richiedere ulteriori dati (*active scanning*) con i pacchetti di SCAN_REQ e SCAN_RSP. L'ADV_DIRECT_IND è come l'ADV_DIRECT ma è diretto verso un dispositivo specifico e prevede solo *passive scanning*. I restanti PDU rappresentano ulteriori combinazioni dei parametri *scannable* e *connectable*.

[1] Un ottetto rappresenta 8 bit, che nella grande maggioranza delle piattaforme coincide ad 1 byte.

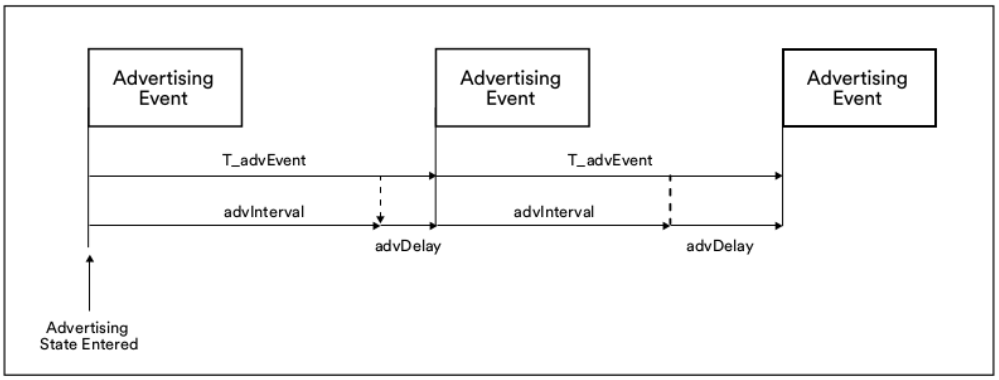


Figura 4.7. Sequenza temporale di advertising events con ritardo pseudocasuale [17]

PDU Name	Description	Channels	PHY(s)	Transmitted By	Scannable	Connectable
ADV_IND	Undirected advertising	primary	LE 1M	Peripheral	Y	Y
ADV_DIRECT_IND	Directed advertising	primary	LE 1M	Peripheral	N	Y
ADV_ NONCONN_IND	Undirected, non-connectable, non-scannable advertising	primary	LE 1M	Peripheral	N	N
ADV_SCAN_IND	Undirected, scannable advertising	primary	LE 1M	Peripheral	Y	N
SCAN_REQ	Scan request	primary	LE 1M	Central	N/A	N/A
SCAN_RSP	Scan response	primary	LE 1M	Peripheral	N/A	N/A
CONNECT_IND	Connect request	primary	LE 1M	Central	N/A	N/A

Tabella 4.1. PDU utilizzabili in Legacy Advertising [17]

4.4.2 Extended Advertising

La versione 5.0 della *Bluetooth Core Specification* ha introdotto alcuni importanti cambiamenti per quanto riguarda le modalità di advertising, aggiungendo otto nuovi tipi di PDU e definendo nuove procedure. Nel complesso, questo insieme di nuove funzionalità è stato denominato *Extended Advertising*. La tabella 4.2 sintetizza le principali differenze tra Legacy ed Extended Advertising. Tra queste, quella che risalta di più è la possibilità di trasmettere quantità di dati molto più grandi rispetto al Legacy Advertising, passando da un massimo di 31 a 1650 bytes.

Canali radio Gli extended advertisement utilizzano i canali radio in modo differente, trasmettendo solamente l'header nei *primary advertising channels* ed il payload su uno dei canali general-purpose. Questi ultimi sono i canali da 0 a 36 e sono chiamati anche

secondary advertising channels. Questa soluzione permette di trasmettere meno dati e di ridurre la possibilità di conflitti sui *primary advertising channels*.

Payload massimo Un singolo pacchetto di Extended Advertising supporta fino a 254 bytes di payload, nei casi in cui siano richiesti ancora più dati il controller può utilizzare tecniche di frammentazione per raggiungere un massimo di 1650 bytes.

Advertising sets Il legacy advertising non prevede formalmente la possibilità di variare il payload o i parametri dell'advertisement. Nell'extended advertising, invece, sono stati introdotti gli *Advertising Sets*, un meccanismo standard per avere set multipli e distinti di advertisement. Ogni set ha i propri parametri, tra cui l'advertising interval e il tipo di PDU da usare. Mentre nel legacy advertising era compito dell'Host programmare e trasmettere i diversi advertisement al momento giusto, con gli Advertising Sets è sufficiente per l'Host informare il Controller dei set da utilizzare ed i relativi parametri, poi sarà il Link Layer a gestire il tutto automaticamente.

	Legacy Advertising	Extended Advertising	
Max. host advertising data size	31 bytes	1,650 bytes	Extended Advertising supports fragmentation which enables a 50x larger maximum host advertising data size to be supported.
Max. host advertising data per packet	31 bytes	254 bytes	Extended Advertising PDUs use the Common Extended Advertising Payload Format which supports an 8x larger advertising data field.
TX channels	37,38,39	0-39	Extended Advertising uses the 37 general-purpose channels as secondary advertising channels. The ADV_EXT_IND PDU type may only be transmitted on the primary advertising channels (37, 38, 39) however.
PHY support	LE 1M	LE 1M LE 2M (excluding ADV_EXT_IND PDUs) LE Coded	All Extended Advertising PDUs except for ADV_EXT_IND may be transmitted using any of the three LE PHYs except for the ADV_EXT_IND PDU which may be transmitted using the LE 1M or LE Coded PHYs.
Max. active advertising configurations	1	16	Extended Advertising includes Advertising Sets which enable advertising devices to support up to 16 different advertising configurations at a time and to interleave advertising for each advertising set according to time intervals defined in the sets.
Communication types	Asynchronous	Asynchronous Synchronous	Extended Advertising includes Periodic Advertising, enabling time-synchronised communication of advertising data between transmitters and receivers.

Tabella 4.2. Principali differenze tra legacy ed extended advertising [17]

4.4.3 Formato pacchetti advertising

Il profilo GAP definisce il formato dei dati di Advertising e Scan Response, come mostrato in figura 4.8. Ogni struttura AD contiene al suo interno dati nel comune formato Type-Length-Value (TLV). Il campo Length definisce la lunghezza della sottostruttura AD Type e AD Data, l'AD Type è un ottetto che rappresenta il tipo di dato contenuto e l'AD Data contiene i dati effettivi da trasmettere. I valori del campo AD Type sono definiti dal Bluetooth SIG nella lista di *Assigned Numbers* [18] mentre i significati dei tipi di dato sono definiti nel *Supplement to the Bluetooth Core Specification* [19][Part A]. Una volta definiti i dati, essi saranno incapsulati all'interno di una delle PDU previste per l'advertising BLE e saranno pronti per la trasmissione.

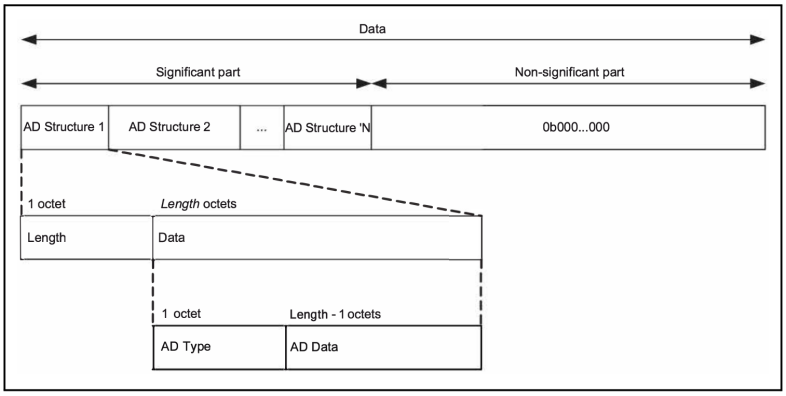


Figura 4.8. Formato dei dati di Advertising e Scan Response [14]

4.5 BLE Scanning

La procedura di *scanning* viene utilizzata dai dispositivi Bluetooth LE di tipo Central per connettersi ad un dispositivo in fase di advertising. Dato che l'advertising utilizza un massimo di tre canali e che l'advertiser e lo scanner non sono sincronizzati in alcun modo, un pacchetto di advertising sarà ricevuto con successo dallo scanner solo quando le due finestre di scanning ed advertising si sovrappongono casualmente, come mostrato in figura 4.9.

Due parametri influenzano la scansione, lo *scan interval* e la *scan window*. Il primo definisce ogni quanto far partire una scansione, il secondo quanto tempo dura. La modifica di questi parametri influenza profondamente il consumo di energia, essendo direttamente collegati alla quantità di tempo per cui dovrà stare accesa la radio.

4.6 BlueZ

Il progetto open-source *BlueZ* rappresenta l'implementazione ufficiale dello stack Bluetooth nella famiglia di sistemi Linux, fornendo agli sviluppatori di applicazioni l'accesso alle

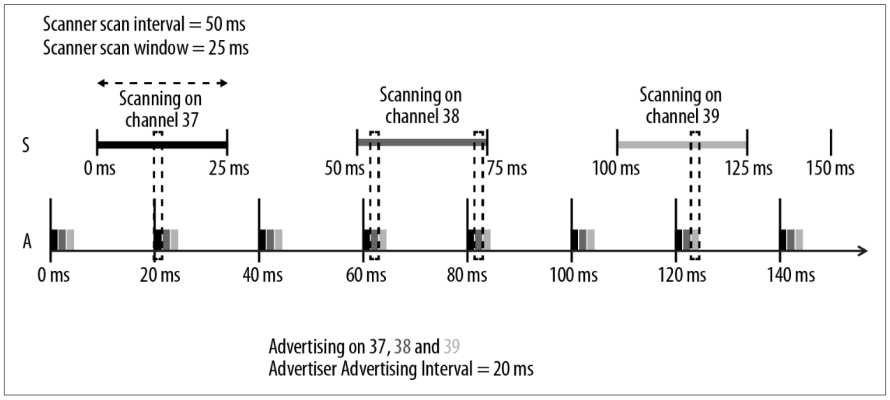


Figura 4.9. Advertising e scanning [13]

funzionalità Bluetooth presenti nel sistema. Come è possibile vedere in figura 4.10, BlueZ implementa i livelli host dello stack Bluetooth. La parte di controller, invece, risiede tipicamente all'interno di un chip dedicato oppure è implementata in una periferica come un dongle USB. Il demone Bluetooth è rappresentato dal servizio *bluetoothd* o *bluetooth-meshd* a seconda dello scopo, ed ha il compito di serializzare e gestire tutte le comunicazioni HCI per conto delle applicazioni.

Per quanto riguarda la comunicazione tra applicazioni e BlueZ, a partire dalla versione 5.52 è stato concluso il passaggio a D-Bus, un sistema di interprocess communication (IPC) presente su Linux. Prima dell'introduzione di D-Bus le applicazioni effettuavano chiamate dirette alle funzioni e ricevevano callback direttamente da BlueZ, includendo nel progetto i relativi header files. Con D-Bus i due ambienti sono completamente disaccoppiati, e per comunicare con lo stack Bluetooth è necessario utilizzare le API D-Bus per inviare e ricevere messaggi. Maggiori informazioni ed esempi sono disponibili all'indirizzo [20]. Le librerie esistenti prima di D-Bus sono ancora disponibili per retrocompatibilità, tuttavia non sono state aggiornate per supportare i cambiamenti introdotti nelle nuove versioni di Bluetooth.

BlueZ fornisce vari strumenti utili per il debugging, tra cui `btmon` per monitorare i comandi e gli eventi in transito dall'interfaccia HCI e `btmgmt` per modificare i parametri dell'adattatore Bluetooth da linea di comando. L'utilizzo di questi strumenti consente di avere un controllo completo del sottosistema Bluetooth.

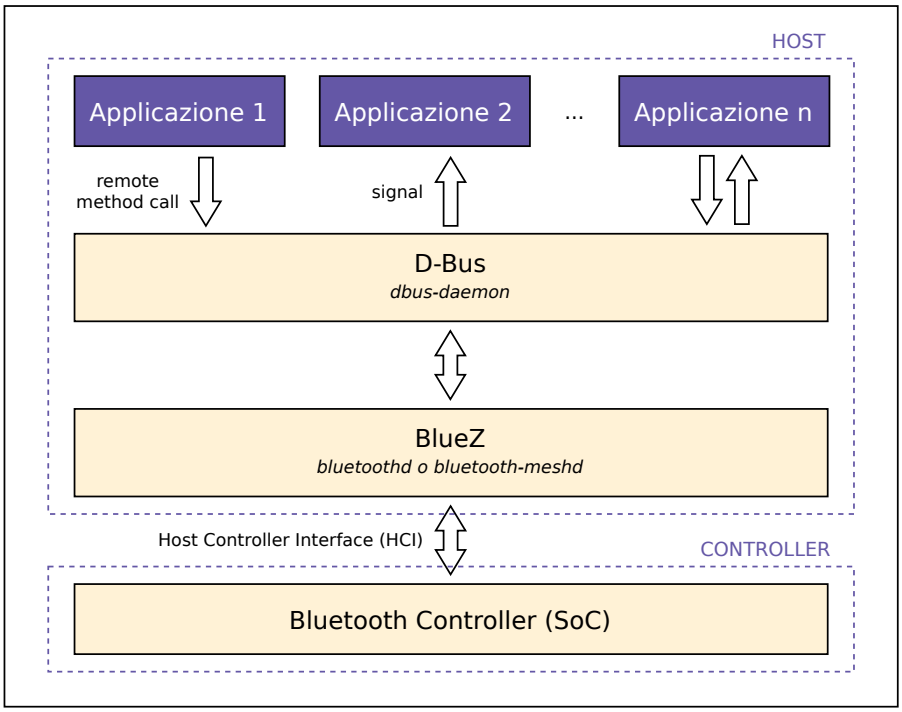


Figura 4.10. Architettura dello stack Bluetooth in Linux

Capitolo 5

Bluetooth Convergence Layer

In questo capitolo verrà presentata l'implementazione aggiornata del Bluetooth Convergence Layer in Æther, esponendo i problemi della vecchia versione e le funzionalità di quella attuale.

5.1 Punto di partenza

Bluetooth rappresenta una tecnologia estremamente diffusa sui dispositivi rientranti nella categoria dell'*Internet of Things* (IoT), grazie al basso costo e alla sua popolarità. Essendo una tecnologia wireless, è caratterizzata da connessioni non necessariamente stabili, con potenziali perdite di pacchetti o ritardi notevoli. L'uso di dispositivi mobili amplifica ancora di più il problema, dovuto al fatto che la disponibilità di connessione varia in base alla distanza tra i dispositivi. Per questo motivo l'uso del paradigma del Delay-Tolerant Networking può rappresentare in alcuni scenari una scelta migliore rispetto a paradigmi di comunicazione tradizionali. Il Bluetooth Convergence Layer viene introdotto in Æther da Pettinato [9] nel 2018, tuttavia un primo tentativo d'uso ha dimostrato come esso non fosse funzionante sui dispositivi moderni che supportano Bluetooth 5.0 e superiori.

5.2 Requisiti convergence layer

Per poter supportare nuovi mezzi di comunicazione, l'architettura DTN richiede due componenti fondamentali:

- Un *convergence layer*, che supporti la connessione e la trasmissione dati sul nuovo stack protocollare;
- Un *discovery agent*, che permetta di rilevare i membri della DTN nel proprio raggio di azione, al fine di abilitare il convergence layer al trasferimento di bundle.

Per quanto riguarda il *discovery agent*, la scelta implementativa effettuata è stata quella di utilizzare il meccanismo di scanning e advertising offerto da Bluetooth Low Energy. Tutti i dispositivi con il Bluetooth Convergence Layer abilitato trasmetteranno messaggi

in broadcast (advertisement) necessari per poter essere rilevati dai vicini attraverso la procedura di scanning. In questo modo è possibile popolare un set di dispositivi disponibili per la connessione, che verrà stabilita se sarà necessario trasmettere bundle.

Mentre il *discovery agent* utilizza Bluetooth LE, il *convergence layer* Bluetooth sfrutta invece Bluetooth BR/EDR per connettersi ai dispositivi individuati. Questa scelta è stata fatta in particolar modo per uniformità con la soluzione già presente, che non presentava problemi implementativi. Un cambiamento verso il trasferimento dati via Bluetooth LE avrebbe richiesto un impiego non indifferente di tempo per l'aggiornamento di buona parte del codice relativo al convergence layer. Una potenziale soluzione avrebbe potuto far uso di tecniche di emulazione di porte seriali su BLE, come descritto in [21]. In tal caso, andrebbero valutate anche eventuali differenze di performance per quanto riguarda il throughput ed il consumo energetico.

Un dispositivo che voglia utilizzare Bluetooth in Æther dovrà quindi necessariamente supportare sia Bluetooth BR/EDR sia Bluetooth LE, in caso contrario la comunicazione non sarà possibile.

5.3 Architettura

Il Bluetooth Convergence Layer è organizzato nelle seguenti classi:

- `ibrcommon`
  - `bluetooth`: libreria che rende disponibili funzioni per l'accesso all'interfaccia HCI del sistema Bluetooth;
  - `btsocket`: libreria per la creazione di socket su Bluetooth;
- `daemon`
  - `BluetoothConvergenceLayer`: rappresenta il convergence layer: crea connessioni in uscita, rimane in ascolto di connessioni in entrata, gestisce l'invio e la ricezione di dati;
  - `BluetoothDiscoveryAgent`: implementa la rilevazione dei dispositivi vicini mediante scanning e advertising, verificandone la loro appartenenza alla DTN;
  - `BluetoothConnection`: modella una connessione Bluetooth RFCOMM tra due nodi, gestendo sia la ricezione che l'invio di bundle;

5.4 Advertisement Æther

Per informare i dispositivi vicini della propria presenza, Æther utilizza advertisement di tipo ADV_IND, inviato su tutti e tre i primary channels disponibili per la procedura di advertisement. La PDU scelta, ADV_IND, è di tipo Legacy, per permettere la compatibilità con dispositivi che non supportano ancora Bluetooth 5.0. Questa decisione va a discapito della disponibilità di spazio per i dati nell'advertisement, che non possono superare i 31 byte. Per quanto riguarda proprio i dati, Æther inserisce nel proprio advertisement una

sola struttura di tipo *Service Data*, che al suo interno contiene, in sequenza, i seguenti campi:

- `IBRDTN_SERVICE_UUID16`
- RFCOMM channel
- EID scheme
- EID scheme-specific part

La struttura *Service Data* è definita dalla *Supplement to the Bluetooth Core Specification* [19] e permette di associare dei dati ad un servizio identificato da UUID. Gli UUID possono essere di tre tipi: a 16 bit, a 32 bit o a 128 bit. Gli UUID a 128 bit sono liberamente utilizzabili da chiunque, data la pressoché certa impossibilità di conflitti, mentre quelli a 16 o 32 bit possono essere utilizzati solo se assegnati dal Bluetooth SIG. Dato che Tierra, per il momento, non possiede un UUID assegnato, dovrebbe utilizzare un UUID a 128 bit. Tuttavia, siccome lo spazio di advertisement disponibile è di soli 31 bytes, perderne 16 per indicare il Service UUID risulta estremamente inefficiente. Per questo motivo Æther utilizza già dalla precedente versione un UUID generato casualmente a 16 bit, che per il momento non è stato assegnato dal Bluetooth SIG. È evidente come, in vista di un rilascio al pubblico, questa scelta non vada più bene, e sarà necessario procedere con la richiesta di assegnazione di un UUID per Tierra. Lo UUID a 16 bit, denominato `IBRDTN_SERVICE_UUID16` nella struttura del Service Data, ha valore `0x19EA`.

Nei restanti campi sono indicati il canale RFCOMM sul quale il dispositivo è in ascolto e l'EID del dispositivo. Il campo EID scheme è codificato per questioni di efficienza, ossia la stringa `dtn://` viene rappresentata col valore `0x01`. In questo modo è possibile utilizzare un solo byte per codificare sei caratteri, liberando spazio per la scheme-specific part.

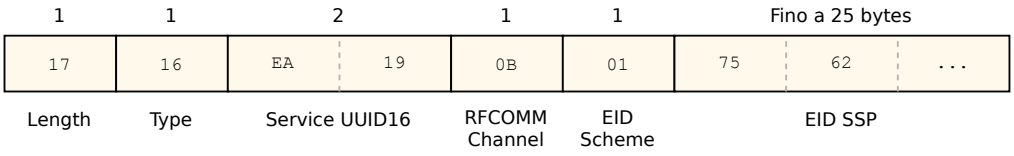


Figura 5.1. Esempio di advertisement Æther con dati reali

Un esempio di advertisement è mostrato in figura 5.1. Bluetooth utilizza il formato *little-endian*, che significa che il primo bit di ciascun numero coincide con il *least significant bit* (LSB). È importante impostare correttamente le conversioni nel codice per evitare errori di codifica.

Ogni dispositivo Æther che riceverà un advertisement di questo tipo esaminerà i dati contenuti e si renderà conto della presenza di un nuovo nodo vicino, al quale potrà trasmettere dati, se necessario, creando una nuova connessione RFCOMM.

5.5 Sicurezza connessioni RFCOMM

La classe `btsocket` imposta i criteri di sicurezza necessari per stabilire una connessione Bluetooth via RFCOMM al dispositivo. In particolare, ogni connessione richiede la mutua autenticazione e la cifratura dei dati che, come già descritto nella sezione 4.2.4, necessita della creazione di una *link key* condivisa. Le modalità di creazione della chiave sono varie, tuttavia l'unico modo per ottenere una chiave autenticata è la presenza di un'utente che verifichi la procedura. Per Æther, però, questo è un problema, in quanto si desidera che i dispositivi riescano a stabilire connessioni protette in modo automatico.

La versione precedente di Æther utilizzava uno script esterno scritto in Python per gestire il pairing di due dispositivi. Il suo scopo era quello di creare un *agent* che, attraverso la procedura di *Passkey Entry*, inserisse sempre lo stesso codice in entrambi dispositivi. In questo modo i due dispositivi si autenticavano con successo ed era possibile stabilire connessioni sicure. Uno degli scopi della seguente tesi è stato di eliminare la dipendenza da questo script esterno, cercando di rendere tutto automatico all'interno di Æther.

Una prima soluzione è stata quella di spostare la creazione dell'*agent* all'interno di Æther, ottenendo le stesse proprietà dello script Python. Anche in questo caso Æther inseriva una passkey statica, definita nel file di configurazione, durante la procedura di *Passkey Entry*. Un secondo sviluppo ha invece rimosso completamente la procedura di pairing, caricando direttamente all'interno del dispositivo una link key statica personalizzabile da file di configurazione. Questa soluzione consente di evitare il processo di pairing, che risulta oneroso in termini di tempo.

È evidente come né una link key né una passkey statica siano le scelte migliori per garantire la massima protezione da attacchi, tuttavia risultano essere le uniche possibilità fornite da Bluetooth che non prevedano l'interazione con l'utente. Questo è anche precisato dalla *Bluetooth Core Specification* [14] nel Vol. 2 Part H Section 7.2.3:

> Static Passkeys should not be used since they can compromise the security of the link.

L'articolo di Sun, Mu e Susilo [22] dimostra come siano possibili attacchi MITM su dispositivi che effettuino la procedura di *Passkey Entry* con passkey non casuale. In ogni caso, ulteriori approfondimenti sono necessari per avere garanzie dal punto di vista della sicurezza. L'uso di processi di autenticazione personalizzati e/o di canali OOB sicuri può essere un'alternativa da considerare per sviluppi futuri di Æther.

5.6 Problematiche Extended Advertising

La soluzione precedentemente adottata in Æther circa la gestione degli advertisement faceva uso di chiamate dirette a comandi del Controller, dichiarati dalla *Bluetooth Core Specification* nell'interfaccia HCI. Consideriamo, per esempio, la HCI request identificata da `OGF_LE_CTL` e `OCF_LE_SET_ADVERTISING_PARAMETERS`, come mostrato nel listing 5.1, che corrisponde al comando `HCI_LE_Set_Advertising_Parameters` descritto dalla specifica nel Vol. 4 Part E Section 7.8.5. Questo comando permette di impostare i parametri per gli advertisement, tra cui l'intervallo di advertising, il tipo e i canali su cui trasmettere.

```
void BluetoothLE::hci_set_le_advertising_parameters(int device_handle, uint16_t
    adv_interval) {

  uint8_t status;
  struct hci_request rq;

  le_set_advertising_parameters_cp adv_params_cp;

  memset(&adv_params_cp, 0, sizeof(adv_params_cp));
  adv_params_cp.min_interval = adv_interval;
  adv_params_cp.max_interval = adv_interval;
  adv_params_cp.chan_map = 7;
  adv_params_cp.advtype = 0x00;
  adv_params_cp.own_bdaddr_type = 0x00;

  memset(&rq, 0, sizeof(rq));
  rq.ogf = OGF_LE_CTL;
  rq.ocf = OCF_LE_SET_ADVERTISING_PARAMETERS;
  rq.cparam = &adv_params_cp;
  rq.clen = LE_SET_ADVERTISING_PARAMETERS_CP_SIZE;
  rq.rparam = &status;
  rq.rlen = 1;

  if(hci_send_req(device_handle, &rq, 10000) < 0) {
    throw hci_exception("cannot setting advertising parameters: " +
        std::string(strerror(errno)));
  }
}
```

Listing 5.1. Funzione `hci_set_le_advertising_parameters` - versione iniziale

L'introduzione di Bluetooth 5.0 ha portato alla nascita degli Extended Advertisement, come già descritto nella sezione 4.4.2. Per poterli gestire, la specifica ha introdotto nuovi comandi HCI, che vanno a integrare i comandi già presenti per i Legacy Advertisement. Il comando `HCI_LE_Set_Advertising_Parameters` rientra tra i comandi per il Legacy Advertisement, mentre l'equivalente Extended è `HCI_LE_Set_Extended_Advertising_Parameters`, descritto nel Vol. 4 Part E Section 7.8.53.

A questo punto è necessario presentare un punto di fondamentale importanza, tratto dalla specifica nel Vol. 4 Part E Section 3.1.1 e riportato qui di seguito:

> If, since the last power-on or reset, the Host has ever issued a legacy advertising command and then issues an extended advertising command, or has ever issued an extended advertising command and then issues a legacy advertising command, the Controller shall return the error code Command Disallowed (0x0C). A Host should not issue legacy commands to a Controller that supports the LE Feature (Extended Advertising).

Qui si presenta il problema, un Host non dovrebbe chiamare comandi Legacy su Controller che supportano l'Extended Advertising. Essendo una condizione non obbligatoria, la

chiamata potrebbe funzionare lo stesso, tuttavia la prima parte dell'estratto crea un altro problema, che si verifica se sono state effettuate chiamate precedenti a comandi Extended.

Sui sistemi Linux è presente e attivo di default il servizio `bluetoothd`, che agisce come parte Host del sistema Bluetooth (fig. 4.10), e che effettua correttamente chiamate HCI a comandi Legacy o Extended in base alla versione di Bluetooth supportata. L'esecuzione di comandi HCI Legacy da parte di Æther bypassa il controllo di `bluetoothd`, e di conseguenza fallisce siccome, in genere, `bluetoothd` ha già effettuato in precedenza chiamate a comandi Extended nel Controller.

Per risolvere il problema vi sono varie alternative:

1. Utilizzare Bluez ed il relativo servizio `bluetoothd` per gestire gli advertisement, attraverso l'interfaccia D-Bus fornita;

2. Aggiornare le chiamate HCI per gestire il caso in cui Bluetooth supporti gli Extended Advertisement, creando un Host Æther che lavori insieme a `bluetoothd`, senza però comunicare con esso;

3. Continuare ad usare chiamate Legacy disabilitando `bluetoothd`, in modo da creare un Host Æther indipendente.

La prima soluzione è la migliore dal punto di vista teorico: Bluez ha necessità di rappresentare l'unico punto di accesso per il Controller Bluetooth, per evitare conflitti con altre applicazioni che possano modificare i parametri del Controller. Bluez inoltre permette di gestire l'interazione con Bluetooth a un livello più alto, per esempio non sarà necessario distinguere tra Legacy o Extended Advertisement perché la scelta corretta sarà fatta automaticamente da Bluez. Questa strada è stata seguita per varie settimane, tuttavia, dopo vari tentativi di sviluppo, è stata abbandonata per alcuni problemi tecnici. In particolare, l'interfaccia D-Bus offerta da Bluez è risultata limitata rispetto alla quantità di comandi disponibili attraverso l'interfaccia HCI. Per esempio, i parametri `MinInterval` e `MaxInterval` sono modificabili solo se Bluez viene attivato in modalità Experimental, il che comporta la modifica della configurazione del sistema, ma anche in quel caso non si è riusciti a fare in modo che le modifiche venissero effettivamente applicate. La possibilità di Scan Response non è prevista da Bluez, limitando la quantità di dati disponibili per i Legacy Advertisement a soli 31 bytes. Più in generale l'uso di D-Bus è risultato complesso, per via della necessità di librerie esterne per l'accesso al sistema di comunicazione e per documentazione assolutamente scarsa sia da parte di D-Bus che da parte di Bluez.

La terza soluzione rappresenta la più drastica, impedendo l'utilizzo di Bluetooth alle normali applicazioni del sistema che fanno affidamento a Bluez. In questo modo, però, Æther ha il pieno controllo del Controller Bluetooth, e può effettuare chiamate HCI liberamente senza creare interferenze con Bluez. Dato che la seguente soluzione è molto limitante per il sistema si è deciso di scartarla.

La seconda soluzione è quella che è stata effettivamente applicata al progetto Æther e che verrà descritta nella sezione a seguire.

5.7 Modifiche al Convergence Layer

Le principali modifiche applicate al Bluetooth Convergence Layer riguardano le classi `BluetoothDiscoveryAgent` e `bluetooth`.

5.7.1 BluetoothDiscoveryAgent

Una prima modifica implementata è stata quella di rendere i parametri di advertisement modificabili da file di configurazione. Il costruttore della classe `BluetoothDiscoveryAgent` fissava i parametri a valori di default, che però non sempre rappresentano la scelta migliore per l'utente. Per esempio, dei test hanno dimostrato come in casi di congestione delle frequenze Bluetooth, ad esempio in zone con un elevato numero di dispositivi, valori alti di *advertising interval* non permettevano al dispositivo di essere rilevato in tempi accettabili.

```
BluetoothDiscoveryAgent::BluetoothDiscoveryAgent(const ibrcommon::vinterface &net,
    uint8_t rfcomm_channel) :
  _running(false), _timeout(20), _priority(10), device_handle(0), _net(net),
      _channel(rfcomm_channel), _scanInterval(160), _scanWindow(160),
      _advInterval(2048), dev_id(0) {
  memset(&of, 0, sizeof(of));
}
```

Listing 5.2. Costruttore BluetoothDiscoveryAgent - prima

```
 BluetoothDiscoveryAgent::BluetoothDiscoveryAgent(const ibrcommon::vinterface &net,
     uint8_t rfcomm_channel) :
  _net(net), _channel(rfcomm_channel),
  _btDevice(net.toString()), _btMgmt(net.toString()) {

  auto &config = dtn::daemon::Configuration::getInstance().getBluetooth();
  _minAdvInterval = config.getBluetoothMinAdvInterval();
  _maxAdvInterval = config.getBluetoothMaxAdvInterval();
  _scanInterval = config.getBluetoothScanInterval();
  _scanWindow = config.getBluetoothScanWindow();
  _linkKey = config.getBluetoothLinkKey();
}
```

Listing 5.3. Costruttore BluetoothDiscoveryAgent - dopo

La classe `BluetoothDiscoveryAgent` implementa l'interfaccia `IndependentComponent`, che rappresenta un componente che esegue il proprio codice, tipicamente un loop, in un thread separato. L'interfaccia prevede tre metodi principali, `initialize()`, `startup()` e `terminate()`, come mostrato in figura 5.2. Ciascun `IndependentComponent` deve implementare i metodi astratti `componentUp()`, `componentRun()` e `componentDown()`, inserendovi il codice che deve essere eseguito rispettivamente nell'inizializzazione, nell'esecuzione e nella terminazione del componente.

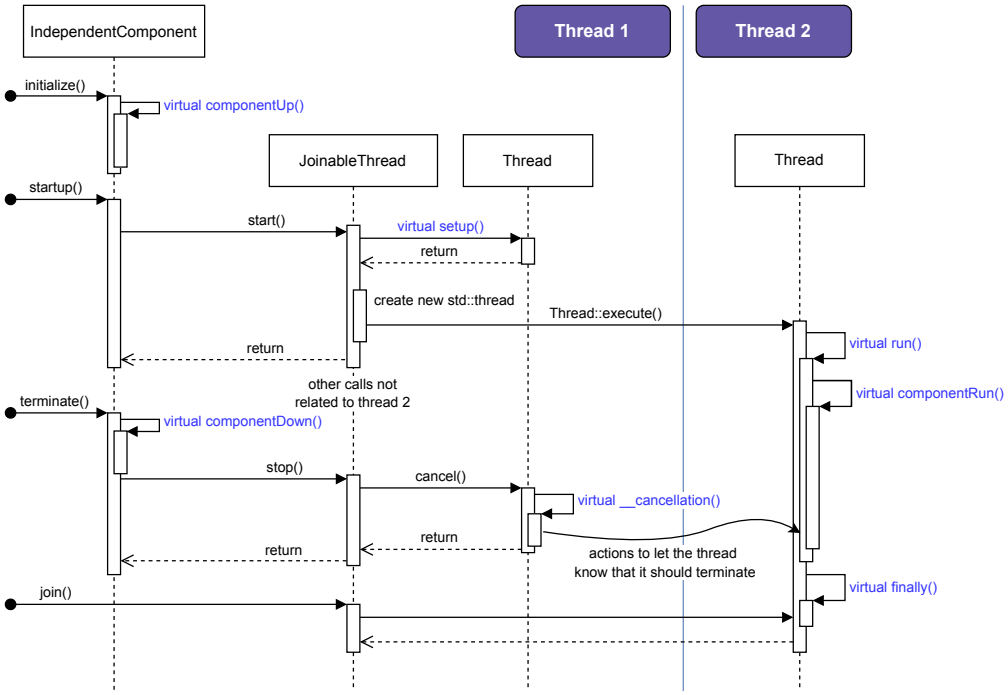


Figura 5.2. Flusso di chiamate relative a un IndependentComponent

`componentUp()` Il metodo `componentUp()` di `BluetoothDiscoveryAgent` si occupa di aprire la connessione verso l'interfaccia HCI, imposta il dispositivo Bluetooth BR/EDR in modalità connectable e configura i filtri sugli eventi Bluetooth da ricevere, in modo da scartare a priori gli eventi dei quali non si ha interesse. Una volta fatto questo, vengono chiamati i metodi `startScan()` e `startAdvertising()`, che hanno il compito di fare partire le fasi di scanning e advertising di Bluetooth LE. Se non si sono verificate eccezioni dall'interfaccia HCI, il metodo ritorna e `BluetoothDiscoveryAgent` può proseguire nel suo percorso.

Il codice relativo alla funzione `startScan()` non ha subito modifiche sostanziali rispetto alla versione iniziale presente in Æther. Esso imposta i parametri della scansione attraverso il metodo `hci_set_le_scan_parameters` e la fa partire con `hci_enable_le_scan`. Entrambi i metodi appartengono alla classe `bluetooth` e contengono le modifiche necessarie per il supporto ad Extended Advertising.

Per la funzione `startAdvertising()` è prevista la chiamata al metodo `hci_set_le_advertising_parameters` per stabilire i parametri di advertising, al metodo `set_le_advertising_data` per impostare i dati di advertising ed a `hci_set_le_scan_response` per impostare i dati di scan response. A questo punto è possibile abilitare l'advertising con la funzione `hci_enable_le_advertising`.

`componentRun()` Il metodo `componentRun()` viene eseguito in un thread indipendente ed ha il compito di verificare se gli advertisement ricevuti da altri dispositivi Bluetooth

provengano da membri della stessa DTN. Per fare questo ogni evento ricevuto dall'interfaccia HCI viene analizzato, e se i dati contenuti al suo interno seguono la struttura definita nella sezione 5.4 allora è possibile supporre che l'advertisement provenga da un dispositivo Æther. Ciò non è garantito, poiché un dispositivo può effettuare *spoofing* e inviare advertisement contenenti dati falsi. In tal caso, Æther si riempirebbe di nodi non esistenti, che possono impattare sensibilmente le performance di routing o impedire la connessione a nodi reali, ma che comunque non riusciranno a ricavare nessuna informazione sensibile dato che la comunicazione richiede l'autenticazione mediante *link key* condivisa.

Un evento di tipo `LE Advertising Report` o l'equivalente `LE Extended Advertising Report` possono contenere al loro interno più di un report, e ciò avviene nel caso in cui il Controller raggruppi in un unico evento gli advertisement di più nodi. Sebbene sia una pratica non comune, il caso va gestito per garantire il funzionamento su tutti i dispositivi. La precedente versione di Æther presentava un bug, in quanto gestiva gli advertisement dello stesso report in sequenza quando in realtà la specifica li definisce interlacciati. La figura 5.3 mostra molto chiaramente il problema, che è stato risolto in questa versione.

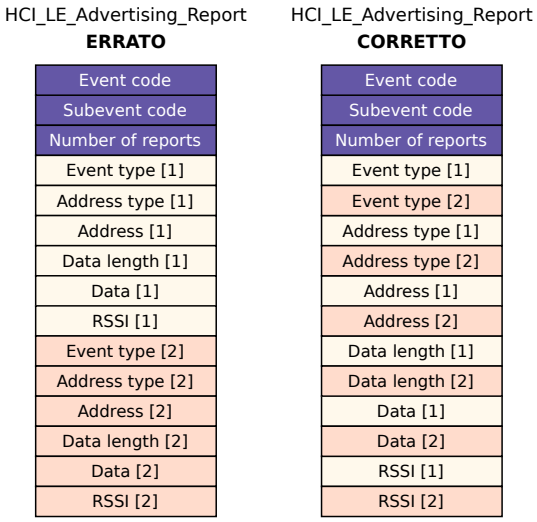


Figura 5.3. Evento LE Advertising Report contenente due report, a sinistra gestito in maniera errata, a destra in maniera corretta

Un'ultima modifica introdotta in `componentRun()` è l'introduzione di un timer, col compito di riabilitare periodicamente la scansione e l'advertising nel Controller Bluetooth. Non essendoci infatti cooperazione tra Æther e `bluetoothd`, si possono verificare casi in cui `bluetoothd` disabiliti la scansione o l'advertising, e in tal caso Æther smetterebbe di funzionare correttamente. Il timer evita, seppure con qualche ritardo, questa situazione spiacevole.

`componentDown()` Il metodo `componentDown()` ha il compito di interrompere il loop di `componentRun()` e far terminare il thread secondario. Inoltre, ferma le operazioni di scan e advertising e disabilita la proprietà connectable.

5.7.2 bluetooth

La classe `bluetooth` è stata aggiornata per supportare le chiamate a comandi HCI Extended e risolvere così le problematiche relative a Bluetooth 5.0. In particolare, tutti i metodi che prevedevano chiamate a comandi HCI di Legacy Advertising sono stati modificati per supportare le chiamate a comandi di Extended Advertising. Un esempio è mostrato nel listing 5.4. I nuovi comandi Extended, oltre ad avere codici di riferimento diversi, spesso dispongono di parametri aggiuntivi rispetto alle controparti Legacy.

```
void Bluetooth::hci_enable_le_scan() const {
  uint8_t status;
  struct hci_request enable_scan_rq{};

  if (_isExtended) {
    le_set_ext_scan_enable_cp scan_cp{};
    scan_cp.enable = 0x01;      // Scanning enabled
    scan_cp.filter_dup = 0x00;  // Duplicate filtering disabled
    scan_cp.duration = 0x00;
    scan_cp.period = 0x00;

    enable_scan_rq.ogf = OGF_LE_CTL;
    enable_scan_rq.ocf = 0x0042;
    enable_scan_rq.cparam = &scan_cp;
    enable_scan_rq.clen = sizeof(scan_cp);
    enable_scan_rq.rparam = &status;
    enable_scan_rq.rlen = 1;
    sendRequest(&enable_scan_rq, "cannot enable BLE scan");

  } else {
    le_set_scan_enable_cp scan_cp{};
    scan_cp.enable = 0x01;      // Scanning enabled
    scan_cp.filter_dup = 0x00;  // Duplicate filtering disabled

    enable_scan_rq.ogf = OGF_LE_CTL;
    enable_scan_rq.ocf = OCF_LE_SET_SCAN_ENABLE;
    enable_scan_rq.cparam = &scan_cp;
    enable_scan_rq.clen = LE_SET_SCAN_ENABLE_CP_SIZE;
    enable_scan_rq.rparam = &status;
    enable_scan_rq.rlen = 1;
    sendRequest(&enable_scan_rq, "cannot enable BLE scan");
  }
}
```

Listing 5.4. Metodo `hci_enable_le_scan()` con supporto a Extended Advertising

Il flag `_isExtended` viene impostato dal costruttore della classe `bluetooth`, chiamando il comando `HCI_LE_Read_Local_Supported_Features` e verificando il bit numero 12 del valore di ritorno. Se il bit ha valore `1` allora il Controller supporterà gli Extended Advertisement, in caso contrario sarà presente supporto solo a Legacy Advertisement. Come da raccomandazioni della *Bluetooth Core Specification*, un dispositivo che supporti gli Extended Advertisement effettuerà chiamate solamente a comandi HCI Extended.

Capitolo 6

Code refactor

Nel seguente capitolo verranno presentate numerose migliorie apportate al codice sorgente di Æther, dall'uso di C++17 all'uso di strumenti di static analysis, dalla rimozione dei warning ad una migliore ereditarietà delle classi.

6.1 Cos'è il code refactoring?

Il *code refactoring*, o più brevemente *refactoring*, è una fase importante dello sviluppo software che spesso viene trascurata. Lo scopo del refactoring è quello di migliorare la leggibilità, la manutenibilità, la riusabilità e l'estensibilità del codice, tutte caratteristiche non funzionali del software, puntando anche a ridurre la complessità del codice applicando a posteriori design pattern. Il refactoring modifica solamente la struttura interna del programma, lasciando inalterato il comportamento esterno del programma.

Senza alcun refactoring, qualsiasi progetto software è destinato a diventare sempre più complesso fino al punto di diventare ingestibile, provocando l'impossibilità di aggiunta di nuove funzionalità o la modifica di quelle presenti. Questo viene bene descritto dalla legge di Lehman sulla complessità crescente [23]:

> As a program is evolved its complexity increases unless work is done to maintain or reduce it.

La scarsa qualità del software è la principale debolezza dell'industria del software, e si traduce in spreco di tempo ma soprattutto di risorse economiche. Un'analisi di Jones [24] dimostra come un progetto valutato di scarsa qualità causi costi superiori dell'86% rispetto a un software classificato come eccellente. È evidente come il code refactoring sia solo una parte della gestione del software, ma a lungo termine può portare al risparmio di considerevoli quantità di denaro e permettere agli sviluppatori di lavorare su nuovi progetti.

È molto importante che le modifiche introdotte per il refactoring non introducano nuovi errori nel software, e per questo è consigliato applicare modifiche semplici e di facile comprensione. La presenza di test automatici può essere un ulteriore aiuto, rilevando eventuali problemi introdotti durante il refactoring. Va evidenziato come più i test sono completi

e più è possibile applicare cambiamenti rischiosi al codice, come spostamento di classi o modifica delle relazioni di ereditarietà.

6.1.1 Code smell

Il refactoring è solitamente determinato dal rilevamento dei cosiddetti *code smell*, ovvero pezzi di codice che presentano problemi non di funzionalità ma di chiarezza. Possiamo paragonare i sintomi di un paziente ad un code smell, mentre la malattia di fondo rappresenta il problema di design. Gli esempi di code smell sono centinaia, alcuni sono comuni altri molto rari, alcuni hanno un livello di gravità alto mentre altri non provocano particolari problemi. Tra i più comuni vale la pena citare:

- metodi molto lunghi;
- codice duplicato;
- nomi di funzioni o variabili non significative;
- classi troppo estese che contengono metodi non correlati tra loro;
- funzioni con troppi parametri;
- alta complessità ciclomatica, ovvero la presenza di un numero eccessivo di cicli o ramificazioni;

I code smell non rappresentano di per sé dei bug, tuttavia aumentano notevolmente la probabilità di inserirne di nuovi nel codice. Fortunatamente, il refactoring ha proprio lo scopo di rimuovere la maggiore quantità di code smell, per rendere il codice il più possibile facile da gestire. Ogni code smell può essere risolto, per esempio una funzione lunga può essere divisa in più sottofunzioni, il codice duplicato può essere dichiarato in un solo posto e condiviso, e così via.

6.1.2 Design smell

Oltre ai code smell esistono anche i *design smell*, problemi di più alto livello che indicano la violazione dei principi di progettazione del software ed hanno un impatto negativo sulla qualità del progetto. I design smell rendono il progetto fragile e difficile da mantenere, arrivando a casi in cui l'applicazione di una qualsiasi modifica può causare danni in parti non correlate del progetto. Tra i design smell più diffusi vi sono [25]:

- astrazioni mancanti, con più di una responsabilità o duplicate;
- incapsulamento lacunoso, se l'accessibilità di uno o più membri è più permissiva di quanto richiesto;
- incapsulamento non sfruttato, quanto il codice utilizza controlli di tipo espliciti anziché sfruttare il polimorfismo dinamico.
- modularizzazione errata, quando i dati e/o i metodi che idealmente avrebbero dovuto essere localizzati in un'unica astrazione sono separati e distribuiti in più astrazioni;

- modularizzazione insufficiente, quando esiste un'astrazione che non è stata completamente scomposta e un'ulteriore decomposizione potrebbe ridurne le dimensioni e/o la complessità di implementazione.

6.1.3 Debito tecnico

Tutte le lacune del software vanno ad accumulare il cosiddetto *debito tecnico* o *technical debt*, ossia quel debito che accresce quando si prendono, consapevolmente o meno, decisioni di progettazione sbagliate o non ottimali. Il debito tecnico è paragonabile al debito finanziario: se vengono pagate regolarmente le rate, il debito viene ripagato e non crea problemi, in caso contrario gli interessi continuano ad aumentare, fino al punto di poter portare il debitore in bancarotta. Nel caso del software, l'applicazione di soluzioni rapide piuttosto che ben progettate porta all'aumento del debito tecnico. Anche qui sono possibili due strade, ripagare in tempo il debito oppure accumulare gli interessi. Più il tempo passa e più è costoso per lo sviluppatore estinguere il debito, arrivando a situazioni in cui l'accumulo è così grande che l'introduzione di modifiche al software diventa estremamente ardua. Il caso estremo viene chiamato *bancarotta tecnica*, che comporta l'abbandono del progetto per via dell'accumulo costante di problemi non risolti.

Le principali fonti di debito tecnico sono le seguenti:

- **Code debt**: dovuto alla presenza di code smell e a stili di coding incoerenti tra loro;
- **Design debt**: a causa di design smell e violazione delle regole di progettazione;
- **Test debt**: mancanza di test, copertura inadeguata o progettazione impropria;
- **Documentation debt**: assenza di documentazione per le operazioni principali, documentazione carente o obsoleta.

I problemi che portano al debito tecnico sono per lo più invisibili e tendono a ricevere poca o nessuna attenzione da parte dei team di sviluppo. Infatti, mentre i bug presenti nel codice hanno un impatto diretto sugli attributi esterni, il debito tecnico impatta solamente la qualità interna del software, che non è direttamente percepibile dagli utenti finali del software. Il debito tecnico intacca i costi di sviluppo del progetto come anche il morale degli sviluppatori, che si troveranno davanti sistemi in cui risulta complesso introdurre cambiamenti, creando frustrazione e irritazione.

6.1.4 Strumenti per gestire il debito tecnico

Esistono varie categorie di strumenti per monitorare e limitare l'espansione del debito tecnico, che verranno descritte in seguito. Tuttavia, è evidente come la chiave per sviluppare un software di qualità sia quella di avere un team di ingegneri del software competenti e motivati, che possano lavorare senza pressioni eccessive dovute a deadline impossibili. Meno debito tecnico si sarà accumulato e meno risorse saranno necessarie per gestirlo.

Comprehension tools Gli strumenti di comprensione permettono di capire la struttura interna del codice a più alto livello, per esempio attraverso ausili visivi come l'analisi del data-flow, l'analisi delle sequenze di chiamate o grafici di ereditarietà. Questi strumenti

sono molto utili per grandi sistemi software, difficili da comprendere a causa delle loro dimensioni e della loro complessità.

Critique tools Gli strumenti di critica analizzano il software e segnalano problemi sotto forma di violazioni di regole predefinite. Questi strumenti possono segnalare violazioni architettoniche del software, design smell o code smell.

Metric tools Gli strumenti metrici consentono di ottenere dei report contenenti valori numerici relativi alla qualità del progetto. Un esempio è la metrica *Weighted Methods per Class* (WMC), che rappresenta la somma della complessità ciclomatica dei metodi di una classe. A questo punto è possibile fare una media di tutte le WMC, e se dovesse risultare superiore a una soglia prefissata allora significherebbe un'insufficiente modularizzazione del progetto.

Refactoring tools Esistono strumenti, tra cui alcuni IDE, che sono in grado di applicare automaticamente semplici passaggi di refactoring. Questi strumenti identificano le entità che necessitano di refactoring e propongono una soluzione al problema, è poi responsabilità del programmatore verificare che la soluzione sia valida. Un esempio di refactoring semi-automatico è l'estrazione di una classe da un'altra.

6.2 Refactoring Æther

6.2.1 C++17

Una delle prime specifiche fissate da Tierra è stata l'aggiornamento della versione di base di C++. Tutta la codebase di Æther era basata principalmente su C++98, e di conseguenza non sfruttava le possibilità introdotte dal linguaggio negli aggiornamenti più recenti. Nel processo di scelta della versione sulla quale sviluppare sono stati considerati due fattori fondamentali:

1. La presenza di nuove funzionalità che avrebbero potuto essere utili in Æther;
2. Il supporto da parte dei compilatori a tale versione.

La scelta finale è stata C++17, in quanto sufficientemente aggiornata con nuove funzionalità, una standard library estesa e il supporto completo da parte di quasi tutti i compilatori. L'eventuale spostamento a C++20 apporterebbe ancora più novità, tra cui concetti, coroutines e moduli, ma il supporto da parte dei compilatori risulta lacunoso su alcune specifiche.

In generale, il passaggio da C++98 a C++17, che comprende le versioni intermedie C++11 e C++14, rende possibile l'utilizzo di:

- Smart pointers, puntatori con gestione automatica dell'allocazione della memoria dinamica, utilizzabili come alternativa ai puntatori nativi;
- Concetto di movimento, che permette di risparmiare l'operazione onerosa di allocazione della memoria;

- Contenitori generici, dotati di algoritmi standard, al posto di array e strutture dati personalizzate.
- Lambda functions e keyword `auto`, a beneficio della compattezza del codice;
- Libreria standard per gestione della concorrenza, con supporto built-in a thread, operazioni atomiche, mutua esclusione e condition variables;
- Libreria standard per gestione del tempo, delle regular expression e del filesystem.

La lista di novità risulta essere veramente estesa, tanto che C++11 viene considerato il crocevia da C++ classico a C++ moderno.

6.2.2 File di configurazione

La configurazione di Æther era gestita da un file di testo semplice chiamato `dtnd.conf`. Il file, come parzialmente riportato nel listing 6.1, era un file non strutturato, notevolmente lungo e poco chiaro. Alcuni parametri venivano indicati in una certa categoria nel file mentre venivano parsati da una classe relativa ad un'altra categoria nel codice. In generale, la presenza di commenti, che chiaramente sono necessari per la comprensione dello scopo dei campi, rendeva il file poco immediato da leggere.

```
####################################
# IBR-DTN daemon                   #
####################################

#
# the local eid of the dtn node
# default is the hostname
#
#local_uri = dtn://Raspi-10

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

#
# Limit the block size of all bundles.
#
# The value accepts different multipliers.
# G = 1,000,000,000 bytes
# M = 1,000,000 bytes
# K = 1,000 bytes
#
#limit_blocksize = 1.3G

#
# Limit the block size of foreign bundles.
# Foreign bundles are not address from or to the
# local node.
#
# The value accepts different multipliers.
```

```
# G = 1,000,000,000 bytes
# M = 1,000,000 bytes
# K = 1,000 bytes
#
#limit_foreign_blocksize = 500M

#
# Limit the offset of predated timestamps to a max value.
# Bundles with an invalid timestamp will be rejected.
#
#limit_predated_timestamp = 604800

#
# Limit the max. lifetime of a bundle.
# Bundles with a lifetime greater than this value will be rejected.
#
limit_lifetime = 600000

# limit the numbers of bundles in transit (default: 5)
limit_bundles_in_transit = 100

# bind API to a named socket instead of an interface
#api_socket = /tmp/ibrdtn.sock

# 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
```

Listing 6.1. Parte iniziale del file `dtnd.conf`

Si è deciso quindi di passare ad una configurazione in formato YAML, in modo tale da avere categorie strutturate all'interno del file. La scelta di YAML rispetto a JSON è stata fatta per via del supporto da parte di YAML ai commenti e dal fatto che YAML risulta visualmente più facile da interpretare, grazie all'assenza di segni di punteggiatura come parentesi o virgolette.

Il risultato è mostrato nel listing 6.2. In particolare, i commenti sono stati spostati nel file `config-guide.pdf` in modo tale da rendere più facile la lettura del file, ed inoltre risultano evidenti le categorie, che forniscono una struttura logica al file. Queste modifiche hanno ridotto la lunghezza del file di configurazione da 777 a 248 righe, migliorando notevolmente la leggibilità.

```
#####################################
# IBR-DTN daemon configuration      #
#####################################

#
# Documentation for the following options
# can be found in the file config-guide.pdf
#
```

```
configuration:

  ####################################
  # General settings                 #
  ####################################
  general:
    #local_uri: "dtn://node.dtn"
    logfile: /var/log/ibrdtn/ibrdtn.log
    profiling: no

    #limit_blocksize: 1.3G
    #limit_foreign_blocksize: 500M
    #limit_predated_timestamp: 604800
    limit_lifetime: 604800
    limit_bundles_in_transit: 100

    #api_socket: /tmp/ibrdtn.sock
    api_interface: any
    api_port: 4550

  ####################################
  # Storage configuration            #
  ####################################
  storage:
    storage: default
    #blob_path: /tmp
    storage_path: /tmp/ibrdtn/bundles
    use_persistent_bundlesets: no
    #limit_storage: 20M
```

Listing 6.2. Parte iniziale del file `dtnd-short.yaml`

Classe Configuration La gestione della configurazione era affidata all'unico file sorgente `Configuration.cpp`, lungo 2250 righe. Esso aveva il compito di leggere l'intero file di configurazione, farne il parsing, memorizzare i dati come variabili membro e fornire metodi getter. È risultata subito evidente la necessità di dividere il codice in più file, in modo da ottenere file più comprensibili. I getter sono stati quindi spostati nel file `Configuration-Getters.cpp`, bilanciando la dimensione dei file. La classe è stata inoltre aggiornata per permettere il parsing di file YAML, mediante l'utilizzo della libreria open-source `yaml-cpp`.

6.2.3 ibrcommon

Tutta la libreria `ibrcommon` è stata aggiornata a C++17. Grazie alla disponibiltà della concurrency support library, introdotta in C++11, è stato possibile rimuovere la dipendenza dalla libreria `pthread` ed utilizzare la standard library al suo posto. Oltre a semplificare notevolmente il codice, la standard library ha il vantaggio di poter funzionare su qualsiasi piattaforma supportata dai compilatori C++.

Un esempio di semplificazione è riportata nel listing 6.3. Mentre in precedenza ogni piattaforma aveva modalità differenti per accedere al numero di processori del sistema, con

C++11 è possibile effettuare la chiamata a `std::thread::hardware_concurrency()` ed ottenere il dato in maniera uniforme.

```
// Prima
size_t Thread::getNumberOfProcessors() {
#ifdef __WIN32__
  SYSTEM_INFO sysinfo;
  GetSystemInfo(&sysinfo);
  return sysinfo.dwNumberOfProcessors;
#elif MACOS
  int nm[2];
  size_t len = 4;
  uint32_t count;

  nm[0] = CTL_HW; nm[1] = HW_AVAILCPU;
  sysctl(nm, 2, &count, &len, nullptr, 0);

  if(count < 1) {
    nm[1] = HW_NCPU;
    sysctl(nm, 2, &count, &len, nullptr, 0);
    if(count < 1) { count = 1; }
  }
  return count;
#else
  return sysconf(_SC_NPROCESSORS_ONLN);
#endif
}

// Dopo
size_t Thread::getNumberOfProcessors() {
  return std::thread::hardware_concurrency();
}
```

Listing 6.3. Funzione `getNumberOfProcessors()`

6.2.4 Routing

Le classi di routing sono state quelle più toccate dal processo di refactoring. In particolare, esse contenevano una notevole quantità di codice duplicato, con minime variazioni da classe a classe. Queste classi, di conseguenza, sono state estratte e rese comuni a tutti i differenti modelli di routing. Vari metodi di centinaia di righe sono stati semplificati, suddividendoli in più funzioni e rendendoli nettamente più comprensibili.

Classe Task La classe `Task` rappresenta un'operazione generica che deve essere gestita dalle classi di routing. Varie classi estendono `Task`, come `SearchNextBundleTask`, `ProcessBundleTask` e altre. Se consideriamo, per esempio, il comportamento della classe `StaticRoutingExtension`, alla ricezione di un evento del tipo `eventBundleQueued` verrà inserito un nuovo task del tipo `ProcessBundleTask` in una coda di `Task` locale, in attesa di essere processato. La gestione dei task nella versione iniziale di Æther è mostrata

nel listing 6.4. È evidente come la gestione del polimorfismo fosse inefficiente, applicando l'operazione di `dynamic_cast` ad ogni task e verificando che il cast avesse successo.

```
void StaticRoutingExtension::run() {
  while (true) {

    Task *t = _taskqueue.poll();

    try {
      SearchNextBundleTask &task = dynamic_cast<SearchNextBundleTask&>(*t);
      ...
    } catch (const std::bad_cast&) {};
    try {
      const ProcessBundleTask &task = dynamic_cast<ProcessBundleTask&>(*t);
      ...
    } catch (const std::bad_cast&) {};
    try {
      const RouteChangeTask &task = dynamic_cast<RouteChangeTask&>(*t);
      ...
    } catch (const std::bad_cast&) {};
    ...
  }
}
```

Listing 6.4. Gestione iniziale `Task` da parte di `StaticRoutingExtension`

Uno degli scopi di questa tesi è stato di migliorare questo pattern try-catch con `dynamic_cast`, soluzione che spreca inutilmente cicli di CPU per l'inevitabile gestione delle eccezioni di tipo `bad_cast`. L'obiettivo è stato quello di sfruttare in modo migliore il polimorfismo, arrivando ad ottenere una singola chiamata del tipo `task.runAction()` per qualsiasi tipologia di task. I paragrafi a seguire descriveranno in dettaglio la nuova organizzazione dei task, mentre il codice relativo è mostrato nel listing 6.5.

In generale, ogni classe di routing che elabora un task dovrà eseguire un'azione specifica della classe stessa, chiamando una funzione membro che abbia come parametro il task stesso. Di conseguenza è stata creata l'interfaccia `TaskAction<T>`, che specifica una funzione da chiamare alla ricezione di uno specifico task `T` definito dal parametro del template. Una classe di routing che volesse creare un task `T` deve implementare `TaskAction<T>` e passare il puntatore `this` al costruttore del task.

La classe template `TemplateTask` rappresenta la classe di base, che tutti i task dovranno estendere. La chiamata del metodo `runAction()` deve comportare la chiamata dell'azione associata al task specifico, e per fare ciò è necessario effettuare un down-cast al tipo derivato di task. Per rendere possibile quest'azione è stato applicato il pattern CRTP, che consiste nell'introduzione di una classe derivata che erediti dalla classe base e che allo stesso tempo usi se stessa come parametro template nella classe base. Dato che si sa con certezza che la classe passata come parametro template eredita dalla classe base, nella classe base è possibile utilizzare l'operazione di `static_cast` al posto di `dynamic_cast` e ottenere una migliore efficienza. Con questo meccanismo, la chiamata della funzione `runAction()` della classe base comporterà l'esecuzione del metodo corretto in base al tipo specifico di

task. Sarebbe stato possibile anche evitare il pattern CRTP, tuttavia ciò avrebbe comportato codice duplicato per ogni classe derivata, che avrebbe dovuto specificare il proprio comportamento per la funzione `runAction()`.

L'ultimo step è l'introduzione della classe `Task`, che rappresenta una classe base di livello ancora superiore che `TemplateTask` dovrà estendere. Questa classe è necessaria poiché le classi template non sono classi concrete. Infatti, il compilatore genererà una classe concreta solamente quando il template verrà istanziato specificando i parametri del template. Di conseguenza, non è possibile creare un container generico di `TemplateTask`, perché `TemplateTask` non è una classe. La classe `Task`, al contrario, è una classe concreta, e ciò rende possibile l'utilizzo in container come `std::vector<Task>` oppure smart pointer come `std::unique_ptr<Task>`.

```
template<class T>
class TaskAction {
  public:
  virtual ~TaskAction() = default;
  virtual void taskAction(const T &) = 0;
};

class Task {
  public:
  virtual ~Task() = default;

  virtual std::string toString() const = 0;
  virtual void runAction() = 0;
};

template<class T>
class TemplateTask : public Task {
  public:
  explicit TemplateTask(TaskAction<T> &obj) : actionObj(obj) {}
  ~TemplateTask() override = default;

  void runAction() override {
    actionObj.taskAction(static_cast<const T &>(*this));
  }

  private:
  TaskAction<T> &actionObj;
};

class SearchNextBundleTask : public TemplateTask<SearchNextBundleTask> {
  public:
  SearchNextBundleTask(const EID &eid, TaskAction<SearchNextBundleTask> &obj);
  ~SearchNextBundleTask() override;

  std::string toString() const override;
};

...
```

Listing 6.5. Gerarchia delle classi `Task`

Un esempio di sequenza di chiamate, basata sulla classe `StaticRoutingExtension` (listing 6.6), è la seguente:

1. La classe di routing riceve un evento che comporta la creazione di un nuovo task, come riportato dai metodi `eventDataChanged` ed `eventDataQueued`;

2. Il task specifico creato, ad esempio `SearchNextBundleTask`, richiede tra i parametri un riferimento ad un oggetto di tipo `TaskAction<SearchNextBundleTask>`. Le classi di routing quindi implementano tale interfaccia e passano come parametro il riferimento a `this`, a significare che l'azione da svolgere per il task è implementata dalla classe corrente;

3. Successivamente il task viene aggiunto alla coda `_taskqueue`, in attesa di essere processato;

4. Una volta estratto il task dalla coda con il metodo `poll()`, viene chiamato il metodo `runAction()` del task. Supponendo ancora che il task sia un `SearchNextBundleTask`, allora verrà chiamato il metodo `taskAction(const SearchNextBundleTask &)` dell'oggetto specificato al momento della creazione del task.

```
class StaticRoutingExtension :
  public TaskAction<SearchNextBundleTask>,
  public TaskAction<ProcessBundleTask>,
  ... {

  public:
  void taskAction(const SearchNextBundleTask &) override;
  void taskAction(const ProcessBundleTask &) override;
  ...

  void eventDataChanged(const EID &peer) override {
    _taskqueue.push(std::make_unique<SearchNextBundleTask>(peer, *this));
  }
  void eventBundleQueued(const EID &peer, const MetaBundle &meta) override {
    _taskqueue.push(std::make_unique<ProcessBundleTask>(meta, peer, *this));
  }

  protected:
  void run() override {
    while (true) {
      std::unique_ptr<Task> t = _taskqueue.poll();
      t->runAction();
    }
  }

  private:
  Queue<std::unique_ptr<Task>> _taskqueue;
}
```

Listing 6.6. Estratto della classe `StaticRoutingExtension`

6.3 Analisi statica del codice

Per portare avanti il refactoring di Æther è stato utilizzato uno strumento di analisi statica del codice sorgente chiamato *SonarLint*. SonarLint rientra tra gli strumenti di critica, in grado di rilevare code smell, bug o vulnerabilità di sicurezza attraverso un database di più di 4800 regole. SonarLint è un plugin installabile nel proprio IDE preferito in modo semplice e veloce, ottenendo i risultati dell'analisi del codice direttamente nell'interfaccia grafica dell'IDE. Sebbene CLion, l'IDE utilizzato in questa tesi, abbia già integrate funzionalità di static analysis, SonarLint sì è dimostrato molto più efficace, specifico e dettagliato. SonarLint non si limita a segnalare i problemi, ma descrive anche il perché vengono riportati e quali potenziali soluzioni esistono.

Due ulteriori fonti di raccomandazioni relative alla scrittura di codice C++ sono le *C++ Core Guidelines* [26] e *SEI CERT C++ Coding Standard* [27]. Le *C++ Core Guidelines* sono linee guida per la scrittura di codice C++ pulito, curate direttamente da Bjarne Stroustrup, creatore del linguaggio C++. Lo *SEI CERT C++ Coding Standard* invece contiene regole e raccomandazioni sulla scrittura di codice C++ sicuro, che permettano di migliorare l'affidabilità e la protezione del software.

A seguire verranno presentate le regole più violate nel codice di Æther e quali modifiche sono state applicate per risultare conforme.

RSPEC-5025 La regola RSPEC-5025 di SonarLint è una delle più importanti, indicata anche dalle *C++ Core Guidelines* R.11 e C.149. Essa riporta il fatto che la memoria non dovrebbe essere gestita in modo manuale, con `new` e `delete`, ma utilizzando gli *smart pointers* introdotti in C++11. Se si gestisce la memoria manualmente, è responsabilità dell'utente liberare tutta la memoria creata con `new` e assicurarsi che venga deallocata una e una sola volta. Assicurarsi che ciò avvenga è soggetto a errori, soprattutto quando la funzione può avere punti di uscita anticipati. C++11 ha definito le funzioni `std::make_unique` e `std::make_shared`, che consentono di creare i corrispettivi `unique_ptr` e `shared_ptr`, puntatori in grado di gestire autonomamente l'allocazione e la deallocazione della memoria. In Æther, sfortunatamente, tutta la gestione della memoria è manuale. Nel seguente lavoro di tesi la maggior parte dei puntatori nativi incontrati sono stati sostituiti con smart pointers, tuttavia rimane ancora larga parte del codice con allocazione manuale della memoria.

```
// Noncompliant
NeighborDatabaseEntry &NeighborDatabase::create(const EID &eid) {
  auto iter = _entries.find(eid);
  if (iter == _entries.end()) {
    auto *entry = new NeighborDatabaseEntry(eid);
    const auto [iter_inserted, hasInserted] = _entries.try_emplace(eid, entry);
    iter = iter_inserted;
  }
}
void NeighborDatabase::remove(const EID &eid) {
  auto iter = _entries.find(eid);
  if (iter == _entries.end()) {
    delete (*iter).second;
```

```
    _entries.erase(iter);
  }
}

// Compliant
NeighborDatabaseEntry &NeighborDatabase::create(const EID &eid) {
  auto iter = _entries.find(eid);
  if (iter == _entries.end()) {
    auto entry = std::make_unique<NeighborDatabaseEntry>(eid);
    const auto [iter_inserted, hasInserted] = _entries.try_emplace(eid,
        std::move(entry));
    iter = iter_inserted;
  }
}
void NeighborDatabase::remove(const EID &eid) {
  auto iter = _entries.find(eid);
  if (iter == _entries.end()) {
    _entries.erase(iter);
  }
}
```

Listing 6.7. Esempio RSPEC-5025

RSPEC-4963 La RSPEC-4963 è un'altra regola alla base dello sviluppo C++, che raccomanda l'utilizzo della cosiddetta *Rule-of-Zero*. La maggior parte delle classi non ha necessità di gestire direttamente le risorse, ma può affidarsi a membri che le gestiscono per loro, come gli *smart pointers* per la memoria e `ifstream` per i file. Proprio per questo motivo tali classi non hanno bisogno di definire nessuna delle funzioni membro speciali necessarie per gestire correttamente le risorse: distruttore, costruttore di copia/movimento e operatore di assegnazione di copia/movimento. Queste funzioni verranno generate automaticamente dal compilatore e svolgeranno la propria funzione in modo corretto. Da qui nasce la *Rule-of-Zero*, che si contrappone alla *Rule-of-Five* dove tutte e cinque le funzioni speciali sono dichiarate. La *Rule-of-Five* viene citata dalla regola RSPEC-3624, indicando il fatto che quando la *Rule-of-Zero* non può essere applicata allora bisogna seguire la *Rule-of-Five*, così da non provocare problemi di integrità dovuti ad un'implementazione errata da parte del compilatore.

RSPEC-5028 La regola RSPEC-5028 si basa sull'equivalente regola ES.31 definita dalle *C++ Core Guidelines*. Essa consiglia di non utilizzare le macro per definire le costanti. Le macro infatti sono sostituzioni testuali, il che significa che non rispettano il sistema dei tipi né le regole di scoping, rappresentando una delle principali fonti di bug. Nella maggior parte dei casi una macro può essere sostituita da una dichiarazione `constexpr`, introdotta in C++11, che permette di calcolare la costante a tempo di compilazione rispettando sia lo scoping che il sistema dei tipi. Nel caso in cui siano presenti varie macro correlate tra loro, è bene considerare la sostituzione con una `enum`, come definito dalla regola Enum.1.

RSPEC-6005 La regola RSPEC-6005 consiglia l'uso del *structured binding*, una sintassi introdotta in C++17 che consente di inizializzare più entità da elementi o membri di un

```
// Noncompliant
#define HEADER_LENGTH 56

#define LEFT   0
#define RIGHT  1
#define JUMP   2
#define SHOOT  3

// Compliant
constexpr int HEADER_LENGTH = 56;
enum class Actions {Left, Right, Jump, Shoot};
```

Listing 6.8. Esempio RSPEC-5028

oggetto. Æther presenta molte funzioni che restituiscono delle coppie, del tipo `std::pair`, che nel caso tipico andrebbero memorizzate in una variabile temporanea per poi effettuare l'accesso ai campi con i metodi `.first` e `.second`. Lo *structured binding* permette di evitare questo passaggio inizializzando direttamente due variabili, come mostrato nel listing 6.9. In questo modo è possibile utilizzare dei nomi che rappresentino lo scopo delle variabili, rendendo il codice più leggibile e più facile da debuggare.

```
// Noncompliant
for (const auto &pair: _latencyTable) {
  if (pair.first.getEID().sameHost(neighbor)) {
    return pair.second._lastavg.get();
  }
}

// Compliant
for (const auto &[node, latency]: _latencyTable) {
  if (node.getEID().sameHost(neighbor)) {
    return latency._lastavg.get();
  }
}
```

Listing 6.9. Esempio RSPEC-6005

RSPEC-6030 La funzione `insert` in una `std::map`, se chiamata con un oggetto temporaneo, comporta una copia o un movimento non necessario e la relativa distruzione del temporaneo. Al suo posto potrebbe essere usato il metodo `emplace`, che crea l'oggetto direttamente nella mappa. Tuttavia, se la chiave è già presente nella mappa, l'operazione di `emplace` crea l'oggetto e lo distrugge immediatamente, con il conseguente spreco di computazione. In C++17 è stato introdotto il metodo `try_emplace`, con lo scopo di costruire l'oggetto nella mappa solo se la chiave non è presente. Ciò permette di migliorare l'efficienza del codice, limitando la costruzione degli oggetti solo al caso in cui sia effettivamente necessaria.

```
// Noncompliant
latencyTable::iterator it;
if ((it = _latencyTable.find(neighbor)) == _latencyTable.end()) {
  LatencyParameter _lp(_A);
  _latencyTable.insert(std::make_pair(neighbor, _lp));
}

// Compliant
LatencyParameter _lp(_A);
_latencyTable.try_emplace(neighbor, _lp);
```

Listing 6.10. Esempio RSPEC-6030

RSPEC-3230 La regola RSPEC-3230 segue quanto indicato dalle *C++ Core Guidelines* C.48 e C.49, e indica il fatto che le variabili membro dovrebbero essere essere inizializzate *in-class* quando possibile. Per le variabili membro di una classe sono infatti possibili tre vie:

1. Inizializzandole *in-class*, da C++11;
2. Nell'elenco di inizializzazione di un costruttore;
3. Nel corpo del costruttore.

Si dovrebbero usare questi metodi in questo ordine di preferenza. Gli inizializzatori *in-class* sono i migliori perché si applicano automaticamente a tutti i costruttori della classe, ma possono usare solo valori costanti. L'elenco di inizializzazione può essere utilizzato se il valore dipende da un parametro, mentre l'inizializzazione nel corpo del costruttore andrebbe evitata.

```
// Noncompliant
class StaticRegexRoute {
  public:
  StaticRegexRoute(const std::string &regex, const EID &dest)
    : _dest(dest), _regex_str(regex), _invalid(false), _expire(0) {}

  private:
  EID _dest;
  std::string _regex_str;
  bool _invalid;
  const Timestamp _expire;
}

// Compliant
class StaticRegexRoute {
  public:
  StaticRegexRoute(const std::string &regex, const EID &dest)
    : _dest(dest), _regex_str(regex) {}

  private:
  EID _dest;
  std::string _regex_str;
```

```
  bool _invalid{false};
  const Timestamp _expire{0};
}
```

Listing 6.11. Esempio RSPEC-3230

RSPEC-1709 I costruttori con un singolo parametro dovrebbero essere dichiarati con la parola chiave `explicit`, per evitare conversioni implicite del parametro. Normalmente, la chiamata di un metodo con argomenti del tipo sbagliato provoca un errore in fase di compilazione. Questo non vale per i costruttori con un singolo argomento, per i quali il compilatore tenta di creare implicitamente un oggetto del tipo corretto. Questo può provocare conversioni involontarie o difficili da capire, che porterebbero a comportamenti inaspettati.

6.4 Testing

Il progetto IBR-DTN portava con sé un numero discreto di test, sfruttando la libreria CppUnit per il controllo delle asserzioni. In Æther, tuttavia, questi test non sono mai stati utilizzati. Per questo motivo nel seguente lavoro di tesi sono stati creati i target appositi nel build file CMake, così da poter lanciare i test e verificare il successo o il fallimento di essi. Mentre la maggior parte dei test ha riportato esito positivo fin dall'inizio, un numero limitato di essi ha evidenziato alcuni problemi del software, che sono stati quindi analizzati e corretti.

6.4.1 Code coverage

La *code coverage* è una metrica che può aiutare a capire quanta parte del codice sorgente viene testato, in modo da fornire una valutazione della qualità della suite di test utilizzata. Gli strumenti di *code coverage* possono rilevare varie statistiche, tra queste:

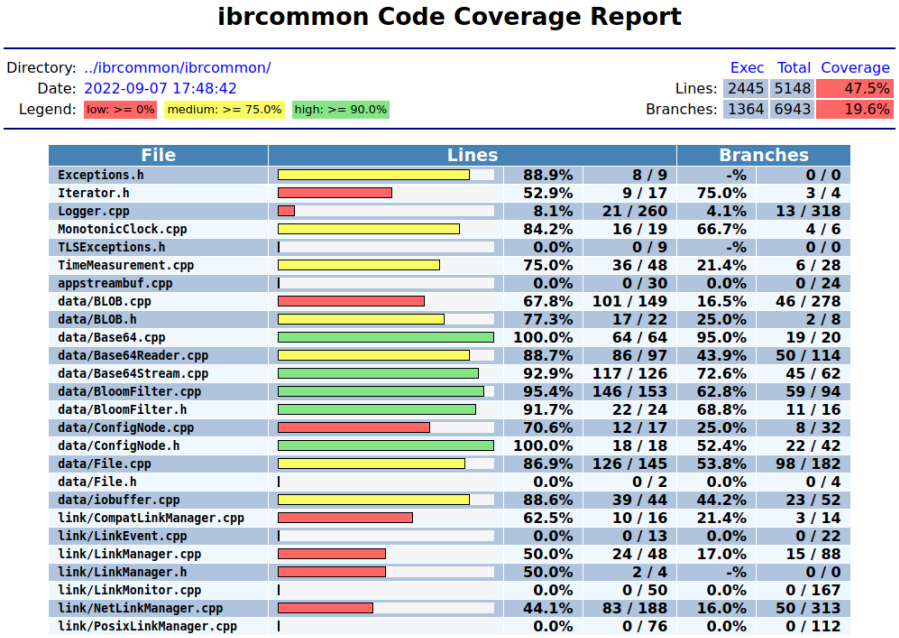
- Function coverage: quante funzioni definite sono state chiamate;
- Statement coverage: quante istruzioni del programma sono state eseguite;
- Branches coverage: quante ramificazioni delle strutture di controllo sono state seguite (es. istruzioni `if`);
- Line coverage: quante righe di codice sono state testate.

Le metriche sono solitamente rappresentate dal rapporto tra elementi testati ed elementi trovati, in modo da poter ottenere una percentuale molto immediata da interpretare. Non esiste una regola fissa per la percentuale di copertura del codice, perché anche in casi con alta copertura possono esserci istruzioni critiche non testate. Detto ciò, è opinione comune che una copertura dell'80% sia un buon obiettivo da raggiungere.

Grazie ai report di code coverage è molto facile individuare quali sono quelle classi non testate a sufficienza, ed è possibile intervenire creando nuovi test che agiscano su tali aree. È bene notare, tuttavia, che una buona code coverage non significa necessariamente che

i test siano anch'essi buoni. Raggiungere un'ottima copertura è un obiettivo importante ma deve essere abbinato a una suite di test robusta, in grado di garantire l'integrità del sistema.

Per poter verificare la code coverage del progetto Æther è stato creato uno script in grado di produrre dei report mediante lo strumento `gcovr`. Più nel dettaglio, lo script `code-coverage.sh` controlla che `gcovr` sia installato nel sistema, lancia i test presenti in Æther e produce dei file HTML contenenti i risultati di code coverage. Un estratto del report del modulo `ibrcommon` è riportato in figura 6.1.



ibrcommon Code Coverage Report

Directory: ../ibrcommon/ibrcommon/
Date: 2022-09-07 17:48:42
Legend: low: >= 0% medium: >= 75.0% high: >= 90.0%

	Exec	Total	Coverage
Lines:	2445	5148	47.5%
Branches:	1364	6943	19.6%

File	Lines			Branches	
Exceptions.h	88.9%	8 / 9		-%	0 / 0
Iterator.h	52.9%	9 / 17		75.0%	3 / 4
Logger.cpp	8.1%	21 / 260		4.1%	13 / 318
MonotonicClock.cpp	84.2%	16 / 19		66.7%	4 / 6
TLSExceptions.h	0.0%	0 / 9		-%	0 / 0
TimeMeasurement.cpp	75.0%	36 / 48		21.4%	6 / 28
appstreambuf.cpp	0.0%	0 / 30		0.0%	0 / 24
data/BLOB.cpp	67.8%	101 / 149		16.5%	46 / 278
data/BLOB.h	77.3%	17 / 22		25.0%	2 / 8
data/Base64.cpp	100.0%	64 / 64		95.0%	19 / 20
data/Base64Reader.cpp	88.7%	86 / 97		43.9%	50 / 114
data/Base64Stream.cpp	92.9%	117 / 126		72.6%	45 / 62
data/BloomFilter.cpp	95.4%	146 / 153		62.8%	59 / 94
data/BloomFilter.h	91.7%	22 / 24		68.8%	11 / 16
data/ConfigNode.cpp	70.6%	12 / 17		25.0%	8 / 32
data/ConfigNode.h	100.0%	18 / 18		52.4%	22 / 42
data/File.cpp	86.9%	126 / 145		53.8%	98 / 182
data/File.h	0.0%	0 / 2		0.0%	0 / 4
data/iobuffer.cpp	88.6%	39 / 44		44.2%	23 / 52
link/CompatLinkManager.cpp	62.5%	10 / 16		21.4%	3 / 14
link/LinkEvent.cpp	0.0%	0 / 13		0.0%	0 / 22
link/LinkManager.cpp	50.0%	24 / 48		17.0%	15 / 88
link/LinkManager.h	50.0%	2 / 4		-%	0 / 0
link/LinkMonitor.cpp	0.0%	0 / 50		0.0%	0 / 167
link/NetLinkManager.cpp	44.1%	83 / 188		16.0%	50 / 313
link/PosixLinkManager.cpp	0.0%	0 / 76		0.0%	0 / 112

Figura 6.1. Estratto di code coverage `ibrcommon`

Capitolo 7

Build process

Nel seguente capitolo verranno presentati gli aggiornamenti relativi al build process di Æther. Nella prima parte verrà presentato lo scopo e i vantaggi di CMake, che rappresenta il build system adottato da Æther, mentre la seconda parte descriverà le modifiche più importanti applicate al progetto.

CMake è una famiglia di strumenti open-source multi-piattaforma per gestire le fasi di build, testing e packaging del software, che permette di controllare la compilazione del software mediante comandi indipendenti dal tipo di piattaforma e dal compilatore utilizzato. CMake consolida le diverse operazioni in un unico formato di file, multi-piattaforma, semplice e di facile comprensione.

7.1 Build systems

CMake, così come moltissimi altri programmi similari, rientra tra gli strumenti di *build automation*. Un *build system* ha il compito di automatizzare la creazione del software, prevedendo le seguenti funzioni:

- compilazione del codice sorgente in codice binario;
- pacchettizzazione del codice binario;
- esecuzione di test automatici.

Storicamente, per ottenere la build automation veniva utilizzato lo strumento Make e i relativi makefiles. Con l'introduzione di progetti sempre più complessi e piattaforme nuove e differenti si è notato come l'uso di Make non fosse più sufficiente, portando alla nascita di nuovi strumenti. Sfortunatamente, ancora ad oggi non è presente un unico build system completo e dominante ma svariate implementazioni ognuna con i propri pregi e difetti. Questo rimane un punto dolente dello sviluppo software: nonostante anni di innovazioni e linguaggi di sempre più alto livello, la build automation rimane più complicata di quanto ci si possa aspettare.

7.1.1 Perché usare un build system?

Il processo di compilazione, linking e installazione del software è da sempre tedioso e pieno di insidie. È necessario organizzare i file sorgente, gestire le dipendenze da file interni o da librerie esterne, gestire le opzioni di compilazione e di linking e numerose altre variabili. Se poi il software è cross-platform sarà necessario avere diverse copie dello stesso build file, una per ciascuna piattaforma da supportare. Storicamente, la gestione di questi file era un compito manuale che richiedeva uno sviluppatore incaricato di mantenerli allineati, in modo tale da non avere incoerenze tra le diverse copie. Il supporto a componenti addizionali, come libreria esterne, rende il tutto ancora più complesso da mantenere. La questione da risolvere è evidente ed è stata la motivazione per la nascita di nuovi build system in grado di generare varie versioni a partire da un unico file.

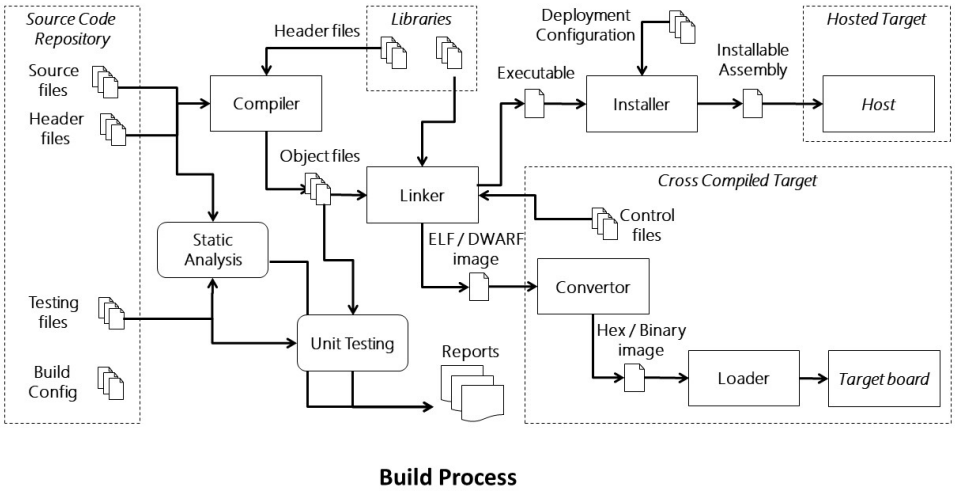


Figura 7.1. Rappresentazione grafica di un build process [28]

Cerchiamo di rendere il tutto più chiaro prendendo come esempio lo sviluppo di un'applicazione per smartphone, con piattaforme di target iOS ed Android. Per testare le modifiche si prevede anche di supportare i sistemi Linux e OSX, al fine di evitare di usare un emulatore o di dover trasferire ogni volta l'applicazione su un dispositivo mobile. In questo caso è quindi richiesto il supporto a quattro piattaforme differenti: saranno necessari dei makefiles per le piattaforme UNIX, un file di progetto per Apple Xcode e dei makefiles differenti per Android. L'obiettivo sarà lo stesso, ma il contenuto dei file varierà tra i diversi file.

7.1.2 Make

GNU Make è uno strumento che controlla la generazione di file eseguibili a partire dai file sorgenti. Make è il più longevo dei build system con il primo rilascio pubblicato nel 1976, addirittura sette anni prima della nascita del progetto GNU. Il funzionamento di Make prevede la preparazione di un makefile che contenga al suo interno le cosiddette

recipes, sequenze di istruzioni che indicano i file da creare, le loro dipendenze e i comandi da eseguire per costruirli. Nella scrittura di makefiles è facile perdere dei pezzi: se per esempio un file a.cpp dipende da b.h che a sua volta include c.h, è necessario elencare tutte e tre le dipendenze nell'applicazione affinché la compilazione venga eseguita quando una qualsiasi di esse venga modificata. Nonostante gli anni, Make continua ad essere ampiamente utilizzato, semplicemente sono stati creati nuovi build system di più alto livello in grado di generare i makefiles automaticamente.

7.1.3 Autotools

Autotools è un build system basato su Make, composto da due componenti principali: Autoconf e Automake. Esso nasce con l'obiettivo di semplificare il processo di build, nascondendo molti dettagli della piattaforma in uso e permettendo di usare regole semplici al posto di complessi makefiles. La realtà è che, a detta di molti, Autotools si evidenzia come uno dei build system più complessi di sempre. Inoltre, permette la portabilità solo su sistemi Unix-like. Per l'ambiente Windows, invece, Autotools richiede l'installazione di molti strumenti aggiuntivi che non sono nativamente presenti, implicando un'ulteriore scomodità per lo sviluppatore.

7.1.4 Build system moderni

Tra i build system più recenti troviamo CMake, SCons, Premake, Bazel, Meson e numerosi altri. Tra questi, CMake risulta essere il più popolare data la sua completezza e maturità. La prima implementazione di CMake risale all'anno 2000, dunque non proprio recente, ciò nonostante il continuo sviluppo ed espansione del linguaggio ha consentito ad esso di restare al passo con i tempi. Secondo il sondaggio annuale degli sviluppatori C++ [29], CMake risulta essere utilizzato dall'80% di essi, dimostrandosi il build system più popolare per tale linguaggio. È interessante però notare, sempre dai risultati del sondaggio, che uno sviluppatore su tre dichiara il fatto che CMake sia un punto dolente dello sviluppo di progetti C++. La gestione di CMake appare come uno dei principali aspetti più frustranti, dopo la gestione delle librerie e dei tempi di compilazione. Per quanto riguarda gli altri build system citati, essi portano vantaggi in alcune aree ma allo stesso tempo difetti in altre. SCons, ad esempio, fornisce la comodità di scrivere i build files direttamente in Python però risulta essere estremamente lento nell'esecuzione.

7.1.5 Æther e CMake

Il progetto IBR-DTN, su cui Æther si fonda, prevede l'uso di Autotools come build system. Æther ha invece deciso di seguire un'altra strada, passando dall'uso di Autotools a CMake, al fine di rendere il processo di build dell'applicazione più moderno, semplice e facile da comprendere.

7.2 CMake

In questa sezione viene presentato più in dettaglio il build system CMake, mostrando le caratteristiche che lo hanno reso uno degli strumenti più diffusi per la gestione di progetti

software. CMake, in realtà, non è un vero e proprio build system ma un generatore di build files di altri build system. Per esempio, CMake può produrre Makefiles per piattaforme Unix oppure progetti per Microsoft Visual Studio. I file di configurazione sono chiamati `CMakeLists.txt` e contengono al loro interno comandi in un linguaggio apposito definito da CMake. Con CMake è possibile compilare il codice sorgente per creare librerie statiche, dinamiche e programmi eseguibili.

7.2.1 Caratteristiche

Tra le numerose funzionalità introdotte in CMake troviamo:

- La possibilità di generare un albero di build al di fuori dell'albero dei sorgenti (*out of source build*), cosicché sia possibile per lo sviluppatore eliminare un'intera directory di build senza temere di eliminare file sorgenti;
- La capacità di ricercare automaticamente i programmi, le librerie e gli header file richiesti dal software in creazione nei percorsi specificati nelle variabili d'ambiente o nel registro di sistema Windows;
- La possibilità di passare facilmente tra build statiche e condivise, grazie al fatto che CMake permette la creazione di entrambe su tutte le piattaforme supportate. I flag del linker specifici di alcune piattaforme sono gestiti automaticamente, così come la ricerca delle librerie condivise a tempo di esecuzione;
- La generazione automatica delle dipendenze dei file e supporto per le compilazioni parallele sulla maggior parte delle piattaforme;
- Il supporto alla cross-compilazione verso altri sistemi operativi o dispositivi embedded.

CMake è in continuo aggiornamento, monitorando attivamente gli strumenti di compilazione più popolari per poterli supportare nativamente. Una volta completato il proprio progetto CMake è possibile ottenere in modo gratuito supporto a nuovi build systems semplicemente aggiornando la versione, senza dover scrivere e mantenere un file aggiuntivo per il nuovo sistema.

7.2.2 Modern CMake

La versione 3.0 di CMake è stata rilasciata nel giugno 2014 ed è stata descritta come la nascita di CMake moderno. Le principali differenze dalla precedente versione riguardano l'uso rilevante di *target* e *properties*, paragonando i target a oggetti con costruttori, variabili membro e funzioni membro. Un documento approfondito sui requisiti e gli utilizzi di CMake moderno è disponibile al link [30].

7.2.3 Confronto con Autotools

Autotools è un build system alternativo a CMake, già esistente prima della creazione di quest'ultimo. Esso fornisce parte delle stesse funzionalità di CMake, ma l'uso di questo strumento richiede particolare conoscenza. Esso risulta particolarmente difficile da

utilizzare o da estendere, ed alcune operazioni che sono facili con CMake sono addirittura impossibili da eseguire con Autotools. Autotools supporta le opzioni specificate dall'utente, ma non supporta quelle dipendenti, ossia opzioni che dipendono da un'altra proprietà. La gestione automatica delle dipendenze non viene effettuata direttamente da Autotools e ciò non ha equivalenti rispetto all'abilità di CMake di salvare e concatenare automaticamente le dipendenze delle varie librerie utilizzate. CMake si propone quindi come miglioria del sistema Autotools, gestendo comandi semplici e leggibili in un unico file di configurazione.

7.3 Aggiornamenti in Æther

In questa sezione verranno presentate le principali modifiche apportate ai `CMakeLists.txt` presenti in Æther.

7.3.1 Librerie statiche e dinamiche

La versione iniziale di Æther prevedeva la creazione di tre librerie dinamiche e dell'eseguibile principale, basato su tali librerie:

- ibrcommon (shared library)
- ibrdtn (shared library)
- dtnd (shared library)
- dtnd-exec (eseguibile)

Oltre a questi, erano disponibili, e continuano ad esserlo, numerosi eseguibili che rappresentano i *tools* di Æther, come dtnping o dtnsend, presentati più in dettaglio nell'appendice A.3.

Nello sviluppo di questa tesi sono stati introdotti nuovi target, qui elencati e in seguito descritti:

- ibrcommon-static (static library)
- ibrdtn-static (static library)
- dtnd-static (static library)
- dtnd-exec-static (eseguibile)
- ibrcommon-tests, ibrcommon-stresstests, ibrcommon-unittests (eseguibili)
- ibrdtn-tests (eseguibile)
- daemon-tests, daemon-unittests (eseguibili)

Una delle principali aggiunte è stata la creazione delle librerie statiche dei moduli ibrcommon, ibrdtn e dtnd. Utilizzando le librerie statiche al posto di quelle dinamiche è possibile installare Æther senza doversi preoccupare della presenza delle librerie da cui dipende né della loro versione, poiché esse sono incluse nell'eseguibile. Un ulteriore vantaggio è la

produzione di un unico file eseguibile, comodo per la distribuzione e l'installazione. Ovviamente, l'uso di librerie statiche comporta un eseguibile di dimensione maggiore rispetto al collegamento dinamico, tuttavia con l'eliminazione dei simboli inutilizzati è spesso possibile ottenere ottime riduzioni di spazio. Per fare un confronto, la versione statica di Æther con tutti i simboli occupa 78,7 MB, senza i simboli non necessari si riduce a 4,6 MB (-94%).

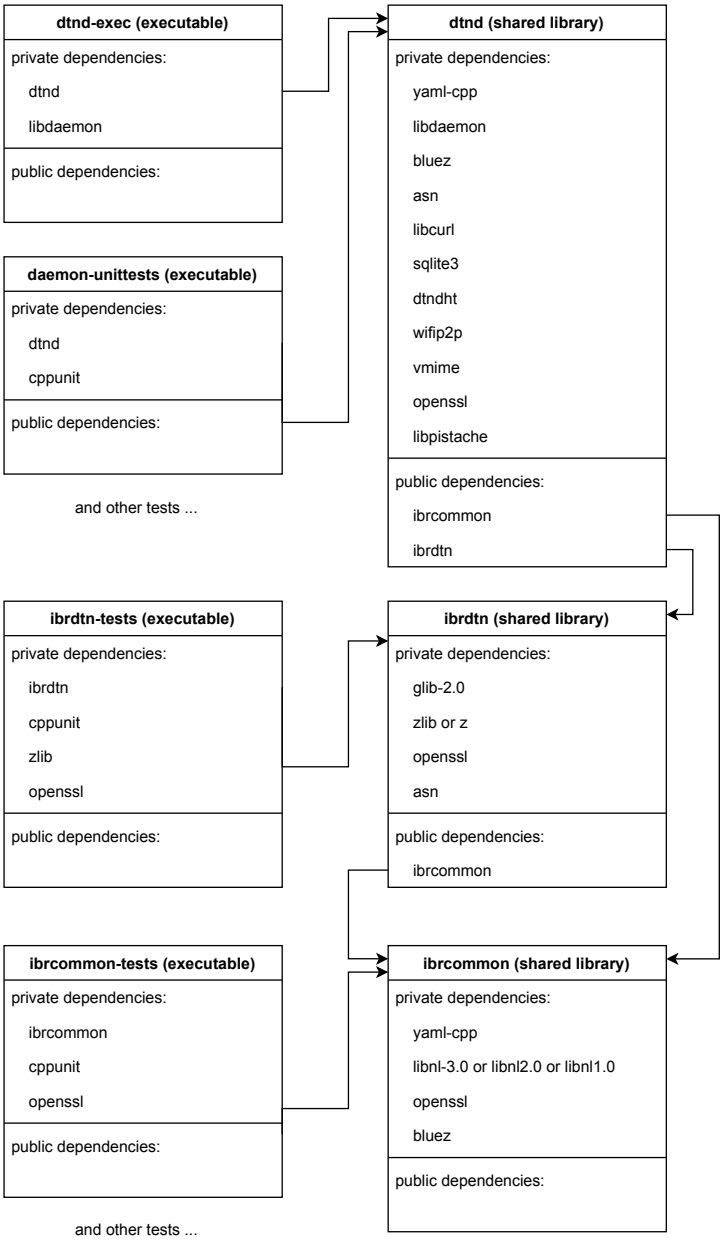


Figura 7.2. Diagramma (parziale) delle librerie ed eseguibili di Æther

7.3.2 Testing

CMake facilita il test del software attraverso speciali comandi di test e l'eseguibile CTest. Per aggiungere il testing a un progetto basato su CMake è necessario includere il modulo CTest e utilizzare il comando `add_test` per ognuno dei test previsti.

Il comando `add_test` ha la seguente sintassi:

```
add_test(NAME TestName COMMAND ExecutableToRun arg1 arg2 ...)
```

Il primo argomento, `TestName`, è semplicemente un nome per il test. Il secondo argomento rappresenta invece l'eseguibile da lanciare. L'eseguibile può essere costruito come parte del progetto o può essere un eseguibile indipendente, come Python o Perl. I restanti argomenti saranno passati all'eseguibile in esecuzione. Di default, un test viene superato se l'eseguibile è stato trovato, se il test non ha prodotto eccezioni e se il valore di ritorno è 0.

Nel caso di Æther, gli eseguibili per il lancio dei test sono stati creati in differenti progetti CMake, seguendo l'organizzazione logica precedentemente definita da IBR-DTN. In ibrcommon, per esempio, sono presenti tre differenti progetti di test:

- ibrcommon-tests, test per il controllo del corretto funzionamento di alcuni moduli;
- ibrcommon-stresstests, test che utilizzano una quantità notevole di memoria e thread;
- ibrcommon-unittests, unit test che verificano a basso livello i metodi e le funzioni di ibrcommon.

Ognuno di questi progetti produce un unico eseguibile di test che viene aggiunto alla lista di test col comando `add_test`. Questo eseguibile utilizza la libreria CppUnit per verificare il successo o il fallimento dei vari test dichiarati nel file `Main.cpp`. Il contenuto del file `CMakeLists.txt` di ibrcommon-unittests è mostrato, come esempio, nel listing 7.1.

```
cmake_minimum_required(VERSION 3.15.2 FATAL_ERROR)
project(ibrcommon-unittests LANGUAGES CXX)

# IBRCOMMON tests source files
file(GLOB TESTS_SRC ${CMAKE_CURRENT_SOURCE_DIR}/*.cpp)


# CPPUNIT library required
pkg_check_modules(CPPUNIT REQUIRED QUIET IMPORTED_TARGET cppunit)

add_executable(${PROJECT_NAME} ${TESTS_SRC})
target_include_directories(${PROJECT_NAME} PRIVATE ${CMAKE_CURRENT_SOURCE_DIR})
target_link_libraries(${PROJECT_NAME} PRIVATE PkgConfig::CPPUNIT ibrcommon)

add_test(NAME ${PROJECT_NAME} COMMAND ${PROJECT_NAME})
```

Listing 7.1. File `CMakeLists.txt` del progetto ibrcommon-unittests

Il lancio dei test è affidato all'eseguibile `ctest`, che ricava la lista dei test dal file `CTestTestfile.cmake`. Questo file viene generato automaticamente da CMake se incontra nel

file `CMakeLists.txt` una chiamata al comando `add_test`. Un progetto che include varie sottocartelle includerà nella lista di test anche quelli trovati nei progetti presenti in tali cartelle.

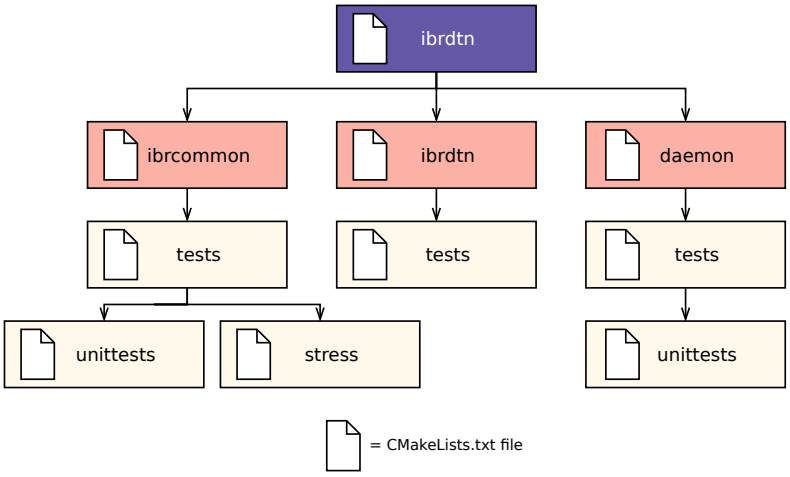


Figura 7.3. Organizzazione dei sottoprogetti CMake

7.3.3 Uso di Modern CMake

Come già accennato in 7.2.2, il documento *Effective Modern CMake* [30] prevede una serie di indicazioni su come trasformare un file CMake *old school* nella corrispettiva versione moderna. In questa sezione verranno analizzate i principali cambiamenti applicati ai `CMakeLists.txt` presenti in Æther seguendo le linee guida presentate da tale documento.

Comandi senza target Una delle regole generali è quella di non utilizzare comandi come `include_directories` o `link_libraries` ma gli equivalenti `target_include_directories` e `target_link_libraries`. Questo perché i primi operano a livello dell'intero progetto, mentre i secondi richiedono che venga specificato un singolo target del progetto a cui applicare il comando. Di conseguenza, nel caso venissero usati i comandi generici e fosse creato un nuovo target, tali proprietà sarebbero applicate anche ad esso, provocando casi di dipendenze nascoste e spesso non necessarie.

Imported targets A partire da CMake 3.4 numerosi moduli per la ricerca di dipendenze possono esportare target in modo da poterli usare nel comando `target_link_libraries`. In particolare, Æther utilizza ampiamente il comando `pkg_check_modules`, che consente di ricercare le dipendenze utilizzando l'eseguibile `pkg-config`. Con CMake 3.6, `pkg_check_modules` può prevedere l'argomento `IMPORTED_TARGET`, che produce un target virtuale che può essere passato direttamente a `target_link_libraries`. In questo modo si utilizza un unico comando per la gestione delle librerie, rendendo il codice più chiaro e uniforme.

7.3.4 Tempo di compilazione

Grazie a varie modifiche applicate ai file CMake ed a una migliore gestione dei file sorgenti e degli header file utilizzati, il tempo di compilazione per il target di default è nettamente diminuito.

Per valutare i miglioramenti, sono state effettuate tre misurazioni del tempo di compilazione con i comandi:

```
cd ibrdtn
sudo ./clean.sh
mkdir build && cd build
cmake --log-level=DEBUG -DBUILD_NOFEATURES=OFF -DBUILD_ALLFEATURES=OFF
    -DBUILD_DEBUG=OFF -DBUILD_TESTS=OFF -DBUILD_V2X=OFF -DBUILD_STATIC=OFF ..
time make -j6
```

Il risultato delle misurazioni è riportato in tabella 7.1. È facile notare come i cambiamenti introdotti nel seguente lavoro di tesi abbiano consentito di ridurre i tempi del 60%. Tra i fattori più importanti rientrano il cambiamento delle opzioni di compilazione ma soprattutto l'ottimizzazione della compilazione della libreria dinamica dtnd e dell'eseguibile dtnd-exec. Questo perché sia dtnd che dtnd-exec utilizzavano come file sorgente gli stessi file, con una doppia compilazione per i due target. Nella nuova versione, i file vengono compilati un'unica volta, creando una libreria del tipo *object library*. Una object library ha il compito di compilare i file sorgenti senza collegare i file oggetto creati in una libreria. Sarà poi compito di altri target utilizzare tali file oggetto nelle proprie librerie o eseguibili, consentendo di evitare l'operazione di compilazione.

Versione Æther	df8248dd	e6cafabc
Compilazione 1 (sec)	135,81	53,18
Compilazione 2 (sec)	147,25	53,05
Compilazione 3 (sec)	132,56	57,69
Tempo medio (sec)	138,54	54,64

Tabella 7.1. Tempi di compilazione della versione iniziale (df8248dd) e finale (e6cafabc) di Æther

7.3.5 Testing compilazione su Docker

Una volta applicate varie modifiche sia ai build file che allo script di installazione delle dipendenze, è stato necessario garantire che tali variazioni non avessero introdotto problemi nella compilazione o nell'installazione del software. Come aiuto per il raggiungimento di questo fine si è sfruttato lo *smoke testing*, un livello base del testing con lo scopo di controllare che la compilazione su altri sistemi operativi continui ad eseguire con successo. Dato che installare sistemi operativi su macchine fisiche e poi testare Æther su ognuno di

essi non è una soluzione efficiente, si è preferito utilizzare i container per semplificare ed automatizzare lo smoke testing.

Lo script creato per automatizzare la compilazione su Docker è chiamato `start-docker-aether.sh`, riportato nel listing 7.2.

```
#!/bin/bash

# Make script independent of current working directory
PARENT_PATH=$(
  cd "$(dirname "${BASH_SOURCE[0]}")" || exit 1
  pwd -P
)
cd "$PARENT_PATH" || exit 1

# File .dockerignore to reduce Docker context
cp -n .dockerignore ../

# aetherbase is an image to avoid downloading apt packages every time
docker build -t aetherbase - <DockerfileBase
docker-compose build
# docker image prune to remove automatically stage 1 image
docker image prune -f
docker-compose up
```

Listing 7.2. Script `start-docker-aether.sh`

Come primo passo il codice copia il file `.dockerignore` nella cartella principale di Æther, così da ridurre il contesto di Docker attraverso l'esclusione di file e cartelle non utili per la creazione di immagini. Successivamente, lo script produce l'immagine `aetherbase`, che rappresenta un passaggio intermedio con tutti gli strumenti necessari per la compilazione, ed infine crea e lancia due istanze di Æther su due container differenti sfruttando lo strumento docker-compose.

```
# syntax=docker/dockerfile:1

##################
# STAGE 0 BUILD #
##################

FROM ubuntu:20.04
ENV DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install -y \
gcc g++ make cmake openssl wget git pkg-config \
autoconf automake software-properties-common build-essential \
zlib1g-dev libsqlite3-dev libcurl4-gnutls-dev libdaemon-dev libtool libcppunit-dev \
libnl-3-dev libnl-cli-3-dev libnl-genl-3-dev libnl-nf-3-dev libnl-route-3-dev \
libarchive-dev libdbus-1-dev libglib2.0-dev libudev-dev libical-dev libreadline-dev \
libbluetooth-dev libssl-dev libyaml-cpp-dev \
doxygen graphviz bash-completion locate subversion asn1c curl unzip meson
```

Listing 7.3. File `DockerfileBase`

```
# syntax=docker/dockerfile:1

##################
# STAGE 1 BUILD #
##################
FROM aetherbase as build

COPY . /home/Aether
ENV DEBIAN_FRONTEND=noninteractive

WORKDIR /home/Aether
RUN ./script_installers/aether_environment.sh -d
WORKDIR aether/ibrdtn/ibrdtn
RUN mkdir build
WORKDIR build
RUN cmake ..
RUN make -j$(nproc)
RUN make install
RUN ldconfig


##################
# STAGE 2 BUILD #
##################
FROM ubuntu:20.04

WORKDIR /home
COPY --from=build /usr/local /usr/local

RUN apt-get update && apt-get install -y \
libyaml-cpp-dev \
libbluetooth-dev \
libdaemon-dev \
libnl-3-dev libnl-cli-3-dev libnl-genl-3-dev libnl-nf-3-dev libnl-route-3-dev \
&& rm -rf /var/lib/apt/lists/*

# No default command set
```

Listing 7.4. File `Dockerfile`

```
services:
  aeth_1:
    image: aether
    build:
      context: ..
      dockerfile: docker-aether/Dockerfile
    networks:
      - aether-network
    volumes:
      - "./conf:/home"
    container_name: aether_1
    hostname: aether_1
    command: dtnd -c /home/dtnd.yaml -d 25
```

```
  aeth_2:
    image: aether
    build:
      context: ..
      dockerfile: docker-aether/Dockerfile
    networks:
      - aether-network
    volumes:
      - "./conf:/home"
    container_name: aether_2
    hostname: aether_2
    command: dtnd -c /home/dtnd.yaml -d 25

networks:
  aether-network:
    name: aether-network
```

Listing 7.5. File `docker-compose.yml`

Trattandosi di una multi-stage build, tutte le immagini intermedie possono essere eliminate per risparmiare spazio. In questo modo, considerando un'immagine basata su ubuntu 20.04, si riesce a passare dall'immagine completa prodotta dallo stage 1 di dimensione 985 MB a un'immagine finale ridotta a 109 MB.

Capitolo 8

Conclusioni

Il risultato finale ottenuto dal seguente lavoro di tesi ha portato alla correzione e all'aggiornamento del software Æther, introducendo un ampio numero di modifiche che possano permettere eventuali futuri sviluppi in maniera più rapida e intuitiva. Dal punto di vista della correttezza del codice, tutti i warning principali segnalati dal compilatore sono stati corretti, e grazie all'utilizzo dei test inclusi nel progetto è stato possibile individuare e rettificare anche alcuni bug. L'aggiornamento del codice relativo alla connettività Bluetooth ha ripristinato il funzionamento del Bluetooth Convergence Layer, permettendo di continuare a sfruttare questa tecnologia che al giorno d'oggi risulta presente sulla gran parte dei dispositivi informatici.

Il processo di aggiornamento dei build file CMake ha consentito di ridurre notevolmente i tempi di compilazione, uno step molto utile per rendere lo sviluppo software più veloce e per poter testare le proprie modifiche in tempi più rapidi. La creazione dei target di test potrà permettere la facile aggiunta di ulteriori test, per raggiungere percentuali migliori di code coverage e per validare la correttezza formale del software. L'esteso lavoro di refactoring, compresa la scrittura di documentazione, darà modo di ridurre i tempi di sviluppo, facilitando la comprensione dell'organizzazione del progetto e semplificando il lavoro di modifica al codice.

Tutte queste importanti modifiche hanno fatto sì che il debito tecnico del progetto sia diminuito in maniera netta, riducendo la presenza di code smell e design smell con la conseguente garanzia di codice più sicuro e con meno possibilità di introdurre comportamenti errati. La maggior parte del lavoro non ha intaccato la qualità esterna di Æther, che è rimasta pressoché identica, tuttavia la qualità interna è sicuramente migliorata.

Lo scopo finale della tesi, ovvero l'ottenimento di un prodotto quanto più pronto per l'uso nel mondo reale, è stato raggiunto. A questo va affiancato l'espansione dei test e l'aggiunta di simulazioni estensive sul campo, che possono aiutare a confermare il corretto funzionamento del software al di fuori del mondo simulativo. Questa necessità può essere considerato uno sviluppo futuro, con il fine di trasformare Æther da progetto sperimentale a software aziendale a tutti gli effetti.

Appendice A

Guida all'utilizzo di Æther

La seguente appendice rappresenta una guida per l'installazione, la configurazione e l'uso di Æther. Nella seconda parte sono presentati alcuni strumenti per scambiare messaggi tra diversi dispositivi con Æther in esecuzione.

A.1 Installazione

Per poter procedere all'installazione di Æther è necessario prima installare le dipendenze e poi procedere con la compilazione sul proprio sistema Linux.

A.1.1 Installazione dipendenze

Æther fa uso di un numero considerevole di dipendenze, tuttavia buona parte di esse sono necessarie solo per la compilazione del software, e un'altra parte è utilizzata solamente in alcuni moduli opzionali. Per evitare che l'utente debba installare tutte le librerie a mano, Æther utilizza uno script shell chiamato `aether_environment.sh` che gestisce automaticamente le dipendenze necessarie.

Lo script `aether_environment.sh` è contenuto nella cartella `script_installers`, insieme al file `environment_function.sh` che contiene funzioni che `aether_environment.sh` utilizza.

```
# Check and install dependencies
# Parameters:
# $1 Dependency name
# $2 Exit status
# $3 Action to install dependency if exit status != 0
function checkDependency() {
  if [[ $2 -eq 0 ]]; then
    echo -e "$1 OK"
  else
    $3
  fi
```

```
}

echo -e "This script will create the environment for Aether"

echo -e "Installing mandatory libraries"
installViaAPT

rm -rf bootstrapping/
mkdir bootstrapping/
cd bootstrapping/ || exit

echo -e "Checking version of gcc"
gcc --version | awk '/gcc/ && ($3+0)<7.0{exit 1}'
checkDependency gcc $? installGCC

echo -e "Checking version of OpenSSL"

pkg-config --atleast-version=1.1.1 openssl
checkDependency OpenSSL $? installOpenSSL

echo -e "Checking for CMake"
createCMakeLists
cmake . &>/dev/null
checkDependency CMake $? installCMake

if [[ -z "$DOCKER_ENV" ]]; then
echo -e "Checking for wifip2p"
pkg-config --exists wifip2p
checkDependency wifip2p $? installWifiP2P

echo -e "Checking for pistache"
pkg-config --exists libpistache
checkDependency pistache $? installPistache

echo -e "Checking for dtndht"
pkg-config --exists dtndht
checkDependency dtndht $? installDTNDHT

echo -e "Creating TFFS"
installTFFS

echo -e "Checking for V2x needed modules"
pkg-config --exists asn
checkDependency asn $? installV2x
fi
```

Listing A.1. Estratto di aether_environment.sh

```
function installViaAPT {
  apt-get update
  apt-get install "${AUTO_YES}" \
  gcc g++ make cmake openssl wget git pkg-config \
  autoconf automake software-properties-common build-essential \
```

```
  zlib1g-dev libsqlite3-dev libcurl4-gnutls-dev libdaemon-dev libtool \
  libcppunit-dev libarchive-dev \
  libnl-3-dev libnl-cli-3-dev libnl-genl-3-dev libnl-nf-3-dev libnl-route-3-dev \
  libdbus-1-dev libglib2.0-dev libudev-dev libical-dev libreadline-dev \
  libbluetooth-dev libssl-dev libyaml-cpp-dev \
  doxygen graphviz bash-completion locate subversion asn1c curl unzip meson

  ldconfig
  logInstalled "APT libraries"
}

function installCMake {
  function upgradeCMake {
    apt-get -y remove cmake
    if [ ! -e cmake-3.16.0.tar.gz ]; then
    wget https://github.com/Kitware/CMake/releases/download/v3.16.0/cmake-3.16.0.tar.
        gz
    fi

    echo -e "Extracting CMake"

    tar -zxf cmake-3.16.0.tar.gz

    cd cmake-3.16.0
    echo -e "Bootstrapping CMake please wait"

    ./bootstrap
    make -j "$NPROC"
    make install

    logInstalled "CMake"
    cd ..
  }

  MSG="CMake is lower than version 3.15.2, Aether will not compile."
  installDependency "CMake" "$MSG" "Upgrade CMake" "Quit and do manually"
      upgradeCMake 0
}
```

Listing A.2. Estratto di environment_function.sh

Una volta eseguito lo script `aether_environment.sh` con successo, saremo pronti per compilare Æther.

A.1.2 Compilazione

Per compilare Æther è necessario clonare la repository di Tierra e proseguire con i comandi elencati nel listing A.3. È bene precisare che Æther consente un'ampia personalizzazione del software, con moduli attivabili o meno a seconda che vengano specificate varie opzioni durante il lancio di CMake. La lista di opzioni sono riportate nel file `README.md` così come nel file `CMakeLists.txt` principale. Per esempio, avviando il processo di build col comando `cmake -DBUILD_DEBUG=ON ..` sarà abilitata la compilazione di Æther in modalità debug,

che significa nessuna ottimizzazione da parte del compilatore e l'inserimento di simboli di debug nel codice macchina.

```
cd aether/ibrdtn/ibrdtn
mkdir build && cd build
cmake ..
make -j$(nproc)
sudo make install
sudo ldconfig
```

Listing A.3. Comandi bash di compilazione

A.2 Avvio

Per lanciare Æther è sufficiente scrivere in console `dtnd`, che appunto avvia l'eseguibile con lo stesso nome. Due importanti parametri possono essere impostati:

-c Specifica il percorso del file di configurazione, nel formato YAML già descritto nella sezione 6.2.2;

-d Specifica il livello di informazioni di debug da 1 a 99, dove più alto è il livello e più informazioni di debug verranno stampate.

Il comando `-h` apre l'help, che elenca le ulteriori impostazioni disponibili.

A.3 Tools

La suite IBR-DTN include vari strumenti, con lo scopo di poter interfacciarsi con il demone Æther e di verificare il corretto funzionamento della DTN. Tra gli strumenti più importanti troviamo:

dtnping Permette di inviare un bundle ad una specifica destinazione, in attesa della risposta contenente lo stesso bundle, secondo la normale procedura di ping.

dtnsend Consente di inviare dati ad un altro dispositivo con `dtnrecv` in esecuzione.

dtntracepath Mostra l'elenco di nodi coinvolti nella trasmissione di un bundle fino al raggiungimento della destinazione.

Tutti i tool disponibili sono presenti nella cartella `tools`.

A.4 Disinstallazione

Per disinstallare Æther è sufficiente utilizzare lo script `clean.sh`, che ha il compito di rimuovere tutti i file e le librerie inserite nel sistema con l'operazione `make install` (o equivalente). Il codice di `clean.sh`, opportunamente aggiornato, è presentato nel listing A.4.

```
#!/bin/bash

if [ ! -e "ibrcommon" ]; then
ln -s ../ibrcommon ibrcommon
fi

INSTALL_FILES="/usr/local/lib/libibrcommon.so*
/usr/local/lib/libibrdtn.so*
/usr/local/lib/libdtnd.so*
/usr/local/lib/libibrcommon-static.a
/usr/local/lib/libibrdtn-static.a
/usr/local/lib/libdtnd-static.a
/usr/local/include/ibrcommon
/usr/local/include/dtnd
/usr/local/include/ibrdtn
/usr/local/lib/pkgconfig/ibrcommon.pc
/usr/local/lib/pkgconfig/ibrdtn.pc
/usr/local/sbin/dtnd
/usr/local/sbin/dtnd-static
/usr/local/bin/dtn*
/usr/local/bin/ibrdtn*
/usr/local/share/dtnd
/usr/local/share/doc/dtnd
/usr/local/etc/json_ibrdtn"


echo "Current directory is $PWD"
echo "This script will delete all subdirectories named \"build\", \"cmake-build-*\"
    or \"CMakeFiles\"."
echo "Do you wish to continue?"
select yn in "Yes" "No"; do
case $yn in
Yes) break ;;
No) exit ;;
esac
done

# If set, bash allows patterns which match no files to expand to a null string,
    rather than themselves.
shopt -s nullglob
echo

# Delete build folders
find -L . -depth \( -name "build" -o -name "cmake-build-*" \) -type d \
-exec echo -e "## REMOVING DIRECTORY\t{}" \; \
-exec rm -rf "{}" \;

find -L . -depth -name "CMakeFiles" -type d \
-exec echo -e "## REMOVING DIRECTORY\t{}" \; \
-exec rm -rf "{}" \;

# Delete installed files
for FILE in $INSTALL_FILES; do
if [[ -f $FILE ]]; then
```

```
echo -e "## REMOVING FILE\t${FILE}"
rm "${FILE:?}"
elif [[ -d $FILE ]]; then
echo -e "## REMOVING DIRECTORY\t${FILE}"
rm -rf "${FILE:?}"
fi
done
```

Listing A.4. `clean.sh`

Bibliografia

[1] Forrest Warthman. *Delay- and Disruption-Tolerant Networks (DTNs). A Tutorial*. Ver. 3.2. 14 Set. 2015. URL: https://www.warthman.com/images/IRTF%20DTN_Tutorial_v3.2.pdf.

[2] Keith Scott e Scott C. Burleigh. *Bundle Protocol Specification*. RFC 5050. Nov. 2007. DOI: 10.17487/RFC5050. URL: https://www.rfc-editor.org/info/rfc5050.

[3] Stephen Farrell et al. *Bundle Security Protocol Specification*. RFC 6257. Mag. 2011. DOI: 10.17487/RFC6257. URL: https://www.rfc-editor.org/info/rfc6257.

[4] Wesley Eddy e Elwyn B. Davies. *Using Self-Delimiting Numeric Values in Protocols*. RFC 6256. Mag. 2011. DOI: 10.17487/RFC6256. URL: https://www.rfc-editor.org/info/rfc6256.

[5] Scott Burleigh, Kevin Fall e Edward J. Birrane. *Bundle Protocol Version 7*. RFC 9171. Gen. 2022. DOI: 10.17487/RFC9171. URL: https://www.rfc-editor.org/info/rfc9171.

[6] Sebastian Schildt et al. «IBR-DTN: A lightweight, modular and highly portable Bundle Protocol implementation». In: *Electronic Communications of the EASST* 37 (gen. 2011), pp. 1–11. URL: https://www.ibr.cs.tu-bs.de/papers/schildt-ibrdtn.pdf.

[7] Johannes Morgenroth. *IBR-DTN repository*. URL: https://github.com/ibrdtn/ibrdtn.

[8] Johannes Morgenroth. «Event-driven Software-Architecture for Delay- and Disruption-Tolerant Networking». Tesi di dott. Lug. 2015. DOI: 10.24355/dbbs.084-201509251022-0. URL: http://www.digibib.tu-bs.de/?docid=00061364.

[9] Carlo Pettinato. «Prototipazione di una architetture di tipo Delay -Tolerant Networks operante su tecnologie di livello fisico eterogenee». Tesi di laurea. Politecnico di Torino, 2018. URL: https://webthesis.biblio.polito.it/9509/.

[10] Giovanni Pironti. «Integrazione dei protocolli V2X nella tecnologia di rete tollerante ai ritardi (IBR)». Tesi di laurea. Politecnico di Torino, 2020. URL: https://webthesis.biblio.polito.it/15986/.

[11] Antonio Nunnari. «Introduzione di primitive service-oriented su reti ad-hoc in ambienti sfidanti». Tesi di laurea. Politecnico di Torino, 2019. URL: https://webthesis.biblio.polito.it/12428/.

[12] Matteo Ottolini. «Fornitura di connettività ottimizzata in un'infrastruttura DTN Service-Oriented». Tesi di laurea. Politecnico di Torino, 2020. URL: https://webthesis.biblio.polito.it/14513/.

[13] Kevin Townsend et al. *Getting Started with Bluetooth Low Energy*. O'Reilly Media, mag. 2014. ISBN: 9781491949511. URL: https://www.oreilly.com/library/view/getting-started-with/9781491900550/.

[14] Bluetooth SIG. *Bluetooth Core Specification*. Ver. 5.3. 13 Lug. 2021. URL: https://www.bluetooth.com/specifications/specs/core-specification-5-3/.

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

[16] John Padgette et al. *Guide to Bluetooth Security*. NIST Special Publication 800-121 Revision 2. Mag. 2017. DOI: 10.6028/NIST.SP.800-121r2-upd1. URL: https://csrc.nist.gov/publications/detail/sp/800-121/rev-2/final.

[17] Martin Woolley. *The Bluetooth Low Energy Primer*. Ver. 1.0.4. Bluetooth SIG. 6 Giu. 2022. URL: https://www.bluetooth.com/blog/introducing-the-bluetooth-low-energy-primer/.

[18] Bluetooth SIG. *Assigned Numbers*. 4 Nov. 2022. URL: https://www.bluetooth.com/specifications/assigned-numbers.

[19] Bluetooth SIG. *Supplement to the Bluetooth Core Specification*. Ver. 10. 13 Lug. 2021. URL: https://www.bluetooth.com/specifications/specs/core-specification-supplement-10/.

[20] Martin Woolley. *Bluetooth Technology for Linux Developers*. Bluetooth SIG. 16 Nov. 2021. URL: https://www.bluetooth.com/bluetooth-resources/bluetooth-for-linux/.

[21] Silicon Labs. *SPP (Serial Port Profile) over BLE*. URL: https://docs.silabs.com/bluetooth/2.13/code-examples/applications/spp-serial-port-profile-over-ble.

[22] Da-Zhi Sun, Yi Mu e Willy Susilo. «Man-in-the-Middle Attacks on Secure Simple Pairing in Bluetooth Standard V5.0 and Its Countermeasure». In: 22.1 (feb. 2018), pp. 55–67. ISSN: 1617-4909. DOI: 10.1007/s00779-017-1081-6. URL: https://doi.org/10.1007/s00779-017-1081-6.

[23] M.M. Lehman. «Programs, life cycles, and laws of software evolution». In: *Proceedings of the IEEE* 68.9 (1980), pp. 1060–1076. DOI: 10.1109/PROC.1980.11805.

[24] Capers Jones. *Quantifying Software: Global and Industry Perspectives*. Ott. 2017, pp. 1–533. ISBN: 9781315314426. DOI: 10.1201/9781315314426.

[25] Girish Suryanarayana, Ganesh Samarthyam e Tushar Sharma. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, nov. 2014, pp. 1–237. ISBN: 9780128013977.

[26] Bjarne Stroustrup e Herb Sutter. *C++ Core Guidelines*. 6 Ott. 2022. URL: https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md.

[27] SEI CERT. *SEI CERT C++ Coding Standard*. URL: https://wiki.sei.cmu.edu/confluence/pages/viewpage.action?pageId=88046682.

[28] Martin Bond. *Why We Need Build Systems*. 4 Giu. 2021. URL: https://blog.feabhas.com/2021/06/why-we-need-build-systems/.

[29] Standard C++ Foundation. *2022 Annual C++ Developer Survey "Lite"*. 7 Giu. 2022. URL: https://isocpp.org/blog/2022/06/results-summary-2022-annual-cpp-developer-survey-lite.

[30] Manuel Binna. *Effective Modern CMake*. 8 Dic. 2017. URL: https://gist.github.com/mbinna/c61dbb39bca0e4fb7d1f73b0d66a4fd1.