POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica (Computer Engineering)

Tesi di Laurea Magistrale

Introduzione di primitive service-oriented su reti ad-hoc in ambienti sfidanti

Relatore
prof. Fulvio Risso

Correlatore:
ing. Gabriele Castellano

Candidato
Antonio Nunnari

Ottobre 2019

A chi non ha smesso di credere in me.

Ringraziamenti

Doveroso ringraziamento al professore Fulvio Risso, docente entrato a far parte del mio percorso universitario al terzo anno, nonchè relatore di questo ultimo grande sforzo chiamato tesi. Con particolare ammirazione verso la sua prospettiva di pensiero, in grado di analizzare i problemi da ogni punto di vista, e alla sua disponibilità per ogni mio dubbio, perplessità o difficoltà. Desidero ringraziare Gabriele Castellano per avermi aiutato nella ricerca delle soluzioni ai vari problemi affrontati, a lui auguro il meglio per il suo futuro post dottorato. Ringrazio inoltre Tierra S.p.A per avermi dato la possibilità di svolgere la tesi con la loro collaborazione ed in particolare Riccardo Loti per avermi accolto inizialmente ed avermi dato tutti gli strumenti per poter lavorare. Ringrazio Calogero Carrabbotta per avermi dato utili consigli e sopportato tutte le mie domande durante il lavoro di tesi. Grazie alla mia famiglia,che ha sempre supportato le mie decisioni, creduto in me e incoraggiato nei momenti bui e gioito dei traguardi raggiunti.Ringrazio Benedetta per essermi stata accanto e sopportato per tutta la tesi. Infine voglio ringraziare i miei amici con cui ho passato momenti stupendi e che sono stati sempre accanto durante questi lunghi anni.

Indice

Capitolo 1

Introduzione

1.1 Introduzione e obiettivi

Una Delay Tolerant Network (DTN) è una rete formata da dispositivi mobili come raspberry, PC oppure smartphone, che hanno una o piú interfacce di rete in grado di comunicare tra loro senza basarsi su una infrastruttura. Nella rete IBR-DTN i nodi partecipanti comunicano attraverso connessioni opportunistiche non stabili.
Data la natura della rete, le comunicazioni hanno come problema principale il ritardo di trasmissione, per esempio un nodo potrebbe essere disconnesso dalla rete diversi giorni prima di poter portare avanti la comunicazione. Una rete DTN ha come compito quello di propagare attraverso protocolli di reti indipendenti i dati e limitare i problemi dovuti al ritardo di consegna. In ogni caso, risolvere solamente il problema della connettività non è sufficiente per un utilizzo quotidiano. Una funzione fondamentale è quella di permettere alle applicazioni presenti sui vari dispositivi mobili di scambiarsi messaggi, informazioni ed usufruire dei servizi offerti sulla rete [1]. Obiettivo della tesi è sviluppare un modulo interno alla rete IBR-DTN in grado di distribuire sulla rete la conoscenza dei servizi, in modo che ogni nodo abbia un database in cui sono presenti tutti i servizi offerti dagli altri dispositivi. L'utilizzo dei servizi di una rete é perciò di particolare importanza ed é necessario definire un'architettura, dei meccanismi e dei protocolli per abilitare la rete a:

- Annunciare i servizi
- Richiedere i servizi agli altri nodi

- Selezionare il miglior servizio
- Consumare il servizio scelto

L'implementazione di queste funzionalità introduce particolari sfide a causa della natura "disrupted" di una rete di tipo DTN, primo obiettivo sarà quindi quello di mostrare le strategie e le implementazioni usate per la loro risoluzione.
Obiettivo successivo è quello di creare un Virtual Service in grado da comportarsi come un servizio presente sulla rete, a cui i nodi si possono connettere per far dirigere il traffico verso il miglior servizio disponibile. Ed in caso la connessione non fosse più accessibile, spostare il traffico verso un nuovo nodo. Tutto ciò in maniera totalmente trasparente verso chi usufruisce del servizio.

In questo documento quindi si partirà dal capitolo 2 con un'analisi dettagliata sul funzionamento del software IBR-DTN su cui viene implementata la Service Discovery. Una volta capita l'idea di IBR-DTN, attraverso il capitolo 3, vengono mostrate le soluzioni proposte in letteratura per risolvere le problematiche descritte prima (annuncio, richiesta e selezione del miglior servizio). Vengono inoltre confrontate ed analizzate per capire i motivi per qui alcune sono più rilevanti ed utili ai fini della Service Discovery. Nel capitolo 4 viene studiato con attenzione il modulo della Service Discovery, inizialmente mostrando i comportamenti del software da un punto di vista esterno ad esso e successivamente osservato il funzionamento interno. Per quanto riguarda il Virtual Service viene approfondito durante lo svolgimento del capitolo 5, esaminando le questioni più rilevanti sulla scelta del miglior servizio ed allo switch trasparente del traffico. Nel capitolo successivo sono riportati i test effettuati per valutare le performance dei nuovi moduli aggiunti e l'impatto che hanno sulla rete IBR-DTN. Come capitolo conclusivo vengono analizzati i risultati dei test precedenti, dando una spiegazione dei comportamenti osservati durante lo studio delle perfomance e come si potrebbe intervenire in futuro per migliorare il sistema complessivo.

Capitolo 2

IBR-DTN

2.1 Delay/Disruption Tolerant Networking

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 eterogena e 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. Oggigiorno, infatti, è sempre più comune scontrarsi con scenari applicativi in cui i dispositivi che devono comunicare sono in movimento e operano a potenza limitata, 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.

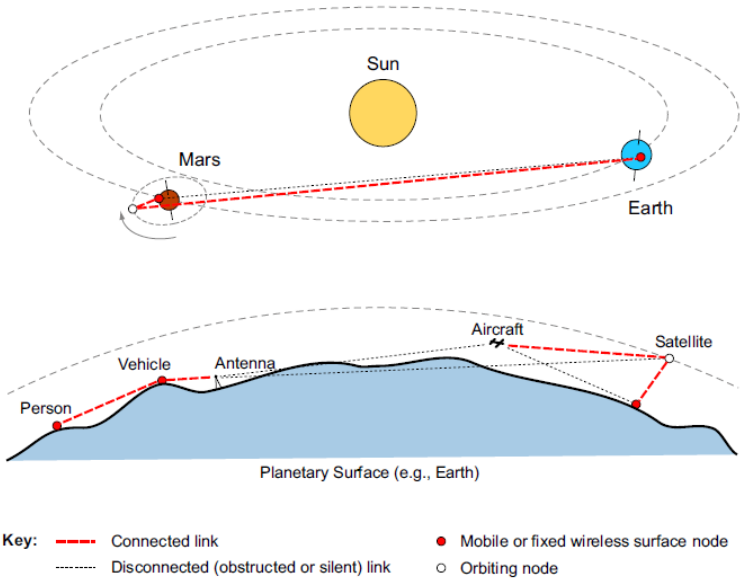


Figura 2.1. Esempi di contatti pianificati (comunicazioni interplanetarie) e opportunistici (comunicazioni sulla superficie terrestre)

Inoltre, parlando di connettività intermittente, è opportuno far distinzione tra i contatti pianificati e quelli opportunistici (figura 2.1). Lo scenario tipico dei contatti pianificati o *scheduled* è quello dello spazio, in cui i nodi si muovono su percorsi orbitali predicibili, tanto che è possibile prevedere o ricevere gli instanti in cui 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 risultato in linea di vista e 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.

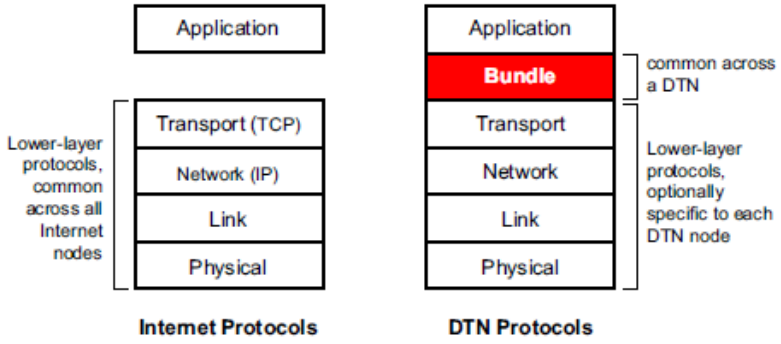


Figura 2.2. Confronto fra uno stack Internet (a sinistra) e uno stack DTN (a destra)

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.

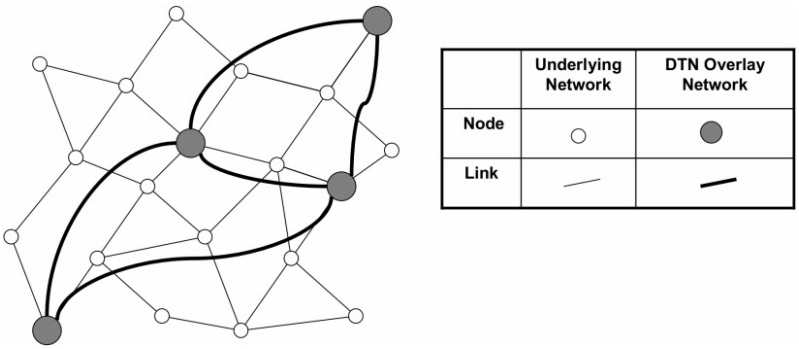


Figura 2.3. La rete DTN in overlay su un altro tipo di rete

La peculiarità di posizionarsi fra il layer di trasporto e il layer applicativo permette l'uso di DTN per creare dei proxy applicativi. 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 ottenee la risposta desiderata.[2]

2.2 Bundle Protocol

Il *Bundle Protocol*[3] è un protocollo sperimentale, corrispondente allo Bundle Layer 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).

- **Convergence Layer Adapter (CLA)**: invia e riceve i bundle per conto del BPA, sfruttando i servizi offerti da un qualche protocollo di trasporto (Es. TCP, UDP, RFCOMM, etc). Il modo in cui il CLA gestisce la trasmissione dei bundle dipende dal protocollo che adottata al livello sottostante.
- **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 i seguenti:

- 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.

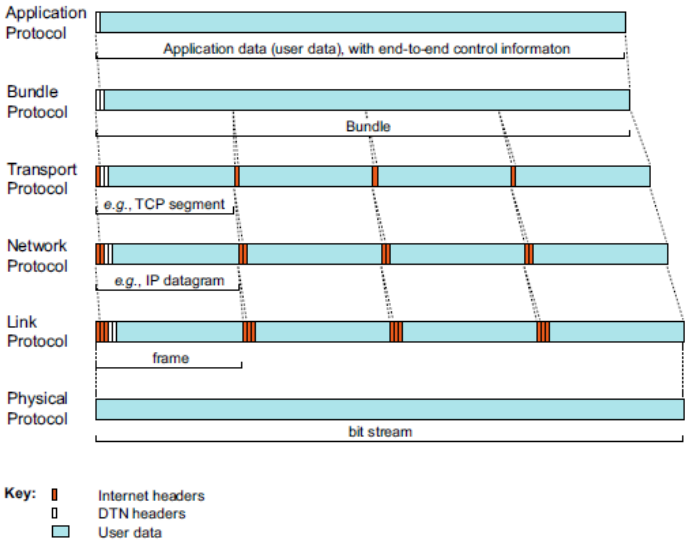


Figura 2.4. Incapsulamento dei protocolli TCP/IP nel Bundle Protocol

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 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 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.), estendibili e scalabile per una diversa varietà di protocolli di rete e dimensioni di payload.

Primary Block

Come si può notare in figura 2.5, oltre a versione, lunghezza del blocco, sorgente e destinzione, il primary block 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:

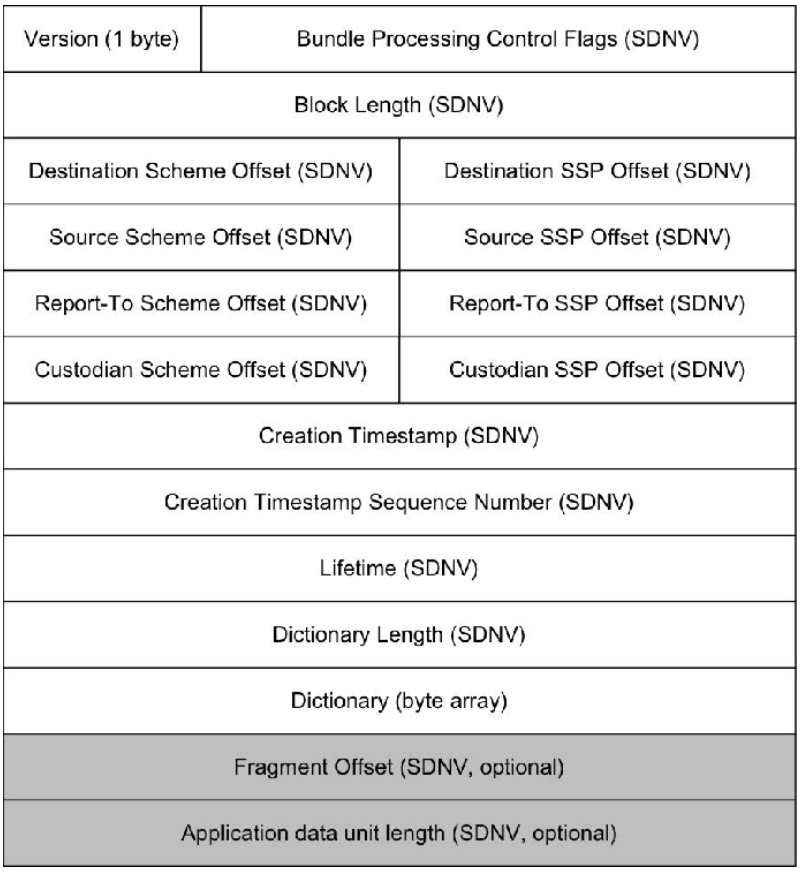


Figura 2.5. Formato del primary block di un bundle

- 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 reporti 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: indicate 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 l'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.

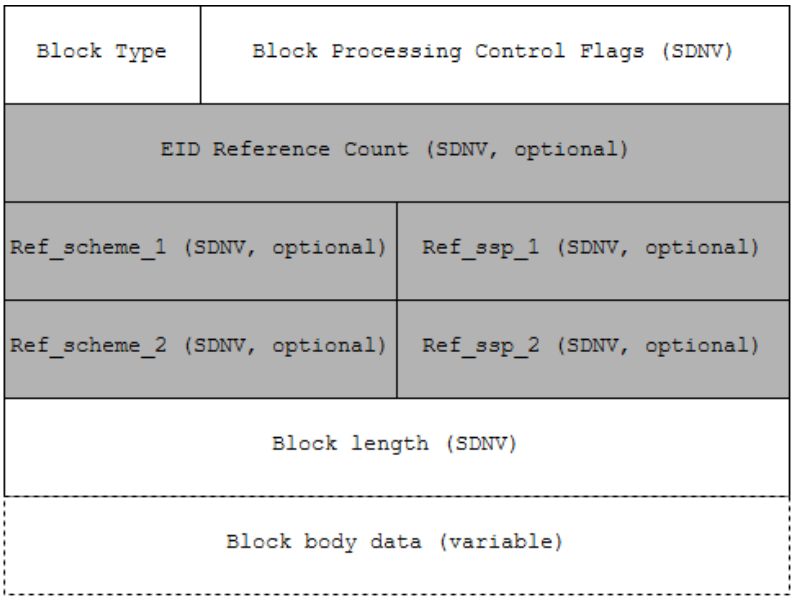


Figura 2.6. Formato generico di un blocco secondario di un bundle

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.

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 caso di 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 di risorse 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[4].

2.3.2 L'architettura

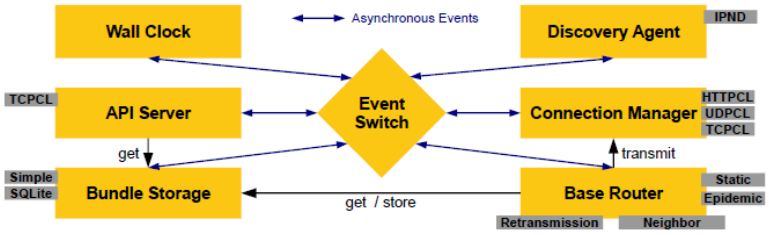


Figura 2.7. Architettura IBR-DTN

La versione di IBR-DTN per i sistemi operativi tradizionali è stata sviluppata in C++. Come si può notare in figura2.7, 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 fare riferimento alla documentazione[5].

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 componente di vitale importanza 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)[6]. 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 contengono l'EID del mittente, per permettere a chi lo riceve di effettuare il binding tra EID e indirizzo IP del vicino.

Connection Manager e Convergence Layer Ad occuparsi della gestione 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[3] sulle Delay Tolerant Network, sono i *convergence layer* a 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 dal sono specificati nella configurazione del demone. Attualmente IBR-DTN offre convergence layer per TCP/IP[7], UDP/IP, HTTP, IEEE 802.15.4 LoWPAN. 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 da *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.

API server L'interazione con IBR-DTN, e quindi la parte più utile al completamento di questa tesi, è ottunuta tramite l'API server. L'API server espone su una 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

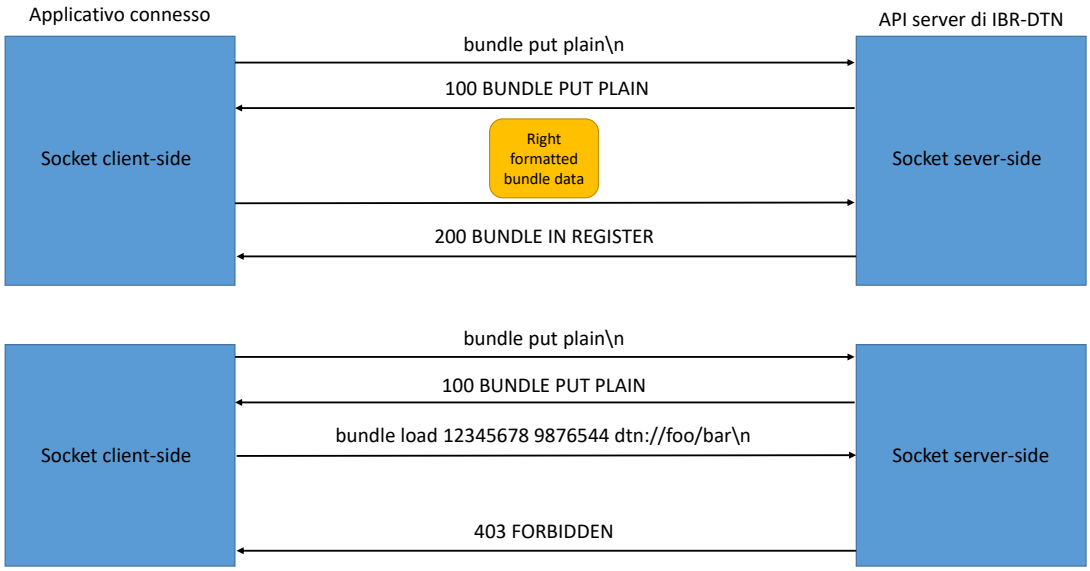


Figura 2.8. Interazione API Server

Le api disponibili sono visibili sul repository ufficiale di IBR-DTN [https://www.ibr.cs.tu-bs.de/projects/ibr-dtn/apidoc/0.12/api.pdf]

2.3.3 Installazione e avvio

In questo paragrafo sono illustrati i passaggi necessari all'installazione e avvio del demone 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.

```
$ 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 di IBR-DTN ora prevede, tramite opzioni attivabili sul comando `./configure`, l'attivazione del modulo bluetooth con l'uso di `-with-bluetooth`. La configurazione di default dei sorgenti è valevole solo per sistemi operativi con completo supporto a tutte le librerie da cui dipende il progetto. Le istruzioni da attivare su device limitati come la NetG5 verranno esposte nei capitoli successivi. Quindi dopo aver clonato il repository, si esegue la configurazione, la compilazione e l'installazione dei sorgenti:

```
$ cd ibrdtn-repository/ibrdtn
$\@./autogen.sh
$\@./configure --with-bluetooth
$ make
$ make install
$ ldconfig
```

E' possibile includere dei componenti opzionali specificandoli come parametri di `configure` nella forma `-with-module`, dove `module` rappresenta il nome del componente. Ad esempio, il flag `-with-sqlite` aggiunge il supporto a SQLite come meccanismo di storage dei bundle, mentre `-with-openssl` permette l'utilizzo di TLS.

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 verranno stampati, etc. Un esempio del comando:

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

Con questa conquesta combinazione di parametri avremo il binding sull'interfaccia di 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 [8]. In seguito sono illustrate le parti più significative.

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

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

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

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

```
# a list (seperated by spaces) of names for convergence layer instances.
net_interfaces = lan0 lan1 hci0

# configuration for a convergence layer named lan0
net_lan0_type = tcp # we want to use TCP as protocol
net_lan0_interface = wlan0 # listen on interface eth0
net_lan0_port = 4556 # with port 4556 (default)

# configuration for a convergence layer named lan1
net_lan1_type = tcp # we want to use TCP as protocol
net_lan1_interface = eth0 # listen on interface eth0
net_lan1_port = 4557 # with port 4557

# configuration for a convergence layer named hci0
net_hci0_type = bluetooth # we want to use bluetooth extension
net_hci0_interface = hci0 # listen on interface hci0
net_hci0_port = 10 # on channel 10
```

Avendo a disposizione diversi convergence layer è possibile listare 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.

```
# routing strategy
# values: default | epidemic | flooding | prophet | none
# In the "default" the daemon only delivers bundles to neighbors and static
```

```
# available nodes. The alternative module "epidemic" spread all bundles to
# all available neighbors. Flooding works like epidemic, but do not send the
# own summary vector to neighbors. Prophet forwards based on the probability
# to encounter other nodes (see RFC 6693).
routing = epidemic
```

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

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

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

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

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

2.3.5 Configurazione della time synchronization

La sincronia temporale è un punto molto 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'esempio seguente:

```
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 essersi collegati 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

Related Work

Il problema della service discovery è stato largamente analizzato in letteratura da molto tempo, nel giugno del 1997 è formalizzato lo standard RFC 2165, comunemente chiamato SLP o Service Location Protocol e negli anni successivi implementati e/o standardizzati Jini, Salutation e UPnP (Universal Plug and Play). In questo capitolo verranno presentati i vari approcci disponibili nello stato dell'arte, oltre ad un'analisi di come questi possano essere utilizzabili per la realizzazione di un meccanismo di service discovery in reti di tipo DTN.

3.1 Service Location Protocol

Questo standard open source è stato creato alla fine degli anni '90 principalmente per reti enterprise, ed è progettato per avere una buona scalabilità in relazione all'aumento della dimensione della rete. Prima di esso, gli utenti della rete, per potersi collegare ad un servizio, dovevano conoscere a priori l'hostname della macchina che ospitava quest'ultimo. Lo scopo di SLP è quello di rendere piú semplice la ricerca dei servizi presenti sulla rete [10]. Per ricercare un servizio ci si basa sulla caratterizzazione del servizo. SLP utilizza uno schema basato su coppie <attributo,valore>. Esso permette di creare un legame tra hostname e caratteristiche del servizio. I protagonisti di questo protocollo sono tre. User Agent(UA), Service Agent(SA) ed infine Directory Agent(DA). Gli SA sono le macchine in cui sono caricati i servizi, quando si collegano alla rete, annunciano i loro servizi. I DA sono macchine speciali che hanno la capacità di memorizzare i servizi e rispondere alle richieste degli UA. Questi ultimi infatti, quando hanno la necessità di un servizio, mandano una richiesta in multicast sulla rete (con le caratteristiche desiderate) ed il SA, con il servizio corrispondente a tutte le caratteristiche, risponde con il proprio hostname. Se sulla rete peró sono presenti DA allora il UA comunica in unicast direttamente con esso.

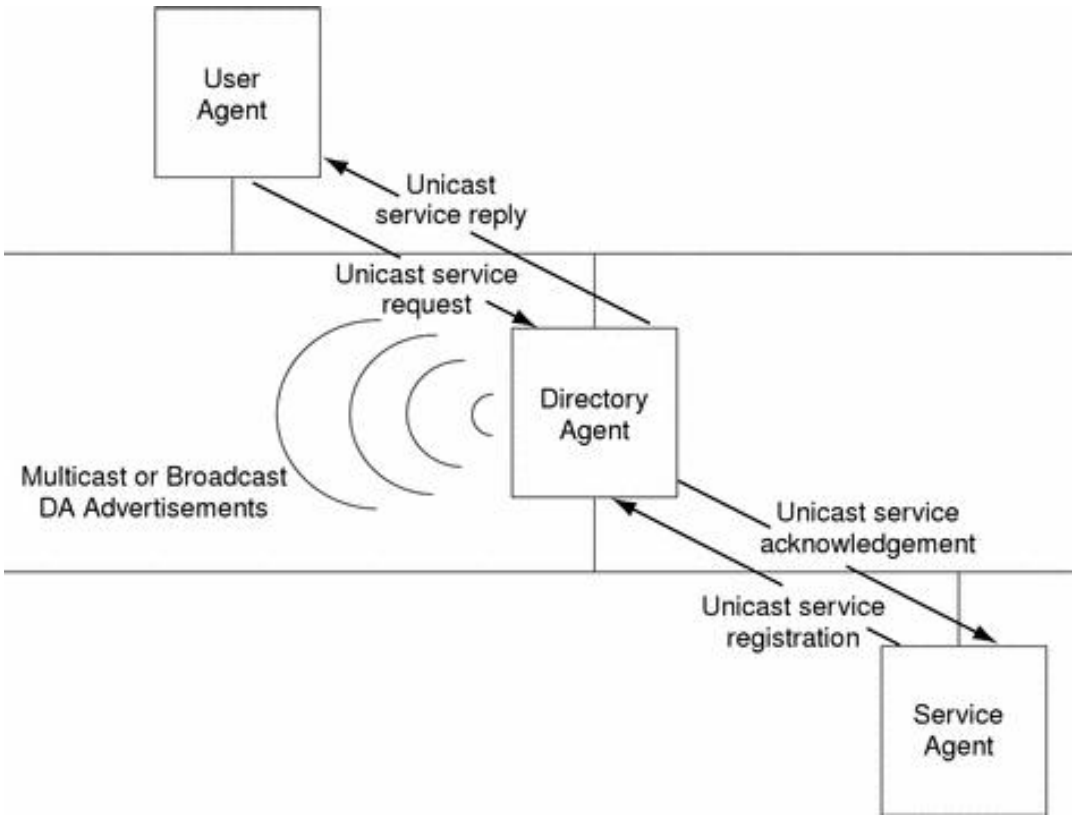


Figura 3.1. Architettura Service Location Protocol [9]

Questo per non inondare la rete con messaggi multicast, la presenza di DA rende la rete molto piú scalabile [11].

3.2 Universal Plug and Play

Universal Plug and Play è un protocollo piú recente, sviluppato da Microsoft e pubblicato nel 2008 nel ISO/IEC 29341. UPnP si pone come obiettivo quello di abilitare l'utente a trovare dispositivi e servizi presenti all'interno della rete. Inoltre permette anche di accedere molto semplicemente ad essi. Per poter raggiungere il suo obiettivo è ha la necessità di avere una descrizione dettagliata di ogni servizio e della macchina su cui si trova. UPnP utilizza una descrizione scritta nel formato XML dove al suo interno specifica caratteristiche e azioni che il servizio ha a disposizione,

inoltre sono presenti anche le descrizioni dei controlli e degli eventi della macchina host.

Figura 3.2. XML Service Description [12]

La fase di service discovery è affidata a SSDP, cioè Simple Service Discovery Protocol. In poche parole ogni servizio si annuncia in multicast alla rete quando si connette (ssdp::alive). In ogni annuncio è presente una lifetime ma il servizio puó comunque annunciare che presto non sarà piú disponibile (ssdp::bye-bye) e disconnettersi. Se una macchina si connette dopo che il servizio ha finito di mandare gli annunci (ssdp::alive) dovrà ricercare il servizio mandando in multicast un messaggio di discovery (ssdp::discovery). Il servizio che corrisponde alla descrizione richiesta risponderà via unicast alla macchina richiedente. Le caratteristiche fondamentali sono di questo approccio sono due. La prima, come nel caso di SLP è l'assenza di messaggi periodici per annunciare i propri servizi alla rete, questo è un aspetto positivo perchè rende la rete libera da pacchetti senza carico di informazione utile. La seconda caratteristica, a differenza di SLP, è l'assenza di nodi speciali (direttori o directory) in grado di memorizzare i servizi e rispondere per la localizzazione di essi. Se da un lato un architettura senza nodi directory è piú facile da mantenere e gestire, dall'altro costringe i nodi a cercare i servizi con messaggi multicast, aggiungendo carico alla rete.

3.3 Jini

Jini è un'architettura di rete service-oriented basata su devices con la capacità di usare l'ambiente Java. La particolarità di Jini è quella di essere scalabile e sapersi

adattare ai cambiamenti di una rete dinamica dove i servizi possono connettersi e disconnettersi.

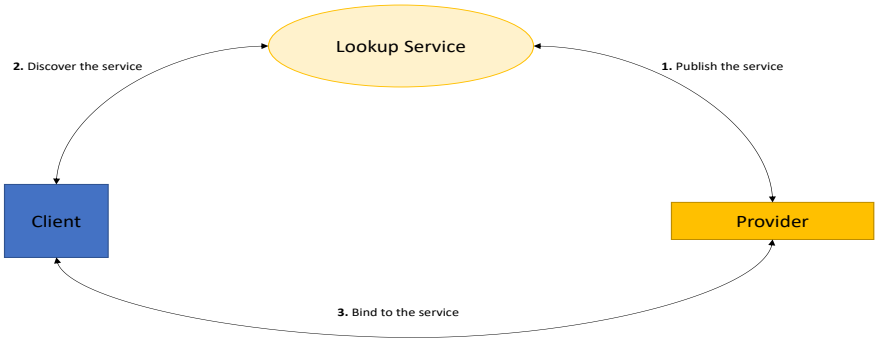


Figura 3.3. Architettura Jini

La componente interessante di questa architettura è il **Lookup Service**. Infatti i servizi si annunciano alla lookup service ed i client le richiedono i servizi. Quindi di fondamentale importanza è la scoperta della Lookup Service. Un client o un service provider possono trovarla attraverso due metodi:

- **Multicast request protocol**: richiesta in multicast per trovare uno o piú lookup service.
- **Multicast announcement protocol**: Annuncio in multicast per annunciare un lookup service.

Una volta trovato il lookup service il client richiede il servizio che gli serve per poi poterlo contattare. C'è da notare che il servizio non viene usato direttamente ma attraverso un proxy. Il proxy viene creato nel momento che il servizio si registra sul lookup service, infatti invia durante la registrazione un interfaccia Java che il client puó usare per usufruire del servizio (Remote method invocation, RMI)[13]. Un'altro aspetto interessante è che il client puó ricevere notifiche sul cambiamento dello stato del servizio attraverso il Java Remote Events [11].

3.4 Confronto

Viene riportata una tabella con i confronti dei maggiori protocol di service discovery, mostrando le varie caratteristiche [12].

Feature	Jini	UPnP	Salutation	SLP	Bluetooth SDP
Initial communicatio -n method	Unicast and multicast	Unicast and multicast	Unicast and broadcast	Unicast, multicast and broadcast	Unicast and broadcast
Descriptio -n capability	Template (Java interfaces)	XML template, vendor defined types supported	Predefined XML-based template	Template, simple service categories	128-bit universal unique IDs for service types
Query capability	Service type and attribute value. Browsing support	Service type, prefix matches supported. Browsing support	Type and attribute matching. Browsing support	Type, attribute value. Browsing support	Type, attribute value. Browsing support
Discovery scope	LAN, administrative domain, location	LAN	LAN	Administr- ative domain, LAN, location	Bluetooth network
State notification	Through agents	Subscription mechanism	Dependent on application design	Not supported	Not supported
Security	Verification, permissions. Interfaces to existing security protocols	Device-user interactions handled, many authorization procedures	Authentica- tion (user-id, password scheme)	Authentica- tion	Trusted discovery, Bluetooth security

Figura 3.4. Confronto fra protocolli di service discovery

Analizzando le varie soluzioni adottate nel tempo si possono trovare spunti utili per risolvere i principali problemi,mostrati nel capitolo introduttivo, su un rete di tipo DTN. Innanzitutto si vede come alcuni protocolli utilizzino un metodo basato su comunicazioni multicast ed altre che preferiscono comunicazioni broadcast per annunciare i servizi. Una rete DTN è composta principalmente da devices con poche risorse, è importante quindi limitare il traffico analizzato dai nodi. Per questo motivo si è scelto un approccio in cui l'annuncio dei servizi avviene in multicast, in questo modo i messaggi vengono analizzati solamente sui devices che hanno interesse a partecipare alla discovery dei servizi. Oggetto di attenzione particolare è

la descrizione di un servizio. La maggior parte delle soluzioni utilizza un template XML. Il problema di XML risiede principalmente nell'overhead. Infatti, mentre per una LAN dove la banda di rete è relativamente alta e stabile, in una rete DTN le connessioni sono brevi ed è dunque necessario trasmettere la maggior quantità di dati utili. La scelta è ricaduta sul formato JSON, il quale ha un overhead minore rispetto ad un file XML. In merito alla richiesta e selezione del miglior servizio, le soluzioni proposte in letteratura coincidono con quella utilizzata nel nostro progetto. Esse si basano su di un set attributo-valore per identificare il servizio consono al richiedente. Nei prossimi capitoli viene mostrato come le soluzioni scelte vengono implementate ed attraverso simulazioni analizzato l'impatto sull'intera rete DTN.

Capitolo 4

Service Discovery

In questo capitolo viene mostrato il modulo della Service Discovery, ovvero il modulo che si occupa della ricerca e diffusione sulla rete DTN dei servizi offerti dai nodi partecipanti.

4.1 Funzionamento e caratteristiche

Viene di seguito riportato in dettaglio il funzionamento del nostro protocollo di Service Discovery. Dopo aver fornito una descrizione dei tipi di messaggi di messaggi definiti, vengono presentati gli algoritmi e le strategie tramite cui tali messaggi vengono scambiati dai nodi della DTN.

4.1.1 Messaggi

Il modulo della Service Discovery per comunicare con gli altri nodi della rete ha a disposizione diversi messaggi che vengono usati al momento del bisogno. In questa sezione si mostrano i messaggi principali e la loro struttura. I messaggi vengono impacchettati nel payload block dei bundles ed hanno una struttura molto semplice che si differenzia in base alla tipologia. La struttura è la seguente:

Packet Type
Payload

Il primo campo indica la tipologia di pacchetto che si vuole mandare, che vedremo successivamente, mentre il secondo è il messaggio effettivo che viene trasportato.

Advertisement Packet

Packet Type = ADVERTISEMENT
Payload:
Service List:
Service #1
...
Service #N

Questo messaggio è quello utilizzato per trasportare una lista di servizi. Ha un duplice utilizzo nella Service Discovery:

- Annunciare i propri servizi in multicast alla rete.
- Rispondere in unicast alle richieste dei nodi.

Request Packet

Packet Type = request
REQUEST :
Service Name
Service Category
Service Protocol
Service Keywords

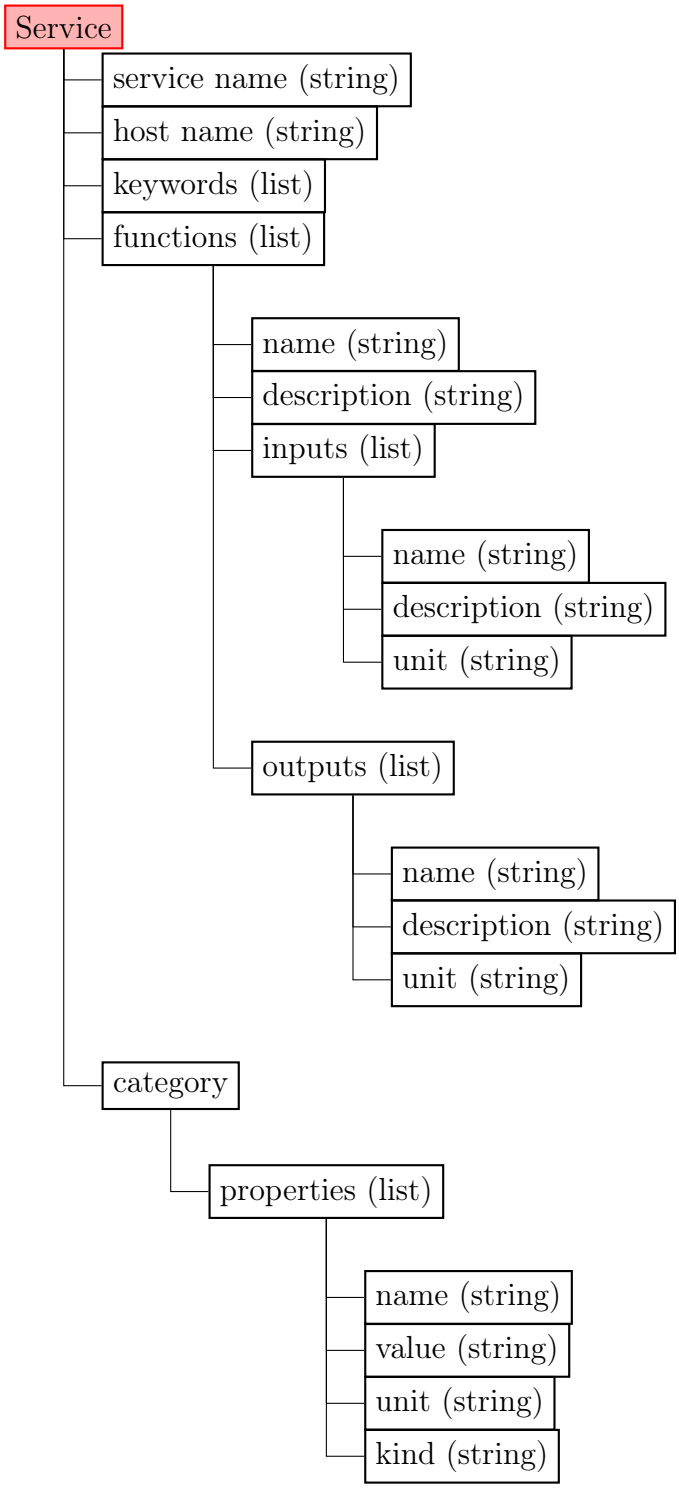
Questo pacchetto può avere due significati differenti a seconda del campo destinazione. Se viene trasmesso in multicast viene chiamato *request*, mentre se la destinazione è unicast viene chiamato *solecitation*. Nel caso fosse un request packet, esso conterrà nel payload una serie di filtri che indicano la tipologia di servizio desiderata. La risposta è un messaggio di advertisement, unicamente verso il richiedente, con all'interno tutti i servizi che soddisfano i vincoli della richiesta. Nel caso fosse un pacchetto col significato di solecitation, il nodo risponde con tutta la lista dei servizi di cui esso è a conoscenza.

Update Packet

Packet Type = UPDATE
Update Type: ADDED | CHANGED | REMOVED
Service

Questo messaggio indica che è avvenuta una modifica nel servizio contenuto nel payload. Il servizio potrebbe essere stato aggiunto, modificato o rimosso, questa informazione viene scritta nel campo Update Type. Questo pacchetto, trasmesso in multicast, viene generato quasi solamente quando è l'applicazione, proprietaria del servizio, a modificare direttamente il servizio, questo viene fatto per ridurre le latenze sull'aggiornamento dei database dei nodi. Così facendo non devono aspettare lo scadere del timeout del servizio. Tuttavia può essere che venga generato anche quando l'applicazione si disconnette, volontariamente o involontariamente, dal socket a cui è connesso con il nodo IBR-DTN. In questo caso, il nodo capisce che non è più disponibile l'applicazione e tutti i servizi a lei collegata, e informa la rete di questo evento.

4.1.2 Struttura del servizio



Nella struttura ad albero riportata qui sopra viene mostrata la struttura del servizio. Come si può vedere, le prime due informazioni che compaiono sono *service name* e *host name* cioè rispettivamente il nome del servizio ed il nodo su cui è attivo il servizio. Successivamente è presente una lista di parole chiamata *keywords*. Esse sono le parole chiavi che si possono inserire durante la richiesta di un servizio, se almeno una keyword corrisponde, il servizio verrà inserito nella risposta. Da qui in avanti la struttura diventa leggeremente più complessa. Infatti il campo seguente è quello che descrive le funzioni che possono essere richieste al servizio. In questa sottostruttura sono presenti i campi: *name*, *description* e due campi simili tra di loro *inputs* e *outputs*. Questi ultimi per indicare il formato dell'eventuale risultato alla chiamata di funzione. Il prossimo campo, *category* ha un ruolo fondamentale nella service discovery ed anche nel virtual service, quindi ora verrà analizzato nel dettaglio.

Categorie

Ogni servizio come già detto appartiene ad una categoria, le categorie implementate sono sei. Ed ognuna contiene una lista di proprietà (name, value, unit, kind) per caratterizzarla appieno.

- **Sensor**
  Servizio in grado di raccogliere dati sull' ambiente che lo circonda.
  - *Sensor Tecnology* (mandatory): analogica, digitale.
  - *Sensor Type* (mandatory): temperature, humidity, brightness, ...
  - *Physical Principle*(mandatory, repeatable): optical, thermocouple, ...
  - *Unit* (mandatory,repeatable): celsius, kelvin, meters, ...
  - *Warm-Up Time* (optional,repeatable): tempo necessario per garantire una misura valida.
  - *Resolution* (mandatory): Rappresenta la minima variazione possibile.
  - *Accuracy* (mandatory): Massimo errore di misurazione.
  - *Max/Min Value*: Fondi scala.
  - *Output Format* (mandatory, repeatable): json, plain, xml, ...
- **Actuator**
  Servizio in grado di effettuare azioni fisiche, come braccia robotiche.
  - *Actuator Type* (mandatory): electric engine, solenoid, comb drive, ...
  - *Physical Principle* (mandatory, repeatable): idraulic, electric, ...

  - *Force or Torque* (mandatory): massima forza o momento torcente dell'attutatore.
  - *Speed or Angular Speed*: (mandatory): velocità massima senza carico.
  - *Acceleration or Angular Acceleration*: massima accelerazione o accelerazione angolare.
  - *Output* (mandatory): descrizione del lavoro che è in grado di svolgere l'attutatore.
  - *Stiffness* (optional): Massima forza prima che l'attutatore venga deformato.
  - *Power* (optional): Potenza necessaria per mettere in moto l'attutatore.
  - *Output Format* (mandatory, repeatable): json, plain, xml or nothing

- **Input/Output**
  - *I/O type* (mandatory): mouse, screen, keyboards. ...
  - *Input* (mandatory, repeatable): coordinates, text, ...
  - *Output* (mandatory, repeatable): the i/o device output
  - *Input Format* (mandatory, repeatable) : json, plain
  - *Output Format* (mandatory, repeatable) : json, plain

- **Network**
  Servizio che offre servizi di rete come router, switch, generic server oppure nat.
  - *Device type* (mandatory): es router.
  - *Min Bandwidth* (mandatory): es. 10MB/s.
  - *Max Bandwidth* (mandatory): es. 100MB/s.
  - *OSI Level* (mandatory, repeatable): data, application, ...
  - *isServer* (mandatory, repeatable) : true, false
  - *Kind of service* (mandatory) : es. FTP.
  - *Protocol* (mandatory) : es. SFTP.
  - *External Connection* (mandatory) : Specify if the device is able to connect DTN devices to the external world.

- **Website**
  Servizio che offre la possibilità di comportarsi come proxy oppure webserver per la rete.
  - *HTTP/HTTPS* (mandatory): Specify if the website uses HTTP o HTTPS.

  - *Chipersuite* (optional, repeatable): indica la lista di protocolli di sicurezza supportati.
  - *Certificate expiration* (optional): data di scadenza del certificato per i protocolli che usano TLS.
  - *SSL Protocol* (optional): Versione del protocollo SSL.
  - *TLS Protocol* (optional): Versione del protocollo TLS.
  - *Web Site Type* (optional): indica l'uso a cui è destinato il sito (configurazione, informazione,...).

- **Storage**
  Servizio che si occupa di gestire un database, in cui i nodi della rete possono inserire e rimuovere files.

  - *Storage Type* (mandatory): es. SAN.
  - *Max Storage* (mandatory)
  - *File System* (mandatory): es. NTFS
  - *Protocol* (mandatory, repeatable): protocollo usato per gestire i files (es. SFTP).
  - *Max file size* (optional)
  - *Accepted Files* (mandatory): es. txt, jpg, mp3.
  - *Unaccepted Files* (optional)

- **Computation**
  Servizi in grado di svolgere funzioni di calcolo più o meno complicate, partendo da calcoli algebrici fino a calcoli crittografici, eleborazione immagini, etc.

  - *Type* (mandatory): specifica quale sarà il tipo di calcolo offerto.
  - *Input* (mandatory): specifica il formato dei file su cui il dovrà lavorare.
  - *Output* (madatory): specifica il formato del file risultante.

4.1.3 Architettura generale

L'architettura generale mostra, come detto in precedenza, i ruoli di ogni nodo all'interno della rete. Per arrivare a definire l'architettura usata andremo prima ad analizzare due tipologie, *Directory-less* e *Directory base*, e successivamente si arriverà alla soluzione scelta, illustrando pro e contro di ogni tipologia e le motivazioni della soluzione adottata.

- *Architettura Directory-less (Senza Direttori)*
  In questa architettura ogni nodo ha lo stesso ruolo. Ogni nodo tenta, in modo attivo o passivo, di creare il database completo dei servizi presenti sull'intera rete. Il vantaggio principale è la semplicità dell'intera struttura, facile da mantenere. Un ulteriore vantaggio è la latenza del nodo per conoscere il servizio che è tendenzialmente bassa in quanto basta che un messaggio di advertisement arrivi al nodo. Questa naturale semplicità ha come rovescio della medaglia un problema. Quando un servizio viene aggiunto, deve essere informata tutta la rete. Poichè la rete in questione è molto instabile ogni nodi dovrà inoltrare messaggi periodici in cui informa i servizi disponibili. Tutto questo traffico sviluppato potrebbe congestionare la rete ed è quindi un punto cruciale impostare il periodo di advertisement in maniera corretta. Cioè da bilanciare le informazioni conosciute in ogni singolo nodo e la quantità di messaggi generati sulla rete [1].

- *Architettura Directory-Based*
  L'architettura directory based, o basata su direttori, ha come idea quella di eleggere nodi al ruolo di direttorio. Un direttorio è un particolare nodo con la funzione di memorizzare tutti i servizi all'interno della rete. I nodi restanti informano i direttori dei loro servizi. Quando un nodo vuole conoscere i servizi sulla rete richiede l'informazione ai direttori. Qui si può fare un ulteriore distinzione tra un'architettura centralizzata ed una distribuita.[1]

  – **Centralizzata** :
    In questo schema esiste una solo nodo con il ruolo di direttorio. Un solo

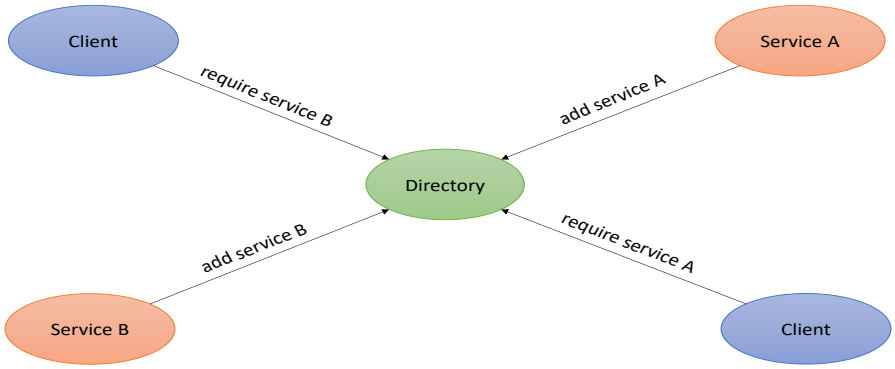


Figura 4.1. Architettura Directory Based centralizzata

direttorio comunque non è una buona scelta per reti con alta instabilità perché esso potrebbe essere irraggiungibile. Inoltre questa soluzione ha problemi con la scalabilità, se la rete si espandesse il direttorio, anche lui un nodo con poche risorse, potrebbe non reggere, rendendo la Service Discovery pressochè inutile.

– **Distribuita** :
A differenza dello schema precedente non esiste un solo nodo sulla rete, ma ce ne sono diversi distribuiti nella rete. Ora i direttori sono scalabili e con più probabilità di essere raggiunti.

Questa architettura è notevolmente più complessa e richiede che i direttori siano replicati o comunque che comunichino tra di loro per reperire i servizi che gli vengono richiesti dai nodi client. Il vantaggio è quello di avere la possibilità di creare poco traffico sulla rete, ma anche in questo caso se la procedura per replicare o aggiornare i direttori non viene fatta con criterio il traffico presente sulla rete potrebbe essere elevato [1].

- *Architettura ibrida*

L'architettura ibrida è un compromesso fra le due precendenti, ed è stata la scelta usata per la service discovery. In questa architettura ogni nodo può decidere se memorizzare i servizi presenti nei messaggi di advertisement oppure no. La differenza con l'architettura directory based, risiede nel fatto che la rete continua a mandare messaggi di advertisement come la soluzione directory-less [1]. La motivazione principale è che si vorrebbe utilizzare la semplicità di un'architettura directory-less, ma molti nodi potrebbero essere poveri di risorse e non sarebbero in grado di memorizzare molti servizi senza saturare tutte le risorse. Con questa soluzione solo i nodi con abbastanza capacità memorizzano il database completo dei servizi presenti sulla rete. Inoltre nel caso fosse necessario, possono rispondere anche alle richieste di servizi ed indicare come raggiungerli. Un ulteriore vantaggio è dato dalla possibilità di abbassare il numero di advertisement mandati in multicast per annunciare i servizi, questo perché ora i nodi, non essendo più legati esclusivamente ai messaggi di advertisement per localizzare un servizio possono richiederli direttamente ai nodi e ridurre le latenze di discovery.

4.1.4 Pubblicazione Servizi

Per usufruire dei servizi messi a disposizione dai vari componenti della rete si rende, ovviamente, necessaria la conoscenza da parte del consumatore della loro disponibilità. Un nodo può scoprire tali informazioni attraverso un approccio reattivo o con uno proattivo.

Gli approcci **reattivi** consistono nell'ascoltare passivamente i messaggi nella rete:

- **Advertisement Packet**:
  Questo tipo di pacchetto contiene i servizi disponibili sul nodo che ha lo inviato, e viene generato periodicamente in multicast a tutti i nodi iscritti alla service discovery. Un nodo ascoltando questo messaggio ha la possibilità di aggiornare il suo service database e resettare i lifetime dei servizi.

- **Update Packet**:
  Questo pacchetto, nel caso fosse del tipo "CHANGE" o "ADDED", contiene un solo servizio. Viene generato e spedito in multicast quando il nodo aggiunge, aggiorna o rimuove un servizio ("DELETE") a quelli da lui offerti. I nodi riceventi possono sfruttare questa informazione per capire i servizi disponibili ed in caso rimuoverli o aggiornarli.

Gli approcci **proattivi** consistono nel richiedere alla rete i servizi di cui si ha bisogno attraverso 2 metodi:

- **Request**:
  In questo approccio il nodo richiedente manda in multicast un pacchetto inserendo al suo interno un messaggio contenente i vincoli che deve soddisfare il servizio richiesto. I nodi riceventi analizzano il messaggio, filtrano i servizi conosciuti e rispondono con un messaggio di advertisement unicast al richiedente.

- **Solecitation**:
  Questo approccio concettualmente diverso dal precedente, viene sviluppato internamente nel codice come sottocaso della Request.
  Sostanzialmente il nodo chiede ad un altro nodo della rete (quindi unicast) tutti i servizi a lui noti, il quale risponderà con un messaggio unicast del tipo advertisement.

4.1.5 Disconnessione di un servizio

Poiché l'intero sistema è continuamente variabile e imprevedibile, un servizio potrebbe essere inaccessibile in qualsiasi momento per diverse cause. è quindi importante che i nodi si accorgano tempestivamente dei servizi che per un motivo o per l'altro non sono più disponibili. In questa sezione si mostrano le varie tecniche utilizzate per accorgersi l'indisponibilità di un servizio sulla rete.
Le tecniche utilizzate sono tre:

- **Disconnessione Volontaria**:
  Quando un servizio vuole disconnettersi dalla rete ha la possibilità di inviare un messaggio multicast chiamato "UPDATE", dove al suo interno inserisce il campo "DELETE". I nodi che lo ricevono sanno immediatamente che il servizio non è piu disponibile.
  Il vantaggio di questo messaggio è la velocità con cui i nodi della rete si accorgono di cosa è accaduto generando un evento interno ad IBR-DTN ed eliminando il servizio non più utile. Questo approccio viene usato solitamente quando l'applicazione attraverso l'EAPI invia il comando **service deactivate <service name>**, in questo modo gli altri nodi sulla rete sapranno che il servizio non è più utilizzabile.

- **Disconnessione Involontaria by Socket**:
  Con questa tecnica il nodo che ospita il servizio monitora il socket a cui esso è connesso. Nel caso cada la connessione tra nodo e socket,per errori vari, il nodo invia la stessa tipologia di messaggio del caso precedente in multicast alla rete. Anche qui il vantaggio è che la rete si accorge velocemente del servizio non è certamente più disponibile.

- **Disconnessione del nodo**:
  Questo è il caso più complicato a causa dell'imprevedibilità dell'ambiente in cui si verifica. Un nodo, in una rete DTN, può essere temporaneamente inaccessibile. Spesso capita, che spostandosi fisicamente non riesca a trovare una connessione opportunistica per collegarsi alla rete. Questo evento comunque non implica che il nodo sia caduto. Infatti dopo diverso tempo potrebbe trovare un punto di connessione e riprendere l'erogazione dei propri servizi. La soluzione scelta a questo problema è stata l'utilizzo di un timeout per ogni servizio. In altre parole, quando si imposta un servizio è possibile settare un timeout che indica per quando tempo il servizio è da considerarsi valido. Ogni qual volta che i nodi ricevono un messaggio di advertisement, il timeout viene resettato al valore iniziale. Nel caso contrario, in cui non vengano ricevuti advertisement per tutta la durata del timeout, il servizio in questione viene eliminato.

4.1.6 Timeout

Come introdotto nel paragrafo precedente, data la casualità della topologia, non si può dare per certo la disconnessione di un nodo e quindi dei servizi a lui collegati. Per affrontare questo problema sono stati aggiunti vari timeout, in modo tale da garantire una stabilità minima al service database di ogni nodo. Infatti senza di essi può capitare, dati valori troppo bassi di timeout per i servizi, che il database venisse popolato e subito dopo spopolato. Nel seguente codice si mostra la logica usata per settare i timeout.

```
1 void SDWorker::addService(std::shared_ptr<SDService> srv) {
2     if(_SD_discard_not_guaranteed){
3         int max = std::max(srv->getTimeout(),_SD_min_lifetime);
4          _service_lifetime = std::min(max,_SD_DB_lifetime);
5     }
6     srv->setTimeout(_service_lifetime);
7     listService.insert(srv);
8 }
```

La variabile **_SD_discard_not_guaranteed** è un parametro configurabile dall'utente di IBR-DTN attraverso l'apposito file di configurazione. Se viene impostato a *true*, quando viene aggiunto un nuovo servizio, viene assegnato un tempo di vita minimo al servizio e timeout troppo bassi vengono alzati. Ovviamente il servizio reale potrebbe non essere più raggiungibile alzando troppo il timeout ma la stabilità del database e il consumo energetico ne traggono beneficio. Dall'altro lato settare timeout troppo lunghi potrebbe corrompere l' efficacia della Service Discovery, per questo problema i servizi non possono avere un timeout maggiore di **_SD_DB_lifetime**. La continuità della presenza del servizio nel database è garantita dal refresh periodico tramite i messaggi di advertisement.

4.1.7 Configurazione

Durante la fase di avvio di IBR-DTN si può, attraverso un file di configurazione, impostare i parametri della Service Discovery. I parametri che possono essere configurati sono:

- **SD_Periodic_Time**: specifica il periodo temporale con il quale la classe SDWorker manda in multicast gli advertisement e allo stesso tempo reimposta il timestamp dei servizi.
- **SD_service_lifetime**: specifica il timeout da impostare ai nuovi servizi aggiunti, generalmente con un valore di 3 volte SD_Periodic_Time.

- **SD_lifetime**: tempo di validità di un bundle contente un messaggio *Request*, la gestione del bundles e la sua cancellazione è svolta all'interno di IBR-DTN.
- **SD_hop_counter_limit**: parametro che indica quanti hop può attraversare un bundle contente un messaggio di *Request*
- **SD_store_flag**: se viene settato ad "0" il nodo ignorerà i messaggi advertisement mandati in multicast, mentre analizzerà messaggi advertisement mandati in unicast, che sono le risposte al messaggio di solecitation.

4.2 Implementazione della Service Discovery

In questo capitolo si va ad analizzare nel dettaglio l'implementazione del modulo di service discovery all'interno del software IBR-DTN. Viene fornita sia la descrizione dell'architettura software e delle API fornite ai servizi esterni.

4.2.1 Architettura Interna

L' architettura della service discovery all'interno di un nodo IBR-DTN è composta da tre moduli: **SDWorker**, **SDEvent** e le classi contenenti le strutture dati chiamate **SDDataModels** (es. SDService, SDProperty, SDFunction, etc..). I moduli interni alla service discovery collaborano tra di loro e con quelli presenti in IBR-DTN al fine di popolare e gestire il database dei servizi disponibili sulla rete.

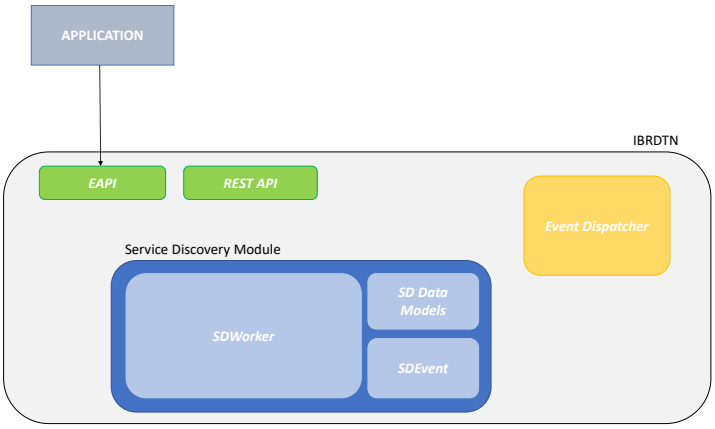


Figura 4.2. Architettura interna

4.2.2 Extended Application Programmi Interface (EAPI)

Questo modulo fa parte della struttura interna di IBR-DTN. Il suo scopo è quello di interfacciare IBR-DTN ed i suoi moduli, tra cui la service discovery, agli applicativi esterni. I comandi vengono passati all'Extended API attraverso uno streaming di dati basato su socket. Una volta ricevuti vengono inseriti nelle strutture dati opportune (grazie al PlainSerializer), validati (secondo uno schema di descrizione) e passati ai moduli incaricati di eseguire i comandi richiesti. L'Extended API si occupa anche di gestire lo stato dei servizi che vengono disconnessi. Infatti al suo interno mantiene la lista dei servizi attivi/inattivi relativa ad ogni applicazione. Inoltre nel momento in cui un applicativo si disconnette dal socket, tutti i servizi relativi ad esso vengono rimossi dal database e viene generato un *evento* (SERVICE_REMOVED), interno ad ibrdtn, che informa tutti i moduli sottoscritti alla Service Discovery della loro rimozione. Una nota, le API hanno un registro che mantiene l'ultimo servizio inserito. Se durante l'uso di un comando non si dichiara quale servizio si vuole utilizzare, quello nel registro sarà quello di preso in considerazione.

La lista dei comandi è la seguente:

- **service list <format>**
  Ritorna il numero e lista dei servizi nel formato preferito. I formati disponibili sono *plain* e *json*.

- **service add <json | plain>**
  Inserisce un nuovo servizio all'interno del database temporaneo dell'EAPI. Questo comando informa il nodo ibrdtn che si vuole aggiungere un nuovo servizio, ma non è ancora completo, per essere annunciato all'intera rete. Una volta inviato il comando, ibrdtn richiederà all'utente di inserire i parametri del servizio.

- **service del [service_name]**
  Permette di eliminare in maniera definitiva il servizio specificato, non sarà più possibile visualizzarlo.

- **service <subscribe | unsubscribe>**
  Permette di sottoscriversi o annullare la sottoscrizione alla ricezione delle notifiche degli eventi relative alla service discovery. Esempi di notifiche possono essere `SERVICE DISCOVERY <"EID"> REMOVED`

- **service function <append | del> <service name>**
  Permette di inserire, o rimuovere, una funzione al servizio indicato da service name. Nel caso si debba aggiungere una nuova funzione, si dovranno successivamente anche inserire i parametri della funzione.

- **service property <append | del> <service name>**
  Permette di inserire, o rimuovere, una proprietà al servizio indicato da service name. Nel caso si debba aggiungere una nuova proprietà, si dovranno successivamente anche inserire i parametri della proprietà.

- **service update function [service name] <function name>**
  Utilizzata quando si vuole modificare una funzione già inserita, se essa non è presente, ritorna un errore.

- **service update property [service name] <property name>**
  Utilizzata quando si vuole modificare una proprietà già inserita, se essa non è presente, ritorna un errore.

- **service activate [service name]**
  Serve per informare il nodo ibrdtn che il servizio è pronto ed utilizzabile dagli altri nodi presenti sulla rete.

- **service deactivate [service name]**
  Se invece si volesse, per esempio, fare manutenzione su un servizio, si potrebbe utilizzare questo comando, per informare il nodo ibrdtn che il servizio non è momentaneamente disponibile. Al contrario del comando "del", non rimuove definitivamente il servizio, infatti sarà possibile riattivarlo con il comando descritto in precedenza.

- **service request name <json | plain> service name**
  Ritorna tutti i dettagli di un servizio (categoria, funzioni, proprietà) in formato json o testuale (human readable). I servizi che saranno mostrati saranno quelli che conterranno all'interno del suo nome il "service name".

- **service request dtn <json | plain> serviceEID**
  Ritorna tutti i dettagli di un servizio (categoria, funzioni, proprietà) in formato json o testuale (human readable). Il servizio che verrà mostrato sarà soltanto uno, identificato dall' EID univoco.

- **service solecitation <EID>**
  Permette di richiedere la lista di tutti i servizi conosciuti ad un particolare nodo, identificato da "EID".

- **service find**
  Permette di richiedere alla rete ibrdtn, quali servizi sono presenti, avendo anche la possibilità di filtrarli grazie ai successivi parametri da immettere.

4.2.3 SD Data Models

L'SD Data Models è composto da una serie di classi che gestiscono il modello di dati usato dalla service discovery. Le classi riflettono il data model descritto nel json schema usato per validare la descrizione del servizio. Un vantaggio di trasporre il modello di dati in classi C++ è quello di poter serializzare e deserializzare agevolmente i dati (sempre in json) attraverso la libreria *nlohmann json* [14].

Le classi più interessanti da osservare sono:

- **SDService**
  Struttura completa del servizio. Comprende il nome, il device ID, l' endpoind dell'applicativo più le funzioni e la categoria del servizio.
- **SDCategory**
  Categoria del servizio, questa è una classe non utilizzata, vengono instanziate le classi che ereditano SDCategory. Infatti un servizio come descritto nel json schema puó avere diverse tipi di categoria (es. sensore, attuatore, computazione, web, etc..). Ogni categoria presenta proprietà tipiche di essa. Ogni proprietà puó essere obbligatoria o facoltativa, unica o ripetibile e alcune potrebbero anche avere dipendenze da altre properietà. La spiegazione e la struttura di ogni categoria viene spiegata nella sezione 5.1.3 "Struttura del servizio".

4.2.4 PlainSerializer

Il PlainSerializer è un modulo che utilizza l'ExtendeAPI per convertire i dati immessi dalle applicazioni in strutture dati appropriate. Come si puó vedere nel codice riportato, quello che si fa è sovrascrivere l'operatore ». Successivamente si prelevano i valori immessi dall'applicazione e si inseriscono nella struttura dati. Nell'esempio viene riportato il procedimento appena descritto per una proprietà del servizio.

```
9  dtn::data::Deserializer &PlainDeserializer::operator>>(dtn::data::SDProperty &obj) {
10             std::string data;
11             // read until the first empty line appears
12             while (_stream.good()) {
13                 getline(_stream, data);
14                 std::string::reverse_iterator iter = data.rbegin();
15                 if ((*iter) == '\r') data = data.substr(0, data.length() - 1);
16                 if (data.size() == 0) break;
17
18                 std::vector<std::string> values = dtn::utils::Utils::tokenize(":", data, 1);
19
20                 // if there are not enough parameter abort with an error
21                 if (values.size() <= 1) throw ibrcommon::Exception(" PARSING ERROR");
```

```
22
23                  if (values[0] == "Name") {
24                      obj.setName(dtn::data::SDService::removeFirst(values[1], ' '));
25                  } else if (values[0] == "Kind") {
26                    std::string value = dtn::data::SDService::removeFirst(values[1], ' ');
27
28                      if (strcasecmp(value.c_str(), "integer") == 0)
29                        obj.setKind("integer");
30            else if (strcasecmp(value.c_str(), "float") == 0)
31              obj.setKind("float");
32            else if (strcasecmp(value.c_str(), "string") == 0)
33              obj.setKind("string");
34                      else if (strcasecmp(value.c_str(), "boolean") == 0)
35                          obj.setKind("boolean");
36            else
37              throw dtn::InvalidDataException(" WRONG INPUT");
38
39                  } else if (values[0] == "Value") {
40                      obj.setValue(dtn::data::SDService::removeFirst(values[1], ' '));
41                  } else if (values[0] == "Unit") {
42                      obj.setUnit(dtn::data::SDService::removeFirst(values[1], ' '));
43                  } else
44                      //if is something that we do not know
45                      throw dtn::InvalidDataException(" WRONG INPUT");
46              }
47
48          [...]
49
50              return (*this);
51  }
```

Il PlainSerializer si occupa anche dell'operazione opposta sovrascrivendo l'operatore corrispondente «. Nell'esempio riportato si traduce la struttura dati in uno stream-socket per il client in lettura sulle API.

```
52  dtn::data::Serializer &PlainSerializer::operator<<(const dtn::data::SDProperty &obj) {
53          _stream << "Name: " << obj.getName() << std::endl;
54          _stream << "Kind: " << obj.getKind() << std::endl;
55          _stream << "Value: "<< obj.getValue() << std::endl;
56          _stream << "Unit: " << obj.getUnit() << std::endl;
57          _stream << std::endl;
58
59    return (*this);
60  }
```

4.2.5 Validazione del servizio

Quando un'applicazione collegata ad IBR-DTN vuole aggiungere un nuovo servzio esso deve attenersi alla descrizione che usa la Service Discovery. Per capire se il servizio che l'applicazione vuole attivare rispetti la descrizione mostrata nella sezione 4.1.2 (Struttura del servizio) bisogna analizzare e validare il servizio. Per validarlo viene usata la libreria *valijson* [15]. Quindi quello che si fa è:

1. Inserire il servizio nelle strutture dati attraverso il Plain Serializer.
2. Convertire la struttura dati in json attraverso la libreria *JSON for Modern C++*
3. Validare il servizio contro il json schema che descrive la struttura del servizio.

Se il servizio supera la validazione viene inserito nei servizi disponibili del nodo altrimenti viene ritornato un messaggio di errore verso l'applicazione indicando quali parametri non sono consoni alla descrizione del servizio. In modo tale da aiutare a capire l'errore commesso in fase di immissione del servizio.

Usando questo procedimento si ha il vantaggio di poter cambiare facilmente la descrizione del servizio ed eventuali vincoli senza andare a modificare codice all'interno di IBR-DTN.

4.2.6 Event Dispatcher

Questo modulo fa parte del pattern che permette di generare e ricevere eventi. Più in particolare ascolta i moduli che hanno intenzione di generare eventi e li inoltra a chi vuole riceverli. Ora verrà analizzato in dettaglio come funziona il pattern chiamato **Observer Pattern** e come viene implementato all'interno di IBR-DTN.

4.2.7 Eventi

All' interno della struttura di IBR-DTN i moduli possono informare l' intera struttura di eventi particolarmente significativi attraverso *l' Observer pattern*. Questo pattern è implementato da IBR-DTN ma è doveroso mostrarlo per capire come le informazioni possono essere inviate a tutti i moduli interni di IBR-DTN, tra cui Service Discovery e Virtual Service, in maniera semplice.

La classe principale è quella chiamata *EventDispatcher*. Essa si occupa di gestire gli eventi. Le classi interessate a ricevere gli eventi, chiamate *EventReceiver*,

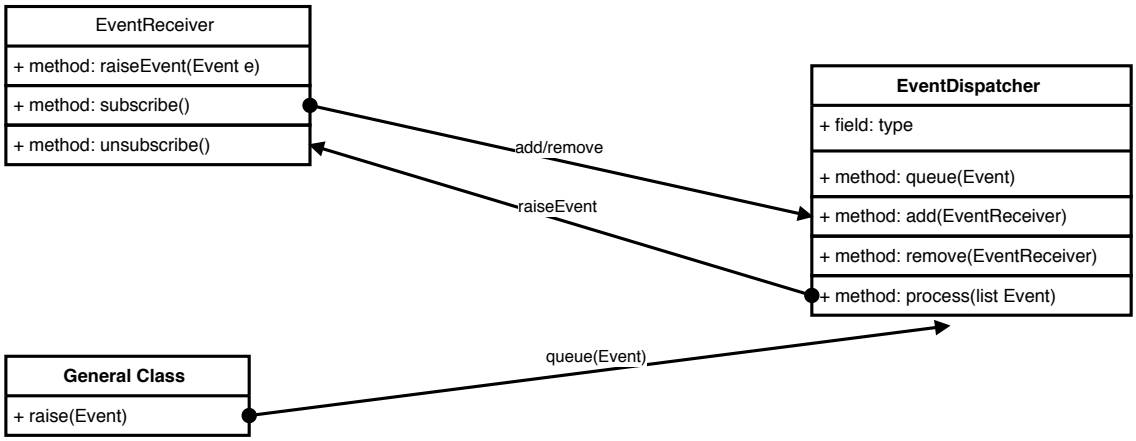


Figura 4.3. Observer Pattern Schema

attraverso i metodo pubblici `add(EventReceiver)` e `remove(EventReceiver)` dell'EventDispatcher, si possono iscrivere o disiscrivere alla ricezione degli eventi.
Le classi che hanno intenzione di generare eventi all'interno della rete, possono farlo chiamando il metodo `EventDispatcher::queue(Event)`, il quale inserisce in una coda la richiesta della generazione dell'evento. Quando è il momento di generare l' evento, l' EventDispatcher lo invia a tutti gli oggetti EventReceiver che sono iscritti, usando il metodo `EventDispatcher::process(Event)`.

```
61 void process(const Event *evt){
62   ibrcommon::MutexLock l(_dispatcher._dispatch_lock);
63
64     for (typename std::list<EventReceiver<E>*>::iterator iter = _dispatcher._receivers.begin();
65       iter != _dispatcher._receivers.end(); ++iter){
66       EventReceiver<E> &receiver = (**iter);
67       receiver.raiseEvent(static_cast<const E&>(*evt));
68     }
69
70     _dispatcher._stat_count++;
71 }
```

Come si puó vedere nel codice sopra riportato, per ogni instanza degli EventReceiver, l' EventDispatcher chiama il loro metodo pubblico `EventReceiver::raiseEvent()`. Una volta chiamato questo metodo, sarà la classe ricevente a decidere cosa fare di questa informazione. Infatti la classe **Event**, che viene inserita nel metodo raiseEvent, porta con sè messaggi e dati utili a capire cosa ha scatenato l'evento e come comportarsi di conseguenza. Gli eventi che vengono generati dalla Service Discovery sono sostanzialmente tre:

- SERVICE_DISCOVERED, quando un nuovo servizio viene scoperto o aggiunto dalle applicazioni locali.
- SERVICE_UPDATED, quando un servizio viene modificato.
- SERVICE_REMOVED, quando un servizio non è più disponibile.

4.2.8 SDWorker

Questo modulo presente nella service discovery è il centro che controlla tutta la service discovery, gestisce e coordina tutte le attività descritte fino ad adesso. In questa sezione si analizzarà nel dettaglio come riesce a fare ciò, mostrando anche esempi di implementazione e mostrando le criticità e come vengono risolte.
SDWorker si occupa di molteplici compiti:

- *Annuncio Periodico (Advertisement Messagge)*: periodicamente si occupa di mandare in multicast sulla rete un messaggio contenente la lista dei servizi che il nodo può offrire. Il periodo di tempo, con cui genera ed invia i messaggi, ha un valore di default (5 secondi), modificabile attraverso il file di configurazione a seconda della variabilità del nodo e del servizio.
- *Rimozione servizi "old"*: con lo stesso *clock*, o periodo temporale, con il quale vengono inviati i messaggi di advertisement, la classe SDWorker si occupa di scansionare la lista dei servizi presenti nel database e verificare che siano ancora validi. Ovvero facendo una differenza tra il tempo in cui sono stati aggiunti(o refreshati) e il tempo corrente. Se il risultato di questa differenza supera il tempo di validità del servizio, esso viene rimosso dal database dei servizi.
- *Comunicazione*: generalmente questa classe ha il compito di comunicare con le istanze della service discovery attivate su i nodi presenti nella rete. Perciò ha la capacità di ricevere i messaggi, analizzarli ed eseguire le conseguenti azioni necessarie. Attraverso queste comunicazioni un nodo è in grado di ottenere informazioni sui servizi presenti sulla rete, cercare servizi che rispettano filtri e capire quando un servizio viene aggiunto o rimosso aggiornando il proprio database.

Questo modulo è sostanzialmente un thread che aspetta e reagisce alle chiamate o agli eventi generati all'interno o all' esterno della service discovery stessa. I principali eventi a cui reagisce sono:

- Clock periodico, configurabile attraverso l'apposito file.(default 15 secondi).

- Bundle in arrivo.
- Nuove connessioni.

Periodic Timer

Di default ogni 15 secondi viene chiamata la funzione **callBackPeriodicTimer()**. Essa presenta due obiettivi principali. Il primo è quello di rimuovere i servizi "old" mentre il secondo è quello di mandare in multicast la lista dei servizi offerti dal nodo. Per raggiungere il primo obiettivo, viene scansionata tutta la lista dei servizi, tranne quelli locali (disponibili sul nodo in questione), e si guarda se dall'ultimo refresh del timestamp, ovvero da quando si è ascoltato un suo advertisement, è passato più tempo della lifetime del servizio, si puó vedere questo nel codice riportato.

```
72 long now_ms = time_since_epoch()).count();
73 long timestamp_ms = service->get_timestamp();
74 auto timelimit = milliseconds{service->getTimeout();
75
76 if(difference >= timelimit){
77     dtn::core::SDEvent::raise(SDEvent::SERVICE_REMOVED, "dtn://" +
           service->getDeviceNodeID() + "/" + service->getName());
78     serviceList.remove(service);
79 }
```

Ogni 15 secondi ci si deve occupare di mandare in multicast la lista dei servizi locali. Quindi bisogna creare il bundle e trasmetterlo in multicast. Per limitare i pacchetti spediti, non si trasmettono messaggi di advertisement se non sono ancora presenti servizi oppure se il nodo è isolato e non ha vicini. Questo perchè anche se non può mandare il messaggio, per come funziona ibr-dtn, viene inserito nello storage e quando si connette qualche nodo vengono spediti tutti insieme, tutto ciò, genererebbe un traffico aggiuntivo e sostazialmente senza portare informazioni aggiuntive. Nel codice riportato sotto viene mostrato come viene settato un bundle del tipo ADVERTSEMENT.

```
80 std::vector<shared_ptr<SDService>> servicesDatabaseVector = listService.getServiceList();
81
82 Bundle outgoingBundle;
83 outgoingBundle.source = getWorkerURI();
84 outgoingBundle.destination = SERVICE_DISCOVERY_ENDPOINT;
85 outgoingBundle.set(dtn::data::PrimaryBlock::DESTINATION_IS_SINGLETON, false);
86 ibrcommon::BLOB::Reference blobReference = ibrcommon::BLOB::create();
87 outgoingBundle.push_back(blobReference);
88
89 auto &blobPayloadStream = *blobReference.iostream();
90
```

```
SDServiceList outgoingLocalServicesList;

for (auto& service : servicesDatabaseVector) {
    if (service->isLocal()) {
        outgoingLocalServicesList.insert(service);
    }
}

json jsonOutgoingLocalServicesList;
SDPacket outgoingPacket;
json jsonOutgoingPacket;
outgoingPacket.setPacketType(SDPacket::PacketType::SERVICE_ADVERTISEMENT);
jsonOutgoingLocalServicesList = outgoingLocalServicesList;
outgoingPacket.setPayload(jsonOutgoingLocalServicesList.dump());
jsonOutgoingPacket = outgoingPacket;
local listservice in the stream payload
(*blobReference.iostream()) << jsonOutgoingPacket << endl;
```

Ricezione bundles

Quando arriva un bundle diretto al modulo della service discovery viene chiamata la funzione **callBackBundleReceived(Bundle b)** dove **b** è il bundle ricevuto. SDWorker si aspetta che i bundles in entrata contengano uno dei messaggi descritti nella sezione 4.1.1. Per leggere i messaggi si usa l'operatore « nello stesso modo di come si deserializza i dati il PlainSerializer ma al posto di deseriallizzare dati in un formato *plain* dal socket, si deserializzano i dati nel formato json contenuti nel bundle. SDWorker a seconda del messaggio che riceve reagisce in maniera diversa:

- **Service advertisement**:
  Per ogni servizio presente all'interno del messaggio esso viene inserito nel database, e settato il timestamp , se non fosse presente. Altrimenti, nel caso il servizio fosse già conosciuto viene solamente aggiornato il timestamp del servizio. Lancia l'evento SERVICE_DISCOVERED.
- **Service Updated**: Questo pacchetto può essere di tre tipi:
  - *Service Added*: stesso comportamento di Service Advertisement.
  - *Service Changed*: sostituisce il vecchio servizio con il nuovo aggiornato se presente, altrimenti aggiunge il nuovo servizio. Lancia l'evento SERVICE_UPDATED.
  - *Service Removed*: rimuove il servizio in questione. Lancia l'evento SERVICE_REMOVED.

- **Service Request**: Filtra il database dei servizi conosciuti in base alla richiesta e risponde in unicast al richiedente.
- **Service Solecitation**: Risponde in unicast al richiedente inserendo nel messaggio di ritorno tutti i servizi a lui conosciuti.

Nodo connesso

Quando un nuovo nodo si connette con un altro presente sulla rete, vengono scambiati in unicast i rispettivi database, attraverso il meccanismo della solecitation, in modo da non dovere aspettare un messaggio del tipo ADVERTISEMENT e quindi velocizzando la service discovery e diminuendo la latenza nella ricerca di un servizio. Questo comportamento è quello di default ma esiste un parametro configurabile per evitare di scambiarsi i database.

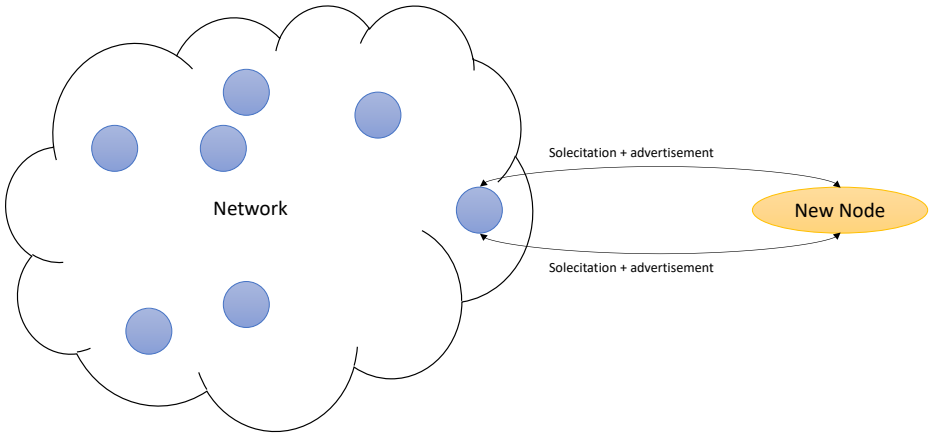


Figura 4.4. Meccanismo di sincronizzazione

Capitolo 5

Virtual Service

In una rete service oriented un servizio particolarmente utile è la capacità della rete di semplificare la connessione verso un servizio. Infatti nella rete potrebbero essere disponibili molti servizi con funzionalità simili. Essi si differenziano per le proprietà. Per esempio un sensore potrebbe essere preferibile rispetto ad un'altro con accuratezza minore. L'utilità del Virtual Service è quella di saper scegliere il servizio migliore tra i due e dirigere il traffico verso un altro servizio analogo, anche peggiore, nel caso il migliore fosse guasto o disconnesso.

5.1 Funzionamento e caratteristiche

Obiettivo di questa sezione è quello di analizzare il concetto di "servizio virtuale", mostrando come viene instaurata una connessione tra client e service provider. Successivamente viene mostrato il concetti di trasparenza del Virtual Service per i nodi.

5.1.1 Architettura

Come mostrato nella figura qui sopra si può vedere insieme all'architettura anche il caso d'uso. Sulla sinistra viene rappresentato un applicativo, esterno ad IBR-DTN, che vuole trovare ed usare un servizio, in questo caso un sensore che misuri la temperatura dell'ambiente. Nell'esempio sono presenti 2 servizi, chiamati "Sensor A" e "Sensor B", dove il sensore A è migliore del servizio B, magari per risoluzione o affidabilità. L' obiettivo è quello di far comunicare client e server nella maniera più semplice possibile, senza che il client si debba interfacciare con la service discovery e fare azioni particolarmente complesse.

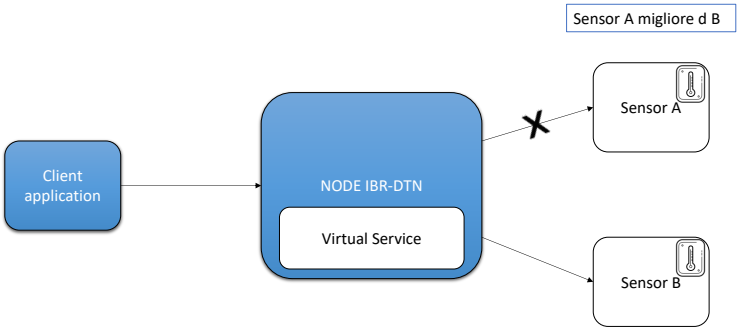


Figura 5.1. Architettura generale Virtual Service

Extended API relative al Virtual Service

A differenza delle API della service discovery, quella del Virtual Service presenta un solo comando. Questo è un primo indizio per capire come lo sviluppo di questo modulo sia orientato a semplificare la connessione tra client e service provider.

- **service search** questo è il comando iniziale per invocare il virtual service, successivamenete verrà chiesto al client di inserire i parametri per capire come selezionare il miglior servizio.

```
1
2  IBR-DTN 1.0.1 (build fb0cef3) API 1.0.1
3  protocol extended
4  200 SWITCHED TO EXTENDED
5  set endpoint test1
6  200 OK
7  service search
8  PUT REQUEST
9  Name: printer
10 Parameter: resolution less
11
12
13 dtn://asus/VSprintertest1
14 200 SEARCH
```

Funzionamento del Virtual Service

Come detto, il ruolo del servizio virtuale è quello di semplificare l'uso dei servizi da parte del client. Per raggiungere questo scopo il virtual service si deve occupare di alcuni compiti essenziali.

- **Trovare i servizi**
  Il primo passo che deve eseguire per raggiungere il suo scopo è quello di trovare i servizi disponibili sulla rete. Questo passaggio è molto semplice poichè si risolve con una chiamata al modulo della Service Discovery (**getServiceList()**), il quale ritorna il database con un tutti i servizi offerti dalla rete.

- **Confrontare i servizi**
  Una volta ricevuto il database con i servizi, deve capire qual è il servizio migliore in base alle richieste del client, espresse tramite attributo preferibile di una proprietà, come sensibilità, tollerenza oppure fondo scala nel caso di un sensore. Questo si risolve facendo una ricerca lineare sulla lista dei servizi, cercando il massimo (o il minimo) della proprietà richiesta.

- **Connettere client-server**
  Ora che il Virtual Service ha scoperto il servizio migliore, si deve occupare di connettere il client con il servizio. Per eseguire questo compito, e rendere trasparente l'utilizzo per il client, il virtual service restituisce al client un indirizzo destinazione, non reale ma solo simbolico, che identifica il virtual service. Il client manda i bundles a questo indirizzo. Il Virtual Service quando li riceve li inoltra al servizio reale. La stessa cosa, ma nella direzione opposta, per le risposte del servizio.

A questo punto il client vede una connessione statica e stabile, quindi non si deve più preoccupare di nulla, nè che il servizio cada nè di usare quello migliore. Infatti il Virtual Service reindirizza il flusso di dati verso il nodo migliore in maniera automatica, senza che il client si accorga delle varie problematiche.

5.2 Architettura software

La struttura interna del "Virtual Service" è composta essenzialmente da due moduli, il primo modulo si chiama "VS Handler" mentre il secondo, quello che effettivamente si comporta da servizio virtuale e quindi da ponte tra applicativo e servizio si chiama "VSWorker". Quest'ultimo riceve tutte le informazioni di cui a bisogno dal VSHandler che a sua volta ha la possibilità di interfacciarsi con il modulo della Service Discovery.

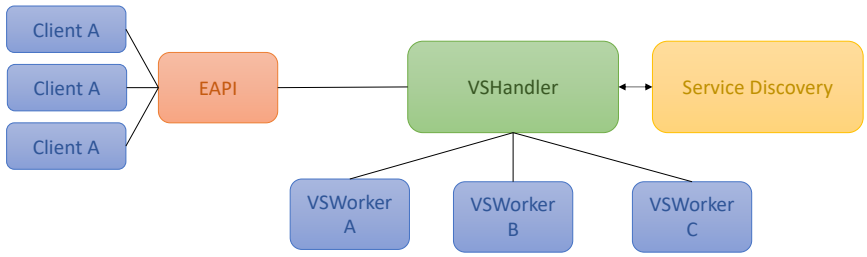


Figura 5.2. Architettura interna Virtual Service

5.2.1 Gestore dei servizi virtuali

Il gestore dei servizi virtuali, o VSHandler, è un modulo cosidetto *singleton*, ovvero istanziato solo una volta, questo perché deve gestire svariati moduli. Esso si occupa di ricevere le richieste provenienti, attraverso le Extended API, del client e istanziare il VSWorker dando a lui, di volta in volta il miglior servizio per il client. Quindi andando a vedere in dettaglio quello che succede:

1. Il client manda una richiesta, attraverso le api, per trovare il miglior servizio. Nella richiesta ha la possibilità di inserire filtri di preferenza, come la sensibilità minima o il range massimo di valori accettati.

2. A questo viene chiamato il metodo **VSHandler::bind()**, come dice il nome crea un legame tra client e virtual service. Il suo compito è quello di instanziare il modulo del VSWorker e fornire ad esso il miglior servizio.Per trovare il migliore servizio, e quindi l' EID, chiama il metodo **VSHandler::findBestService()**, che scorrendo tutto il database dei servizi, ottenuto attraverso la service discovery, cerca il servizio che rispetti le richieste del client.

3. Infine ritorna l'indirizzo simbolico del virtual service appena instaziato

Una volta eseguiti questi due passaggi, si è creato un collegamento tra client e server. Tuttavia il VSHandler rimane comunque attivo per due motivi. Il primo motivo è quello di accettare nuovo richieste da altri applicativi presenti sul nodo. Il secondo motivo invece è quello di fornire un' interfaccia tra il VSWorker ed il modulo della Service Discovery, in particolare quando è necessario spostare il flusso dati per qualche motivo (nodo non più raggiungibile o servizio inattivo per esempio), esso si occupa della ricerca del nuovo miglior servizio.

5.2.2 Servizio Virtuale (VSWorker)

Questo modulo, a differenza del precedente, non è singleton, ma viene istanziato tante volte quante sono le richieste di connessione con un servizio. Una volta istaziato, il suo compito principale è quello di inoltrare i bundles ricevuti dal client fini al miglior servizio e naturalmente il viceversa.

Il client, ora, non manda bundles direttamente al servizio, ma li trasmette al virtual service, inserendo come destinatario un indirizzo simbolico che ha come forma la seguente:

$$dtn://<host\_nodename>/<ServiceName><RAND>$$.

Il paramentro "RAND" serve per distinguere in maniera univoca tutti i servizi virtuali che vengono istanziati. Questo indirizzo viene ritornato da VSHandler quando un client richiede la funzione di servizio virtuale.

Una volta ricevuto il pacchetto, il virtual service semplicemente sostituisce:

1. L'indirizzo di destinazione con quello reale del servizio
2. l'indirizzo mittente con il proprio, in questo modo il servizio reale risponderà al virtual service, che farà da ponte. Questo per rendere il virtual service trasparente in entrambe le direzioni.

```
108 proxy_packet.destination =
        dtn::data::EID("dtn://"+_best_service_node_id+"/"+_best_service_endpoint);
109
110 proxy_packet.source = getWorkerURI();
111
112 transmit( proxy_packet );
```

Una volta sostituiti opportunamente gli indirizzi del bundle, esso viene inoltrato attraverso il metodo **transmit(packet)**.

Switching trasparente

Questa è la connessione "base" tra client e service. Il virtual service tuttavia ha come obiettivo principale la trasparenza dello switch del flusso di dati nel caso il servizio migliore dovesse cambiare. Le motivazioni per cui serve cambiare servizio sono sostanzialmente 3:

- Il servizio non è piu attivo.
- Il nodo su cui risiede il servizio non è più raggiungibile.
- Un servizio migliore viene aggiunto alla lista dei servizi disponibili sulla rete.
- Un servizio viene aggiornato e sarebbe il migliore.
- Il servizio corrente viene aggiornato cancellando lo stato di migliore.

Il virtual service per capire quando si ha la necessità di cambiare il miglior servizio, si affida agli eventi di ibr-dtn, in particolare, generati dalla service discovery. Gli eventi a cui è interessato e perché lo è sono:

- `SERVICE_DISCOVERED` :
  Questo è un indicatore che un nuovo servizio è stato scoperto sulla rete, se fosse uno migliore di quello attualmente usato bisognerebbe fare lo switch.
- `SERVICE_UPDATE` :
  Questo evento indica che un servizio presente sulla rete è stato aggiunto, modificato o cancellato. Il caso che interessa al virtual service è se ha subito modifiche. Questo perché esso potrebbe essere diventato il miglior service per il client.
- `SERVICE_REMOVED` :
  Questo evento serve per indicare che un servizio non è più disponibile sulla rete e quindi bisognerà switchare su un servizio, anche peggiore, al più presto possibile.

Una volta ricevuto l'evento si comporta di conseguenza. Nel caso fosse un evento di service discovered, cerca di capire in primo luogo se è stato aggiunto un servizio simile a quello che il client ha richiesto. In caso affermativo , confronta il servizio corrente con quello appena aggiunto. Se quest'ultimo presenta parametri migliori esso diventa il migliore ed il traffico rediretto.

Se invece l'evento si trattasse del secondo punto, ovvero un service update la procedura è simile a quella descritta sopra ma si analizza solamente il servizio modificato per capire se il servizio corrente sia ancora migliore.

Nell'ultimo tipo di evento, cioè service removed, ci si comporta in maniera differente. Per prima cosa si blocca il flusso di dati, per non perdere dati e mandarli a quello nuovo. Successivamente si richiede l'intervento del gestore dei servizi (VSHandler), il quale, utilizzando la service discovery, ottiene la lista dei servizi presenti e ritorna al modulo VSWorker il servizio migliore, il VSWorker a questo punto può dirigere il traffico verso il nuovo nodo scelto. In caso invece nessuno servizio fosse disponibile, il VSWorker semplicemente sta in attesa dell'evento service added, verificando che rispetti i parametri ed inserendolo come miglior servizio.

Capitolo 6

Test e risultati ottenuti

Per definire caratteristiche e performance dei due principali moduli sviluppati, in una situazione reale dove ogni nodo trasmette dati grazie alle diverse connessioni sporadiche, si è utilizzato un simulatore. Il simulatore utilizza lo "Universal Node" [16], sviluppato dal Politecnico di Torino. Esso ha la capacità di instanziare i nodi come docker containers e i link come regole di uno switch virtuale [17]. All'interno dei docker viene caricato il software ibrdtn esteso con i moduli della nostra service discovery ed attraverso universal-node cambiate le regole di openvswitch per simulare un cambio di topologia della rete.

6.1 Ambiente di simulazione

I test sono stati eseguiti su una macchina virtuale del Politecnico di Torino, alla quale sono stati assegnati 16 processori @2GHz 64bit, 8GB di RAM e 40GB di storage. La macchina che ospita la VM presenta two octa-core Intel XeonE5-2660 @ 2.2 GHz CPUs.

6.2 Temporizzazione Service Discovery

Di importanza fondamentale è la caratterizzazione dei tempi di scoperta dei servizi, in ambienti più o meno sfidanti. Con questo fine si sono analizzati i tempi medi di scoperta del servizio al variare di alcuni parametri chiave della rete.

6.2.1 Conduzione del test

Il test è coinvolge 5 tipi di servizi ed un numero variabile di nodi e connessioni. Ogni nodo ha una probabilità del 10% di contenere ognuno dei 5 servizi, ed è garantita la presenza di tutti e cinque i servizi. La durata media di ogni simulazione è di 30 minuti.

6.2.2 Funzionamento

1. Viene distribuito un servizio sui nodi
2. Il nodo client richiede sulla rete il primo servizio, salvando il timestamp.
3. Il nodo aspetta la risposta e calcola il tempo che essa ha impiegato ad arrivare.
4. Si ripetono i punti 1 e 2 per per ognuno dei cinque servizi. Dopodichè viene resettata la topologia. Questo per 20 iterazioni. Per un totale di 100 richieste di servizio.

6.2.3 Considerazioni

Cambio topologia

Il simulatore cambia la topologia della rete ogni 15 secondi per effettuare questa operazione di creazione e abbattimento dei link impiega del tempo, mentre si desidera simulare un cambio di topologia immediato. Per ottenere risultati che riflettono il comportamento ideale, ai tempi della risposta alla request vengono sottratti i tempi in cui la rete è sostanzialmente ferma.

6.2.4 Parametri variabili

Per capire l'impatto di una variabile sulle performance, si eseguono varie simulazioni, cambiando il valore di un parametro e tenendo gli altri fermi.
I parametri analizzati sono:

1. **Probabilità di connessione tra due nodi**
   Range: 10% - 70%.
2. **Percentuale di nodi directory**
   Range: 30% - 60%.

3. **Numero di nodi nella simulazione**
   Range: 15 nodi - 50 nodi.
4. **Hop Range iniziale**:
   Range: 1-4.

6.2.5 Analisi Risultati

Una volta eseguite le simulazioni si analizzano:

- **Tempo medio di risposta**
- **Distribuzione dei tempi**
- **Failure ratio**, rapporto numero richieste senza risposta e numero di richieste entro 5 minuti.

Analisi parametro: Percentuale Directory

- Numero nodi: 25
- Probabilità di link: 20%
- Percentuale directory: [30% fino al 70%]

Percentuale directory	30%	40%	50%	60%	70%
Tempo medio (s)	0.32	0.31	0.28	0.26	0.25
Failure rate	0%	0%	0%	0%	0%

Tabella 6.1. Tempi medi e failure rate del parametro percentuale directory.

Nella tabella qui sopra vengono mostrati i tempi medi (rappresentati nel successivo grafico), ed il failure rate. Poichè il servizio viene sempre trovato, il failure rate è sempre dello 0%.

Come si può vedere del grafico seguente, i tempi medi di scoperta del servizio diminuiscono aumentando la percentuale di directory. Questo perchè aumenta la probabilità che ci sia un nodo che sappia la posizione del servizio richiesto dal nodo client.

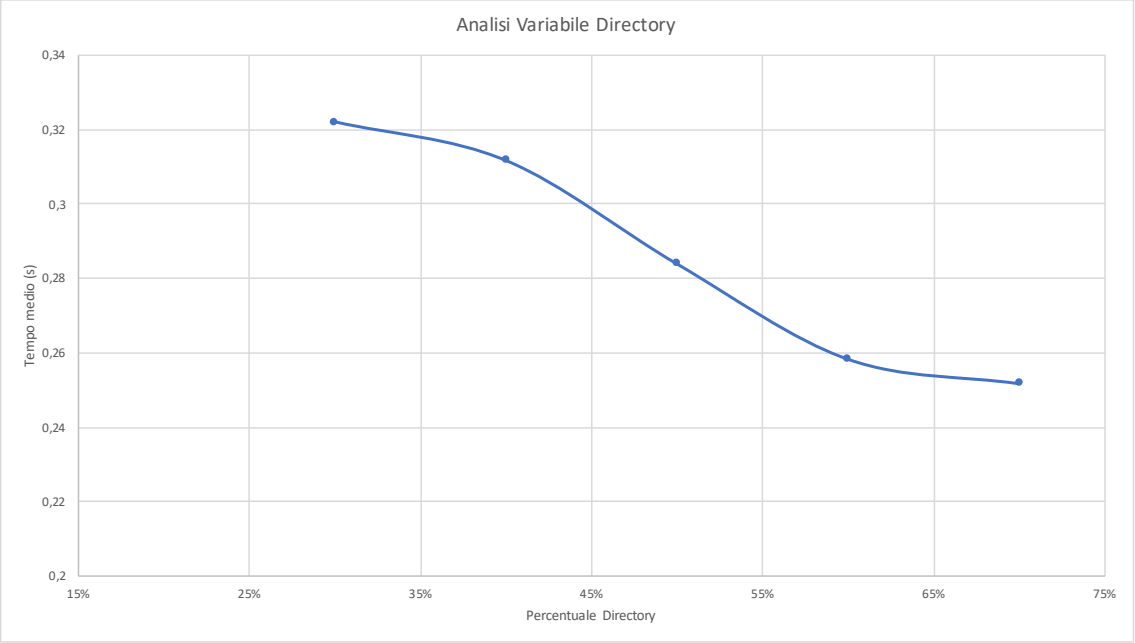


Figura 6.1. Tempo medio di risposta rispetto alla percentuale di directory

Analisi parametro: probabilità link

Questo parametro rappresenta la probabilità che 2 nodi qualsiasi all'interno della rete DTN siano connessi direttamente.

- Percentuale directory: 50%
- Numero di nodi: 25
- Probabilità di link: [dal 5% al 70%]

Nella tabella sottostante vengono riportate le misurazioni dei tempi medi di scoperta a seguito della simulazione e rappresentate graficamente nella figura successiva.

Probabilità link %	5%	20%	30%	40%	60%	70%
Tempo medio (s)	0.42	0.33	0.32	0.30	0.26	0.25
Failure rate	0%	0%	0%	0%	0%	0%

Tabella 6.2. Tempi medi e failure rate del parametro probabilità di link.

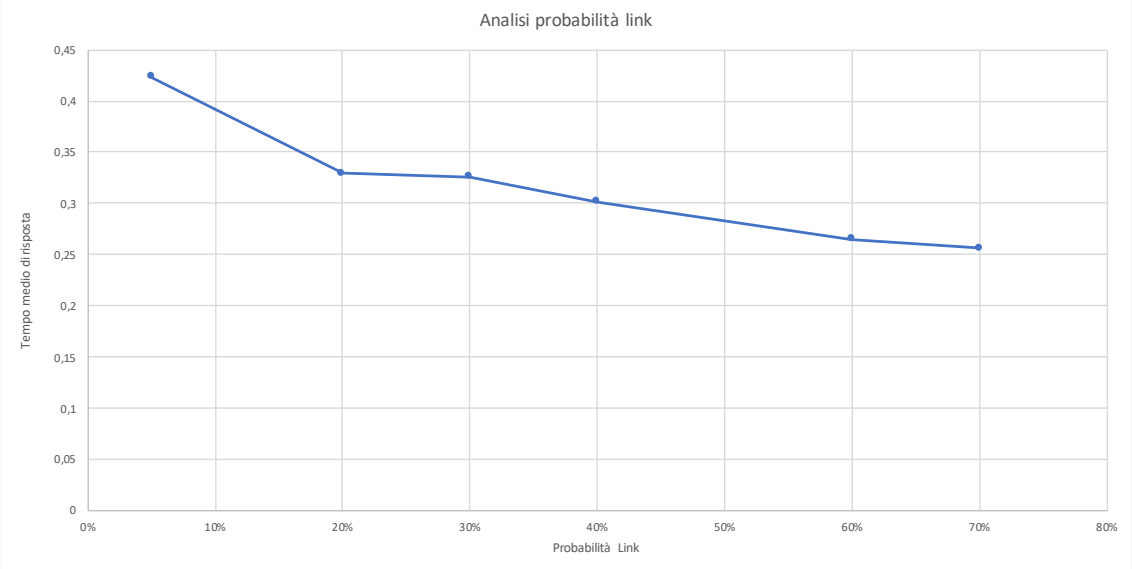


Figura 6.2. Tempo medio di risposta rispetto alla probabilità di link

Analisi parametro: numero nodi

- Percentuale directory: 50%
- Numero di nodi: [da 15 fino 50]
- Probabilità di link: 20%

Nella tabella sottostante vengono riportate le misurazioni dei tempi medi di scoperta a seguito della simulazione e rappresentate graficamente nella figura successiva.

Numero di nodi	15	20	25	30	35	40	45	50
Tempo medio (s)	0.12	0.16	0.18	0.26	1.18	2.85	3.9	4.1
Failure rate	0%	0%	0%	0%	0%	0%	4%	11%

Tabella 6.3. Tempi medi e failure rate del parametro probabilità di link.

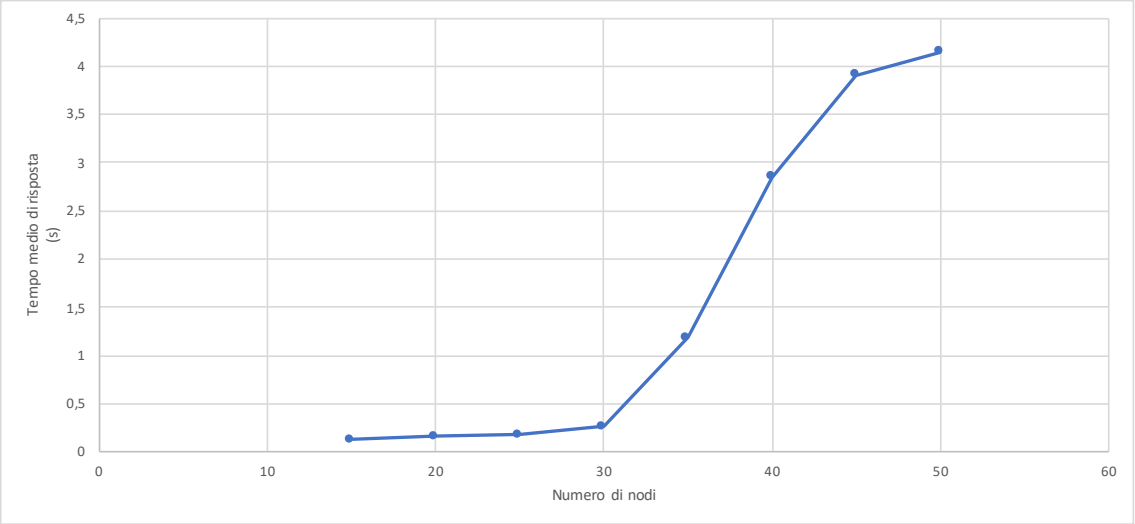


Figura 6.3. Tempo medio di risposta rispetto al numero di nodi.

Analizzando i risultati si vede come superando il numero di 30 nodi i tempi di risposta aumentano drasticamente. Questo comportamento si verifica poichè il diametro della rete aumenta e di conseguenza aumenta anche il tempo che impiegano i bundles ad attraversare la rete. Aumentando ulteriormente la dimensione della rete alcuni bundles superano il tempo della durata della simulazione.Essi vengono inseriti nel conteggio delle richieste senza risposta e si può notare questo evento quando la rete ha una dimensione di 45 nodi.

Analisi parametro: Hop range iniziale

Questa è un ulteriore analisi per capire quanto tempo ci mettano i servizi ad essere trovati, ricercando solo in una porzione di rete. Il nodo client manda un query limitando lo scope di ricerca e, in caso di timeout, aumentandolo linearmente.

- Numero di nodi: 25
- Probabilità di link: 10%
- Percentuale directory: 20%

Numero di hop di partenza	1	2	3
Tempo medio (s)	0.38	0.28	0.28
Max hop raggiunto	2	3	3

Tabella 6.4. Analisi porzione rete.

Quello che si vede è che entro 3 hop nel 100% dei casi viene trovato il servizio. Una volta superato il range di un hop, i tempi si stabilizzano intorno al valore 0.28 secondi e non ha più senso aumentare il range di rete alla ricerca del servizio. Quindi tutto ciò ci da un informazione su come settare il valore *hop_limiter* del bundle che contiene il pacchetto request. In questo modo si può ridurre in maniera significativa il numero di bundles generati sulla rete.

6.2.6 Conteggio Bundles

In questa analisi, sono stati contati quanti bundles la service discovery genera e la quantità di pacchetti progressivamente trasportati nello storage. In modo tale di avere una caratterizzazione per settare la service discovery in modo ottimale. La configurazione è la seguente:

- 25 nodi
- Probabiltà di link: 20%
- Percentuale Directory: 50%
- Epidemic routing
- Service Advertisement Period di 10 secondi
- Un solo servizio su un nodo.
- Cambio topologia ogni 5 secondi

Pacchetti generati da un ping

Per prima cosa si sono contati quanti bundles generasse IBR-DTN con una semplice trasmissione dati tra due nodi della rete, in questo caso si è scelto di generare un ping. Sulla rete usata sono stati generati 180 bundles, circa 360 se si conta pure il pacchetto di ritorno. Questo può servire come punto di riferimento per le analisi successive.

Pacchetti generati dalla Service Discovery

Nella simulazionione sono presenti 15 nodi, di cui soltanto uno ospita un servizio e quindi è l'unico a generare pacchetti di service advertisement. Il conteggio dei pacchetti generati è stato letto sulle statistiche interne ad IBR-DTN.

Nel seguente grafico si mostrano la quantità totale di bundle trasmessi nella rete.

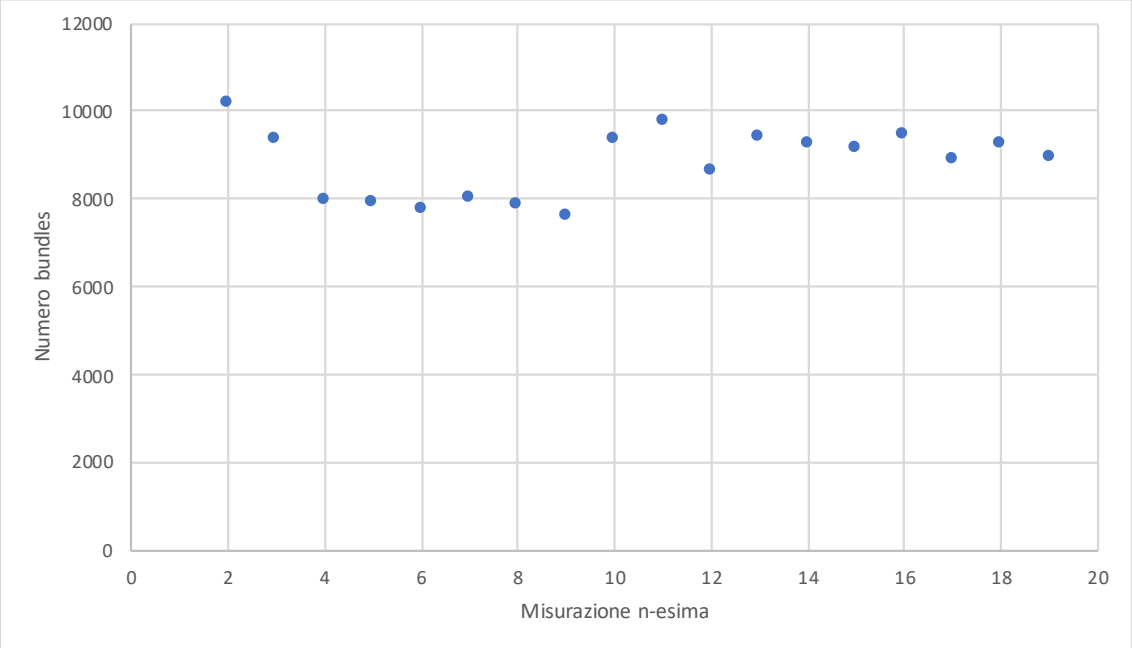


Figura 6.4. Bundles generati per ogni simulazione.

Se non sono presenti servizi all'interno della rete la service discovery non genera bundles.

Durante le simulazioni vengono generati in media 8812 bundles. Con un massimo di 10180 ed un minimo di 7740.

Analisi temporale dello storage

Scopo di questa analisi è quello di capire la quantità di bundle presente nella memoria dei singoli nodi. Per fare ciò si interroga periodicamente il nodo sulla quantità di bundle memorizzati attraverso le api messe ha disposizione da IBR-DTN.

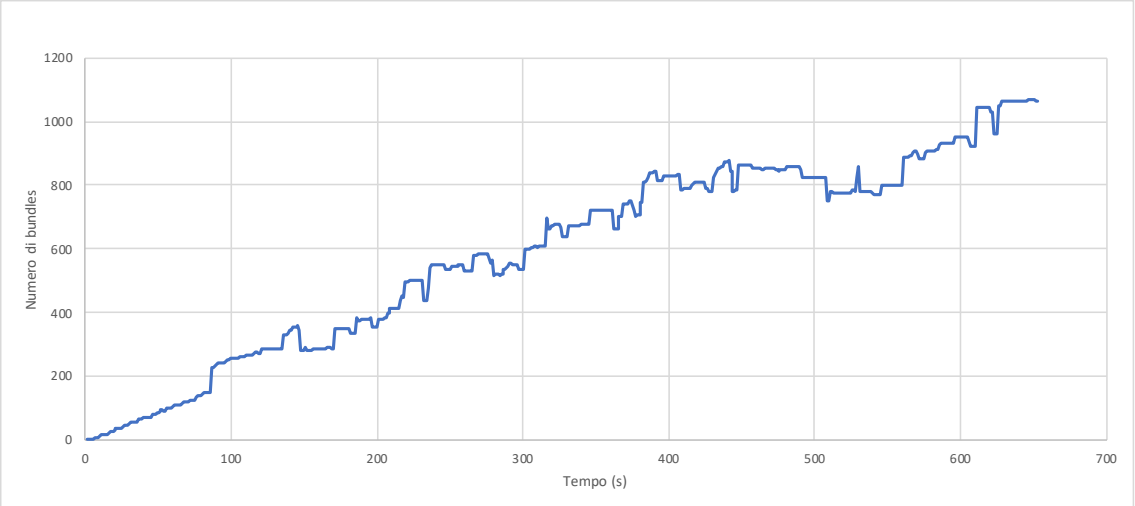


Figura 6.5. Storico storage

In figura 6.5 si può vedere come lo storage aumenti progressivamente col passare del tempo, questo comportamento si verifica poichè il protocollo di routing utilizzato è di tipo epidemico, ovvero che per mandare un bundles da un nodo A ad un nodo B, lo si manda in flooding su tutta la rete, fino a quando non raggiunge la destinazione. Quindi si capisce che con un protocollo di routing epidemico, non si riescono a smaltire con lo stesso ritmo i pacchetti che vengono generati. Si può inoltre notare come i bundles vengano rilasciati, questo si verifica quando scade il lifetime del bundle o si raggiunge la destinazione.

6.2.7 Analisi Virtual Service

In questa sezione vengono analizzate le performance relative al virtual service. I test che si andranno a svolgere sono:

- Latenza di connessione.
- Confronto rispetto alla connessione diretta.

Topologia di simulazione La topologia su cui andremo a svolgere i test è la seguente:

- Un nodo client che si connette al miglior servizio usando il virtual service.
- Un nodo provider, *"best"*, in cui è presente il servizio migliore.
- Un nodo provider, *"worst"*, dove viene inserito un servizio peggiore rispetto a quello precedente.
- Un nodo di collegamento a tutti e tre i nodi.

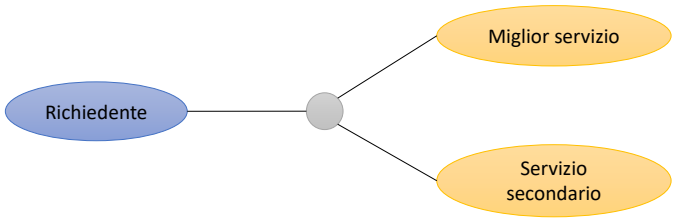


Figura 6.6. Topologia test per il virtual service

Latenza di connessione

Questo test analizza il tempo totale da quando il client richiede un servizio fino a quando la comunicazione è stabilita. Per calcolare questo valore quello che si fa è:

1. Il nodo client richiede il servizio, e fa partire il cronometro.
2. Una volta ricevuto l'indirizzo del VS manda un pacchetto *echo*.
3. Il provider quando riceve *echo* risponde un messaggio uguale.
4. Quando il client riceve la risposta interrompe il cronometro.

Le operazioni descritte sono state ripetute 100 volte, ottenendo una latenza media di 0.13 secondi. La distribuzione dei tempi di latenza ottenuti è visibile in figura 6.7.

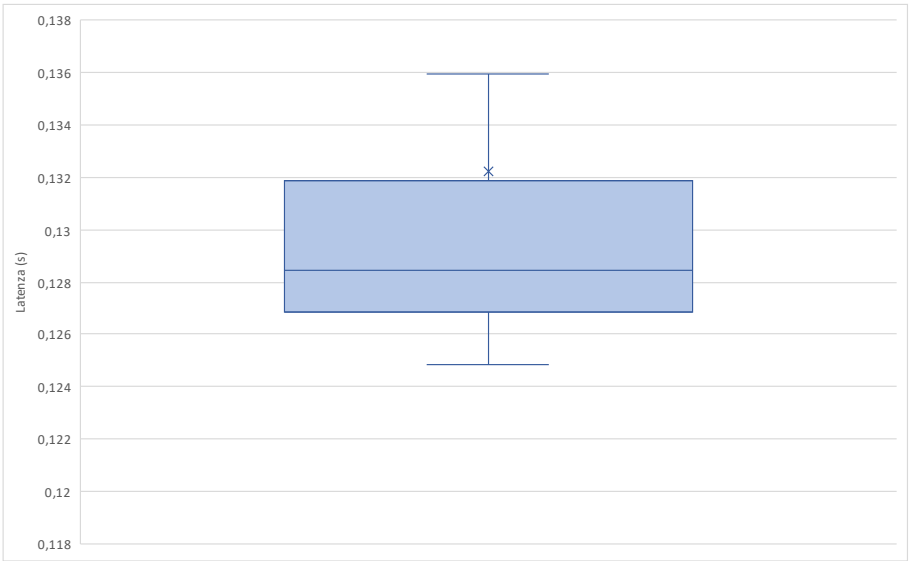


Figura 6.7. Distribuzione dei tempi di latenza

Confronto con connessione diretta

In questa sezione si analizza quanto tempo occorre in più al virtual service rispetto alla connessione diretta per mandare un file delle dimensioni di 10MB, 100MB e 500MB. Per questo confronto la topologia della rete è la stessa del test precedente.Rappresentando i risultati ottenuti si nota meglio come aumentando la dimensione del file da inviare la differenza sia più netta, ma comunque di solo qualche decimo di secondo.

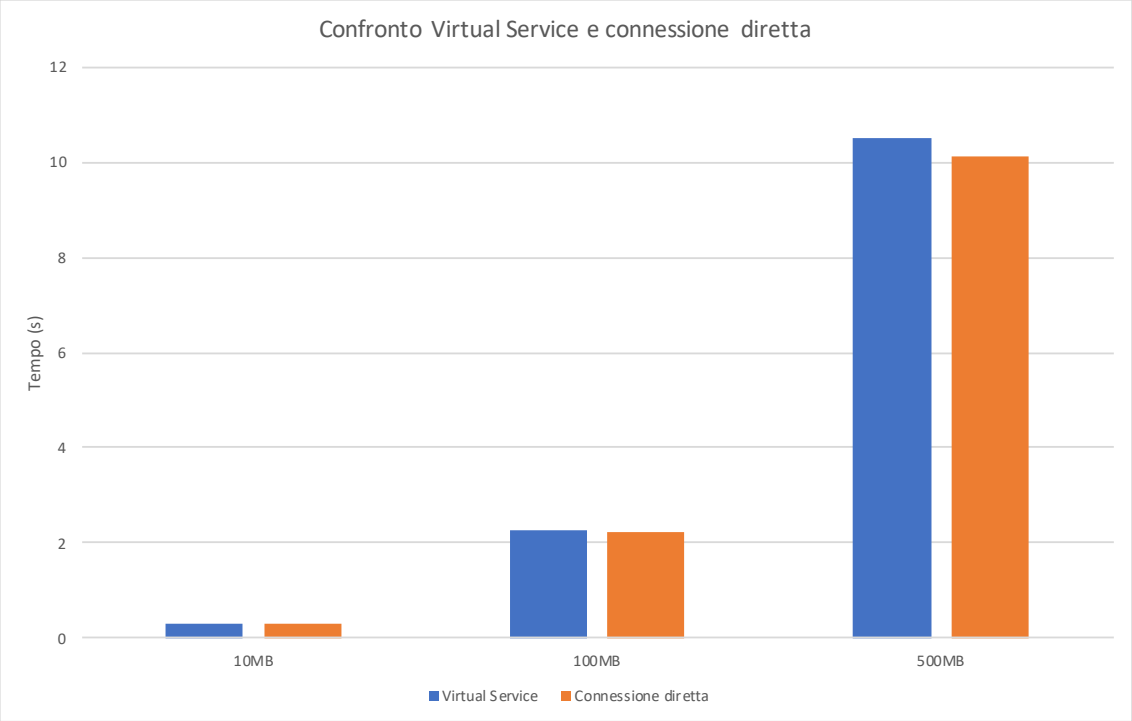


Figura 6.8. Confronto trasferimento file con Virtual Service e connessione diretta

Capitolo 7

Conclusioni

In questa tesi è stato presentato un progetto svolto in collaborazione con Tierra S.p.A., azienda che lavora nel settore Industrial IoT. Il progetto ha avuto come obiettivo abilitare lo scambio di servizi all'interno di una Delay Tolerant Network (DTN). A tal fine si sono implementati due moduli: la Service Discovery ed il Virtual Service. Il primo modulo ha come scopo quello di diffondere la conoscenza dei servizi sull'intera rete, mentre il secondo si pone l'obiettivo di semplificare l'utilizzo dei servizi ai nodi client, che non devono preoccuparsi di conoscere l'effettiva ubicazione del servizio che stanno utilizzando. Il design finale prevede una architettura ibrida, che può comprendere sia nodi che agiscono come directory che nodi che non memorizzano i servizi esistenti, e permette di propagare i servizi sia in maniera reattiva che proattiva. I nodi directory possono infatti apprendere i servizi in maniera passiva, per via della propagazione periodica degli stessi, mentre altri nodi "light" hanno comunque la possibilità di effettuare una query che gli permetta di conoscere l'esistenza di un servizio solo nel momento del bisogno. Infine, effettuando diverse simulazioni, si è caratterizzata in termini di risorse utilizzate, la Service Discovery ed il Virtual Service. Per il modulo della Service Discovery, quello che si evince è che aumentando il numero di nodi i tempi medi di scoperta aumentano drasticamente ed in alcuni casi non vengono trovati in un tempo limite. Questo avviene perchè il tempo necessario ad un bundle per attraversare la rete è naturalmente maggiore. Anche andando a diminuire le connessioni tra nodi sulla rete o i nodi che si comportano come directory i tempi di scoperta crescono, ma non cosí drasticamente come nel caso del numero dei nodi. Per quanto riguarda le risorse di storage utilizzate si nota facilmente come un routing di tipo epidemico sia poco adatto per dispositivi con poca memoria. Il dispositivo infatti si troverà davanti al problema di dover scartare i bundles che non è più in grado di conservare, diminuendo le prestazioni della rete. È stata inoltre misurata la distanza media che un messaggio di request percorre per trovare un nodo che sappia rispondere alla query. Si è visto che entro i primi 2 hop si ha quasi la totale certezza di poter localizzare il servizio. Questo dato ritorna molto

utile su una rete di questa tipologia perchè permette di limitare i bundles generati e utilizzare meno risorse sui nodi. Per quanto riguarda il Virtual Service, è stata misurata la latenza che intercorre tra la richiesta del servizio fino a quando diventa possibile consumare lo stesso, ottenendo tempi trascurabili nel caso di topologia connessa. I tempi registrati, in una rete dove client e service provider sono connessi attraverso un terzo nodo, sono dell' ordine di 130 millisecondi. Successivamente è stato misurato l'overhead introdotto dall'utilizzo di un Virtual Service rispetto ad una connessione diretta, per un servizio di scambio file. Quello che si vede è che i tempi di trasferimento sono leggermente più alti, di qualche decimo di secondo, se si utilizza il Virtual Service. Questo è dato principalmente dal tempo iniziale per instanziare la connessione attraverso il servizio virtuale e dalle operazioni di modifica (sorgente destinazione) ed inoltro dei bundles inviati dai dispositivi end-to-end. Una parte sui cui si può ancora lavorare per migliorare il servizio offerto è l'algoritmo di routing. Infatti un algoritmo epidemico è evidentemente poco efficiente dal punto del traffico generato, anche se garantisce la consegna del pacchetto. Per quanto riguarda le possibili evoluzioni della Service Discovery e della Virtual Service sarebbe molto utile ed interessante avere un interfaccia standard per usufruire dei servizi, senza essere legati in nessun modo alla semantica ed ai protocolli del service provider.

Elenco delle tabelle

Elenco delle figure

Bibliografia

[1] Christopher N. Ververidis and George C. Polizos. Service discovery for mobile ad hoc networks: a survey of issues and techniques. *IEEE Communications Surveys & Tutorials,Volume: 10 , Issue: 3 , Third Quarter 2008*, 2008.

[2] K. Scott. Disruption tolerant networking proxies for on-the-move tactical networks. In *MILCOM 2005 - 2005 IEEE Military Communications Conference*, 2005.

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

[4] S. Schildt, J. Morgenroth, W.B. P¨ o ttner, and L. Wolf. Ibr-dtn: A lightweight, modular and highly portable bundle protocol implementation. *Electronic Communications of the EASST, Volume 37: Kommunikation in Verteilten Systemen 2011*, 2011.

[5] Johannes Morgenroth. Ibr-dtn api. https://github.com/ibrdtn/ibrdtn/wiki/API.

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

[7] M. Demmer, J. Ott, and S. Perreault. Delay-tolerant networking tcp convergence-layer protocol. RFC 7242, IETF, June 2014.

[8] Johannes Morgenroth. Ibr-dtn configuration file example.

[9] Service location protocol administration guide. https://docs.oracle.com/cd/E19455-01/806-1412/6jamu408b/index.html.

[10] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. RFC 7242, IETF, June 1997.

[11] Bendaoud Karim Talal and Merzougui Rachid. Service discovery – a survey and comparison. *International Journal of UbiComp volume 4 issue 3 on pages 23 to 39*, 2013.

[12] Supurna De and Klaus Moessner. Context gathering in ubiquitous environments: Enhanced service discovery. In *Conference: Proceedings 3rd Workshop on Context Awareness for Proactive Systems(CAPS 2007)*, 2007.

[13] https://docs.gigaspaces.com/xap/14.0/overview/about-jini.html.

[14] Niels Lohmann. Json for modern c++. https://github.com/nlohmann/json.

[15] Tristan Penman. Header-only c++ library for json schema validation. https://github.com/tristanpenman/valijson.

[16] Politecnico di Torino. Universal node. https://github.com/netgroup-polito/un-orchestrator, 2016.

[17] Open virtual switch. https://www.openvswitch.org/.