POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica orientamento Reti

Tesi di Laurea Magistrale

Comunicazione self-optimized tra dispositivi eterogeneamente connessi ad una rete con tolleranza ai ritardi

Inserimento di un dispositivo reale all'interno di reti DTN

Relatore
prof. Risso Fulvio Giovanni Ottavio

Laureando
Calogero Carrabbotta
matricola: s233057

Supervisore aziendale
Tierra Telematics Design
dott. Amparore Elvio, dott. Loti Riccardo

Anno accademico 2018/2019

C'è una grossa differenza tra un lavoro fatto e un lavoro fatto bene.

Sommario

Con la validazione e creazione della rete basata sul paradigma DTN, si è potuto configurare e inserire un dispositivo embedded di reale utilizzo, con limitazioni in termini di RAM e CPU, all'interno della rete, permettendo inoltre il raggiungimento delle sue applicazioni che non utilizzano il bundle protocol, tramite modifiche di appositi software che ora sfruttando il collegamento DTN possono trasmettere informazioni native TCP/IP attraverso il bundle protocol. La tesi è atta a spiegare il modello utilizzato per creare un ponte di comunicazione tra un comune browser e la rete DTN, a illustrare come configurare, compilare ed utilizzare gli applicativi che permettono tale risultato e ad analizzare i benefici di tale approccio.

Ringraziamenti

Per questa tesi sono molte le persone da ringraziare, in particolare il Professor Fulvio Risso che mi ha dato la possibilità di partecipare a questo progetto e ad eventi dove ho dovuto mettere in prima persona le mie capacità di esposizione, Gabriele Castellano che mi ha aiutato durante tutto il periodo di permanenza in laboratorio, ai tesisti precedenti che mi hanno agevolato il lavoro anticipandomi le soluzioni a problemi che loro avevano già incontrato. Un grande riconoscimento va ad Elvio Amparore che mi ha ampliamente supportato, insegnato tutto ciò che poteva essermi utile e corretto i miei tanti sbagli, per arrivare ad avere un risultato quanto più perfetto possibile. Un grazie va a anche a Riccardo Loti che mi ha permesso di lavorare all'interno di Tierra con dispositivi reali, dandomi così la possibilità di lavorare in un ambiente d'ufficio e permettendomi di mostrare i miei risultati anche all'estero.

Infinito riconoscimento alla mia famiglia che ha sempre creduto in me, ha sempre rispettato il mio impegno con l'università e sostenuto i miei lunghi studi su ogni lato, senza di loro non sarei mai arrivato a completare questo importante traguardo della mia vita.

Ultimi ma non meno importanti sono tutti gli amici che mi sono stati accanto durante questi lunghi anni di Politecnico: Andrea, Francesco, Sofia, Anna, Gaspare che da quando mi conosce sa che io devo studiare, Erika, Rebecca che mi ha tenuto compagnia durante tutte le notti di stesura della tesi, Fabrizio che mi ha sempre incoraggiato, Marco, Michel, Giuseppe, Bruno e tutti gli altri amici del Politecnico e della palestra di Via Quarello.

Indice

Capitolo 1

Introduzione

La sempre crescente diffusione di dispositivi *IoT*, in concomitanza con la rapida evoluzione delle tecnologie di comunicazione wireless, introducono una serie di nuove sfide dal punto di vista del *networking*. È sempre più comune imbattersi in contesti in cui le entità coinvolte nella comunicazione sono in continuo movimento e caratterizzate, nella maggior parte dei casi, da risorse hardware limitate. In tali scenari, identificabili con il nome di "*challenged networks*", non è possibile fare affidamento ai paradigmi di comunicazione tradizionali. Le comunicazioni in Internet, ad esempio, si fondano sull'assunzione secondo cui, in ogni istante, è garantita l'esistenza di almeno un percorso end-to-end tra la sorgente e la destinazione del traffico. Questo non è assolutamente garantito nel contesto di una *challenged network*, che è invece caratterizzata da continue interruzioni o addirittura assenza di connettività, perciò non è mai assicurato un percorso stabile tra sorgente e destinazione. Oltre che alla mobilità dei dispositivi, la connettività intermittente può essere dovuta a fattori intrinseci dell'ambiente in cui la *challenged network* è collocata, come la presenza di ostacoli che si interpongono tra i dispositivi atti a comunicare, oppure il verificarsi di fenomeni atmosferici ed ambientali avversi. La connettività intermittente porta con sé una serie di ulteriori problemi, come il partizionamento della rete, ritardi lunghi e/o variabili, alto tasso di perdità di informazioni, i quali rendono la comunicazione ancora più complessa. Gli scenari soggetti a tali problematiche sono molteplici e spaziano dall'ambito militare, all' interplanetario, alle reti di sensori nelle aree non dotate di alcuna infrastruttura di telecomunicazione, come ad esempio siti di costruzione in zone montuose o rurali.

Le comunicazioni militari sono tipicamente situate in zone dove la connessione non è in alcun modo garantita per ovvi motivi, ma al contempo è richiesta la massima precisione nella creazione dei dati e la minima latenza di propagazione delle informazioni possibile. Le comunicazioni interplanetarie sono caratterizzate da lunghi ritardi di propagazione e da significative interruzioni della visibilità tra le entità coinvolte, le quali si muovono continuamente lungo percorsi orbitali. Le reti di sensori o *Wireless Sensor Network (WSN)*, invece, coinvolgono dispositivi tipicamente mobili e dotati di limitate risorse computazionali e di memorizzazione, per cui, spesso, alcuni link di comunicazione sono intenzionalmente spenti al fine di risparmiare energia.

In tutti gli scenari sopra descritti non è da trascurare l'impatto di disturbi metereologici, i quali possono facilmente interrompere per qualche tempo la connettività, e il movimento di materiale industriale che può generare disturbi alla comunicazione.

In più, la maggior parte delle applicazioni esistenti suppongono di operare su reti connesse, caratterizzate da ritardi minimi o perlomeno trascurabili. Poiché la modifica di tali applicazioni richiederebbe uno sforzo materialmente insostenibile, date le assunzioni precedenti, l'approccio più semplice è quello di introdurre un framework comune a tutti i nodi della rete, una sorta di interfaccia, che permetta di far fronte alle esigenze tipiche delle *challenged network*.

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

1.1 Obiettivo della tesi

Il lavoro svolto in questa tesi si colloca all'interno di un più ampio progetto atto a prototipare un framework di comunicazione robusto basato sul paradigma Delay Tolerant Networking (DTN), che consente ai nodi coinvolti di operare su tecnologie di trasmissione eterogenea in maniera trasparente. Nello specifico, questa tesi valida l'impiego dell suddetto framework su un reale dispositivo impiegato in campo lavorativo. Utilizzando il dispositivo TOPCON NetG5, ci si è posti l'obiettivo di rendere disponibili, su una rete basata su protocollo bundle, i servizi da questo offerti.

Il compito perciò è quello di rendere eseguibili sulla piattaforma di esecuzione del dispositivo tutti gli applicativi precedentemente sviluppati e date le poche risorse disponibili

di questo prodotto, ci si è dovuti impegnare a rendere il più possibile leggeri, in termini di occupazione di memoria e di risorse computazionali, i componenti atti al rendere la NetG5 un nodo valido sulla rete *DTN*. In particolare è stato posto l'obiettivo di raggiungere tramite protocollo *bundle* in primis il servizio web di controllo del dispositivo, modificando il percorso delle richieste HTTP, per poi estendere la raggiungibilità a qualsiasi tipo di richiesta TCP/IP in cui la NetG5 può essere vista come brigde o come endpoint per la richiesta.

L'utente quindi potrà usare qualsiasi applicazione basata su TCP/IP, la quale tramite opportuna configurazione, potrà sfruttare la rete di tipo *DTN*, senza la richiesta di modifica del codice sorgente del programma scelto.

Capitolo 2

Delay Tolerant Network

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.

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

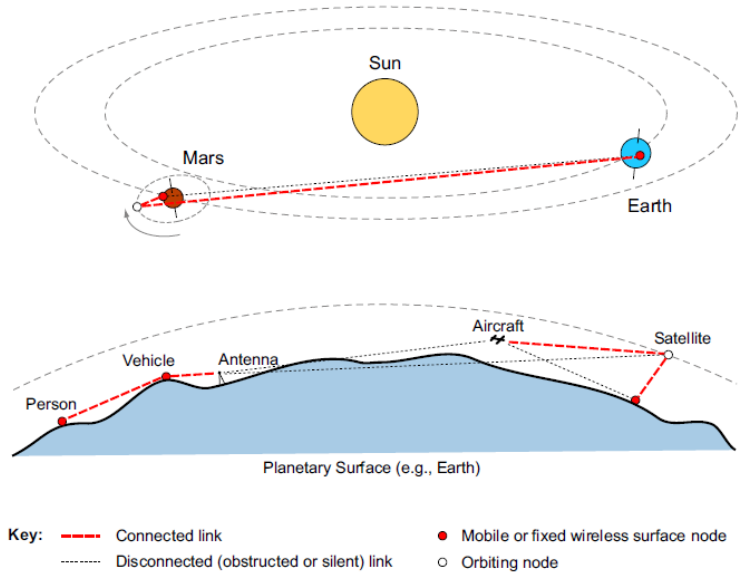


Figura 2.1: Esempi di contatti pianificati (comunicazioni interplanetarie) e opportunistici (comunicazioni sulla superficie terrestre)

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.

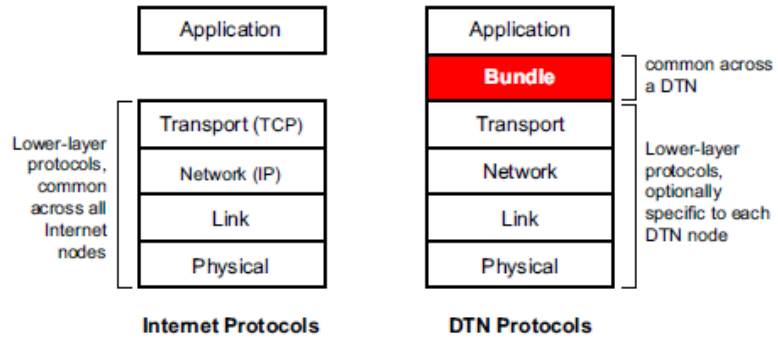


Figura 2.2: Confronto fra uno stack Internet (a sinistra) e uno stack DTN (a destra)

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.

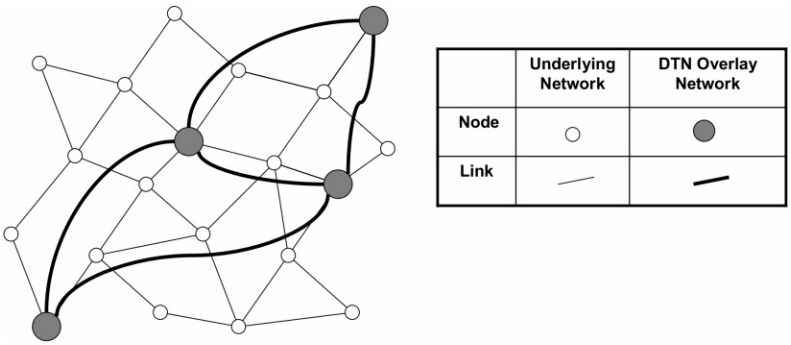


Figura 2.3: La rete DTN in overlay su un altro tipo di rete

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.[1]

2.2 Bundle Protocol

Il *Bundle Protocol*[2] 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.

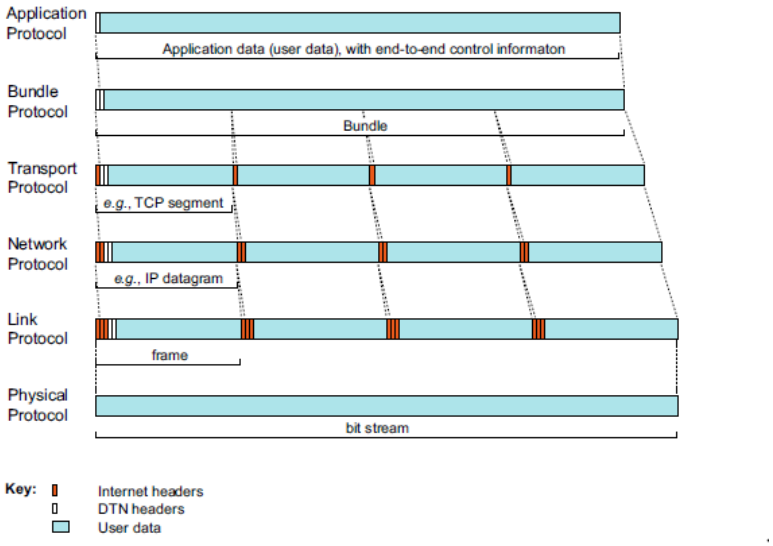


Figura 2.4: Incapsulamento dei protocolli TCP/IP nel Bundle Protocol

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.

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.

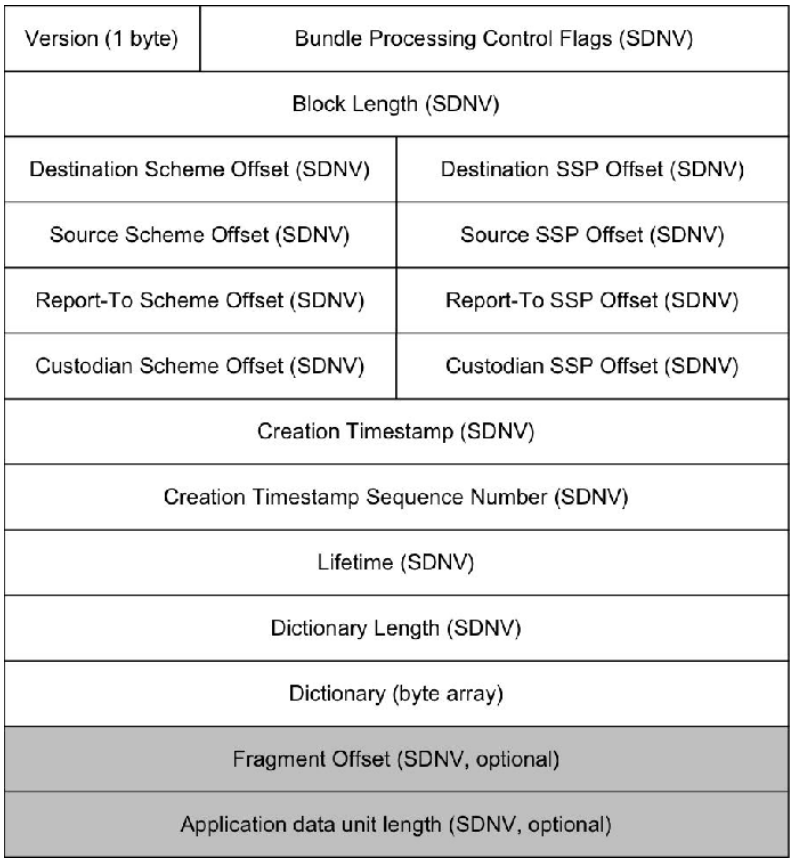


Figura 2.5: Formato del primary block di un bundle

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

- 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.

Altri blocchi

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

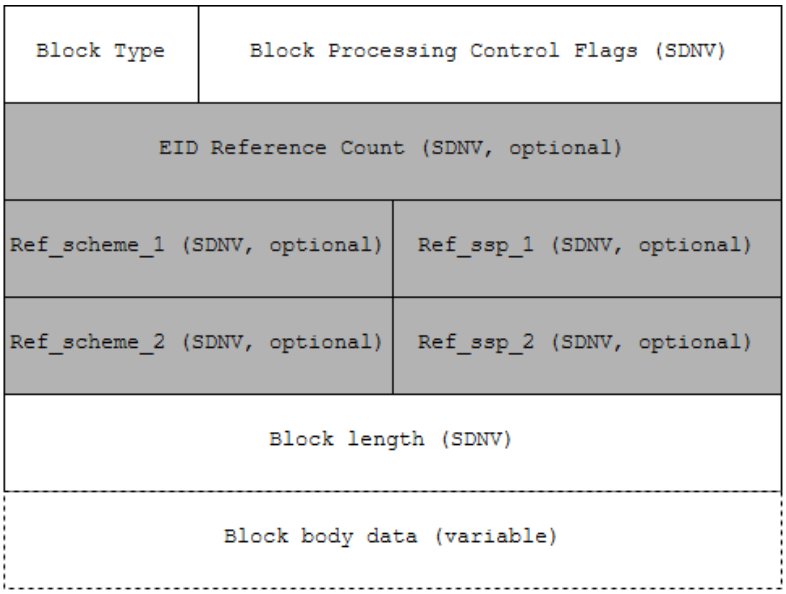


Figura 2.6: Formato generico di un blocco secondario di un bundle

Block Processing Control Flag I *Block Processing Control Flags* costituiscono una stringa di bit utili al processamento del blocco. E' un campo SDNV attualmente formato da 7 bit, indicanti alcuni particolari accorgimenti sul blocco. Per esempio abbiamo la possibilità di replicare il blocco in ogni frammento (in 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

2.3.1 Introduzione

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

2.3.2 L'architettura

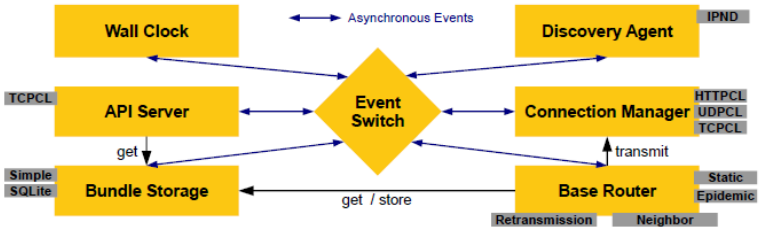


Figura 2.7: Architettura IBR-DTN

La versione di IBR-DTN per i sistemi operativi tradizionali è stata sviluppata in C++. Come si può notare in 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[4].

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

Discovery Agent Un altro 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)[5]. Tale modulo rimane in ascolto di piccoli datagrammi UDP detti *beacon*, utilizzati dai nodi per annunciare la propria presenza ai vicini, e periodicamente si annuncia tramite i medesimi datagrammi. I *beacon* sono spediti ad un indirizzo IP multicast noto (e specificabile in configurazione) e 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[2] 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[6], 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

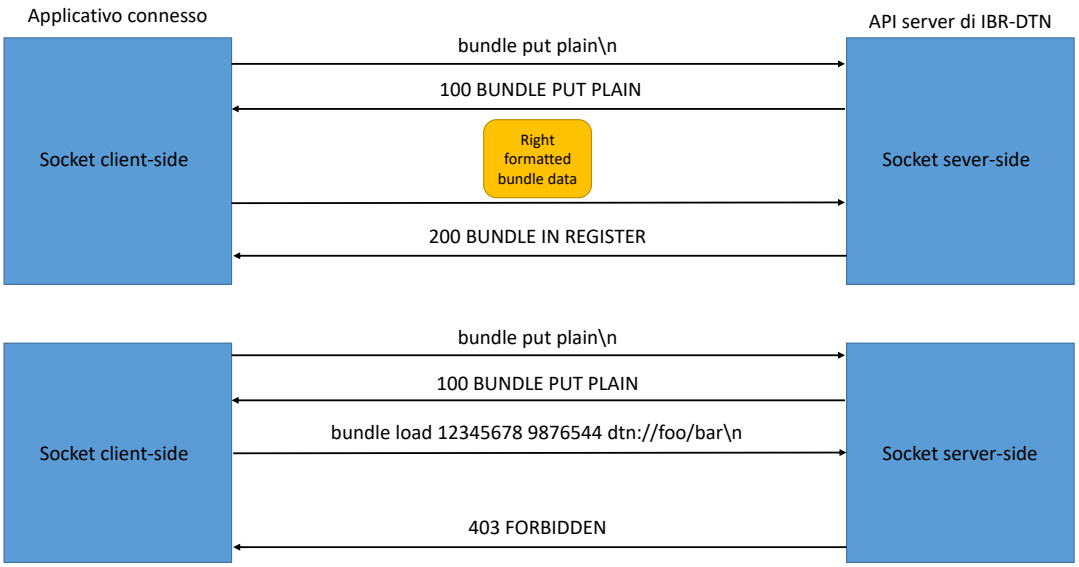


Figura 2.8: Interazione API Server

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 [7]. 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 per interagire 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:

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

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 Works

La rete DTN è un modello che può risultare utile in diverse occasioni di emergenza, dove la connessione non è garantita, la rete AnonNet [8] è l'esempio di un possibile utilizzo di tali vie di comunicazioni.

3.1 Rete di sensori, adHoc e resistente ai ritardi per la risposta ai disastri

AnonumousNet (AnonNet) è un sistema per rispondere alle emergenze in aree con disastri in larga scala, come ad esempio terremoti e tsunami. Nonostante l'attenzione posta sull'argomento delle risposte alle emergenze, non si riescono ancora ad avere dati per una totale risoluzione spaziale e temporale necessaria per salvare vite umane e per sostenere gli sforzi di disaster recovery. Le vittime di un disastro vengono salvate dopo giorni, se non settimane; le informazioni fisiologiche delle vittime non sono fornite in modo affidabile in tempo, manca un buon coordinamento tra i soccorritori, o è basato su metodi arcaici (scritte cartacee o segni sui muri). Il ritardo nel ricevere grandi quantità di informazioni è limitato dal tempo impiegato per trasportare fisicamente documenti o hard disk e nessun sistema di rilevamento/comunicazione costruito e implementato dura più di pochi giorni. AnonNet, progettato in collaborazione con i soccorritori di Urban Search & Rescue (US&R), è un primo passo per affrontare queste sfide. È progettato per aiutare a identificare le vittime in edifici crollati, fornire le informazioni fisiologiche delle vittime in tempo, fornire elevati volumi di dati sul campo ad alta velocità e in modo efficiente dal punto di vista energetico e integra nuovi paradigmi di social networking. AnonNet è un grande sforzo accademico, che propone sistemi aperti, piuttosto che soluzioni proprietarie. AnonNet e i suoi sottosistemi sono valutati in implementazioni e simulazioni reali.

3.1.1 Architettura del software

I dispositivi in AnonNet si dividono in quattro ampie categorie: sensori, smartphone, router DTN e maglie di router, come mostrato nella Figura 3.1

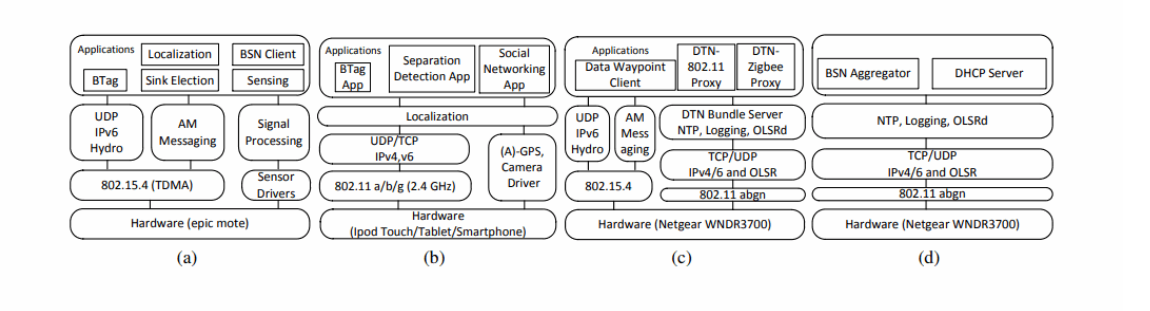


Figura 3.1: Struttura software AnonNet di (a) sensori, (b) device mobili, (c) DTN router e (d) mesh router

Le applicazioni vengono distribuite sui dispositivi degli utenti finali come app, e sui sensori di AnonNet (localizzazione e affondamento nella figura 3.1(a)). Tutte le applicazioni possono utilizzare il proprio formato dati o utilizzare uno schema pubblico in quanto la rete è indipendente dai dati. Nel caso in cui debbano usare le funzionalità DTN, si deve utilizzare un proxy DTN situato nel range di comunicazione del dispositivo. Un reverse-proxy (sviluppato anche questo per supportare la rete DTN) posto nella destinazione converte il formato DTN nel formato nativo dell'applicazione. In questo modo, le applicazioni di terzi possono essere facilmente integrate poiché i server proxy (inversi) hanno la possibilità di configurare le applicazioni su porte di sistema determinate.

Nel caso in cui l'applicazione di terze parti non supporti IP o il networking in generale è possibile, ad esempio, ritrasmettere i dati tramite un dispositivo IP collegato ad una delle interfaccie disponibili.

Networking e gestione dei dati

L'architettura di rete si estende su più protocolli a tutti i livelli dello stack di rete. Lo stack dominante utilizzato è 802.11abgn in modalità IBSS sotto IP/UDP (3.1(c) 3.1(d) e 3.1(b)). Poiché i sensori utilizzano 802.15.4, è necessario un edge-router o un gateway prima che i dati possano raggiungere le reti tradizionali.

Questa conversione avviene nella classe di dispositivi di backbone in cui i router 802.15.4 (3.1(c)) fungono da proxy tra i sensori e i dispositivi 802.11. Gli smartphone compatibili 802.11 possono comunicare tra loro in modalità IBSS e anche con dispositivi 802.15.4 tramite proxy. I dispositivi 15.4 non devono avere uno stack basato su IP, ma deve essere disponibile un'interfaccia compatibile sul router corrispondente. Le funzionalità DTN sono implementate come uno strato di rete sovrapposto nel livello applicazione solo sui router DTN (server bundle in 3.1(c)). Ogni dispositivo compatibile con le funzionalità DTN ha un server locale con cui le "app" DTN (client Waypoint dati nella 3.1(c)) possono comunicare attraverso un'API locale. In AnonNet, queste app possono fungere da proxy e presentare un'interfaccia DTN agli smartphone ("DTN IP Proxy" nella 3.1(c)) e 802.15.4 dispositivi ("DTN 802.15.4 Proxy" nella 3.1(c)). Quando tali dispositivi richiedono che i dati vengano inviati tramite DTN, il proxy accodata i dati nella coda del server locale. Quando verrà

trovato un vicino adatto, direttamente o tramite instradamento con tolleranza ai ritardi, i dati verranno inviati. La rete statica funziona come una normale WMN (wireless mesh network). I router che formano questa rete non sono compatibili con DTN. I servizi forniti da questi dispositivi includono DHCP ("Server DHCP" nella 3.1(d)) e funzionalità di aggregatore BSN che aggregano i dati dai client BSN in movimento.

3.2 Reti IP autoconfiguranti

La creazione di una rete DTN non è l'unica via per permettere la comuncazione di dispositivi senza una configurazione manuale. In tal senso, con l'unica limitazione dell'utilizzo del solo protocollo IP, Zeroconf è un possibilità che permette l'auto-configurazione dei dispositivi nella rete.

3.2.1 Zeroconf

Zeroconf [9] o Zero Configuration Networking è un protocollo standard dell'IETF per la configurazione dinamica dei nodi di una rete utilizzante il protocollo IP. Lo standard non è ancora definitivo, infatti non esiste ancora nessun documento RFC sul protocollo Zeroconf sebbene ci siano già diverse sue implementazioni che vengono utilizzate quotidianamente da moltissimi utenti essendo incluse nei sistemi operativi.

L'idea base di Zeroconf è che, collegando due computer tramite cavo Ethernet, questi dobrebbero essere in grado di comunicare tra loro senza il bisogno di interventi da parte dell'utente. Attualmente, invece, bisogna impostare dei parametri per consentire a due computer di comunicare.

L'obiettivo è quello di ottenere una rete IP funzionante senza dover dipendere da infrastrutture (server DHCP, server DNS o simili) o da conoscenze specifiche (per esempio indirizzamenti RFC 1918). Zeroconf è stato inizialmente sviluppato da Apple Inc. come un componente del sistema operativo per facilitare il passaggio dalle reti AppleTalk alle reti IP. Zeroconf è stato utilizzato da molti produttori che cercavano uno strumento in grado di semplificare l'integrazione dei loro prodotti nelle reti locali. Un buon esempio sono le stampanti di rete. I produttori delle stampanti che vengono collegate direttamente alla rete non possono fornire i loro prodotti di schermi o tastiere da utilizzare per inserire i parametri di rete come indirizzo IP o subnet masks. Si cercava anche uno strumento utilizzabile dall'utente comune e la tecnologia Zeroconf era la soluzione ideale. Attualmente tutte le stampanti di rete in commercio implementano Zeroconf.

Il sistema Zeroconf si basa sull'assegnazione automatica degli indirizzi IP da parte dei dispositivi che lo utilizzano, senza utilizzare un server DHCP. Gli indirizzi IP utilizzati da Zeroconf appartengono alla sottorete 169.254.0.0/16 (IPv4 link local). Poiché i primi e gli ultimi 256 indirizzi sono riservati ad usi futuri, quelli utilizzabili sono compresi tra

169.254.1.0 e 169.254.254.255 (RFC 3927 sezione 2.1). Solitamente i sistemi Zeroconf si occupano di verificare periodicamente l'esistenza di un server DHCP e in questo caso di utilizzare gli indirizzi forniti dal server stesso.

Capitolo 4

HTTP over Bundle Protocol

La Net-G5, riportata in dettaglio nel capitolo successivo, dispone di un server web in ascolto sulla porta 80, l'obiettivo finale è visualizzare la pagina web di configurazione incapsulando lo scambio dei pacchetti in bundle. Ogni endpoint disporrà di un HTTP/Bundle protocol proxy accompagnato da IBR-DTN che permetterà all'HTTP di operare End-To-End, attraverso la DTN, da un IP di una rete all'altra. In questo caso il bundle protocol sarà totalmente trasparente sia al browser che al web server, i quali continueranno a svolgere il proprio normale compito con i propri protocolli senza prestare attenzione ai dettagli della DTN.

4.1 HTTP 1.0

HTTP [10] è un protocollo che lavora con un'architettura di tipo client/server: il client esegue una richiesta e il server restituisce la risposta. Nell'uso comune il client corrisponde al browser ed il server alla macchina su cui risiede il sito web. Vi sono quindi due tipi di messaggi HTTP: messaggi richiesta e messaggi risposta. HTTP differisce da altri protocolli di livello 7 come FTP per il fatto che le connessioni vengono generalmente chiuse una volta che una particolare richiesta (o una serie di richieste correlate) è stata soddisfatta. Questo comportamento rende il protocollo HTTP ideale per il World Wide Web, in cui le pagine molto spesso contengono dei collegamenti (link) a pagine ospitate da altri server diminuendo così il numero di connessioni attive limitandole a quelle effettivamente necessarie, con aumento quindi di efficienza (minor carico e occupazione) sia sul client che sul server. Data la sua natura testuale e la possibilità di impostare la chiusura forzata della connessione una volta ottenuta la risorsa è un protocollo che si adegua perfettamente ad essere incapsulato in un bundle.

Il porting su DTN di questo protocollo amplia di molto le possibilità della rete bundle, difatti è sempre più usuale avere su dispositivi embedded una pagina web di configurazione o la presenza di una REST API che consente l'utilizzo remoto, dunque avere la possibilità di utilizzare un normale browser consente una totale trasparenza di utilizzo all'utenza finale

e la totale compatibilità delle recenti tecnologie [11].

4.2 Proxy

In informatica e telecomunicazioni, un server proxy è un server (inteso applicazione in questo caso) che funge da intermediario per le richieste da parte dei client alla ricerca di risorse su altri server, disaccoppiando l'accesso al web dal browser. Un client si connette al server proxy, richiedendo qualche servizio (ad esempio un file, una pagina web o qualsiasi altra risorsa disponibile su un altro server), e quest'ultimo valuta ed esegue la richiesta in modo da semplificare e gestire la sua complessità.

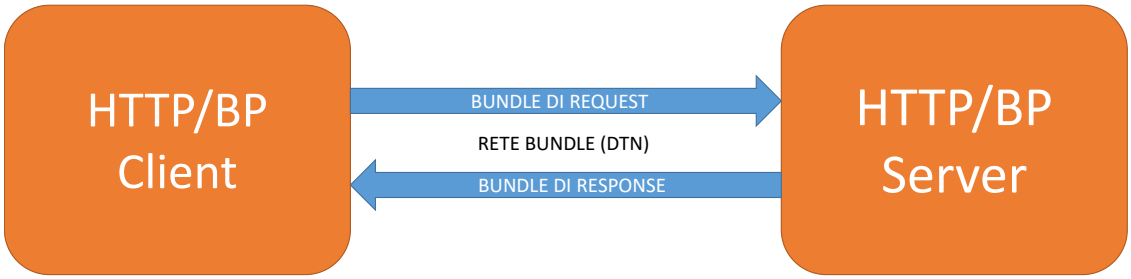


Figura 4.1: Scenario di incapsulamento diretto

La creazione di un processo proxy permette di evitare lo scenario diretto riportato in figura 4.1, che comporta la creazione di specifici applicativi di visualizzazione di risorse web che da una normale richesta HTTP creino direttamente il bundle in uscita verso l'altro endpoint.

Questa soluzione non è efficace in quanto non si potrebbe ottenere la stessa user-experience e facilità d'uso, ormai più che affermata, di un qualunque browser contemporaneo.

Grazie al proxy, dunque, l'utente potrà aprire il proprio browser preferito e tramite l'utilizzo di un URL specifico per le risorse DTN demandare al proxy l'incapsulamento di una qualunque sua richiesta HTTP in un bundle per la rete DTN.

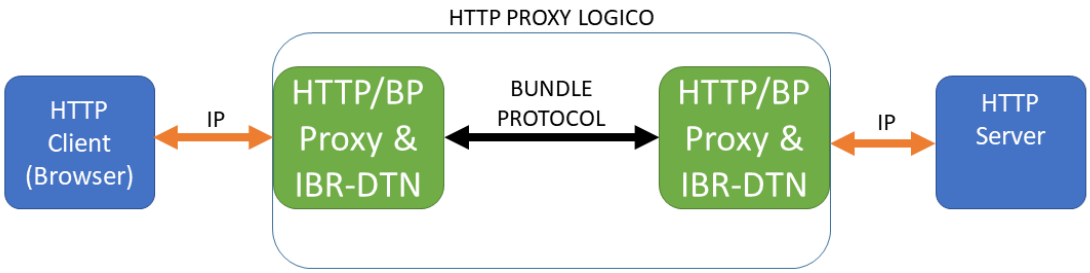


Figura 4.2: Scenario di incapsulamento attraverso proxy

Seguendo il modello riportato in figura 4.2 avverrà la seguente interazione, mostrata nelle figure 4.3 e 4.4:

1. L'utente aprirà un browser, opportunamente configurato per rimandare sul proxy ogni sua connessione.
2. L'utente digiterà l'URL speciale formattato appositamente per l'utilizzo della rete DTN
3. Il browser in automatico ridirigerà la richiesta HTTP verso il proxy.
4. Il proxy, riconoscendo l'URL speciale per la rete DTN, comincierà l'interazione con IBR-DTN per la creazione di un bundle.
5. Una volta incapusulati tutti i dati verrà inviato il comando di inoltro del nuovo bundle.
6. Il bundle transiterà attraverso la rete Bundle e arriverà all'endpoint di destinazione.
7. IBR-DTN segnalerà al proxy un nuovo bundle per l'applicazione IP nativa.
8. Il proxy si occuperà di parsificare la richiesta e la inoltrerà al proprio server web.
9. Il server riceve la richiesta HTTP e creerà una response con tutti i dati necessari alla visualizzazione della pagina.
10. Ora la risposta subirà un trattamento del tutto simmetrico alla richiesta per arrivare al browser dell'utente.

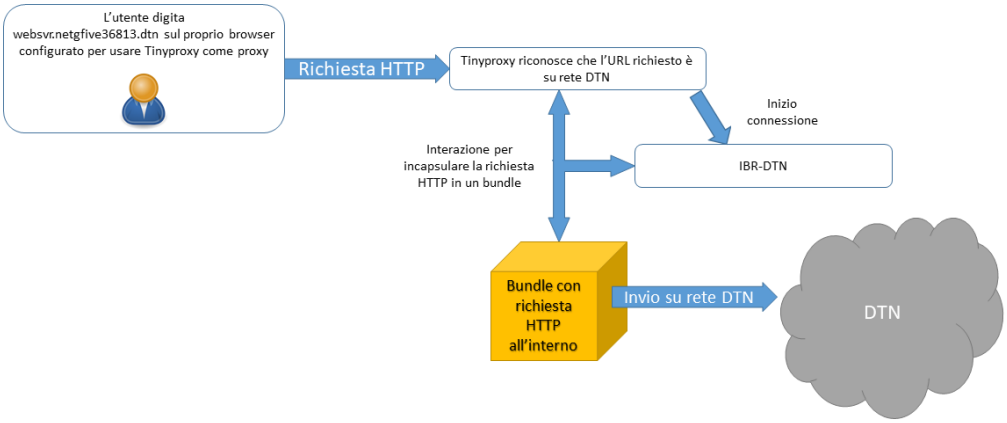


Figura 4.3: Passi di comunicazione lato utente

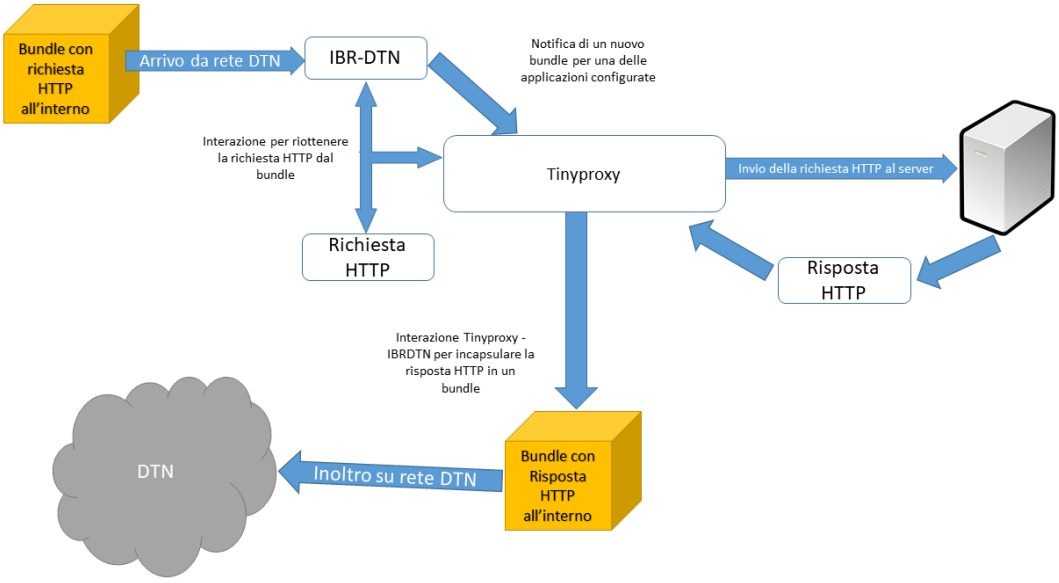


Figura 4.4: Passi di comunicazione lato web server

4.3 Tinyproxy come HTTP/Bundle protocol proxy

Tinyproxy nasce come proxy HTTP/HTTPS, quindi è stato necessario apportare delle aggiunte al codice per permettere l'utilizzo del protocolo a bundle. Anzitutto è necessario spiegare l'estensione su Bundle Protocol di Tinyproxy segue due flussi di esecuzione diversi in base al ruolo del nodo su cui è installato, verrà denomitato come proxy `client-side` nel momento in cui Tinyproxy si trova su una macchina dove è in ascolto per le richieste di un browser e proxy `server-side` nel momento in cui si trova su un device dove è presente un web server. In questo capitolo verranno mostrati i passaggi ad alto livello di come viene gestita la connessione, nel capitolo successivo saranno mostrati i dettagli del codice.

4.3.1 Formattazione URL specifica per risorse su DTN

Tinyproxy continua a conservare la sua possibilità di far da proxy per traffico HTTP/-HTTPS, infatti tutto il comportamento nativo è stato mantenuto e in qualche caso corretto. Il nuovo flow viene attivato nel momento in cui l'URL digitato segue la seguente convenzione

[dotted_app_name].nodename.dtn/<resources>

come ad esempio *websvr.netgfive36813.dtn/*

Grazie a questa formattazione si possono selezionare ad una ad una le applicazioni presenti sull'endpoint desiderato, con l'implicazione che viene persa l'informazione della porta TCP/IP la quale non è presente nel protocollo bundle. Sarà poi grazie alla corretta configurazione di Tinyproxy prensente sul device con i diversi web server che, una volta ripassati ad IP, con l'opportuno rewriting, verrà reinserita l'informazione sulla porta da utilizzare.

4.3.2 Client-Side

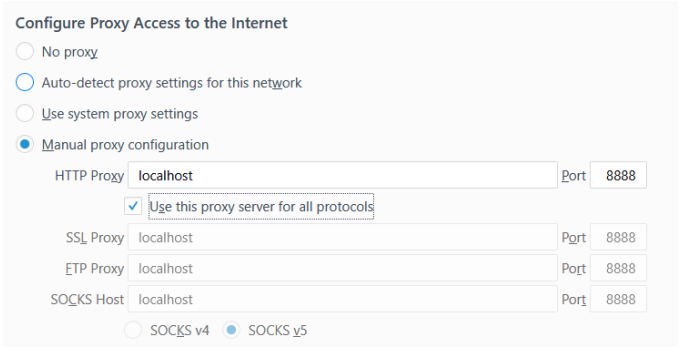


Figura 4.5: Firefox Correttamente configurato

Tinyproxy in modalita `client-side` richiede una configurazione minimale della macchina su cui è avviato. Anzitutto è necessario impostare l'utilizzo del proxy nel browser,

come riportato in figura 4.5. Successivamente, nel file di configurazione di Tinyproxy, bisogna disattivare la possibilià di lavorare `server-side` e specificare l'indirizzo IP e la porta di IBR-DTN.
Una volta fatto si potrà procedere con l'operazione 2, ovvero digitare l'indirizzo `websvr.netgfive36813.dtn` (da ora in poi specificato come URL) così da inviare una richiesta HTTP.

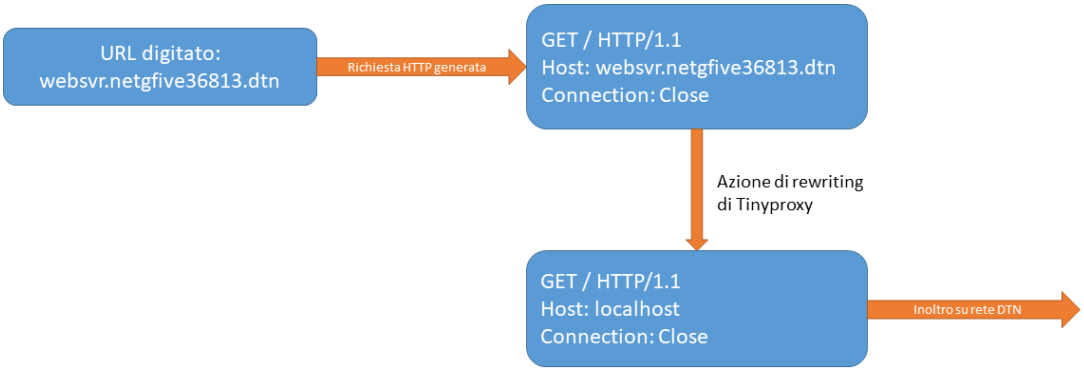


Figura 4.6: Riscrittura dell'header

A questo punto il browser invierà la HTTP-request direttamente a nuovo ed esteso Tinyproxy che provvederà a:

- Leggere il tipo di richiesta (GET, POST, ...).
- Parsificare tutti gli header della richiesta (Host, Connection, ...).
- Riconoscere se la richiesta è da inviare su rete IP o su rete Bundle.
- Dopo lo switch a bundle protocol, riscrivere il campo `Host` con il valore `localhost` così da permettere il corretto funzionamento del web-server, mostrato in figura 4.6.
- Assegnare un nome univoco temporaneo sulla rete DTN così da permettere il corretto scorrere delle infomazioni.
- Ricreare l'EID valido per la il nodo destinazione attraverso l'URL scritto dall'utente.
- Ottenuti sorgente e destinazione in formato DTN creare il bundle al quale appendere tutte gli header precedentemente parsificati.
- Finiti gli header inviare il bundle e si mettersi in attesa di un bundle di risposta.
- Nel momento in cui IBR-DTN segnalerà un nuovo bundle per la connessione in corso, caricare in memoria il bundle.

- Il nuovo bundle viene quindi aperto e ne viene controllata l'integrità, ora bisogna parsicare le righe del bundle protocol fino ad arrivare al primo header HTTP da rimandare al browser
- Si riprende il normale comportamento reinoltrando le righe della request al browser.
- In fine segnalare che il bundle di risposta è stato consegnato, permettendo l'eliminazione corretta del bundle da parte di IBR-DTN.

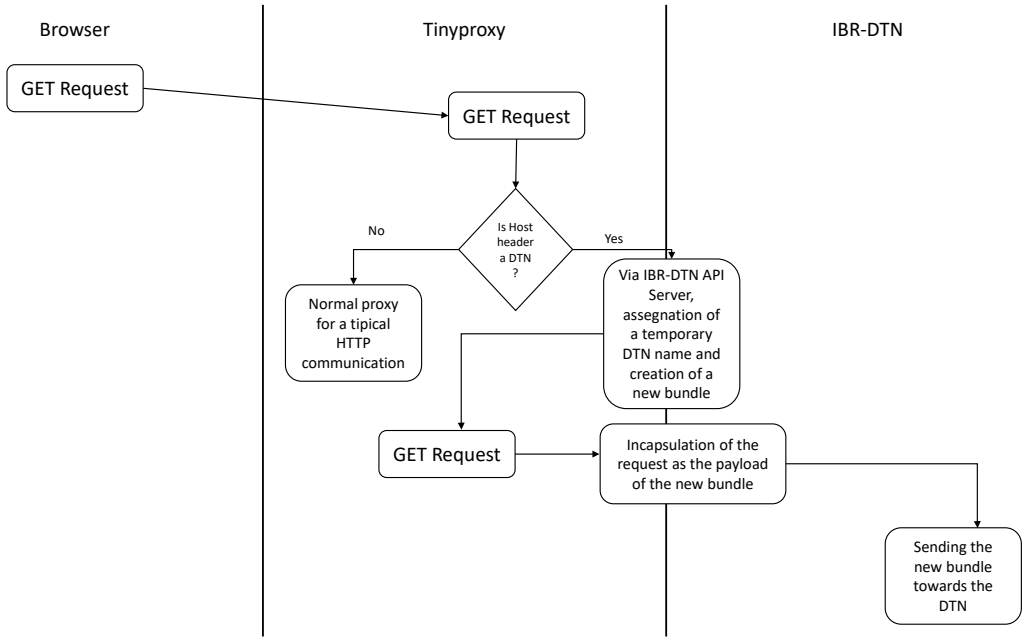


Figura 4.7: Diagramma di creazione di un bundle che incapsula la HTTP request

4.3.3 Server-Side

Tinyproxy in modalità `server-side` richiede una maggiore configurazione, che però permette l'inizializzazione di più server-web sullo stesso dispositivo. Anzitutto questo comportamento è disattivabile in quanto le funzionalità `server-side` potrebbero interferire con quelle `client-side` se attivate in contemporanea sullo stesso device; per evitare questo tipo di errore si è implementata questa semplice protezione. Come per la parte `client-side` bisogna specificare a quale indirizzo IP e su quale porta TCP/IP risiede l'API server di IBR-DTN. L'ulteriore configurazione consiste nell'elencare e decidere dei nomi per tutti i web server che si vogliono raggiungere, come ad esempio:

```
# If you have more than 1 application, you can set over IBR-DTN as
    much endpoint as you want.
# You only have to specify an unique app name, an Ip address and a
    port.
# Default one is web 80 127.0.0.1
```

```
#
dtnAppName "websvr" 80 127.0.0.1
dtnAppName "websvr2" 80 192.168.1.54
```

Con questa configurazione abbiamo deciso che esiste un server-web locale al nodo sulla porta 80 e uno remoto sull'indirizzo IP segnalato in ascolto sempre sulla porta 80. La possibilità di configurare anche applicazioni web esterne al nodo è data grazie al *rewriting* dell' header *Host*, il chè rende ancora più utile il collegamento DTN che ora si propone anche come bridge tra reti completamente diverse. Quindi, una volta avviato Tinyproxy in modalità `server-side` verranno immediamente dichiarate, tramite apposito comando, a IBR-DTN le applicazioni per il quale esso deve attendere dei bundle. Una volta arrivato un bundle perciò avverà la seguente interazione:

- IBR-DTN invierà una notifica alla connessione registrata per l'EID destinatario.
- La connessione provverderà a inizializzare una nuova connessione temporanea verso IBR-DTN che caricherà il bundle avendo a disposizione i dati per la richiesta. Ciò è necessario in quanto in questo modo più richieste contemporanee possono essere eseguite in parallelo da più processi su socket diversi che caricano. bundle per una specifica applicazione DTN.
- Il nuovo bundle verrà quindi aperto, ne verrà controllata l'integrità e si parsificheranno le righe del bundle fino ad arrivare al primo header da inviare al web-server.
- Inizio parsing della request.
- La request in arrivo non avrà più il campo Host in formato DTN in quanto come descritto prima è stato sostituito, perciò Tinyproxy tenterà la connessione verso il web-server utilizzando però l'indirizzo IP e la porta configurati in precedenza.
- Stabilita la connessione verrà inviata la request al server che procederà processarla.
- Essendo la request recapitata, per concludere l'elaborazione del bundle è necessario segnare come consegnato il bundle processato.
- Nel mentre che si attende l'elaborazione del web-server, ci si occupa di aprire un nuovo bundle a cui appendere la risposta.
- Una volta che il server comincia a scrivere sul socket la risposta, Tinyproxy appende tutta la risposta in un unico bundle.
- Completato l'arrivo dei dati dal web-server Tinyproxy invierà il comando di invio del bundle di risposta.

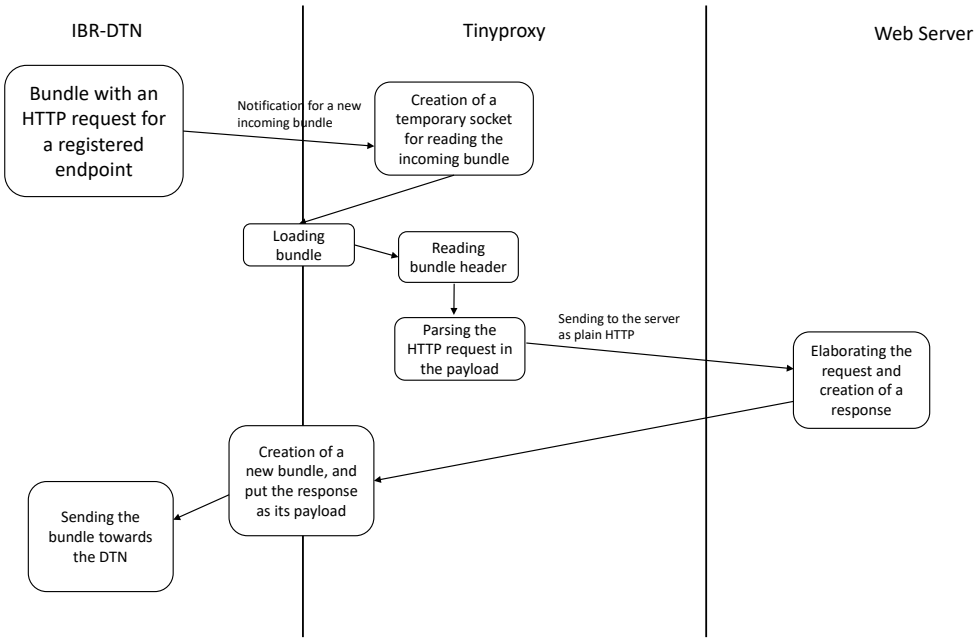


Figura 4.8: Diagramma di lettura di un nuovo bundle e incapsulamento della risposta in un nuovo bundle

4.4 Estensione al supporto di HTTP/1.1

Per supportare la completa navigazione verso qualsiasi web-server sia all'interno di una rete DTN sia esterno, è stato necessario estendere le capacità di Tinyproxy all'utilizzo del protocollo HTTP/1.1. HTTP/1.1 è la versione più in uso nella rete internet ad oggi ed introduce delle ottimizzazioni sia in termini di performace che in termini di features. Permette, anzitutto l'uso di connessioni persistenti e pipelined, trasferimenti a chunck, compressione e decompressione, supporto al caching e altre caratteristiche utili alla navigazione.

L'estensione alla versione 1.1 è stata necessaria in principio per l'utilizzo di connessioni persistenti. In questo modo si attiva il supporto a messaggi HTTP di lunghezza arbitraria, come ad esempio quelli di uno streaming; in più si permette la comunicazione attraverso il metodo `CONNECT` che implicitamente supporta la comunicazione via HTTPS.

4.4.1 Connessioni Keep-Alive

Attraverso l'uso dell'header `Connection:Keep-Alive` è possibile mantenere un socket aperto sul quale possono passare più richieste HTTP, rendendo il caricamento dei vari componenti molto più veloce e sprecando meno banda non dovendo ripetere il TCP-Handshake ogni volta (4.9). Il comportamento di Tinyproxy è stato dunque modificato per non riscrivere a priori l'header `Connection:Close` ma lasciar decidere al server, il quale è più

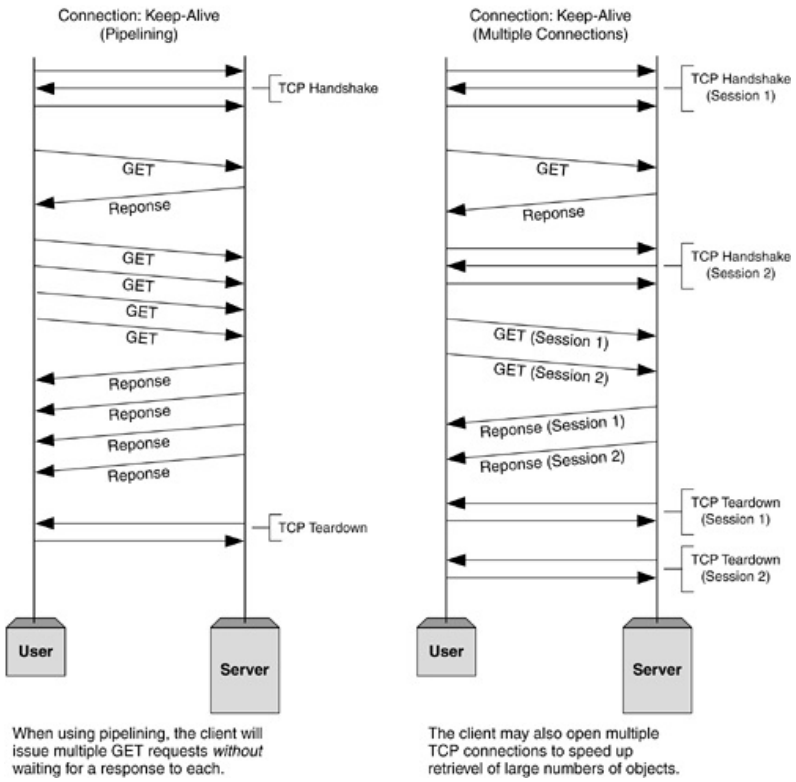


Figura 4.9: Connessione HTTP 1.1 Pipelined

autoritativo sulla connessione, il tipo di connessione da mantenere, dunque se il server nella risposta richiede la persistenza della connessione, ora Tinyproxy manterrà vivi i socket sia lato browser che lato server. L'estensione a questo comportamento poteva non essere così essenziale, in quanto se il browser richiede a prescindere la chiusura della comunicazione il server adatta le sue risposte a tale richiesta; tuttavia la connessione `Keep-Alive` risulta essenziale nel momento in cui si vuole supportare il metodo `CONNECT` per l'utilizzo di un WebSocket.

4.4.2 WebSocket

WebSocket è una tecnologia web che fornisce canali di comunicazione full-duplex attraverso una singola connessione TCP. L'API del WebSocket è stata standardizzata dal W3C e il protocollo WebSocket è stato standardizzato dall'IETF come RFC 6455 [12].

WebSocket è disegnato per essere implementato sia lato browser che lato server, ma può essere utilizzato anche da qualsiasi applicazione client-server. Il protocollo è un'implementazione basata sul TCP. La sua unica correlazione con l'HTTP è nel modo in cui fa l'handshake durante una Upgrade request tra server. Il protocollo WebSocket permette maggiore interazione tra un browser e un server, facilitando la realizzazione di applicazioni che forniscono contenuti e giochi in tempo reale. Questo è reso possibile fornendo un modo

standard per il server di mandare contenuti al browser senza dover essere sollecitato dal client e permettendo ai messaggi di andare e venire tenendo la connessione aperta, come mostrato in figura 4.10.

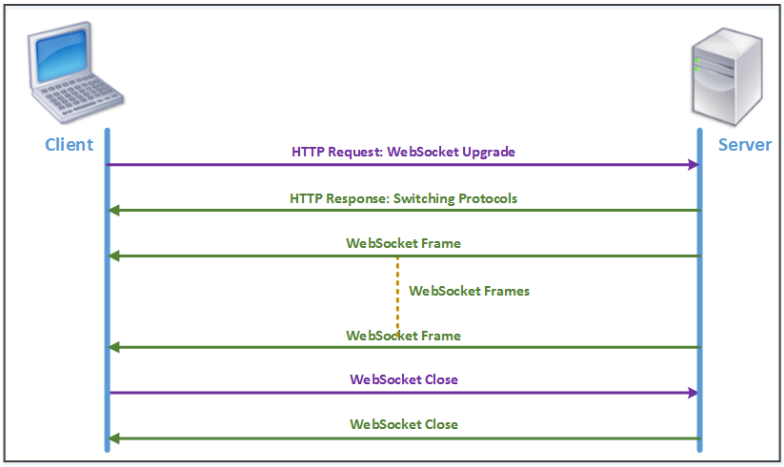


Figura 4.10: Comunicazione tramite WebSocket

Il WebSocket viene richiamato nel momento in cui il browser fa una richiesta di `CONNECT` a Tinyproxy; quest'ultimo si occupa dunque di rispondere all'handshake inziale per poi inviare la richiesta di risorse binarie verso il server, il quale richiederà la necessità di un upgrade al protocollo di `web-socket`. In questo momento il socket è da mantenere vivo per tutta la durata dello streaming di dati binari, fino a quando uno dei due endpoint non rilascia la connessione o il socket va in timeout.

Capitolo 5

Componenti per l'inserimento del dispositivo all'interno della rete DTN

5.1 TOPCON NetG5

Figura 5.1: La TOPCON NetG5

La Topcon Net-G5 è una GNSS [13] (Global Navigation Satellite System), cioè un sistema di geo-radiolocalizzazione e navigazione terrestre, marittima e aerea, che utilizza una rete di satelliti artificiali in orbita e pseudoliti.
I sistemi di geolocalizzazione forniscono un servizio di posizionamento geo-spaziale a copertura globale che consente a piccoli ed appositi ricevitori elettronici di determinare le loro coordinate geografiche (longitudine, latitudine ed altitudine) su un qualunque punto della superficie terrestre o dell'atmosfera con un errore di pochi metri, elaborando segnali

a radiofrequenza trasmessi in linea di vista da tali satelliti.
La Net-G5 è multi frequenza, ha la possibilità di comunicare attraverso ethernet, bluetooth 2.3, wi-fi e segnale 3G. Il ricevitore è dotato di slot per SD fino a 32GB, permettendo così la possibilità di customizzazione del dispostivo attraverso lo sviluppo di applicativi, basata su un'architettura di tipo SPARC con una CPU da 250MHz e offre una RAM di soli 256MB per l'esecuzione di programmi. Date queste limitazioni è stato opportuno scegliere applicativi con un'impronta in RAM molto leggera e, tramite la disattivazione di moduli non utilizzati, snellire i programmi da usare, come IBR-DTN.
Per ulteriori dettagli è possibile trovare online la documentazione completa e specifica del dispositivo [14] [15].

Grazie ai layer di comunicazione WI-FI o ethernet e al servizio Telnet in ascolto sulla Net-G5 è possibile accedere direttamente al sistema operativo di Linux che però è in modalità read-only, perciò non è possibile installare direttamente sulla macchina qualsiasi tipo di software, evitando di fatto un brick dell'oggetto. Quindi è necessario montare correttamente, tramite appositi comandi, la SD su cui vengono copiati gli applicativi cross-compilati per l'archettura SPARC.
Il dettaglio della configurazione è presente nell'appendice A.

5.2 Docker con immagine OpenEmbedded

Per permettere l'avvio di applicativi sulla Net-G5 è necessaria la cross-compilazione per architettura SPARC attraverso l'immagine OpenEmbedded [16] avviata tramite DOCKER [17].

5.2.1 Docker

Docker è una piattaforma software che permette di creare build, testare e distribuire applicazioni con la massima rapidità. Docker raccoglie il software in unità standardizzate dette container (Figura:5.2) che offrono librerie, strumenti di sistema, codice e runtime. Un container offre la stessa capacità di isolamento delle risorse e gli stessi benefici di una VM, ma virtualizza solamente il sistema operativo e non tutto l'hardware sottostante, permettendo così una maggiore efficienza e portabilità.

Più container possono essere eseguiti sulla stessa macchina, le quali condivideranno le risorse Kernel del SO ospitante tra loro, rimanendo comunque processi isolati in user space.

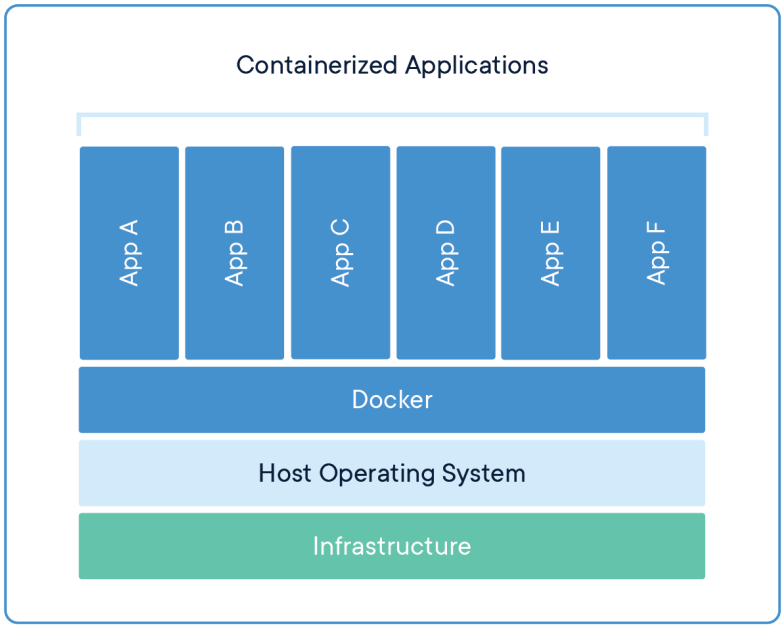


Figura 5.2: Struttura a blocchi dei containers

5.2.2 OpenEmbedded Distribuzione Ångström

Ångström è una distribuzione Linux creata per supportare una grande varietà di dispositivi embedded. La release è il risultato del lavoro degli sviluppatori dei progetti di OpenZaurus, OpenEmbedded e OpenSIMpad.

Pensata per essere altamente portabile e riconfigurabile è fatta per essere eseguita su tutti i device che supportano il kernel OpenEmbedded 2.6.x, con speciale attenzione ai dispositivi embedded quali: PDA, telefoni, router, ecc..
Ångström offre le più moderne caratteristiche di una distribuzione "Desktop" come: completo management dei pacchetti, bug tracking, possibilità di build flessibile e una communty attiva.
La distro è in coopetizione, ovvero sia in competizione che in cooperazione, con Poky Linux. Ångström è basata sul OpenEmbedded project, specificatamente sul layer OpenEmbedded-Core (OE-Core). Mentre sia Ångström che Poky Linux sono basati su OE-Core, utilizzino per lo più la stessa toolchain di strumenti e sono "Yocto compatibili", solo Poky Linux è ufficialmente parte del Yocto Project.

Ångström si differenzia da Poky Linux per essere una distribuzione binaria (come ad esempio Debian, Fedora, OpenSuse o Ubuntu) e per l'utilizzo di opkg come package manager.

Perciò una parte essenziale di Ångström builds è la possibilità di uso di pacchetti binari, che consente di installare semplicemente software distribuiti come pacchetti opkg, senza doverli compilare prima (proprio come si potrebbe installare un pacchetto binario con aptitude o dpkg su debian).

L'immagine Ångström fornita da TOPCON racchiude in se tutti gli strumenti per permettere la cross compilazione del codice sorgente e di tutte le librerie necessarie al funzionamento degli applicativi scelti.

5.3 Tinyproxy

Tinyproxy è un proxy HTTP/HTTPS per sistemi operativi POSIX. Costruito per essere sia veloce che piccolo è l'ideale per soluzioni di tipo embedded con risorse limitate.
È un applicativo molto leggero che richiede poche risorse, con una occupazione di memoria di circa 2MB; l'uso della CPU è lineare nel numero di richieste simultanee fatte al proxy. Richiede un ambiente POSIX minimale, riducendo così l'uso di librerie esterne e facilitando la compilazione per architettura SPARC. Scritto in C e sotto licenza GNU è stato modificato permettere al proxy di passare su rete bundle partendo da richieste HTTP/HTTPS e viceversa.

Per ulteriori dettagli è possibile consultare il sito di Tinyproxy [18].

Capitolo 6

Implementazione di Tinyproxy

In questo capitolo viene trattata nel dettaglio l'implementazione di estensione di Tinyproxy.
È necessario specificare che Tinyproxy è stato esteso sia per supportare il bundle protocol che per supportare l'HTTP/1.1 come specificato nei capitoli precedenti. L'implementazione è unica ma il flow di esecuzione varia a seconda di dove viene logicamente installato Tinyproxy, se per inoltrare i pacchetti di un browser (Client-Side) o quelli di un web-server (Server-Side).

6.1 Stato iniziale di Tinyproxy

Tinyproxy nasce come un semplice proxy HTTP/1.0 scritto in C che instanzia un numero configurato di processi per soddifisfare le richieste di un browser, utilizzando solo il protocollo HTTP/1.0 non permette l'uso di connessioni keep-alive e perciò ha un flow molto lineare dove cattura con l'uso della system call `select` la richiesta del browser che sarà parsificata e inoltrata verso il web-server.
Grazie a questo flow molto semplice da seguire le modifiche sono state effettuate, per la maggior parte, nella funzione di gestione dei dati scambiati tra browser e web-server. Tinyproxy si affida all'uso di processi separati per la gestione delle richieste, ma purtroppo la gestione del ciclo di vita di questi non è efficientissima, ad ognuno di essi viene associato uno stato e fino a quando questi non ritornano allo stato di attesa, il processo principale tenterà la creazione di nuovi sotto-processi, questo viene immediatamente corrotto se uno dei suoi processi figli va in crash.

6.2 Struttura del codice per l'estensione

Le modifiche a Tinyproxy sono orientate a far comunicare con l'`Api-Server` di IBR-DTN, il quale si aspetta dei comandi in una sequenza predefinita, l'utilizzo di comandi che non rispettano la corretta linea di comunicazione comporta una risposta negativa da parte di

IBR-DTN. Un messaggio d'errore da parte di IBR-DTN, ovviamente, non è assolutamente desiderato, in quanto crea un risultato non atteso che non permette la corretta inizializzazione delle strutture dati a supporto della comunicazione, rendendo il codice instabile e con alte probabilità di crash dovuti a puntatori non correttamente utilizzati o dando vita a memory leak dannosi.
Inoltre data la considerazione sul metodo di creazione dei processi, è molto importante che il ogni possibile errore venga gestito correttamente.
Perciò si è strutturato il codice seguendo il modello di una `macchina a stati`. L'uso di una macchina a stati permette la definizione di momenti in cui è permesso utilizzare determinate funzioni e quindi la creazione di un meccanismo che forza l'uso dei corretti comandi verso IBR-DTN e l'attesa del corretto messaggio da parte di questo.

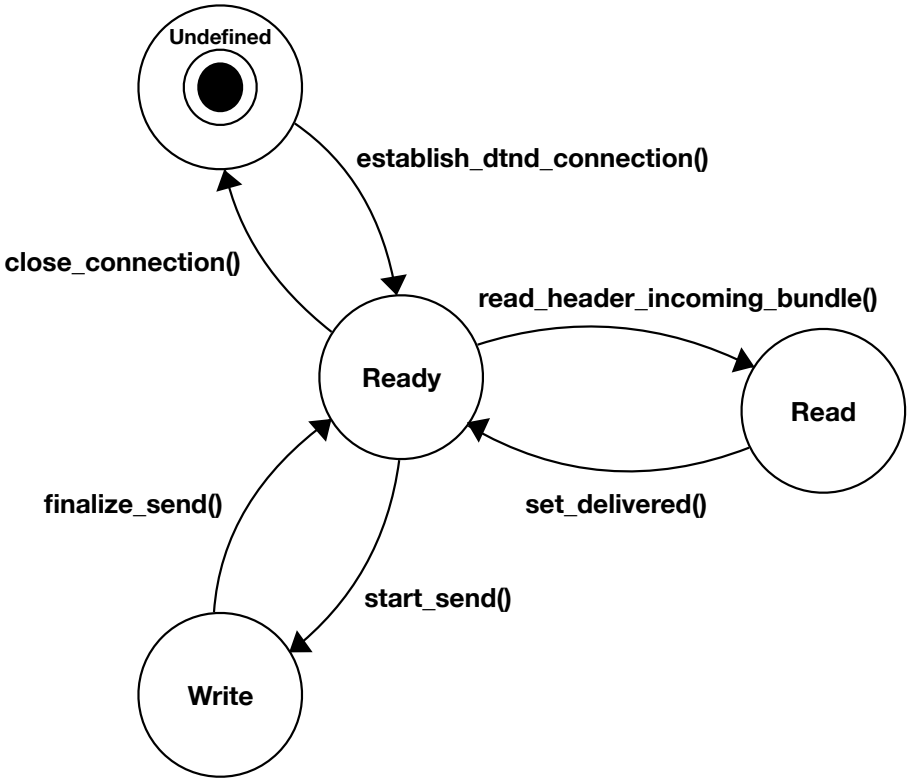


Figura 6.1: State machine su cui si basa il codice delle modifiche

Grazie all'uso della macchina a stati è stato possibile riconoscere ogni possibile bug e lo scorretto comportamento nei confronti di IBR-DTN, in quanto una violazione della macchina a stati, porta il codice ad una istruzione di `abort()` che grazie al tool di debug `valgrind` permette la visualizzazione di quale percorso ha portato a tale errore.
Inoltre la macchina a stati ha permesso il riconoscimento anche di comportamenti inaspettati da parte di IBR-DTN, come l'invio di notifiche durante uno stato in cui queste non

sono attese.
Nel file `bundle.h` vengono quindi definiti gli stati possibili.

```
    /* state machine of a connection with IBR-DTN using the Extended
       API (EAPI) */
typedef enum{
       EAPI_STATUS_UNDEFINED,
       EAPI_STATUS_READY,
       EAPI_STATUS_PAYLOAD_READ,
       EAPI_STATUS_PAYLOAD_WRITE,
} EAPI_STATUS;
```

- EAPI_STATUS_UNDEFINED: definisce lo stato in cui il socket non è ancora stato creato o è stato appena chiuso. Ovviamente in questo stato il file descriptor non deve essere utilizzato
- EAPI_STATUS_READY: definisce lo stato in cui il socket è connesso e ha appena concluso una operazione di read o di write nei confronti di IBR-DTN, in questo stato è possibile passare a tutti gli altri 3 possibili
- EAPI_STATUS_READ: definisce lo stato in cui il socket è in modalità di lettura di un bundle, in questa fase non è permessa alcuna scrittura ma bisogna porre l'attenzione a delle letture di notifiche inaspettate che andranno conservate in una lista di elaborazione
- EAPI_STATUS_WRITE: definisce lo stato in cui il socket è in modalità di scrittura di un bundle o durante l'invio di un comando verso IBR-DTN

Il modello che segue la macchina a stati è lo stesso per le due posizioni client-side e server-side.

Come è possibile notare il socket inizia e deve finire nello stato `EAPI_STATUS_UNDEFINED` che garantisce il corretto rilascio di tutte le risorse utilizzate.

6.3 Strutture globali

I socket una volta aperti vengono associati ad un file descriptor, il quale viene rappresentato da un numero intero e sequenziale che può essere riutilizzato dai vari processi esistenti in quanto sono mantenuti dal kernel nella file descriptor table differente per processo.
Sotto questa garanzia sono stati allocati dei vettori da massimo 1024 celle che contengono le informazioni relative alle demarcazioni, ai dati da utilizzare e alle proprietà associate ai file descriptor. Gli array, descritti nel file `bundle.c`, utilizzati sono:

```
static unsigned char marked_fd[MAX_FD];
static ssize_t payload_length[MAX_FD];
static Dtnd_bundle_id* dtnd_ids_queue[MAX_FD];
```

```
static Dtn_service* services[MAX_FD];
static EAPI_STATUS fd_status[MAX_FD];
static char* remote_eids[MAX_FD];
```

- `static unsigned char marked_fd[MAX_FD]`: array che contiene i valori 0 o 1, inizialmente tutte le caselle sono a zero descrivendo una situazione iniziale in cui non si conosce se il socket verrà utilizzato verso IBR-DTN o verso la rete internet.
  Zero rappresenta l'uso di una normale connessione TCP mentre la presenza di un uno significa che il socket è connesso a IBR-DTN.

- `static ssize_t payload_length[MAX_FD]`: array che contiene la lunghezza del payload dell'ultimo bundle che quel particolare file descriptor dovrà totalmente consumare, fino a quando questa non è zero non è possibile considerare come processato il bundle.

- `static Dtnd_bundle_id* dtnd_ids_queue[MAX_FD]`: array che contiene la lista di informazioni sui bundle che il file descriptor dovrà processare. La lista viene popolata all'arrivo di una notifica 602 contente le informazioni sul bundle da caricare.
  Si è reso necessario l'uso di una lista in quanto le notifiche 602 possono essere inoltrate da IBR-DTN in qualunque momento, rischiando di infrangere la state machine; con una lista si ovvia a questo problema perchè si permette di posticipare il lavoro sul prossimo bundle.

- `static Dtn_service* services[MAX_FD]`: array utilizzato solo se Tinyproxy è posto sul lato in cui è presente un servizio web da raggiungere. Contiene la struttura che descrive i dettagli IP del web server e i nomi DTN del servizio offerto (EID e nome applicazione).

- `static EAPI_STATUS fd_status[MAX_FD]`: array che associa ad un file descriptor uno stato della macchina a stati.

- `static char* remote_eids[MAX_FD]`: array che associa ad un file descriptor l'ultimo EID posto in source all'ultimo bundle processato. Consentendo in questo modo di poter continuare ad inviare bundle di risposta anche senza ricevere bundle di richiesta, nel caso in cui si tratti di una comunicazione tramite web socket.

I dettagli delle struct `Dtnd_bundle_id e Dtn_service` sono riportati in appendice B.

6.4 Funzioni di lettura e scrittura su bundle

Per permettere la lettura e la scrittura di dati sui due socket di connessione, Tinyproxy ci fornisce 2 funzioni che inglobano in loro tutti i controlli necessari a leggere o a scrivere in modo sicuro e controllato.
Per non dover propagare le modifiche a tutto il codice e permettere la perfetta compatibilità sono state create altre due funzioni con lo stesso nome che inglobano la chiamata alla

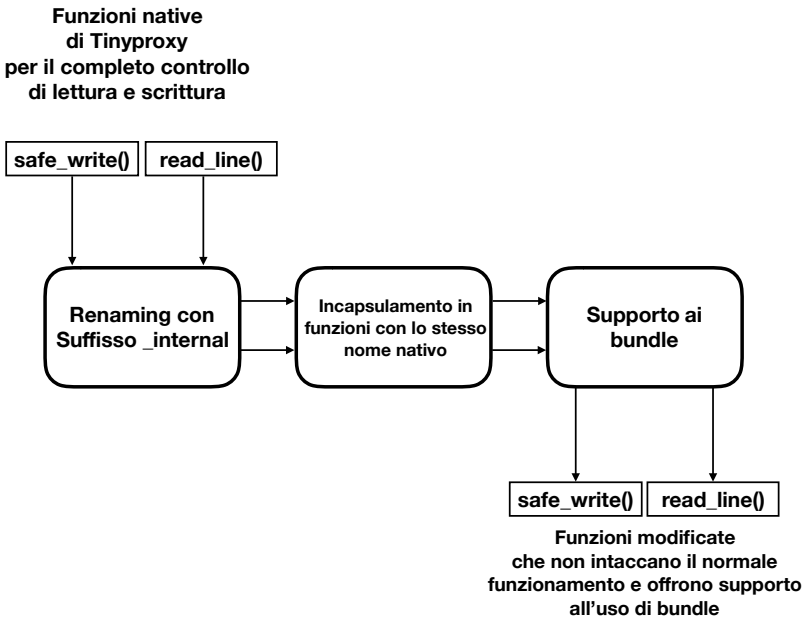


Figura 6.2: Percorso di modifica delle funzioni di lettura e scrittura

loro rispettiva funzione nativa, che per lo scopo è stata richiamata utilizzando il suffisso `_internal`. In questo modo la gestione della scelta se scrivere su protocollo HTTP o bundle è molto più semplice e in caso di disattivazione del codice bundle, si ha la piena sicurezza che tutto il codice continui a funzionare.

6.4.1 Lettura

La lettura di un bundle ha bisogno in più solo del controllo sul decremento del numero di byte rimanenti del bundle. Per permettere questo si cerca primo il minimo tra una costante e i byte rimanenti del bundle, di modo da non tentare di leggere dati inesistenti.

```
    #define MAXIMUM_BUFFER_LENGTH (128 * 1024)
    ssize_t readline(int fd, char **whole_buffer){
#ifdef BUNDLE_PROXY
        ssize_t max_chars, read_chars;

        if(bundle_is_fd_dtnd(fd)){
                ssize_t payload_length;
                bundle_verify_status(fd, EAPI_STATUS_PAYLOAD_READ);
                payload_length = bundle_get_payload_length(fd);

                if(payload_length == 0)
```

```
                        return 0;

                max_chars = min(MAXIMUM_BUFFER_LENGTH, payload_length);
        }
        else
                max_chars = MAXIMUM_BUFFER_LENGTH;

    /*call to the renamed real readline*/
        read_chars = readline_internal(fd, whole_buffer, max_chars);

        if(bundle_is_fd_dtnd(fd))
                bundle_dec_payload_size(fd, read_chars);

        return read_chars;
#else
        return readline_internal(fd, whole_buffer,
           MAXIMUM_BUFFER_LENGTH);
#endif
}
```

6.4.2 Scrittura

La scrittura al contrario ha bisogno di molto più supporto, in quanto per permettere la scrittura completa di un bundle nel momento in cui si ha la disponibilità dei dati da inviare, IBR-DTN ci permette l'uso del comando `payload append`. In questo modo possiamo appendere ad un bundle in costruzione tutti i dati necessari sapendo in un dato momento qual è la loro dimensione. Questo metodo è ottimo ma costa in termini di perfomance in quanto IBR-DTN non sembra essere efficiente nell'utilizzo di questo comando. La modifica sostanziale alla chiamata di questa funzione, consiste nel contornare la chiamata alla `safe_write_internal` con l'invio di tale comando a IBR-DTN.

```
ssize_t safe_write (int fd, const void *buf, ssize_t count)
{
#ifndef BUNDLE_PROXY
        return safe_write_internal(fd, buf, count);
#else
        char command[512];
        ssize_t len;

        if(count == 0)
                return 0;

        if (!bundle_is_fd_dtnd(fd)){
                return safe_write_internal(fd, buf, count);
```

```
        }
        bundle_verify_status(fd, EAPI_STATUS_PAYLOAD_WRITE);
        /* compose a PAYLOAD APPEND command*/
        if (bundle_send_command("payload append\n", fd) == -1){
               return -1;
        }

        /*compose the payload header*/
        snprintf(command, 512,
                       "Length: %zu\n"
                       "Encoding: raw\n"
                       "\n", count);

        len = strlen(command);
    /*/*call to the renamed real safe_write*/
    if (safe_write_internal(fd, command, len) != len)
               return -1;
        /*send the actual data*/
        if (safe_write_internal(fd, buf, count) != count)
               return -1;
        /*send the footer*/
        if (bundle_send_command("\n\n", fd) == -1)
               return -1;

        return count;
#endif
}
```

6.5 Tinyproxy lato server execution flow

Dopo aver parsificato da file di configurazione i dettagli IP di IBR-DTN e sul nome delle applicazioni che vengono gestite da Tinyproxy, vengono connessi a IBR-DTN tanti socket quanti sono i nomi configurati.
Durante l'implementazione di questa parte si è notato che non è corretto eseguire, subito aver creato la connessione verso IBR-DTN, il comando `set endpoint`, in quanto, anche se questo permette di ottenere in minor tempo l'EID completo, se esistono dei bundle fermi in attesa, questi vengono subito recapitati al socket corrompendo così il processo di connessione a IBR-DTN del socket con dei messaggi 602 inattesi. Perciò si è utilizzato il comando `nodename` che permette di conoscere il nome del nodo DTN su cui l'applicazione sta girando a cui poi appendere l'app name preso da file di configurazione, successivamente una volta compiuti tutti i passi collegamento, allora è possibile chiamare senza errori il comando `set endpoint`.

```
/* if on the config file is specified that Tinyproxy is on the
    server*/
if (config.dtnd_serverside) {
    if(config.dtn_appnames == NULL){
        log_message (LOG_ERR, "No application names for your servers
            are specified");
    }
    else{
        /*getting all the configured application names*/
        for (i = 0; i < vector_length(config.dtn_appnames); i++) {
            Dtn_service *ds;
            ds = (Dtn_service*) vector_getentry(config.dtn_appnames,
                i, NULL);

            if (ds == NULL) {
                log_message(LOG_WARNING,
                        "got NULL from Appnames - skipping");
                continue;
            }
            /*establish a connection towards IBR-DTN*/
            ret = bundle_establish_dtnd_connection();

            if (ret < 0) {
                return ret;
            }

            /*getting only the node name*/
            ds->app_eid = bundle_build_nodename(ret, ds->app_name);

            /*filling the services[] array*/
            bundle_set_service(ret,ds);

            log_message(LOG_INFO,"Listening for application %s
                %s::%d %s", ds->app_name, ds->ip_address, ds->port,
                ds->app_eid);

            /*last command we can send to DTND for receiving the 602
                messages*/
            bundle_set_endpoint(ret, ds->app_name);

            /*putting the new socket into the vector that Tinyproxy
                uses for its execution flow
            vector_append (listen_fds, &ret, sizeof(int));

        }
    }
```

```
}
```

Successivamente alla creazione di questi socket di ascolto, Tinyproxy rimane in attesa per dei dati in arrivo su tutti i socket presenti nel vettore `listen_fds`. Al primo dato in lettura che system call `select` riconosce, risveglia tutti processi in attesa sui socket; si è perciò utilizzato un meccanismo di lock per il quale solo uno dei processi potrà continuare ad elaborare una richiesta, gli altri andranno in attesa forzata di 10ms.
Se il dato in lettura è una notifica 602 allora solo i socket che sono stati marcati come DTN nel vettore `marked_fd` potranno raccogliere il nuovo bundle in arrivo. Dopo aver parsificato la 602 NOTIFY e quindi creato un dato di tipo `Dtnd_bundle_id` che identifica univocamente un nuovo bundle, si procede alla creazione di un nuovo socket DTN che perdura per tutta la durata dello scambio di bundle per una determinata connessione, il quale avrà esattamente le stesse strutture dati del socket globale di ascolto ma non utilizzando il comando `set endpoint` non sarà possibile inviare dei bundle ad esso senza che questi provengano da una connessione di tipo `keep-alive`.

```
if (bundle_is_fd_dtnd(listenfd)) {
    Dtnd_bundle_id *dtnd;
    /*Lock for concurrency*/
    SERVER_COUNT_LOCK();
    /*The listening fd parse the 602 NOTIFICATION and returns a
        Dtnd_bundle_id
    dtnd = bundle_react_to_notify(listenfd);

    /*if the current process has got a valid Dtnd_bundle_id
    if (dtnd != NULL) {
        /*create a new socket*/
        connfd = bundle_establish_dtnd_connection();

        /*copy information about the service, from the global socket
            to the temporary socket*/
        bundle_copy_service(listenfd, connfd);

        /*enqueue the new bundle to the task list of the temporary
            socket*/
        bundle_enqueue_bundle_id(connfd, dtnd);

        /*setting the name that will be use as destination of this
            communication*/
        bundle_set_remote_eid(connfd,
            safestrdup(bundle_get_remote_eid(listenfd)));

        /*global socket do not communicate, so in order to avoid
            memory leaks, the destination EID is free'd*/
        bundle_clear_remote_eid(listenfd);
```

```
            socket_blocking(connfd);
        }
        /*release concurrency structures*/
        SERVER_COUNT_UNLOCK();

        /*if the Dtnd_bundle_id is invalid means that the process must
            not continue so it will rest for 10ms*/
        if (dtnd == NULL) {
            usleep(10);

            /*continue the cycle and return to the select for waiting
                new data*/
            continue;
        }
    }
```

A questo punto viene richiamato il cuore di tutta la comunicazione, ovvero, la funzione `handle_connection`. Questa si occupa di tutto il parsing degli header, dell' inoltro di richiesta e risposta e di gestione del socket temporaneo.

6.5.1 Funzione Handle Connection

Quando il processo entra in questa funzione, con il comportamento `server-side` attivo e dopo le dovute inizializzazioni delle strutture dati, si procede ad ottenere l'endpoint temporaneo con il seguente codice:

```
    if(bundle_is_fd_dtnd(connptr->client_fd)){
        /*getting the random EID that IBR-DTN created for us*/
        connptr->client_local_EID =
            bundle_get_local_endpoint_eid(connptr->client_fd);
    }
```

dove `connptr->client_fd` rappresenta il socket di connessione verso IBR-DTN. L'ottenimento del nome temporaneo è essenziale per permettere le comunicazioni keep-alive in quanto consente la creazione di bundle con source e destination definiti in base alle richieste fatte sulla connessione.
Quindi una volta pronte tutte le informazioni necessarie alla comunicazione, grazie all'estensione del comportamento normale di Tinyproxy al supporto di HTTP/1.1, ritroviamo la scelta di metterci in attesa su una `select` per aspettare nuovi bundle o iniziare a parsificare un possibile nuovo bundle.

```
/*label for the keep-alive connection*/
init_conn:
```

```
                /* connection will be closed at the end, unless the
                   server responds with a Keep-Alive. */
                connptr->conn_close = CLOSE;

#ifdef BUNDLE_PROXY
                /* start a new request/response cycle */
                client_receiving = client_sending = server_receiving =
                   server_sending = 0;

        /*if the last header Connection was Keep-alive, we are still
           connected to a web-server*/
        if(bundle_is_fd_dtnd(connptr->client_fd) && server_connected) {

    /* wait for the next request bundle in a kept-live connection */
             if(bundle_wait_bundle_on_keep_alive_connection(
                 connptr->client_fd)){
                             request_not_created = 1;
                             goto fail;
                      }
                }

        /*in every case if there is a new bundle, we need to process
           it*/
        if(bundle_is_fd_dtnd(connptr->client_fd)){
                log_message (LOG_INFO,"Parsing header of the incoming
                   bundle");

            /*here Tinyproxy will parse all the Bundle headers that wont
               be sent to the web server*/
                if(bundle_read_header_incoming_bundle(connptr->client_fd)
                   < 0){
                             indicate_http_error (connptr, 500, "Unable
                                to connect",
                             "detail",
                             PACKAGE_NAME " "
                             "Cannot communicate with IBR-DTN",
                             "error", strerror (errno), NULL);
                             goto fail;
            }
            /*now we are receving data from the IBR-DTN logical client*/
                client_receiving = 1;
        }
#endif
```

Le funzioni `bundle_wait_bundle_on_keep_alive_connection` e `bundle_read_header_incoming_bundle` sono riportate in appendice B.

Arrivati a settare la variabile `client_receiving`, il lavoro di Tinyproxy continua come se fosse un normale Proxy HTTP/1.1, parsificando l'azione richiesta e tutti gli header che questa porta con sè, facendo particolare attenzione al rewriting del campo `Host`. Il rewriting è necessario in quanto si è scelto che tutte le request abbiano tale campo settato a `localhost`, quindi senza questa azione il web server rifiuterà la request impedendo la corretta comunicazione. Quindi al momento della funzione `get_all_headers` si è aggiunto un pezzo di codice che riscrive nel campo `Host` l'indirizzo IP settato in fase di configurazione.

```
    /*cycle for getting the header line from the socket*/
    for (count = 0; count < MAX_HEADERS; count++) {
        /*reading a line*/
        if ((linelen = readline (fd, &line)) <= 0) {
                safefree (header);
                safefree (line);
                return -1;
        }

#ifdef BUNDLE_PROXY
        /*if we are on a bundle connection and we recognise the Host
            header
        if(bundle_is_fd_dtnd(fd) && strncasecmp(line, "Host:", 5) == 0){

            char *tmp2 = safestrdup(line), *res;
            /*get the right value for the header*/
            res = bundle_rewrite_host(tmp2);
            /*if there is no error*/
            if (res != NULL){
                safefree(line);
                /*silently substitute the header value*/
                line = safestrdup(res);
            }

            log_message(LOG_INFO,"rewrited Host: %s", line);
           }

#endif
...
```

La funzione `bundle_rewrite_host` è riportata in appendice B.

Dopo queste minime modifiche il ciclo di funzionamento di Tinyproxy rimane invariato, connettendosi al web-server richiesto e inviando la request ad esso, fino al momento del parsing degli header della response da parte del server.
Supportando l'HTTP/1.1 è necessario controllare l'header Connection della risposta del

web-server; la scelta implementativa impone che sia il web-server a decidere le sorti della connessione su cui le informazioni sono trasportate.
In più per dar supporto alla possibilità di utilizzo del metodo HTTP `CONNECT` è necessario controllare se il web-server richiede l'upgrade della connessione su protocollo websocket (di fatto scambiando dati binari puri senza l'uso di HTTP).

```
static int process_server_headers (struct conn_s *connptr)
{
    ...
    /*getting from the Tinyproxy hashmap of header the header
        connection*/
    length = hashmap_entry_by_key (hashofheaders, "connection", (void
        **) &data2);

    /*if exists*/
    if(length > 0){
        /*check what type of connection the server wants*/
        connptr->conn_close = check_connection_type(data2);
    }
    /*getting from the Tinyproxy hashmap of header the header upgrade*/
    length = hashmap_entry_by_key (hashofheaders, "upgrade", (void **)
        &data2);

    /*if exists*/
    if(length > 0)
    {
        /*check if the server wants to use a websocket*/
        if(strcasecmp(data2, "websocket") == 0){
            log_message(LOG_INFO,"Upgrade to Websocket");
            /*setting that we need a websocket*/
            connptr->upgrade_to_connect_method = 1;
        }
    }
    ...
```

A questo punto si entra nel ciclo di relay: questa funzione potrà essere utilizzata 2 volte, se la response del server necessità dell'upgrade al protocollo web-socket.

```
 relay_connection (connptr);

    if (connptr->upgrade_to_connect_method) {
            connptr->connect_method = 1;
            relay_connection (connptr);
    }
```

Il supporto all'HTTP/1.1 richiede l'immediato controllo di una possibile risposta senza payload da parte del server.

In questo caso viene immediatamente inviato il bundle o se si tratta di una risposta HTTP pura si esce dal ciclo di relay evitando così inutili attese.

```
    /*if it's no a connect method, so there is a HTTP response/*
    if(!connptr->connect_method){
        /*if it's a keep-alive connection*/
        if(!connptr->conn_close){
            /*but the response has no payload*/
            if(connptr->content_length.server <= 0){
#ifdef BUNDLE_PROXY
                /*send the bundle, it will contains only headers*/
                if (bundle_is_fd_dtnd(connptr->client_fd) &&
                    bundle_finalize_send(connptr->client_fd,
                    connptr->client_local_EID)) {
                            log_message(LOG_ERR,"Failed to finalize
                                response");
                            return;
                        }
#endif
                            return;
            }
            /*there is something to relay*/
            else
                has_content_length = 1;
        }
    }
```

Quindi per supportare al meglio la connessione, se si tratta di un socket connesso a IBR-DTN, bisogna evitare il settaggio in modalità `non-blocking` del socket. Se impostato non verranno attesi i dati sul socket e la comunicazione verrà interrotta.

```
    #ifdef BUNDLE_PROXY
        if(!bundle_is_fd_dtnd(connptr->client_fd)) {
#endif
                        ret = socket_nonblocking (connptr->client_fd);
                        if (ret != 0) {
                                    log_message(LOG_ERR, "Failed to set
                                        the client socket "
                                    "to non-blocking: %s",
                                        strerror(errno));
                                    return;
                        }
#ifdef BUNDLE_PROXY
        }
```

Tinyproxy, a questo punto, inizia a settare i file descriptor presenti per la successiva chiamata alla system call `select`. In questo caso abbiamo una particolarità fondamentale che permette il corretto funzionamento di tutto. Come già spiegato il codice è progettato

secondo una macchina a stati ma esiste in qualunque momento la possibilità di ricevere una 602 NOTIFY che deve essere gestita. Quindi prima di metterci in attesa sul timer della `select` è bene controllare se non ci siano già delle notifiche arrivate inspettatamente. Se esse sono presenti si procede ad azzerare il timer, evitandfo così un'inutile attesa.

```
#ifdef BUNDLE_PROXY
/*if we are in connect method over a DTN socket
if(connptr->connect_method && bundle_is_fd_dtnd(connptr->client_fd)){

    /*we already have a bundle to process?*/
    if(bundle_is_bundle_id_queue_empty(connptr->client_fd))
           FD_SET (connptr->client_fd, &rset);
        else
            /*if yes do not wait on the select*/
            tv.tv_sec=0;
    }else
#endif
           FD_SET (connptr->client_fd, &rset);
}
```

Se quindi è stato richiesto l'upgrade al web-socket il ciclo di relay viene impostato per inviare o ricevere dei bundle ogni qual volta sia presente qualcosa sul socket.

```
    if(connptr->connect_method) {
        ...
        /*if we are reading from the IBR-DTN socket*/
        if(bundle_is_fd_dtnd(connptr->client_fd)){

            /*isn't there at least a bundle in the queue?*/
            if(bundle_is_bundle_id_queue_empty(connptr->client_fd)){

                /*reading from the IBR-DTN socket
                if(FD_ISSET (connptr->client_fd, &rset)){
                    Dtnd_bundle_id *db =
                        bundle_react_to_notify(connptr->client_fd);

                    /*put the bundle in the queue*/
                    bundle_enqueue_bundle_id(connptr->client_fd, db);
                }
            }
            /*getting the first bundle that was enqueued
            * getting the HTTP request and set the bundle as delivered
            */
            if(!bundle_is_bundle_id_queue_empty(connptr->client_fd)){
```

```
                bundle_read_header_incoming_bundle(connptr->client_fd);

                bytes_received = read_buffer (connptr->client_fd,
                    connptr->cbuffer);
                if (bytes_received < 0)
                    break;

                bundle_set_delivered(connptr->client_fd);
                continue;
            }
        }
        ...
        /*if we are writing into the IBR-DTN socket*/
        if(FD_ISSET (connptr->client_fd, &wset) &&
            bundle_is_fd_dtnd(connptr->client_fd)){
            int bytessent;
            /*getting the destination*/
            const char* remote_eid =
                bundle_get_remote_eid(connptr->client_fd);

            if(remote_eid == NULL)
                abort();

            /*create a new bundle and send it towards the DTN*/
            bundle_start_send(connptr->client_fd,
                connptr->client_local_EID, remote_eid);
            bytessent = write_buffer (connptr->client_fd,
                connptr->sbuffer);

            if(bytessent < 0)
                break;

            bundle_finalize_send(connptr->client_fd,
                connptr->client_local_EID);
            continue;
        }
    }
```

Il normale ciclo di relay non è comunque disattivato perchè la comunicazione HTTP con il web-server deve essere ancora supportata. Dopo questa fase, Tinyproxy controlla se sono rimasti ancora dei dati da inviare e quindi l'estensione al supporto del bundle protocol permette di incapsulare su bundle questi ultimi dati.
Come ultimo e importante passaggio controlla se è stato richiesta la chiusura della connessione, in caso affermativo viene tolta la possibilità di scrivere sul socket verso IBR-DTN.

```
    if(connptr->conn_close == CLOSE){
                log_message (LOG_INFO,
```

```
                                "Closed connection between local
                                    client (fd:%d) "
                                "and remote client (fd:%d)",
                                connptr->client_fd,
                                    connptr->server_fd);
            shutdown (connptr->client_fd, SHUT_WR);
    }
```

Se tutti i passaggi sono stati eseguiti con successo, il codice esegue un `goto` alla label `done` che indica la parte di rilascio delle risorse. L'estensione di Tinyproxy ad HTTP/1.1, però impedisce il rilascio delle risorse e la chiusura dei socket se il server ha bisogno di una connessione `keep-alive`:

```
if(connptr->conn_close == KEEP_ALIVE && !connptr->connect_method){
    goto init_conn;
}
```

Grazie alla label `init_conn` si ritorna all'inizio di tutto il ciclo utile della `handle_connection` per rimettersi in attesa di un nuovo bundle di richiesta o elaborarne uno già accodato da una 602 NOTIFY inaspettata. Se invece la connessione è da chiudere vengono rilasciate tutte le risorse e chiusi i socket. L'uscita dalla funzione `handle_connection` prevede solo l'aggiornamento dei counter e il cambiamento dello stato del processo, esattamente come già programmato nella versione originale di Tinyproxy.

Gestione del fallimento

Se uno dei passaggi necessari per la corretta comunicazione va in errore, Tinyproxy esegue un `goto` alla label di fail, che permette la corretta liberazione dei socket e l'invio di un messaggio di HTML ERROR di risposta.
Nel momento in cui si fallisce è possibile essere in due momenti, mentre si stava leggendo un bundle o mentre lo si stava scrivendo; a seconda della variabile `client_receiving` è possibile capire in quale stato è capitato il fallimento. Se stavamo leggendo un bundle è necessario consumare tutto il payload rimasto, invece se stavamo scrivendo in un nuovo bundle è necessario che venga inviato un comando a IBR-DTN per resettare l'ultimo bundle in costruzione.
Come ultimo passaggio si provvederà ad inviare un bundle contenente una pagina HTML di errore.

```
fail:
          /*
     * First, get the body if there is one.
     * If we don't read all there is from the socket first,
     * it is still marked for reading and we won't be able
     * to send our data properly.
```

```
         */
        if (get_request_entity (connptr) < 0) {
            log_message (LOG_WARNING,
                         "Could not retrieve request entity");
            indicate_http_error (connptr, 400, "Bad Request",
                                 "detail",
                                 "Could not retrieve the request entity
                                    "
                                 "the client.", NULL);
            update_stats (STAT_BADCONN);
        }

#ifdef BUNDLE_PROXY
        /*delivering the last bundle that we were reading, so IBR-DTN
            can destroy it*/
        if(bundle_is_fd_dtnd(connptr->client_fd) && client_receiving){
                bundle_set_delivered(connptr->client_fd);
        }
#endif

        if (connptr->error_variables) {
#ifdef BUNDLE_PROXY
                if(bundle_is_fd_dtnd(connptr->client_fd)){
                if(client_receiving)
                    /*start the bundle that will contains the HTML ERRROR
                        page*/
                    bundle_start_send(connptr->client_fd,
                        connptr->client_local_EID,
                    bundle_get_remote_eid(connptr->client_fd));
                else
                    /*consume the remaining payload*/
                        bundle_clear_payload(connptr->client_fd);
                }
#endif

                send_http_error_message (connptr);

#ifdef BUNDLE_PROXY
                if(bundle_is_fd_dtnd(connptr->client_fd))
                /*send the bundle that contains the error*/
                bundle_finalize_send(connptr->client_fd,
                    connptr->client_local_EID);
#endif
        }
        else if (connptr->show_stats) {
            showstats (connptr);
```

```
}

/* always close in case of failure */
connptr->conn_close = CLOSE;
```

6.6 Tinyproxy lato client: flusso di esecuzione metodi non CONNECT

Il flow di esecuzione di Tinyproxy lato client è molto più semplice e rispetto a quello server. Anzitutto fino al momento della `handle_connection` il codice di Tinyproxy non viene toccato, in quanto le connessioni a IBR-DTN vengono inizializzate solo nel momento in cui viene riconosciuto un URL di tipo DTN (4.3.1).
Una volta entrati nella fuzione, si arriva subito alla `process_request` la quale parsifica tutti gli header in arrivo sul socket da parte del browser, se si tratta della prima request per una connessione di tipo `KEEP-ALIVE` oppure una con connessione di tipo `CLOSE`, si procede a parsificare l'URL e a decidere se il server remoto è un nodo DTN oppure IP, il tutto è ottenuto grazie alla funzione `bundle_host_is_dtn` (riportato in appendice B).

```
    /*if we aren't in the middle of a KEEP-ALIVE connection
    if (!server_connected) {
        /* Try to connect to the remote server (first request in a HTTP
            1.1 stream) */
#ifdef BUNDLE_PROXY
        /*at this point we know if weed need to switch on bp*/
        if (bundle_host_is_dtn(request->host)){ /* remote server is
            DTND */
            char *remote_eid;
            connptr->server_fd = bundle_establish_dtnd_connection();

            ...
            /*getting a temporary EID from IBR-DTN for the connection*/
            connptr->server_local_EID =
                bundle_get_local_endpoint_eid(connptr->server_fd);
            log_message(LOG_INFO, "This is the UUID assigned from
                IBR-DTN: %s",
                    connptr->server_local_EID);

            /*from the header "host" format the right DTN destination
                EID*/
            remote_eid = bundle_format_server_eid(request->host);

            ...
            /*the once for all the remote EID*/
```

```
            bundle_set_remote_eid(connptr->server_fd, remote_eid);
            /*ibrdtn is up*/
            log_message (LOG_INFO,"Connected to IBR-DTND, sending
                bundles to \"%s\" using file descriptor %d.",
                request->host, connptr->server_fd);
        }
        else { /* remote TCP/IP server */
#endif
        /*Normal code of Tinyproxy that connect to a pure IP web
            server*/
        ...

#ifdef BUNDLE_PROXY
        }
#endif
        /*now we are officially connected to a server, if is a
            KEEP-ALIVE connection we must jump over this code*/
        server_connected = 1;
    }
```

A questo punto non essendo stato richiesto il metodo `CONNECT` da parte del browser si procede a creare un bundle che conterrà gli header di richiesta della risorsa; se siamo alla prima request allora il bundle sarà spedito verso l'EID costruito a partire dall'URL scritto dall'utente, mentre se siamo già nel mezzo di una connessione `KEEP-ALIVE` l'EID di destinazione equivale al nome temporaneo con il quale il server ha risposto alla nostra richiesta (6.5.1).

```
    if (!connptr->connect_method) {
#ifdef BUNDLE_PROXY
        /* prepare to send the request bundle to the remote server that
            is in the DTN network*/
        if(bundle_is_fd_dtnd(connptr->server_fd)){
            /*if is a KEEP-ALIVE connection the destination EID is
                reparsed from the responde bundle*/
            if(bundle_get_remote_eid(connptr->server_fd) == NULL){
                /*start to open a new bundle in IBR-DTN*/
                if(bundle_start_send(connptr->server_fd,
                    connptr->server_local_eid,
                    bundle_get_remote_eid(connptr->server_fd)) != 0)
                    goto fail;
            }
            /*first request, so the EID is the one got from the header
                "Host"*/
            else{
```

```
                /*start to open a new bundle in IBR-DTN*/
                if(bundle_start_send(connptr->server_fd,
                    connptr->server_local_EID,
                    bundle_get_remote_eid(connptr->server_fd)) != 0)
                        goto fail;
            }
        }
        /*now we are officially send something to the server*/
        server_sending = 1;
#endif
        /*rewrite header "host"*/
        establish_http_connection (connptr, request);
    }
}
```

IBR-DTN ha quindi ricevuto il comando di aprire un nuovo bundle e ha cominciato inserire nel payload, grazie alla funzione `establish_http_connection`, gli header necessari, attuando il rewriting del campo `Host` con il valore `localhost`. Nel caso in cui non sia realmente `localhost` l'indirizzo corretto del web server sul nodo destinazione, questo sarà corretto una volta arrivato al Tinyproxy sul lato server. La funzione `establish_http_connection` è riportata in appendice B.

```
        ...
#ifdef BUNDLE_PROXY

        ...
    /*Tinyproxy with its normal code has written all the remaining
        header,
    * our bundle is ready to be sent*/
    if(bundle_is_fd_dtnd(connptr->server_fd) &&
        !connptr->connect_method){
            if (bundle_finalize_send(connptr->server_fd,
                connptr->server_local_EID))
                goto fail;
            /* we are no more sending to the server*/
            server_sending = 0;

            /*wait on a select for the response bundle from the server*/
            if(bundle_wait_bundle_on_keep_alive_connection(connptr->server_fd))
                {
                indicate_http_error (connptr, 406, "Unable to
                    connect","detail",PACKAGE_NAME "Bad formatted bundle",
                        "error", strerror (errno), NULL);
                goto fail;
```

```
            }
            /*parse the bundle headers so the normal code of Tinyproxy
                will deal only with the HTTP response
            if(bundle_read_header_incoming_bundle(connptr->server_fd)) {
                indicate_http_error (connptr, 406, "Unable to
                    connect","detail",PACKAGE_NAME "Bad formatted bundle",
                        "error", strerror (errno), NULL);
                goto fail;
            }
            /*officially we are receving data from the server*/
            server_receiving = 1;
        }
#endif
```

Una volta scritta tutta la request, utilizzando completamente i metodi nativi di Tinyproxy, si procede ad inviare il bundle di richiesta e a mettersi in attesa del bundle di risposta, nello stesso modo in cui si è ottenuto questo comportamento lato server. Quindi una volta parsificati gli header del bundle, si procede a ricevere gli header HTTP e attraverso il metodo `relay_connection` a ricevere tutti i bundle contenenti i dati necessari alla connessione. In caso di successo di tutte le operazioni da eseguire l'ultima cosa che deve fare il client è notificare a IBR-DTN che l'ultimo bundle è stato correttamente processato.

```
    #ifdef BUNDLE_PROXY
        if(bundle_is_fd_dtnd(connptr->server_fd) &&
            !connptr->connect_method) {
        if (server_receiving){

            /* if there was an error free socket from useless data*/
            bundle_consume_payload(connptr->server_fd);

            /* set the last bundle as delivered*/
            bundle_set_delivered(connptr->server_fd);

            /* if we where sendind to the server something and there
            * was an error, force the state machine in order
            * to not block the execution*/
            if (server_sending)
            bundle_finalize_send_for_failed_bundle(connptr->server_fd);
        }
        }
#endif
```

6.6.1 Tinyproxy lato client: Flusso di esecuzione metodo CONNECT

Solo se posto in logicamente su un client, Tinyproxy, può ricevere una request con un metodo `CONNECT`. Questo deve essere trattato in modo speciale, in quanto c'è bisogno di permettere il passaggio da protocollo HTTP a protocollo Websocket.
Grazie a questa supporto del metodo `CONNECT` si guadagna in automatico il funzionamento di una connessione basata su HTTPS. Quindi, dopo aver fatto la connessione al server come nel caso normale, se viene riconosciuto l'uso del metodo `CONNECT` non verrà chiamato il metodo `establish_http_connection` ma Tinyproxy procederà ad inviare un messaggio di risposta preconfigurato attraverso la scrittura diretta sul socket verso il browser, con il metodo `send_ssl_response`.

```
/*
 * These two defines are for the SSL tunnelling.
 */
#define SSL_CONNECTION_RESPONSE "HTTP/1.1 200 Connection established"

#define PROXY_AGENT "Proxy-agent: " PACKAGE "/" VERSION

/*
 * Send the appropriate response to the client to establish a SSL
 * connection.
 */
static int send_ssl_response (struct conn_s *connptr)
{
       return write_message (connptr->client_fd,
                             "%s\r\n"
                             "%s\r\n"
                             "\r\n", SSL_CONNECTION_RESPONSE,
                                PROXY_AGENT);
}
```

In questo modo il browser riceverà una conferma di continuare l'handshake per la `CONNECT` e quindi invierà la successiva `GET` verso il web-server.
La richiesta richiederà una risorsa alla quale il web-server risponderà settando l'header `Upgrade: Websocket` implicando quindi l'uso di dati binari, in questo caso Tinyproxy sul lato client, una volta parsificati gli header in arrivo dal server e impostando il valore di `connptr->upgrade_to_connect_method` a 1, dovrà ripetere l'operazione di `relay_connection` per dedicare un processo ad attendere e a processare in continuazione i bundle contenenti dati binari inviati dal server ad intervalli regolari.

```
       /*when we receive the response in process_server_header we
           recognise that
```

```
  * the server wants to upgrade to websocket, so
      connptr->upgrade_to_connect_method became true
  */
    if (!connptr->connect_method || UPSTREAM_IS_HTTP(connptr)) {
          if (process_server_headers (connptr) < 0) {
                update_stats (STAT_BADCONN);
                goto fail;
       }
    }

    ...
    /*now this relay is used only to send the binary data from the
       client*/
    relay_connection (connptr);

    /*now we enter here and a process is dedicated only to relay
       binary data between client and server*/
    if (connptr->upgrade_to_connect_method) {
          connptr->connect_method = 1;
          relay_connection (connptr);
    }
```

Gestione degli errori

Quando Tinyproxy è posto sul lato di browser la gestione degli errori è minima e molto più semplice.
Ogni errore è solamente da recapitare in HTTP e l'unica azione da compiere su IBR-DTN è liberare il socket dai dati non processati, confermare la consegna del bundle e forzare la chiusura del socket e quindi del processo corrotto di Tinyproxy.

```
fail:
          /*
     * First, get the body if there is one.
     * If we don't read all there is from the socket first,
     * it is still marked for reading and we won't be able
     * to send our data properly.
     */
    if (get_request_entity (connptr) < 0) {
       log_message (LOG_WARNING,
                    "Could not retrieve request entity");
       indicate_http_error (connptr, 400, "Bad Request",
                            "detail",
                            "Could not retrieve the request entity
                               "
                            "the client.", NULL);
```

```
            update_stats (STAT_BADCONN);
        }
...
        if (connptr->error_variables) {
...
        send_http_error_message (connptr);
...
        }
        else if (connptr->show_stats) {
            showstats (connptr);
        }

        /* always close in case of failure */
        connptr->conn_close = CLOSE;
done:
#ifdef BUNDLE_PROXY
        if(bundle_is_fd_dtnd(connptr->server_fd) &&
            !connptr->connect_method) {
        if (server_receiving){
            /*consume remaining bundle data from IBR-DTN*/
            bundle_consume_payload(connptr->server_fd);
            /*set as delivered the last bundle*/
            bundle_set_delivered(connptr->server_fd);
            if (server_sending)
                /*in this case we need to force the state machine*/
                bundle_finalize_send_for_failed_bundle(connptr->server_fd);
        }
        }
#endif
    /*if the browser didn't send a request, these structures are
        invalid and unallocated
    *else we need to free these structures
    */
    if(!request_not_created){
        free_request_struct (request);
        hashmap_delete (hashofheaders);
    }

    ...
    /*connptr->conn_close is for sure CLOSE so we jump over this*/
    if(connptr->conn_close == KEEP_ALIVE && !connptr->connect_method){
        goto init_conn;
    }

    destroy_conn (connptr);
    log_message(LOG_INFO, "Connection successful closed");
```

```
    return;
}
```

Capitolo 7

Misurazioni e testing

Come ultimo step ci si è incentrati sul misurare l'impatto delle applicazioni utilizzate, IBR-DTN e Tinyproxy, per capire quanto queste siano adatte allo scopo. Sono stati effettuati sia test di connessione che test di performance su un normale portatile e sulla NetG5.

7.0.1 Profilazione di Tinyproxy

Per avere la sicurezza che il costo di Tinyproxy sia il minor possibile si è utilizzato il tool `callgrind` che permette di profilare tutte le chiamate del programma analizzato. Dal profiling si sono subito notate troppe chiamate ad una system call effettuate da un ciclo di lettura errato. Grazie a questo tool di poter imputare il ritardo alla creazione dei bundle e non al tempo di elaborazione di Tinyproxy.

7.1 Test Computazionale

In questo tipo di test si è andato a leggere grazie al comando `usr/bin/time` o all'istruzione `getrusage(RUSAGE_SELF, ...)` il tempo computazionale definito come `Self` ovvero il tempo passato nelle funzioni scritte dal programmatore e come `Sys` ovvero quello misurato nelle system call utilizzate dal processo. I tempi misurati si intendono per singolo processo istanziato.

7.1.1 Timing NetG5

I test sono stati fatti utilizzando solo la configurazione `server-side` in quanto la risorsa da ottenere era posta sul dispositivo. Si ricordano le specifiche della NetG5

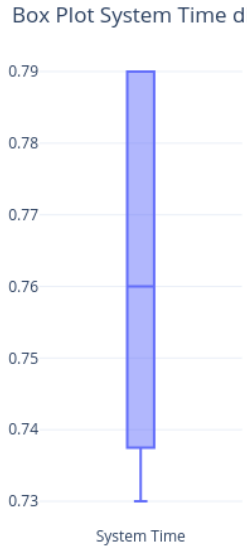
- SD 4GB
- 256MB Ram
- 250MHz processor

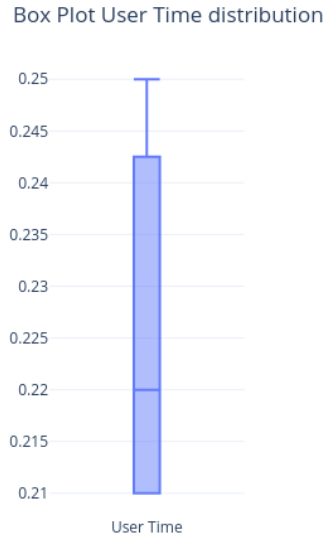
Tinyproxy

Per avere una corretta misurazione, è stato ripetuto per 5 volte lo stesso test di modo da ridurre il rumore dovuto al sistema su cui è installato il tutto. La configurazione di Tinyproxy è la seguente:

- Un processo padre che si occupa del loading del config
- 10 processi figli che si occupano di processare i dati

Come prima cosa si è andato a provare per 5 volte il tempo di caricamento per il processo padre del file di configurazione con `usr/bin/time`. Ottenendo un tempo medio, con la somma delle medie di `sys time` e `usr time`, di `0.988 s`





Dopo si è proceduto ad richiedere la prima pagina del sito di gestione della NetG5 per 5 volte con 10 processi attivi (7.1). In questo caso i risultati sono stati ottenuti con la stampa del risultato di `getrusage(RUSAGE_SELF, ...)` in quanto il comando `usr/bin/time` non dà informazioni utili su un programma multiprocesso. I risultati dei singoli box plot sono da intendere come la media dei tempi di tutti e 10 i processi attivi.

Sommando tutte le medie delle 5 prove si ha che Tinyproxy per sopperire alla richiesta della prima pagina posizionata sulla NetG5 ha bisogno di `2,664 s` in totale a cui bisogna sottrarre i `0.988 s` precendenti, ottenendo il vero tempo di esecuzione di `1.706 s`

IBR-DTN

Per avere una corretta misurazione, è stato ripetuto per 5 volte lo stesso test di modo da ridurre il rumore dovuto al sistema su cui è installato il tutto. Il test di loading è stato

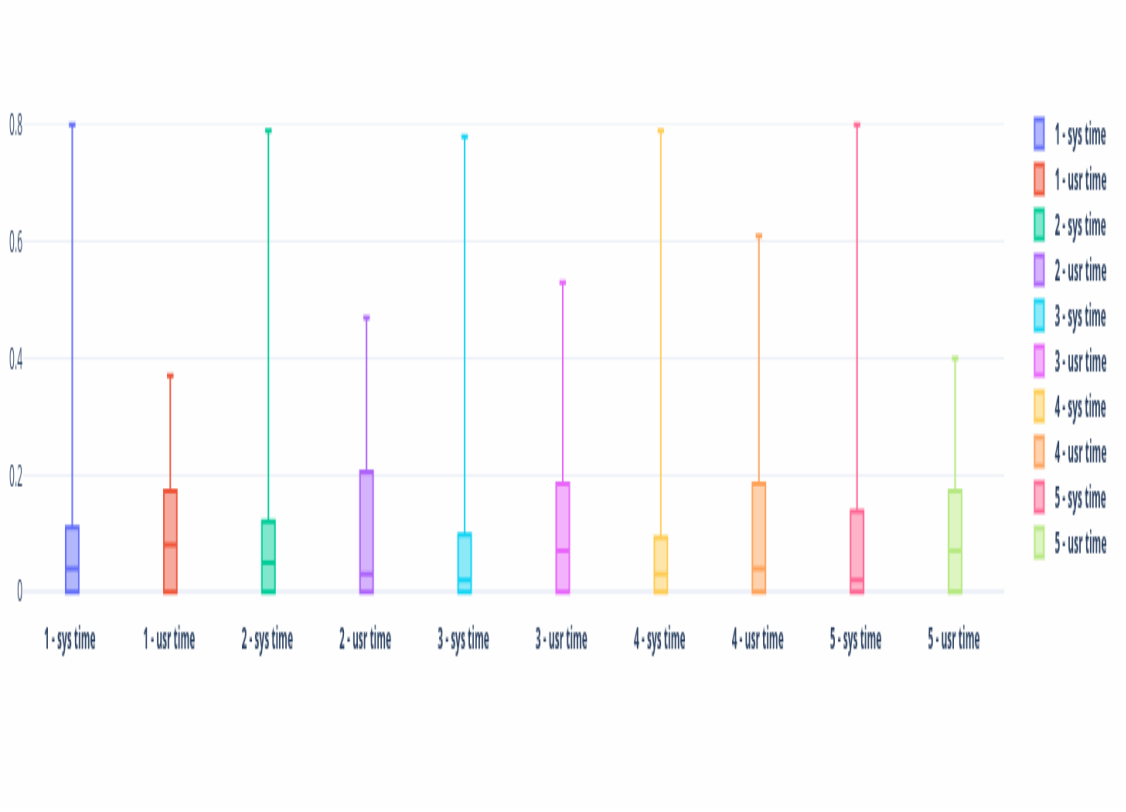


Figura 7.1: Tempi di elaborazione sulle 5 prove di caricamento della prima pagina

eseguito resettando tutte la cache di IBR-DTN e arrivando al primo output in cui IBR-DTN dichiara di essere connesso a tutte le interfacce. Per evitare di modificare il codice di IBR-DTN si è scelto di prendere tutti i tempi con il comando `usr/bin/time`
Come prima cosa si è andato a provare per 5 volte il tempo di caricamento di configurazione di IBR-DTN con `usr/bin/time` (7.2), ottenendo un tempo medio, con la somma delle medie di `sys time` e `usr time`, di `0.626 s`.

Dopo si è proceduto a richiedere la prima pagina del sito di gestione della NetG5 per 5 volte (7.3); IBR-DTN non usa fork ma crea solo dei thread. Quindi è possibile utilizzare ancora una volta il comando `usr/bin/time`. In questo caso i box plot sono da intendere come la rappresenzatione grafica di `sys time` e `usr time` sulle 5 prove. Non essendo multiprocesso non è necessario fare medie temporali per ogni prova.

Come si può ben notare dalla scala delle ordinate del grafico gran parte del tempo di elaborazione per la trasmissione è svolto per sopperire ai bisogni di IBR-DTN. In questo caso si ottiene un tempo medio di `5,922 s` a cui sottraendo il tempo di configurazione e binding si ottiene un tempo di utilizzo pari a `5,396 s`.

Figura 7.2: Tempi di configurazione IBR-DTN sys e usr time

7.1.2 Test su computer personale

Sono stati ripetuti gli stessi test sia in modalità `server-side` che `client-side` su un computer dalle seguenti specifiche:

- HDD 5400 rpm
- 8GB Ram
- Intel® Core™ i5-2410M @2.30GHz x 4

Tinyproxy Server-Side

Per avere una corretta misurazione, è stato ripetuto per 5 volte lo stesso test di modo da ridurre il rumore dovuto al sistema su cui è installato il tutto. La configurazione di Tinyproxy è la seguente:

- Un processo padre che si occupa del loading del config
- 10 processi figli che si occupano di processare i dati

Come prima cosa si è andato a provare per 5 volte il tempo di caricamento per il processo padre del file di configurazione con `usr/bin/time`. Ottenendo un tempo medio, con la somma delle medie di `sys time` e `usr time`, di `0.019282 s` (7.4).

Figura 7.3: Tempi di caricamento IBR-DTN sys e usr time

Dopo si è proceduto a richiedere la prima pagina del sito di gestione della NetG5 per 5 volte con 10 processi attivi (7.5), questa volta i risultati sono stati ottenuti con la stampa del risultato di `getrusage(RUSAGE_SELF, ...)` in quanto il comando `usr/bin/time` non dà informazioni utili su un programma multiprocesso. I risultati dei singoli box plot sono da intendere come la media dei tempi di tutti e 10 i processi attivi.

Sommando tutte le medie delle 5 prove si ha che Tinyproxy per sopperire alla richiesta della prima pagina posizionata sulla NetG5 ha bisogno di `0.2457116 s` in totale a cui bisogna sottrarre i `0.019282 s` precendenti, ottenendo il vero tempo di esecuzione di `0.2264296 s`.

Tinyproxy Client-Side

Per avere una corretta misurazione, è stato ripetuto per 5 volte lo stesso test di modo da ridurre il rumore dovuto al sistema su cui è installato il tutto. La configurazione di Tinyproxy è la seguente:

- Un processo padre che si occupa del loading del config
- 10 processi figli che si occupano di processare i dati

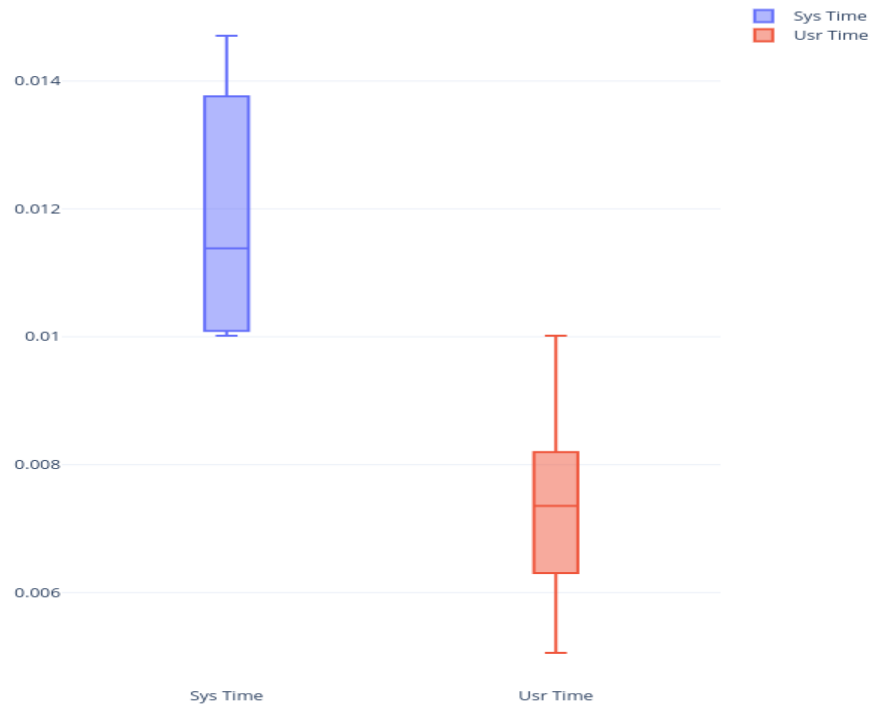


Figura 7.4: Tempi di caricamento Tinyproxy sys e usr time

Come prima cosa si è andato a provare per 5 volte il tempo di caricamento per il processo padre del file di configurazione con `usr/bin/time`. Ottenendo un tempo medio, con la somma delle medie di `sys time` e `usr time`, di `0.0182952 s` (7.6).

Dopo si è proceduto a richiedere la prima pagina del sito di gestione della NetG5 per 5 volte con 10 processi attivi 7.7, questa volta i risultati sono stati ottenuti con la stampa del risultato di `getrusage(RUSAGE_SELF, ...)` in quanto il comando `usr/bin/time` non dà informazioni utili su un programma multiprocesso. I risultati dei singoli box plot sono da intendere come la media dei tempi di tutti e 10 i processi attivi.

Sommando tutte le medie delle 5 prove si ha che Tinyproxy per sopperire alla richiesta della prima pagina posizionata sulla NetG5 ha bisogno di `0.0960548 s` in totale a cui bisogna sottrarre i `0.0182952 s` precendenti, ottenendo il vero tempo di esecuzione di `0.077596 s`.

IBR-DTN

Per avere una corretta misurazione, è stato ripetuto per 5 volte lo stesso test di modo da ridurre il rumore dovuto al sistema su cui è installato il tutto. Il test di loading è stato

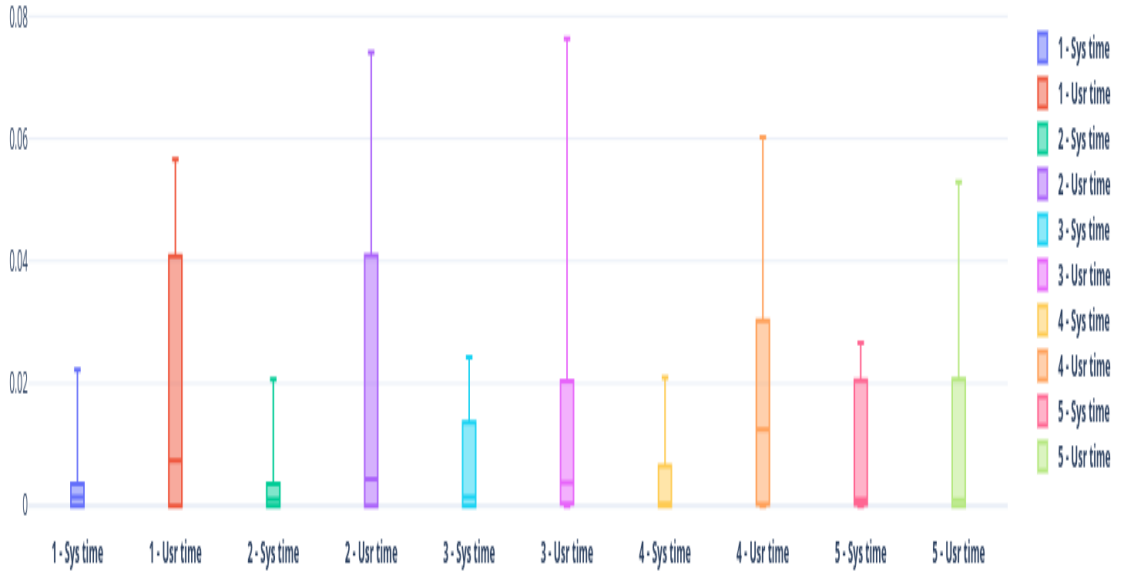


Figura 7.5: Tempi di elaborazione sulle 5 prove di caricamento della prima pagina

eseguito resettando tutte la cache di IBR-DTN e arrivando al primo output in cui IBR-DTN dichiara di essere connesso a tutte le interfacce. Per evitare di modificare il codice di IBR-DTN si è scelto di prendere tutti i tempi con il comando `usr/bin/time`.
Come prima cosa si è andato a provare per 5 volte il tempo di caricamento di configurazione di IBR-DTN con `usr/bin/time` (7.8). Ottenendo un tempo medio, con la somma delle medie di `sys time` e `usr time`, di `0.034 s`.

Dopo si è proceduto a richiedere la prima pagina del sito di gestione della NetG5 per 5 volte (7.9), IBR-DTN non usa fork ma crea solo dei thread. Quindi è possibile utilizzare ancora una volta il comando `usr/bin/time`. In questo caso i box plot sono da intendere come la rappresenzatione grafica di `sys time` e `usr time` sulle 5 prove. Non essendo multiprocesso non è necessario fare medie temporali per ogni prova.

Come si può ben notare dalla scala delle ordinate del grafico gran parte del tempo di elaborazione per la trasmissione è svolto per sopperire ai bisogni di IBR-DTN. In questo caso si ottiene un tempo medio di `0.608 s` a cui sottraendo il tempo di configurazione e binding si ottiene un tempo di utilizzo pari a `0.574 s`.

Figura 7.6: Tempi di caricamento Tinyproxy sys e usr time

Confronto e Risultati

Come è possibile subito notare, a causa di un CPU meno potente e delle risorse limitate, la NetG5 ha un tempo di elaborazione di ben 2 ordini di grandezza superiori per quanto riguarda i tempi di IBR-DTN e di un ordine di grandezza per quanto riguarda Tinyproxy. Il maggior tempo di elaborazione è ben dimostrato dall'incremento del tempo di caricamento della prima pagina, che passa da `2.30 s` medi a `15.27 s` in media.

7.2 Networking Test

Per il test di Networking si è scelto di testare il confronto del passaggio di dati dalla grandezza fissata attraverso Tinyproxy prima in HTTP puro e poi in Bundle Proxy. I file sono stati creati con l'uso del programma di sistema `/dev/urandom` e non potendo aggiungere file al web-server della NetG5 è stata sfruttata una Raspberry ARM per eseguire i test, connessa via Ethernet al computer su cui avvengono le misurazioni.

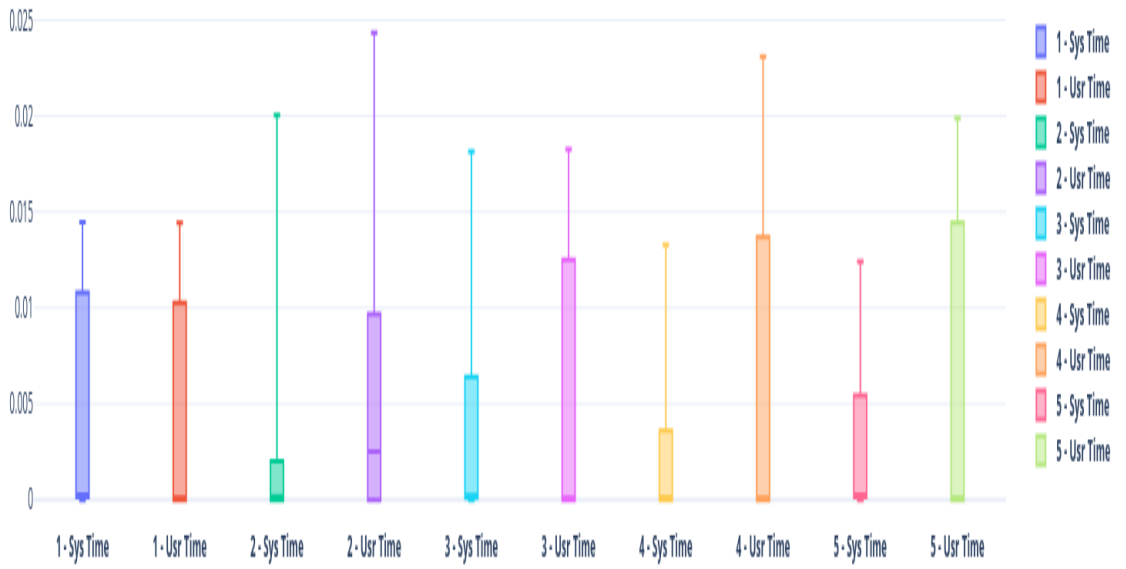


Figura 7.7: Tempi di elaborazione sulle 5 prove di caricamento della prima pagina

7.2.1 Test con 100 file da 10KB

Come si può notare in figura 7.10 si nota un netto incremento delle tempistiche di trasfermento, il tempo totale di trasferimento di HTTP è di `326 ms` con una latenza di totale di `227 ms`, mentre su Bundle protocol il tempo totale di trasferimento è di `99 s` che corrispondono esattamente al tempo di latenza `98 s`, questo significa la parte significativa che introduce ritardo è la creazione del bundle di risposta tramite il comando `payload append`

7.2.2 Test con 10 file da 1MB

Come si può notare in figura 7.11 si nota un netto incremento delle tempistiche di trasfermento, il tempo totale di trasferimento di HTTP è di `329 ms` con una latenza di totale di `26 ms`, mentre su Bundle protocol il tempo totale di trasferimento è di `137 s` che di poco maggiore al tempo di latenza `136 s`, questo significa la parte significativa che introduce ritardo è la creazione del bundle di risposta tramite il comando `payload append`

Figura 7.8: Tempi di configurazione IBR-DTN sys e usr time

7.2.3 Test con 1 file da 10MB

Come si può notare in figura 7.12 si nota un netto incremento delle tempistiche di trasfermento, il tempo totale di trasferimento di HTTP è di `307 ms` con una latenza di totale di `2 ms`, mentre su Bundle protocol il tempo totale di trasferimento è di `129 s` che corrispondono esattamente al tempo di latenza, questo significa la parte significativa che introduce ritardo è la creazione del bundle di risposta tramite il comando `payload append`

In tutti e tre i test si nota che tutti i trasferimenti vengono completati sia in HTTP che in Bundle protocol e non si segnalano corruzioni di dati. Il problema evidente è nella gestione della creazione del bundle di risposta.

Figura 7.9: Tempi di caricamento IBR-DTN sys e usr time

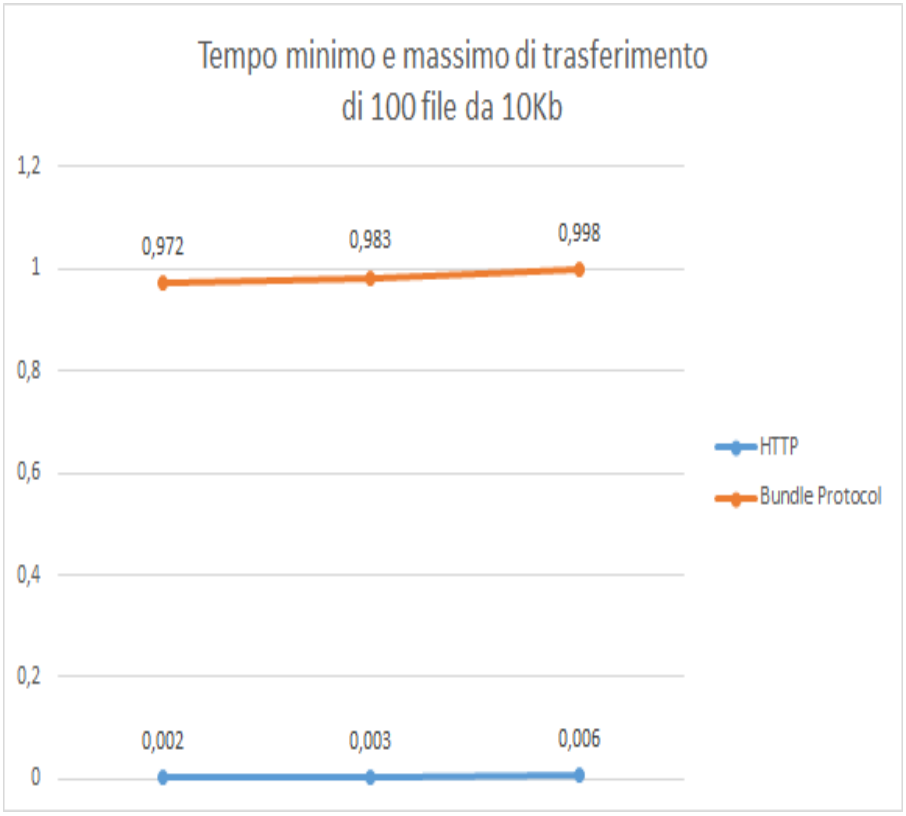


Figura 7.10: I tre punti rappresentano il minimo, il tempo medio e il tempo massimo di trasferimento in secondi

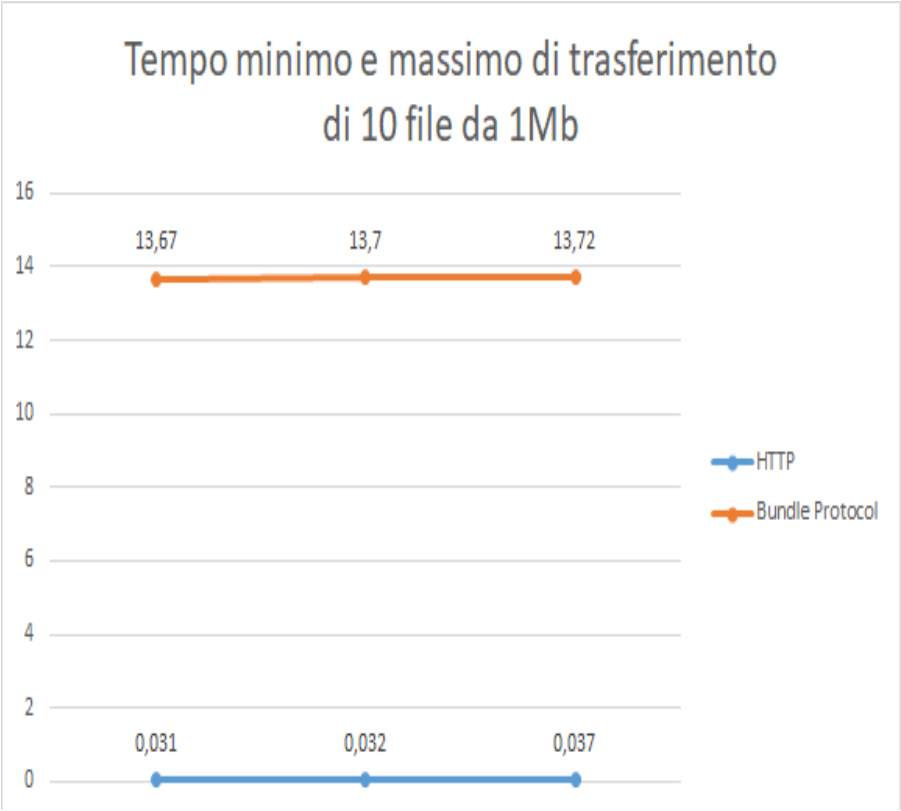


Figura 7.11: I tre punti rappresentano il minimo, il tempo medio e il tempo massimo di trasferimento in secondi

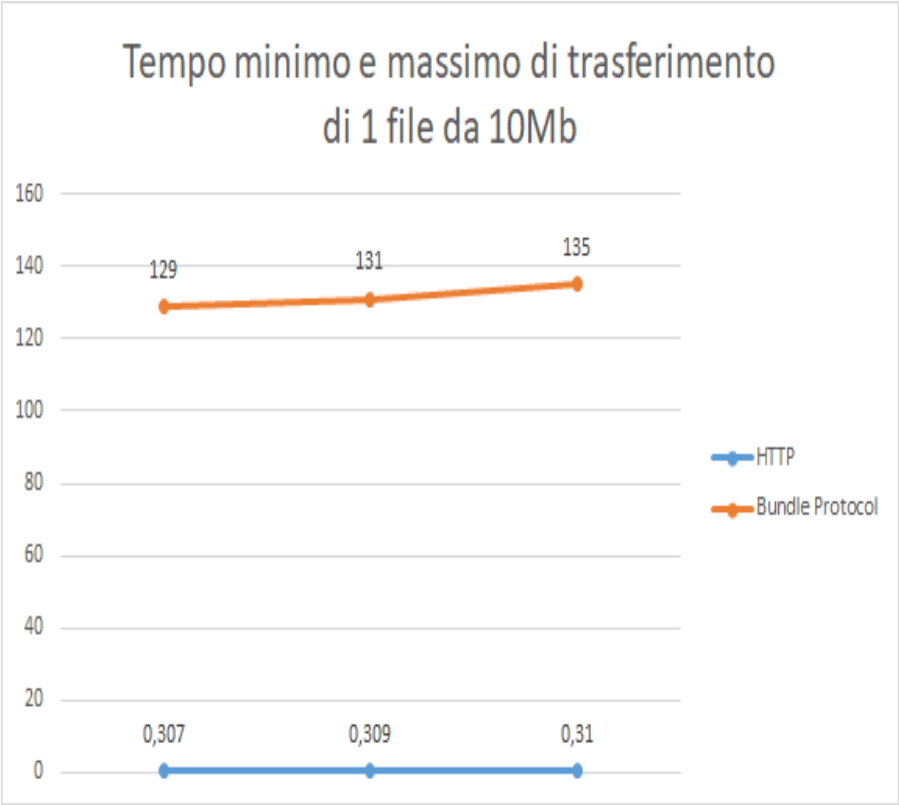


Figura 7.12: I tre punti rappresentano il minimo, il tempo medio e il tempo massimo di trasferimento in secondi

Capitolo 8

Conclusioni

Il lavoro svolto per ampliare le funzionalità di Tinyproxy da semplice proxy HTTP/1.0 ad HTTP/1.1 con supporto al Bundle Protocol permette di ampliare fortemente le potenzialità di una rete DTN. Come spiegato nei capitoli precendenti al giorno d'oggi avere un'interfaccia HTTP a disposizione su un sensore è molto comune e avere un applicativo che permette l'incapsulamento dei dati in un altro protocollo senza implicare la riscrittura di nessuna parte di software già perfettamente funzionante, è un pezzo necessario per rendere questo tipo di rete utilizzabile in qualunque ambito. Tinyproxy, in questo momento, richiede solo una configurazione molto semplice lato client, ovvero l'impostazione di un proxy a livello di browser o di sistema e la corretta compilazione del file di configurazione lato server.
Inoltre grazie all'uso di un dispositivo reale sono state permesse misurazioni realistiche e il supporto ad ogni metodo HTTP richiesto, la semplice creazione di un mini sito, avrebbe potuto non involvere tutte le complicazioni usate dal sito nativo presente sulla NetG5. Tutto il codice di Tinyproxy è stato sempre monitorato attraverso `valgrind` che ha permesso di trovare tutti gli errori di cattiva gestione della memoria e ogni possibile crash dovuto a questa.
L'uso della macchina a stati ha permesso il costante rilevamento di uso di azioni non permesse o non corrette, permettendo così l'esatta esecuzione di ogni step per arrivare alla fine della comunicazione.

Come è stato possibile notare dai test è evidente che la maggior parte del ritardo di trasferimento dei dati è dovuto soprattutto alla creazione del bundle di risposta attraverso IBR-DTN, difatti il tempo di latenza risulta essere quasi identico al tempo totale di trasferimento.
Come ulteriori sviluppi è necessario indagare sul motivo di questo impatto prestazionale da parte di IBR-DTN e tentare l'implentazione al supporto di altri protocolli, come ad esempio FTP, che sfruttano `SOCKS` i quali sono implementati da Tinyproxy nativamente ma non sono sfruttati per il collegamento tramite bundle.

Appendice

Appendice A

Configurazione applicativi e cross-compilazione per piattaforma SPARC

A.1 Configurazione degli applicativi

Tutti gli applicativi scelti per questo progetto sono stati configurati e compilati con l'utilizzo del tool Autoconf.
Autoconf è un package che produce script shell per la configurazione automatica dei pacchetti contenente il codice sorgente [19]. Date le limitatissime risorse della Net-G5 è stato opportuno disattivare, quanto più possibile, tutti i componenti opzionali di ogni programma scelto.

A.1.1 Dipendenze e moduli di IBR-DTN

IBR-DTN è il programma che in assoluto occupa più risorse in tutto il processo, costituito da molteplici moduli e dipendente da molte librerie difficoltose da riportare in ambiente SPARC, dunque è stato necessario cercare all'interno dei file per l'esecuzione di Autoconf tutte le opzioni disponibili. La documentazione presente online non è stata sufficiente per questo scopo, ma con un'attenta analisi del file `configure.ac` è stato possibile isolare le possibili funzionalità disattivabili. Tuttavia per permettere il corretto funzionamento della versione minima di IBR-DTN è stato necessario compilare alcune librerie non presenti già nella distribuzione Ångström fornita.

A.1.2 Moduli disattivabili

`--disable-dtndht`
Disabilità l'utilizzo delle Distributed Hash Tables.
Le dht sono un sistema chiave-valore che permettono identificare un nodo appartenente alla rete in modo efficiente attraverso un sistema di chiave, valore. Questa estensione è quella in assoluto più pesante e perciò per rendere l'applicativo più leggero è stata disattivata [20].

`--enable-libmount=no`
Libmount è una libreria da cui può dipendere IBR-DTN. Se tolta non affligge in alcun modo l'esecuzione e semplifica la cross compilazione utilizzando una libreria in meno.

`--without-glib`
Glib è una libreria che permette alcune ottimizzazione per la manipolazione di dati da parte di IBR-DTN.
Se tolta non affligge in alcun modo l'esecuzione e semplifica di molto la cross compilazione, in quanto è una libreria che a sua volta ha molte dipendenze per le quali non è assicurata la versione SPARC.

`--without-vmime`
Permette l'utilizzo dell'email convergence layer. Non è utile ai fini di questo progetto e perciò non è stato compilato.

`--without-curl`
Permette l'utilizzo di routing statico verso URL configurati. Non è utile ai fini di questo progetto e perciò non è stato compilato.

`--without-sqlite`
Permette l'utilizzo di sqlite per lo store di bundle in memoria. Non è utile ai fini di questo progetto e perciò non è stato compilato.

`--without-lowpan`
Disattiva il convergence layer LoWPAN. La Net-G5 non è dotata di tale via di comunicazione.

`--without-dtnsec`
Disattiva il protocollo di sicurezza su rete DTN. Per la fase iniziale del progetto non è necessario un layer di sicurezza, quindi è possibile disattivarlo.

`--without-compression`
Disattiva la compressione dei bundle per l'invio.

`--without-tls`
Disattiva il TLS.

`--without-flooding`

Disattiva un protocollo di routing del tutto inutilizzato. Per generare un eseguibile ancora più leggero è stato disattivato.

`--without-prophet`
Disattiva un protocollo di routing del tutto inutilizzato. Per generare un eseguibile ancora più leggero è stato disattivato.

A.1.3 Librerie richieste

Uno degli aspetti più utili e importanti di IBR-DTN è sicuramente la possibilità di rendere il processo un demone, ovvero permettere all'applicazione di continuare al lavorare in background mentre si eseguono altre applicazioni. Per permettere questo tipo di esecuzione è stata richiesta la compilazione della libreria `libdaemon` [21].
Un'altra libreria richiesta per la compilazione su SPARC è il package `libpcre`. Tale libreria permette la gestione corretta delle regex utilizzate dal IBR-DTN.

Tinyproxy non ha richiesto altre librerie o la disattivazione di moduli troppo pesanti.

A.2 Cross-Compilazione

A.2.1 Toolchain di cross compilazione

Per permettere la cross compilazione, l'immagine docker è stata corredata della toolchain GCC specifica per passare da architettura x86_64 a SPARC. La toolchain è un set di software per lo sviluppo legati tra loro attraverso gli specifici stage dello sviluppo di un software, come per l'appunto, i vari stadi di GCC. Molto spesso la toolchain è usata per la cross-compilazione per device di tipo embedded, conosciuto ai più come cross-compiler [22]. In questo caso `sparc-continuous-linux-gnu` permette di partire dall'architettura del sistema host (x86_64) per poi produrre codice binario eseguibile da far avviare sull'architettura SPARC.

A.2.2 Setup delle variabili d'ambiente

Per permettere la creazione di un corretto ambiente di avvio delle applicazioni è consigliabile, prima di cominciare a cross-compilare, creare un file con il setup delle variabili d'ambiente, nominato `env.sh`. Così facendo si può tramite, il comando `source env.sh`, settare a priori i valori dei campi citati semplificando la scrittura del comando `./configure`. Questi campi permettono al compilatore di ritrovare ed utilizzare le corrette librerie compilate in precedenza. Vengono quindi definite:

`TARGET`
Definisce l'architettura per la quale GCC producerà il codice compilato.

`HOST`
Definisce l'architettura della macchina sulla quale GCC sta producendo il codice compilato per la macchina TARGET.

`PREFIX`
Definisce il prefisso del path dove installare gli applicativi compilati una volta utilizzato il comando `make install`.

`SYSROOT`
Definisce il path dove gli script devono cercare le librerie già presenti di base, da utilizzare e quindi compilare.

`PATH`
Definisce il path in cui verrano storati gli eseguibili una volta compilati.

`CC`
Definisce quale compilatore usare per sorgenti scritte in C.

`CPP`
Definisce il percorso del preprocessore C.

`LIBS`
Definisce le librerie da passare al linker.

`CXX`
Definisce quale compilatore usare per sorgenti scritte in C++.

`LDFLAGS`
Definisce il percorso per le librerie non standard, tipo `libdaemon.c`, per il linker.

`CPPFLAGS`
Definisce il percorso per gli header non standard, tipo `libdaemon.h`, per il preprocessore C/C++.

`CXXFLAGS`
Definisce il percorso per gli header non standard, tipo `libdaemon.h`, per il compilatore C/C++.

`CFLAGS`
Definisce il percorso per gli header non standard, tipo `libdaemon.h`, per il compilatore C.

`PKG_CONFIG_PATH`
Definisce i path addizioni in cui pkg-config cercherà i file *.pc.

L'esatto contenuto del file è riportato in appendice.

Inoltre per la corretta compilazione di alcune librerie, altrimenti impossibili da compilare è stato necessario creare un altro file ausiliario denominato `config.sparc-leon-linux-gnu`, ovvero un file in cui vengono definite alcune proprietà dell'architettura SPARC.

L'esatto contenuto del file è riportato in appendice.

A.2.3 Procedura di Cross-Compilazione

Una volta creati tutti i file sopracitati si può procedere alla compilazione delle librerie e a quella di tutti i programmi utili al nostro scopo.
Per utilità viene riportato l'esempio di compilazione di IBR-DTN, attraverso l'immagine docker nota come `netg5_latest`.

```
$ docker start netg5_latest
$ docker attach netg5_latest
```

Il terminale così passa a controllare l'ambiente di sviluppo di Ångström

```
$ cd /mnt
$ source env.sh
$ cd /ibrdtn/ibrdtn
$ ./autogen.sh
$ CONFIG_SITE=/mnt/config.sparc-leon-linux-gnu ./configure
    --target=$TARGET --prefix="$PREFIX" --with-sysroot="$SYSROOT"
    CC="$CC" CPP="$CPP" LIBS="$LIBS" CXX="$CXX" LDFLAGS="$LDFLAGS"
    CPPFLAGS="$CPPFLAGS" CXXFLAGS="$CXXFLAGS" CFLAGS="$CFLAGS"
    --host=$HOST --enable-libmount=no --without-glib
    --without-vmime --without-curl --without-sqlite
    --without-lowpan --without-dtnsec --without-compression
    --without-tls --without-flooding --without-prophet
    --disable-dtndht --with-gnu-as --with-gnu-ld
$ make
$ make install
```

Dopo questa procedura otteniamo quindi in `/mnt/aether/sbin` l'eseguibile denominato `dtnd` che permette l'avvio di tutto l'applicativo. Grazie a tutte le ottimizzazioni in termini di snellimento si ottiene un file di 18.4 MB, il quale può essere eventualmente ridotto a 2 MB se lanciato il comando:

```
$ strip /mnt/aether/sbin/dtnd
```

Analizzando il file con il comando `file` avremo in output le informazioni della piattaforma di rifermento per eseguire il programma.

```
$ file dtnd
dtnd: ELF 32-bit MSB executable, SPARC, version 1 (GNU/Linux),
    dynamically linked, interpreter /lib/ld-linux.so.2, for
    GNU/Linux 2.6.32, stripped
```

Il ché ci conferma il successo della cross-compilazione.

Librerie, Tinyproxy e affini utilizzeranno lo stesso comando di `./configure` accompagnato dalle stesse variabili d'ambiente, tolte quelle proprie di IBR-DTN.

A.3 Caricamento su SD degli applicativi

A.3.1 Preparazione scheda SD

Come già sottolineato, la Net-G5 dispone di un sistema operativo Linux in sola lettura, quindi per caricare tutti gli applicativi è necessario passare per la scheda SD. Di default la SD non è montata con i privilegi di esecuzione e quindi è opportuno rimontarla con i dovuti privilegi.
Una volta copiato tutto l'albero delle cartelle dal proprio computer alla scheda SD e averla inserita nella Net-G5, bisogna riconoscere il nome con cui il sistema operativo rileva il file system, tramite il comando `fdisk -l` che ci lista tutte le partizioni attive e i loro nomi e il comando `findmnt` che ci permetterà di leggere il nome del mounting point. Una volta ottenute queste informazioni possiamo procedere a dare al SD i privilegi di esecuzione.

```
$ umount /tmp/mnt/a
$ mount -o umask=000 /dev/sda1 /tmp/mnt/a
```

Grazie a questa sequenza di comandi abbiamo guadagnato tutti i privilegi e per tutti i tipi di utente, permettendo così anche il trasferimento via FTP.

A.3.2 Caricamento via FTP

La Net-G5 ci permette l'utilizzo del protocollo FTP per l'invio di dati, perciò per evitare di ripetere l'operazione di `mount` ogni volta in cui servirà apportare modifiche ai file già presenti, fare un upload senza toccare la SD è caldamente consigliato. Il protocollo FTP ha bisogno di un indirizzo IP per poter funzionare, quindi dopo aver connesso il pc alla Net-G5 tramite cavo o tramite WI-FI, con il tramite `nmap` potremo ottenere questa informazione. Nel nostro caso l'indirizzo utilizzato è `192.168.4.10`. Quindi con la creazione di un semplice script bash, come quello riportato qui sotto, possiamo permettere in modo molto veloce l'upload di tutti gli eseguibili via FTP.

```
$ ftp -n 192.168.4.10
$ user topcon
$ password topcon
$ cd a/aether/bin
$ put tinyproxy
$ echo Command Completed
```

A questo punto se il file non è in uso e la SD ha correttamente ottunuto i privilegi di scrittura per tutti i gli utenti, ci verrà comunicato il successo dell'operazione di trasferimento.

A.4 Configurazione e avvio degli applicativi

IBR-DTN e Tinyproxy dispongono entrambi di un file di configurazione, che per permettere il corretto avvio degli applicativi, ai quali bisogna apportare ancora delle modifiche.

A.4.1 Configurazione di IBR-DTN

IBR-DTN è in grado di utilizzare molti layer di comunicazione prensenti sulla Net-G5, ma si è constatato dai log di errore dati al momento di avvio che lo stack IPV6 pur essendo presente non è del tutto completo cosìcchè IBR-DTN al tentativo di legarsi a questo vada in errore perciò, come prima cosa, bisogna eliminare il rifermento all'IPv6 dal discovery address, la quale utilizzerà solo un advertisement su IPv4. Successivamente bisognerà specificare le interfacce da usare per il collegamento. Purtroppo l'estensione per il Bluetooth è compatibile solo con il BT 4.0 LTE del quale la Net-G5 non dispone.

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

#
# configuration for a convergence layer named lan0
#
net_lan0_type = tcp   # we want to use tcp as protocol
net_lan0_interface = eth0  # listen on interface eth0
net_lan0_port = 4557   # with port 4557

#
# configuration for a convergence layer named lan1
#
net_lan1_type = tcp    # we want to use TCP as protocol
```

```
net_lan1_interface = wlan0  # listen on interface wlan0
net_lan1_port = 4556   # with port 4556 (default)
```

Durante l'utilizzo del comando `./configure` è stata richiesta la disattivazione dei protocolli di routing inutilizzati quali flooding e prophet. Sarà quindi opportuno configurare il file impostando il valore del routing ad `epidemic`.

Ultima ma molto importate è la time synchronization da disattivare, in quanto la NetG5, pur avendo la possibilità di ottenere la data dal servizio GPS, non garantisce in alcun modo che i satelliti siano sempre raggiungibili e quindi abbia sempre l'ora esatta, soprattuto se utilizzata in spazi chiusi.

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

Con questo file di configurazione, situato in `/tmp/mnt/a/etc/dtnd.conf`, IBR-DTN non causerà errori e riuscirà a fare il bind su tutti i layer di comunicazione possibili.

A.4.2 Configurazione di Tinyproxy

Tinyproxy necessità solo della configurazione creata ad hoc per l'utilizzo dei bundle presente nella cartella `/tmp/mnt/a/etc/tinyproxy/tinyproxy.conf`. In particolare Tinyproxy verrà posto accanto ad un web server e quindi sarà necessario attivare la possibilità di essere in ascolto per bundle diretti a quel particolare server. Questo comportamento si ottiene attivando la funzionalità serverside e configurando almeno un `app_name` attraverso il quale sarà possibile raggiungere il web-server.

```
# Configure the port and the IP address of the IBR-DTN daemon
    Server-Side.
# Defaults are IP = 127.0.0.1 and Port = 4550.
# These parameter permits, if compiled, to use IBR-DTN to communicate
    over BUNDLE PROTOCOL.
#
dtndServerSide Yes
dtndServerPort 4550
dtndServerIpAddress 127.0.0.1

# If you have more than 1 application, you can set over IBR-DTN as
    much endpoint as you want.
# You only have to specify an unique app name, an Ip address and a
    port.
```

```
# Default one is web 80 127.0.0.1
#
dtnAppName "websvr" 80 127.0.0.1
dtnAppName "websvr2" 82 127.0.0.1
```

Con questa configurazione l'applicativo farà il bind sull'API server di IBR-DTN all'indirizzo `127.0.0.1:4550` e rimarrà in ascolto per gli `app_name`: *websvr* e *websvr2* presenti sul proprio localhost, rispettivamente sulle porte 80 e 82.

Nota: il web server sulla porta 82 non esiste; è stato riportato come esempio per permettere al lettore di capire come poter configurare correttamente Tinyproxy.

A.4.3 Avvio

Con la corretta modifica e configurazione di IBR-DTN e di Tinyproxy è possibile procedere all'avvio e a rendere disponibile il servizio web tramite bundle. Una volta collegati in telnet e con i tutti i privilegi sbloccati, basta far avviare un semplice script posizionato in `/tmp/mnt/a` per avviare il entrambi.

File: start.sh

```
#!/bin/bash
echo Starting IBR-DTN
./aether/sbin/dtnd -c ./aether/etc/ibrdtnd.conf &

sleep 5
clear
echo Starting Tinyproxy
./aether/bin/tinyproxy -d -c ./aether/etc/tinyproxy/tinyproxy.conf
```

Leggendo l'output prodotto su console si può facilemente constatare che i due sono in attesa di qualche bundle da processare.

```
    Starting IBR-DTN
Thu Feb 21 21:57:24 2019 INFO NativeDaemon: IBR-DTN daemon 1.0.1 (build
    44746a46)
Thu Feb 21 21:57:24 2019 INFO Configuration: Configuration:
    ./aether/etc/ibrdtnd.conf
Thu Feb 21 21:57:24 2019 INFO NativeDaemon: use logfile for output:
    /var/log/ibrdtn/ibrdtn.log
Thu Feb 21 21:57:24 2019 INFO BundleCore: Local node name:
    dtn://netgfive36813
Thu Feb 21 21:57:24 2019 INFO BundleCore: Forwarding of bundles enabled.
Thu Feb 21 21:57:24 2019 INFO BundleCore: Non-singleton bundles are
    accepted.
```

```
Thu Feb 21 21:57:24 2019 INFO NativeDaemon: using bundle storage in
    memory-only mode
Thu Feb 21 21:57:24 2019 INFO NativeDaemon: API initialized using tcp
    socket: <loopback>:4550
Thu Feb 21 21:57:24 2019 INFO NativeDaemon: TCP ConvergenceLayer added
    on wlan0:4556
Thu Feb 21 21:57:24 2019 INFO IPNDAgent: listen to [224.0.0.142]:4551
Thu Feb 21 21:57:24 2019 INFO IPNDAgent: advertise on interface wlan0
Thu Feb 21 21:57:24 2019 INFO NativeDaemon: Using epidemic routing
    extensions
Thu Feb 21 21:57:24 2019 INFO EpidemicRoutingExtension: Initializing
    epidemic routing module

    Starting Tinyproxy
INFO Feb 21 21:57:35 [1919]: Initializing tinyproxy ...
INFO Feb 21 21:57:35 [1919]: Reloading config file
INFO Feb 21 21:57:35 [1919]: Setting "Via" header to 'tinyproxy'
INFO Feb 21 21:57:35 [1919]: IBR-DTN Configured IP is: 127.0.0.1
    client-side
INFO Feb 21 21:57:35 [1919]: IBR-DTN Configured IP is: 127.0.0.1
    server-side
INFO Feb 21 21:57:35 [1919]: Added web application on port 80 with
    ip_address 127.0.0.1.
CONNECT Feb 21 21:57:35 [1919]: Connected to IBR-DTN
INFO Feb 21 21:57:35 [1919]: IBR-DTN says: IBR-DTN 1.0.1 (build
    44746a46) API 1.0.1
INFO Feb 21 21:57:35 [1919]: Response: 200 SWITCHED TO EXTENDED
INFO Feb 21 21:57:35 [1919]: Response: 200 OK
INFO Feb 21 21:57:35 [1919]: Response: 200 ENDPOINT GET
    dtn://netgfive36813/websvr
INFO Feb 21 21:57:35 [1919]: Listening for application web
    127.0.0.1::80 dtn://netgfive36813/websvr
INFO Feb 21 21:57:35 [1919]: Added web application on port 82 with
    ip_address 127.0.0.1.
CONNECT Feb 21 21:57:35 [1919]: Connected to IBR-DTN
INFO Feb 21 21:57:35 [1919]: IBR-DTN says: IBR-DTN 1.0.1 (build
    44746a46) API 1.0.1
INFO Feb 21 21:57:35 [1919]: Response: 200 SWITCHED TO EXTENDED
INFO Feb 21 21:57:35 [1919]: Response: 200 OK
INFO Feb 21 21:57:35 [1919]: Response: 200 ENDPOINT GET
    dtn://netgfive36813/websvr2
INFO Feb 21 21:57:35 [1919]: Listening for application web
    127.0.0.1::82 dtn://netgfive36813/websvr2
INFO Feb 21 21:57:35 [1919]: listen_sock called with addr = '(NULL)'
INFO Feb 21 21:57:35 [1919]: trying to listen on host[0.0.0.0],
    family[2], socktype[1], proto[6]
```

```
INFO Feb 21 21:57:35 [1919]: listening on fd [5]
INFO Feb 21 21:57:35 [1919]: trying to listen on host[::], family[10],
   socktype[1], proto[6]
INFO Feb 21 21:57:35 [1919]: listening on fd [6]
INFO Feb 21 21:57:35 [1919]: Now running as group "nobody".
INFO Feb 21 21:57:35 [1919]: Now running as user "nobody".
INFO Feb 21 21:57:35 [1919]: Creating child number 1 of 1 ...
INFO Feb 21 21:57:35 [1919]: Finished creating all children.
INFO Feb 21 21:57:35 [1919]: Setting the various signals.
INFO Feb 21 21:57:35 [1919]: Starting main loop. Accepting connections.
```

Appendice B

Dettagli Codice

B.0.1 Contenuto file env.sh

```
export TARGET=sparc-continuous-linux-gnu
export PREFIX=/mnt/aether
export SYSROOT=/opt/sysroot-sparc-leon-linux-gnu
export PATH=$PATH:$PREFIX/bin
export CC=/opt/sparc-leon-linux-gnu/bin/sparc-leon-linux-gnu-gcc
export CPP=/opt/sparc-leon-linux-gnu/bin/sparc-leon-linux-gnu-cpp
export LIBS=/opt/sparc-leon-linux-gnu/lib/gcc/
       sparc-continuous-linux-gnu/6.4.0/libgcc.a
export CXX=/opt/sparc-leon-linux-gnu/bin/sparc-leon-linux-gnu-g++
export LDFLAGS="-Os -L/mnt/aether/lib"
export CPPFLAGS="-Os -I/mnt/aether/include"
export CXXFLAGS="-Os -I/mnt/aether/include"
export CFLAGS="-Os -I/mnt/aether/include"
export PKG_CONFIG_PATH="/mnt/aether/lib/pkgconfig"
export HOST=sparc-continuous-linux-gnu
export BUILD=x86_64-pc-linux-gnu

mkdir -p $PREFIX
```

B.0.2 Contenuto file config.sparc-leon-linux-gnu

```
glib_cv_stack_grows=no
glib_cv_uscore=yes
AUTOCONFIG_POSTFIX_EXTRAS+=\
              ac_cv_func_getpgrp_void=no \
              ac_cv_func_setpgrp_void=yes \
              ac_cv_func_memcmp_working=yes \
```

```
                rb_cv_binary_elf=no \
                rb_cv_negative_time_t=no
```

B.0.3 Dettaglio struttura dati Dtnd_bundle_id

```
typedef struct Dtnd_bundle_id{
       size_t timestamp; /*timestamp of the received bundle*/
       size_t sequence_n; /*sequence number of the received bundle*/
       struct Dtnd_bundle_id *next; /*pointer to scan the received
          bundle notification list*/
} Dtnd_bundle_id;
```

B.0.4 Dettaglio struttura dati Dtn_service

```
typedef struct dtn_service{
       char *ip_address; /*real IP address of the web server*/
       uint16_t port; /*real TCP port of the web server*/
       char *app_name; /*configured application name*/
       char *app_eid; /*obtained completed EID*/
} Dtn_service;
```

B.0.5 Dettaglio funzione bundle_wait_bundle_on_keep_alive_connection

```
int bundle_wait_bundle_on_keep_alive_connection(int fd) {
       fd_set read_fds;
       int fdmax = fd;
       int socket_ready;
       struct timeval timeout;
       Dtnd_bundle_id *db = NULL;

       bundle_verify_status(fd, EAPI_STATUS_READY);

    /* Did we already received new incoming bundles (asinchronously)? */
       if(!bundle_is_bundle_id_queue_empty(fd))
              return 0;

       while (1) {
              FD_ZERO(&read_fds);
              FD_SET(fd, &read_fds);
              timeout.tv_sec = config.headerlinetimeout;
              timeout.tv_usec = 0;
```

```
                socket_ready = select(fdmax + 1, &read_fds, NULL, NULL,
                    &timeout);

                if (socket_ready == 0){
                        log_message(LOG_WARNING, "Timeout for bundle
                            waiting expired, No new bundle incoming");
                        return -1;
                }

                if (socket_ready == -1) {
                        if (errno == EINTR)
                                continue;
                        return -1;
                }
                else {
                        db = bundle_react_to_notify(fd);
                        if(db == NULL){
                                log_message(LOG_CRIT,"Fail while reading
                                    602 notification");
                                abort();
                        }
                        bundle_enqueue_bundle_id(fd,db);
                        return 0;
        }
    }
}
```

B.0.6 Dettaglio funzione bundle_read_header_incoming_bundle

```
/*
* Reads only the header of a new incoming bundle
*/
int bundle_read_header_incoming_bundle(int fd) {
        Dtnd_bundle_id* db;
        char *buffer = NULL;
        char command[MAX_EID_LENGTH + 256];
        size_t len;
        int raw_flag;

        bundle_verify_status(fd, EAPI_STATUS_READY);
        db = bundle_get_bundle_id(fd);
        snprintf(command, sizeof(command), "bundle load %zu %zu %s\n",
                                        db->timestamp, db->sequence_n,
                                            bundle_get_remote_eid(fd));
```

```
        if (bundle_send_command(command, fd) == -1)
                return -1;
        if (bundle_send_command("set encoding raw\n", fd) == -1)
                return -1;
        if (bundle_send_command("payload get\n", fd) == -1)
                return -1;

        bundle_set_status(fd, EAPI_STATUS_PAYLOAD_READ);

        while (1) {
                readline_internal(fd, &buffer, CHUNCK);

                if (sscanf(buffer, "Length: %zu", &len) == 1)
                        bundle_set_payload_length(fd, len);

                if (strcmp(buffer, "Encoding: raw\n") == 0)
                        raw_flag = 1;

                if (strncmp(buffer, "\n", 1) == 0) { /* last header line
                    */
                        safefree(buffer);
                        break;
                }
                safefree(buffer);
        }

        if (raw_flag == 0) {
                log_message(LOG_ERR, "Encoding is not set to raw");
                return -1;
        }

        return 0;
}
```

B.0.7 Dettaglio funzione bundle_rewrite_host

```
char* bundle_rewrite_host(char* line){
    char *host;
    char *eid;
    int i;
    host = safestrdup(strchr(line,' ')+1),
    log_message(LOG_INFO,"host: %s", host);
```

```
    eid = bundle_format_server_eid(host);
    safefree(host);

    if(eid == NULL)
        return NULL;

    for(i=0; i<MAX_FD; i++){
        if(services[i] == NULL)
            continue;

        if(strcmp(eid, services[i]->app_eid) == 0){
            char c[strlen(services[i]->ip_address)+16];
            sprintf(c,"Host: %s:%d",services[i]->ip_address,
                services[i]->port );
            safefree(eid);
            return safestrdup(c);
        }
    }
    return NULL;
}
```

B.0.8 Dettaglio funzione bundle_host_is_dtn

```
int bundle_host_is_dtn(char *host) {

        if(strncmp(host+strlen(host)-4,".dtn", 4))
                return 0;

        return 1;
}
```

B.0.9 Dettaglio funzione establish_http_connection

```
    static int
establish_http_connection (struct conn_s *connptr, struct request_s
    *request)
{
        char portbuff[7];
        char dst[sizeof(struct in6_addr)];

        /* Build a port string if it's not a standard port */
        if (request->port != HTTP_PORT && request->port !=
            HTTP_PORT_SSL)
```

```
                snprintf (portbuff, 7, ":%u", request->port);
        else
                portbuff[0] = '\0';

/*If we need to use bundle protocol only this code is usefull*/
#ifdef BUNDLE_PROXY

        if(bundle_is_fd_dtnd(connptr->server_fd)){
                bundle_verify_status(connptr->server_fd,
                    EAPI_STATUS_PAYLOAD_WRITE);

                log_message(LOG_INFO,"Sending the request header for
                    %s", request->path);

                return write_message (connptr->server_fd,
                                "%s %s HTTP/1.1\r\n"
                                "Host: localhost\r\n",
                                        request->method,
                                        request->path
                                        );
        }

        if(bundle_is_fd_dtnd(connptr->client_fd)){

            log_message(LOG_INFO,"Sending the request header for %s",
                request->path);
                return write_message (connptr->server_fd,
                                        "%s %s HTTP/1.1\r\n"
                                        "Host: localhost\r\n",
                                                request->method,
                                                request->path
                                                );
        }
#endif
/*else try a normal HTTP/1.1 connection
                if (inet_pton(AF_INET6, request->host, dst) > 0) {
                        /* host is an IPv6 address literal, so surround
                            it with
                               ... /*IPv6 case, never used*/
                    } else if(connptr->upstream_proxy &&
                            connptr->upstream_proxy->type == PT_HTTP &&
                            connptr->upstream_proxy->ua.authstr){
                            ... /*With upstream proxy case, never used*/
                        } else {
                                return write_message (connptr->server_fd,
                                    "%s %s HTTP/1.1\r\n"
```

```
                                    "Host: %s%s\r\n"
                                    /*"Connection: close\r\n"*/,
                                    request->method, request->path,
                                    request->host, portbuff);

                        }


}
```

Bibliografia

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

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

[3] S. Schildt et al. «IBR-DTN: A lightweight, modular and highly portable Bundle Protocol implementation». In: *Electronic Communications of the EASST, Volume 37: Kommunikation in Verteilten Systemen 2011* (2011). DOI: http://dx.doi.org/10.14279/tuj.eceasst.37.512.544.

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

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

[6] M. Demmer, J. Ott e S. Perreault. *Delay-Tolerant Networking TCP Convergence-Layer Protocol*. RFC 7242. IETF, 2014. URL: https://tools.ietf.org/html/rfc7242.

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

[8] H. Chenji et al. «A Wireless Sensor, AdHoc and Delay Tolerant Network System for Disaster Response». In: (2011). URL: https://s3.amazonaws.com/academia.edu.documents/42807286/A_Wireless_Sensor_AdHoc_and_Delay_Tolera20160218-10104-1mw070n.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1554157014&Signature=pBo8jvdzK3XA14NGGnaFprOsBFU%3D&response-content-disposition=inline%3B%20filename%3DA_Wireless_Sensor_AdHoc_and_Delay_Tolera.pdf.

[9] Zeroconf Working Group. *Zero Configuration Networking (Zeroconf)*. URL: http://www.zeroconf.org/.

[10] *RFC7230*. URL: https://tools.ietf.org/html/rfc7230.

[11] *HTTP over Bundle Protocol for Delay-Tolerant Networks (DTN)*. URL: https://tools.ietf.org/id/draft-perreault-dtnrg-http-00.html.

[12] *RFC6455*. URL: https://tools.ietf.org/html/rfc6455.

[13] *Satellite navigation*. URL: https://en.wikipedia.org/wiki/Satellite_navigation.

[14] *Topcon NetG5 Operator manual*. URL: http://topconcare.com/files/5914/4406/8879/1004636-01-RVC_Net-G5_OM_Secured.pdf.

[15] *Topcon NetG5 specifications*. URL: https://www.topconpositioning.com/sites/default/files/product_files/net-g5_broch_7010_2145_revc_sm.pdf.

[16] *Open Angstrom*. URL: https://en.wikipedia.org/wiki/ÃĚngstrÃűm_distribution.

[17] *Official Docker site*. URL: https://www.docker.com.

[18] *Tinyproxy Official Site*. URL: https://tinyproxy.github.io.

[19] *Autoconf Official Site*. URL: https://www.gnu.org/software/autoconf/.

[20] *Distributed hash table*. URL: https://en.wikipedia.org/wiki/Distributed_hash_table.

[21] *Libdaemon Official site*. URL: http://0pointer.de/lennart/projects/libdaemon/.

[22] *Toolchains manual*. URL: https://elinux.org/Toolchains.