POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Integrazione del protocollo V2X nella tecnologia di rete tollerante ai ritardi (IBR-DTN)

Relatore
prof. Fulvio Risso

Correlatore:
dott. Gabriele Castellano

Candidato
Giovanni Pironti
matricola: s255257

Coordinatore aziendale
Tierra S.p.a.
dott. Riccardo Loti

Ottobre 2020

Ringraziamenti

Vorrei cominciare ringraziando il Professor Fulvio Risso, che mi ha permesso di partecipare a questo progetto, che ha migliorato le mie conoscenze sotto tutti i punti i vista e Gabriele Castellano che mi ha aiutato sulla validazione dell'elaborato. Un ringraziamento speciale va a Calogero Carrabbotta che ogni giorno mi ha aiutato a portare avanti questo progetto,nonostante i mille problemi incontrati. Un ringraziamento anche a Riccardo Loti che mi ha permesso di entrare nel mondo di Tierra. Il ringraziamento più importante va alla mia famiglia, ma soprattutto a mia madre e mio padre che hanno sempre creduto in me e permesso con il loro sforzo il raggiungimento di questo importante obbiettivo. Infine un dovuto ringraziamento agli amici e colleghi, Marco, Francesco, Benedetto, e Antonio che hanno condiviso questo lungo percorso con me e senza i quali questo traguardo non avrebbe lo stesso valore.

Indice

Capitolo 1

Introduzione

La sempre crescente diffusione di dispositivi IoT, in concomitanza con la rapida evoluzione delle tecnologie di comunicazione wireless, introduce 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 da risorse hardware limitate (Industrial IoT). In questi contesti i dispositivi coinvolti possono essere flotte di mezzi, macchinari, o persino sensori per l'agricoltura di precisione, i quali non sempre possono fare affidamento su connessioni WAN stabili (ad esempio tramite rete 3G) e sui paradigmi di comunicazione tradizionali. Questi scenari vengono identificati con il nome di "challenged networks".

Sulla rete internet, le comunicazioni 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. Inoltre, i collegamenti tra due dispositivi di rete tendono ad essere stabili. Questo non può però essere garantito nel contesto di una challenged network, in cui i dispositivi hanno a disposizione soltanto canali di comunicazione wireless a corto raggio (e.g., WiFi, Bluetooth), caratterizzati da continue interruzioni o addirittura assenza di connettività per periodi indefiniti. Ciò non permette di assicurare percorsi stabili tra sorgente e destinazione.

Oltre che alla mobilità dei dispositivi, la connettività intermittente può derivare da fattori intrinseci dell'ambiente in cui la challenged network è collocata, come la presenza di ostacoli che si interpongono tra i dispositivi, 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, alto tasso di perdita di informazioni, i quali rendono la comunicazione ancora più complessa. Gli scenari soggetti a tali problematiche sono molteplici e spaziano dall'ambito militare, all'interplanetario, alle reti di sensori nelle aree non dotate di alcuna infrastruttura di telecomunicazione, come ad esempio siti edili, agricoli o per l'allevamento del bestiame, in zone montuose o rurali.

La maggior parte delle applicazioni IoT esistenti suppongono di operare su reti connesse, caratterizzate da ritardi minimi o perlomeno trascurabili. Modificare tali

applicazioni per consentirne l'operabilità su reti sfidanti richiederebbe uno sforzo notevole. Un approccio alternativo più praticabile è quello di introdurre un framework di rete comune a tutti i nodi della rete, che, agendo come una sorta di interfaccia, permetta di sopperire in maniera trasparente a tutte le esigenze tipiche delle challenged networks. A tal proposito, il paradigma del Delay-Tolerant Networking (DTN) rappresenta una potenziale soluzione al problema. Il suo obiettivo principale è quello di consentire il trasporto di un pacchetto da sorgente a destinazione nonostante la temporanea 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. Attraverso quest'approccio, un messaggio viene mantenuto localmente fin quando non risulta possibile consegnarlo direttamente a destinazione, oppure inoltrarlo a qualche altro nodo intermedio, ritenuto un potenziale next-hop verso la destinazione. Una DTN è quindi una rete formata da dispositivi fissi e mobili quali sensori e attuatori a bordo di veicoli, terminali utente, unità computazionali ai confini della rete, ecc., dotati di una o più interfacce di rete in grado di comunicare tra di loro tramite connessioni opportunistiche, senza fare affidamento su un'infrastruttura di rete fissa. Questo paradigma fornisce una base per consentire alle varie applicazioni sui dispositivi mobili di scambiarsi informazioni e quindi offrire e consumare servizi, tenendo eventualmente conto dei possibili ritardi di comunicazione.

L'obiettivo di questa tesi è quello di estendere il framework IBR-DTN (una implementazione opensource del paradigma DTN) aggiungendo il supporto per le comunicazioni tramite protocollo V2X (Vehicol to Everything). Dopo aver descritto il framework IBR-DTN (Capitolo 2, viene dettagliato lo standard V2X e vengono analizzate le implementazioni disponibili (Capitoli 3). A partire da questi studi preliminari, questo lavoro individua la strategia più congeniale all'integrazione del protocollo V2X su IBR-DTN, descrivendo le scelte architetturali e gli adattamenti apportati per adattare lo standard V2X all'architettura software del framework (Capitolo 4). Vengono infine analizzati i risultati sperimentali (Capitolo 5) e discusse le possibili direzioni per ulteriori miglioramenti e sviluppi futuri (Capitolo 6).

Capitolo 2

IBR-DTN

2.1 Delay/Disruption Tolerant Network

Le *Delay/Disruption Tolerant Network* (DTN) definiscono un'architettura end-to-end capace di fornire connettività nelle cosiddette "challenged networks". Queste reti sono caratterizzate da connettività intermittente, nodi di tipologia eterogenae condizioni di rete molto diverse. Il concetto di Delay Tolerant Networking nasce nell'ambito delle comunicazioni interplanetarie, ma attualmente trova moltissime applicazioni in ambito commerciale, scientifico, militare e di servizio pubblico. I protocolli Internet tradizionali non riescono a fornire comunicazione efficientemente,in quanto le assunzioni sulla quale sono basati non sono valide per questa particolare tipologia di reti. Oggi giorno, infatti, è sempre più comune scontrarsi con scenari applicativi in cui i dispositivi che devono comunicare sono in movimento e operano a potenza limitata, questo può portare all'interruzione di un collegamento per la presenza di un ostacolo, oppure, in certe situazioni l'interruzione del link fisico al fine di preservare energia. La conseguenza di questi fenomeni di connettività intermittente, è un naturale partizionamento della rete.

In tali scenari, la comunicazione mediante i protocolli basati su IP è particolarmente inefficiente. Il protocollo IP, si basa sull'idea che in ogni istante esista un percorso end-to-end che colleghi sorgente e destinazione di un pacchetto. Questo non è assolutamente ipotizzabile in una "challenged network", che è invece caratterizzata da connettività intermittente, ritardi lunghi e/o variabili, alto tasso di errori e asimmetria nelle trasmissioni. Basta pensare a TCP/IP, il suo utilizzo per comunicare all'interno di una rete instabile causerebbe un numero significativo di dati persi. Infatti, nel caso di un pacchetto che non possa essere inoltrato immediatamente, il TCP assumerà il congestionamento della rete, scarterà il pacchetto e proverà a ritrasmetterlo abbassando gradualmente la velocità di ritrasmissione,fino a chiudere la sessione nel caso di intermittenza troppo elevata.

Inoltre, parlando di connettività intermittente, è opportuno far distinzione tra i contatti pianificati e quelli opportunistici (2.1). Lo scenario tipico dei contatti

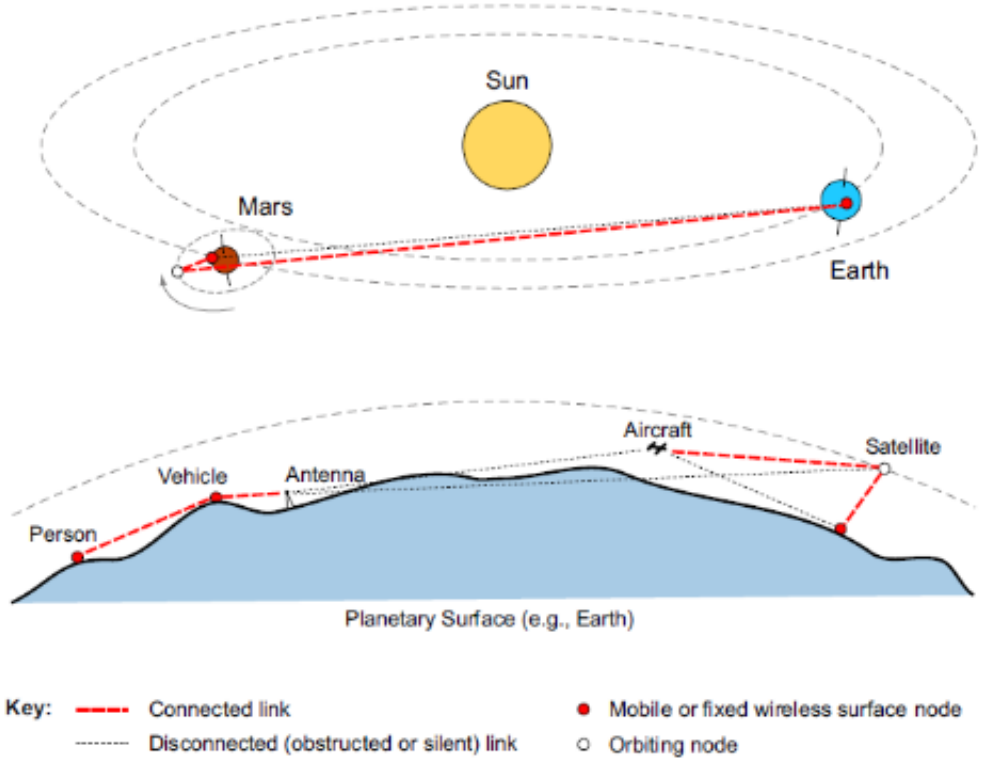

Figura 2.1: Esempi di contatti pianificati (comunicazioni interplanetarie) e opportunistici (comunicazioni sulla superficie terrestre).

pianificati o *scheduled* è quello dello spazio, in cui i nodi si muovono su percorsi orbitali predicibili, tanto che è possibile prevedere o ricevere gli instanti in cui occuperanno le loro future posizioni e quindi organizzare le future sessioni di comunicazione. I contatti di tipo pianificato, perciò, richiedono la sincronizzazione temporale dell'intera DTN. Per contatti opportunistici, invece, si intendono i contatti tra un trasmettitore e un ricevitore in istanti non programmati. E' il caso di persone, veicoli, aerei o satelliti che potrebbero voler scambiare informazioni quando risultano abbastanza vicini da poter comunicare usando la loro potenza, seppure limitata.

Per far fronte alle problematiche tipiche delle "challenged networks" e trarre beneficio dai contatti pianificati e/o opportunistici, le DTN utilizzano la tecnica dello *store-and-forward message switching*. Secondo questo paradigma, analogo 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 la persistenza dell'informazione. Questo è in contrapposizione rispetto a quanto accade nei router IP, che utilizzano dei buffer di memoria per accodare i pacchetti in attesa di essere inoltrati, garantendone una persistenza dell'ordine dei millisecondi. E' 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.

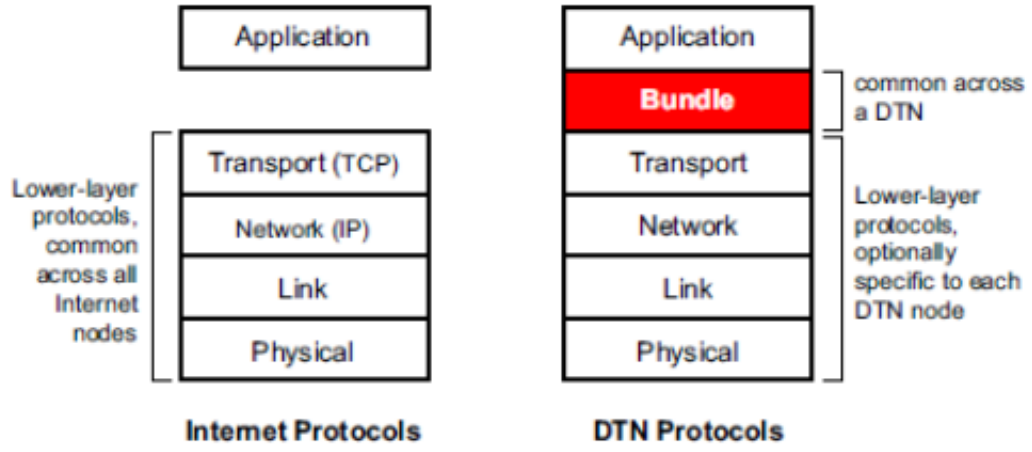

Figura 2.2: Confronto fra uno stack Internet (a sinistra) e uno stack DTN (a destra).

La DTN realizza una rete "overlay" introducendo 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. L'obiettivo principale di questo layer è quello di rendere i programmi applicativi agnostici rispetto ai livelli di trasporto utilizzati, favorendo la creazione di reti eterogenee. Due nodi che vogliono instaurare una comunicazione interagiranno con il Bundle Layer, senza preoccuparsi della natura dei protocolli utilizzati nei livelli inferiori. Il bundle layer sarà responsabile dell'instradamento di questi messaggi, detti appunto Bundle, da sorgente a destinazione. Le DTN utilizzano un modello non conversazionale asincrono, in contrasto al meccanismo di comunicazione richiesta/risposta tipico della famiglia TCP/IP. I protocolli conversazionali, come il TCP, implicano lunghi RTT e spesso falliscono. Il Bundle Layer comunica tramite un protocollo non conversazionale che minimizza i round trips necessari a confermare le trasmissioni, rendendo opzionali gli acknowledgment.

La peculiarità di posizionarsi fra il layer di trasporto e il layer applicativo permette l'uso di DTN per creare dei proxy applicativi.

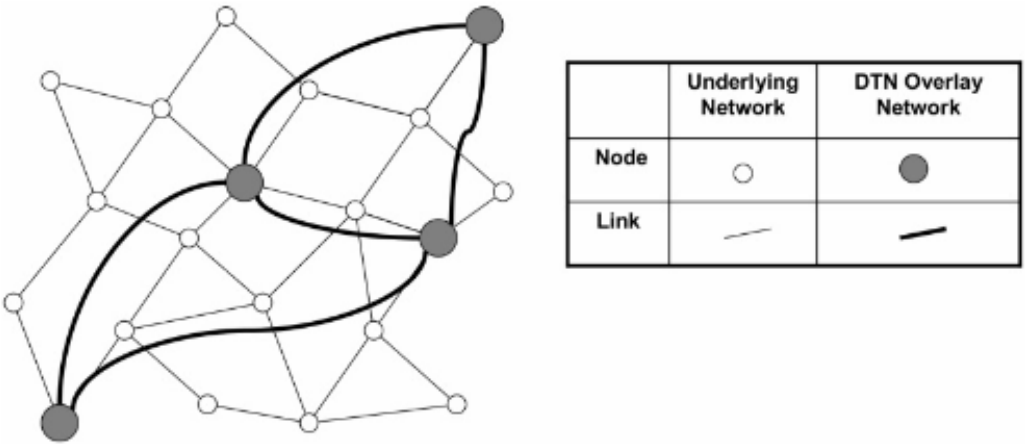

Figura 2.3: La rete DTN in overlay su un altro tipo di rete.

Prendiamo come esempio applicazioni che girano su TCP/IP, esse tipicamente usano le API socket Berkeley,e non hanno accesso ai servizi di DTN. Inoltre se volessero usarli dovrebbero essere scritte in maniera da essere tolleranti ad interruzioni e ritardi, e potrebbero avere bisogno di numerosi scambi di messaggi per effettuare le proprie operazioni, come SMTP. Riscrivere le applicazioni per sfruttare le API, richiederebbe modifiche a tutte le applicazioni. L'altro uso che possiamo ipotizzare di DTN è quello di creare un Application Layer Gateway. Esso sarebbe un terminatore di protocollo, e prenderebbe le informazioni necessarie per ricreare lo stesso dialogo avuto con il client,così da riproporlo al server e ottenere la risposta desiderata.[1]

2.2 Bundle Protocol

Il *Bundle Protocol*[2] è un protocollo sperimentale, corrispondente al BundleLayer dell'architettura DTN, sviluppato all'interno del Delay Tolerant Networking Research Group (DTNRG) dell'IRTF.

2.2.1 Architettura

Nel contesto delle DTN, con il termine *bundle node* si indica un'entità capace di ricevere e trasmettere bundle. Secondo le specifiche del Bundle Protocol, un *bundle node* è concettualmente costituito da tre componenti fondamentali:

- **Bundle Protocol Agent (BPA)**: è il fornitore dei servizi del bundle protocol. Il modo con cui tali servizi sono offerti dipende dalla sua implementazione. Infatti, il BPA può essere implementato in hardware, come libreria condivisa tra più nodi su una singola macchina, come un processo(un demone)con cui i nodi su una o più macchine possono interagire tramite meccanismi di comunicazione tra processi o comunicazione di rete (Es. Socket Berkeley).

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

I principali servizi che un BPA dovrebbe fornire all'AA di un nodo sono iseguenti:

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

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.

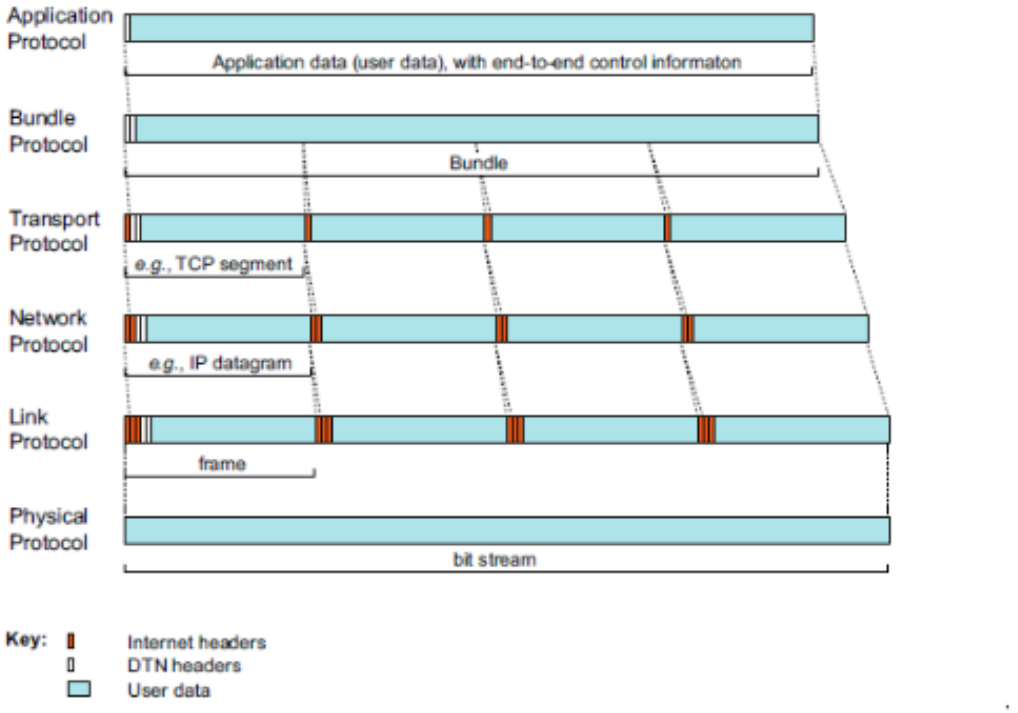

Figura 2.4: Incapsulamento dei protocolli TCP/IP nel Bundle Protocol.

2.2.3 Frammentazione

Per assicurarsi che i volumi di contatto siano usati pienamente e per evitare la ritrasmissione di Bundle parzialmente inoltrati, DTN offre un meccanismo di frammentazione. Due tipi di frammentazione sono previsti da DTN: proattiva e reattiva.La frammentazione *proattiva* avviene su scelta arbitraria da parte di un nodo inoltrante il Bundle, sarà poi compito del nodo, o nodi, destinatari il riassemblaggio dei frammenti. La frammentazione *reattiva* avviene invece a seguito del non completo trasferimento di un bundle verso un nodo. Il nodo ricevente deciderà di trattare la porzione ricevuta come se fosse un frammento, e il mittente di inviare la parte rimanente come se fosse un secondo frammento, direttamente al ricevente o passando da altri nodi se dovesse cambiare la topologia. Solo la frammentazione *proattiva* è di obbligatoria implementazione. La frammentazione a livello di Bundle Protocol è supportata grazie all'uso di un header che indica la lunghezza e l'offset del frammento rispetto al bundle originario, secondo un meccanismo simile a quello utilizzato in IP. I frammenti originati a partire dallo stesso bundle saranno identificati da sorgente, destinazione e tempo di creazione. Per un Bundle è inoltre

possibile richiedere la non frammentazione tramite uno dei Control Flag del primary Block. Inoltre tutti i blocchi prima del payload sono inseriti nel frammento di offset minore, e quelli dopo il blocco di payload sono inserite nel frammento di offset maggiore.

2.2.4 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 convenzionalmente sono usati schemi conformi allo schema URI (Unified Resource Identifier), e caratterizzati da uno <scheme-specific part> suddiviso in due porzioni: la prima indicante il nodo, la seconda il *demux-token*, ovvero una singola applicazione. Uno degli schemi più diffusi è quello identificato dalla stringa **dtn**, che assume la forma **dtn://node/demux-token**. Mentre la presenza del **node** è obbligatoria, il **demux-token** può anche non esserci,come nel caso di bundle amministrativi diretti al BPA del nodo. Un EID tipicamente rappresenta un solo nodo (o meglio un applicazione su un solo nodo) ed è detto Singleton, ma può anche rappresentare un gruppo di nodi DTN, "multicast"o "anycast", gruppi contenti più nodi.

2.2.5 Formato di un bundle

Ogni bundle è costituito dalla concatenazione di almeno due blocchi. Il primo blocco della sequenza, o *primary block*, contiene informazioni analoghe a quelle di un'intestazione IP, necessarie all'instradamento del bundle verso destinazione. Ogni bundle può avere un solo primary block, ma può essere seguito da una serie di blocchi per supportare le estensioni del protocollo, come il Bundle Security Protocol (BSP). Può esistere nei blocchi successivi al primo, al massimo un blocco di payload. La maggior parte dei campi hanno lunghezza variabile e utilizzano una notazione compatta detta *self-delimiting numerical values* (SDNVs) (rif.), estendibile scalabile per una diversa varietà di protocolli di rete e dimensioni di payload.

Primary Block

Il Primary Block (Figure 2.5), oltre a versione, lunghezza del blocco, sorgente e destinzione, contiene una serie di informazioni tipiche del Bundle Protocol.

Bundle Processing Control Flag

I Bundle Processing Control Flags costituiscono una stringa di bit utili al processamento del bundle. Sono suddivisi in 3 categorie:

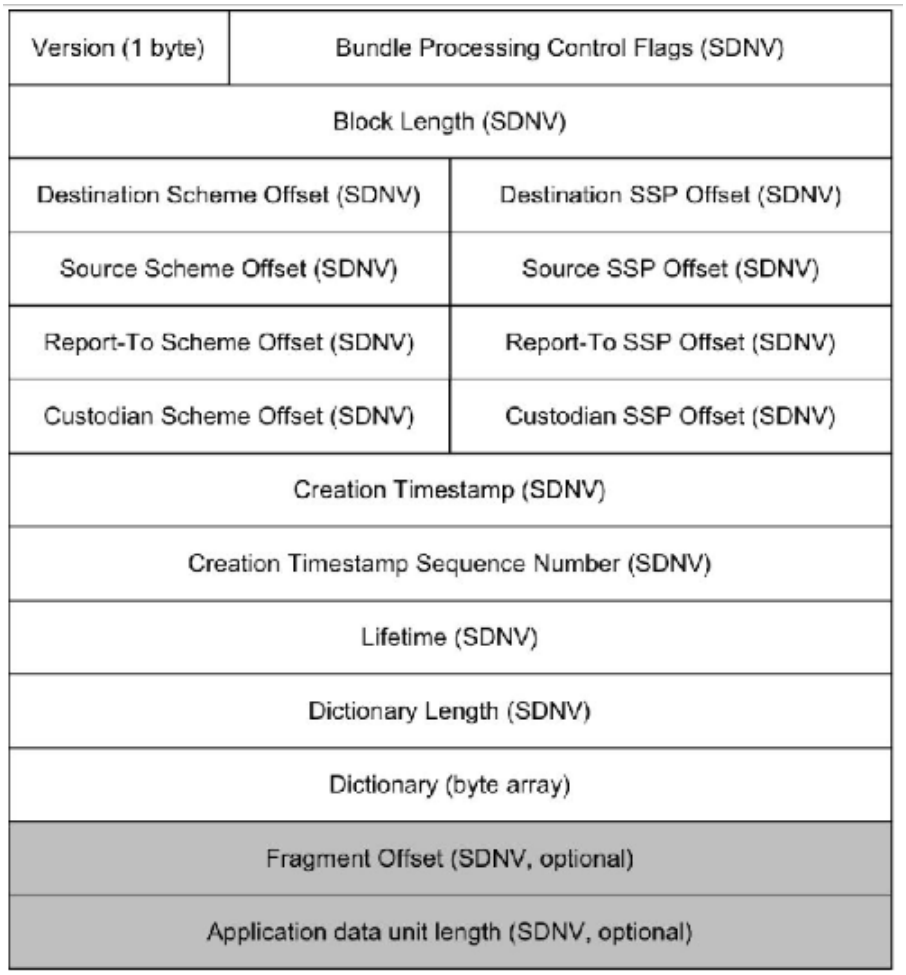

Figura 2.5: Formato del primary block di un bundle.

- General [0-6]: specificano informazioni di carattere generale sul bundle, ad esempio, se è regolare o amministrativo, lo stato di frammentazione, se la destinazione è un EID singleton, se sono richiesti acknowledgment o trasferimento di custodia.

- Class of Service [7-13]: specificano la priorità del bundle, dove un valore elevato indica una priorità elevata, e altre informazioni utili al routing del pacchetto.

- Status Report [14-20]: specificano i report richiesti per questo bundle, ad esempio se è richiesto il report di consegna, di inoltro, di accettazione di custdia, ecc.

Priorità

Dei bit "Class of Service" due vengono usati per definire la priorità del Bundle. Tipicamente vale solo tra bundle aventi la stessa sorgente, e può non essere rispettata nei confronti di bundle con sorgente diversa.

Tre sono i valori fino ad ora adoperati:

- Bulk: indica bundle che devono essere spediti con il minimo dello sforzo,consegnati solo al termine della consegna di tutti bundle con la stessa sorgente e destinazione.
- Normal: per i bundle che vengono spediti prima di quelli a priorità Bulk.
- Expedited: per i bundle con priorità maggiore, da essere spediti prima di quelli con priorità Normal e Bulk.

Endpoints

Il primary block include quattro EID di lunghezza variabile, ognuno 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 posizionato successivamente nel blocco.

- Source: contiene l'endpoint dalla quale proviene il bundle,
- Destination: è l'endpoint di destinazione del bundle,
- Report-to: indica il nodo a cui inviare gli status report per eventi che coinvolgono il bundle,
- Custodian: identifica l'ultimo nodo che ha accettato la custodia del bundle.

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 l' EID sorgente e report-to coincidano, compariranno due riferimenti a tale EID, ma un'unica stringa all'interno del dizionario.

Tempo

Altre informazioni significative per l'elaborazione di un bundle sono il *creation timestamp* e il *lifetime*. Il *creation timestamp* indica il tempo di creazione del bundle, espresso come il numero di secondi trascorsi dall'inizio dell'anno 2000 nel fuso orario UTC. Questo valore è calcolato nell'istante in cui il BPA riceve la richiesta di trasmissione. Il *lifetime*, invece, rappresenta il tempo di vita del bundle,

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

Altri blocchi

Oltre al Primary Block all'interno di un bundle possono essere inseriti diversi altri blocchi. Come si può notare in figura 2.6, ognuno di questi blocchi è identificato dal *Block Type*, una stringa di 8 bit. Il valore '1' indica un blocco payload e un bundle ne può contenere massimo uno, i valori tra 192 e 255 sono ad uso sperimentale e privato, mentre i restanti sono riservati per usi futuri. Tutti i blocchi diversi da quello primario e dal payload sono detti extension block. Poi sono ci sono i flag di controllo del blocco, che danno indicazioni su come il blocco deve essere trattato. Infine completano il blocco il body e la loro lunghezza. E' inoltre possibile inserire il riferimento ad alcuni EID contenuti nel dizionario. Un contatore ne terra traccia e due puntatori, uno all'inizio dello schema e uno all'inizio dell'SSP nel dizionario per ogni entry.

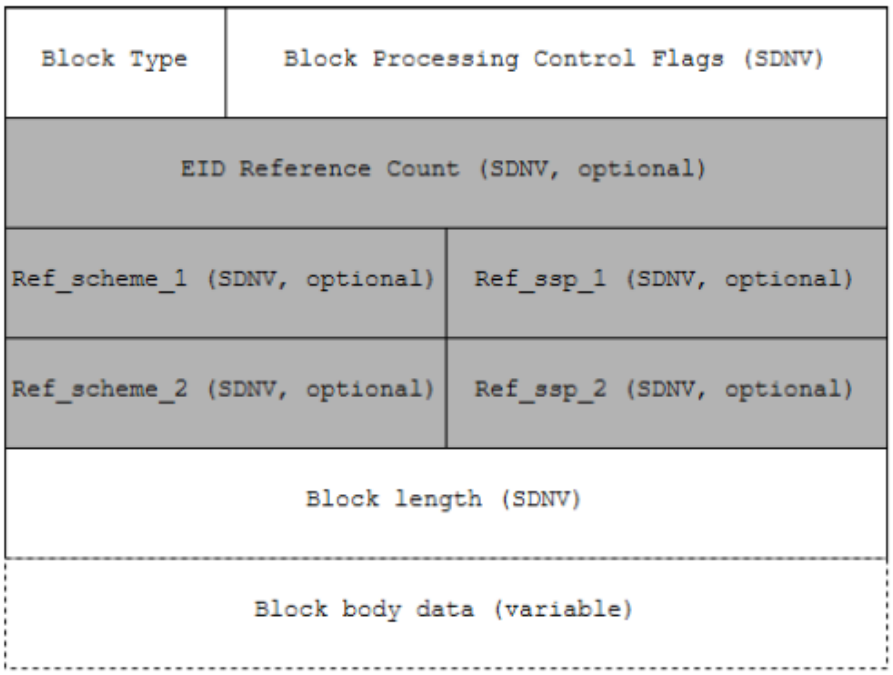

Figura 2.6: Formato generico di un blocco secondario di un bundle.

Block Processing Control Flag

I *Block Processing Control Flags* costituiscono una stringa di bit utili al processamento del blocco. E' un campo SDNV attualmente formato da 7 bit, indicanti alcuni particolari accorgimenti sul blocco. Per esempio abbiamo la possibilità di replicare il blocco in ogni frammento (in casodi frammentazione), indicare di scartare il blocco o l'intero bundle o inviare un report se non si è in grado di processare il blocco, se contiene degli EID-Reference, e sopratutto il flag che indica se è l'ultimo blocco del Bundle. Il bit di replicazione nei frammenti però non può essere settato a uno sui blocchi successivi a quello di payload.

2.2.6 Affidabilità delle trasmissioni

Le DTN supportano meccanismi di ritrasmissione di dati persi e/o corrotti sia a livello dei protocolli di trasporto che a livello di Bundle Protocol. Tuttavia, poiché le DTN presentano tipicamente un'eterogeneità nei protocolli di trasporto utilizzati dai nodi, l'affidabilità deve essere realizzata a livello di Bundle Protocol, mediante un meccanismo di ritrasmissione da nodo a nodo detto *trasferimento in custodia*. Di base, quando il custode corrente di un bundle deve inoltrarlo, richiede il trasferimento in custodia e fa partire un timer di ritrasmissione. Se il BPA del nodo ricevente decide di accettare la custodia, invia un acknowledgement al mittente. Se non viene ricevuto alcun acknowledgement 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 uno 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 dirisorse disponibili, per una questione di policy o di implementazione. 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, permette 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 la consegna a destinazione di un bundle destinato al mittente dello stesso. Tuttavia, un'eccessiva quantità di bundle o frammenti di essi può portare ad un eccessivo consumo 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, attività che dovrebbe essere intrapresa comunque con regolarità, trasferire dei bundle ad altri, non accettare bundle con trasferimento in custodia, piuttosto che bundle regolari, eliminare bundle non scaduti, anche se il nodo ne è il custode. L'utilizzo di quest'ultima opzione è assolutamente sconsigliato, poiché chiaramente contraddittoria rispetto ai principi cardine delle DTN.

2.3 IBR-DTN Node

2.3.1 Introduzione

IBR-DTN è l'applicativo scelto per la realizzazione di una infrastruttura DTN che una volta installato su dispositivi ne permette l'inserimento in rete e gestisce la comunicazione via bundle. Modulare e leggera IBR-DTN è stata creata dal gruppo di ricerca sulle DTN del Technische Universität Braunschweig. Studiato per essere installato su sistemi embedded fornisce allo sviluppatore un framework per creare applicazioni DTN[3].

2.3.2 L'architettura

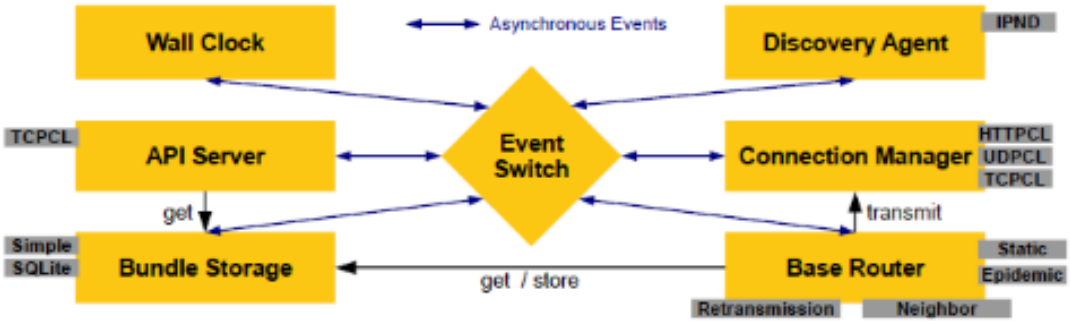

Figura 2.7: Architettura IBR-DTN.

La versione di IBR-DTN per i sistemi operativi tradizionali è stata sviluppata in C++. Come si può notare in figura 2.7, l'implementazione del bundle protocol di IBR-DTN è contraddistinta da un'organizzazione fortemente modulare, tale da permettere agli sviluppatori di estendere il software in maniera semplice e poco invasiva. Il Bundle Protocol Agent è implementato come processo demone ed espone una API basata su socket che le applicazioni possono contattare per interagire

con il Bundle Layer. Di default l'API è disponibile alla porta TCP 4550 in formato testuale e binario. Per maggiori informazioni sulle funzionalità esposte si può fareriferimento alla documentazione[4].

Event Switch

I moduli sono collegati in maniera flessibile e comunicano tra loro tramite un meccanismo basato su eventi, rendendo fondamentale l'*Event Switch*, incaricato di affidare la gestione dei singoli eventi ai sotto-moduli corrispondenti. Tutti i moduli possono ricevere o scatenare eventi per comunicare con le altre parti del software. Nell'implementazione attuale sono integrati una serie di eventi per notificare le operazioni di storage, la presenza e scomparsa di nodi nel vicinato, le operazioni di routing dei bundle, ecc.

Discovery Agent

Un altro importante componente è il *Discovery Agent*, responsabile di scoprire i nodi nel vicinato. Sotto l'ipotesi di voler far comunicare nodi IP, IBR-DTN utilizza un modulo che implementa il protocollo DTN IP Neighbor Discovery (IPND)[5]. Tale modulo rimane in ascolto di piccoli datagrammi UDP detti *beacon*, utilizzati dai nodi per annunciare la propria presenza ai vicini, e periodicamente si annuncia tramite i medesimi datagrammi. I *beacon* sono spediti ad un indirizzo IP multicast noto (e specificabile in configurazione) e contengonol'EID del mittente, per permettere a chi lo riceve di effettuare il binding tra EID eindirizzo IP del vicino.

Connection Manager e Convergence Layer

Ad occuparsi della gestione delle connessioni con i nodi vicini e dell'invio e della ricezione di bundle è il modulo *Connection Manager*. Il *Connection Manager* a sua volta per l'implementazione del trasferimento di informazioni sfrutta diversi *convergence layer*. Come descritto dall'RFC 5050[2] sulle Delay Tolerant Network, sono i *convergence layer* ad occuparsi della comunicazione tra due nodi. Ognuno di essi definisce un'interfaccia verso il livello di trasporto sottostante, permettendo il trasferimento di bundle astraendosi dai protocolli di livello inferiore. I convergence layer utilizzati sono specificati nella configurazione del demone. Attualmente IBR-DTN offre convergence layer per TCP/IP[6], UDP/IP, HTTP, Bluethooth, IEEE 802.15.4 LoWPAN e grazie a questo lavoro di tesi, per V2X. Esiste anche un'estensione del TCP/IP CL per il supporto a TLS.

Bundle Storage

Poiché le DTN sono basate sul paradigma store-and-forward, ogni nodo deve essere capace di memorizzare bundle per un certo periodo di tempo. In IBR-DTN l'interazione con lo storage è gestita dal *Bundle Storage*, modulo che fornisce primitive per la lettura, cancellazione e memorizzazione dei bundle da/verso lo storage. Sono supportati diversi meccanismi di memorizzazione: in memoria RAM, su disco (file-system) e su basi di dati SQLite.

Base Router

Il routing dei bundle è invece realizzato dal modulo *Base Router*, che si occupa di gestire il forwarding dei bundle che ha in carico. Il *Base Router* suddivide il proprio lavoro tra i diversi moduli di routing. Ognuno di essi implementa uno specifico algoritmo di routing DTN ed è agganciato al *Base Router* come una sorta di plugin. Tutti i moduli di routing sono notificati dal *Discovery Agent* al verificarsi di eventi legati al vicinato del nodo e dal *Bundle Storage* nel momento in cui un nuovo bundle giunge al demone. Il modulo di routing che si riterrà responsabile dell'inoltro del bundle contatterà poi il Connection Manager per attivare il convergence layer opportuno. IBR-DTN presenta moduli per il supporto al routing statico, epidemico e PRoPHET e MaxProp.

API Server

L'interazione con IBR-DTN è ottunuta tramite l'API server. L'API server espone su un' interfaccia socket, configurabile tramite config file, un protocollo testuale con cui effettuare richieste al core dell'applicativo. I comandi da inviare devono seguire la specifica logica e sintassi dell'azione da eseguire, all'invio di un comando e all'inserimento di tutte le informazioni che esso richiede, si riceve sempre un response formato come <status-code> <message> [Additional data], come mostrato in Figura 2.8.

Service Discovery

É un modulo creato per propagare i servizi presenti su ogni nodo, all'interno della rete DTN. Con servizi si intende la descrizione delle capacità di un'applicazione, residente sul nodo, che può utilizzare la rete DTN per inviare e/o ricevere dati. La Service Discovery permette a tutti i nodi la conoscenza di tale servizio grazie a messaggi multicast inviati ad un indirizzo DTN di gruppo. Grazie alla Service Discovery le applicazoni possono selezionare il nodo DTN che più si avvicina alle loro richieste ed iniziare ad inviare pacchetti verso di esso.

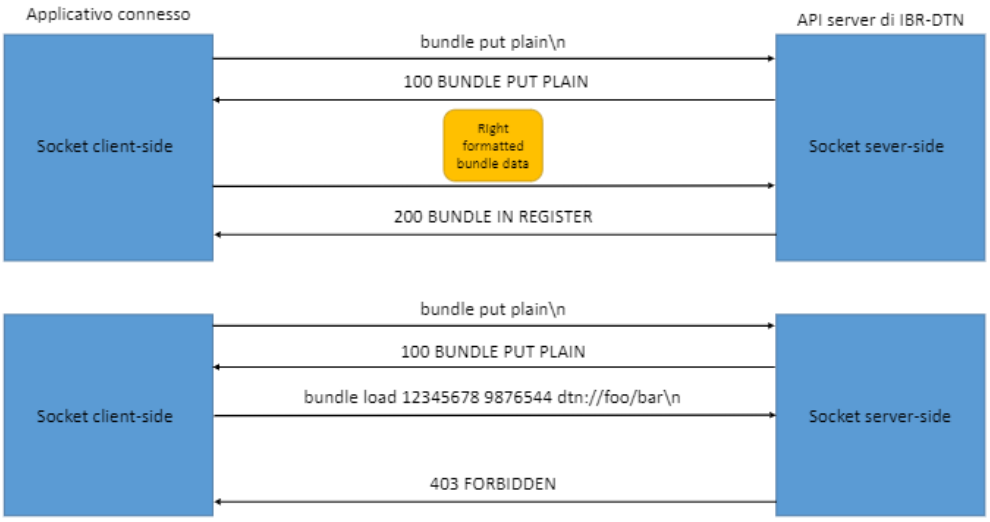

Figura 2.8: Interazione API Server.

2.3.3 Installazione e avvio

In questo paragrafo sono illustrati i passaggi necessari all'installazione e avvio deldemone IBR-DTN. Le procedure utilizzate sono valide per distribuzioni Linux,Debian e derivate (Raspbian).

Installazione

Il primo passo consiste nell'installazione delle librerie necessarie. In realtà, l'installazione di alcune librerie elencate in seguito è opzionale, in quanto esse risultano utili nel momento in cui si aggiungano moduli opzionali.

```
1 $ apt-get install build-essential libssl-dev zlib1g-dev libsqlite3-dev libcurl4-gnutls-dev
      libdaemon-dev automake autoconf pkg-config libtool libcppunit-dev libnl-3-dev
      libnl-cli-3-dev libnl-genl-3-dev libnl-nf-3-dev libnl-route-3-dev libarchive-dev git
```

L'installazione e l'attivazione/disattivazione dei moduli opzionali di IBR-DTN prevede l'utilizzo di `CMAKE`. Dopo aver creato la cartella build in */ibrdtn/ibrdtn*, si procederà a lanciare il comando `cmake ..` con l'opzione per attivare tutti i moduli che il sistema operativo può supportare. Quindi dopo aver clonato il repository, si esegue la configurazione, la compilazione e l'installazione dei sorgenti:

Nel caso in cui si voglia avere un controllo più specifico delle opzioni, basterà utilizzare le variabili cmake

Il dettaglio di tutte le opzioni possibili è riportato nel file *CMakeList.txt* all'interno della cartella */ibrdtn/ibrdtn* presente nel repository.

```
$ cd ibrdtn-repository/ibrdtn
$ mkdir build
$ cd build
$ cmake .. -DBUILD_ALLFEATURES=ON
$ make
$ make install
$ ldconfig
```

```
$ cmake .. -DBUILD_<OPTIONS>=ON/OFF
```

Avvio

Dopo aver completato l'installazione, è possibile avviare il demone IBR-DTN utilizzando il comando **dtnd**. Tale comando, invocato senza opzioni, avvia il demone utilizzando la configurazione di default. E' possibile utilizzare l'opzione **-i** per specificare l'interfaccia di rete alla quale associare il processo demone, oppure **-v** per abilitare la stampa dei messaggi di log, **-d** per scegliere il livello di log che verrà stampato, etc. Un esempio del comando:

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

Con questa combinazione di parametri avremo il binding sull'interfacciadi rete *eth0* e log sulla console per le informazioni pricipali. Per un elenco completo delle opzioni disponibili utilizzare il flag **-h**. Una volta avviato, il demone IBR-DTN rileverà automaticamente la presenza di demoni in esecuzione su macchine direttamente raggiungibili tramite il modulo di IP Neighbor Discovery e simultaneamente, annuncerà il suo EID locale per essere scoperto dagli altri. Nella configurazione di default, tale EID utilizza lo schema DTN e il nome della macchina locale come SSP,nella forma **dtn://hostname**.

2.3.4 Configurazione

Per modificare il comportamento predefinito del demone è necessario specificare i parametri da utilizzare all'interno di un file di configurazione. Un esempio di configurazione è reperibile al path **ibrdtn/daemon/etc/ibrdtnd.conf**, all'interno del repository utilizzato per l'installazione, o all'indirizzo [7]. In seguito sono illustratele parti più significative.

permette di personalizzare l'EID locale, se non specificato IBR-DTN ne creerà uno per noi secondo la formattazione standard degli URI dtn **dtn://hostname**.

Definisce in che modo salvare fino alla scadenza del TTL i bundle. In memoria volatile (RAM) o su disco se specificato un path di salvataggio.

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

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

```
#a list(separated by spaces) of names
#for convergence layer instances.
net_interfaces = lan1 lan0

#configuration for a convergence layer named lan0

net_lan0_type = tcp # we want to use TCP as protocol
net_lan0_interface = eth0 # listen on interface wlan0
net_lan0_port = 4556 # with port 4556 (default)

#configuration for a convergence layer named lan1

net_lan1_type = tcp      # we want to use UDP as protocol
net_lan1_interface = eth1 # listen on interface eth0
net_lan1_port = 4557 # with port 4556 (default)

#configuration for a convergence layer named blue0

#net_hci0_type = bluetooth # use bluetooth as protocol
net_hci0_interface = hci0 # listen on interface
net_hci0_port = 10 # with RFCOMM channel

#configuration for a convergence layer named v2x0

net_v2x0_type = v2x # we want to use v2x as protocol
net_v2x0_interface = eth0 # listen on interface
net_v2x0_port = 0 # with no channel
```

Avendo a disposizione diversi convergence layer è possibile elencare nel file di configurazione tutte le interfacce disponibili sul dispositivo. IBR-DTN proverà, dunque, a fare un bind sui protocolli scelti, permettendo così la comunicazione su bundle.

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

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

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

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

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

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

2.3.5 Configurazione della time synchronization

La sincronia temporale è un punto critico della configurazione di IBR-DTN. Nel momento in cui si utilizzano dei dispositivi reali, che non hanno la possibilità di avere un orologio sempre sincronizzato con il resto del mondo, è necessario disattivarla.

Attivare la time synchronization significa avere la possibilità di scartare i bundle all'arrivo se questi sono troppo vecchi e quindi ritenuti inutili, ma questo è un comportamento che può portare all'impossibilità di comunicazione tra i dispositivi DTN. La rete DTN in quanto tale non prevede che i nodi all'interno abbiano perennemente accesso ad un servizio di time synchronization esterno come ad esempio l'NTP e non è in alcun modo garantito che i device abbiano l'orologio interno settato entro un certo ritardo. L'esempio possibile è quello di un dispositivo con sola connessione bluetooth che viene utilizzato per poche ore, per poi essere riacceso molto tempo più avanti. L'orologio di sistema di quest'ultimo non sarà mai sincronizzato con il resto della rete, perciò se la time synchronization viene attivata il dispositivo non creerà mai dei bundle validi all'interno della rete. È perciò consigliabile disattivare tale comportamento.

```
# set to yes if this node is connected to a high precision time reference
# like GPS, DCF77, NTP, etc.
#
time_reference = no
```

2.3.6 Applicativi di interazione con IBR-DTN

Al fine di sperimentare l'utilizzo del DTN Bundle Protocol, oltre al processo demone, il software IBR-DTN mette a disposizione una serie di tool a linea di comando. **dtnping** invia dei bundle ad uno specifico EID destinazione e si mette in attesa delle risposte, misurando il tempo di andata/ritorno. **dtnsend** e **dtnrecv** permettono il trasferimento di file tra nodi DTN. Qualora si voglia testare l'API testuale esposta dal demone, è possibile usare strumenti come **telnet** o **netcat**, come nell'esempioseguente:

```
1  $ telnet localhost 4550
2  Trying ::1...
3  Connected to localhost.
4  Escape character is '^]'.
5  IBR-DTN 0.11.0 (build dfb7402) API 1.0
6  protocol management
7  200 SWITCHED TO MANAGEMENT
8  neighbor list
9  200 NEIGHBOR LIST
10 dtn://neighbor1
11 dtn://neighbor2
```

In questo esempio, dopo la connessione al demone IBR-DTN in esecuzione localmente alla porta 4550, si invoca il comando **protocol management** per accedere alla API di Management. È possibile richiedere la lista dei nodi DTN adiacenti al nodo locale, utilizzando il comando **neighbor list**, come riportato in esempio o inviare comandi per la gestione dei bundle.

Capitolo 3

Architettura V2X

3.1 Standard ETSI ITS-G5

3.1.1 Introduzione

ITSC (Intelligent Transport System Communication) [8] è un nuovo tipo di sistema di comunicazione dedicato agli scenari di trasporto (Figura 3.1), ed è basato su due domini:

- dominio ITS,
- Dominio generico

Gli aspetti essenziali da considerare nel processo di sviluppo degli standard ITSC sono:

- la mobilità delle stazioni ITS (ITS-S) che determina una elevata dinamicità delle loro topologie,
- il supporto potenziale di qualsiasi tipo di tecnologia di comunicazione, inclusi Internet, reti pubbliche e private, sistemi come Bluetooth, tecnologie per il pedaggio stradale (DSRC),
- il supporto potenziale per ogni tipologia di applicazione:
  - applicazioni designate per ITSC,
  - applicazioni che usano le stazioni ITS come mezzi di comunucazione in maniera trasparente,
- la considerazione dinamica e flessibile delle esigenze degli utenti, ad es. rispetto alla capacità di comunicazione (velocità dati), costo delle comunicazioni (in termini di denaro), affidabilità delle comunicazioni, disponibilità delle comunicazioni, privacy delle comunicazioni,

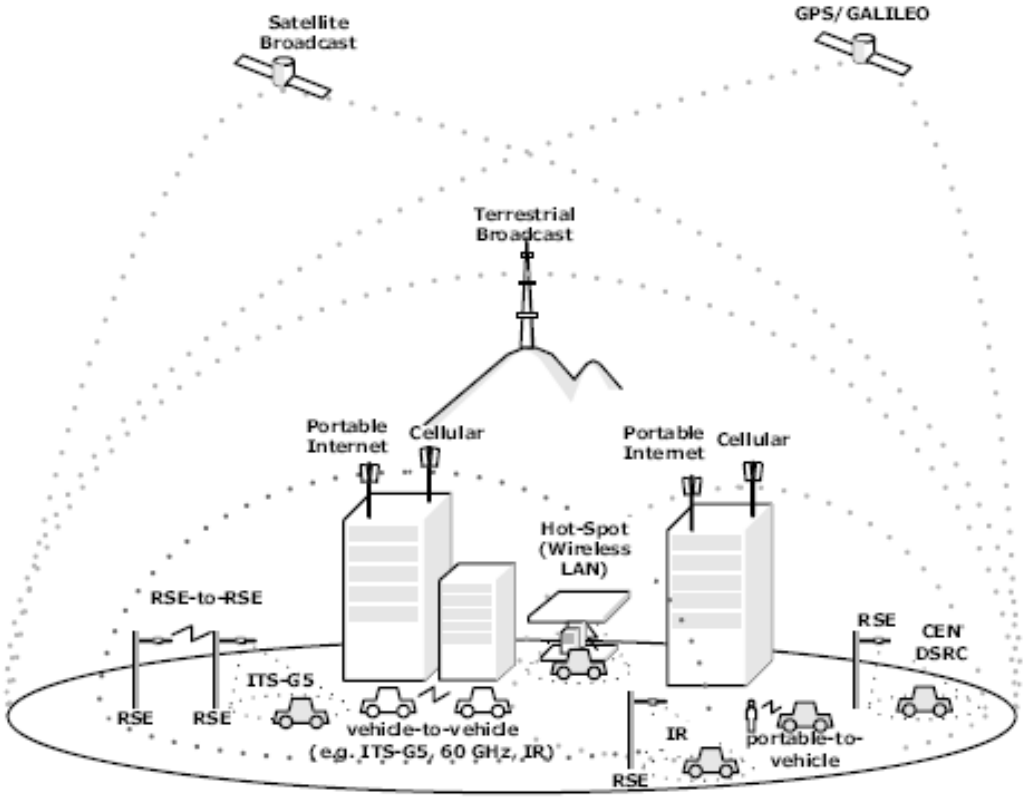

Figura 3.1: Scenario ITSC [8].

- il potenziale di un meccanismo prioritario per classi d'accesso,
- il supporto di implementazioni modulari con più unità fisiche collegate in rete in una singola stazione ITS,
- l'applicabilità globale.

L'architettura di riferimento di una stazione ITS segue i principi del modello OSI per i protocolli di comunicazione a più livelli che è estesa per l'inclusione delle applicazioni ITS. I tre blocchi inferiori al centro delle figure seguenti contengono funzionalità dello stack del protocollo di comunicazione OSI:

- "Access" che rappresenta i livelli OSI 1 e 2 di ITSC,
- "Networking & Transport" che rappresenta i livelli OSI 3 e 4 di ITSC,
- "Facilities" che rappresenta i livelli OSI 5, 6 e 7 di ITSC.

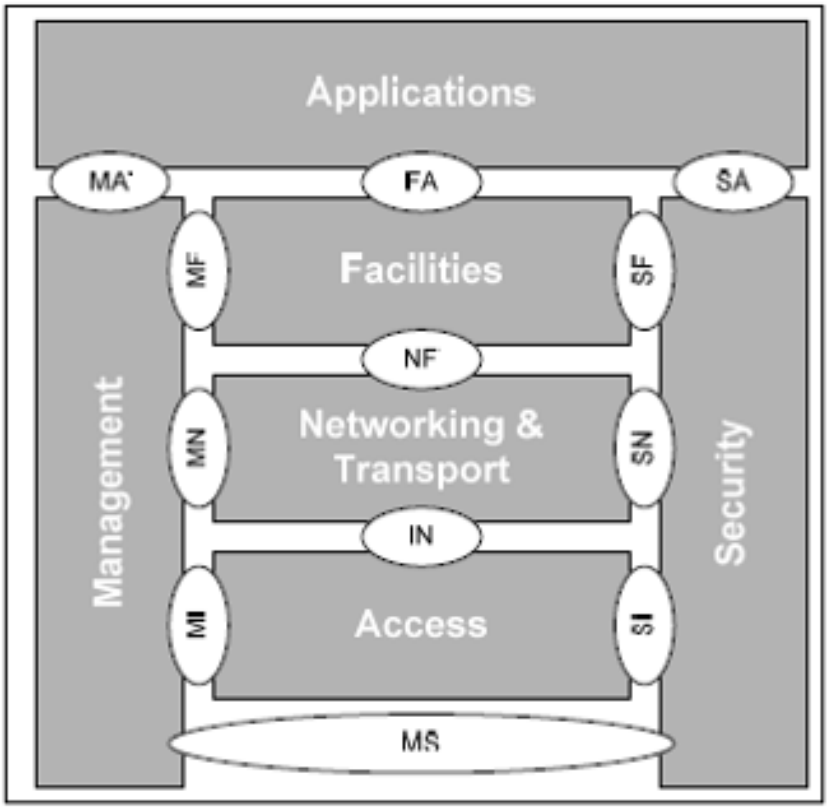

Figura 3.2: Architettura di riferimento di una stazion ITS.

3.1.2 Access Layer

Il livello di accesso raggruppa il livello di data link e il livello fisico ed è situato nella parte inferiore della pila protocollare ITS.

Lo strato fisico di ITS-G5 utilizza un "half-clock" orthogonal frequency division multiplexing (OFDM) utilizzando i canali di frequenza a 10 MHz. L'idea di base è quella di dividere lo spettro di frequenza disponibile in sottocanali più stretti (sottovettori). Il flusso di dati ad alta velocità è suddiviso in un numero di flussi di dati di velocità inferiore trasmessi simultaneamente su un numero di vettori secondari. Lo strato OFDM PHY ha supporto per otto diverse velocità di trasferimento, ottenute utilizzando diversi schemi di modulazione e velocità di codifica. Il supporto di tre velocità di trasferimento è obbligatorio; 3 Mbit/s, 6 Mbit/s e 12 Mbit/s.

Il livello data link è costituito da due sublayer: Logical Link Control (LLC) e Medium Access Control(MAC) - dove il primo fornisce mezzi per distinguere tra diversi protocolli di livello di rete e il secondo è responsabile della programmazione delle trasmissioni per ridurre al minimo le interferenze tra le stazioni ITS.

L'algoritmo MAC decide quando, nel tempo, un nodo può trasmettere in base

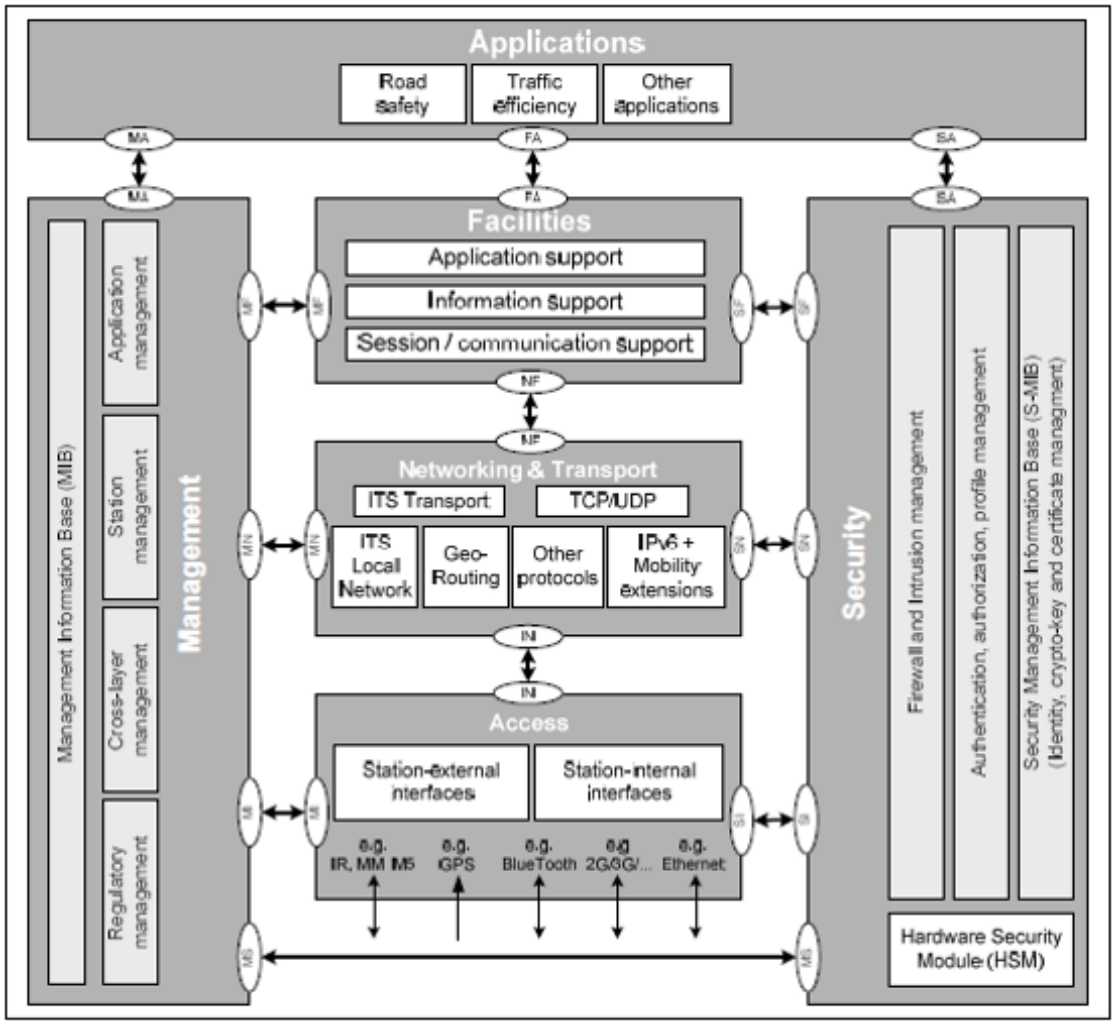

Figura 3.3: Esempio dei possibili elementi presenti nell'architettura di ruferimento di una stazione ITS.

allo stato del canale corrente e alla schedulazione MAC della trasmissione con l'obiettivo di ridurre al minimo le interferenze nel sistema per aumentare la probabilità di ricezione dei pacchetti. L'algoritmo MAC si chiama Enhanced Distributed Coordination Access (EDCA). Si basa sulla funzione Distributed Coordination Function (DCF) di base ma aggiunge attributi come la QoS. DCF è un algoritmo Carrier Sense Multiple Access con Collision Avoidance (CSMA/CA).

Decentalized Congestion Control - DCC

L'obiettivo del Decentralized Congestion Control (DCC) è di adattare i parametri di trasmissione della stazione ITS (ITS-S) date le attuali condizioni del canale

radio, al fine di massimizzare la probabilità di ricezione ai destinatari previsti. Il DCC tenta di fornire uguale accesso alle risorse del canale tra le vicine ITS-S. Le risorse del canale assegnato dal DCC ad un ITS-S dovrebbero essere distribuite tra le applicazioni in base alle loro esigenze. Un ITS-S determina le priorità tra i diversi messaggi e scarta i messaggi se i requisiti dell'applicazione superano le risorse assegnate. In caso di una situazione di emergenza del traffico stradale anche durante un alto periodo di utilizzo della rete, in cui ogni ITS-S ha pochissime risorse, un ITS-S può ancora trasmettere una raffica di messaggi per un breve periodo di tempo per mantenere un ambiente sicuro nel traffico stradale, i messaggi trasmessi a tale scopo sono solo quelli della massima importanza.

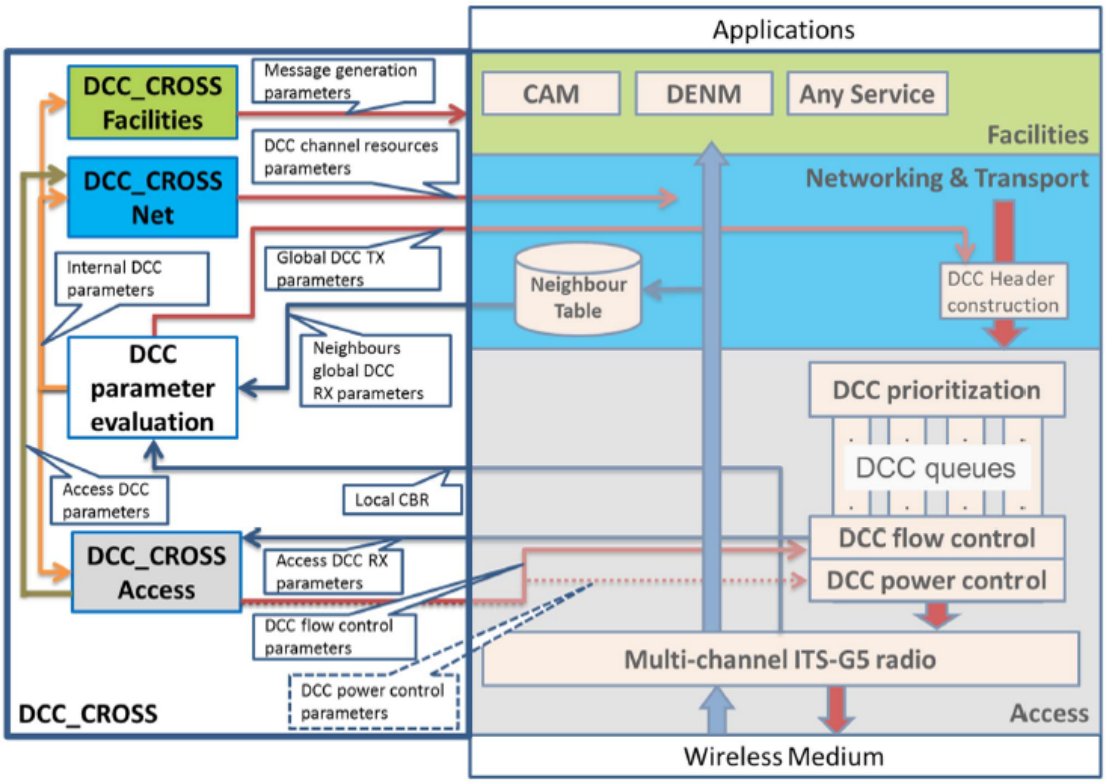

Figura 3.4: Overview del DCC in una stazion ITS.

L'entità di gestione DCC (DCC_CROSS) nel piano di gestione contiene per ogni livello una funzione che è collegata all'interfaccia verso l'entità DCC corrispondente nello stack di comunicazione.

DCC_ACC

- *Valutazione CBR:* Il componente DCC_ACC fornisce il valore CBR (Channel Busy Ratio) locale all'algoritmo DCC e deriva il CBR locale dal carico del

canale (CL), per tutti i canali radio utilizzati dalla ITS-S,

- *Definizione delle priorità DCC:* Seleziona la coda DCC a cui inoltrare il messaggio in base alla classe di traffico (TC) indicata nel messaggio. La TC corrispondente alla classe di accesso EDCA più alta è mappato alla più alta coda DCC prioritaria, in modo che venga prima rimossa dalla coda dal controllo di flusso DCC,
- *Coda DCC:* Memorizza temporaneamente un messaggio se il canale radio è sovraccarico. Rilascia il messaggio quando il tempo di accodamento supera la durata del messaggio. Questo processo è un meccanismo interno a DCC_ACC. Esso invia un'indicazione di quell'evento a DCC_CROSS, che viene inoltrato al mittente del messaggio se la relativa interfaccia DCC è implementata,
- *Controllo dell'alimentazione DCC:* Se disponibile, determina il livello di potenza TX associato al messaggio in base a informazioni fornite dall'entità DCC_CROSS,
- *Controllo del flusso DCC:* Esegue lo shaping del traffico in base ai parametri forniti da DCC_CROSS_Access. A seguito di questi requisiti, elimina il messaggio con la massima priorità archiviato nella coda DCC e lo inoltra al canale radio ITS-G5, per la trasmissione.

DCC_NET

- Memorizza i parametri DCC globali ricevuti da altri ITS-S e li valuta per inoltrare il CBR globale all'Entità DCC_CROSS,
- Diffonde i parametri DCC locali ai vicini ITS-S inserendo il loro valore nell'intestazione GeoNetworking,
- *Coda DCC:* Memorizza temporaneamente un messaggio se il canale radio è sovraccarico. Rilascia il messaggio quando il tempo di accodamento supera la durata del messaggio. Questo processo è un meccanismo interno a DCC_ACC. Esso invia un'indicazione di quell'evento a DCC_CROSS, che viene inoltrato al mittente del messaggio se la relativa interfaccia DCC è implementata,
- Indica i seguenti parametri delle risorse del canale DCC all'algoritmo di inoltro GeoNetworking:
  - Il tempo di inattività Toff corrente per canale radio,
  - Se disponibile, il tempo in cui l'ultimo messaggio è stato inoltrato alla radio ITS-G5, come riportato dall'entità DCC_ACC.

DCC_FAC

Nessun messaggio viene eliminato da questa funzione. DCC_FAC include le seguenti funzionalità:

- Controllo del carico generato dai messaggi di servizio CAM e DENM sul canale radio. Il carico è controllato da un'indicazione fornita al servizio delle strutture di base o all'applicazione che genera i messaggi,
- Potenziale trigger del cambio di canale nel caso in cui ITS-S abbia la capacità di eseguire questa funzione,
- Mappare la priorità del messaggio impostata dal servizio delle strutture di base o dall'applicazione nel campo della classe di traffico del messaggio.

Esistono diverse tecniche per controllare il carico di rete:

- Transmit Power Control (TPC),
- Transmit Rate Control (TRC),
- Transmit Datarate Control (TDC).

Una o più tecniche combinate possono essere utilizzate dall'algoritmo DCC per controllare il carico di rete [9].

Technique	Description
TPC	In TPC, the output power is altered to adjust the current channel load. For example, during high utilization periods the ITS-S can reduce its output power and thereby, is a reduction in interference range achieved. This results in that ITS-Ss further away will experience a reduced CBR.
TRC	TRC regulates the time between two consecutive packets from an ITS-S. During high utilization periods, the TRC increases the time between two packets for the ITS-S, T_{off} time.
TDC	TDC is a mechanism that can be used by wireless systems offering several transfer rate options. During high utilization periods and depending on application, a higher transfer rate can be used to decreased the T_{on} time.

Figura 3.5: Meccanismi di controllo del carico del canale [9].

L'algoritmo DCC simula una macchina a stati finiti e a seconda dello stato attuale, può combinare diverse tecniche per il controllo del carico del canale. Nel caso più semplice, viene utilizzata solo una delle tecniche proposte. Le due tecniche più scrutinate in letteratura sono il controllo della velocità di trasmissione e il controllo della potenza. L'algoritmo inoltre è soggetto ai seguenti requisiti:

- deve funzionare su ciascun canale di frequenza in modo indipendente,
- deve essere eseguito in un ciclo infinito,
- deve essere attivato almeno ogni 200 ms,

- non deve superare i limiti previsti,
- la valutazione CBR deve essere conforme al punto 4.2.10 della norma ETSI EN 302 571 [10].

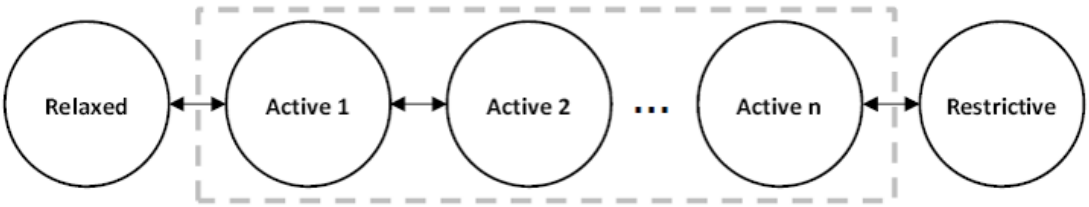

Figura 3.6: Outline generico della macchina a stati simulata dall'algoritmo DCC [9].

Di seguito un esempio di valori e ripartizioni del carico e delle opportunità di trasmissione per una macchina con 3 active state.

State	Channel load	Packet rate	T_{off}
Relaxed	< 30 %	10 Hz	100 ms
Active 1	30 % to 39 %	5 Hz	200 ms
Active 2	40 % to 49 %	2,5 Hz	400 ms
Active 3	50 % to 60 %	2 Hz	500 ms
Restrictive	> 60 %	1 Hz	1 000 ms

Figura 3.7: Mapping tra valori del CBR e stati e alle opportunità di trasmissioni al secondo permesse [9].

Utilizzo e gestione dei canali

L'entità di configurazione del canale distribuisce il traffico offerto al livello di accesso su un set fisso di canali disponibili, secondo determinati requisiti sotto il controllo del Management Layer e dell'entità di gestione DCC corrispondente. L'Access Layer deve monitorare i seguenti indicatori di stato dei canali ITS G5A e G5B :

- Channel Busy Ratio: tempo relativo in % in cui il canale è occupato in base alla definizione del tempo di occupazione del canale,
- Statistiche del Receiver Signal Strength Indicator(statistica RSSI),
- Indicazione di trasmissione dei frame (se un messaggio è stato trasmesso o eliminato correttamente).

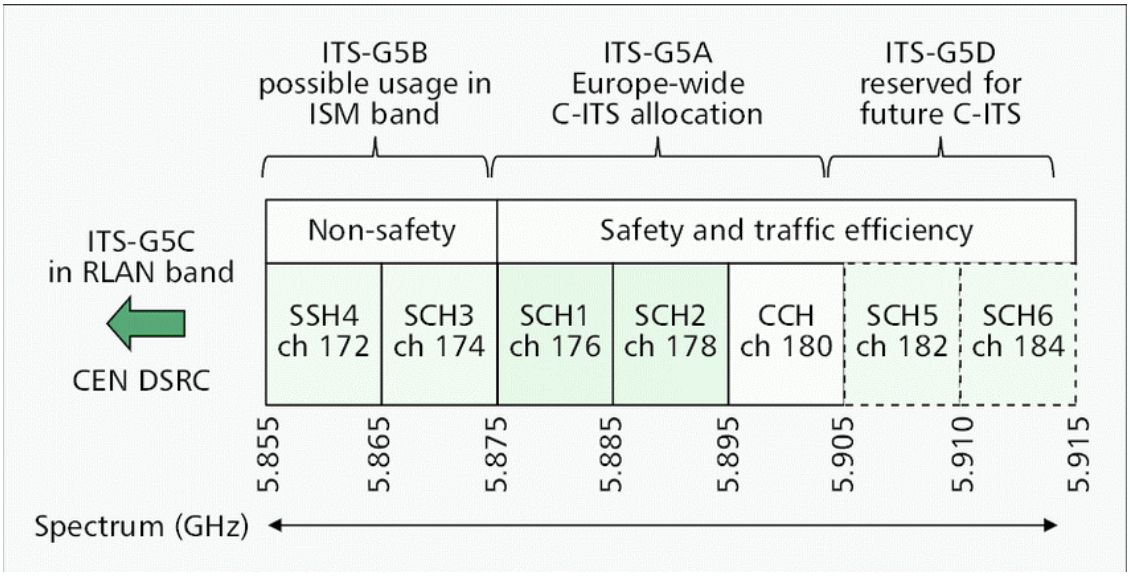

Figura 3.8: Allocaziosne dei canali [11].

Inoltre il livello di accesso deve notificare della riduzione della potenza di Tx su una base per messaggio nel caso in cui il messaggio non potesse essere trasmesso tramite il livello di potenza di TX richiesto. Il livello di accesso deve fornire queste informazioni all'entità DCC_Management tramite SAP.

La banda **ITS G5A** (da 5.875 GHz a 5.905 GHz) contiene i canali CCH, SCH1 e SCH2 dedicati a servizi relativi alla sicurezza stradale. L'uso dei canali CCH, SCH1 e SCH2 deve essere controllato dal DCC [11].

Il **Control Channel (CCH)** è essenzialmente dedicato alla sicurezza stradale cooperativa.

Il **Service Channel 1 (SCH1)** è il canale predefinito per l'annuncio e l'offerta dei servizi ITS per la sicurezza e l'efficienza stradale sotto lo stato DCC "ACTIVE" e "RESTRICTED" del CCH. Le trasmissioni di altri tipi di messaggi su SCH1 sono consentite se le condizioni del canale in base alle restrizioni indicate in seguito lo consentono.

Il **Service Channel 2 (SCH2)** è il secondo canale di servizio su ITS G5A e viene utilizzato come canale alternativo per il traffico di servizi relativi alla sicurezza. A causa della sua allocazione di banda di frequenza tra CCH e SCH1 e i potenziali problemi di interferenza con il canale adiacente, la flessibilità della distribuzione di SCH2 è limitata.

La banda **ITS G5B** (da 5.855 GHz a 5.9875 GHz) contiene i canali SCH3 e SCH4. È utilizzata per servizi ITS general-purpose non sicuri (ad es. efficienza stradale, annunci di servizio, percorsi multipli, ecc.). La banda ITS G5B non è allocata a livello europeo. Una stazione ITS che opera sul canale ITS G5B deve essere conforme ai requisiti indicati nelle tabelle dello standard[11]. Per i profili

DCC relativi a più di un singolo canale, la stazione ITS deve distribuire il seguente ordine di utilizzo:

- Service Channel 3, SCH3,
- Service Channel 4, SCH4

Il messaggio ricevuto dal livello di accesso verrà trasmesso utilizzando il primo canale possibile nell'ordine di utilizzo a seconda dello stato di congestione e del profilo DCC del messaggio [11].

3.1.3 Networking and Transport Layer

Il livello di rete e trasporto ITS comprende diversi protocolli di rete e trasporto. Nel dettaglio una stazione ITS può eseguire i seguenti protocolli a livello di rete e trasporto ITS:

- **Protocollo GeoNetworking** per l'utilizzo di GeoNetworking su diverse tecnologie di accesso ITS, le specifiche del protocollo sono suddivise in una parte indipendente dai media e una parte dipendente dai media,
- **Protocolli di trasporto su GeoNetworking** come il protocollo di trasporto di base (BTP) e altri protocolli di trasporto GeoNetworking,
- **Protocollo Internet IPv6** con supporto per la mobilità IP o altri approcci a seconda dello scenario di distribuzione,
- **Protocollo Internet IPv4** per passaggio a IPv6,
- **Protocolli UDP/TCP**,
- **Altri protocolli di trasporto, come SCTP**,
- **Altri protocolli di rete**.

3.1.4 Facilities Layer

Le applicazioni ITS devono elaborare dati statici, temporanei e dinamici da altre stazioni ITS nell'area circostante alla stazione ospitante (veicolo e roadside). I dati rilevanti devono essere archiviati e gestiti nella Local Dynamic Map. Le informazioni nella LDM sono ricevute da messaggi pertinenti come i messaggi ITS CAM (Cooperative Awareness Message), i messaggi DENM (Decentralized Environmental Notification Message) e altri messaggi.

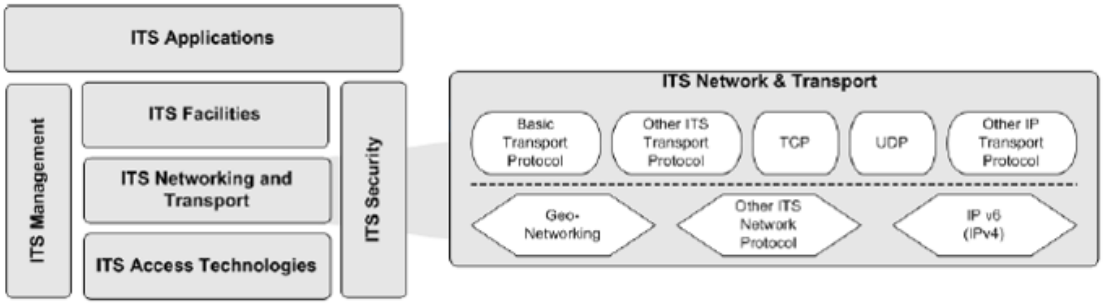

Figura 3.9: Overview del livello Network and Transport ITS [12].

Poiché il contenuto di questi messaggi viene utilizzato da diverse applicazioni, la plausibilità e i controlli di autorizzazione vengano eseguiti direttamente dalla LDM. Su richiesta di una specifica applicazione ITS, gli oggetti richiesti devono essere estratti e passati direttamente alle applicazioni appropriate in cui vengono elaborate. Le applicazioni richiedono meccanismi per aggiornare la LDM memorizzando le informazioni elaborate sugli oggetti richiesti nella LDM in modo che possano essere rese disponibili per altre applicazioni.

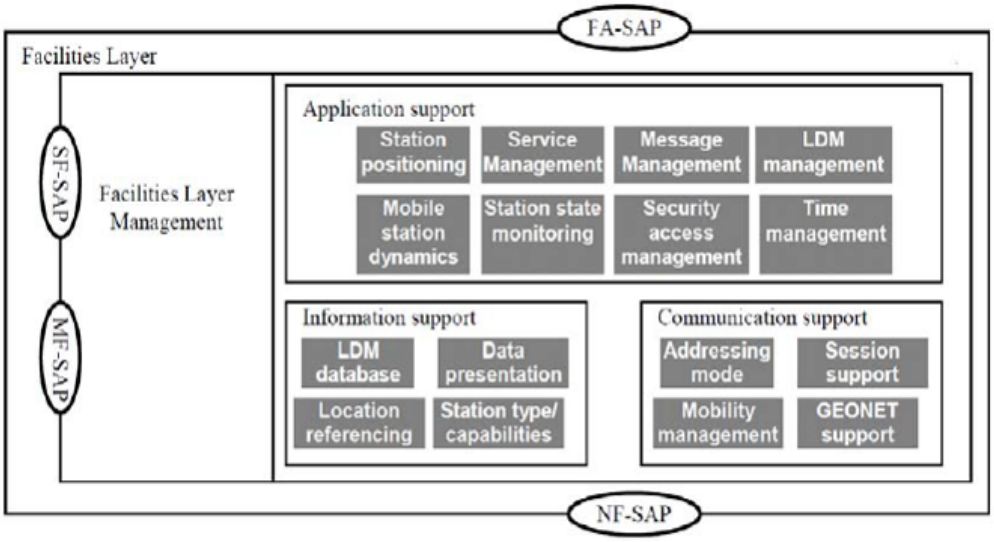

Figura 3.10: Overview del livello Facilities ITS [13].

Local Dynamic Map (LDM)

Le informazioni sull'ambiente locale sono essenziali nei sistemi ITS cooperativi. Le sue applicazioni richiedono informazioni sia su oggetti in movimento come altri veicoli nelle vicinanze sia su oggetti fermi come segnali stradali. Le informazioni

comuni richieste da diverse applicazioni possono essere gestite in una mappa dinamica locale. La Local Dynamic Map (LDM) è un archivio di dati concettuali situato all'interno di una stazione ITS contenente informazioni che sono rilevanti per il funzionamento sicuro delle applicazioni ITS. I dati possono essere ricevuti da un insieme di diverse fonti come veicoli, unità infrastrutturali, centri di traffico e sensori di bordo. L'accesso in lettura e scrittura ai dati tenuti all'interno della LDM è ottenuto tramite un'interfaccia. La LDM offre meccanismi per garantire un accesso sicuro ai dati. Pertanto la LDM è in grado di fornire a tutti informazioni sul traffico circostante e sull'infrastruttura RSU a tutte le applicazioni che lo richiedono. Per approfondimenti sulle strutture dati e sul funzionamento si rimanda allo standard[14].

Cooperative Awareness Basic Service

Il servizio di base CA è un'entità a livello facilities che gestisce il protocollo CAM. I messaggi CAM sono messaggi scambiati nella rete ITS tra ITS-S per creare e mantenere la consapevolezza reciproca e sostenere le prestazioni cooperative dei veicoli che utilizzano la rete stradale. Un CAM contiene informazioni sullo stato e sugli attributi dell'ITS-S di origine. Il contenuto varia a seconda del tipo di ITS-S. Per i veicoli ITS-S le informazioni sullo stato includono tempo, posizione, stato del movimento, sistemi attivati e le informazioni sugli attributi includono dati relativi alle dimensioni, al tipo di veicolo e al ruolo nel traffico stradale. Alla ricezione di un CAM l'ITS-S ricevente viene a conoscenza della presenza, del tipo e dello stato dell'ITS-S di origine. Le informazioni ricevute possono essere utilizzate dall'ITS-S ricevente per supportare diverse applicazioni ITS. Ad esempio, confrontando lo stato dell'ITS-S di origine con il proprio stato, un ITS-S ricevente è in grado di stimare il rischio di collisione con l'ITS-S originario e, se necessario, può informare il conducente del veicolo tramite l'HMI. Più applicazioni ITS possono fare affidamento sul servizio di base CA. Oltre al supporto delle applicazioni, è possibile utilizzare la consapevolezza di altri ITS-S acquisiti dal servizio di base CA a livello di rete e di trasporto per la diffusione di messaggi dipendenti dalla posizione. La generazione e la trasmissione di CAM è gestita dal servizio di base CA implementando il protocollo CAM [15].

I CAM vengono generati periodicamente con una frequenza controllata dal servizio di base CA nell'ITS-S di origine. La frequenza di generazione è determinata tenendo conto del cambiamento del proprio stato ITS-S, ad es. cambio di posizione o velocità e carico del canale radio.

Alla ricezione di una CAM, il servizio di base di CA rende il contenuto del CAM disponibile per le applicazioni ITS e/o per altre strutture all'interno dell'ITS-S ricevente, come una Local Dynamic Map (LDM).

Diffusione di un CAM

La comunicazione punto-multipunto deve essere utilizzata per la trasmissione di CAM. Il CAM deve essere trasmesso solo dagli ITS-S di origine agli ITS-S di ricezione situati nel raggio di comunicazione dell'ITS-S di origine. Un CAM ricevuto non deve essere inoltrato ad altri ITS-S.

Tempo di generazione di un CAM

Oltre alla frequenza di generazione di un CAM, anche il tempo necessario per la generazione CAM e la tempestività dei dati acquisiti per la costruzione del messaggio sono decisivi per l'applicabilità dei dati nelle ITS-S riceventi. Al fine di garantire la corretta interpretazione dei CAM ricevuti, ogni CAM deve avere un timestamp. Il tempo richiesto per una generazione CAM deve essere inferiore a *50 ms*. Il tempo richiesto per una generazione CAM si riferisce alla differenza di tempo tra il tempo in cui il CAM viene generato e il tempo in cui viene consegnato al livello di trasporto e rete.

Specifiche del formato di un CAM

Un CAM è composto da un'intestazione ITS PDU comune e da più container, che insieme costituiscono un CAM. L'intestazione ITS PDU è un'intestazione comune che include le informazioni sulla versione del protocollo, sul tipo di messaggio e l'ID dell'ITS-S di origine. Per i veicoli ITS-S una CAM deve comprendere un container di base e un container ad alta frequenza e può anche includere un container a bassa frequenza e uno o più altri contenitori speciali:

- il container di base include informazioni di base relative all'ITS-S di origine,
- il container ad alta frequenza contiene informazioni altamente dinamiche dell'ITS-S di origine,
- il container a bassa frequenza contiene informazioni statiche dell'ITS-S di origine,
- il container per veicoli speciali contiene informazioni specifiche sul ruolo del veicolo di origine ITS-S.

Per approfondimenti sulle strutture dati e sul funzionamento si rimanda allo standard[15].

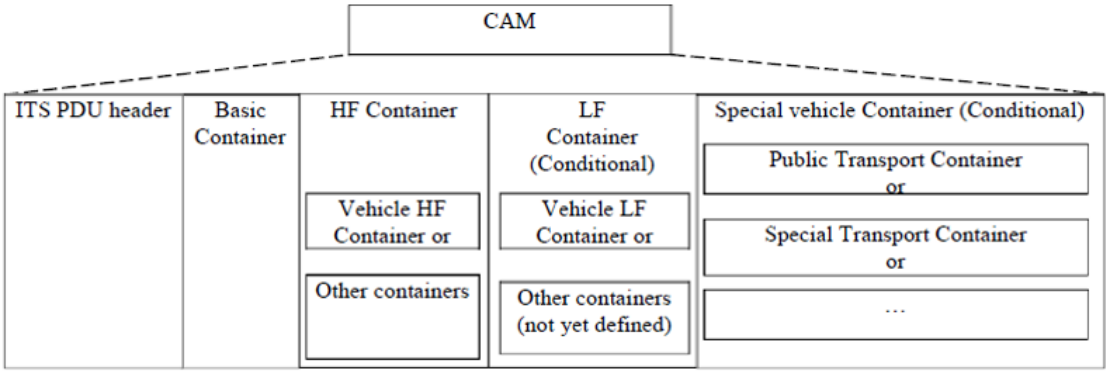

Figura 3.11: Struttura generale di un CAM [15].

3.2 Altre implementazioni dello standard V2X

Di seguito verranno illustrati alcuni dei progetti open-source esistenti che implementano alcune delle potenzialità del V2X, in modo da avere un confronto sulle possibilità di scelte implementative.

3.2.1 OpenC2X

OpenC2X è una piattaforma di prototipazione sperimentale e open-source sviluppata in C++ dai CSS Labs[16] che supporta lo standard ETSI-ITS G5; è costituita da moduli altamente indipendenti che possono essere facilmente estesi e riutilizzati ed è in esecuzione su sistemi Linux standard. Tutti i moduli comunicano tra loro tramite socket ZeroMQ [17].

OpenC2X è uno dei pochi progetti ad implementare le funzionalità del DCC, inoltre utilizza una LDM (Local Dynamic Map) come DB su cui fare lo storage delle informazioni dei veicoli, ma non ha implementato il GeoNetworking. I messaggi CAM vengono generati dal modulo corrispondente, inviati poi al DCC, e dopo essere stati codificati e impacchettati come payload di un pacchetto ethernet vengono inviati in broadcast; mentre i messaggi DENM come da standard sono messaggi event-driven, vengono quindi triggerati all'occorrenza di determinati eventi. OpenC2X attualmente supporta l'implementazione dei moduli di GPS e OBD-2 che forniscono una varietà di informazioni ai moduli superiori.

3.2.2 GeoNetworking

GeoNetworking[18] è un'implementazione dello stack ETSI GeoNetworking, implementata in java, basato su indirizzamento geografico e inoltro per comunicazioni point-to-point e point-to-multipoint con l'utilizzo di BTP come protocollo di livello

trasporto; supporta inoltre i messaggi di CAM e DENM e un modulo di GPS. Non è però implementata nessuna funzione di sicurezza.

Nell'implementazione si assume che ci sia un'entità separata (in esecuzione sulla stessa macchina o su una macchina diversa), che si occupa del livello data link, e la comunicazione tra questa entità e lo stack GeoNetworking è implementato in UDP.

3.2.3 Vanetza

Vanetza[19] è anch'essa un'implementazione, scritta in C++, open source dello standard ETSI-ITS G5 e fa parte di un lavoro di ricerca al Technische Hochschule Ingolstadt [20] che supporta i moduli di DCC, CAM, DENM, GN e BTP con l'implementazione anche di un livello per la security.

3.2.4 Conclusioni

Per concludere possiamo dire che tutte le implementazioni open-source dello standard V2X sono incomplete e si concentrano o sull'implementazione del DCC o del GeoNetworking; ma tutte hanno il supporto per la codifica/decodifica di messaggi in formato ASN.1 come da standard. Tra tutte quelle disponibili si è scelta OpenC2X in quanto scritta nello stesso linguaggio di IBR-DTN, e fornisce le principali funzionalità richieste all'integrazione con IBR-DTN, mentre i moduli mancanti, come ad esempio il routing, non sono necessari in quanto IBR-DTN ne possiede già una propria implementazione.

Capitolo 4

Convergence Layer V2X su IBR-DTN

4.1 Obiettivo

Lo sviluppo di questa tesi si propone di creare il prototipo di un'architettura di rete che, sfruttando le caratteristiche DTN, permetta di interconnettere dispositivi che utilizzano tecnologie di rete di natura eterogenea.

Più precisamente, si desidera che due dispositivi possano scambiarsi dei dati, incapsulandoli all'interno di bundle, senza dover tener conto dell'effettiva tecnologia di rete con cui avviene la trasmissione.

Un qualunque dispositivo, per poter agire da bundle node, ovvero, essere in grado di ricevere e trasmettere bundle, necessita di un Bundle Protocol Agent (BPA), uno strato software che estende lo stack di rete del dispositivo, consentendo ad esso di offrire i servizi previsti dal Bundle Protocol. Il software IBR-DTN offre supporto nativo alla suite di protocolli Internet (TCP/IP, UDP/IP), operanti su Ethernet o WiFi, allo standard IEEE 802.15.4 e allo standard Bluetooth.

Si è deciso di estenderne il codice, implementando uno strato software che consenta un'integrazione della tecnologia V2X.

4.2 Estensione di IBR-DTN

IBR-DTN offre supporto allo sviluppo del software tramite la definizione di una serie di classi interfaccia, ognuna pensata per modellare il comportamento di uno specifico modulo, perciò, ogni volta che si desidera aggiungere un nuovo modulo all'architettura IBR-DTN, è opportuno che esso implementi la classe interfaccia definita per la categoria di moduli a cui esso appartiene (convergence layer, ad esempio), e che quindi fornisca l'implementazione dei metodi definiti dall'interfaccia stessa.

Una volta creato un nuovo modulo, questo può interagire con il resto dei componenti presenti nell'architettura secondo due modalità: mediante chiamate sincrone o tramite il meccanismo basato su eventi.

In particolare, il modulo può scatenare eventi specifici al verificarsi di certe condizioni, allo scopo di "svegliare" i componenti interessati al suddetto evento e fornire ad essi tutte le informazioni necessarie alla sua gestione. A sua volta, il modulo può sottoscrivere il proprio interesse ad essere notificato nel momento in cui si verifica un certo evento, ed implementare una propria logica di gestione dello stesso. Dal punto di vista operativo, invece, il modulo può essere di due tipi: integrated, come modulo che rimane in ascolto (passivo) di specifici eventi e reagisce opportunamente, oppure come componente indipendent, eseguito in un thread a sè stante.

In quest'ultimo caso, solitamente, il modulo esegue una routine principale, in un ciclo infinito fino alla sua terminazione. Il software IBR-DTN supporta la realizzazione di entrambe le modalità operative, mettendo a disposizione apposite classi interfaccia, IndipendentComponent e IntegratedComponent, entrambe derivate dalla classe baseComponent.

4.3 Implementazione del supporto all'aggiunta di un nuovo stack protocollare

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

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

4.3.1 Convergence Layer

Nel momento in cui il core DTN desidera inviare un bundle, passa i dati del bundle e le informazioni sul nodo destinazione all'opportuno convergence layer, tramite una coda. Tuttavia, in generale, ogni convergence layer dovrebbe fornire almeno le seguenti funzionalità:

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

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

4.3.2 Discovery Agent

Il funzionamento di tale modulo è basato sull'invio periodico di beacon, dei messaggi molto piccoli utilizzati dai nodi per annunciare la propria presenza e scambiarsi tutte le informazioni necessarie alla comunicazione.

Ogni nodo annuncia il proprio EID all'interno di tali messaggi, in modo da permettere, ai nodi che lo riceveranno, di creare un mapping univoco tra l'EID e l'indirizzo del nodo mittente.

Il nodo dovrà specificare, all'interno dei beacon di scoperta, il protocollo che utilizza per il trasporto dei bundle e la porta sulla quale è in ascolto. Queste informazioni, insieme a EID e indirizzo, costituiscono quanto necessario all'instaurazione di una comunicazione tra i nodi.

4.3.3 Variazioni tra implementazione e standard

In questa sezione saranno presentati alcuni elementi usati in fase di implementazione che differiscono dallo standard ETSI ITS-G5.

- Modulo di routing: lo standard per il livello di rete e trasporto prevede l'utilizzo rispettivamente dei protocolli di GeoNetworkin e Basic Transport Protocol (BTP). In questa implementazione però questi protocolli non sono stati considerati in quanto IBR-DTN prevede già al suo interno diversi moduli di routing che una volta ricevuto un bundle cercano il perorso migliore per il suo inoltro.

  I moduli previsti da IBR-DTN sono:

  - Epidemic: in questo protocollo di routing ogni nodo replica il messaggio ad ogni altro nodo che incontra. I messaggi, quindi, verranno potenzialmente replicati su tutti i nodi della rete. La destinazione avrà una probabilità di ricezione del messaggio pari al 100%, tranne nei casi in cui ci si ritroverà in una rete formata da nodi con buffer limitati, che potrebbero quindi scartare pacchetti per via della loro saturazione o sufficientemente ampi, ma non in grado di replicare a tutti i bundle per via del limitato contatto. La perdita dei pacchetti, prima dell'arrivo a

destinazione può anche essere dovuta al raggiungimento della soglia del Time To Live (TTL).

  - PRoPHET: Probabilistic Routing Protocol using History of Encounters and Transitivity. Questo algoritmo è, anche esso, un algoritmo di routing basato sulla replica illimitata dei pacchetti; sfrutta la non casualità in una rete DTN, mantenendo una lista di probabilità di incontro di altri nodi ed utilizzando, appunto, queste probabilità per la spedizione e la replica dei messaggi ad un nodo avente una probabilità di consegna, verso la destinazione, migliore della sua.
  - MaxProp: anche conosciuto come Maximum Priority è come l'algoritmo precedente, basato sul calcolo di una probabilità per stabilire quale nodo debba prendere in carico il pacchetto da spedire verso la destinazione. Ogni nodo possiede una lista di nodi "vicini" con una probabilità di incontro. Ogni nodo memorizza, inoltre, per ogni destinazione, i relativi percorsi con il correlato costo di raggiungibilità, vengono utilizzate, quindi, le probabilità per la stima del percorso.I nodi "vicini" si scambieranno le informazioni immagazzinate sui percorsi disponibili, in modo da sfruttare queste informazioni nella fase di forwarding, per scegliere il percorso più adatto al raggiungimento della destinazione. I messaggi vengono ordinati secondo il costo più basso, all'interno del buffer di inoltro, e spediti in ordine dal meno al più costoso, che di conseguenza sarà scartato in caso di buffer pieno. I nodi destinazione inviano un messaggio di Acknowledge per certificare l'arrivo del messaggio e garantire ai nodi di inoltro la cancellazione del pacchetto dai propri database. Come per il protocollo precedente, si contraddistingue per: la sua più elevata complessità algoritmica, la replica del messaggio a nodi specifici e, pero, ad un rapporto di consegna più basso rispetto al primo.

- Architettura di un messaggio CAM: per ovviare alla creazione di un apposito Discovery Agent si è sfruttata la potenzialità del CAM andandolo a modificare in modo tale da poterlo sfuttare come messaggio di Beacon.

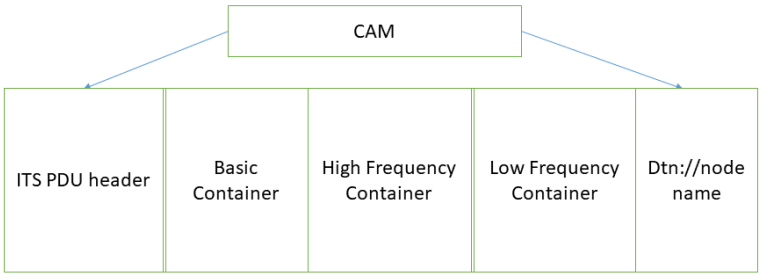

Figura 4.1: Schema di CAM modificato.

4.4 Implementazione

Il contenuto di questa sezione fornisce una descrizione dettagliata dell'architettura realizzata allo scopo di integrare, all'interno della suite IBR-DTN, il supporto alla tecnologia V2X.

4.4.1 Requisiti di sistema

La soluzione è concepita per sistemi Linux. Lo sviluppo è basato sull'utilizzo di una re-implementazione parziale e un adattamento di OpenC2X per l'integrazione in IBR-DTN. Primo passo è l'istallazione della libreria di supporto per ASN.1 tramite un CMakeFile, con i seguenti comandi:

- **make**
- **make install**
- **ldconfig**

che servono rispettivamente a compilare, installare e caricare le nuove librerie.

Le funzionalità accessorie desiderate, che tipicamente richiedono l'utilizzo di librerie esterne, devono essere abilitate in modo esplicito, modificando un'apposita sezione nel file di configurazione del processo demone.

4.4.2 Architettura

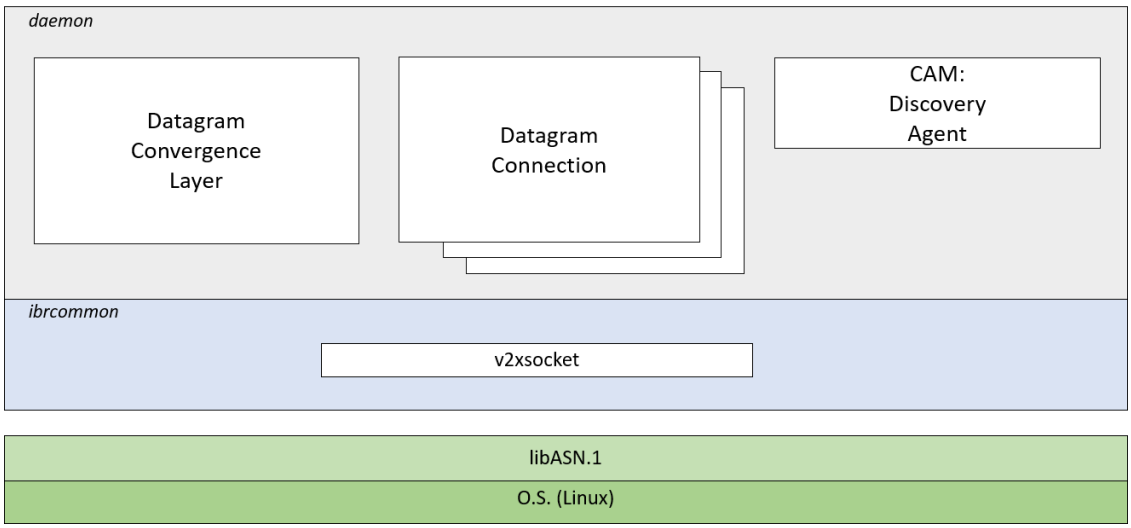

Figura 4.2: Moduli dell'architettura IBR-DTN integrata con V2X.

L'implementazione del supporto alla tecnologia V2X da parte di IBR-DTN, ha richiesto la creazione e l'integrazione di alcuni componenti, i principali sono:

- v2x_socket: classe di funzioni utilizzata per l'invio di messaggi V2X (CAM, DENM) da parte del DCC,
- DCC: classe che implementa il modulo [10] dello standard ,
- CAservice: classe che implementa il modulo [15] dello standard,
- v2xDatagramSocket: classe di funzioni utilizzata per l'invio dei bundle,
- v2xDatagramService: classe che si "appoggia" al Datagram Convergence Layer di IBR-DTN per la gestione di connessioni e invii dei bundle,
- DCCEvent: classe che eredita dalla classe Event di IBR-DTN utilizzata per la comunicazione tra i vari moduli V2X,
- V2xEvent:classe che eredita dalla classe Event di IBR-DTN utilizzata per la gestione dei timer del DCC.

Cooperative Awareness Basic Service

Il modulo CAservice (CAM) è un *Indipendent Component*, ed è responsabile della ricezione di CAM provenienti da altri nodi dal DCC e della generazione, codifica e invio di nuovi CAM. Questi, prima di essere inviati al DCC, vengono incapsulati in una struttura apposita per permettere di aggiungere ulteriori informazioni, come una marca temporale, la validità del messaggio e il tipo di priorità. Essendo il CAM un messaggio di Cooperative Awareness, questo modulo dispone di tutti i metodi necessari a calcolare velocità, posizione, altitudine e tutte le variabili che influiscono sul moto di un veicolo che saranno poi inserite nei container contenuti all'interno del pacchetto. Verrà ora mostrata la funzione di generazione e invio del CAM al DCC che è la ComponentRun(), eseguita in un ciclo infinito.

```
 //generate CAM and send to DCC
void Caservice::componentRun() {
    try{
    while(running){

        string serializedData;
        ibrdtn::data::DataPackage *data = new
        ibrdtn::data::DataPackage();

        CAM_t *cam = generateCam();

        vector <uint8_t> encodedCam =
        mMsgUtils->encodeMessage(&asn_DEF_CAM, cam);

        string strCam(encodedCam.begin(), encodedCam.end());
```

```
        IBRCOMMON_LOGGER_DEBUG_TAG(Caservice::TAG, 75) <<
        ``Encoded CAM size: '' <<
        to_string(encodedCam.size())<< IBRCOMMON_LOGGER_ENDL;

        data->id = messageID_cam;

        data->priority = ibrdtn::data::DataPackage::BE;

        int64_t currTime = dtn::utils::Clock::currentTime();
        data->createTime = currTime;
        data->validUntil = currTime +
        dconf.expirationTime * 1000 * 1000 * 1000;
        data->content = strCam;

        IBRCOMMON_LOGGER_DEBUG_TAG(Caservice::TAG, 75) <<
        ``Send new CAM of size'' << ``strCam.size()<<
        to DCC'' << IBRCOMMON_LOGGER_ENDL;

        dtn::core::V2xEvent::raise
        (EventV2xReceiver::SERVICE_DCC, ``CAM'', data);

        asn_DEF_CAM.free_struct(&asn_DEF_CAM, cam, 0);
        yield();
      }
    }
```

La codifica in fase di invio e la decodifica in fase di ricezione vengono fatte utilizzando i metodi di encodeMessage() e decodeMessage() della classe MessageUtils che utilizza funzioni della libreria libASN installata.

Decentralized Congestion Control

Il modulo DCC è un *Indipendent Component* ed è responsabile del controllo del carico sui canali e quindi del re-indirizzamento dei pacchetti a seconda della classe di traffico (TC) e dello stato del canale. Tramite il Channel Prober e il PktStatsCollector il DCC calcola il carico di ogni canale e in combinazione con i seguenti timer controlla l'Access Layer V2X:

- Measure Channel: restituisce il CBR ad intervalli di tempo prestabiliti tramite il Channel Prober,
- Measure Pkt Stats: calcola statistiche sui pacchetti che restitiusce al DCC ad intervalli di tempo prestabiliti tramite il PktStatsCollector,
- Update State: esegue allo scadere di ogni intervallo il check sul CBR e se necessario "triggera" il passaggio da uno stato (ad es. Relaxed) ad uno vicino (ad es. Active) del canale,

- Add Token: allo scadere di ogni intervallo aggiunge un token (= permesso di invio) alla queue dei pacchetti in attesa di essere inviati,

Inoltre il DCC si registra alla ricezione degli eventi (DCC Event) legati ai vari timer descritti prima e agli eventi V2X (V2X Event) legati alla ricezione/invio di dati dai livelli superiori (CAM). La gestione dei dati è bidirezionale, ossia il DCC può ricevere messaggi di CAM dal modulo Caservice ed inviarli in broadcast sull'hardware, tramite i socket RAW (v2x_socket), per raggiungere tutti i nodi vicini, o può ricevere un pacchetto broadcast dai nodi vicini e mandarlo al modulo corrispondente; ma il compito principale del DCC è quello di mettersi in attesa sulla coda contenete i dati da inviare, ed inviarli appena possibile, cioè quando c'è almeno un pacchetto in coda e c'è almeno un token per l'invio.

```
void DCC::componentRun(){

    try{
    while(1){
    Channels::t_access_category
    ac = AccessCategoryQueue.poll();

    while(ibrdtn::data::DataPackage
    *data = mBucket[ac]->dequeue()){

    int64_t nowTime = dtn::utils::Clock::currentTime();
    if (data->validUntil >= nowTime) {
    //message still valid
        setMessageLimits(data);
        string byteMessage;
        byteMessage = data->content;
        mSenderToHw->send(&byteMessage, ac);
        IBRCOMMON_LOGGER_DEBUG_TAG(DCC::TAG, 25)
        << ``AC '' << to_string(ac) << ``:Sent data ''
        << to_string(data->id) << `` to HW -> queue lenght:''
        << to_string(mBucket[ac]->getQueuedPackets())<<
        ``, tokens:''<<to_string(mBucket[ac]->availableTokens)
        << IBRCOMMON_LOGGER_ENDL;
        delete data;
    } else {     //message expired
         mBucket[ac]->increment();
         IBRCOMMON_LOGGER_DEBUG_TAG(DCC::TAG, 90) <<
         ``message ( packet '' << to_string(data->id)<<
         `` ) expired''<< IBRCOMMON_LOGGER_ENDL;
         delete data;
          }
    }
     yield();
}
```

Leaky Bucket La coda su cui il DCC è in attesa e che contiene i pacchetti da inviare è implementata con una classe template, il Leaky Bucket. Oltre ai classici metodi associati ad una coda il Leaky Bucket deve anche gestire i token disponibili, ossia i "permessi" per l'invio dei pacchetti.

Quindi come si può notare dal codice di invio del DCC il Leaky Bucket deve implementare due metodi per inviare, uno che decrementi il numero di token, se disponibili (decrement), uno che estragga e ritorni il primo pacchetto dalla coda (dequeue), e due speculari per ricevere, increment() per aggiungere un token e enqueue() per aggiungere un pacchetto in coda.

Verrà ora illustrata la loro implementazione.

```
/**
 * Adds a token to the bucket if not full.
 * @return true if the token was added successfully
 */
bool increment() {
        mutex_bucket.lock();
        bool ret = false;
        if(availableTokens < maxBucketSize) {
                availableTokens++;
                ret = true;
        } else {
                ret = false;
        }
        mutex_bucket.unlock();
        return ret;
}

/**
 * Adds a packet to the queue if not full.
 * @param p Packet to be enqueued
 * @param validUntil Life time of the packet
 * @return true if the packet was enqueued successfully
 */
bool enqueue(T* p, int64_t validUntil) {
        mutex_queue.lock();
        bool ret = false;
        if(queue.size() < queueSize) {
                queue.push_back(std::make_pair(validUntil, p));
                ret = true;
        } else {
                ret = false;
        }
        mutex_queue.unlock();
        return ret;
}
```

```
/**
 * Removes a token from the bucket if there is any.
 * @return true if a token was removed successfully
 */
bool decrement() {
        mutex_bucket.lock();
        bool ret = false;
        if(availableTokens > 0) {
                availableTokens--;
                ret = true;
        } else {
                ret = false;
        }
        mutex_bucket.unlock();
        return ret;
}

/**
 * Removes and returns the first packet from the queue if
 there is any and there is a token available.
 * @return The first packet (or NULL if no packet or token
 available)
 */
T* dequeue() {
    mutex_queue.lock();
    T* ret = NULL;
    if(queue.size() > 0) {
        if(decrement()) {
            ret = queue.front().second;
            queue.pop_front();
        } else {
            ret = NULL;
        }
    } else {
        ret = NULL;
    }
    mutex_queue.unlock();
    return ret;
}
```

Timer Update State Il trigger della funzione per l' aggiornamento dello stato del canale è effetuato da un timer con la seguente funzione:

```
 state_mutex.lock();
double clMinInTimeUp = mChannelLoadInTimeUp.min();
//minimal channel load during TimeUp-interval
```

```
double clMaxInTimeDown = mChannelLoadInTimeDown.max();
//TimeDown-interval > TimeUp-interval

state_mutex.unlock();

if ((mCurrentStateId == STATE_RELAXED) &&
    (clMinInTimeUp >= dconf.NDL_minChannelLoad))
    //relaxed -> active1
    setCurrentState(STATE_ACTIVE1);
else if ((mCurrentStateId == STATE_RESTRICTED) &&
        (clMaxInTimeDown < dconf.NDL_maxChannelLoad))
        //restricted -> active1
    setCurrentState(STATE_ACTIVE1);
else if ((mCurrentStateId != STATE_UNDEF && mCurrentStateId
!= STATE_RELAXED && mCurrentStateId != STATE_RESTRICTED)
&& (clMaxInTimeDown < dconf.NDL_minChannelLoad))
//active -> relaxed
    setCurrentState(STATE_RELAXED);
else if ((mCurrentStateId != STATE_UNDEF && mCurrentStateId
!= STATE_RELAXED && mCurrentStateId != STATE_RESTRICTED) &&
        (clMinInTimeUp >= dconf.NDL_maxChannelLoad))
        //active -> restricted
    setCurrentState(STATE_RESTRICTED);
```

Come si può notare dal codice precedente il cambiamanto di stato è consentito solo da uno stato ad un altro adiacente.

Channel Prober Il Channel Prober è composto da due thread; uno funge da timer e ogni qual volta scade il timeout associato al timer, viene chiamata una funzione che invia su un socket netlink dei messaggi netlink 802.11 che serviranno poi al thread principale a verificare le condizione del canale e a calcolarne il carico.

La funzione di invio è definita come segue:

```
int ChannelProber::send(uint8_t msgCmd, void *payload,
unsigned int length, int attrType, unsigned int seq,
int protocolId, int flags, uint8_t protocolVersion) {

int ret = 0;

// create a new Netlink message
nl_msg *msg = nlmsg_alloc();

// create message header
if (genlmsg_put(msg, 0, seq, protocolId, 0, flags, msgCmd,
protocolVersion) == NULL) {
        IBRCOMMON_LOGGER_DEBUG_TAG(ChannelProber::TAG, 25)<<
        ``Failed to create netlink header for the message''
```

```
        << IBRCOMMON_LOGGER_ENDL;
        return -1;
}

// add message attributes (=payload)
if (length > 0) {
        ret = nla_put(msg, attrType, length, payload);
        if (ret < 0) {
        IBRCOMMON_LOGGER_DEBUG_TAG(ChannelProber::TAG, 25) <<
        ``Error when adding attributes''
        << IBRCOMMON_LOGGER_ENDL;
                return ret;
        }
}

ret = nl_send_auto(mSocket, msg);
if (ret < 0) {
        IBRCOMMON_LOGGER_DEBUG_TAG(ChannelProber::TAG, 25) <<
        ``Error when sending the message'' <<
        IBRCOMMON_LOGGER_ENDL;
}
nlmsg_free(msg);

return ret;
}
```

Questa funzione ritorna un valore intero corrispondente al numero di byte inviati.

Il thread principale invece nella funzione di init(), fa partire il timeout del timer e si definisce una callback che alla ricezione di qualunque messaggio sul socket netlink esegua la funzione receivedNetlinkMsg(). Viene ora illustrato come la funzione viene chiamata.

```
// initialize socket now
mSocket = nl_socket_alloc();
if (mSocket == NULL) {
        IBRCOMMON_LOGGER_DEBUG_TAG(ChannelProber::TAG, 25)
        << ``Could not allocate netlink socket'' <<
        IBRCOMMON_LOGGER_ENDL;
        exit(1);
}

// disable sequence number checking. We want to receive
notifications.
nl_socket_disable_seq_check(mSocket);

ret = nl_socket_modify_cb(mSocket, NL_CB_VALID,
```

```
NL_CB_CUSTOM, ChannelProber::receivedNetlinkMsg, this);
if (ret != 0) {
        IBRCOMMON_LOGGER_DEBUG_TAG(ChannelProber::TAG, 25)
        << ``Failed to modify the callback''
        << IBRCOMMON_LOGGER_ENDL;
        exit(1);
}
// connect socket. Protocol: generic netlink
ret = genl_connect(mSocket);
if (ret != 0) {
        IBRCOMMON_LOGGER_DEBUG_TAG(ChannelProber::TAG, 25)
        << ``Connection to netlink socket failed''
        << IBRCOMMON_LOGGER_ENDL;
        exit(1);
}

// resolve mNl80211Id
mNl80211Id = genl_ctrl_resolve(mSocket, ``nl80211'');
if (mNl80211Id < 0) {
        IBRCOMMON_LOGGER_DEBUG_TAG(ChannelProber::TAG, 25)
        << ``Could not get NL80211 id''
        << IBRCOMMON_LOGGER_ENDL;
        exit(1);
}
startTimer();
```

Il compito della funzione receivedNetlinkMsg() è di calcolare le info del canale attualmente in uso quali *total active time, noise e total busy time* che sono quelli necessari per aggiornare i dati e calcolare il carico sul canale che poi sarà inviato al DCC che a seconda della sua politica algoritmica deciderà o meno se effettuare un cambiamento di stato; c'è poi la possibilità di calcolare anche ulteriori parametri opzionali come *il tempo totale di trasmissione e quello di ricezione*.

Di seguito una breve illustrazione della sua implementazione.

```
int ChannelProber::receivedNetlinkMsg(nl_msg *msg, void *arg)
{
ChannelProber *cp = (ChannelProber *) arg;

struct nlattr *tb[NL80211_ATTR_MAX + 1];
struct genlmsghdr *gnlh = (genlmsghdr *)
nlmsg_data(nlmsg_hdr(msg));
struct nlattr *sinfo[NL80211_SURVEY_INFO_MAX + 1];
char dev[20];

uint64_t total_time = 0, busy_time = 0;
uint32_t channel = 0;
int8_t noise;
```

```
nla_parse(tb, NL80211_ATTR_MAX, genlmsg_attrdata(gnlh, 0),
genlmsg_attrlen(gnlh, 0), NULL);

if_indextoname(nla_get_u32(tb[NL80211_ATTR_IFINDEX]),dev);

if (!tb[NL80211_ATTR_SURVEY_INFO]) {
        IBRCOMMON_LOGGER_DEBUG_TAG(ChannelProber::TAG, 25)
        << ''ChannelProber: survey data missing!''
        <<      IBRCOMMON_LOGGER_ENDL;
        return NL_SKIP;
}
static struct nla_policy
survey_policy[NL80211_SURVEY_INFO_MAX + 1] = {};

if (nla_parse_nested(sinfo, NL80211_SURVEY_INFO_MAX,
tb[NL80211_ATTR_SURVEY_INFO], survey_policy)) {
        IBRCOMMON_LOGGER_DEBUG_TAG(ChannelProber::TAG, 25)
        << ''Failed to parse nested attributes''
        << IBRCOMMON_LOGGER_ENDL;
        return NL_SKIP;
}

//If this info is not about the channel in use, then skip
if (!sinfo[NL80211_SURVEY_INFO_IN_USE]) {
        return NL_SKIP;
}

//Current channel in use
if (sinfo[NL80211_SURVEY_INFO_FREQUENCY]) {
        channel =
        nla_get_u32(sinfo[NL80211_SURVEY_INFO_FREQUENCY]);
}

//Noise
if (sinfo[NL80211_SURVEY_INFO_NOISE]) {
        noise = (int8_t)
        nla_get_u8(sinfo[NL80211_SURVEY_INFO_NOISE]);
}

//Total active time
if (sinfo[NL80211_SURVEY_INFO_CHANNEL_TIME]) {
        total_time = nla_get_u64
        (sinfo[NL80211_SURVEY_INFO_CHANNEL_TIME]);
}
```

```
//Total busy time
if (sinfo[NL80211_SURVEY_INFO_CHANNEL_TIME_BUSY]) {
        busy_time = nla_get_u64
        (sinfo[NL80211_SURVEY_INFO_CHANNEL_TIME_BUSY];
}

//Do we need info about extension channel?
if (sinfo[NL80211_SURVEY_INFO_CHANNEL_TIME_EXT_BUSY]) {
        // cout <<``\textension channel busy time:\t''
        //              <<
        nla_get_u64(sinfo
        [NL80211_SURVEY_INFO_CHANNEL_TIME_EXT_BUSY])
        //              << `` ms'' << endl;
}

//Total receiving time
if (sinfo[NL80211_SURVEY_INFO_CHANNEL_TIME_RX]) {
        //cout << ``\tchannel receive time:\t\t''
        //<< nla_get_u64(sinfo
        [NL80211_SURVEY_INFO_CHANNEL_TIME_RX])
        //      << `` ms'' << endl;
}

//Total transmitting time
if (sinfo[NL80211_SURVEY_INFO_CHANNEL_TIME_TX]) {
        //cout << ``\tchannel transmit time:\t\t''
        //<< nla_get_u64(sinfo
        [NL80211_SURVEY_INFO_CHANNEL_TIME_TX])
        //      << `` ms'' << endl;
}

//Update statistics
uint64_t busy_time_diff = busy_time -
cp->mWifi->load.busyTimeLast;
uint64_t total_time_diff = total_time -
cp->mWifi->load.totalTimeLast;
double load = (double) (((100 * busy_time_diff) /
total_time_diff)) / 100.0;
cp->mWifi->load.mtx.lock();
cp->mWifi->load.load = load;
cp->mWifi->load.totalTimeLast = total_time;
cp->mWifi->load.busyTimeLast = busy_time;
cp->mWifi->load.noise = noise;
cp->mWifi->load.mtx.unlock();
return NL_SKIP;
}
```

RAW V2X Socket

I socket utilizzati sono definiti con la seguente istruzione:

```
mSocket = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

a cui si passa come family **AF_PACKET**, il tipo di socket , come detto, è **RAW** e il flag **htons(ETH_P_ALL)** sta ad indicare che vengono presi tutti i pacchetti in transito sulla rete, come uno sniffer. Dopo aver sperimentato si è inoltre visto che la lettura da un socket di questo tipo non "consuma" i dati, nel senso che due diversi socket che effettuano una lettura dal file descriptor associato, leggeranno gli stessi dati. Altri socket come gli stream socket e i datagram socket ricevono dati dal livello di trasporto che non contengono intestazioni ma solo il payload, ciò significa che non sono disponibili informazioni sull'indirizzo IP di origine e l'indirizzo MAC. Lo scopo di un socket raw è assolutamente diverso. Un socket raw consente a un'applicazione di accedere direttamente ai protocolli di livello inferiore, il che significa che un socket raw riceve pacchetti non estratti [21].

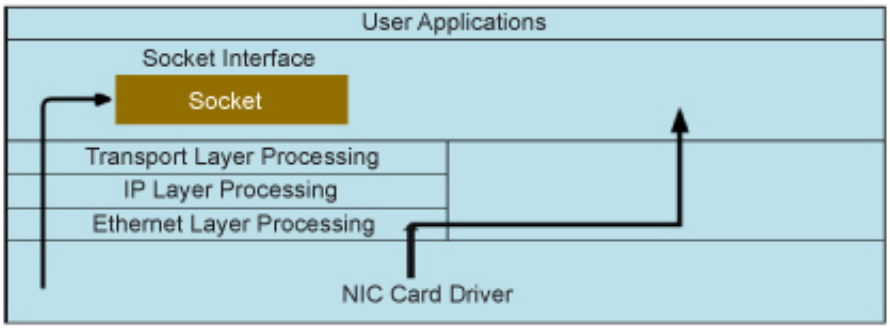

Figura 4.3: Schema di funzionamento di un RAW socket rispetto ad altri tipi di socket [21].

Non è necessario fornire la porta e l'indirizzo IP a un socket "grezzo", a differenza del caso di socket stream e datagram. Data la loro particolare struttura questo tipo di socket a differenza di un socket comune non supporta i metodi listen() e accept() e connect(), utilizzati per restare in ascolto di una connessione (listen()), effettuare una richiesta di connessione (connect()) e accettare un'eventuale richesta di connessione in arrivo (accept()).

Questi socket vengono infatti utilizzati in due punti diversi del programma: il DCC li utilizza per mandare/ricevere messaggi broadcast e per effettuare la Neighborg Discovery; inoltre, con le dovute modifiche, il V2X Datagram Service li utilizza per inoltrare i bundle verso specifiche destinazioni.

Essendo socket RAW i pachetti sono gestiti a livello MAC quindi i dati da inviare diventano semplicemente il payload di un pacchetto ethernet.

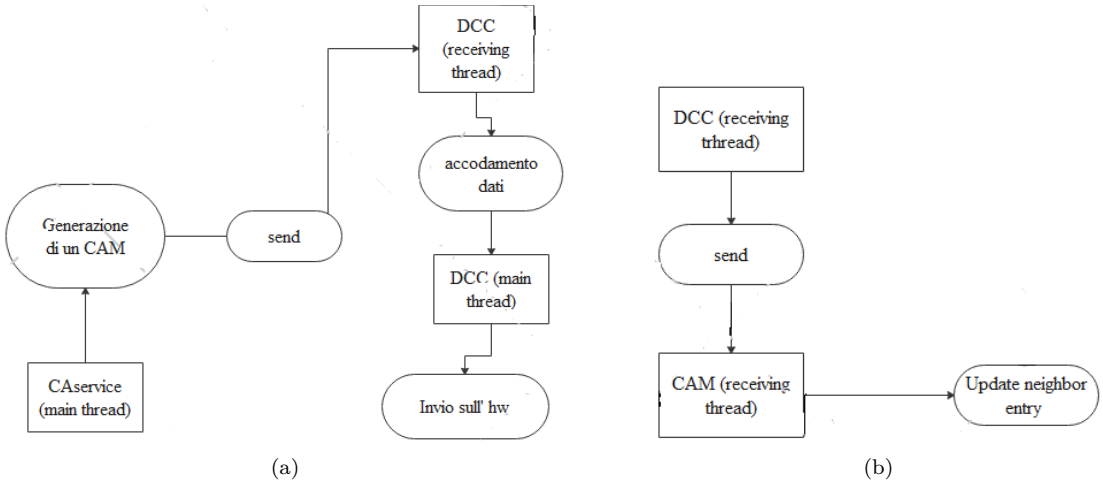

Figura 4.4: Schema della fase di invio (a) e ricezione (b) di un CAM.

V2X Datagram Service e Datagram Convergence Layer

Per l'implementazione del Convergence Layer si è seguito un approccio diverso dal crearne uno ad-hoc per il V2X, infatti si è sfruttato il Datagram Convergence Layer presente in IBR-DTN su cui si è andati ad istanziare un service, il V2X Datagram Service (Figura 4.5). Il Datagram Convergence Layer è istanziato dal modulo ConnectionManager, invocato all'interno del processo demone (NativeDaemon) per configurare e mettere in esecuzione i singoli convergence layer.

Il V2X Datagram Service si occupa di definire i parametri di comunicazione come MTU, numero massimo di ripetizioni e di sequence number per i pacchetti e tipo di controllo del flusso; a tal proposito IBR-DTN supporta tre diversi tipi di controllo del flusso:

- NONE: in questo tipo di controllo del flusso il nodo mittente invia tutto il bundle, eventualmente frammentato; il nodo ricevitore si occupa solo di ricevere ed eventualmente riassemblare i frammneti ricevuti, senza però inviare nessun tipo di messaggio (ack) al mittente per segnalare la corretta ricezione di tutti i dati. Con questo tipo di controllo di flusso, ovviamente, nel caso in cui il bundle da inviare sia superiore alla dimensione massima di un singolo pacchetto (MTU), e sia quindi necessario frammentare il pacchetto, l'eventuale perdita di uno o più frammenti non sarebbe individuata e il pacchetto arriverebbe alla detinazione corrotto.

- SLIDING_WINDOW: in questo tipo di controllo di flusso il nodo mittente invia i frammenti del pacchetto a blocchi in numero pari alla dimensione della finestra, ad ogni finestra viene inoltre associato un timeout che viene fatto partire nell'istante in cui parte il primo frammento; questo timeout serve ad

```
class ConvergenceLayer
{
        public:

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

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

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

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

        virtual void resetStats();

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

Figura 4.5: Interfaccia Convergence Layer di IBR-DTN.

attendere tutti gli ack corrispondenti ad ogni frammento inviato presente nella finestra. Se allo scadere del timeout uno o più ack non sono stati ricevuti la finestra resta fissa e vengono re-inviati gli n frammenti presenti nella finestra; in caso contrario la finestra di invio trasla in avanti per inviare gli n frammenti successivi. Con questo tipo di controllo di flusso, a differenza del precedente, si ha la certezza che il pacchetto, frammentato o meno, arrivi correttamente alla destinazione; ma dato il tipo del protocollo che non è connection-oriented e quindi una probabilità non trascurabile di perdita di alcuni pacchetti di dati o di ack, le prestazioni potrebbero diminuire, specialmente nel caso in cui si inviasse un pacchetto di grandi dimensioni, formato quindi da un elevato numero di frammenti da inviare.

- STOP_AND_WAIT: in questo tipo di controllo di flusso, che è quello scelto per questa implementazione, il nodo mittente ogni volta che invia un pacchetto, frammentato o meno, aspetta la ricezione dell'ack corrispondente prima di procedere con l'invio del frammento seguente; per fare ciò nell'istante in cui parte l'invio viene fatto partire un timeout, se questo timeout scade il mittente prova a re-inviare lo stesso frammento, fino ad un numero massimo di volte uguale ad un parametro configurabile. Con questo tipo di controllo di flusso, come nel caso precedente l'arrivo del pacchetto a destinazione è garantito, e la diminuzione di prestazioni a fronte della stessa perdita di pacchetti del caso

precedente è migliore in quanto il re-invio avviene solo per il pacchetto di cui non si è ricevuto l'ack e non dell'intera finestra di trasmissione.

Inoltre il V2X Datagram Service si occupa di avviare i moduli V2X (CAM E DCC) e di far partire la Neighborg Discovery; ma sopratutto di gestire l'invio e la ricezione dei bundle.

Datgram Convergence Layer

Il modulo Datagram Convergence Layer, una volta istanziato, riceve dal ConnectionManager due parametri, caricati dal file di configurazione (se presente, altrimenti viene usata la configurazione di default): l'interfaccia di rete e il canale su cui rimanere in ascolto. Questi parametri sono passati tramite chiamata sincrona del metodo *add()*, che a sua volta invoca il metodo *getConnection()* del Datagram Convergence Layer. Un modulo che desideri agire da Convergence Layer, deve fornire, come minimo, l'implementazione dei metodi getDiscoveryProtocol() e queue().

Il metodo getDiscoveryProtocol() ritorna un identificativo del protocollo implementato dal Convergence Layer. Ogni volta che i moduli del core DTN desiderano avviare il trasferimento di un bundle verso un nodo vicino, confrontano l'identificativo di protocollo ottenuto dalla fase di discovery con quello ritornato dal metodo getDiscoveryProtocol(), al fine di attivare il Convergence Layer opportuno per il trasferimento.

Più precisamente, per attivare il trasferimento, viene invocato il metodo queue() del Convergence Layer selezionato, che inserisce il bundle in attesa di essere trasferito all'interno di una coda. Il metodo riceve come parametri un oggetto di tipo Node, che identifica il nodo destinatario del bundle, e un oggetto di tipo BundleTransfer, che incapsula un EID e un riferimento al bundle da trasferire.

Il modulo Datagram Convergence Layer è stato concepito come componente indipendent, da eseguire in un thread a sè stante. Come tutti i componenti di questo tipo, che implementano l'interfaccia IndipendentComponent, il modulo deve implementare i metodi ComponentUp(), ComponentDown() e ComponentRun(). I metodi ComponentUp() e ComponentDown() sono invocati, rispettivamente, subito dopo il costruttore e immediatamente prima del distruttore del convergence layer. Il metodo ComponentUp() chiama la bind() del V2X Datagram Service, si registra alla ricezione di un NodeEvent e infine registra il V2X Datagram Service per la gestione dei beacon di scoperta. In maniera duale, la ComponentDown() chiama i metodi per la cancellazione della registrazione agli eventi.

Il metodo ComponentRun() implementa la routine principale eseguita dal thread all'interno di un loop infinito. In particolare, il thread del Datagram Convergence Layer inizializza e fa partire il sender thread; poi rimane in attesa sulla action queue, fin quando non sarà disponibile e pronto all'esecuzione un nuovo task.

Nel momento in cui giunge una richiesta di connessione da parte di un nodo client, cioè quando viene ricevuto qualcosa (ack,nack,segmento dati), viene creata

una nuova Datagram Connection associata all'indirizzo ricevuto, e viene chiamato il metodo queue() della Datagram Connection che scrive il dato in coda allo streamBuffer e attiva la callback per la ricezione dell'ack corrispondente.

Datagram Connection

Di base, ogni connessione tra due bundle node inizia con uno scambio di Stream Contact Header, una struttura dati contenente l'EID del nodo mittente ed una serie di flag che abilitano o meno delle funzionalità opzionali per il trasferimento dati. Nel caso specifico di Datagram Connection, tali flag forzano la richiesta di acknoledgement, con cui il nodo locale chiede che sia confermata la ricezione dei segmenti di dati, da parte del destinatario.

La Datagram Connection creata, è eseguita come thread detached, ed è incaricata di gestire la ricezione dei bundle in arrivo sul socket. Poichè due bundle node non possono scambiare direttamente oggetti tra di loro, i bundle devono essere codificati in una rappresentazione binaria, secondo un processo noto come serializzazione. IBR-DTN fornisce delle classi che realizzano le funzionalità di serializzazione e deserializzazione, che permettono di leggere e scrivere bundle su degli streamBuffer in maniera molto semplice. Dunque, la parte di ricezione di Datagram Connection riceve dati in ingresso sulla connessione e provvede alla loro deserializzazione, in modo da ottenere il bundle originario (o un frammento di esso, nel caso in cui sia stata eseguita la frammentazione).

Dopo aver eseguito dei controlli minimali sui campi del bundle ottenuto, quest'ultimo attraversa delle tabelle di filtraggio. Questa operazione risulta sensata solamente nel caso in cui siano abilitate le estensioni del Bundle Security Protocol (BSP), che permettono di effettuare controlli di integrità ed autenticazione a livello di Bundle Protocol, infatti, nel momento in cui le estensioni del BSP sono disabilitate, tutti i bundle in ingresso sono accettati automaticamente. In caso contrario, a seconda dell'esito del filtraggio, il bundle ricevuto può essere scartato o accettato.

In caso di accettazione, il bundle viene iniettato nel Bundle Core, il quale, dopo averlo memorizzato nel modulo di storage, scatenerà un opportuno evento (BUNDLE_RECEIVED) per attivare i moduli di routing, incaricati del suo inoltro. Dall'altro lato, la trasmissione dei bundle è gestita tramite un thread separato, detto sender thread, istanziato dalla Datagram Connection stessa. Tale thread rimane bloccato sulla coda dei bundle in attesa di essere trasferiti. Nel momento in cui un bundle deve essere inviato, dalla coda ne viene prelevato il riferimento, mentre il bundle vero e proprio viene recuperato dal modulo di storage.

A questo punto, valgono le stesse considerazioni sulle tabelle di filtraggio fatte per la fase di ricezione. Una volta che il bundle da trasferire viene accettato, questo viene serializzato e trasmesso verso il nodo remoto sul flusso di comunicazione. Nel caso in cui sia abilitata la frammentazione, durante questa fase non viene inviato

l'intero bundle, ma i frammenti di esso, uno per volta, tenendo traccia dell'offset rispetto al bundle originale.

4.5 Neighborg Discovery

In generale, con "vicinato di un nodo" si intende l'insieme dei dispositivi collocati all'interno del raggio d'azione del nodo stesso. Tale concetto si traduce, in un'architettura di tipo Delay-Tolerant Network, come l'insieme dei nodi, partecipanti alla DTN, raggiungibili direttamente dal nodo di riferimento.

Per la scoperta del vicinato, si è realizzato un meccanismo ad-hoc basato sul fatto che il DCC ad intervalli di tempo regolari mandi un messaggio CAM in broadcast. Si è quindi deciso di sfruttare questi messaggi anche per la fase di Discovery per permettere ad ogni nodo di annunciare costantemente, e ad intervalli di tempo regolari, la propria presenza sulla rete e scambiare tutte le informazioni utili alla comunicazione; facendo diventare così i messaggi CAM anche una sorta di messaggio di beaconing.

Tale meccanismo è stato implementato all'interno della classe Caservice. Per eseguire tale operazione però è stato necessario modificare la struttura del CAM che, come definito dallo standard [15], è composto da vari container che contengono informazioni logistiche e ambientali sul veicolo, (velocità, posizione, altitudine, dimensione del veicolo) quindi è stato necessario aggiungere al file un ulteriore campo opzionale denominato content (OCTET_STRING) usato per trasferire le informazioni necessarie alla comunicazione, cioè indirizzi MAC e nome dtn del nodo.

```
/* CAM */
typedef struct CAM {
    ItsPduHeader_t header;
    CoopAwareness_t cam;
    OCTET_STRING_t *content; /* OPTIONAL */

    /* Context for parsing across buffer boundaries*/
    asn_struct_ctx_t _asn_ctx;
}CAM_t;
```

In pratica ogni volta che viene generato un CAM si inserisce al suo interno il nome dtn del nodo corrente nel campo content, e dopo la sua codifica viene inviato al DCC che lo invia in broadcast sulla rete, quando il DCC di un nodo vicino riceve questo messaggio effettua la raise di un evento per spedire al proprio modulo Caservice il messaggio ricevuto (CAM) più l'indirizzo MAC del mittente.

```
dtn::core::V2xEvent::raise(EventV2xReceiver::SERVICE_CAM, ''
   DCC'',
std::make_pair(serializedData, senderMac));
```

Dato che il CAM non può contenere informazioni diverse da quelle contenute nei container che lo compongono, è stato necessario creare una struttura dati apposita (classe DataPackage) per fare da wrapper ai messaggi CAM e inserire in essa anche altre informazioni fondamentali per la gestione dei messaggi sui vari canali e per apporre ai messaggi una marca temporale (timestamp) che ne indicasse la validità e la durata nel tempo.

```
class DataPackage{

            public:

            DataPackage(){};
            ~DataPackage(){};

            enum Priority {
                BK = 1,
                BE = 0,
                VI = 4,
                VO = 6
            };

            int id;
            Priority priority;
            int64_t createTime;
            int64_t validUntil;
            double txPower;
            double bitRate;
            std::string content;

        };
```

Una volta ricevuto il messaggio dal DCC, in un thread dedicato, il modulo Caservice, effettua a sua volta la raise di un altro evento che serve a passare al thread principale il nome dtn del vicino e il suo indirizzo MAC, che vengono utilizzati per effettuare la fase di annuncio e aggiornamento del vicinato come segue:

- si crea un EID identificativo del nodo a partire dal nome dtn del nodo stesso,
- si crea un URI contenente il tipo di nodo, una stringa identificativa composta da indirizzo e canale, un timeout e un livello di priorità,
- si crea un oggetto Node a partire dall' EID sopra definito,
- viene chiamato il metodo add() sul nodo per aggiungergli l'URI corrispondente,
- il nuovo nodo scoperto viene annunciato al Connection Manager tramite il metodo updateNeighbor(node).

```
else if(evt.getReceiver() == SERVICE_NAME){
    string envelope = evt.getEnvelope();
    std::pair<std::string,std::string> data = evt.getPair();

    string NodeName = data.first;
    string senderMac = data.second;

    char neighbor_eid[MAX_EID_SIZE + 1];
    snprintf(neighbor_eid, 32, ``%s'', NodeName.c_str());

    // create a uri for the discovered node
    const dtn::core::Node::URI uri
    (dtn::core::Node::NODE_DISCOVERED,
    dtn::core::Node::CONN_DGRAM_V2X,
    std::string(``addr='' + senderMac + ``;channel=0;''),
    _timeout, _priority);
    const dtn::data::EID neighborEID(neighbor_eid);

    // create the EID for the discovered node
    dtn::core::Node n(neighborEID);
    n.add(uri);

    // announce the discovered node to ConnectionManager
    dtn::core::BundleCore::getInstance()
    .getConnectionManager().updateNeighbor(n);
```

4.6 Trasmissione di un Bundle

Ogni volta che il modulo Datagram Convergence Layer riceve una richiesta di connessione da parte di un nodo, viene creata un'istanza di Datagram Connection, che riceve una serie di parametri, tra cui un riferimento al nodo remoto e il socket RAW aperto per poter comunicare con esso. L'oggetto così creato, eseguito come thread detached, è incaricato di gestire la ricezione dei bundle in arrivo sul socket. Poichè due bundle node non possono scambiare direttamente oggetti tra di loro, i bundle devono essere codificati in una rappresentazione binaria, secondo un processo noto come serializzazione. IBR-DTN fornisce delle classi che realizzano le funzionalità di serializzazione e deserializzazione, che permettono di leggere e scrivere bundle su degli streambuffer in maniera molto semplice tramite i metodi di underflow() per la lettura e overflow() per la scrittura.

La trasmissione dei bundle è gestita tramite un thread separato, detto sender thread, istanziato dalla Datagram Connection stessa; tale thread rimane bloccato sulla coda dei bundle in attesa di essere trasferiti. Supponendo che il bundle da trasferire sia stato accettato, questo viene serializzato e trasmesso al V2X Datagram

Service che anche in questo caso come per la Neighbor Discovery utilizza il CAM modificato e "wrappato" in un DataPackage andando ad incapsulare il bundle da inviare nel suo campo content; il service poi passa il CAM codificato contenente il bundle al V2X Datagram Socket che lo inoltra verso il nodo remoto sul flusso di comunicazione. Nel caso in cui sia abilitata la frammentazione, durante questa fase non viene inviato l'intero bundle, ma i frammenti di esso, uno per volta, tenendo traccia dell'offset rispetto al bundle originale; in questo caso inoltre essendo la comunicazione a livello MAC, l'MTU del pacchetto da inviare non può eccedere i 1500 Bytes.

La parte di ricezione, riceve dati in ingresso sul V2X Datagram Socket che, tramite una funzione custom, la decodepkt(pkt,dim) che riceve in ingresso il pacchetto ricevuto e la sua dimensione, "spezzetta" il pachetto separando gli header ethernet dal payload e controlla che il pacchetto sia un pacchetto V2X (ethType = 0x8947), che la destinazione corrisponda all'indirizzo MAC del nodo su cui è in esecuzione e poi "passa" al V2X Datagram Service solo il payload del messaggio, ossia il CAM contenente il bundle che prima di essere inoltrato al Datagram Convergence Layer viene opportunamente decodificato.

4.7 Configurazione

Al fine di avviare il software IBR-DTN con il supporto V2X, è necessario modificare nel file di configurazione del processo demone, la parte relativa alle istanze dei convergence layer. Qui di seguito è illustrato un esempio di configurazione del V2X Datagram Convergence Layer:

```
# a list (separated by spaces) of names for convergence
layer instances.

...
net_interfaces = v2x0
...

# configuration for a convergence layer named v2x0
net_v2x0_type = v2x
net_v2x0_interface = eth0
net_v2x0_port = 0
```

Con questa configurazione, si richiede che venga istanziato un convergence layer, di nome "v2x0"che rimanga in ascolto sull'interfaccia eth0. Una volta impostati i parametri di configurazione, per abilitare l'esecuzione del convergence layer,è sufficiente che il suo nome compaia in net interfaces, una lista che contiene i nomi dei convergence layer da istanziare.

Capitolo 5

Risultati Sperimentali

In questo capitolo ci si occuperà di eseguire alcuni test e di valutare i relativi risultati. A tal proposito, è stato configurato uno scenario in cui due nodi sono connessi tramite una rete Wi-Fi dedicata, e la comunicazione avviene con controllo di flusso disabilitato, cioè, l'invio viene eseguito senza aspettare gli ack relativi ai singoli pacchetti. I sample presi in considerazione sono solo quelli in cui l'invio è avvenuto correttamente.

Le metriche misurate riguardano throughput, latenza della connessione, overhead introdotto dallo strato software implementato e il consumo di CPU associato.

5.1 Setup

I test sono stati effettuati con l'utilizzo di due computer connessi ad una rete Wi-Fi dedicata con access point a 300 MBps. I due dispositivi utilizzati sono:

- Asus F550C, 8 GB di RAM, processore i5-2410M con frequenza 2,3 GHz e quattro core su sistema operativo Ubuntu 16.04,
- Dell vostro 3550, 8 GB di RAM, processore i5 con frequenza 2,7 GHz e quattro core su sistema operativo XUbuntu.

Per la generazione e ricezione del traffico sono stati utilizzati due dei tool offerti da IBR-DTN, ovvero dtnsend e dtnrecv, che sono lanciati da terminale con i seguenti comandi:

```
dtnrecv --name abc
```

il quale mette il nodo ricevitore in ascolto; il parametro passato è il nome che si vuole assegnare all'applicazione;

```
dtnsend dtn://nodename/abc F.txt
```

con il quale il mittente invia il file F.txt al nodo con il nome dtn corrispondente tramite l'applicazione di nome *abc*.

Per le misurazioni sono stati apposti dei timestamp in formato Unix durante le fasi di invio e ricezione dei dati.

5.2 Throughput e latenza

In questa sezione si andrà ad analizzare il throughput e la latenza della trasmissione al variare delle caratteristiche di invio.

Il primo caso di prova è stato quello dell'invio di file di diverse dimensioni per verificare il cambiamento dei parametri misurati al variare delle dimensioni dei file. Successivamente si è andati a testare la rete aumentando il carico dati e inviando in modalità burst, tramite script bash, ripetutamente lo stesso file fino a raggiungere una dimensione complessiva di 100MB trasferiti.

I tempi misurati di seguito sono riportati in secondi.

5.2.1 Troughput

Come si può notare dalla seguente figura, il throughput è stato misurato in quattro scenari differenti:

- invio di un singolo file da 1MB,
- invio di un singolo file da 10MB,
- invio burst di 100 file da 1MB,
- invio burst di 10 file da 10MB.

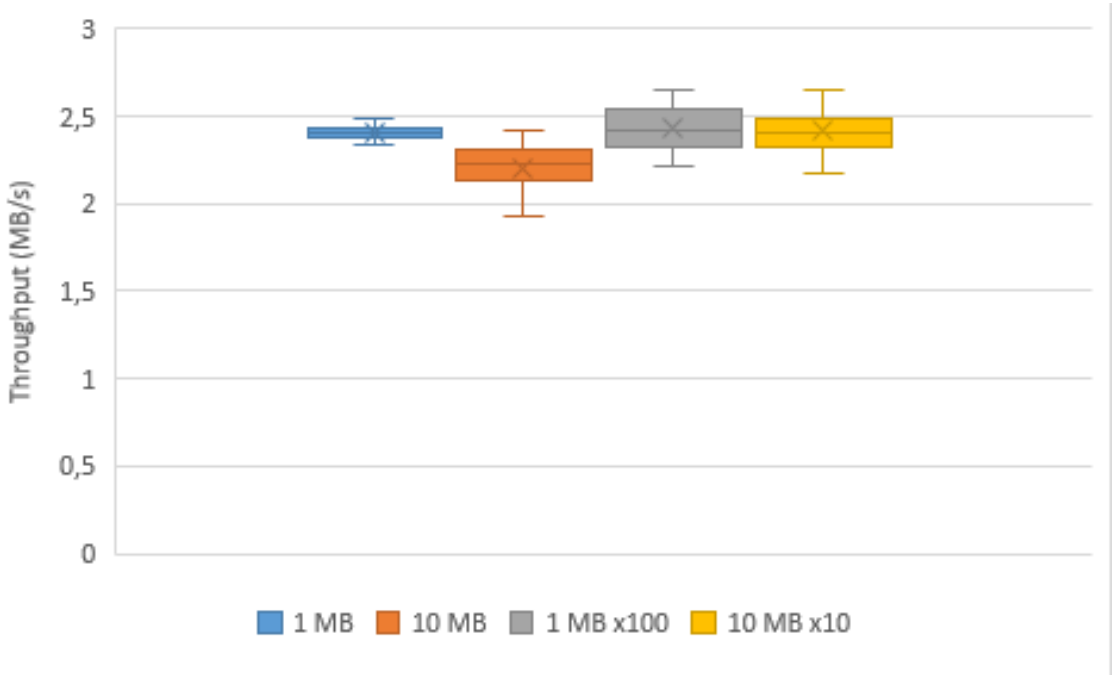

Figura 5.1: Throughput.

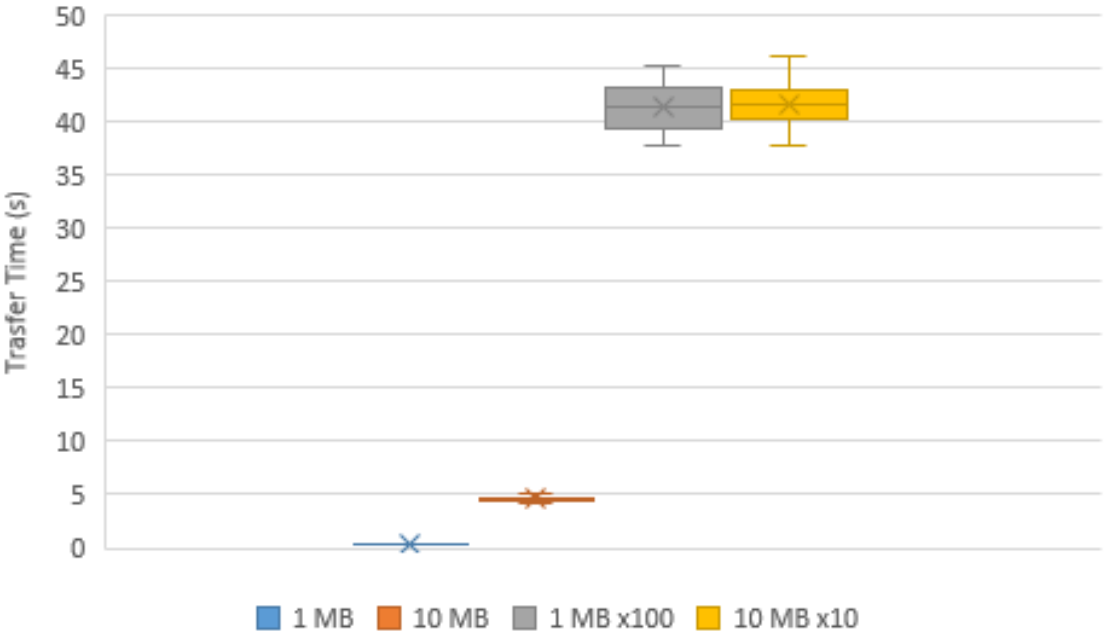

Figura 5.2: Transfer time.

Per quanto riguarda l'invio di un singolo file da 1MB il tempo di trasferimento

medio è risultato essere uguale a 0,416 s. Ciò implica un throughput medio dato da:

$$Throughput = \frac{2^{20}B}{0{,}416s} \simeq 2{,}4MB/s \tag{5.1}$$

Lo stesso esperimento ripetuto con un file da 10MB mostra un tempo di trasferimento medio pari a 4,553 s, con un throughput dato da:

$$Throughput = \frac{10 \times 2^{20}B}{4{,}553s} \simeq 2{,}2MB/s \tag{5.2}$$

Come si può notare, in questo caso le prestazioni sono inferiori al caso precedente di circa l' 8,33%; ciò è dovuto all'eccessivo carico della rete e, quindi, ad un'attesa dei dati in transito nei buffer di rete.

Gli stessi file precedenti sono stati usati per un trasferimento burst di dimensione complessiva 100MB. Si è quindi calcolato il tempo medio di trasferimento di 100 file da 1MB che è risultato essere uguale a 41,307 s, quindi il throughput in questo esperimento è:

$$Throughput = \frac{100 \times 2^{20}B}{41{,}307s} \simeq 2{,}42MB/s \tag{5.3}$$

Come si può notare le prestazioni in questo caso sono molto vicine a quelle dell'invio di un singolo file da 1MB. C'è però da notare come nonostante la maggior dimensione di dati inviati rispetto al caso del singolo file da 10MB il throughput in questo caso sia migliore, questo può essere dato dal fatto che il bundle inviato non è unico ma ne vengono inviati 100 di dimensioni minori, quindi l'overhead introdotto per il passaggio da un bundle al successivo permette un invio più rapido, nel senso che nel caso da 10MB il ricevitore deve aspettare la ricezione di tutti i frammenti, riordinarli e ricomporre il file, mentre inviando file di dimensioni minori i dati da ordinare e riasseblare sono di dimensioni minori, anche se in numero maggiore ciò comporta un leggero miglioramento di prestazioni.

Per fare un confronto di prestazioni come avvenuto per il singolo invio, anche per l'invio burst si è effettuato il confronto inviando 10 file da 10MB. Il tempo di trasferimento medio calcolato è 41,565 s; quindi il throughput in questo esperimento risulta essere:

$$Throughput = \frac{10 \times 10 \times 2^{20}B}{41{,}565s} \simeq 2{,}4MB/s \tag{5.4}$$

Le prestazioni in questo caso risultano essere in linea con il caso precedente.

5.2.2 Latenza

Per il calcolo della latenza si è scelto di utilizzare l'applicazione *dtnping* offerta da IBR-DTN. Viene utilizzato il seguente comando per far partire l'applicazione, come mostrato nella seguente figura:

```
dtnping dtn://nodename/echo
```

```
anto@dell:~/Scrivania$ dtnping dtn://asus/echo
ECHO dtn://asus/echo 64 bytes of data.
64 bytes from dtn://asus/echo: seq=1 ttl=30 time=17.12 ms
64 bytes from dtn://asus/echo: seq=2 ttl=30 time=31.69 ms
64 bytes from dtn://asus/echo: seq=3 ttl=30 time=18.88 ms
64 bytes from dtn://asus/echo: seq=4 ttl=30 time=18.48 ms
64 bytes from dtn://asus/echo: seq=5 ttl=30 time=20.89 ms
64 bytes from dtn://asus/echo: seq=6 ttl=30 time=21.96 ms
64 bytes from dtn://asus/echo: seq=7 ttl=30 time=18.00 ms
64 bytes from dtn://asus/echo: seq=8 ttl=30 time=18.42 ms
64 bytes from dtn://asus/echo: seq=9 ttl=30 time=20.27 ms
64 bytes from dtn://asus/echo: seq=10 ttl=30 time=17.04 ms
64 bytes from dtn://asus/echo: seq=11 ttl=30 time=20.22 ms
64 bytes from dtn://asus/echo: seq=12 ttl=30 time=18.68 ms
64 bytes from dtn://asus/echo: seq=13 ttl=30 time=29.06 ms
64 bytes from dtn://asus/echo: seq=14 ttl=30 time=18.29 ms
64 bytes from dtn://asus/echo: seq=15 ttl=30 time=16.80 ms
64 bytes from dtn://asus/echo: seq=16 ttl=30 time=26.17 ms
64 bytes from dtn://asus/echo: seq=17 ttl=30 time=18.96 ms
64 bytes from dtn://asus/echo: seq=18 ttl=30 time=17.81 ms
64 bytes from dtn://asus/echo: seq=19 ttl=30 time=31.74 ms
64 bytes from dtn://asus/echo: seq=20 ttl=30 time=17.54 ms
64 bytes from dtn://asus/echo: seq=21 ttl=30 time=20.76 ms
64 bytes from dtn://asus/echo: seq=22 ttl=30 time=17.08 ms
```

Figura 5.3: Output della dtnping.

L'aplicazione inoltre una volta stoppata mostra le statistiche relative al trasferimento che come mostrato nella seguente figura mostrano su 100 invii una latenza media di 20,29 ms.

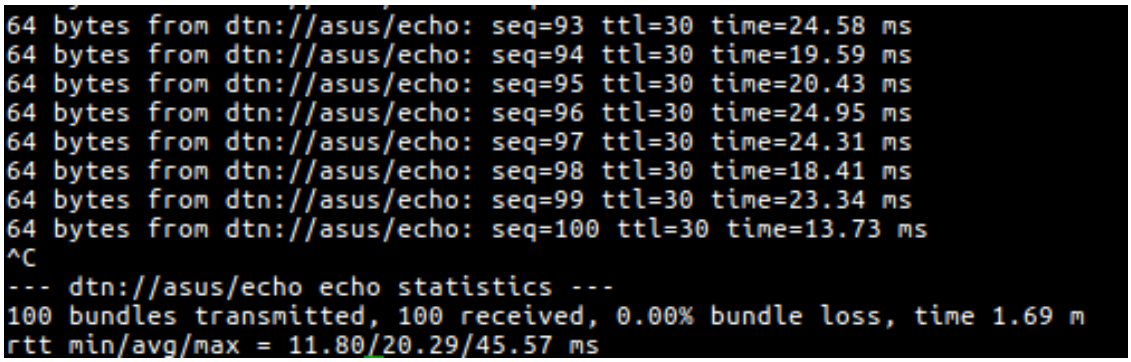

Figura 5.4: Statistiche della dtnping.

5.3 Processing

In questa sezione viene descritto e validato l'impatto che ha lo strato software sviluppato sulle prestazioni a livello di processing, sia dal punto di vista di overhead sul tempo di processamento rispetto al tempo complessivo, sia dal punto di vista dell'energia consumata. Queste misurazioni risultano particolarmente significative considerando che il lavoro è destinato a dispositivi IoT il cui consumo energetico è un fattore determinante per il loro funzionamento.

5.3.1 Overhead di processamento

Per overhead di processamento si è inteso in questo contesto il tempo trascorso da quando il bundle arriva al convergence layer, fino a quando l'intero bundle è stato inviato dal convergence layer stesso, normalizzato rispetto all'intero tempo trasmissione. Come nei test precedenti si sono presi in considerazione due scenari, quello dell'invio di un file da 1MB e quello di un file da 10MB.

File da 1MB

Dalla media delle misurazione effettuate sull'invio di un file di 1MB si è ottenuto un overhead di 0,376 s, che espresso in percentuale rispetto al tempo totale medio per un trasferimento risulta essere:

$$Overhead = \frac{0{,}376s}{0{,}416s} \simeq 90{,}38\%. \tag{5.5}$$

ciò significa per per circa il 90% del tempo il bundle viene elaborato dallo strato software V2X, mentre il restante 10% del tempo è necessario alla ricezione e ricostruzione del bundle.

File da 10MB

L'overhead misurato in questo caso è 4,165 s, che calcolato in percentuale rispetto al tempo totale medio per un trasferimento risulta essere:

$$Overhead = \frac{4{,}165s}{4{,}625s} \simeq 90{,}05\%. \tag{5.6}$$

Come si può notare il valore è molto vicino a quello del caso precedente, quindi l'overhead resta pressochè costante al variare della dimensione dei bundle da inviare.

5.3.2 Consumo di CPU

In questa sezione si è calcolato il consumo percentuale di CPU in vari scenari, confrontando il normale consumo di CPU dei dispositivi, con quello in cui si fa partire IBR-DTN e infine durante il trasferimento di bundle di diverse dimensioni. Per la misurazione del consumo di CPU si è utilizzato il comando **sar (System Activity Report)** che mostra le statistiche sulla CPU. Inoltre con il parametro *-u* x le statistiche vengono aggiornate ogni x secondi.

File di 1224B

In questo esperimento si è scelto un file della dimensione massima possibile come payload di un pacchetto ethrnet che non preveda la frammentazione, quindi il bundle viene inviato in un unico pacchetto. Questa dimensione è data da 1500B, l'MTU su Ethernet, meno la dimensione dei vari header. L'output del comando sar per questo esperimento è generato ogni secondo per via della velocità dell'invio del pacchetto. Di seguito sono mostrati quattro diversi scenari:

```
anto@dell:~/Scrivania$ sar -u 1
Linux 4.15.0-117-generic (dell)         24/09/2020      _x86_64_        (4 CPU)

11:36:31        CPU     %user     %nice   %system   %iowait    %steal     %idle
11:36:32        all      0,25      0,00      0,76      0,00      0,00     98,98
11:36:33        all      0,00      0,00      0,00      0,00      0,00    100,00
11:36:34        all      0,76      0,00      0,00      0,00      0,00     99,24
11:36:35        all      1,74      0,00      1,74      0,00      0,00     96,53
11:36:36        all      0,25      0,00      0,25      0,50      0,00     99,00
```

Figura 5.5: Consumo di CPU con dispositivo in "idle".

Come si può notare dalla figura in questo caso il consumo di CPU è decisamente ridotto sia in user space (colonna %user) che in kernel space (colonna %system): in

media, negli intervalli di tempo considerati, si tratta dello 0,6% per lo user space e dello 0,55% per il kernel space.

	CPU	%user	%nice	%system	%iowait	%steal	%idle
11:36:37	all	6,52	0,00	2,26	0,00	0,00	91,23
11:36:38	all	3,28	0,00	1,01	0,00	0,00	95,71
11:36:39	all	1,02	0,00	0,26	0,00	0,00	98,72
11:36:40	all	1,00	0,00	0,50	0,00	0,00	98,50

Figura 5.6: Consumo di CPU con dtn attivo sul nodo locale.

All'accensione della dtn sul dispositivo si può notare un incremento condiderevole del consumo sia in user space che in kernel space, con un picco iniziale dovuto all'avvio e alla configurazione dei moduli dtn. In media questo aumento risulta essere +2,35% in user space e +0,46% in kernel space.

	CPU	%user	%nice	%system	%iowait	%steal	%idle
11:36:41	all	0,75	0,00	0,25	0,00	0,00	98,99
11:36:42	all	0,50	0,00	0,25	0,00	0,00	99,25
11:36:43	all	1,28	0,00	0,26	0,26	0,00	98,21
11:36:44	all	1,78	0,00	0,51	0,25	0,00	97,46

Figura 5.7: Consumo di CPU con dtn attivo su entrami i dispositivi.

All'attivazione della dtn sull'atro nodo della rete il consumo non varia significativamente tranne che per gli ultimi due intervalli in cui il nodo corrente ha ricevuto un messaggio di beaconing ed ha avviato la fase di scoperta del "vicino". Ciò si può notare dal fatto che i valori nella colonna di i/o sono diversi da zero.

	CPU	%user	%nice	%system	%iowait	%steal	%idle
11:36:45	all	3,33	0,00	0,26	0,00	0,00	96,41
11:36:46	all	0,50	0,00	0,50	0,00	0,00	98,99
11:36:47	all	1,50	0,00	1,00	0,00	0,00	97,51
11:36:48	all	0,50	0,00	0,76	0,00	0,00	98,74
11:36:49	all	1,02	0,00	0,00	0,00	0,00	98,98

Figura 5.8: Consumo di CPU durante l'invio di un file da 1224B.

Infine, è stato effettuato l'invio del file da 1224B e come si può notare questo ha comportato un aumento del consumo di CPU solo nel primo intervallo considerato, dato che il file viene inviato con un singolo pacchetto, per poi tornare su livelli normali.

File da 10MB

In questo esperimento si è ripetuta la stessa esecuzione del test precedente ma con un ultimo scenario in cui il file inviato è un file da 10MB, e si è scelto come intervallo

	CPU	%user	%nice	%system	%iowait	%steal	%idle
11:13:41	all	8,28	0,00	3,14	0,00	0,00	88,58
11:13:43	all	3,14	0,00	1,00	0,38	0,00	95,48
11:13:45	all	5,26	0,00	1,63	0,00	0,00	93,11
11:13:47	all	8,38	0,00	1,38	0,00	0,00	90,25
11:13:49	all	9,40	0,00	1,91	1,14	0,00	87,55
11:13:51	all	5,41	0,00	1,51	0,00	0,00	93,08
11:13:53	all	4,91	0,00	1,51	1,13	0,00	92,44
11:13:55	all	6,07	0,00	1,39	0,13	0,00	92,41
11:13:57	all	5,82	0,00	1,77	0,00	0,00	92,41
11:13:59	all	4,74	0,00	2,62	0,25	0,00	92,39

Figura 5.9: Consumo di CPU durante l'invio di un file da 10MB.

di tempo per le statistiche 2 secondi. Per i primi tre casi i valori sono molto simili a quelli dell'esempio precedente.

In questo caso risulta evidente che il consumo di CPU sia in user space che in kernel space è molto superiore al caso precedente in quanto questa volta il pacchetto da inviare deve essere frammentato, inviato, e ricostruito alla ricezione. Il consumo medio di CPU negli intervalli di tempo considerati è circa del 6,14% in user space, e 1,79% in kernel space.

10 File da 10MB

Come ultima prova si è voluta caricare la rete effettuando un invio burst di 10 file consecutivi da 10MB ciascuno per un totale di 100MB trasferiti.

In quest'ultimo caso il consumo di CPU è decisamente maggiore al caso precedente: per quanto riguarda lo user space si attesta al 16,07%, mentre quello in kernel space risulta essere di 2,09%, leggermente superiore al caso precedente; questo suggerisce che le operazioni eseguite in kernel space sono pressochè indipendenti dalla dimensione dei dati trasferiti.

	CPU	%user	%nice	%system	%iowait	%steal	%idle
11:05:09	all	29,00	0,00	12,39	0,43	0,00	58,18
11:05:12	all	10,00	0,00	3,11	0,08	0,00	86,81
11:05:15	all	7,18	0,00	2,00	0,17	0,00	90,65
11:05:18	all	7,47	0,00	1,68	0,08	0,00	90,76
11:05:21	all	7,21	0,00	2,01	0,00	0,00	90,77
11:05:24	all	6,92	0,00	2,33	0,00	0,00	90,75
11:05:27	all	8,58	0,00	1,68	0,00	0,00	89,74
11:05:30	all	7,81	0,00	2,00	0,00	0,00	90,19
11:05:33	all	6,71	0,00	1,59	0,08	0,00	91,61
11:05:36	all	12,54	0,00	2,09	0,08	0,00	85,28
11:05:39	all	7,33	0,00	1,60	0,08	0,00	90,99
11:05:42	all	13,05	0,00	1,26	0,00	0,00	85,69
11:05:45	all	14,48	0,00	2,26	0,08	0,00	83,18
11:05:48	all	8,21	0,00	1,93	0,00	0,00	89,87
11:05:51	all	6,99	0,00	2,33	0,08	0,00	90,59
11:05:54	all	6,23	0,00	1,26	0,00	0,00	92,50
11:05:57	all	7,51	0,00	2,25	0,00	0,00	90,24
11:06:00	all	6,81	0,00	1,51	0,08	0,00	91,60
11:06:03	all	8,05	0,00	1,93	0,75	0,00	89,27
11:06:06	all	5,45	0,00	1,34	0,00	0,00	93,20
11:06:09	all	7,62	0,00	1,59	0,00	0,00	90,79
11:06:12	all	14,00	0,00	2,10	0,00	0,00	83,91
11:06:15	all	6,12	0,00	1,01	0,00	0,00	92,87
11:06:18	all	7,36	0,00	1,59	0,08	0,00	90,96
11:06:21	all	8,53	0,00	2,34	0,08	0,00	89,05
11:06:24	all	5,68	0,08	1,59	0,00	0,00	92,65
11:06:27	all	7,94	0,00	1,92	0,00	0,00	90,13
11:06:30	all	14,47	0,00	2,02	0,08	0,00	83,43
11:06:33	all	8,07	0,00	1,93	0,00	0,00	90,00
11:06:36	all	7,45	0,00	2,18	0,08	0,00	90,28
11:06:39	all	6,81	0,00	1,68	0,00	0,00	91,51
11:06:42	all	6,95	0,00	1,51	0,08	0,00	91,46
11:06:45	all	6,31	0,00	1,60	0,00	0,00	92,09
11:06:48	all	12,38	0,00	1,84	0,08	0,00	85,69
11:06:51	all	7,92	0,00	2,00	3,09	0,00	86,99
11:06:54	all	7,46	0,00	2,01	0,67	0,00	89,86
11:06:54	CPU	%user	%nice	%system	%iowait	%steal	%idle
11:06:57	all	7,18	0,00	2,17	0,00	0,00	90,65
11:07:00	all	8,19	0,00	1,77	0,08	0,00	89,95
11:07:03	all	21,02	0,00	1,59	0,08	0,00	77,30
11:07:06	all	5,02	0,00	1,51	0,00	0,00	93,47
11:07:09	all	7,29	0,00	2,18	0,08	0,00	90,45
11:07:12	all	20,40	0,00	1,77	0,00	0,00	77,82
11:07:15	all	8,37	0,00	1,26	0,00	0,00	90,38

Figura 5.10: Consumo di CPU durante l'invio di 100MB.

Capitolo 6

Conclusioni

Questo lavoro di tesi amplia la scelta protocollare di IBR-DTN, con il supporto al V2X. Questo garantisce un'importante flessibilità al framework, data la sempre più crescente diffusione di dispositivi IoT e veicoli "smart".

Come è evidente dai risultati sperimentali, la maggior parte del ritardo di trasferimento dei dati è dovuto soprattutto alla creazione del bundle di risposta attraverso IBR-DTN. Difatti, il tempo di latenza risulta essere quasi identico al tempo totale di trasferimento. Rimane necessario indagare sul motivo di questo impatto prestazionale da parte di IBR-DTN e tentare l'implentazione al supporto di altri protocolli. Inoltre, questo lavoro può essere migliorato in futuro con l'aggiunta del supporto di altri tipi di sensori veicolari, come ad esempio OBD (On Board Diagnostic), i quali permetteranno al DCC un miglior controllo del carico e gestione dei canali.

Infine, un aspetto che merita particolare attenzione nell'ottica di sviluppi futuri è la potenziale "universalità" dell'implementazione: nel lavoro proposto in questa tesi, un bundle viene incapsulato in un pacchetto che viene "costruito manualmente" (in questo caso una trama Ethernet); per questo motivo, questa soluzione può essere facilmente adattata per supportare ogni tipo di protocollo di rete desiderato. Pur dovendo introdurre una minima personalizzazioine per ognuno dei protocolli desiderati, questo consentirebbe di integrare una serie di Convergence Layers utilizzando un'unica implementazione.

Bibliografia

[1] K. Scott. Disruption tolerant networking proxies for on-the-move tactical networks. Technical report, 2005.

[2] K. Scott and S. Burleigh. Bundle protocol specification. RFC 5050, IETF, November 2007.

[3] Sebastian Schildt, Johannes Morgenroth, Wolf-Bastian Pöttner, and Lars Wolf. Ibr-dtn: A light-weight, modular and highly portable bundle protocol implementation. *Electronic Communications of the EASST, Volume 37: Kommunikation in Verteilten Systemen 2011*, 2011.

[4] Johannes Morgenroth. Ibr-dtn api. Technical report.

[5] D. Ellard and D. Brown. Dtn ip neighbor discovery (ipnd). Internet-Draft draft-irtf-dtnrg-ipnd-01, IETF, March 2010.

[6] Michael Demmer, Jörg Ott, and Simon Perreault. Delay-tolerant networking tcp convergence-layer protocol. RFC 7242, IETF, June 2014.

[7] Johannes Morgenroth. Ibr-dtn configuration file example. Technical report.

[8] Martin Arndt Hans-Joachim Fischer. Intelligent transport systems (its); communications architecture. ETSI document EN 302 665, ETSI, 2010.

[9] Andrea Lorelli Katrin Sjoeberg. Decentralized congestion control mechanisms for intelligent transport systems operating in the 5 ghz range; access layer part. ETSI document TS 102 687, ETSI, 2018.

[10] Andrea Lorelli Katrin Sjoeberg. Intelligent transport systems (its); radiocommunications equipment operating in the 5 855 mhz to 5 925 mhz frequency band; harmonised standard covering the essential requirements of article 3.2 of directive 2014/53/eu. ETSI document EN 302 571, ETSI, 2017.

[11] Martin Arndt Friedbert Berens. Harmonized channel specifications for intelligent transport systems operating in the 5 ghz frequency band. ETSI document TS 102 724, ETSI, 2012.

[12] Martin Arndt Andreas Festag. Intelligent transport systems (its); vehicular communications; geonetworking; part 3: Network architecture. ETSI document EN 302 636-3, ETSI, 2014.

[13] Martin Arndt Gérard Ségarra. Intelligent transport systems (its); vehicular communications; basic set of applications; definitions. ETSI document TR 102 638, ETSI, 2009.

[14] Martin Arndt Paulus Spaanderman. Basic set of applications; local dynamic map (ldm). ETSI document EN 302 895, ETSI, 2014.
[15] Andrea Lorelli Vincent Park. Basic set of applications; part 2: Specification of cooperative awareness basic service. ETSI document EN 302 637-2, ETSI, 2019.
[16] Openc2x project. `http://www.ccs-labs.org/software/openc2x`.
[17] Zeromq documentation. `https://zeromq.org`.
[18] Geonetworking project. `https://github.com/alexvoronov/geonetworking`.
[19] Vanetza project. `https://github.com/riebl/vanetza`.
[20] Car2x-labor website. `https://www.thi.de/forschung/carissma/labore/car2x-labor`.
[21] A guide to using raw sockets. `https://www.opensourceforu.com/2015/03/a-guide-to-using-raw-sockets/`.