POLITECNICO DI TORINO

Corso di laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

Scenari applicativi nell'ambito del paradigma Fog Computing in presenza di reti partizionate e non affidabili

Relatore
prof. Fulvio Risso

Candidato
Luigi Maio

Dicembre 2017

Alla mia famiglia.

Indice

Elenco delle figure v

1 Introduzione 1

2 Scenario 4
2.1 Comunicazione in presenza di reti partizionate e non affidabili 4
2.2 Applicazioni IIoT basate su architettura Fog Computing in contesti sfidanti per la connettività 6
2.3 Obiettivo della tesi 8

3 Il Fog computing 10
3.1 Definizione 10
3.2 Casi d'uso 12
3.2.1 Sistema per il controllo intelligente del traffico 12
3.2.2 Consegna dei pacchi tramite droni 12
3.2.3 Videosorveglianza 13
3.3 La tecnologia di Nebbiolo Tecnologies 14
3.3.1 Cenni sull'azienda 14
3.3.2 Architettura 14
3.3.3 Configurazione del fogNode™ 19

4 Delay/Disruption-Tolerant Networking 23
4.1 Introduzione 23
4.2 Architettura 24
4.2.1 Endpoint Identifiers (EIDs) 26
4.2.2 Affidabilità e Trasferimento di Custodia 27
4.3 Routing 28
4.3.1 Disponibilità delle informazioni topologiche 28
4.3.2 Principali algoritmi di routing per DTN 29
4.4 Formato di un Bundle 32
4.4.1 Primary Block 32
4.4.2 Blocchi di payload 34

4.5 IBR-DTN . . . 34
4.5.1 Installazione da sorgenti, configurazione ed avvio . . . 35

5 Altre tecnologie utilizzate 38
5.1 Docker . . . 38
5.1.1 Container . . . 38
5.1.2 Networking . . . 40
5.2 RabbitMQ . . . 41
5.2.1 Tipologie di Exchange . . . 41
5.2.2 Sistema di gestione . . . 43
5.3 Reti wireless peer-to-peer . . . 45
5.3.1 Wi-Fi Direct . . . 45
5.3.2 IEEE 802.11 IBSS (Ad-hoc) . . . 47
5.3.3 Confronto tra Wi-Fi Direct e modalità Ad-hoc . . . 48
5.4 L'orchestratore Universal Node . . . 49
5.4.1 Il network controller . . . 50
5.4.2 Il compute controller . . . 51

6 Architettura del sistema di comunicazione 52
6.1 Soluzione proposta . . . 52
6.2 Architettura generale . . . 53
6.2.1 Comunicazione in upstream . . . 53
6.2.2 Comunicazione in downstream . . . 55
6.3 Proof-of-Concept: Sistema di telemetria . . . 57

7 Implementazione del sistema di telemetria 59
7.1 Sensing Edge Device . . . 59
7.1.1 Configurazione di rete . . . 60
7.1.2 Configurazione del Bundle Protocol Agent IBR-DTN . . . 61
7.1.3 Connessione opportunistica tra dispositivi . . . 64
7.1.4 MQTT Publisher . . . 67
7.1.5 Gateway MQTT-to-DTN . . . 69
7.2 Nebbiolo fogNode . . . 72
7.2.1 Configurazione RabbitMQ . . . 73
7.2.2 Configurazione dello stack ELK . . . 74
7.2.3 Configurazione del Bundle Protocol Agent IBR-DTN . . . 75
7.2.4 Gateway DTN-to-MQTT . . . 76
7.2.5 Data Filter . . . 80
7.3 Cloud . . . 82
7.4 Sincronizzazione del tempo . . . 83

8 Validazione 85
8.1 Test con dispositivi fisici 85
8.1.1 Banco prova 85
8.1.2 Connessione opportunistica dei dispositivi 86
8.2 Test con sistema di orchestrazione Universal Node 88
8.2.1 Ambiente di simulazione 88
8.2.2 Script di simulazione 89
8.2.3 Tempo medio di consegna di un bundle 91
8.2.4 Spazio occupato su ciascun nodo 93
8.2.5 Latenza in caso di topologia connessa 94

9 Conclusioni e sviluppi futuri 96

Elenco delle figure

1.1 Applicazioni IIoT in presenza di reti partizionate e non affidabili: le macchine operatrici sfruttano i contatti occasionali per far giungere l'informazione al nodo Fog. 2

2.1 Paradigma Store-Carry-and-Forward (Fonte: Interplanetary Internet Special Interest Group [4]) 5
2.2 Le macchine forestali in campo, i Bus, i Minivan, le officine (Workshop) e i mezzi di servizio (Service Wagon) formano una "*Wireless Mesh Network*" 7

3.1 Architettura Fog Computing (Fonte: OpenFog Consortium [6]) 11
3.2 Architettura della piattaforma Fog di Nebbiolo Technologies. 15
3.3 Diverse configurazioni di fogNode™ (Fonte: Nebbiolo.tech [10]). . . . 16
3.4 Lo stack che compone il fogOS™ (Fonte: Nebbiolo.tech [12]). 17
3.5 Screenshot dell'interfaccia web del fogSM™ - sezione "inventario". . . 18
3.6 Esempio di provisioning di un'applicazione (Fonte: Nebbiolo Technologies). 20
3.7 Architettura di rete e configurazione interna del fogLet (Fonte: Nebbiolo Technologies) 21

4.1 Stack di rete di un nodo DTN 24
4.2 Un esempio di nodo DTN mostra come questo deve interagire con le diverse componenti che ne costituiscono l'architettura (Fonte: [15]) . 25
4.3 Trasferimento di custodia di un bundle in una DTN (Fonte: [4]) . . . 27
4.4 Formato del "*primary block*" di un Bundle (Fonte: [15]) 33
4.5 Formato di un blocco di payload all'interno di un Bundle 34

5.1 Differenze tra container e virtual machine (Fonte: Docker [25]). 39
5.2 Flusso di messaggi in RabbitMQ (Fonte: CloudAMQP [28]). 42
5.3 Architettura di alto livello dello Universal Node (Fonte: Universal Node public repository [31]). 50

6.1 Esempio di comunicazione tra nodi non direttamente connessi: una coppia di gateway con funzioni speculari permette ai messaggi MQTT di essere veicolati attraverso la DTN 53
6.2 Architettura del sistema di comunicazione: Upstream 54
6.3 Architettura del sistema di comunicazione: Downstream 55
6.4 Overview del sistema di telemetria realizzato come Proof-Of-Concept dell'architettura di comunicazione proposta 57

7.1 Prototipi di Sensing Edge Device: i due LED restituiscono un feedback visivo sullo stato di connessione ad un dispositivo vicino o al fogNode 65
7.2 Screenshot dell'interfaccia di Kibana in esecuzione sul fogNode 74
7.3 Screenshot dell'interfaccia di Kibana in esecuzione sul Cloud 83

8.1 Cattura del traffico tra due Sensing Edge Device: i due dispositivi effettuano dapprima il processo di IBSS merge per stabilire il link fisico e successivamente scambiano i beacon IPND per annunciarsi a vicenda come nodi della DTN 86
8.2 Cattura del traffico tra due Sensing Edge Device: i due dispositivi iniziano lo scambio dei bundle successivamente alla fase di discovery del protocollo IPND 87
8.3 Tempo medio di consegna di un bundle 91
8.4 Distribuzione di frequenza cumulata dei tempi di consegna di un bundle 92
8.5 Numero medio di bundle memorizzati su ciascun nodo durante la simulazione 93
8.6 Latenzia media a confronto in caso di topologia connessa: DTN vs. MQTT 94

Capitolo 1

Introduzione

Negli ultimi anni è sempre più crescente l'attenzione del mercato tecnologico verso l'*Internet Of Things*: dai moderni elettrodomestici ai sensori per la domotica, dagli smartphone ai dispositivi indossabili, dai veicoli stradali alle macchine agricole, fino al mondo dell'automazione industriale, il numero di dispositivi connessi sta crescendo vertiginosamente. Secondo un report del 2015 stilato dalla società *McKinsey&Company* [1], i dispositivi connessi entro il 2025 saranno circa 50 miliardi. Stime che trovano riscontro anche in studi più recenti come quello pubblicato da *Gartner, Inc.* nel Gennaio 2017 [2], che prevede circa 20,4 miliardi di dispositivi entro il 2020.

Questi numeri esorbitanti sono in gran parte dovuti all'enorme valore dei dati che questi dispositivi producono e che rappresentato una sorgente preziosa di informazioni. Il miglioramento dei processi produttivi nel campo industriale, agricolo, delle costruzioni e dell'energia o il miglioramento della qualità della vita nei grandi centri urbani sono solo alcuni dei benefici che possono derivare dall'analisi di queste grandi moli di dati. Oggi il paradigma del *Cloud Computing*, che fornisce in modo elastico risorse computazionali e di immagazzinamento di dati alle applicazioni, è quello prevalentemente utilizzato nell'ambito dell'IoT. Gli end-points sono per lo più collettori di dati o centraline che controllano attuatori, collegati per mezzo di un *gateway IoT* ad un qualche servizio sul Cloud dove viene implementata la "*business logic*" dell'applicazione.

Nonostante questo modello garantisca scalabilità e risorse potenzialmente illimitate alle applicazioni, esistono alcuni domini applicativi per cui un'architettura Cloud non è sufficiente a soddisfarne i requisiti. Per varie applicazioni risultano fondamentali caratteristiche come basse latenze di comunicazione, il supporto alla mobilità, la geo-distribuzione delle risorse e la cosiddetta "*location-awareness*" delle applicazioni stesse. Per tale motivo sta destando sempre più interesse un nuovo paradigma chiamato *Fog Computing*, che si prefigge di sopperire a tali lacune andando a distribuire risorse e servizi lungo l'infrastruttura che connette i dispositivi finali al Cloud. Lo scopo di chi sta proponendo questa nuova tecnologia, quindi, non è

quello di soppiantare il Cloud, ma di estenderlo per fornire una soluzione efficace alle attuali limitazioni.

Tuttavia, non tutti i domini applicativi hanno contemporaneamente bisogno di tutte le potenziali caratteristiche che un'architettura basata sul Fog Computing può fornire. In alcuni particolari contesti operativi dell'*IIoT – Industrial Internet of Things* – è difficile garantire comunicazioni stabili tra dispositivi finali, ovvero le macchine operatrici in campo, e nodi di elaborazione. Le operazioni vengono svolte in ambienti sfidanti per le comunicazioni su canali wireless, come nel caso di operazioni di estrazione in miniera o cantieri molto estesi in zone remote. In tali contesti le macchine operatrici possono trascorrere in campo lunghi periodi senza possibilità di comunicazione, corrispondenti a volte al loro intero ciclo di operatività; spesso è solo al momento del rientro alla base che hanno la possibilità di "scaricare" i dati raccolti e prelevare eventuali aggiornamenti. In un tale scenario non è strettamente necessario che il flusso di dati sia real-time, ma sarebbe utile che questi si propagassero con tempistiche accettabili per un efficiente svolgimento delle operazioni in campo sfruttando la geo-distribuzione e la mobilità delle macchine operatrici. Ne sono un esempio la raccolta di dati verso i nodi Fog ed il Cloud per applicazioni di telemetria e di "*batch processing*", l'invio alle macchine operatrici di informazioni di contesto per migliorarne l'operatività o la propagazione a queste ultime di aggiornamenti software.

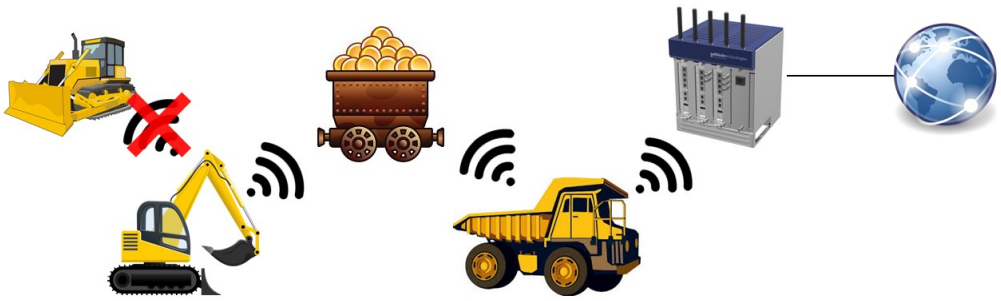


Figura 1.1. Applicazioni IIoT in presenza di reti partizionate e non affidabili: le macchine operatrici sfruttano i contatti occasionali per far giungere l'informazione al nodo Fog.

Questa tesi si propone di analizzare le criticità derivanti dalle particolari condizioni operative in tali scenari. Verrà quindi proposta una soluzione architetturale, che tragga vantaggio dai contatti occasionali delle macchine operatrici durante i loro spostamenti in campo, per propagare l'informazione verso i nodi Fog ed il Cloud e,

viceversa, da un nodo Fog ai mezzi mobili (Figura 1.1). In seguito l'architettura proposta verrà parzialmente implementata ed utilizzata per sviluppare un prototipo di un sistema di telemetria. Per la realizzazione di tale prototipo, saranno utilizzati due Raspberry Pi in qualità di collettori di misure da inoltrare verso un nodo Fog. La controparte verrà implementata sul cosiddetto *fogNode*^{TM}, che *Nebbiolo Technologies* ha messo a disposizione del gruppo di ricerca *Netgroup* del Politecnico di Torino. Infine verrà condotta una validazione preliminare dell'architettura proposta, sia mediante dei test effettuati con i prototipi realizzati, sia mediante una simulazione in ambiente virtualizzato per mezzo del sistema di orchestrazione *Universal Node*.

Questo elaborato è strutturato come segue:

- **Capitolo 2:** espone il problema che questa tesi si propone di risolvere, descrivendone lo scenario in cui si colloca e delineandone gli obiettivi.
- **Capitolo 3:** presenta il paradigma del "*Fog Computing*" e la piattaforma realizzata da *Nebbiolo Technologies* utilizzata per lo sviluppo di questa tesi.
- **Capitolo 4:** presenta le "*Delay/Disruption Tolerant Networks*" ed il *Bundle Protocol* e ne introduce l'implementazione software IBR-DTN, utilizzata per lo sviluppo di questa tesi.
- **Capitolo 5:** descrive gli ulteriori applicativi e le ulteriori tecnologie impiegate per la realizzazione del prototipo, mostrando le loro funzionalità e le loro caratteristiche principali.
- **Capitolo 6:** presenta l'architettura del sistema di comunicazione proposto per affrontare le criticità descritte nel Capitolo 2.
- **Capitolo 7:** descrive l'implementazione del prototipo di un sistema di telemetria, realizzato sfruttando l'architettura proposta nel capitolo precedente e la piattaforma di Fog Computing di Nebbiolo Technologies.
- **Capitolo 8:** fornisce una panoramica dei risultati ricavati misurando le performance del sistema, sopratutto in termini di tempo di consegna dei messaggi.
- **Capitolo 9:** espone le conclusioni e presenta alcuni progetti futuri che avranno come base il lavoro svolto in questa tesi.

Capitolo 2

Scenario

Nella prima parte di questo capitolo viene esposto, con carattere generale, il problema della comunicazione nel caso di reti partizionate e non affidabili. Il problema viene successivamente contestualizzato ad un particolare scenario applicativo nell'ambito del Fog Computing. Infine viene delineato l'obiettivo della tesi in base alle considerazioni esposte.

2.1 Comunicazione in presenza di reti partizionate e non affidabili

La comunicazione su Internet si basa sulla commutazione di pacchetto. Ogni pacchetto, che rappresenta un frammento di informazione, viaggia in modo del tutto indipendente da un nodo sorgente a un nodo destinazione attraverso l'enorme rete di router che compone l'infrastruttura di Internet. Nel caso di un collegamento interrotto o congestionato, i router possono inoltrare ciascun pacchetto verso un next-hop alternativo. I pacchetti che compongono un dato messaggio, quindi, potrebbero arrivare a destinazione fuori sequenza o alcuni di questi perdersi.

Un protocollo di trasporto affidabile come TCP provvede a ritrasmettere i pacchetti, in caso di presunte perdite, e a consegnarli ai protocolli di livello superiore nell'ordine corretto. L'usabilità di Internet dipende, però, da alcune ipotesi piuttosto forti riguardanti l'architettura di rete:

- Per tutta la durata della comunicazione deve esistere almeno un percorso End-to-End tra sorgente e destinazione, in entrambe le direzioni di trasmissione.
- Ritardi piccoli, nell'ordine dei millisecondi, e consistenti tra invio di pacchetti e ricezione dei corrispondenti "*acknowledgements*".
- Errori di trasmissione e perdite di pacchetti relativamente bassi.

Sono dette "*challenged networks*" [3] quelle reti per cui almeno una di queste ipotesi non è soddisfatta. Un tipico scenario caratterizzato da ritardi di trasmissione, tassi di errore e di perdita molto elevati è quello della comunicazione interplanetaria. Un'altro contesto particolarmente interessante è quello delle reti di sensori, dove un requisito fondamentale è il risparmio energetico dei dispositivi; sono reti, pertanto, caratterizzate da trasmissioni sporadiche ed intermittenti.

In questi e molti altri scenari applicativi, inoltre, i nodi sono spesso mobili e collegati per mezzo di canali wireless. A causa della mobilità dei nodi, possono interporsi ostacoli che impediscano la comunicazione o alcuni di essi allontanarsi eccessivamente; di conseguenza la topologia della rete cambia continuamente nel tempo. Quando un percorso End-to-End tra sorgente e destinazione viene a mancare, si verifica quello che viene chiamato un "*partizionamento*" della rete.

Con i tradizionali algoritmi di routing, i pacchetti che non possono essere instradati immediatamente dai router vengono scartati. Il protocollo TCP proverebbe a ritrasmetterli utilizzando dei tempi di ritrasmissione sempre più aggressivi. Quando la perdita di pacchetti diventa troppo severa, il TCP termina la sessione e ciò potrebbe portare un'applicazione a fallire. Connessioni intermittenti e partizionamenti della rete, ritardi elevati e frequenti errori di trasmissione rendono, quindi, impraticabile l'uso dei protocolli comunemente adottati per le comunicazioni su Internet.

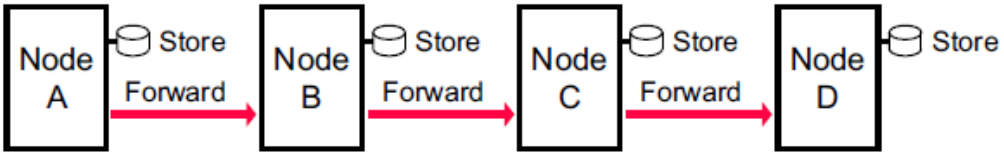


Figura 2.1. Paradigma Store-Carry-and-Forward (Fonte: Interplanetary Internet Special Interest Group [4])

Il paradigma "*store-carry-and-forward*" può essere utilizzato per far fronte a queste problematiche. Si tratta di un metodo concettualmente simile a quello del sistema di consegna postale. In una tale rete, i nodi non scartano immediatamente i pacchetti nel caso in cui non sia possibile inoltrarli verso la destinazione. Invece, gli algoritmi di routing vengono impiegati per valutare se in futuro potrebbe esserci un'opportunità di inoltrare questi pacchetti verso la destinazione. Se questa valutazione ha esito positivo, i pacchetti vengono mantenuti dal nodo finché non diventa possibile l'inoltro ad un'altro nodo verso la destinazione, anch'esso in grado di conservare il messaggio fino all'inoltro successivo (Figura 2.1). Sfruttando, quindi, la

mobilità dei nodi e gli occasionali contatti tra essi, è possibile inoltrare i pacchetti verso la destinazione anche in caso di assenza continuata di un percorso End-to-End tra sorgente e destinazione.

Le cosidette *Delay/Disruption-Tolerant Networks* (DTN) implementano questa logica di inoltro inserendo un nuovo strato – il *Bundle Protocol* [5] – tra livello applicativo e livello trasporto. L'architettura di rete, i protocolli e le logiche di inoltro delle reti DTN vengono approfondite nel Capitolo 4.

2.2 Applicazioni IIoT basate su architettura Fog Computing in contesti sfidanti per la connettività

Il contesto da cui prende spunto questa tesi è quello del *IIoT – Industrial Internet of Things* – in ambienti particolarmente sfidanti per la connettività. Il partner industriale con il quale è stato sviluppato questo lavoro di tesi – *Tierra Telematics* – ci ha fornito alcuni esempi di sistemi telematici sviluppati in tali contesti:

- Semina e mietitura nei campi di grano del Nord-America, estesi per centinaia di chilometri, effettuate per mezzo di grandi macchine agricole.
- Operazioni di estrazione in miniera.
- Operazioni di estrazione di "*petrolio di scisto*" (o *shale oil*) da giacimenti di scisto bituminoso.
- Operazioni forestali.

Per l'ultimo caso citato, ci è stato illustrato l'esempio di un sistema per la raccolta di informazioni e per la gestione di una flotta di macchine forestali. Si riporta di seguito una descrizione della possibile infrastruttura di rete per la comunicazione tra i mezzi in campo ed il sistema stesso, i cui servizi ed applicazioni si considerano istanziati su una Server Farm remota.

In Figura 2.2 sono raffigurati tutti gli elementi – *peers* – che partecipano alla grande rete magliata che permette la comunicazione all'interno del sistema. Le macchine forestali (*Harvester*, *Forwarder*, *Excavator*) sono equipaggiate con un apparato wireless da esterno. Questi dispositivi sono installati anche nei Bus e Minivan, che trasportano il personale, nelle officine e nei mezzi di servizio sparsi nella foresta. Ognuno di questi dispositivi è configurato in "*Mesh Mode*". Questa modalità di funzionamento costituisce una rete wireless cooperativa dove ogni nodo funge contemporaneamente da ricevitore, trasmettitore e ripetitore. Si tratta di un'infrastruttura decentralizzata, molto adattabile e resistente, dal momento che ogni nodo

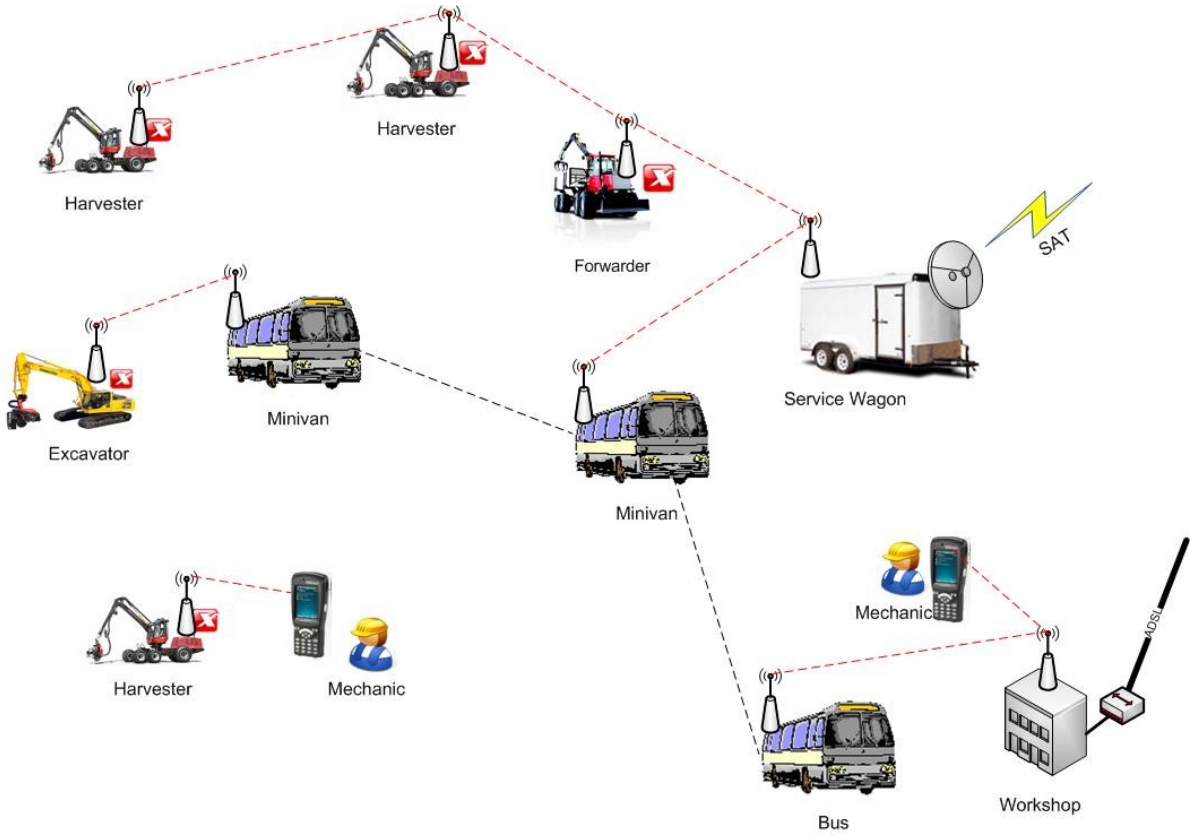


Figura 2.2. Le macchine forestali in campo, i Bus, i Minivan, le officine (Workshop) e i mezzi di servizio (Service Wagon) formano una "*Wireless Mesh Network*"

deve trasmettere un segnale al massimo fino ai nodi più vicini. I nodi fungono da ripetitori per trasmettere il segnale ricevuto in broadcast dai peers più vicini a quelli che sono troppo distanti per essere raggiunti direttamente.

Ciascuno di questi apparati wireless è anche in grado di funzionare da Access-Point tradizionale, in modo da permettere la connessione alle macchine a meccanici ed operatori per mezzo dei loro PDA. I Service Wagon possono inoltre comunicare con la Server Farm, che ospita i webservices e le basi di dati del sistema, per mezzo di un collegamento satellitare. Ciò permette alle macchine in campo di scambiare dati con le applicazioni del sistema. Le officine, che hanno una diffusione meno capillare rispetto ai mezzi di servizio e sono più distanti dal luogo delle operazioni, sono dotate di connettività DSL; questo link viene utilizzato come backup per la comunicazione tra le macchine forestali e la Server Farm, quando queste rientrano per operazioni di manutenzione.

In un tale scenario, la logica collocazione di nodi di Fog Computing è all'interno

dei Service Wagon che sono sparsi nella foresta e relativamente vicini alle macchine operatrici. Altri nodi di Fog Computing potrebbero essere ospitati all'interno delle officine. Nei nodi Fog potrebbe avvenire il processo di preparazione dei dati prima dell'invio degli stessi verso la Server Farm in modo da ottenere un risparmio di banda, vantaggioso considerando la connettività non convenzionale verso Internet. Si potrebbero effettuare delle analisi preliminari sui dati, che permetterebbero di segnalare agli operatori eventuali anomalie. Ma il vantaggio più importante che apporterebbe il Fog Computing in questo contesto sarebbe quello di slegare le operazioni di Fleet Management dalla necessità di una connessione verso la Server Farm. Si potrebbe pensare di ospitare parte di queste funzionalità nei nodi Fog, in modo che gli operatori in campo ne possano avere accesso anche "off-line". La Server Farm, dotata di risorse maggiori, avrebbe quindi lo scopo di fornire storage permanente dei dati, servizi di reporting per clienti che accedono da remoto ai servizi esposti dal sistema e l'esecuzione di analitiche più approfondite sui dati storici.

Nonostante gli apparati wireless adottati siano progettati per garantire coperture maggiori rispetto al Wi-Fi tradizionale e nonostante la robustezza fornita dalla modalità di funzionamento "Mesh" degli apparati, la topologia della rete può subire sostanziali variazioni nel tempo. Ciò è causato sia dall'ambiente ricco di ostacoli naturali che si frappongono tra i vari nodi, sia per la mobilità dei nodi stessi all'interno di una superficie di lavoro molto estesa. Le Wireless Mesh Network, se ben progettate, funzionano bene per la copertura di aree a diversi chilometri di distanza dove però i nodi sono per lo più fissi: è il caso di apparati ed antenne dislocati all'esterno degli edifici. Quindi, potrebbero verificarsi partizionamenti di rete tali per cui alcuni dispositivi o gruppi di dispositivi, compresi i nodi Fog, rimangano isolati per molto tempo con conseguenti potenziali perdite di dati o quantomeno un loro trasferimento molto ritardato. Si rende quindi necessaria una soluzione ai livelli più alti dello stack protocollare, come l'adozione del paradigma *store-carry-and-forward*, discusso nella Sezione 2.2.

La conservazione dei pacchetti non immediatamente inoltrabili verso la destinazione unitamente alle caratteristiche della rete magliata, consentirebbe oltre ad una maggiore affidabilità anche lo sfruttamento della mobilità dei nodi per trasferire l'informazione tra "*isole*" non direttamente connesse. Questa operazione viene in gergo chiamata "*data muling*" e, in particolare, Minivan e Bus potrebbero essere sfruttati a tale scopo.

2.3 Obiettivo della tesi

Sulla base delle considerazione precedenti, in questa tesi viene delineato un sistema di comunicazione per applicazioni *IIoT* che facciano uso di un'architettura basata sul Fog Computing in contesti sfidanti per la connettività.

L'architettura proposta si prefigge di essere trasparente nei confronti delle applicazioni esistenti, che per la comunicazione fanno uso di protocolli di messaggistica basati sul paradigma "*publish/subscribe*".

Per garantire affidabilità al protocollo di comunicazione e un inoltro dei messaggi che sfrutti la mobilità dei nodi, si è utilizzata un'implementazione del Bundle Protocol chiamata IBR-DTN, descritta nel Capitolo 4.

Capitolo 3

Il Fog computing

In questo capitolo[1] viene introdotto il concetto del Fog computing, quali sono le differenze e i vantaggi rispetto al Cloud tradizionale e vengono esaminati in dettaglio alcuni casi d'uso significativi. Il capitolo si conclude con una disamina della piattaforma realizzata da Nebbiolo Technologies, utilizzata per lo sviluppo di questa tesi.

3.1 Definizione

Il Fog computing (o Edge computing) è un'*architettura orizzontale*, a livello di sistema, utile a distribuire risorse e servizi di calcolo, immagazzinamento di dati, controllo e funzionalità di rete più vicino agli utilizzatori (dispositivi), lungo l'infrastruttura che li connette al Cloud [6].

Il Fog computing, dunque, estende il modello tradizionale del Cloud avvicinando in parte risorse e servizi all'edge della rete: è metaforicamente *una "nuvola" più vicina al suolo*. Un esempio dell'architettura è mostrato in Figura 3.1.

Fog e Cloud offrono la stessa tipologia di risorse (capacità di elaborazione e immagazzinamento dati, funzionalità di rete) e condividono gli stessi meccanismi di funzionamento (virtualizzazione, multi-tenancy), tuttavia esiste una fondamentale differenza che dà al Fog la sua ragion d'essere. Il Cloud è un'architettura fortemente centralizzata dove grosse moli di dati, provenienti dai dispositivi che ne utilizzano i servizi, sono costretti a transitare lungo il core della rete fino a raggiungere i datacenter di un service provider. Contrariamente, il Fog offre risorse e servizi seguendo la naturale distribuzione geografica degli utilizzatori. Nonostante questa apparente controtendenza presente nei due paradigmi, Fog e Cloud traggono reciproci benefici dalla loro mutua interazione.

[1] Il capitolo è stato scritto in collaborazione con il tesista Andrea Alfò

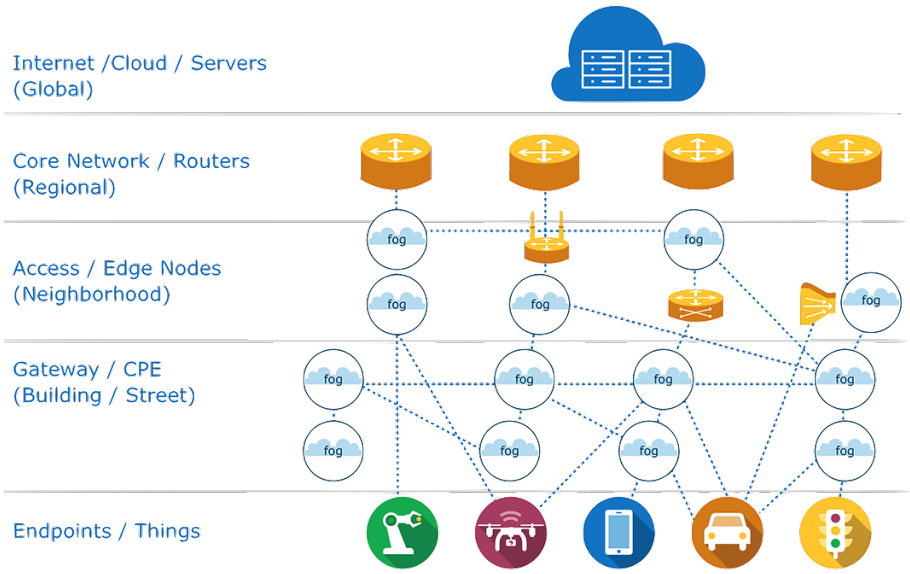


Figura 3.1. Architettura Fog Computing (Fonte: OpenFog Consortium [6])

Le applicazioni real-time o quelle installate su dispositivi con mobilità veloce o scarsa connettività non possono fare affidamento sul Cloud, ed è qui che il Fog computing entra in gioco: la vicinanza all'edge implica ritardi molto bassi; l'elaborazione locale consente l'invio di una quantità ridotta di dati ai datacenter, con relativo risparmio di banda e costi; la perdita temporanea di connettivà non pregiudica lo svolgimento delle operazioni in corso. Tutti questi vantaggi vengono riassunti, dall'OpenFog Consortium, con il termine ***SCALE*** [6]:

- **S**ecurity: dati sensibili elaborati localmente su un nodo fidato
- **C**ognition: approccio client-centric che favorisce una maggiore autonomia dal Cloud
- **A**gility: innovazione rapida e applicazioni scalabili utilizzando un'infrastruttura comune
- **L**atency: basso ritardo di comunicazione
- **E**fficiency: condivisione di risorse localmente inutilizzate, derivanti dalla partecipazione dei dispositivi finali

3.2 Casi d'uso

In questa sezione vengono descritti dei casi d'uso rilevanti per mettere in luce i vantaggi e le caratteristiche chiave del Fog computing [7].

3.2.1 Sistema per il controllo intelligente del traffico

La congestione del traffico è un problema rilevante nei grandi agglomerati urbani e può portare a paralizzare le grandi città. Gli impatti negativi che una situazione di congestione stradale ha sulla comunità sono diversi: non solo ritardi e disguidi, ma anche stress sui veicoli e sui conducenti, ostacolamento di mezzi impegnati in emergenze, maggior consumo di carburante e conseguente aumento delle emissioni.

Un sistema per il controllo intelligente del traffico si occupa della prevenzione degli incidenti, di mantenere costante il flusso del traffico e di raccogliere dati rilevanti per migliorare il sistema stesso. Questi tre obiettivi sono molto diversi tra loro dal punto di vista della scala del tempo: il primo richiede tempi di reazione tipici di un sistema real-time; il secondo può tollerare ritardi decisamente maggiori; l'ultimo riguarda l'analisi a lungo termine dei dati raccolti e delle decisioni prese dal sistema.

L'architettura flessibile e geograficamente distribuita del Fog Computing permette di soddisfare questi requisiti favorendo tempi di latenza molto bassi, integrazione e analisi di dati provenienti da dispositivi eterogenei e precedentemente non interconnessi (dispositivi di controllo del traffico e segnaletica elettronica, sensori sul ciglio della strada e dispositivi a bordo dei veicoli) e demandando al Cloud l'analisi in batch delle performance del sistema.

3.2.2 Consegna dei pacchi tramite droni

Ogni giorno miliardi di pacchi vengono distribuiti in tutto il mondo e i costi legati al loro trasporto (tramite treni, aerei, furgoni, ecc...) diventano sempre maggiori. La tecnologia IoT ha migliorato alcuni aspetti di questo settore, introducendo una migliore gestione del traffico e ottimizzando i percorsi da seguire. L'idea di utilizzare dei droni UAV per la consegna dei pacchi diventa sempre più concreta per abbattere ulteriormente i costi e migliorare il servizio, garantendo tempi di consegna inferiori anche ai 30 minuti. Le sfide da affrontare sono molteplici:

1. Collisioni e sicurezza: in uno scenario in cui milioni di droni volano contemporaneamente, il pericolo di una collisione tra loro diventa concreto; in aggiunta piccoli aerei, uccelli o grandi edifici rappresentano fin da subito una minaccia reale. Diventa quindi fondamentale riuscire ad avere tempi di reazione dell'ordine di pochi millisecondi per evitare incidenti.

2. Banda limitata: la costante comunicazione di dati e il monitoraggio in tempo reale dei droni potrebbero saturare la banda che si ha a disposizione. Nelle

zone in cui la connettività è limitata o assente bisognerebbe utilizzare canali satellitari per comunicare con il Cloud, il che comporterebbe costi proibitivi.

3. Hub per il controllo: così come per gli aerei di linea e quelli commerciali, anche i droni hanno bisogno di un sistema di controllo per coordinare i piani di volo dei vari soggetti.

Supponiamo di voler utilizzare il Cloud: se si considera che un drone può volare a 160km/h e che il minor RTT ottenibile è di circa 80ms, significa che tra una comunicazione e l'altra il drone ha percorso la distanza di circa 214m [8]. Far volare il drone per 214m (nel caso migliore) senza alcun tipo di controllo è veramente impensabile, questo ci fa capire come la scelta del Cloud in questa circostanza non può essere la soluzione. Utilizzando il Fog computing nella comunicazione tra il drone e il "centro di controllo", i comandi di update possono essere inviati con un ritardo così basso da far muovere il drone di pochi centimetri prima che il prossimo messaggio gli venga consegnato. Il Cloud rimane tuttavia sempre presente e i dati di monitoraggio vengono inviati, in questo caso senza fretta, ai grossi datacenter per essere poi elaborati e/o salvati.

3.2.3 Videosorveglianza

Ai fini di garantire sicurezza alle persone, alle cose e ai luoghi negli ultimi anni sono state installate sempre più telecamere di sorveglianza. Una singola telecamera può generare (dipende dalla risoluzione e dalla qualità scelta) anche 1TB di dati al giorno. Le telecamere dei sistemi di sorveglianza generano dati che devono essere analizzati in tempo reale per garantire la pubblica sicurezza. Il numero crescente di telecamere e l'aumentare della risoluzione e della qualità dei video prodotti, hanno generato un enorme flusso di dati verso il Cloud. I problemi quindi legati a questo scenario sono molteplici:

1. Alta definizione: il modello tradizionale del cloud era stato realizzato per sistemi a bassa risoluzione, attualmente non è difficile trovare in commercio telecamere a basso costo che generano filmati ad una risoluzione Full-HD (1080p) o 4K, questo aumenta drasticamente la dimensione dei dati da trasmettere.

2. Tempi di risposta: nel caso di un'intrusione all'interno di un edificio, è necessario accedere immediatamente ai dati per poter permettere alla sicurezza di localizzare la minaccia in tempi brevi, lo stesso discorso vale per un fuggitivo a bordo di un mezzo che scappa per le vie di una città.

3. Scalabilità: nel caso di luoghi con numerosi dispositivi di acquisizione video (es: aeroporti, stazioni, stadi) la quantità di dati da trasmettere al cloud supera di gran lunga la banda che si ha a disposizione.

Il Fog computing forma una maglia di nodi che intelligentemente partizionano il processamento video tra i nodi fog stessi e il Cloud garantendo il tracciamento real-time, il rilevamento di anomalie e la raccolta di informazioni significative. Gli algoritmi di analisi video possono essere installati direttamente sui nodi fog, riducendo il traffico verso il cloud e garantendo tempi di latenza ridotti.

3.3 La tecnologia di Nebbiolo Tecnologies

La piattaforma di Fog Computing sviluppata da Nebbiolo Technologies è composta da piccoli server distribuiti all'edge della rete, con gestione delle risorse tipiche del Cloud e funzionalità real-time, al fine di colmare il gap tra i dispositivi IoT e l'infrastruttura Cloud [9].

3.3.1 Cenni sull'azienda

Nebbiolo Tecnologies, start-up californiana con radici italiane, nasce nel 2015 con l'obiettivo di progettare e costruire apparati per il Fog computing. L'idea di questa emergente tecnologia è nata nel lontano 2010 dalla mente dell'attuale CEO e founder Flavio Bonomi, quando ancora ricopriva il ruolo di vicepresidente del dipartimento di ricerca di Cisco Systems. Gli ingegneri di Nebbiolo Tecnologies sono pertanto i pionieri del Fog computing con la loro prima serie di dispositivi, denominati fogNode™, lanciati sul mercato il 9 febbraio 2017. Dispositivi che, essendo in grado di fornire bassa latenza di comunicazione e capacità di elaborazione real-time, hanno attirato l'attenzione di grandi player nel campo dell'automazione industriale come Kuka e TTTech.

3.3.2 Architettura

L'architettura della piattaforma di Fog computing di Nebbiolo Technologies è suddivisa, come mostrato in Figura 3.2, in tre livelli: i nodi hardware chiamati fogNode™, il livello software fogOS™ e il sistema di gestione fogSM™ (o NSM - Nebbiolo System Management).

fogNode™

Si basano su un'architettura fisica modulare e pertanto possono essere costruiti con diversi fattori di forma e diverse funzionalità (Figura 3.3) in base alle esigenze, permettendone il loro utilizzo in diversi ambiti applicativi. Ogni fogNode™, dunque, può essere composto da uno o più fogLet: si tratta di un modulo con risorse computazionali, di storage e connettività indipendenti. I fogLet utilizzano CPU di elevate prestazioni e dispongono di storage veloci (SSD). Sono equipaggiati con switch

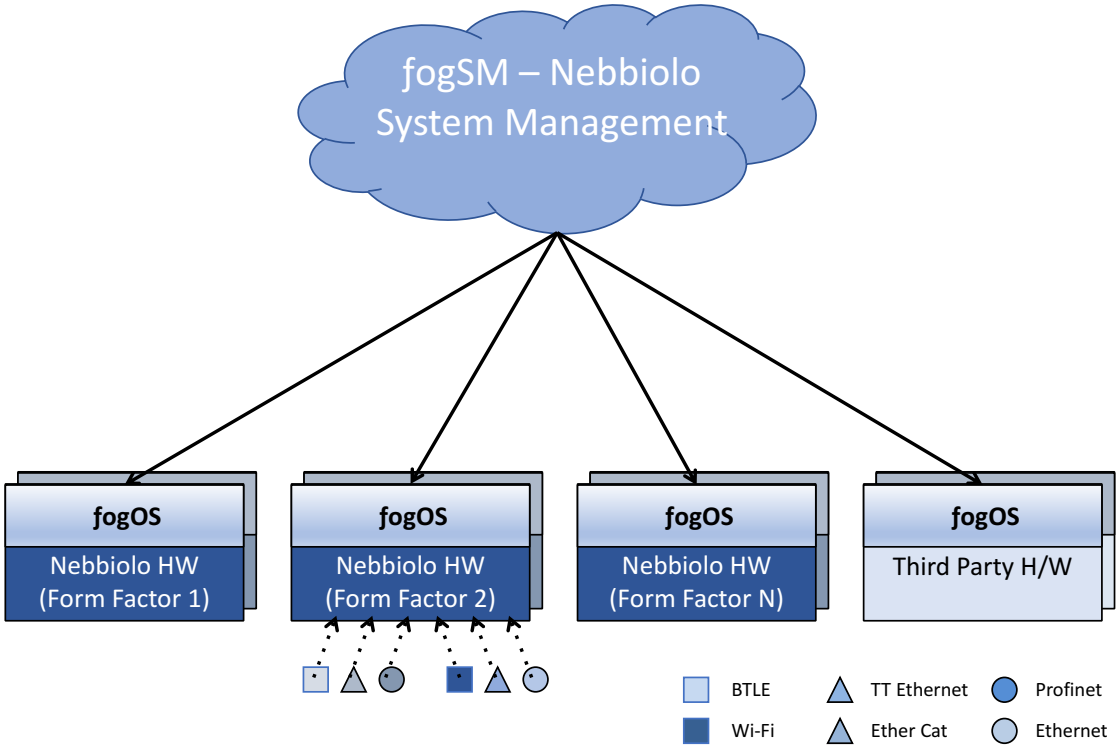


Figura 3.2. Architettura della piattaforma Fog di Nebbiolo Technologies.

Ethernet con capacità TSN - *Time-Sensitive Networking* - per fornire connettività con latenze deterministiche, fondamentali per un utilizzo in ambiti real-time. Per espanderne la connettività, possono opzionalmente essere equipaggiati con moduli Wi-Fi ed LTE [10].

La versione utilizzata in questa tesi è il *Nebbiolo fogNode NFN-300* e comprende al suo interno un solo fogLet *NFL-1000-C*, dotato di 6 porte Gigabit Ethernet IEEE 802.3ab, una porta VGA, una USB 2.0 ed una porta seriale RJ45. All'interno è presente un SSD SATA 3 da 120GB ed un processore Dual Core Intel Haswell di quarta generazione (Core-i5 4402E). Il consumo energetico è di circa 100W, pesa 7.6KG ed è certificato IP20 (apparecchio da interno) [11].

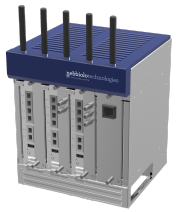
fogNode™ Series	
	• Fan cooled chassis • Per Slot: 4-8 core x86 i5/i7, • 128-512G Storage, • 8-16G memory, • LTE and WiFi • Secure Hardware, • Real Time capable with embedded Switch • 3 slots connected backplane for High Availability, Scale and Aux cards (e.g. GPU, Storage, Safety)
	• Fanless, 24V DC powered • 4-8 core x86 Corei5/i7, • 128-512G Storage, • 8-16G memory, • LTE and WiFi, • Secure Hardware, • Real Time capable with Embedded Switch
	• 4 core Atom, • 32-128G Storage, • 8G memory, • LTE and WiFi, • Secure Hardware, • Real Time capable with Embedded Switch
	• 2 core Atom, 32G Storage, 4-8G memory, • 3G and WiFi Gateway functions

Figura 3.3. Diverse configurazioni di fogNode™ (Fonte: Nebbiolo.tech [10]).

fogOS™

È il sistema operativo realizzato da Nebbiolo per i nodi Fog. È composto da due parti (Figura 3.4): la prima, basata sulla distribuzione Linux CentOS, racchiude al proprio interno tutto l'insieme di librerie e software utili per l'esecuzione degli applicativi in locale; la seconda, invece, gira sul Cloud e si occupa della gestione globale dei nodi.

- **Host OS/Hypervisor**: un gradino sopra l'hardware del fogLet è presente il sistema operativo base, che comprende un real-time hypervisor basato su KVM. Quest'ultimo ha il compito di occuparsi della gestione del ciclo di vita di virtual machine e container.

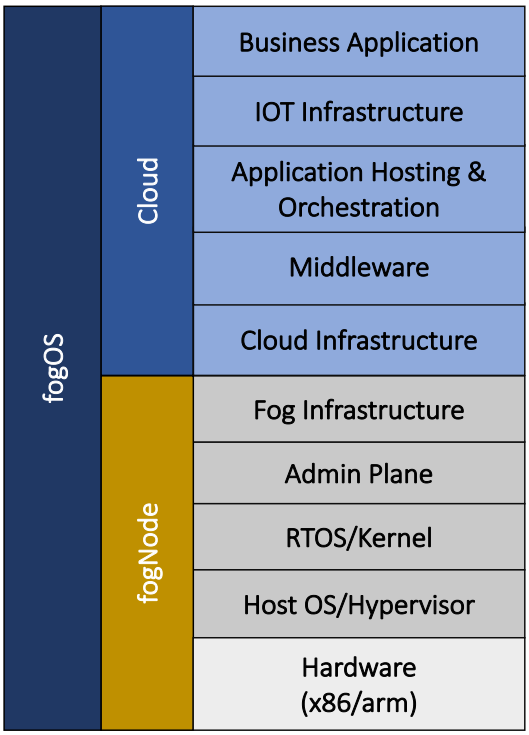


Figura 3.4. Lo stack che compone il fogOS^{TM} (Fonte: Nebbiolo.tech [12]).

- **RTOS**: è possibile virtualizzare sistemi operativi dotati di capacità di real-time, così come sistemi operativi non real-time.

- **Admin Plane**: si occupa della gestione delle connessioni TLS o VPN, dei certificati e della registrazione del device nel sistema di management. Fornisce inoltre un meccanismo per la generazione di allarmi e per il monitoraggio dei guasti interni al nodo. È in questo layer che avviene il riconoscimento di periferiche esterne, o *asset*, e la gestione della rete e dello storage.

- **Fog Infrastructure**: mette a disposizione un client OPC UA, basato sul protocollo TCP, che permette di realizzare un ponte tra i componenti di automazione integrati e i sistemi operativi tradizionali. Questo client fornisce tutte le informazioni desiderate a quegli applicativi che dispongono di opportune autorizzazioni [13]. In questo livello avviene inoltre la gestione del TSDB (Time-Series Database)[2], dei sensori e del bus dati.

[2]Database ottimizzato per la gestione di dati e array di numeri indicizzati in base al tempo.

- **Cloud Infrastructure**: a partire da questo livello si parla di componenti che vengono eseguiti sul Cloud. Si occupa della catalogazione dei nodi, degli upgrade software e dello scaling orizzontale. Esegue un server OPC UA al quale i nodi si collegano per ottenere informazioni e dati. Fornisce inoltre diverse API server.

- **Application Hosting & Orchestration**: mette a disposizione un *App Store* dal quale è possibile effettuare il provisioning delle applicazioni. Si occupa della gestione di certificati, policy, RBAC (controllo accessi), topologia nodi e analytics.

fogSM™

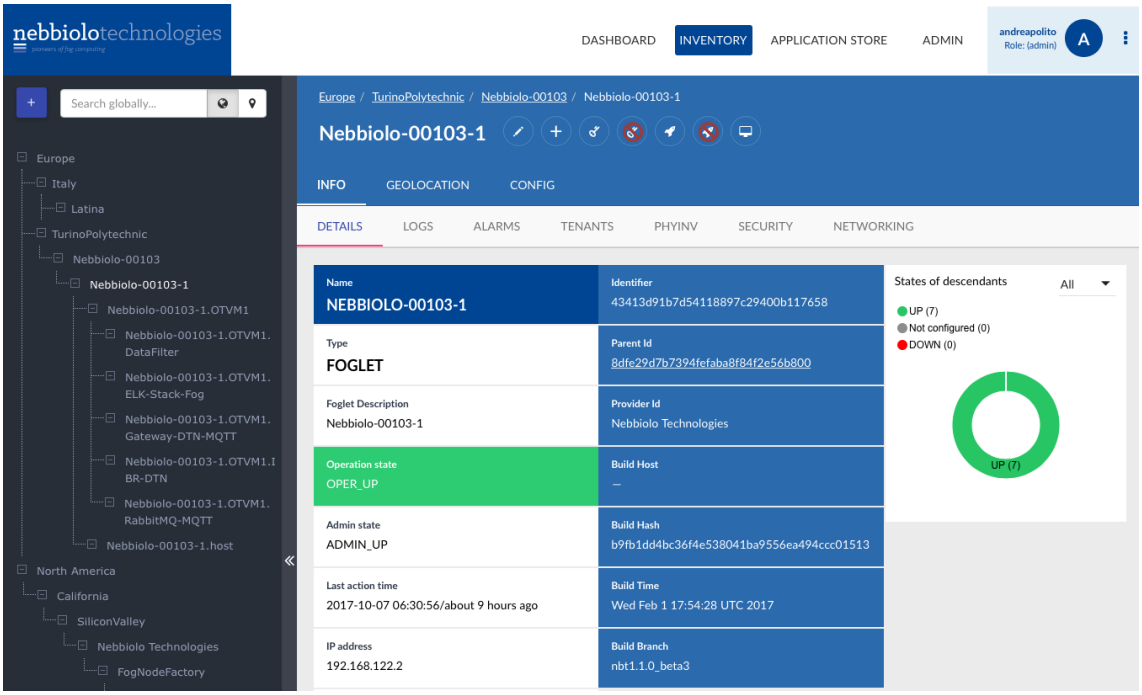


Figura 3.5. Screenshot dell'interfaccia web del fogSM™ - sezione "inventario".

Il fogSM™ (o NSM - Nebbiolo System Management) fornisce un'interfaccia di gestione unificata per una federazione di fogNode™. Si raggiunge attraverso un browser web e permette di eseguire diverse operazioni. È formato da quattro tab:

1. **Dashboard**: subito dopo il login, si accede a questa pagina. Da qui è possibile visualizzare lo stato dei nodi e degli asset (attivo, inattivo, non configurato),

controllare le ultime notifiche e visualizzare la disposizione geografica dei nodi all'interno di una mappa.

2. **Inventory**: vengono mostrati i nodi aggregati in base alle policy scelte. È possibile espandere le varie voci per poter visualizzare quali container o applicazioni sono in esecuzione. Da questa schermata si può eseguire il deploy di un'applicazione, il suo undeploy o la cancellazione di un nodo. Sono disponibili inoltre dettagli aggiuntivi riguardanti la versione del sistema installato, la configurazione di rete e i vari log generati (Figura 3.5).

3. **Application Store**: permette di caricare applicazioni, macchine virtuali e container Docker all'interno dell'NSM. È possibile definire categorie di applicazioni ed autorizzare l'onboard solo per alcune di esse. I container possono essere caricati direttamente dal Docker Hub, specificando repository e tag. Tutti gli eventuali parametri da passare al container, compreso variabili d'ambiente e port mapping, vengono gestiti all'interno di questo tab. Il provisioning avviene come mostrato in Figura 3.6: l'applicativo deve essere prima caricato all'interno dell'Application Store; successivamente si sceglie l'asset o il nodo su cui eseguire il deploy.

4. **Admin**: in questa ultima schermata è possibile configurare notifiche, ruoli, aggiungere o rimuovere utenti e controllare le informazioni relative all'NSM stesso.

3.3.3 Configurazione del fogNode™

In questa sezione viene descritta la configurazione del nodo utilizzato per lo sviluppo di questa tesi, collocato all'interno del laboratorio del *Netgroup* presso il Politecnico di Torino.

Macchine virtuali

Di base è presente una macchina virtuale, denominata ***AdminVM***, che rappresenta l'interfaccia per la gestione del fogLet da remoto o dall'apposita porta di management, collocata sullo switch dello stesso. La AdminVM contiene la versione 1.12.1 di Docker e tutto lo stack software necessario per eseguire il deployment di applicazioni dal fogSM™, nonché il message broker RabbitMQ. Questa macchina virtuale deve essere sempre accessibile da parte di Nebbiolo per poter aggiornare il software del sistema e per poter sincronizzare il nodo con il Nebbiolo System Manager. Per questioni di privacy è quindi sconsigliato effettuare il deployment delle proprie applicazioni all'interno di essa.

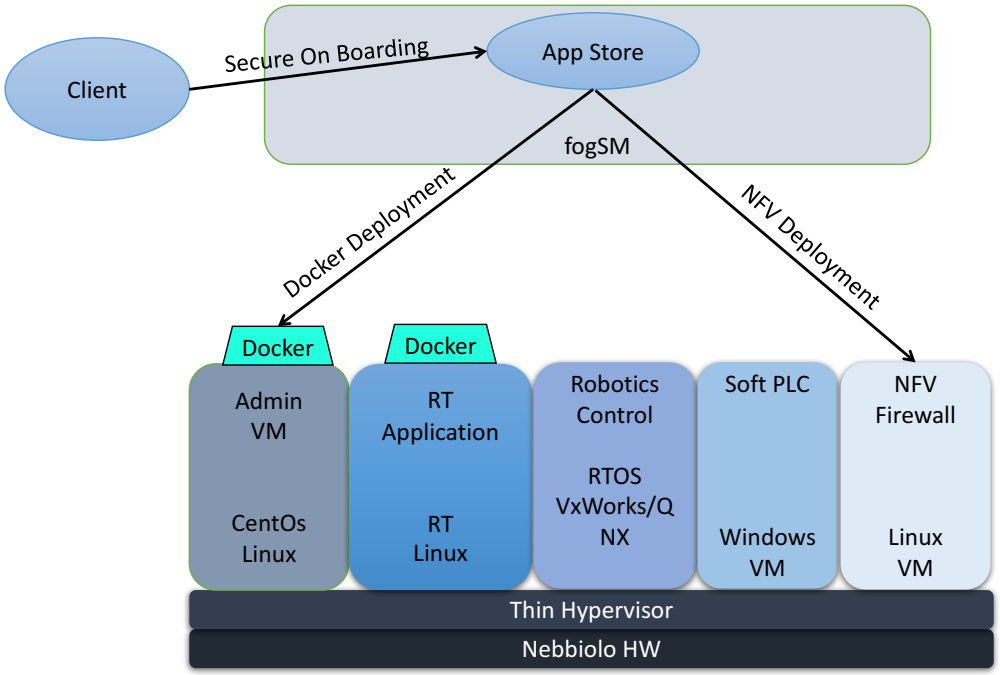


Figura 3.6. Esempio di provisioning di un'applicazione (Fonte: Nebbiolo Technologies).

A tale scopo è preferibile istanziare altre virtual machine, denominate ***OTVM***, preposte a contenere applicazioni e servizi. L'immagine di queste OTVM è fornita da Nebbiolo Technologies nell'Application Store e contiene anch'essa Docker e tutto il software necessario per eseguire il deployment di applicazioni da remoto. A differenza della AdminVM, nell'immagine base della OTVM non è compreso RabbitMQ, ma questo può essere facilmente istanziato dall'NSM sottoforma di container Docker. Nel caso specifico del nodo utilizzato è stata istanziata una singola OTVM.

Generalmente è presente anche un'altra macchina virtuale, la *Forwarder Virtual Machine* – ***FWD VM***, che fornisce la funzionalità di inoltro del traffico verso Internet a tutte le OTVM. Quest'ultime, infatti, non sono connesse verso l'esterno ma possono comunicare tra loro all'interno del fogLet che le ospita. Questa scelta progettuale nasce dalla volontà di isolare le OTVM, sulle quali solitamente vengono ospitate applicazioni e servizi che elaborano dati sensibili. Solo i dati che eventualmente devono uscire dal nodo saranno inoltrati, tramite il meccanismo del message brokering, alla FWD VM. Anch'essa è dotata, come la AdminVM, del message broker RabbitMQ. Grazie agli svariati protocolli supportati e alla flessibilità dell'architettura publish/subscribe, esso rappresenta il punto d'uscita verso svariati servizi sul Cloud. Per le future release software sono previsti appositi *"connector"*

per facilitare il collegamento alle piattaforme Cloud dei maggiori service providers (Microsoft Azure, Amazon Web Services, Google Container Engine).

Configurazione di rete

Il fogLet presenta 6 interfacce Gigabit Ethernet, 4 sono collegate ad uno switch interno, le restanti due sono dirette. Le porte dello switch prendono i nomi S0, S1, S2 ed S3, mentre quelle dirette E0 ed E1. La connettività ad Internet deve essere fornita attraverso la porta E0, mentre la E1 viene utilizzata per il management locale.

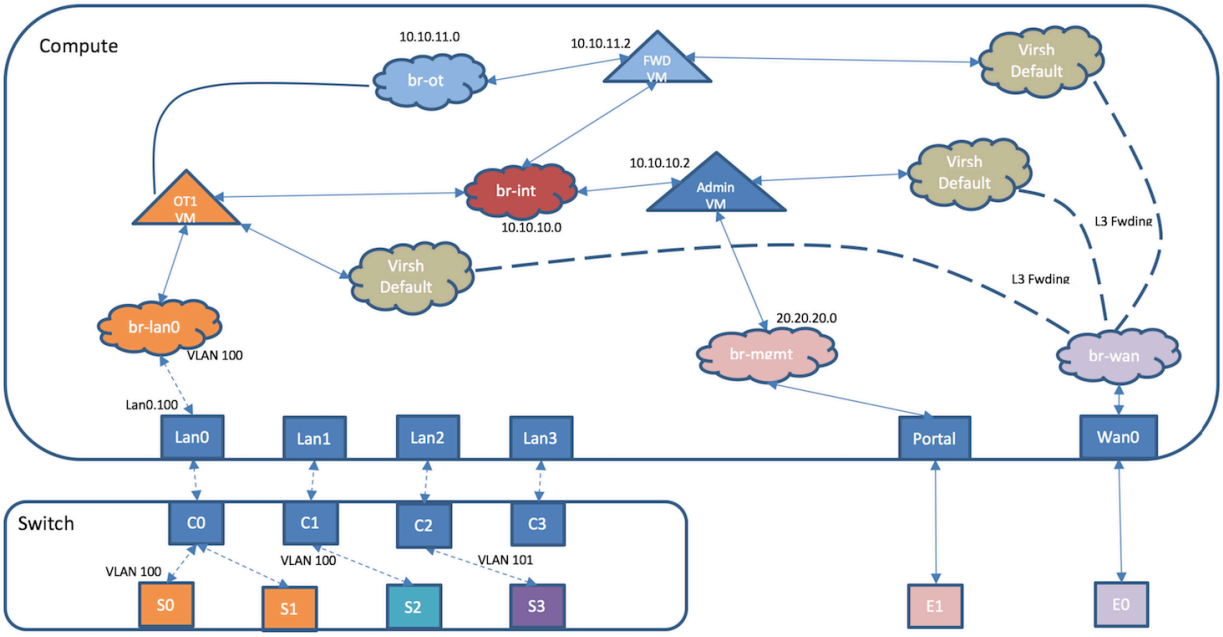


Figura 3.7. Architettura di rete e configurazione interna del fogLet (Fonte: Nebbiolo Technologies)

In Figura 3.7 viene mostrata la specifica configurazione di rete del fogLet a disposizione, da cui si può notare la presenza di numerose "nuvole" che interconnettono le varie macchine virtuali: si tratta di bridge Open vSwitch (OVS). Di seguito si elencano i bridge presenti e le loro rispettive funzioni:

- "**br-int**" usato per la comunicazione interna tra le VM;
- "**br-mgmt**" connette la AdminVM alla porta fisica E1 per le funzionalità di gestione;

- "**br-wan**" fornisce la connettività verso Internet;
- "**br-lan0**" utilizzato per connettere le porte fisiche S0 ed S1 dello switch con la OTVM1;
- "**br-ot**" collega tutte le OTVM (in questo caso solo una) con la Forwarder VM (FWD VM);

Sempre in Figura 3.7 si può notare come l'unica OTVM presente, diversamente da quanto descritto in precedenza, sia collegata al bridge denominato "br-wan". Nella nostra configurazione, infatti, la OTVM1 può fare uscire traffico verso Internet senza passare dalla FWD VM. Si tratta, però, di una "feature" di testing e dovrebbe essere eliminata nelle future release software.

Si noti, inoltre, come le porte dello switch S2 ed S3 siano al momento prive di alcun collegamento con gli elementi interni del fogLet. Le porte S0 ed S1 dello switch, come detto, sono invece connesse alla OTVM1 attraverso il bridge "br-lan0". Nello specifico le porte S0 ed S1 sono rispettivamente collegate alle interfacce "eth0" ed "eth1" della OTVM1. Alla porta S0 è stato connesso un Access Point Wi-Fi, mentre sull'interfaccia eth0 è stato configurato un server DHCP: tutti i dispositivi che si connettono all'access point vengono configurati con un indirizzo appartenente alla rete privata 192.168.1.0/24, con l'ultimo ottetto nel range 100-199. L'interfaccia eth0 e l'access point hanno invece indirizzo rispettivamente 192.168.1.1 e 192.168.1.2.

Capitolo 4

Delay/Disruption-Tolerant Networking

Nella prima parte di questo capitolo viene introdotto il concetto di *Delay/Disruption-Tolerant Networking* (DTN) e vengono illustrati l'architettura e i protocolli che operano nelle suddette reti. Successivamente viene introdotta un'implementazione software open-source di DTN – IBR-DTN – utilizzata per lo sviluppo di questa tesi. Il contenuto di questo capitolo è tratto da [4], [14], [5], [15], [16].

4.1 Introduzione

Il concetto di *Delay-Tolerant Networking* nasce nell'ambito dell'*Interplanetary Networking* (IPN). Questo scenario è caratterizzato da ritardi di trasmissione molto elevati, ad esempio dell'ordine delle decine di minuti sulla distanza Terra-Marte, e da lunghe interruzioni della linea di visibilità necessaria alla comunicazione radio, dovute al moto dei pianeti e alla rivoluzione dei satelliti intorno ad essi.

Il dominio di applicazione è stato poi esteso a tutti quei contesti terresti che presentano criticità simili. È il caso di reti di sensori che, per vincoli energetici, si connettono ad intervalli prestabiliti o di reti wireless che non riescono a mantenere costantemente una topologia che garantisca ai nodi una connettività end-to-end, a causa di frequenti interruzioni dovute a fattori ambientali o alla mobilità dei nodi stessi. La differenza principale tra le applicazioni terresti e la comunicazione nello spazio sta nei ritardi di trasmissione che sono generalmente molto minori nel primo caso e determinati dall'intermittenza delle connessioni. Per questo motivo si usa spesso il termine *Disruption-Tolerant Networking*.

Per ovviare a queste criticità, le reti DTN rompono la *semantica end-to-end* tipica del *Internet Protocol* (IP) con un approccio di tipo *hop-by-hop* denominato *store-carry-and-forward*, come descritto nella Sezione 2.1.

Punto di riferimento per l'attività di ricerca e standardizzazione nell'ambito DTN è stato il gruppo DTNRG – *DTN Research Group* – in seno all'IRTF – *Internet Research Task Force*. Nel 2007, allo scopo di fornire un framework comune per lo sviluppo di algoritmi ed applicazioni per reti DTN, questo gruppo ha pubblicato la RFC4838 [14] e la RFC5050 [5] che descrivono rispettivamente l'architettura delle reti DTN e il *Bundle Protocol*. La maturazione dei risultati ha portato al passaggio delle attività da IRTF a IETF – *Internet Engineering Task Force*. Di conseguenza il DTNRG ha cessato la propria attività ed è stato sostituito dal gruppo denominato *DTN Working Group* (DTNWG).

4.2 Architettura

L'architettura di un nodo DTN prevede l'aggiunta di un nuovo strato allo stack di rete: il *Bundle Layer*. Il protocollo relativo a questo nuovo strato è il *Bundle Protocol* (BP). Questo protocollo specifica come un'implementazione di DTN deve comportarsi e definisce un formato per lo scambio di messaggi denominati "*bundle*", descritti nella Sezione 4.4.

Questo nuovo strato si colloca generalmente tra il livello applicativo e il livello trasporto (Figura 4.1). Al di sotto del Bundle Layer è presente un sottostrato detto *Convergence Layer Adapter* (CLA), che funge da interfaccia verso un protocollo di livello inferiore. Esistono CLA per i protocolli di trasporto utilizzati in Internet, come TCP e UDP, e per quelli pensati specificamente per le DTN, come LTP – *Licklider Transmission Protocol* – impiegato per collegamenti attraverso lo spazio.

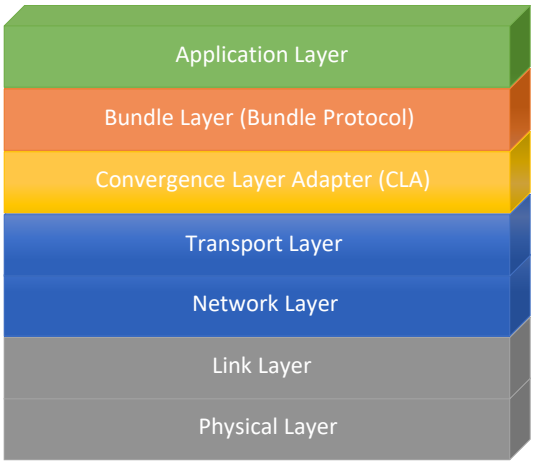


Figura 4.1. Stack di rete di un nodo DTN

Un'altra possibilità prevede l'utilizzo del Bundle Protocol come protocollo di livello rete. In tal caso è necessario un Convergence Layer Adapter opportuno, che dovrà interfacciarsi direttamente con il livello *data-link*.

Il Bundle Layer, quindi, è agnostico rispetto ai livelli sottostanti. In una DTN possono coesistere e comunicare nodi che utilizzano protocolli di trasporto (e di livello inferiore) differenti; inoltre i contatti possono eventualmente propagarsi attraverso nodi intermedi che non possiedono il Bundle Layer nel proprio stack di rete.

Un'applicazione istanziata su un nodo DTN invia messaggi di lunghezza arbitraria detti *Application Data Unit* (ADU) al cosidetto *Bundle Protocol Agent* (BPA), ovvero il componente di un nodo DTN che offre i servizi del Bundle Protocol. Il BPA incapsula le ADU ricevute dall'applicazione in uno o più *Protocol Data Unit* (PDU) chiamati "*bundle*" e si occupa del loro instradamento attraverso la DTN, se questo è possibile, e della loro conservazione.

Si ricorda, infatti, che l'architettura DTN non si aspetta che i collegamenti siano sempre disponibili o affidabili, ma richiede che ogni nodo possa memorizzare i bundle in modo persistente finché necessario. Lo storage diventa quindi una risorsa fondamentale per un'architettura DTN, che deve essere gestita e protetta. Questo introduce nuovi problemi sui quali attualmente ruota la maggior parte della ricerca riguardante le DTN.

In Figura 4.2 viene mostrato come un nodo DTN (Bundle Forwarder) deve interagire con le applicazioni, con lo storage, con le decisioni prese dai protocolli di routing e con uno o più CLA per le reti sottostanti.

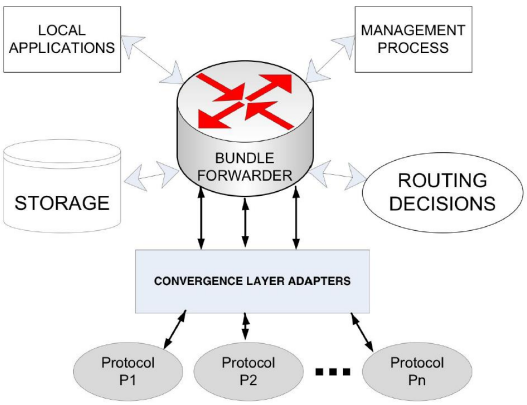


Figura 4.2. Un esempio di nodo DTN mostra come questo deve interagire con le diverse componenti che ne costituiscono l'architettura (Fonte: [15])

4.2.1 Endpoint Identifiers (EIDs)

Un *endpoint* DTN è un insieme di uno o più nodi DTN identificati da un *Endpoint Identifier* (EID). Il caso più comune è quello in cui esiste un solo nodo identificato da un determinato EID, pertanto questo endpoint è detto *singleton*. Se un singolo EID identifica più nodi, allora questi nodi formano un gruppo multicast. Un nodo può anche essere identificato da più di un EID ed essere quindi membro di più gruppi di endpoint.

Gli EID seguono la sintassi delle *URI*: `<scheme_name>:<scheme-specific_part>`. Lo schema originale delle DTN è identificato dalla stringa "`dtn`". Successivamente è stato introdotto lo schema denominato *Compressed Bundle Header Encoding* (CBHE) che è identificato dalla stringa "`ipn`". Quest'ultimo è solo una diversa convenzione per rappresentare l'EID in una formato compresso ed è utilizzato per le applicazioni interplanetarie; non vi sono differenze sostanziali tra i bundle identificati da uno schema piuttosto che l'altro. In entrambi gli schemi una prima porzione della "scheme-specific part" indica il nodo, un'altra, il "*demux token*", individua la singola applicazione.

Un EID nello schema DTN ha la forma `dtn://nodo/demux_token` dove "*node*" e "*demux_token*" sono stringhe qualsiasi, esclusi alcuni caratteri speciali. Un EID nello schema CBHE ha la forma `ipn:node.demu_token`, dove "*node*" e "*demux_token*" sono stringhe numeriche. Di seguito sono riportati alcuni esempi di EID:

- dtn://bobsPC/files
- dtn://bobsPC/
- ipn:81.2
- ipn:81.0

Si noti che nello schema DTN il demux token non è sempre presente: questo è il caso di bundle amministrativi diretti al Bundle Protocol Agent. Nello schema CBHE, al contrario, il demux token numerico deve sempre essere presente; in particolare per indicare il Bundle Protocol Agent si utilizza il numero "0".

La richiesta da parte di un'applicazione di ricevere Application Data Unit destinate a un determinato EID è detta "*registrazione*", e in generale viene mantenuta in modo persistente dal nodo DTN. Questo permette alle registrazioni effettuate dalle applicazioni di resistere ad un riavvio dell'applicazione stessa o ad un riavvio di sistema. Una richiesta di registrazione non ha garanzia di successo. Ad esempio, un'applicazione potrebbe registrarsi per ricevere ADU specificando un EID non interpetabile o non corrispondente al nodo DTN che sta servendo la richiesta.

4.2.2 Affidabilità e Trasferimento di Custodia

Le DTN supportano la ritrasmissione da nodo a nodo di dati persi o corrotti sia a livello di Bundle Layer sia a livello trasporto. Ma, poiché nessun protocollo di trasporto è tipicamente in grado di operare end-to-end attraverso una DTN, l'affidabilità end-to-end può essere implementata solamente nel Bundle Layer. Quest'ultimo mette opzionalmente a disposizione due strumenti per migliorare l'affidabilità di consegna dei bundle: *acknowledgments end-to-end* e il meccanismo denominato "*trasferimento di custodia*".

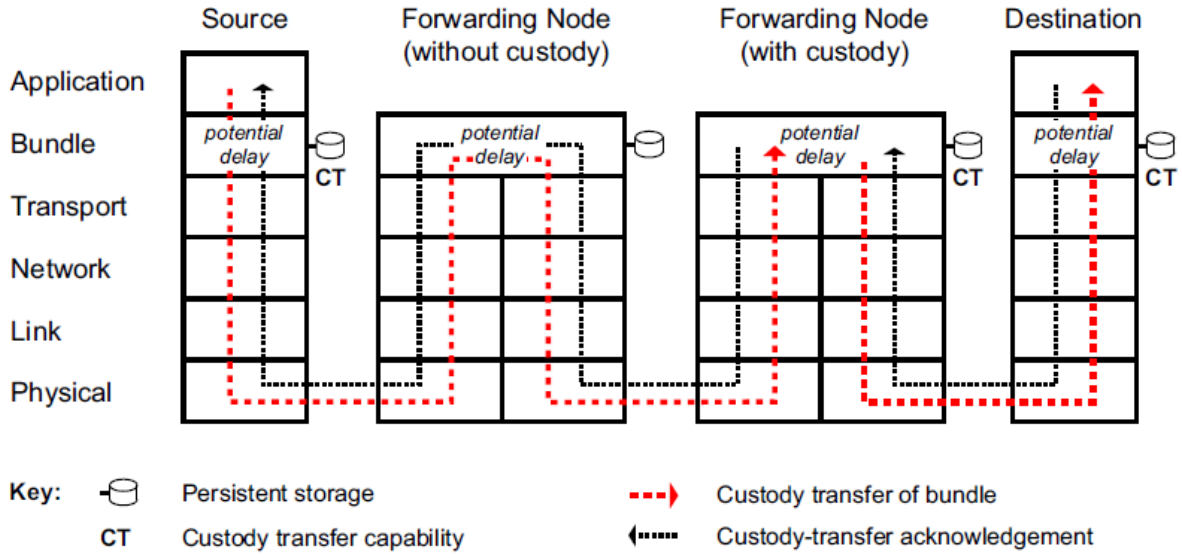


Figura 4.3. Trasferimento di custodia di un bundle in una DTN (Fonte: [4])

Tuttavia, il trasferimento di custodia dell'architettura DTN specifica solamente un meccanismo di ritrasmissione alla granularità di bundle. Resta dunque alle applicazioni la responsabilità di implementare il proprio meccanismo di affidabilità di consegna dei messaggi (ADU) end-to-end, utilizzando questi due strumenti messi a disposizione dal Bundle Layer.

Quando un'applicazione emette un bundle con la richiesta di custodia, dapprima il nodo sorgente e in seguito quelli successivi sono liberi di accettare o meno la richiesta; se un nodo la accetta diventa il "*custode*" del bundle, cioè il nodo che si impegna a far giungere il bundle stesso a destinazione. Per poter accettare la custodia del bundle, il nodo ricevente deve avere le necessarie risorse a disposizione;

inoltre, nel caso di utilizzo di algoritmi di routing basati su contatti schedulati o sulle statistiche di incontro dei nodi, sarebbe plausibile da parte del nodo accettare la custodia in base alla probabilità di favorire l'inoltro del bundle verso la destinazione.

Quando il custode corrente invia un bundle al nodo successivo, viene richiesto il trasferimento della custodia e avviato un timer di ritrasmissione. Se il custode corrente riceve dal nodo successivo un "*custody signal*", uno speciale bundle amministrativo di accettazione della custodia, prima dello scadere del timer significa che il nodo successivo ha accettato la custodia: quest'ultimo diventa il nuovo custode del bundle e il vecchio custode può rilasciarne la custodia liberando le risorse impiegate. In caso contrario, il custode corrente ritrasmette il bundle.

Come mostrato in Figura 4.3 il meccanismo di ritrasmissione non è strettamente hop-to-hop, in quanto uno o più nodi intermedi possono rifiutare la custodia inoltrando comunque il bundle a dei nodi successivi. In tal caso la ritrasmissione può coprire più hop, coinvolgendo due custodi successivi ma non immediatamente adiacenti.

Si noti che se l'applicazione non richiede il trasferimento con custodia, il successo della consegna del bundle dipende soltanto dall'affidabilità dei protocolli sottostanti il Bundle Layer.

4.3 Routing

A prima vista il problema del routing nelle DTN potrebbe apparire del tutto simile al problema del *routing dinamico* ma con numerosi collegamenti guasti per periodi di tempo più o meno lunghi. Ma non è così, poiché nel problema standard del routing dinamico si assume che la topologia sia *connessa* (o partizionata per brevi intervalli di tempo) e che la funzione obiettivo dell'algoritmo di routing sia trovare, ad un determinato istante, il miglior percorso end-to-end per instradare il traffico. In una DTN, invece, un percorso end-to-end potrebbe non esistere mai; lo scopo del routing è quello di riuscire, in un futuro istante, a consegnare un bundle sfruttando i nodi intermedi e le loro risorse di storage permanente.

Il problema del routing nelle DTN corrisponde ad un problema di ottimizzazione vincolata su un grafo dove gli archi potrebbero non essere disponibili per lunghi periodi di tempo e dove esiste un vincolo sulle risorse di storage di ogni singolo nodo. Questa formulazione rivela che il problema del routing nelle DTN è piuttosto diverso e presenta delle sfide maggiori.

4.3.1 Disponibilità delle informazioni topologiche

Ogni DTN può essere caratterizzata in base all'evoluzione topologica nel tempo. Possono esservi DTN che hanno una predicibilità totale della topologia rispetto al tempo, come nel caso di reti di sensori che comunicano a intervalli prestabiliti. In tal

caso gli algoritmi di routing avrebbero a loro disposizione le informazioni necessarie per risolvere il problema in modo *deterministico*.

Nella maggior parte dei casi però, le applicazioni reali di DTN hanno dei pattern evolutivi solo parzialmente prevedibili o addirittura del tutto imprevedibili. Ad esempio nel caso di una DTN che sfrutti il trasporto pubblico come data-mule, un algoritmo di routing potrebbe affrontare il problema utilizzando un *modello stocastico* che sfrutti la conoscenza dei passaggi previsti.

4.3.2 Principali algoritmi di routing per DTN

In questa sezione vengono discussi i più comuni e generici algoritmi di routing impiegati nelle reti DTN. Si tratta, inoltre, degli algoritmi disponibili nella maggior parte delle implementazioni software di DTN attualmente esistenti. Questa panoramica non rappresenta dunque una lista esaustiva, infatti in letteratura si trovano numerosi algoritmi sviluppati ed ottimizzati per particolari domini applicativi. A tal proposito è possibile consultare [17], [18] e [19] per una più ampia panoramica sullo stato dell'arte del routing nell'ambito delle DTN.

Trasmissione diretta e First Contact

La *trasmissione diretta*, il più triviale degli algoritmi, prevede che il nodo sorgente trasmetta un bundle esclusivamente al nodo di destinazione. Quindi, ad ogni istante di tempo, una sola copia del bundle è presente nella rete. Lo svantaggio principale di questo semplice approccio è che il nodo sorgente potrebbe dover attendere per un periodo piuttosto lungo e indefinito prima di poter trasmettere il bundle alla destinazione tramite un contatto diretto. La trasmissione diretta può essere utile in DTN dove il pattern di mobilità dei nodi può essere predetto in modo deterministico.

L'algoritmo *First Contact* permette di inoltrare un bundle attraverso un nodo scelto casualmente tra quelli correntemente raggiungibili. Quando un bundle viene inoltrato ad un nodo successivo, questo viene cancellato dal precedente. Potrebbe essere necessario l'attraversamento di diversi nodi prima che un bundle arrivi a destinazione. Inoltre, ogni bundle può mantenere la lista di hop attraversati in modo da non essere inoltrato verso un nodo da cui è transitato in precedenza. Le performance in termini di rapidità e di probabilità di consegna di un bundle non sono particolarmente migliori rispetto alla trasmissione diretta. Questo perché i nodi vicini, scelti in modo casuale, potrebbero non essere i migliori candidati per "trasportare" quel determinato bundle verso la destinazione.

Flooding

Nel caso di DTN in cui non sia possibile prevedere i futuri contatti tra nodi e la conseguente evoluzione nel tempo della topologia della rete stessa, la trasmissione

diretta e l'algoritmo First Contact non garantiscono buone prestazioni nella consegna dei bundle. Si tratta di due strategie accomunate dal fatto che non inondano la rete con repliche dello stesso bundle. Un approccio totalmente opposto, che massimizzi la probabilità di consegna dei bundle e la rapidità con cui questi raggiungono la destinazione a fronte di un overhead maggiore, è rappresentato dal *flooding*.

Ad ogni contatto con un vicino, un nodo trasmette quanti più bundle è possibile tra quelli da inoltrare[1]. Chiaramente, questo approccio impiega molte risorse in termini di storage; per ogni bundle generato sono presenti nella DTN molteplici copie del bundle stesso. Inoltre, non vengono sfruttati a pieno gli occasionali intervalli di trasmissione dato che si potrebbero trasmettere ad un nodo dei bundle di cui questo è già in possesso e che quindi scarterebbe.

Routing epidemico

Questo algoritmo è per natura basato sul flooding ma, a differenza di quest'ultimo, evita di inviare ad un nodo vicino dei bundle di cui questo sia già in possesso. In questo modo si cerca di mitigare l'overhead dovuto alla trasmissione di bundle che verrebbero scartati, sfruttando al meglio i contatti occasionali tra nodi.

Per realizzare questo, ogni nodo mantiene una hash table che indicizza tutti i bundle in memoria per mezzo del loro identificativo univoco (Si veda il formato dei Bundle alla Sezione 4.4). Inoltre ogni nodo mantiene un particolare vettore di bit, chiamato "*summary vector*", che indica quali bundle sono indicizzati dalla hash table locale.

Quando avviene un contatto tra due nodi, questi iniziano una fase di negoziazione in cui si scambiano i rispettivi summary vector. A questo punto, ciascuno dei nodi calcola un vettore di bit risultante dall'operazione di *AND logico* tra il proprio summary vector e quello del vicino. Questo vettore rappresenta i bundle in possesso del nodo vicino, ma non presenti nello storage locale. I due nodi quindi procedono nello scambio delle rispettive "*richieste*" ponendo termine alla fase di negoziazione. A questo punto i due nodi inviano al rispettivo vicino solamente i bundle richiesti.

Spray-and-Wait

Flooding e routing epidemico richiedono elevate risorse di memoria, specie in reti con un elevato numero di nodi, molto traffico generato e numerosi contatti. Per mitigare questo overhead causato da molteplici copie di bundle sparse per i vari nodi della DTN, è possibile adottare un approccio *Spray-and-Wait*.

Si tratta di un algoritmo che, nella sua variante più semplice, è composto da due fasi denominate rispettivamente "*spray*" e "*wait*". Ogni bundle che viene generato è

[1]Si intendono tutti i bundle attualmente memorizzati nel nodo, ovvero sia quelli generati dal nodo stesso che quelli da inoltrare per conto di nodi vicini

associato ad un numero massimo $\boldsymbol{L}$ di copie che possono essere presenti nella rete. Durante la fase di *spray*, la sorgente di un bundle è responsabile di trasmetterne una copia ad un massimo di $\boldsymbol{L}$ vicini distinti. Quando un nodo intermedio riceve una copia entra nella fase di *wait*: si comporta da *relay* ed inoltrerà il bundle solo alla destinazione finale nell'eventualità di un contatto con quest'ultima.

Rispetto al routing epidemico viene ridotto l'overhead dovuto alla duplicazione di bundle e la conseguente domanda di risorse ma, poiché la fase di spray è totalmente casuale, proprio per il numero inferiore di copie presenti nella rete si potrebbe avere come conseguenza una probabilità di consegna dei bundle inferiore.

PRoPHET

L'algoritmo denominato *Probabilistic Routing Protocol using History of Encounters and Transitivity* (PRoPHET) è un algoritmo basato su una metrica probabilistica chiamata "*delivery predictability*", che rappresenta la probabilità $P(a, b)$ per un nodo a di consegnare un messaggio ad un nodo b.

Ogni nodo a mantiene le suddette probabilità $P(a, b)$ per ogni destinazione b conosciuta. Nel caso di destinazioni sconosciute questa probabilità assume valore nullo. Ogni nodo utilizza un'algoritmo adattativo per aggiornare questa metrica ad ogni contatto con un'altro nodo. L'algoritmo funziona seguendo questi tre passi:

1. Quando un nodo a incontra un nodo b la delivery predictability $P(a, b)$ viene aggiornata:

$$P(a,b)_{new} = P(a,b)_{old} \; + \; (1 \; - \; P(a,b)_{old}) \, P_{init}, \; P_{init} \in [0,1]$$

   dove P_{init} è una costante di inizializzazione.

2. Le restanti probabilità per tutte le destinazioni conosciute d vengono invecchiate:

$$P(a,d)_{new} = P(a,d)_{old} \, \gamma^K, \; \gamma \in [0,1]$$

   dove γ è una costante di invecchiamento e K sono le unità di tempo trascorse dall'ultima operazione di invecchiamento.

3. Le delivery predictability di a e b vengono scambiate e la proprietà transitiva viene utilizzata per aggiornare le probabilità di consegna di a verso i nodi d per cui b ha $P(b, d)$:

$$P(a,d)_{new} = P(a,d)_{old} \; + \; (1 \; - \; P(a,d)_{old}) \; \cdot \; P(a,b) \; \cdot \; P(b,d) \; \cdot \beta, \; \beta \in [0,1]$$

   dove β è una costante che esprime quanto la proprietà transitiva incide sulla delivery predictability.

4.4 Formato di un Bundle

In un'architettura DTN, le applicazioni operano su messaggi trasportati in PDU di lunghezza variabile chiamati bundle. Un bundle è composto da un insieme di *blocchi* che possono contenere meta-dati e/o dati applicativi. I blocchi possono essere concatenati, uno dopo l'altro, in modo del tutto simile agli *extension header* di IPv6.

Il primo blocco di ciascun bundle prende il nome di "*Primary Block*" e, di fatto, ne rappresenta l'header. Tutti i blocchi che seguono quello primario sono espressi secondo il formato "*type-lenght-value*" (TLV) che garantisce l'estendibilità del protocollo e offre supporto alla sperimentazione. Infatti, alla ricezione di un bundle contenente un blocco di tipo sconosciuto un nodo DTN può processare il bundle saltando il blocco oppure può scartare l'intero bundle; si tratta di una decisione vincolata da alcuni flag all'interno di un blocco, denominati "*Block Processing Flags*".

4.4.1 Primary Block

Come detto, il primo blocco di ciascun bundle prende il nome di *Primary Block* ed illustrato in Figura 4.4. Questo blocco rappresenta l'header di un bundle e contiene dei meta-dati che rappresentano per le DTN l'equivalente di quelli presenti in un header IP: *versione*, *EID sorgente* ed *EID destinazione* del bundle, *lunghezza*, alcuni flag per il processamento del bundle e informazioni opzionali sulla frammentazione. Contiene anche altri campi addizionali, più caratteristici del Bundle Protocol e che vengono di seguito descritti:

- ***Report-to EID***: si tratta dell'EID del nodo al quale inviare i "*Bundle Status Report*", dei bundle amministrativi che notificano il nodo in questione dell'accadimento di un certo evento che coinvolge il bundle.
- ***EID del custode corrente***: indica il custode corrente del bundle e serve a supportare il meccanismo del trasferimento di custodia, come descritto alla Sezione 4.2.2.
- ***Timestamp di creazione*** e ***Sequence number***: il timestamp di creazione di un bundle corrisponde al momento in cui il Bundle Protocol Agent del nodo sorgente riceve la richiesta di trasmissione del bundle, che quindi viene creato. È un offset espresso in secondi a partire dall'inizio dell'anno 2000 nel fuso orario UTC. La tripletta formata dal timestamp di creazione, dall'EID del nodo sorgente e dal sequence number identifica in modo univoco un bundle. Il sequence number è l'ultimo valore di un contatore gestito dal Bundle Protocol Agent (BPA) del nodo sorgente e che genera una sequenza monotona crescente di interi. Questo contatore può essere reinizializzato a zero quando il tempo corrente avanza di un secondo.

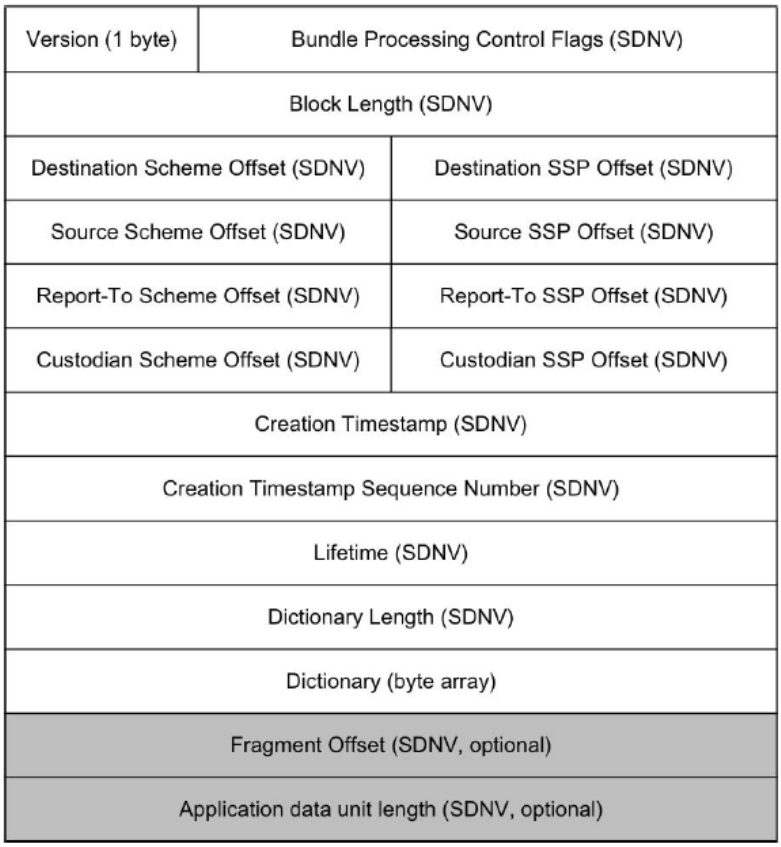


Figura 4.4. Formato del "*primary block*" di un Bundle (Fonte: [15])

- ***Lifetime***: è un offset espresso in secondi, a partire dal timestamp di creazione, per cui il bundle può essere considerato valido. Quindi, secondo una logica stabilita da chi amministra la DTN, se un bundle ha esaurito la sua validità può essere scartato mentre è in transito e cancellato dal BPA dei nodi che ne conservano una copia.

- ***Dictionary***: si tratta di un array di byte contenente tutti gli schemi (ossia `dtn`, `ipn`) ed SSP concatenati di tutti gli EID referenziati nel Primary Block ed eventualmente da campi contenuti in altri potenziali blocchi futuri del protocollo. Questi campi del Primary Block, infatti, non contengono direttamente gli EID ma il rispettivo offset nel dizionario.

La maggior parte dei campi sono di lunghezza variabile e seguono una notazione relativamente compatta denominata "*self-delimiting numerical values*" (SDNV) [20].

Si noti che il *timestamp di creazione* del bundle e conseguentemente il campo *lifetime* sono espressi utilizzando il tempo reale. Questo implica che ogni nodo della rete DTN debba essere sincronizzato, almeno grossolanamente. Si tratta di un aspetto di cui tenere conto e che è stato affrontato anche durante l'implementazione del proof-of-concept sviluppato per questa tesi.

Un altro aspetto particolare del Bundle Protocol è quello di non prevedere nessun campo per la verifica dell'integrità di un bundle. Si tratta di una scelta progettuale consapevole da parte del DTNRG, poichè il Bundle Protocol è stato pensato per due usi primari. Il primo, utilizzarlo al di sopra di uno dei protocolli di trasporto esistenti, che solitamente sono dotati di meccanismi di verifica dell'integrità. Alternativamente, per operare a livello rete andando di fatto a sostituirsi ad IP. In questo caso, per ragionamenti simili a quelli fatti per la definizione di IPv6, si lascia il processo di rilevamento e correzione dell'errore ai protocolli di livello superiore. Esiste comunque la possibilità di avvalersi dei cosidetti "*Security Blocks*", definiti dal "*Bundle Security Protocol*" [21], per assicurare integrità e/o confidenzialità ai bundle.

4.4.2 Blocchi di payload

Solitamente, le ADU provenienti dalle applicazioni vengono inseriti in un o più *blocchi di payload* che fanno seguito al blocco primario. Come tutti i blocchi successivi a quello primario, i blocchi di payload (Figura 4.5) seguono il formato TLV. Per cui il primo campo, di 1 byte, esprime il tipo: un blocco di payload è identificato dal valore "1". Sia il campo contenente i flag relativi al blocco che il campo che esprime la lunghezza rimanente del blocco (i.e. lunghezza del payload) sono espressi nel formato compatto SDNV.

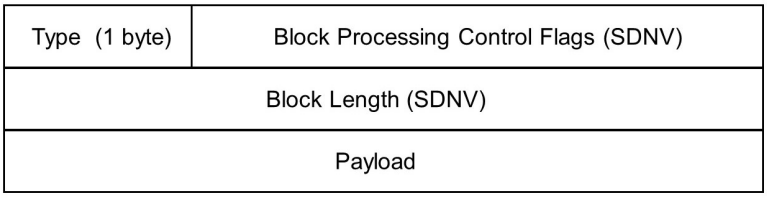


Figura 4.5. Formato di un blocco di payload all'interno di un Bundle

4.5 IBR-DTN

IBR-DTN è l'implementazione del Bundle Protocol utilizzata per lo sviluppo di questa tesi. È contraddistinta da leggerezza, efficienza ed elevata portabilità per

rivolgersi a sistemi embedded e piattaforme mobile. In tal senso, oltre all'implementazione per i tradizionali sistemi operativi, scritta in C++, è disponibile una versione per la piattaforma Android.

Nella versione per sistemi tradizionali, Il Bundle Protocol Agent è un *processo demone* ed espone un'interfaccia, basata su socket, con cui le applicazioni possono interagire. Le API sono esposte di default alla porta TCP 4550 sia in forma testuale che secondo un protocollo binario. Per lo sviluppo di questa tesi la scelta è ricaduta sulle API testuali, poiché queste in un fase di prototipazione sono molto semplici e convenienti da utilizzare e non vincolano all'uso di un linguaggio specifico. Il protocollo binario è da preferire in caso di sistemi embedded con risorse limitate; in tal caso è possibile utilizzare le librerie C++, fornite con i sorgenti, per interagire con il Bundle Layer. Per avere una panoramica delle API esposte si faccia riferimento alla documentazione ufficiale [22].

L'implementazione di IBR-DTN è organizzata in vari moduli software per aprire all'estensione delle proprie funzionalità in maniera semplice. Al momento offre *Convergence Layer Adapter* (CLA) per TCP, UDP, HTTP, IEEE 802.15.4 LoWPAN. È presente anche un'estensione del TCP-CLA per il supporto di TLS. Per quanto concerne il routing, IBR-DTN mette a disposizione diversi moduli per il supporto del routing statico, della trasmissione diretta e degli algoritmi *First Contact*, *Flooding*, *Epidemico* e *PRoPHET*. I moduli di storage invece permettono di configurare il Bundle Protocol Agent per l'utilizzo di tre diverse forme di memoria: memoria principale, file su disco o un database SQLite. È supportato il Bundle Security Protocol, nel caso in cui si necessiti di garantire integrità e/o riservatezza dei bundle.

Qualora si utilizzi IBR-DTN per implementare una "*rete sovrapposta*" su nodi IP, un ulteriore modulo implementa il protocollo *DTN Internet Protocol Neighbor Discovery* (IPND) [23] che permette di effettuare la scoperta di nuovi nodi vicini ed effettuarne il binding tra EID, che identifica un nodo nella DTN, e il rispettivo indirizzo IP. Tramite questo protocollo, il demone di IBR-DTN invia e contemporaneamente resta in ascolto di piccoli datagrammi UDP chiamati "*beacon*", che annunciano la presenza di un nodo a tutti quelli correntemente connessi nella topologia (es. a tutti i nodi nel range di copertura di una rete wireless ad-hoc). Questi beacon, inviati in multicast, contengono al loro interno l'EID, per favorire il binding con il rispettivo IP sorgente presente nell'intestazione IP/UDP, e opzionalmente i servizi offerti dal nodo. Nell'implementazione IBR-DTN sono supportati sia nodi IPv4 che IPv6.

4.5.1 Installazione da sorgenti, configurazione ed avvio

Al fine di avere una versione sempre allineata alle ultime modifiche, è possibile installare IBR-DTN e tutte le sue componenti direttamente dal repository GitHub

dell'autore. Queste istruzioni sono pensante per chi utilizza un ambiente GNU/Linux. In particolare i comandi sotto riportati sono validi per distribuzioni Debian e derivate (Ubuntu, Raspbian) con APT come packet manager, ma sono facilmente adattabili per altre distribuzioni. Per ambienti Windows è possibile scaricare una versione già compilata di IBR-DTN [24].

Innanzitutto è necessario preparare l'ambiente installando alcune librerie necessarie alla compilazione dei sorgenti:

```
$ sudo apt-get install build-essential libssl-dev \
    libz-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 \
    libarchive-dev
```

Dopo essersi posizionati su una cartella di lavoro è possibile clonare il repository del progetto e successivamente lanciare la fase di configurazione, compilazione ed installazione come segue:

```
$ git clone https://github.com/ibrdtn/ibrdtn.git \
    ibrdtn-repo
$ cd ibrdtn-repo/ibrdtn
$ bash autogen.sh
$ ./configure --with-sqlite
$ make
$ sudo make install
```

Si noti che al comando "`configure`" è stato passato il parametro "`-with-sqlite`". Questo fa si che venga installato il modulo opzionale che permette di utilizzare un database SQLite come storage per i bundle. In generale con l'opzione "`-with-module`" è possibile installare ogni modulo addizionale. Per gli scopi di questa tesi tuttavia è sufficiente l'installazione base con il solo modulo SQLite aggiuntivo.

Una volta eseguita l'installazione per avviare il demone IBR-DTN è sufficiente utilizzare il seguente comando:

```
$ dtnd -i interface
```

dove ad "`interface`" deve essere sostituita l'interfaccia di rete a cui collegarsi per uscire dal nodo (es. `eth0`). Se si desidera invece configurare i parametri del demone è possibile farlo specificando il seguente comando:

```
$ dtnd -c config_file
```

Sarà possibile, ad esempio, lanciare il demone su più di un'interfaccia di rete, cambiare algoritmo di routing, la modalità di memorizzazione dei bundle e molto altro.

Per una trattazione approfondita della configurazione del demone si può fare riferimento ai commenti disponibili nel file di configurazione di esempio fornito con i sorgenti. È possibile trovarlo al path relativo `ibrdtn/daemon/etc/ibrdtnd.conf`, dentro la cartella dei sorgenti.

Capitolo 5

Altre tecnologie utilizzate

Oltre alla piattaforma di Fog Computing di Nebbiolo Technologies e all'implementazione software IBR-DTN del Bundle Protocol, diversi altri strumenti software e tecnologie sono state utilizzate per la realizzazione del sistema di telemetria, la cui implementazione verrà discussa nel Capitolo 7. Prima di entrare nel dettaglio implementativo è dunque importante esaminare tali applicativi. Molte sezioni di questo capitolo sono fortemente ispirate ai siti e alle documentazioni degli stessi.

5.1 Docker

Docker è una delle piattaforme software più famose al mondo che permette di eliminare il problema "*sulla mia macchina funziona*" quando si collabora con altre persone sulla realizzazione di un applicativo. Sviluppatori, operatori ed aziende utilizzano docker per risolvere i problemi di distribuzione delle applicazioni. Il progetto è open source e si basa sull'utilizzo dei **container** [25].

5.1.1 Container

Un container è un pezzo di software che include al proprio interno tutto quello di cui ha bisogno per essere eseguito: codice, tool di sistema, librerie e settaggi. I software che vengono racchiusi in un container possono essere eseguiti sia su Linux che su Windows e MacOS e, nonostante ciò, funzioneranno sempre allo stesso modo. Sono quindi indipendenti dall'ambiente in cui vengono istanziati e questo riduce, come accennato prima, i problemi che nascono all'interno dei team quando si utilizza software differente sulla stessa infrastruttura.

L'idea dei container non è nuova, ma la si può vedere come una naturale evoluzione dei *chroot* e delle FreeBSD *jails* disponibili già da molto tempo [26]. In passato altre soluzioni commerciali, quali Virtuozzo/OpenVZ e LXC, avevano provato ad utilizzare questo tipo di tecnologia, ma senza molto successo. Bisogna fare

attenzione però: un container non è una macchina virtuale, anzi le due entità sono totalmente differenti (Figura 5.1).

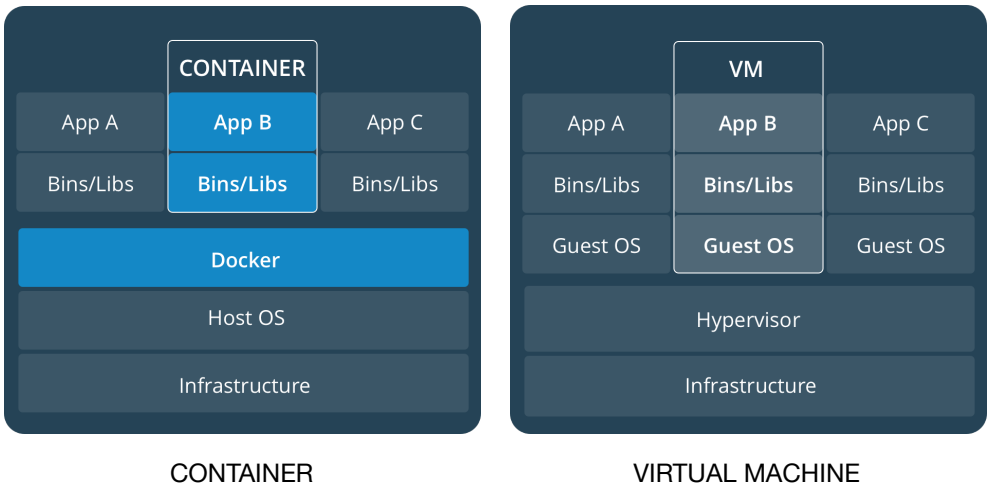


Figura 5.1. Differenze tra container e virtual machine (Fonte: Docker [25]).

Container vs Virtual Machine

Container e VM, seppur sembrino molto simili tra loro e sfruttino gli stessi benefici legati all'isolamento delle risorse, nella sostanza risultano essere molto diversi. Detto in termini semplici: un container virtualizza un sistema operativo, mentre una virtual machine l'hardware. Per questo motivo i primi sono molto più efficienti dei secondi.

Un container è un'astrazione del layer applicativo che combina in un'unica soluzione codice e dipendenze. Più container possono essere istanziati sulla stessa macchina e condividere contemporaneamente lo stesso kernel, ognuno di essi viene infatti eseguito come un singolo processo in *user space*. I container hanno bisogno ovviamente di minor spazio delle VM ed il loro avvio avviene quasi immediatamente.

Con Virtual Machine si intende un'astrazione fisica dell'hardware. C'è la necessità di avere un hypervisor che si occupi di gestire più VM in una sola macchina. Ogni VM include una copia **intera** del sistema operativo, una o più applicazioni e tutte le librerie necessarie, per un totale di decine di GB. Il boot è molto spesso lento e non paragonabile ai tempi di avvio di un container.

5.1.2 Networking

Un altro aspetto importante di Docker è quello legato al networking. Durante il primo avvio, vengono create automaticamente tre reti: *bridge*, *none* e *host* [27]. Nel momento in cui si manda in esecuzione un container, è possibile selezionare tramite il comando `-network` quale rete utilizzare.

- **bridge**: è la rete di default. Se non specificato diversamente, tutti i nuovi container vengono collegati a questa rete. È associata all'interfaccia virtuale `docker0` e gli IP assegnati sono del tipo 172.17.0.1/16. Conoscendo l'IP di un altro container, è possibile comunicare con esso.

- **none**: quando un container viene collegato a questa rete non riceve alcun indirizzo IP se non quello di loopback. Non può accedere alla rete esterna e a nessun altro container. In pratica viene totalmente isolato.

- **host**: permette ai container ad essa collegati di condividere tutto lo stack network dell'host. Tutte le interfacce dell'host saranno quindi visibili dal container.

- **custom**: è possibile creare delle nuove reti, assegnandone nomi e range di indirizzi personalizzati. I container che vengono collegati alle reti *personalizzate* possono comunicare tra loro conoscendo semplicemente il nome del container, non servono in questo caso gli indirizzi IP. Questo è merito del server DNS interno che il demone di Docker lancia alla creazione della rete definita dall'utente. Se la risoluzione dei nomi non va a buon fine con il server DNS locale, allora vengono contattati direttamente i server DNS esterni.

Per la mappatura delle porte verso l'esterno esistono due keywords, `EXPOSE` e `PUBLISH`:

- **EXPOSE**: serve ai fini di documentazione. L'utilizzo di questa keyword è pertanto opzionale e si inserisce del Dockerfile (file utile per la creazione di un container) per indicare a chi manderà in esecuzione il container quali porte pubblicare verso l'esterno.

- **PUBLISH**: utilizzando "publish" è possibile demandare direttamente a Docker l'apertura delle porte richieste nel momento in cui viene istanziato il container. Generalmente Docker sceglie una porta a random con un valore maggiore di 3000. È possibile richiedere il mapping con una porta specifica dell'host, ma in questo caso bisogna fare attenzione perché tale porta potrebbe essere già occupata da qualche altro processo.

5.2 RabbitMQ

RabbitMQ è un message broker open-source che implementa il protocollo *Advanced Message Queue Protocol (AMQP)* per lo scambio di dati tra processi, applicazioni e server. Scritto in Erlang, si basa sul framework *Open Telecom Platform (OTP)* e fornisce librerie client per diversi linguaggi di programmazione [28].

All'interno di RabbitMQ possono essere definite diverse **code**, alle quali le applicazioni hanno la capacità di connettersi per scambiare messaggi. Un **messaggio** può includere qualsiasi tipo di informazione per poter sincronizzare due o più applicazioni. Questi non vengono direttamente spediti nelle code, ma devono essere gestiti dagli **exchange**. Un exchange si occupa del routing dei messaggi: dopo averli prelevati dall'applicazione producer, li smista all'interno delle code grazie all'aiuto di vari attributi contenuti al loro interno, come le *routing key*. Ecco come funziona il flusso di messaggi con RabbitMQ:

1. Il produttore pubblica il messaggio verso un exchange (ne esistono diverse tipologie, si veda la Sezione 5.2.1).

2. L'exchange, non appena ricevuto il messaggio, ha la responsabilità di gestirne il routing e può farlo sfruttando vari attributi contenuti al suo interno, come la routing key, a seconda del tipo di exchange.

3. Exchange e code devono essere associati da un *binding*. In Figura 5.2 è possibile vedere come uno stesso exchange può essere associato a più code. Questo non è un problema: gli attributi interni al messaggio permettono all'exchange di sapere a quale coda inoltrare i messaggi senza commettere errori. Se un messaggio soddisfa più di un binding, questo viene inviato in copia a tutte quelle code che ne soddisfano i criteri.

4. I messaggi restano in una coda fin quando non vengono "prelevati" da un consumatore.

5. Il consumatore adesso può finalmente gestire il messaggio ricevuto.

5.2.1 Tipologie di Exchange

Come accennato precedentemente, esistono diverse tipologie di exchange ed i client possono utilizzare quelli predefiniti o crearne di nuovi. Le varie tipologie sono:

- **direct**: i messaggi vengono consegnati alle code in base alla routing key specificata nel messaggio. Un exchange di tipo direct consegna i messaggi a tutte

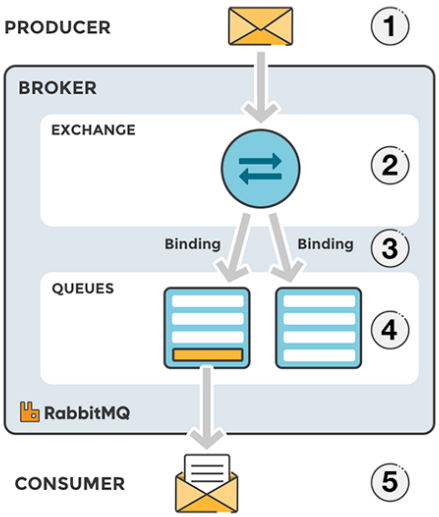


Figura 5.2. Flusso di messaggi in RabbitMQ (Fonte: CloudAMQP [28]).

quelle code che hanno una *binding key*[1], verso quel exchange, esattamente uguale alla routing key contenuta nel messaggio. Su RabbitMQ esiste un exchange predefinito di tipo direct, che prende il nome di `amq.direct`.

- **default**: è un exchange predefinito di tipo direct senza alcun nome, cui generalmente si fa riferimento con una stringa vuota. Un messaggio che viene inviato a questo exchange verrà consegnato alla coda il cui nome è uguale a quello della routing key. Si tratta di un binding implicito, in quanto tutte le code vengono automaticamente collegate a questo exchange.

- **topic**: effettua il routing in base alle parole chiave inserite nella routing key e nelle binding key. La routing key deve essere una lista di parole separate da un punto; al suo interno possono essere specificate anche delle *wildcard*, che si esprimono con i simboli "*" e "#". L'asterisco specifica che in quella determinata posizione va bene qualsiasi parola presente nella binding key. Ad esempio: routing key = `example.*.*.b.*` – in questo caso il messaggio verrà consegnato a tutte quelle code la cui binding key contiene esattamente 5 parole, in cui la parola in prima posizione è "`example`" e quella in quarta posizione è "`b`". Il cancelletto "#", invece, è una wildcard che comprende una o più

[1]La *binding key* è una stringa di testo che lega una coda ad un determinato exchange e viene indicata in fase di definizione di un binding.

posizioni. Si consideri ad esempio la routing key "`polito.it.#`" , tutte le binding key che iniziano per "`polito.it`" soddisferanno il matching.

- **fanout**: questo exchange copia e inoltra il messaggio ricevuto a tutte le code ad esso collegato. La routing key viene semplicemente ignorata, tutte le code connesse vanno bene. Non ha quindi bisogno di alcun meccanismo di matching e quello che viene scritto nella routing key dal produttore non ha nessuna valenza. Questo exchange risulta essere molto utile e veloce nel caso in cui lo stesso messaggio deve essere trasmesso a una o più code nelle quali i consumatori processano i dati ricevuti in maniera indipendente. Ne è presente uno predefinito che prende il nome di `amq.fanout`.

- **headers**: questa tipologia è molto simile a quella del *topic*, ma sfrutta gli argomenti presenti sull'header del messaggio invece della routing key. Un messaggio viene inoltrato in una determinata coda se gli argomenti dell'header combaciano con quelli del binding. È quindi possibile avere più argomenti. Uno in particolare, che prende il nome di `x-match`, indica se a combaciare siano tutti gli argomenti o va bene anche soltanto uno di essi. Può avere due valori differenti: *all* (di default) oppure *any*. Se nulla viene specificato, allora tutti gli argomenti saranno obbligatori; con la presenza di *any* ne basta soltanto uno per dare l'ok al sistema di matching. Anche in questo caso è possibile trovarne uno predefinito: `amq.headers`.

- **dead letter**: in tutti i casi precedenti, se nessuna coda rispetta i requisiti richiesti, allora il messaggio viene scartato silenziosamente. Ci sono degli scenari in cui questo non va bene, per tale ragione RabbitMQ fornisce quest'ultimo tipo di exchange che si occupa di recuperare i messaggi che non sono stati consegnati a nessuno (detti anche "*lettere morte*").

5.2.2 Sistema di gestione

Il sistema di gestione, chiamato *RabbitMQ management*, fornisce un'interfaccia user-friendly che permette di monitorare e gestire il server RabbitMQ direttamente da un browser web. Code, connessioni, exchange, canali e molto altro possono essere gestiti direttamente da qui. È possibile anche monitorare il rapporto di messaggi inviati e ricevuti oppure scambiare messaggi manualmente attraverso l'interfaccia web. Il RabbitMQ management è di fatto un plugin che sfrutta le API HTTP di RabbitMQ fornendo informazioni fondamentali in caso di debug o controllo del sistema.

Nella schermata principale sono presenti due grafici che mostrano in tempo reale il numero di messaggi in coda e la velocità con cui questi arrivano. Sono presenti anche altri campi che danno informazioni più dettagliate:

- **ready**: indica il numero di tutti i messaggi che sono pronti per essere consegnati nelle varie code.
- **unacked**: i messaggi, nel momento in cui vengono inviati dal server, devono essere "confermati" tramite l'invio di un *ack*. Non possono essere cancellati dalla memoria di RabbitMQ fino a quando l'ack non è stato ricevuto.
- **total**: la somma dei due campi precedenti.

Un'importante sezione è quella che riguarda le connessioni e i canali.

Connessione e canale

Per *connessione* si intende una tradizionale connessione TCP tra l'applicazione client e RabbitMQ; il termine *canale* invece indica un collegamento virtuale che si instaura all'interno di una connessione. Connessioni e canali possono trovarsi in diversi stati: *starting*, *tuning*, *opening*, *running*, *flow*, *blocking*, *blocked*, *closing*, *closed*. Ad esempio se un client sta trasmettendo troppi dati e supera spesso la soglia massima, allora lo stato della connessione in questo caso diventa *flow* e RabbitMQ si occupa in qualche modo di limitarlo.

Controllo del flusso

A partire dalla versione 3.3 di RabbitMQ è possibile identificare i colli di bottiglia e prendere delle contromisure per evitare le congestioni delle code. Una prima versione del controllo del flusso era presente già dalla versione 2.8 ed inseriva nello stato *flow* tutte le connessioni che stavano generando troppi messaggi. Il problema principale è che, all'interno della stessa connessione TCP, il client può inviare messaggi che finiscono su code diverse ed è quindi impossibile sapere quale sia la coda troppo piena. Per questo motivo, con le nuove versioni, è stata effettuata la scelta di separare la connessione dal canale [29]. Gli scenari possibili sono:

1. La connessione è in stato di *flow*, ma nessuno dei suoi canali lo è: questo significa che uno o più canali sono la causa della congestione. Probabilmente troppi e piccoli messaggi arrivano al server e la CPU è impegnata con il meccanismo del routing.
2. La connessione è in stato di *flow* e alcuni suoi canali lo sono, ma nessuna coda sta pubblicando: anche in questo caso uno o più canali sono la causa. Le situazioni possono essere due: o il server è a corto di CPU per via del routing oppure ci sono troppe operazioni di I/O su disco. Succede spesso se si spediscono moltissimi piccoli messaggi persistenti.

3. La connessione, alcuni canali e alcune code sono nello stato di *flow*: il message store è la causa della congestione. La velocità del disco del server non basta per scrivere i messaggi. È il tipico caso di messaggi molto grandi e persistenti.

Per merito della distizione tra canale e connessione è adesso possibile trovare e risolvere le cause di congestione. In aggiunta è presente un ulteriore strumento, il *consumer utilisation*, che permette di capire se e quando il client, a causa di una congestione di rete, non riesce a ricevere tutti i messaggi ad esso diretti.

5.3 Reti wireless peer-to-peer

In questa sezione vengono descritte le basi del *Wi-Fi Direct* (*P2P*) e della modalità *IBSS* dello standard *IEEE 802.11*. Queste due tecnologie permettono di instaurare una rete Wi-Fi, senza l'utilizzo di un tradizionale Access Point. Ne vengono dunque messi a confronto vantaggi e svantaggi che hanno orientato la scelta verso l'impiego della modalità IBSS al fine di realizzare il sistema di comunicazione proposto in questa tesi.

5.3.1 Wi-Fi Direct

Wi-Fi Direct non è uno standard IEEE, ma si tratta del nome commerciale di una specifica tecnica definita dalla *Wi-Fi Alliance* e conosciuta come "*Wi-Fi Peer-to-Peer (P2P) Specification*" [30]. Permette ai moderni dispositivi Wi-Fi di connettersi tra di loro e di formare gruppi, solitamente *uno-a-uno* ma anche *uno-a-molti*. I dispositivi certificati Wi-Fi Direct negoziano il proprio ruolo con gli altri: uno di questi viene eletto *Group Owner* (GO), che rappresenta l'equivalente di un Access Point in una Wi-Fi tradizionale, gli altri, inclusi i dispositivi *legacy*, si connettono al GO in modalità *Station*.

I dispositivi Wi-Fi Direct utilizzano l'indirizzo MAC della NIC – "*globally unique*", ovvero l'indirizzo fisico assegnato dal produttore – come identificativo durante le fasi di *discovery* e di *negoziazione*. Diversamente, definiscono ed utilizzano un "*P2P Interface Address*", ovvero un indirizzo MAC locale – "*locally administered*", per tutti i frame inviati al GO o ad un Client all'interno di uno stesso gruppo. Gli *Action Frames* e gli *Information Elements* (IE), degli speciali frame IEEE 802.11, possono essere utilizzati per trasportare i dettagli del protocollo.

I dispositivi P2P alternativamente trasmettono e restano in ascolto per delle *Probe Request* sui cosiddetti *social channels*, ovvero i canali 1, 6 e 11 nella banda dei 2,4Ghz. I dispositivi non connessi e gli eventuali GO (di gruppi già formati) rispondono con delle *Probe Response* e degli Information Elements addizionali che descrivono le caratteristiche del dispositivo o del gruppo. I GO rispondono anche per conto dei dispositivi che fanno parte del loro gruppo, ma i client possono decidere

di non essere "visibili" mentre sono connessi ad un gruppo. Le caratteristiche di un dispositivo includono:

- Nome del dispositivo definito dall'utente;
- Tipologia del dispositivo (es. "Media Server", "Smartphone", "Stampante");
- Metodi di configurazione WPS supportati (*PIN* o *PushButton*);
- Supporto alla connessione da parte di altri dispositivi;
- Supporto al *Service Discovery*

Le caratteristiche di un gruppo includono:

- Numero massimo di dispositivi che possono prendere parte al gruppo (*group-limit*);
- Supporto allo scambio di dati tra client (comunicazioni *intra-BSS*), ovvero se un dispositivo qualunque del gruppo può contattare qualunque altro dispositivo;
- Supporto alla *cross-connection*, ovvero se il GO permette la connessione ad un'altra WLAN o l'accesso ad Internet;
- Persistenza del gruppo

Quando un dispositivo P2P viene scoperto, e prima di instaurarci un gruppo, può essergli richiesto di descrivere i servizi che può offrire. Si tratta di uno scambio di frame opzionali che supporta differenti protocolli per la descrizione di servizi, come "*Bonjour*" (*DNS-SD*), *UPnP* o *WS-Discovery*. Un'applicazione o un utente può scegliere di connettersi a un dispositivo in base al suo nome o ai servizi offerti (es. una fotocamera può connettersi automaticamente ad un display per mostrare le foto).

Se un dispositivo desidera connettersi ad un altro può inviare una *P2P Invitation Request*, ma i ruoli all'interno del gruppo devono essere negoziati. Lo scopo principale della negoziazione è quello di determinare chi dei due dispositivi partecipanti diventerà il GO e di scambiare alcune informazioni, come il canale su cui trasmettere o il metodo di configurazione WPS – *Wi-Fi Protected Setup*. Ogni dispositivo può comunicare la propria intenzione di diventare un GO esprimendo un attributo numerico (0-15, con 15 massima priorità a diventare GO), chiamato *GO Intent*, all'interno dei frame *GO Negotiation Request/Response*.

Quando la fase di negoziazione termina, viene inviata una *GO Negotiation confirmation* ed entrambi i dispositivi si spostano sul canale scelto. Il GO inizia ad operare in modalità AP, inviando *Beacon Frame* con l'SSID negoziato e con il *group*

formation bit impostato ad 1, poiché la formazione del gruppo non è del tutto completata. L'SSID è standardizzato come "`DIRECT-xy...`", dove `xy` sono caratteri alfanumerici casuali e può essere presente qualsiasi suffisso.

A questo punto inizia la fase di *provisioning*, dove il dispositivo client stabilisce una connessione sicura con il GO attraverso il protocollo WPS. Per instaurare la connessione, normalmente l'utente deve inserire un codice PIN o premere un bottone sul dispositivo.

Successivamente avviene il normale *4-way handshake* del protocollo *RSN* (*WPA2*) per lo scambio delle chiavi di cifratura, dove il GO assume il ruolo di *autenticatore* e il client quello di *supplicant*. Infine, il client richiederà un indirizzo al GO che da specifica deve implementare un DHCP server. A questo punto può avvenire lo scambio dati.

Da specifica la formazione di un gruppo dovrebbe impiegare meno di 15 secondi, ma poiché normalmente viene richiesto l'intervento dell'utente per instaurare una connessione sicura tra i dispositivi (WPS), potrebbe richiedere più tempo.

Per evitare che gli utenti debbano eseguire un'interazione ogni qual volta un gruppo viene formato tra gli stessi dispositivi, il gruppo può essere impostato come *persistente*. In tal caso, i dispositivi memorizzano le credenziali e possono riconnettersi automaticamente attraverso lo scambio di *Provision Discovery Request/Response* preventivamente alla fase di negoziazione.

Se il GO abbandona il gruppo, tutti i client vengono disconnessi ed un nuovo gruppo deve essere eventualmente negoziato tra i dispositivi vicini che intendono continuare a scambiarsi dati.

5.3.2 IEEE 802.11 IBSS (Ad-hoc)

Una rete *Ad-Hoc* è ufficialmente chiamata *IBSS* (*Independent Basic Service Set*) nello standard IEEE 802.11. Tutti i dispositivi che supportano lo standard devono implementare questa modalità. Il meccanismo di funzionamento è molto semplice e permette di creare una vera rete *peer-to-peer*, senza alcun tipo di gerarchia.

Similarmente ad un Access Point, un nodo IBSS invia regolarmente dei *Beacon Frame* contenenti il nome della rete (*SSID*), un identificativo della rete (*BSSID*) e un timestamp (*TSF*). I dispositivi facenti parte della stessa IBSS si suddividono il compito di inviare i beacon: se un nodo lo ha già ricevuto da un altro dispositivo all'interno del cosidetto *Beacon Period*, evita di inviarne un altro in quel intervallo di tempo. Per dare l'opportunità a tutte le stazioni di inviare i beacon, gli intervalli vengono determinati per mezzo di un'algoritmo di back-off casuale con distribuzione uniforme.

Per un'interfaccia di rete configurata in modalità Ad-Hoc deve essere specificato l'SSID della IBSS, nonché il canale radio. Quando l'interfaccia viene attivata effettuerà un scansione sul canale prestabilito per rilevare la presenza di altri nodi

facenti parte della stessa IBSS. La scansione può essere passiva, ascoltando i beacon inviati da altri nodi, oppure attiva, inviando una Probe Request. Se viene rilevata la presenza della rete Ad-Hoc desiderata, il dispositivo imposta il BSSID contenuto nei beacon o nella Probe Response ricevuta e può iniziare immediatamente a scambiare frame di dati con tutti i nodi IBSS all'interno del proprio range di copertura. In caso contrario sceglie un BSSID casuale ed inizia la fase di beaconing.

C'è solo un caso in cui la connessione a livello fisico non è immediata, ovvero quando due o più nodi IBSS che condividono lo stesso SSID e lo stesso canale hanno preventivamente scelto due differenti BSSID. Questo può accadere quando dei dispositivi precedentemente lontani entrano improvvisamente nel reciproco range di copertura. In questi casi avviene la cosidetta fase di *IBSS merge* o *IBSS coalescing*: deve essere scelto lo stesso BSSID da tutti i nodi, in particolare quello della IBSS più "vecchia". Per stabilire l'anzianità di una rete Ad-Hoc si utilizza il valore del campo TSF, contenuto nei beacon scambiati. Lo standard 802.11 descrive l'algoritmo di sincronizzazione dei nodi attraverso il campo TSF.

Per quanto concerne la sicurezza, lo standard non specifica alcun requisito per la modalità IBSS. Di conseguenza le reti Ad-Hoc possono scegliere tra nessuna cifratura, WEP, WPA o RSN (WPA2). Quindi anche le reti Ad-Hoc, se configurate per utilizzare le *pre-shared keys* del WPA2, possono garantire lo stesso livello di sicurezza delle reti Wi-Fi in modalità infrastruttura e del Wi-Fi Direct. Tuttavia il supporto per WPA2 è stato introdotto solo recentemente in Linux dal client *wpa_supplicant* a partire dalla versione 2.0. Non è raro trovare delle reti Ad-Hoc *aperte*, che lasciano il compito di implementare la cifratura ai livelli superiori (es. *OpenVPN* o protocollo applicativo) quando desiderato.

5.3.3 Confronto tra Wi-Fi Direct e modalità Ad-hoc

Per definizione i nodi partecipanti ad una rete *peer-to-peer* possiedono gli stessi privilegi, offrono le stesse funzionalità e non sono organizzati in gerarchie. Nonostante il Wi-Fi Direct sia definito anche come Wi-Fi P2P, non rispecchia le caratteristiche di una reale tecnologia peer-to-peer. Si tratta essenzialmente di un protocollo ideato per formare dei gruppi gerarchici e principalmente utilizzato per connettere pochi dispositivi. Questi sono effettivamente dei peer fino al momento del provisioning, poiché negoziano il loro ruolo nel gruppo da formare, ma dopo questa fase seguono le gerarchie *AP/Client – master/slave*. È dunque una tecnologia per instaurare una connessione temporanea tra pochi dispositivi in modo semplice e sicuro per l'utente.

Ma è una soluzione complessa e poco versatile, non utilizzabile per instaurare rapidamente una connessione senza l'intervento umano a causa della procedura WPS (PIN, Push Button, NFC etc.), dunque non scalabile con il numero di dispositivi da connettere. Il Wi-Fi Direct non è inoltre adatto a scenari caratterizzati da elevata mobilità dei nodi e frequenti cambi topologici, in quanto è necessario procedere

alla formazione di un nuovo gruppo quando un GO non è più raggiungibile, con conseguente perdita delle sessioni in corso.

Dall'altra parte la modalità Ad-Hoc rappresenta una vera soluzione peer-to-peer senza gerarchie, dove tutti i partecipanti sono uguali. Questo apre ad un vasto spettro di possibilità e modalità di comunicazione in cui una IBSS può rappresentare la base per la realizzazione di reti opportunistiche, DTN e Wireless Mesh Networks, estremamente utili in scenari dove realizzare un'infrastruttura di rete non è una soluzione praticabile. La modalità IBSS è piuttosto semplice ed estremamente versatile, ma necessita di essere coadiuvata da protocolli di livello superiore che implementino funzionalità come l'allocazione di indirizzi IP, il service discovery, la cifratura e l'inoltro multi-hop.

5.4 L'orchestratore Universal Node

Lo *Universal Node* nasce nell'ambito del progetto europeo *UNIFY* ed è stato sviluppato dal gruppo ricerca *Netgroup* del Politecnico di Torino [31]. Può essere considerato una sorta di "*datacenter in una scatola*", che fornisce funzionalità simili ad un cluster *OpenStack* ma sfruttando una singola macchina. In breve, lo Universal Node riceve la descrizione di un servizio di rete e provvede all'allocazione e all'orchestrazione di risorse computazionali e di rete per implementare il servizio richiesto. L'orchestratore gestisce l'intero ciclo di vita di nodi di computing autocontenuti (es. VM, Docker, processi DPDK) e primitive di rete (es. regole OpenFlow, istanze di switch software). Grazie alle sue peculiari caratteristiche può essere eseguito sia su un server tradizionale (es. una workstation) che su un dispositivo dalle risorse limitate, come un *home gateway* residenziale.

Lo Universal Node riceve i comandi attraverso delle REST API che seguono il formalismo dei *Network Functions Forwarding Graph* (NF-FG) e provvede ad implementarli sul nodo fisico. Questo formalismo esprime un servizio da eseguire in termini di Virtual Network Functions (VNF) e interconnessioni tra loro. Sono espressi in formato JSON di cui, per ulteriori dettagli riguardo la sintassi, si rimanda alla documentazione ufficiale che illustra vari esempi di NF-FG [32].

Quando lo Universal Node riceve il comando di istanziare un nuovo NF-FG, provvede all'espletamento delle seguenti operazioni:

- recupera dal datastore le immagini più appropriate per tutte le VNF che compongono il grafo;
- configura lo switch virtuale (vSwitch) per creare una nuova istanza di switch logico (LSI) e ne configura le porte richieste per connetterle alle VNF da istanziare;
- istanzia e inizializza tutte le VNF che compongono il grafo;

- traduce le regole per direzionare il traffico in messaggi *FlowMod* di OpenFlow da inviare allo switch virtuale.

Similarmente, lo Universal Node provvede ad aggiornare o eliminare un grafo, quando riceve il messaggio appropriato.

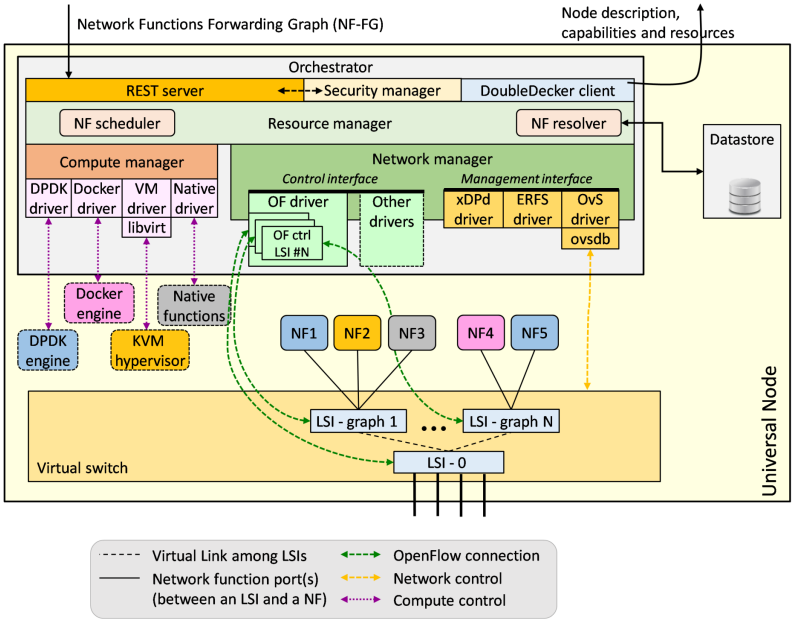


Figura 5.3. Architettura di alto livello dello Universal Node (Fonte: Universal Node public repository [31]).

In Figura 5.3 è mostrata l'architettura di alto livello dello Universal Node. È composto da diversi moduli, tra cui i principali sono il *network controller* e il *compute controller*.

5.4.1 Il network controller

Il *network controller* è il modulo che si occupa di interagire con lo switch virtuale (vSwitch). Consiste a sua volta di due parti:

- il ***controller OpenFlow***: un nuovo controller OpenFlow viene creato per ogni switch logico (LSI), che direziona il traffico attraverso le proprie porte;
- lo ***switch manager***: crea/distrugge gli LSI, le porte virtuali ed altro. In pratica, permette all'orchestratore di interagire con lo switch virtuale per effettuare le operazioni di gestione.

Allo stato attuale l'orchestratore supporta OpenvSwitch (OvS), l'extensible Data Path daemon (xDPd) e l'Ericsson Research Flow Switch (ERFS) come implementazioni di vSwitch.

Si noti che, come illustrato in Figura 5.3, più LSI possono essere istanziati sullo Universal Node. In particolare, durante la fase di inizializzazione il network controller crea un primo switch logico (LSI-0), che è connesso alle interfacce fisiche del nodo su cui è in esecuzione e che sarà connesso agli altri LSI. Ognuno di questi LSI aggiuntivi corrisponde agli NF-FG istanziati ed è dunque connesso alle VNF che compongono il relativo NF-FG; ne direziona il traffico attraverso le VNF come richiesto dalla descrizione del grafo stesso. Invece, lo switch LSI-0, essendo l'unico connesso alle interfacce fisiche del nodo e a tutti gli NF-FG, invia il traffico entrante nel nodo fisico al NF-FG appropriato e gestisce i pacchetti già processati in un grafo.

5.4.2 Il compute controller

Il *compute controller* è il modulo che interagisce con l'ambiente di esecuzione virtuale (es. l'*hypervisor*) per gestire il ciclo di vita delle VNF, includendo anche le operazioni necessarie a collegare le porte già create sullo switch virtuale alle VNF stesse.

Attualmente, lo Universal Node supporta VNF implementate come macchine virtuali (KVM), container Docker, processi DPDK e in un formato nativo, anche se solo un sottoinsieme di queste potrebbe essere disponibile a seconda dell'implementazione di vSwitch selezionata. In particolare per lo sviluppo dei test di validazione, descritti al Capitolo 8, è stata utilizzata l'accoppiata OpenvSwitch-Docker.

Capitolo 6

Architettura del sistema di comunicazione

In questo capitolo viene presentata l'architettura del sistema di comunicazione proposto per affrontare le criticità descritte nel Capitolo 2. Utilizzando quest'architettura è stato realizzato il prototipo di un sistema di telemetria, illustrato alla fine di questo capitolo.

6.1 Soluzione proposta

Nell'architettura proposta sia le macchine operatrici che i nodi di Fog Computing sono nodi di una DTN. Grazie al paradigma di inoltro *store-carry-and-forward* implementato dalla DTN, quest'architettura consente oltre ad una maggiore affidabilità anche lo sfruttamento della mobilità delle macchine operatrici e dei loro occasionali incontri per favorire il flusso di informazioni verso la destinazione.

Uno degli obiettivi dell'architettura è quello di essere trasparente nei confronti delle applicazioni esistenti. Queste applicazioni utilizzano protocolli di messaggistica basati sul paradigma publish/subscribe, ampiamente diffusi nell'ambito della comunicazione *machine-to-machine*. In particolare, per lo sviluppo di questa tesi l'attenzione è stata focalizzata sul protocollo MQTT. Per soddisfare il requisito di trasparenza verso le applicazioni, su ciascun nodo è presente un *gateway* che permette di veicolare i messaggi MQTT attraverso la DTN.

Normalmente un client MQTT, sia che abbia il ruolo di *publisher* che di *subscriber*, ha la necessità di essere direttamente connesso ad un server MQTT – o *message broker* – per poter pubblicare o ricevere messaggi appartenenti ad un certo *topic*. Nella soluzione proposta questo vincolo viene rilassato sfruttando le caratteristiche della DTN .

In Figura 6.1 è illustrato un esempio di come l'architettura proposta permetta la comunicazione tra un macchina operatrice in campo e un nodo di Fog Computing,

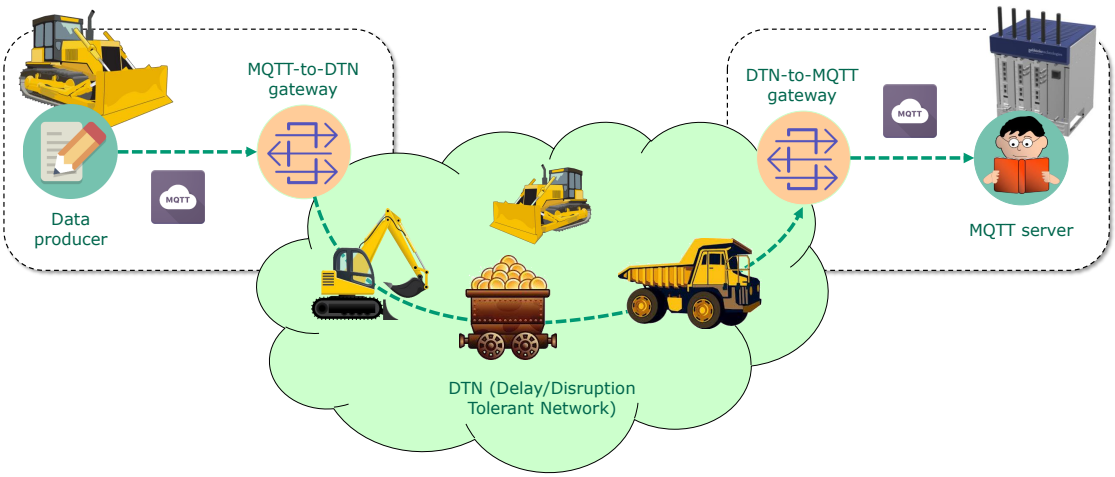


Figura 6.1. Esempio di comunicazione tra nodi non direttamente connessi: una coppia di gateway con funzioni speculari permette ai messaggi MQTT di essere veicolati attraverso la DTN

nonostante questi non siano direttamente connessi. In questo esempio la macchina operatrice produce dei dati da pubblicare, tramite il protocollo MQTT, sul message broker presente all'interno del nodo Fog. Un *gateway MQTT-to-DTN*, istanziato all'interno della stessa macchina, intercetta il messaggio MQTT e lo incapsula in un bundle che può essere veicolato attraverso la DTN fino a giungere al nodo Fog. Al suo interno è istanziato un gateway DTN-to-MQTT che riceve il bundle, ne estrae il messaggio MQTT incapsulato al suo interno e lo pubblica sul message broker.

6.2 Architettura generale

In questa sezione viene illustrata nel dettaglio l'architettura del sistema di comunicazione nella sua duplice funzionalità: comunicazione in upstream (*Device-to-Fog*) e comunicazione in downstream (*Fog-to-Device*).

6.2.1 Comunicazione in upstream

I nodi mobili sono stati denominati "*Sensing Edge Device*" per sottolineare come la sensoristica con cui sono equipaggiati sia fonte di informazioni che devono essere raccolte e veicolate verso i nodi di Fog Computing dislocati in campo, che si trovano appunto "*all'edge della rete*".

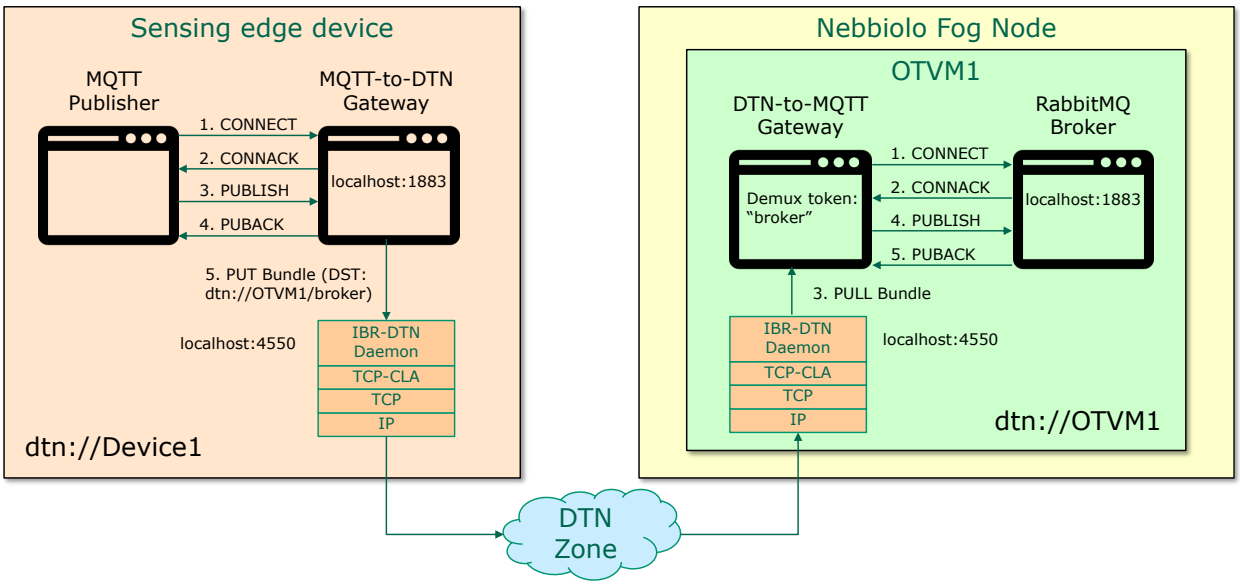


Figura 6.2. Architettura del sistema di comunicazione: Upstream

In Figura 6.2 è illustrata l'architettura del sistema nel caso della comunicazione in *upstream*, ovvero da un Sensing Edge Device verso il fogNode di Nebbiolo. Quest'ultimo è configurato come descritto nella Sezione 3.3.3, pertanto al suo interno è istanziata la macchina virtuale OTVM1 che ospita il broker RabbitMQ e un gateway DTN-to-MQTT. All'interno della OTVM1 è presente anche il software IBR-DTN che espone un *demone* in ascolto alla porta locale 4550. Il gateway DTN-to-MQTT ha una duplice funzionalità: effettua una registrazione con il demux token (Sezione 4.2.1) "*broker*" attraverso il demone, restando pertanto in ascolto delle notifiche relative ai bundle in arrivo per l'EID "`dtn://OTVM1/broker`"; si connette inoltre al broker RabbitMQ.

Anche all'interno del Sensing Edge Device troviamo il software IBR-DTN che va ad espanderne lo stack di rete. Qui troviamo inoltre un gateway MQTT-to-DTN in ascolto sulla porta locale 1883, corrispondente al protocollo MQTT. Infatti il gateway MQTT-to-DTN agisce da *broker locale* nei confronti del MQTT publisher, l'applicazione che emette i dati. Quando questa invia un messaggio "`PUBLISH`" il gateway lo riceve, inviando in risposta il messaggio "`PUBACK`" che ne conferma la ricezione, e ne incapsula *topic* e *payload* in un bundle diretto verso l'EID "`dtn://OTVM1/broker`". A questo punto, tramite le API esposte dal demone, inserisce il messaggio nella coda in uscita del Bundle Protocol Agent: il bundle è dunque

pronto per l'inoltro attraverso la DTN formata da tutti gli altri dispositivi mobili.

Quando il bundle raggiunge lo stack di rete del nodo Fog e lo risale fino a giungere al Bundle Layer, il demone invia una notifica di ricezione al gateway DTN-to-MQTT per mezzo del socket su cui quest'ultimo era in ascolto. Il gateway provvederà dunque a prelevare il bundle, estrarne topic e payload che costituivano il messaggio originale ed infine lo pubblicherà per mezzo di un messaggio "`PUBLISH`" nel broker del nodo Fog.

Questo processo introduce quindi un livello di astrazione della connettività nei confronti del publisher: questo continuerà a pubblicare i propri messaggi come se fosse continuamente connesso al broker del nodo Fog. Quest'architettura non comporta alcuna modifica né ai client né al message broker grazie alla coppia di gateway che fanno da interfaccia verso il protocollo MQTT. La responsabilità della consegna dunque è affidata in tutto e per tutto al Bundle Protocol Agent di IBR-DTN.

6.2.2 Comunicazione in downstream

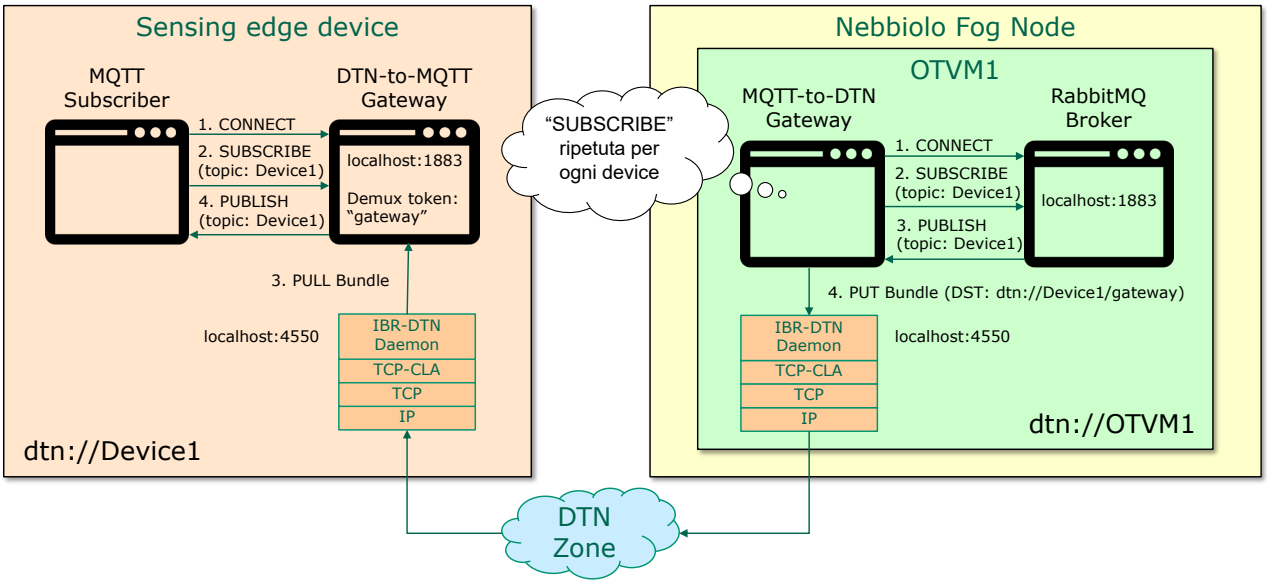


Figura 6.3. Architettura del sistema di comunicazione: Downstream

Nel caso della comunicazione in *downstream*, ovvero dal fogNode verso un Sensing Edge Device, vengono scambiati i ruoli dei due gateway e l'interazione si complica

leggermente. Come mostrato in Figura 6.3[1], il gateway MQTT-to-DTN del nodo Fog assume il ruolo di *subscriber* nei confronti del broker RabbitMQ: per ogni dispositivo in campo invia al broker un messaggio di "`SUBSCRIBE`", sottoscrivendosi al *topic* corrispondente all'EID che identifica il dispositivo nella DTN.

La lista contenente i dispositivi esistenti nel sistema deve essere tenuta e gestita all'interno del fogNode per essere disponibile sia al gateway MQTT-to-DTN che alle applicazioni che desiderano inviare messaggi ai dispositivi. Questa lista può essere gestita manualmente andando ad aggiungere un nuovo record per ogni nuovo dispositivo aggiunto al sistema. In alternativa, un'idea per rendere il sistema "*plug-and-play*" può essere quella di includere un semplice meccanismo di discovery all'interno della coppia di gateway: all'avvio il gateway DTN-to-MQTT, all'interno di un Sensing Edge Device, può inviare al fogNode un bundle con un messaggio speciale[2], contenente l'EID della macchina che lo ospita; alla ricezione del bundle il gateway MQTT-to-DTN all'interno del fogNode, se necessario, aggiorna la lista aggiungendo il dispositivo ed effettua la "`SUBSCRIBE`" per quell'EID presso il message broker.

Il gateway DTN-to-MQTT, presente all'interno del Sensing Edge Device, si pone come *broker locale* nei confronti del client MQTT che potrà sottoscriversi al topic corrispondente all'EID del nodo ospitante. Il gateway a sua volta effettua la registrazione con il demux token "*gateway*" attraverso il *demone* IBR-DTN in modo da poter essere notificato all'arrivo di bundle destinati al dispositivo.

A questo punto, quando un'applicazione sul nodo Fog effettua una "`PUBLISH`" presso il boker RabbitMQ al topic corrispondente ad un dei dispositivi, il gateway riceve il messaggio di "`PUBLISH`" inoltrato dal broker. Crea dunque un bundle diretto verso l'EID del nodo DTN corrispondente al topic estratto dal messaggio MQTT e lo inserisce nella coda in uscita del Bundle Protocol Agent attraverso le API esposte dal demone.

Come nel caso precedente, il bundle attraversa la DTN fino a giungere al Bundle Protocol Agent del dispositivo di destinazione. Qui il demone IBR-DTN notificherà il gateway DTN-to-MQTT della ricezione di un bundle, il quale andrà a estrarre dal bundle topic e payload del messaggio MQTT originale ed effettuerà a sua volta una "`PUBLISH`" verso l'applicazione subscriber.

Anche questo processo introduce un livello di astrazione della connettività, questa volta nei confronti del message broker presente nel nodo Fog: questo vedrà continuamente connessi dei "*dispositivi ombra*", che rappresentano in prima istanza i dispositivi in campo, e potrà recapitare loro i messaggi di "`PUBLISH`" che le applicazioni sul Fog vorranno inviare ad ognuno di essi indipendentemente dalla presenza della connessione.

[1] I messaggi di "ack" sono stati omessi per semplicità di rappresentazione

[2] ossia non contenente payload MQTT

6.3 Proof-of-Concept: Sistema di telemetria

L'architettura presentata nel caso della comunicazione in *upstream*, descritta nella Sezione 6.2.1, è stata utilizzata per realizzare il prototipo di un sistema di telemetria basato sulla piattaforma di Fog Computing di Nebbiolo Technologies.

Per realizzare i Sensing Edge Device è stata utilizzata una coppia di Raspberry Pi dotati di un dongle Wi-Fi. I Raspberry Pi così come il fogNode contengono al loro interno tutto il software necessario per la comunicazione attraverso la DTN. L'implementazione dei gateway MQTT-DTN così come la configurazione dettagliata di dispositivi e fogNode verrà discussa nel Capitolo 7.

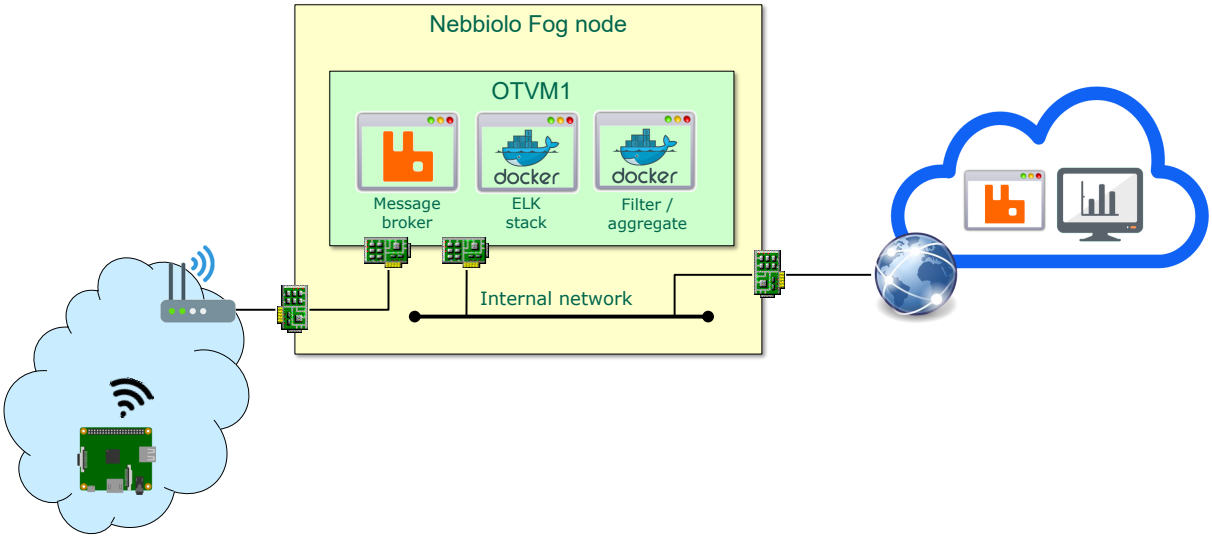


Figura 6.4. Overview del sistema di telemetria realizzato come Proof-Of-Concept dell'architettura di comunicazione proposta

In Figura 6.4 è illustrata l'architettura generale del Proof-Of-Concept realizzato. Ogni dispositivo è in grado di comunicare con il nodo Nebbiolo quando si trova nel range di copertura dell'Access Point ad esso collegato. Questo offre accesso alla rete interna della macchina virtuale OTVM1, dove è ospitato il message broker RabbitMQ insieme alle varie applicazioni che compongono il sistema. Ogni dispositivo emette ogni secondo dei dati relativi a temperatura, pressione ed umidità. Quando un dispositivo si connette al suddetto Access Point può pubblicare i propri dati, così come quelli trasportati per conto degli altri dispositivi che compongono la DTN, direttamente sul RabbitMQ del fogNode attraverso i meccanismi precedentemente discussi.

All'interno della OTVM1 troviamo due container Docker. Lo *stack ELK* permette di raccogliere, indicizzare e visualizzare graficamente i dati provenienti da ogni dispositivo, direttamente dall'interfaccia del *Nebbiolo System Management*. Il container denominato "*Filter/aggregate*" è una semplice applicazione Python che filtra le misure superiori ad un certa soglia e inserisce su una nuova coda i messaggi così filtrati.

La nuvola, invece, rappresenta la componente Cloud: si tratta di una macchina remota in cui è presente un'altra istanza di RabbitMQ e dello stack ELK. L'istanza di RabbitMQ all'interno del fogNode e quella remota sono sincronizzate sulla coda dei messaggi prodotti dall'applicazione filtro. Dal "Cloud", dunque, è possibile visualizzare i dati filtrati ed aggregati a granularità diversa rispetto quelli presenti sul fogNode.

Questo Proof-Of-Concept vuole essere soltanto un esempio di come sia possibile far interagire Fog e Cloud Computing. Il suo scopo principale è quello di fornire un'implementazione dell'architettura proposta in questo capitolo per veicolare messaggi provenienti da dispositivi mobili verso i nodi di Fog Computing in presenza di reti partizionate e non affidabili.

Capitolo 7

Implementazione del sistema di telemetria

In questo capitolo viene discussa l'implementazione del prototipo del sistema di telemetria presentato nel Capitolo 6. Il sistema fa uso del *fogNode* di Nebbiolo Technologies, di una coppia di Raspberry Pi utilizzati per realizzare i *Sensing Edge Device* e di una componente *Cloud*. Il capitolo si suddivide in sezioni che rappresentano i suddetti elementi, per ognuno dei quali si descrive sia la loro configurazione che i componenti implementati per la realizzazione delle rispettive funzionalità. Infine viene discusso il problema della sincronizzazione del tempo nei nodi che compongono la DTN. Tutti i sorgenti a cui si fa riferimento in questo capitolo sono pubblicati sui seguenti repository *GitHub*: MadLuigi/fog-over-dtn[1] e MadLuigi/rtl8188eu[2]. Le immagini Docker sviluppate per questo progetto sono raccolte nel seguente repository Docker Hub: luigimaio/nebbiolo.

7.1 Sensing Edge Device

Questa sezione illustra la configurazione dei Raspberry Pi utilizzati come Sensing Edge Device e viene descritta l'implementazione del *gateway MQTT-to-DTN*. Viene inoltre descritto come è stato realizzato il processo di connessione opportunistica tra due dispositivi che si incontrano e come i dispositivi si connettono al fogNode. I Raspberry Pi utilizzati sono due *Model B+ v1.2* con 512MB di RAM, mentre come sistema operativo stata utilizzata la distribuzione *Raspbian GNU/Linux 9.1 (stretch)* e *kernel 4.9.41*.

[1]Contiene i sorgenti dei componenti sviluppati per i Sensing Edge Device e il fogNode

[2]Contiene i sorgenti del driver del dongle Wi-Fi utilizzato nei due Raspberry Pi

7.1.1 Configurazione di rete

Ogni Raspberry Pi è equipaggiato con un dongle Wi-Fi *TP-Link TL-WN725N* con chipset Realtek *RTL8188EU*. Questo chipset garantisce il pieno supporto agli standard *IEEE 802.11b/g/n* con velocità fino a 150Mbps e agli standard di sicurezza *WEP*, *WPA-PSK/WPA2-PSK*. Inoltre, anche se non ufficialmente documentato, il dispositivo supporta anche la modalità *IBSS* o modalità Ad-Hoc.

I driver integrati nel kernel Linux per il dispositivo in questione non sono stabili e fanno parte del ramo di "*staging*[3]". Pertanto sono stati utilizzati dei driver derivati da un fork dei Realtek RTL8188EU (v4.3.0.8_13968.20150417) per Linux. Si tratta di driver piuttosto vecchi che sono stati modificati da un manteiner per essere compatibili con le recenti versioni di kernel (4.9+). Inoltre, nel fork realizzato per questo progetto, sono state apportate ulteriori modifiche di seguito elencate.

I driver Realtek supportano la modalità di funzionamento cosiddetta "*concurrent mode*", che permette di esporre al sistema operativo due interfacce di rete virtuali a partire da un solo dispositivo fisico e di utilizzarle contemporaneamente anche in modalità di funzionamento differenti. Per attivare questa modalità è stato necessario attivare il flag "`CONFIG_CONCURRENT_MODE`" decommentando la relativa direttiva "`#define`" nell'header file "`include/autoconf.h`".

La seconda modifica apportata è un workaround che fissa il BSSID che viene trasmesso dai beacon in modalità IBSS. Questa modifica si è resa necessaria poiché l'implementazione dell'algoritmo di *IBSS merge* (Sezione 5.3) dei suddetti driver ha diversi bug che causano il fenomeno del "*cell splitting*". Quando questo avviene i due dispositivi Wi-Fi non sono in grado di comunicare poiché, pur trasmettendo nello stesso canale e con lo stesso SSID, finché i BSSID non coincidono le reti Wi-Fi sono disgiunte.

Come detto, al sistema operativo sono esposte due interfacce wireless: "`wlan0`" e "`wlan1`". L'interfaccia "`wlan0`" è configurata in modalità Ad-Hoc per instaurare un collegamento con qualsiasi Sensing Edge Device nelle vicinanze. Su questa interfaccia è stato disabilitato il protocollo IPv4, lasciando invece abilitato il protocollo IPv6 che fa uso dell'indirizzo IPv6 Link-Local automaticamente generato; sarà quest'ultimo ad essere utilizzato per la comunicazione tra Sensing Edge Device. In questo modo tutti i peer sono paritetici poiché non avrebbe senso che uno in particolare scelga l'indirizzo di un altro.

L'interfaccia "`wlan1`", invece, è configurata in modalità *Managed* o *Station* e permette ad un dispositivo di connettersi all'Access Point del fogNode. In tal caso è il Nebbiolo stesso ad assegnare un indirizzo a tutti i dispositivi che vi si collegano per mezzo di un DHCP server, configurato come illustrato in Sezione 3.3.3.

[3]Si tratta di un albero di sorgenti dedicato a quei driver che si trovano ad uno stadio di sviluppo ancora preliminare o incompleto, e che non verrebbero normalmente resi disponibili per l'introduzione ufficiale nel kernel per il pubblico.

Le due interfacce di rete sono configurate per utilizzare il client `wpa_supplicant`, presente in versione v2.4 nella distribuzione utilizzata, in modo da stabilire la connessione alle rispettive reti wireless. Dalla versione v2.0 in poi, il `wpa_supplicant` supporta la modalità IBSS con chiavi WPA2-PSK; questo permette di realizzare reti Wi-Fi Ad-Hoc protette.

7.1.2 Configurazione del Bundle Protocol Agent IBR-DTN

Durante lo sviluppo del prototipo si è reso necessario contribuire alla risoluzione di un bug che non permetteva il corretto funzionamento del protocollo IPND quando i nodi DTN venivano configurati esclusivamente con un indirizzo IPv6; questo accade nel caso della connessione *device-to-device* attraverso l'interfaccia "`wlan0`" in cui è configurato soltanto un indirizzo IPv6 Link-Local auto-generato (Sezione 7.1.1).

Le modifiche apportate sono state riunite in una *pull request* accettata dall'autore di IBR-DTN, che l'ha inclusa nel proprio repository ufficiale. Pertanto l'installazione del software IBR-DTN è stata effettuata direttamente dai sorgenti, come descritto nella Sezione 4.5.1.

Per quel che concerne la configurazione del demone IBR-DTN vengono di seguito riportate le parti più rilevanti del file di configurazione utilizzato:

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

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

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

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

È stato specificato un percorso per il file di log ed è stato indicato di esporre le API del demone sull'interfaccia di *loopback* (alla porta di default 4550, in quanto non diversamente specificato). Si noti che, non essendo stato esplicitamente configurato, l'EID del nodo corrisponderà all'*hostname* del Raspberry Pi sul quale sta eseguendo l'istanza del software IBR-DTN.

```
#####################################
# storage configuration #
#####################################

#
# define a folder for temporary storage of bundles
# if this is not defined bundles will processed in memory
#
blob_path = /tmp

#
# define a folder for persistent storage of bundles
# if this is not defined bundles will stored in memory only
#
storage_path = /var/spool/ibrdtn/bundles

#
# 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 = sqlite
```

È stato specificato sia il percorso per il salvataggio temporaneo dei bundle, durante il loro processamento, che il percorso per il loro salvataggio persistente. Inoltre, avendo compilato il software IBR-DTN con il modulo *SQLite*, questo è stato selezionato come modalità di memorizzazione dei bundle.

```
#####################################
# convergence layer configuration #
#####################################

#
# discovery over UDP/IP
#
# You can specify an multicast address to listen to for discovery
# announcements.
# If no address is specified the multicast equivalent of broadcast is
# used.
#
discovery_address = ff02::142 224.0.0.142

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

```
# seconds.
discovery_interval = 1

#
# 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 = wlan0   # listen on interface wlan0
net_lan0_port = 4556    # with port 4556 (default)

#
# configuration for a convergence layer named lan1
#
net_lan1_type = tcp     # we want to use TCP as protocol
net_lan1_interface = wlan1   # listen on interface wlan1
net_lan1_port = 4556    # with port 4556 (default)
```

Sono stati configurati gli indirizzi multicast IPv4 ed IPv6 sui quali porsi in ascolto per ricevere i beacon IPND che annunciano la presenza di nuovi nodi DTN. L'intervallo temporale tra l'emissione di un beacon ed il successivo è stato impostato ad 1 secondo. Inoltre sono stati configurati due Convergence Layer Adapter per TCP, "`lan0`" e "`lan1`", rispettivamente sulle interfacce "`wlan0`" e "`wlan1`".

```
####################################
# routing configuration #
####################################

#
# routing strategy
#
# values: default  epidemic  flooding  prophet  none
#
routing = epidemic

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

#
```

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

È stato configurato l'algoritmo di routing epidemico, descritto in Sezione 4.3.2, ed è stata attivata la possibilità di inoltrare bundle attraverso i nodi incontrati, così da permettere il data-muling su cui si basa l'architettura proposta. Inoltre è stata indicata la preferenza di inoltrare i bundle direttamente alla destinazione se questa si trova tra i nodi correntemente connessi, evitando in questo caso ideale l'inoltro in flooding.

7.1.3 Connessione opportunistica tra dispositivi

Come visto in Sezione 7.1.1, ciascuno dei Sensing Edge Device ha un'interfaccia wireless configurata in modalità Ad-Hoc. Secondo il meccanismo descritto in Sezione 5.3, quando questi entrano nel range di copertura wireless l'uno dell'altro avviene l'*IBSS merge* e da quel momento i due dispositivi fanno parte della stessa rete wireless.

A questo punto entra in gioco il protocollo IPND: i dispositivi emettono ogni secondo dei beacon in multicast per annunciare la loro presenza. Viene emesso un beacon IPND per ogni Convergence Layer Adapter configurato, quindi nel caso dei Sensing Edge Device ne vengono emessi due: uno per l'interfaccia "`wlan0`" ed uno per l'interfaccia "`wlan1`" (Sezione 7.1.2).

Quindi, successivamente alla connessione a livello fisico i due dispositivi saranno in grado di rilevare, con un ritardo di circa 1 secondo, il rispettivo vicino. Una volta effettuato il binding IP-EID attraverso lo scambio di beacon IPND, i due vicini possono instaurare una connessione TCP e scambiarsi dei bundle attraverso il Convergence Layer Adapter.

Tutto questo processo è automatico e totalmente "*stateless*", infatti nessun indirizzo IP deve essere configurato per i dispositivi connessi alla rete Ad-Hoc: il protocollo IPND rileva l'indirizzo IPv6 Link-Local di ciascun vicino connesso a livello fisico.

Quando i nodi escono reciprocamente dal range di copertura wireless del vicino, continueranno per un pò di tempo a non notare la disconnessione. Tipicamente un nodo rileva la disconnessione di un vicino a seguito del fallimento dell'eventuale canale TCP instaurato.

Un discorso analogo può essere fatto per la connessione tra Sensing Edge Device e fogNode. Quando un dispositivo entra nel range di copertura wireless dell'Access Point connesso al fogNode, si connette ad esso in modo automatico sfruttando il client `wpa_supplicant`. L'unica differenza sta nel fatto che il fogNode assegnerà, tramite DHCP, l'indirizzo al dispositivo. Da quel momento si ripete il processo di scambio di beacon IPND, già illustrato nel caso precedente, e il dispositivo potrà inviare al fogNode i bundle contenenti le misure.

Allo scopo di restituire un feedback visivo dell'avvenuta connessione a livello di bundle layer tra due dispositivi, è stato creato un semplice circuito con due LED colorati su una *breadboard*. Il circuito è stato collegato ad un Raspberry Pi per mezzo dei suoi pin GPIO ed uno script Python è stato realizzato per pilotare i LED ogni qual volta cambia lo stato di connessione ad un vicino.

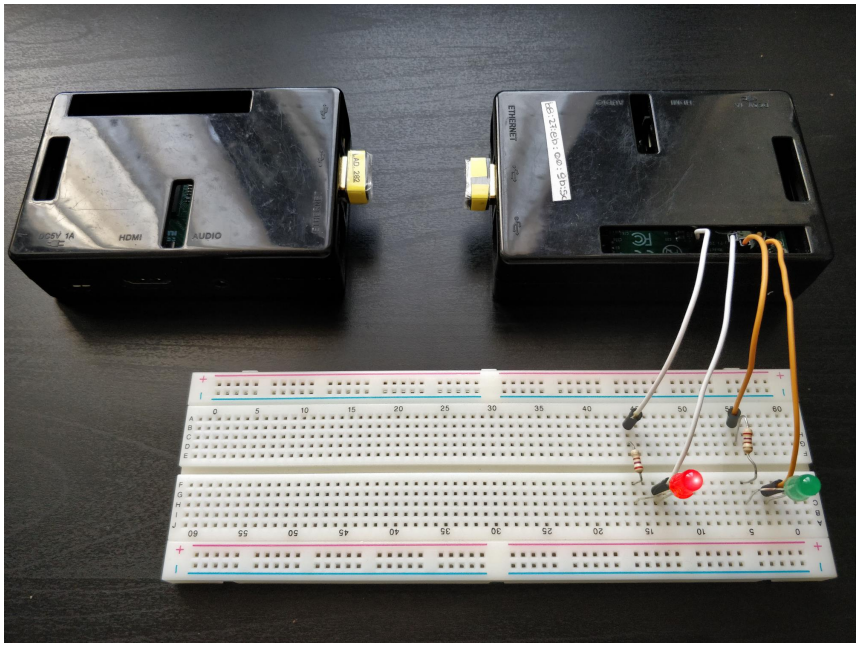


Figura 7.1. Prototipi di Sensing Edge Device: i due LED restituiscono un feedback visivo sullo stato di connessione ad un dispositivo vicino o al fogNode

In Figura 7.1 sono mostrati i due Sensing Edge Device, uno dei quali è collegato al circuito con i LED. Come si può vedere in foto, ci sono due LED: l'accensione di quello "rosso" sta ad indicare che il Sensing Edge Device è attualmente "vicino" ad un altro suo peer; l'accensione di quello "verde", invece, sta ad indicare che il dispositivo si trova nel range di copertura del fogNode e che quindi può scambiare bundle con quest'ultimo.

Di seguito vengono riportate le parti significative dello script Python che permette di pilotare il circuito a LED:

```
6  # Address and port of the DTN daemon
7  DAEMON_ADDRESS = 'localhost'
8  DAEMON_PORT = 4550
10
11 def exit_handler(signal, frame):
12     # Turn off LEDs before exit
```

```
18
19 # Ctrl-C Signal Handler
20 signal.signal(signal.SIGINT, exit_handler)
27
28 # Create the socket to communicate with the DTN daemon
29 d = socket.socket()
30 # Connect to the DTN daemon
31 d.connect((DAEMON_ADDRESS, DAEMON_PORT))
32 # Get a file object associated with the daemon's socket
33 fd = d.makefile()
40
41 nebbioloLEDOn = False
42 anyDeviceLEDOn = False
43
44 while True:
45     nebbioloDiscovered = False
46     anyDeviceDiscovered = False
47     neighboors = []
48
49     d.send("neighbor list\n")
50     fd.readline()
51     while True:
52         res = fd.readline()
53         if res == "\n":
54             # List end
55             break
56         neighboors.append(res)
57
58     for neighbor in neighboors:
59         if neighbor.startswith("dtn://otvm1"):
60             nebbioloDiscovered = True
61         else:
62             anyDeviceDiscovered = True
63
64     if nebbioloDiscovered and not nebbioloLEDOn:
65         print "FogNode Connected - Green LED On\n"
66         nebbioloLEDOn = True
67         GPIO.output(17, GPIO.HIGH)
68     elif not nebbioloDiscovered and nebbioloLEDOn:
69         print "FogNode Disconnected - Green LED Off\n"
70         nebbioloLEDOn = False
```

```
71          GPIO.output(17, GPIO.LOW)
72
73      if anyDeviceDiscovered and not anyDeviceLEDOn:
74          print "Peer device(s) connected - Red LED On\n"
75          anyDeviceLEDOn = True
76          GPIO.output(22, GPIO.HIGH)
77      elif not anyDeviceDiscovered and anyDeviceLEDOn:
78          print "No Peer device(s) connected - Red LED
               Off\n"
79          anyDeviceLEDOn = False
80          GPIO.output(22, GPIO.LOW)
```

Alle righe 29-31 viene creato un socket tramite il quale lo script Python può connettersi alle API esposte dal demone IBR-DTN alla porta 4550.

Alla riga 44 inizia il *polling* al demone per stabilire se un nuovo dispositivo o il fogNode sono stati rilevati nelle vicinanze. Nel caso in cui all'iterazione precedente nessun dispositivo fosse nelle vicinanze e il rispettivo LED rosso fosse spento, questo viene accesso se all'iterazione corrente viene rilevato un nuovo vicino. Lo stesso vale per il fogNode: se all'iterazione precedente non fosse tra la lista di vicini mentre all'iterazione corrente si, il LED verde viene accesso.

Al contrario se all'iterazione precedente il fogNode o un peer fossero connessi al Sensing Edge Device e questi non vengono più rilevati all'iterazione corrente, allora il rispettivo LED viene spento. L'accensione e lo spegnimento dei LED viene effettuato attraverso la libreria "`RPi.GPIO`".

Alla riga 20 lo script registra anche un *handler* del segnale "`SIGINT`", che se catturato chiama la callback "`exit_handler`" per spegnere i LED eventualmente accesi prima di terminare l'esecuzione dello script.

7.1.4 MQTT Publisher

Il publisher MQTT è un'applicazione scritta in Python che sfrutta la libreria *Eclipse Paho MQTT*. Questa libreria permette di scrivere un client MQTT in Python che rispetti le specifiche MQTT v3.1 e v3.1.1. È possibile installare la libreria da *PyPi*, il popolare repository di software di terze parti per Python, per mezzo dello strumento "`pip`" eseguendo su una shell il comando "`pip install paho-mqtt`".

Di seguito vengono riportate le parti più significative del MQTT Publisher:

```
1  import paho.mqtt.client
7
8  sleepTime = 1
9  # MQTT details
10 mqttDeviceId = socket.gethostname()
```

```
11 mqttBrokerHost = "localhost"
12 mqttBrokerPort = 1883
15 mqttTelemetryTopic = "Tierra.SensorsData"
40 # Set MQTT client
41 mqttClient = paho.mqtt.client.Client()
48 # Connect to MQTT broker
49 mqttClient.connect(mqttBrokerHost, mqttBrokerPort, 60)
50 mqttClient.loop_start()
51
52 # Collect telemetry information and publish to MQTT
      broker in JSON format
53 while True:
54     telemetryData = {}
55     telemetryData["DeviceId"] = mqttDeviceId
56     telemetryData["Timestamp"] =
          datetime.datetime.utcnow().strftime('%Y-%m-%d
          %H:%M:%S.%f')[:-3]
57     telemetryData["Temperature"] =
          str(round(random.uniform(20, 30), 2))
58     telemetryData["Humidity"] =
          str(round(random.uniform(0, 100), 2))
59     telemetryData["Pressure"] =
          str(round(random.uniform(900, 1100), 2))
60     telemetryDataJson = json.dumps(telemetryData)
61     mqttClient.publish(mqttTelemetryTopic,
          telemetryDataJson, 0)
62     time.sleep(sleepTime)
63
64 mqttClient.loop_stop()
65 mqttClient.disconnect()
```

Come si può vedere dal listato, il client si connette alla porta locale 1883: qui ci sarà in ascolto il *gateway MQTT-DTN*, che verrà descritto in Sezione 7.1.5. Questo client MQTT è dunque un esempio di come l'architettura proposta sia trasparente nei confronti delle applicazioni esistenti; il publisher percepirà il gateway come un broker locale a cui connettersi.

Alla riga 53 inizia un loop infinito che, ad intervalli di 1 secondo (si veda la variabile "`sleepTime`"), genera una serie di misure casuali (temperatura, umidità, pressione) e le mette all'interno di un JSON che costituisce il payload del messaggio MQTT da pubblicare al topic "`Tierra.SensorsData`". Nel JSON sono inoltre presenti un campo "`DeviceId`", che identifica il dispositivo che ha generato le misure tramite il suo *hostname*, e un campo "`Timestamp`", che tiene traccia dell'istante in

cui le misure sono state generate.

7.1.5 Gateway MQTT-to-DTN

Il gateway MQTT-to-DTN è un'applicazione scritta in Python che ha una duplice funzionalità: in primo luogo resta in ascolto dei messaggi MQTT provenienti dal publisher e li processa per restituire l'opportuna risposta (es. "`PUBACK`" o "`PINGRESP`") e per estrarre topic e payload dai messaggi di "`PUBLISH`"; in secondo luogo interagisce con le API esposte dal demone IBR-DTN per creare i bundle, a partire da topic e payload estratti dai messaggi MQTT, e per inserirli nella coda di uscita del Bundle Protocol Agent.

Si riportano di seguito le parti più significative di codice:

```
6   # Address and port to listen to MQTT messages
7   SERVER_ADDRESS = 'localhost'
8   SERVER_PORT = 1883
9   # Address and port of the DTN daemon
10  DAEMON_ADDRESS = 'localhost'
11  DAEMON_PORT = 4550
16  DESTINATION_EID = 'dtn://otvm1/broker'
18
19  def decode_remaining_length(length_bytes):
20      """
21      Decode remaining message length according to MQTT
            specifications
28      """
47
48  def parse_message(msg):
49      """
50      Parse a MQTT message received from a client and
            possibly reply with the appropriate response
60      """
130
131 def send_bundle(topic, message, qos=0, mid=0):
132     """
133     Create a bundle from a MQTT message and send it
            through IBR-DTN deamon
140     """
177
178 # Create the socket to communicate with the DTN daemon
179 d = socket.socket()
180 # Connect to the DTN daemon
```

```
181 d.connect((DAEMON_ADDRESS, DAEMON_PORT))
182 # Get a file object associated with the daemon's socket
183 fd = d.makefile()
200 # Create the socket to listen to MQTT messages coming
       from client
201 s = socket.socket()
205 s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
206 # Bind to the desired address(es) and port
207 s.bind((SERVER_ADDRESS, SERVER_PORT))
211 s.listen(5)
212
213 print("Listening on address %s. Kill server with Ctrl-C"
       % str((SERVER_ADDRESS, SERVER_PORT)))
214
218 while True:
219     c, addr = s.accept()
220     print("\nConnection received from %s" % str(addr))
221
222     unprocessed_buffer = ''
223     current_raw_message = ''
224     first_message = True
225     new_message = True
226     while True:
227
228         try:
229             data = c.recv(1024)
230         except socket.error as error:
231             # In Windows WSAECONNRESET: 10054
232             if error.errno == errno.ECONNRESET or
                   error.errno == 10054:
233                 print("Connection reset by peer.
                       Resetting")
234                 break
235
236         if not data:
237             print("Client closed connection. Resetting")
238             break
239
240         res = parse_message(data)
241         first_message = False
242         if res == -2:
```

```
243             print("The first message received must be a
                    CONNECT message. Closing connection and
                    resetting")
244             break
245         elif res == -1:
246             print("Wrong message format. Closing
                    connection and resetting")
247             break
248         elif res == 0:
249             new_message = False
250             continue
251
252         new_message = True
253         while len(unprocessed_buffer) > 0:
254             res = parse_message(unprocessed_buffer)
255             if res == -1:
256                 print("Wrong message format. Closing
                        connection and resetting")
257                 break
258             elif res == 0:
259                 new_message = False
260                 break
261             new_message = True
262
263         if res == -1:
264             break
265
266     c.close()
267
268 d.close()
```

Alla riga 19 è dichiarata la funzione "`decode_remaining_length`" che serve a decodificare il campo "*Remaining Length*" di un messaggio MQTT secondo le specifiche del protocollo [33]. Questa funzione restituisce una tupla contenente in prima posizione la lunghezza rimanente del messaggio MQTT decodificata ed in seconda posizione il numero di byte occupati dalla codifica del suo valore (fino ad un massimo di 4).

La funzione "`parse_message`", dichiarata alla riga 48, serve ad effettuare la *parsificazione* di un messaggio MQTT. In output fornisce un intero che può assumere i seguenti valori:

-2. Se il primo messaggio ricevuto da un client MQTT è diverso da quello di

"`CONNECT`"; un flag discrimina se si tratta del primo messaggio da parte di un client.

-1. Se il messaggio ricevuto è in un formato sconosciuto ed è occorso un errore durante la parsificazione.

0. Se il messaggio MQTT è incompleto e occorre leggere altri byte dal socket connesso al client.

1. Se la parsificazione è stata completata con successo.

Questa funzione si occupa anche di rispondere al client MQTT con il messaggio appropriato (es. "`CONNACK`", "`PUBACK`", "`PINGRESP`"...).

La funzione "`send_bundle`" (riga 131) si occupa invece di creare un bundle a partire da topic, payload ed altri parametri estratti dal messaggio MQTT originario. Questa funzione si occupa inoltre di inserire il bundle creato nella coda di uscita del Bundle Protocol Agent tramite le API esposte dal demone. Una volta effettuata questa operazione sarà cura del software IBR-DTN di occuparsi della conservazione e dell'inoltro del bundle.

Alle righe 179-181 viene creato e connesso il socket che interagisce con il demone IBR-DTN, mentre alle righe 201-211 viene creato e messo in ascolto alla porta 1883 il socket a cui si connetteranno i client MQTT. Il gateway infatti agirà da broker locale nei confronti dell'applicazione publisher.

Alla riga 218 inizia il *loop* per servire i client MQTT: si tratta di un'implementazione semplice che può gestire un client alla volta, perché di fatto all'interno del Sensing Edge Device ci sarà sempre e solo un'applicazione publisher. Nel caso di disconnessione da parte del client o di errori, il gateway può servire una nuova connessione.

Il ciclo interno (riga 226) legge 1kB di dati alla volta dal socket connesso al client MQTT e ne effettua la parsificazione per mezzo dell'apposita funzione. Alla riga 253 del listato c'è un'ulteriore ciclo interno che, nel caso in cui i dati parsificati non costituiscano un messaggio MQTT completo, innesca un'altra lettura dal socket.

7.2 Nebbiolo fogNode

Questa sezione illustra la configurazione degli elementi che sono in esecuzione sul fogNode, fatta eccezione per le macchine virtuali e la configurazione di rete per cui si fa riferimento a quanto scritto nella Sezione 3.3.3. Viene inoltre descritta l'implementazione del *gateway DTN-to-MQTT* e dell'applicazione *Data Filter*.

7.2.1 Configurazione RabbitMQ

Il message broker RabbitMQ è stato istanziato come container Docker all'interno della OTVM1, tramite l'interfaccia del *Nebbiolo System Management*. L'immagine utilizzata è quella disponibile sul repository ufficiale di *Docker Hub*, in versione "*3-management*". Questa versione permette di istanziare RabbitMQ con il plug-in di management già abilitato.

Poiché i Sensing Edge Device pubblicano i messaggi contenenti le misure tramite il protocollo MQTT, è necessario abilitare anche il relativo plug-in. È possibile avviare una shell all'interno del container in esecuzione di RabbitMQ tramite il comando "`docker exec -it RabbitMQ /bin/sh`" e da questa eseguire l'abilitazione del plug-in digitando "`rabbitmq-plugins enable rabbitmq_mqtt`".

In alternativa è stato preparato un *Dockerfile*[4] che estende l'immagine ufficiale di RabbitMQ andando ad eseguire il comando per abilitare il plug-in MQTT durante il deployment del container.

Quando un messaggio viene pubblicato su RabbitMQ tramite il protocollo MQTT, internamente viene utilizzato un exchange di tipo "*topic*" (di default l'exchange "`amq.topic`") e per recapitare il messaggio alla coda opportuna viene utilizzato il *topic MQTT* come *routing key*.

Nel fogNode sono state configurate due code denominate "`Tierra.Telemetry`" e "`Tierra.Telemetry.DataToFilter`". Entrambe le code sono configurate per ricevere messaggi dall'exchange `amq.topic` con la stessa *binding key*: "`Tierra.SensorsData`". Quando i dispositivi pubblicano sul fogNode delle misure tramite il protocollo MQTT al topic "`Tierra.SensorsData`", i messaggi confluiscono in entrambe le code.

La necessità di avere i messaggi duplicati su due code è dovuta al fatto che sul fogNode sono presenti due applicazioni che fungono da consumatori delle misure provenienti dai dispositivi: lo stack ELK e l'applicazione "Data Filter". In questo modo si evita che le due applicazioni si sottraggano messaggi a vicenda.

Per sincronizzare l'istanza di RabbitMQ presente sul fogNode e quella presente sul "Cloud" (una macchina remota) è stata utilizzata una *shovel*. Si tratta di una particolare funzionalità di RabbitMQ che permette di muovere i messaggi da una coda o da un exchange di un broker sorgente a una coda o ad un exchange di un broker destinazione, che può essere eseguito sullo stesso nodo o su un nodo remoto, in modo affidabile ed automatico.

La shovel configurata ha come sorgente tutti i messaggi pubblicati sull'exchange "`amq.topic`" con una routing key "`Tierra.FilteredData.#`" e come destinazione l'exchange "`amq.topic`" del broker remoto. Si noti che la routing key sorgente ha una wildcard, per cui tutti i messaggi che hanno una routing key che inizi per "`Tierra.FilteredData`" verranno presi in considerazione.

[4]La relativa immagine nel repository Docker Hub è `luigimaio/nebbiolo:rabbitmq-mqtt`

L'applicazione "*Data Filter*", come verrà descritto nella sezione 7.2.5, pubblica le misure filtrate nell'exchange "`amq.topic`" con tre differenti routing key, tutte con il prefisso "`Tierra.FilteredData`", a seconda che si tratti di misure di temperatura, umidità o pressione. Nel broker presente all'interno del fogNode non esiste nessuna coda con un binding per quelle routing key; quei messaggi sono "trasportati" sul broker remoto attraverso la shovel.

7.2.2 Configurazione dello stack ELK

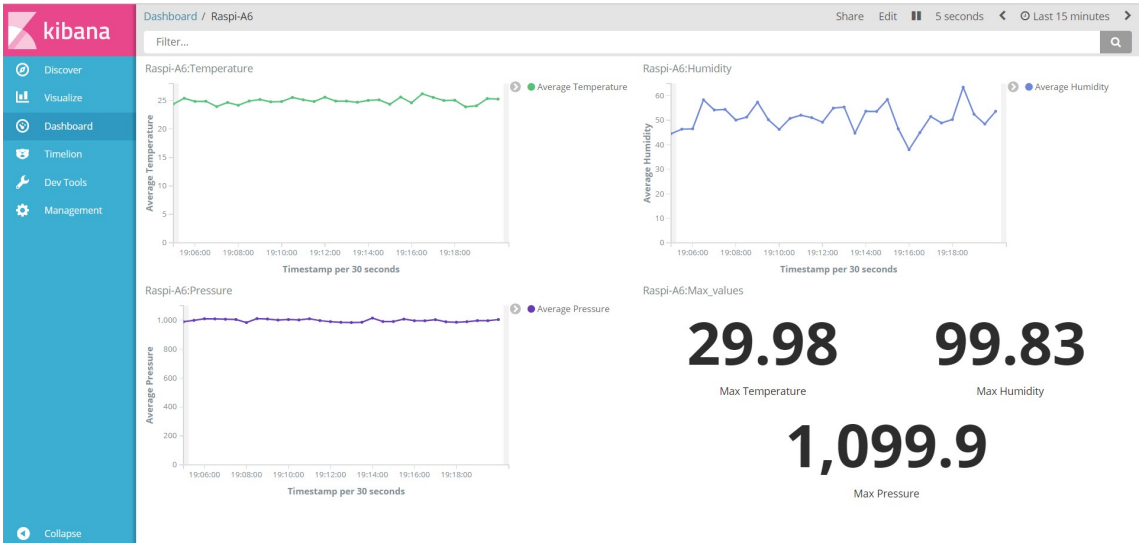


Figura 7.2. Screenshot dell'interfaccia di Kibana in esecuzione sul fogNode

Per raccogliere, indicizzare e visualizzare le misure contenute nei messaggi pubblicati dai Sensing Edge Device è stato utilizzato lo stack composto da Elasticsearch, Logstash e Kibana. Si tratta di una versione containerizzata dell'intero stack, in modo da poter essere istanziato come container Docker all'interno del fogNode.

Logstash consuma i dati provenienti dalla coda "`Tierra.Telemetry`" contenente i messaggi provenienti dai dispositivi; Elasticsearch li indicizza e tramite l'interfaccia web di Kibana possono essere prodotti diversi grafici visualizzabili collegandosi direttamente al fogNode.

L'immagine utilizzata necessita di una configurazione che descriva la sorgente dei dati per Logstash e il loro formato (i.e. JSON). È stata dunque prodotta un'immagine derivata[5] contenente uno script di avvio che permette la configurazione del

[5]L'immagine presente nel repository Docker Hub è `luigimaio/nebbiolo:elk-rmq-fog`

container per mezzo delle variabili d'ambiente. L'interfaccia del fogSM permette di specificare le variabili d'ambiente da passare al container in fase di deployment.

Elasticsearch memorizza gli indici su dei file mappati in memoria utilizzando la system call `mmap()`. Il limite di default del parametro del kernel "`vm.max_map_count`", che controlla il numero di aree che un processo può mappare in memoria, nella maggior parte dei sistemi Linux è troppo basso per l'esecuzione di Elasticsearch. Pertanto prima di istanziare il container dal fogSM è stato necessario aumentare questo limite eseguendo il seguente comando all'interno di una root shell della OTVM1: "`sysctl -w vm.max_map_count=262144`".

Per indicizzare i dati viene utilizzato il campo "`Timestamp`" del corpo di ogni messaggio. Viene quindi creata una serie temporale delle misure in base al momento di generazione del messaggio. Elasticsearch aggiunge automaticamente il meta-campo "`@timestamp`" corrispondente al momento in cui i dati vengono ingeriti. Questo campo non può essere utilizzato per indicizzare le misure in quanto, in ottica di un inoltro ritardato a causa della connettività intermittente, il suo valore non rappresenterebbe un riferimento temporale valido.

Tramite l'interfaccia di Kibana sono state create diverse visualizzazioni che riportano l'andamento temporale delle misure raccolte per ogni singolo dispositivo. Tutte le visualizzazioni sono state inserite in una dashboard, che è stata esportata ed inclusa nel progetto per essere facilmente importata in una nuova istanza dello stack. In Figura 7.2 viene mostrato l'andamento delle misure raccolte per uno dei dispositivi.

7.2.3 Configurazione del Bundle Protocol Agent IBR-DTN

Il software IBR-DTN è istanziato come container Docker all'interno del nodo Nebbiolo. A tale scopo è stato creato un apposito *Dockerfile*[6] che automatizza il processo di installazione, descritto in Sezione 4.5.1, durante il deployment del container Docker.

Questo container è l'unico che non è stato possibile istanziare tramite Nebbiolo System Management. Il suo deployment è stato effettuato manualmente con il comando "`docker run`" poiché si è reso necessario associare il container alla rete "host" piuttosto che a quella "bridge", che viene associata di default. Nella release attuale del fogSM, infatti, non è prevista la possibilità di istanziare un container su una rete diversa da quella "bridge".

Questa necessità è scaturita dal fatto che i container Docker non supportano il traffico multicast proveniente dall'esterno [34], rendendo quindi inefficace il protocollo IPND per il discovery dei vicini all'interno della DTN. Associando il container di IBR-DTN alla rete "host", questo utilizza direttamente lo stack di rete della OTVM1 alla quale si connettono i dispositivi.

[6] La relativa immagine nel repository Docker Hub è `luigimaio/nebbiolo:ibr-dtn`

Per quanto riguarda la configurazione del demone, questa è sostanzialmente identica a quella descritta per i Sensing Edge Device in Sezione 7.1.2. Anche in questo caso, non essendo stato specificato diversamente, l'EID corrisponderà all'hostname della macchina su cui è in esecuzione. Avendo istanziato il container utilizzando la rete "host", l'EID corrisponderà all'hostname della OTVM1 ovvero "`dtn://otvm1`".

L'unica differenza con la configurazione dei Sensing Edge Device è l'aver specificato un solo Convergence Layer Adapter per TCP sull'interfaccia "`eth0`". Si ricorda che quest'interfaccia della OTVM1, come descritto in sezione 3.3.3, ha corrispondenza con la porta fisica "S0" alla quale è collegato l'Access Point del fogNode.

7.2.4 Gateway DTN-to-MQTT

Il *gateway DTN-to-MQTT*, scritto in Python, riceve ed elabora i bundle diretti al fogNode. È stata realizzata un'immagine Docker[7] anche per quest'applicazione, così da poter sfruttare la possibilità di deployment offerta dal fogSM.

L'immagine realizzata sfrutta uno script di avvio che permette di specificare alcuni parametri come variabili d'ambiente (i.e. *indirizzo* e *porta* dei container contenenti il demone IBR-DTN e RabbitMQ, *demux_token* del gateway etc.). Questo evita di dover ricostruire l'immagine Docker ogni qual volta cambi la configurazione sul fogNode; sarà sufficiente rieseguire il deploy del container.

Di seguito vengono riportate le parti significative dell'implementazione del gateway DTN-to-MQTT:

```
23 def daemon_reader_thread(cv):
24     """
25     The reader thread fetches notifications of incoming
          bundles and responses to requests through the
          daemon's socket
29     """
30     global response
31     global response_is_ready
32
33     while True:
34         remaining_lines = 0
35         res = fd.readline().rstrip()
36         if res.startswith("602 NOTIFY BUNDLE"):
37             # Put the notification in a queue. The main
                thread will be responsible to process
                notifications.
```

[7] L'immagine presente nel repository Docker Hub è `luigimaio/nebbiolo:gateway-dtn-mqtt`

```
38              notifications.put(res)
39              # No further processing is needed.
40              continue
41          elif res.startswith("200 BUNDLE LOADED"):
42              remaining_lines = 0
43          elif ...
47
48          with cv:
49              response = []
50              for i in range(0, remaining_lines):
51                  response.append(fd.readline().rstrip())
52              response.insert(0, res)
53
54              if res.startswith("200 PAYLOAD GET"):
55                  ...
69
70              # Set the ready flag and wake up the main
                  thread which is waiting for the response
                  to a request
71              response_is_ready = True
72              cv.notify()
73
75  def wait_for_response(cv):
76      """
77      Synchronized read of the response to a request made
          to the DTN daemon
84      """
85      global response_is_ready
86
87      with cv:
88          while not response_is_ready:
89              condition.wait()
90          response_is_ready = False
93
94          ret = response[:]
95
96      return ret
120
121 # Set MQTT client
122 mqttClient = paho.mqtt.client.Client()
129 # Connect to MQTT broker
```

```
130 mqttClient.connect(mqttBrokerHost, mqttBrokerPort, 60)
131 # Start the network loop to process the communication
       with the broker in a separated thread
132 mqttClient.loop_start()
133
134 # Create the socket to communicate with the DTN daemon
135 d = socket.socket()
136 # Connect to the DTN daemon
137 d.connect((DAEMON_ADDRESS, DAEMON_PORT))
138 # Get a file object associated with the daemon's socket
139 fd = d.makefile()
146 # Set endpoint identifier
147 d.send("set endpoint %s\n" % SOURCE_DEMUX_TOKEN)
148
149 # Create a worker thread that reads from daemon's socket
       for responses to requests or notifications of
       incoming bundles.
155 response = []
156 response_is_ready = False
157 condition = threading.Condition()
158 notifications = Queue.Queue()
159 reader = threading.Thread(name='daemon_reader',
       target=daemon_reader_thread, args=(condition,))
160 reader.start()
161
162 # Main thread loop:
165 while True:
166     notification = notifications.get()
167     query_string = notification.split(' ', 3)[3]
168     d.send("bundle load %s\n" % query_string)
169     wait_for_response(condition)
170
171     d.send("payload get\n")
172     res = wait_for_response(condition)
173     mqttData = base64.b64decode(res[4]).split('\n')
174
175     mqttTopic = mqttData[0]
176     mqttDataJson = mqttData[1]
177     mqttClient.publish(mqttTopic, mqttDataJson, 1)
178
179     d.send("bundle free\n")
```

```
180     wait_for_response(condition)
181
182     notifications.task_done()
183
184 mqttClient.loop_stop()
185 mqttClient.disconnect()
```

L'applicazione si connette tramite socket al demone IBR-DTN (riga 137) ed effettua una registrazione per l'EID "`dtn://otvm1/broker`"[8] (riga 147). Il demone notificherà l'applicazione dell'avvenuta ricezione di ogni nuovo bundle.

Le API esposte dal demone IBR-DTN si basano su un protocollo testuale: ogni richiesta – comando – è una linea di testo; le risposte del demone possono spaziare su una o più linee. Poiché le notifiche di ricezione di un bundle da parte del demone sono asincrone, queste potrebbero arrivare all'applicazione anche durante l'attesa di una risposta ad un comando. Ciò ha reso necessario un meccanismo di sincronizzazione per deinterlacciare le notifiche dalle risposte del demone.

Per fare ciò alle righe 159-160 viene istanziato un *worker thread* che si occupa di leggere, una riga alla volta, ciò che arriva sul socket connesso al demone. L'implementazione del worker thread è all'interno della funzione "`daemon_reader_thread`" alla riga 23. Se la linea letta è una notifica di ricezione di un bundle, allora questa viene inserita nella coda "`notifications`". Si tratta di un'istanza della classe `Queue.Queue` di Python, che implementa una coda sincronizzata. Se la linea è una notifica non viene effettuato nessun ulteriore processamento e il thread passa a leggere la linea successiva dal socket. Al contrario se si tratta di una risposta ad un comando, viene determinato il numero di linee che compongono la risposta e si procede alla lettura. Questa lettura è effettuata all'interno del blocco "`with`" (riga 48) che sincronizza il "*main loop*" dell'applicazione per mezzo di una *condition variable*.

Come si può vedere a partire dalla riga 165 del listato, il loop principale dell'applicazione si occupa di smaltire le notifiche di ricezione provenienti dal worker thread. Estrae dalla coda una notifica alla volta, quando queste sono disponibili, e tramite le API esposte dal demone recupera il bundle relativo alla notifica per ricostruire il messaggio MQTT originario. Dopo ogni comando, il thread principale si mette in attesa della risposta del demone per mezzo del metodo sincronizzato "`wait_for_response`". Questo metodo, implementato alla riga 75, effettua una "`wait()`" sulla *condition variable* e blocca il main thread fino all'arrivo di una "`notify()`" da parte del worker thread. Quando il main loop ha recuperato il bundle e ricostruito il messaggio MQTT, lo pubblica sul message broker RabbitMQ del

[8] "broker" è il *demux_token* di default, ma è un parametro personalizzabile in fase di deployment del container Docker

fogNode sfruttando la libreria Eclipse Paho. A questo punto il main loop ricominicia elaborando la notifica successiva.

7.2.5 Data Filter

Questa semplice applicazione, scritta in Python, prende in input dal message broker del fogNode le misure provenienti dai dispositivi e si occupa di filtrare quelle superiori ad una certa soglia. Le soglie, così come i parametri di connessione al broker RabbitMQ, sono configurabili per mezzo di variabili d'ambiente da specificare al momento del deployment del container Docker[9] che contiene l'applicazione.

Di seguito vengono riportate le parti significative dell'implementazione del Data Filter:

```
2  import pika
4
8  rabbitmq_queue = 'RABBITMQ_QUEUE'
9  rabbitmq_topic = 'RABBITMQ_TOPIC'
13 queue_to_filter = rabbitmq_queue + '.DataToFilter'
14
16 connection =
      pika.BlockingConnection(pika.ConnectionParameters(
      host=rabbitmq_host, credentials=cred))
17 channel = connection.channel()
18
19 channel.queue_declare(queue=queue_to_filter,
      durable=True)
20 channel.queue_bind(exchange='amq.topic',
      queue=queue_to_filter, routing_key=rabbitmq_topic)
22
23 def callback(ch, method, properties, body):
24     readings = json.loads(body)
25
26     if float(readings["Temperature"]) >= threshold_temp:
27         message = {}
28         message["DeviceId"] = readings["DeviceId"]
29         message["Timestamp"] = readings["Timestamp"]
30         message["Temperature"] = readings["Temperature"]
31
```

[9]La relativa immagine nel repository Docker Hub è `luigimaio/nebbiolo:data-filter`

```
32          channel.basic_publish(exchange='amq.topic',
                routing_key='Tierra.FilteredData.Temp',
                body=json.dumps(message))
33          print(' Message published: ' +
                json.dumps(message))
34
35      if float(readings["Pressure"]) >= threshold_pres:
36          message = {}
37          ...
41          channel.basic_publish(exchange='amq.topic',
                routing_key='Tierra.FilteredData.Pres',
                body=json.dumps(message))
43
44      if float(readings["Humidity"]) >= threshold_hum:
45          message = {}
46          ...
50          channel.basic_publish(exchange='amq.topic',
                routing_key='Tierra.FilteredData.Hum',
                body=json.dumps(message))
53
54  channel.basic_consume(callback, queue=queue_to_filter,
        no_ack=True)
55
56  try:
57      channel.start_consuming()
58  except KeyboardInterrupt:
59      channel.stop_consuming()
60
61  connection.close()
```

Per connettersi a RabbitMQ l'applicazione fa uso del protocollo nativo AMQP attraverso la libreria "`pika`". Una volta stabilita la connessione con il broker e aver aperto un canale (righe 16-17), l'applicazione dichiara la coda da cui consumare i messaggi (riga 19) e ne esegue il binding (riga 20). La dichiarazione della coda ed il binding della stessa sono operazioni che hanno effetto solo alla prima esecuzione. Se la coda è già configurata nel message broker le due operazioni non apportano nessuna variazione.

Alla riga 54 viene associata la coda da cui consumare alla `callback` definita alla riga 23. Per ogni messaggio consumato dalla coda ne viene estratto il contenuto (variabile "`body`"). Si ricorda che ogni messaggio pubblicato dai dispositivi contiene vari campi contenenti le misure di temperatura, umidità, pressione, un identificativo del dispositivo e un timestamp di emissione della misura. Per ogni messaggio

consumato vengono paragonate le misure con una soglia: se la singola misura supera la soglia viene creato un nuovo messaggio contenente la misura che ha ecceduto la soglia, unitamente al timestamp della misura e all'identificativo del dispositivo. Questo nuovo messaggio viene pubblicato sempre nell'exchange "`amq.topic`" con una *routing key* differente a seconda di che tipo di misura si tratti.

I messaggi generati dall'applicazione non finiscono su nessuna coda del RabbitMQ presente nel fogNode. Questi messaggi arrivano all'exchange "`amq.topic`" per il quale è stata configurata una shovel che "muove" questi messaggi su un'istanza di RabbitMQ sul Cloud (Sezione 7.2.1).

7.3 Cloud

La componente Cloud del sistema di telemetria è stata implementata su una macchina remota, sulla quale è stato installato Docker. In questo modo è stato possibile riutilizzare le immagini Docker realizzate per il fogNode. Ovviamente la configurazione dei container sul Cloud è leggermente differente, ma sfruttando la versatilità delle immagini sviluppate è stato possibile effettuare queste configurazioni nella fase di deployment dei container specificando le variabili d'ambiente da passare al comando "`docker run`".

Nel RabbitMQ presente sul Cloud sono state configurate tre code che rispecchiano la destinazione dei dati prodotti dall'applicazione Data Filter sul fogNode. Le tre code contengono le misure di temperatura, umidità e pressione che superano una certa soglia e sono rispettivamente "`Tierra.Telemetry.FilteredTemp`, "`Tierra.Telemetry.FilteredHum`", "`Tierra.Telemetry.FilteredPres`". Tutte e tre le code hanno un binding con l'exchange "`amq.topic`" e le loro binding key sono rispettivamente "`Tierra.FilteredData.Temp`", "`Tierra.FilteredData.Hum`" e "`Tierra.FilteredData.Pres`".

Si ricorda che l'istanza di RabbitMQ presente sul fogNode ha una *shovel* configurata per "muovere" tutti i messaggi che confluiscono sull'exchange "`amq.topic`" con una *routing key* "`Tierra.FiltereData.#`" sul corrispettivo exchange dell'istanza sul Cloud. Considerata la wildcard presente nella routing key, tutti i messaggi prodotti dall'applicazione Data Filter sul fogNode rientrano tra quelli da "spostare" per mezzo della shovel. Questo permette il trasferimento automatico dei soli messaggi filtrati dal fogNode al Cloud.

Sul Cloud è presente anche uno stack ELK[10] configurato per consumare da queste tre code e mostrare un aggregato giornaliero, per dispositivo, di quante misurazioni di temperatura, umidità e pressione eccedono le soglie imposte (Figura 7.3). Anche in questo caso le diverse visualizzazioni create e la dashboard che le raccoglie sono

[10]La relativa immagine nel repository Docker Hub è `luigimaio/nebbiolo:elk-rmq-cloud`

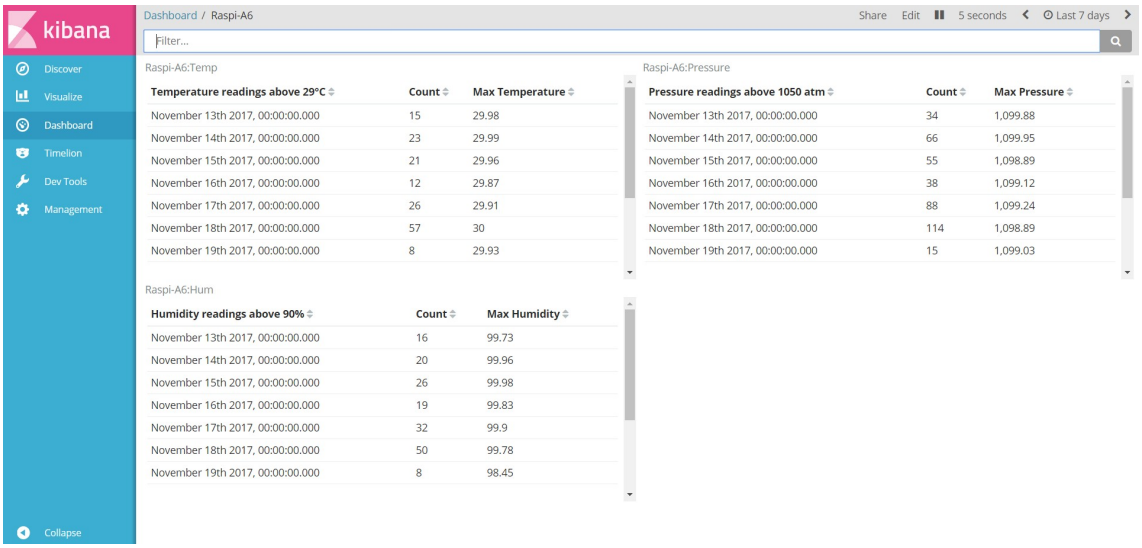


Figura 7.3. Screenshot dell'interfaccia di Kibana in esecuzione sul Cloud

state esportate da Kibana ed incluse nel progetto in modo da poter essere facilmente importate su una nuova istanza di ELK.

7.4 Sincronizzazione del tempo

Come descritto in Sezione 4.4, il Bundle Protocol utilizza il tempo reale per esprimere il timestamp di creazione di un bundle e la sua validità come offset da quest'ultimo. Questo implica che ogni nodo della rete DTN debba essere sincronizzato, almeno grossolanamente. In caso contrario i bundle potrebbero essere scartati dai nodi riceventi durante l'inoltro o eliminati dai nodi che ne custodiscono una copia se il campo "*lifetime*" risultasse erroneamente passato.

I Raspberry Pi non sono dotati di un modulo RTC – *Real Time Clock* – per cui ad ogni spegnimento si limitano a salvare l'ultimo valore temporale. Questo valore viene utilizzato per ripristinare data e ora ad una successiva accensione, a meno che il dispositivo si connetta ad una rete con accesso ad Internet per poter riallineare l'orario di sistema. Nel sistema creato i Sensing Edge Device si connettono alla rete Ad-Hoc con gli altri peer e alla rete del fogNode, quando questa è disponibile, e nessuna delle due offre accesso ad Internet.

Per ovviare al problema e mantenere una sincronizzazione approssimativa del tempo viene utilizzato il protocollo NTP – *Network Time Protocol*. È stato configurato un NTP Server sulla OTVM1 all'interno del fogNode. Su ogni Raspberry Pi è stato configurato un NTP Client che sincronizza l'ora con quella del fogNode.

All'accensione dei dispositivi è sufficiente porli nelle vicinanze del fogNode per fare in modo che questi aggiornino l'ora di sistema con un valore sufficientemente preciso.

Capitolo 8

Validazione

In questo capitolo vengono presentati i risultati dei test condotti sul sistema di comunicazione proposto e sviluppato nell'ambito di questa tesi. I test sono stati effettuati sia con i prototipi dei dispositivi realizzati, che per mezzo di una simulazione condotta con il sistema di orchestrazione *Universal Node*.

8.1 Test con dispositivi fisici

In questa sezione viene illustrato il test effettuato con i due prototipi di Sensing Edge Device. Si tratta di una prova volta a stimare il tempo necessario a due dispositivi per effettuare il *discovery* di un vicino durante un contatto opportunistico.

8.1.1 Banco prova

Come descritto in Sezione 7.1, i dispositivi utilizzati sono due *Raspberry Pi Model B+ v1.2* con 512MB di RAM, mentre come sistema operativo è stata utilizzata la distribuzione *Raspbian GNU/Linux 9.1 (stretch)* con kernel *4.9.41*. Ogni Raspberry Pi è equipaggiato con un dongle Wi-Fi *TP-Link TL-WN725N* con chipset Realtek *RTL8188EU*, i cui driver sono stati personalizzati e compilati a partire da un fork dei sorgenti originali come descritto in Sezione 7.1.1. La versione del software *IBR-DTN*, compilato dai più recenti sorgenti disponibili al momento dei test, è la *v1.0.1*.

È stato inoltre utilizzato un laptop per effettuare delle catture del traffico Wi-Fi scambiato tra i due dispositivi. Il laptotp è equipaggiato con un *Wi-Fi USB Adapter* con chipset *Realtek 8187L* e con antenna omnidirezionale da *5dBi*. Le catture sono state effettuate utilizzando la distribuzione *Kali Linux 2017.2* e il software *Wireshark v2.4.1*, ponendo la scheda Wi-Fi in "*Monitor Mode*" per mezzo del tool `airmon-ng`.

Infine è stato utilizzato un comune forno a microonde per emulare con ragionevole precisione il momento di contatto tra i due dispositivi: la schermatura del forno permette di isolare il dispositivo posto al suo interno fino al momento dell'apertura

dello sportello, che corrisponderà all'istante di "contatto" tra i due Sensing Edge Device.

8.1.2 Connessione opportunistica dei dispositivi

Dopo aver inserito uno dei dispositivi nel microonde è stata fatta partire la cattura con il laptop posto nelle immediate vicinanze. Precedentemente in entrambi i dispositivi sono stati attivati il gateway MQTT-to-DTN e l'applicazione publisher per iniziare a generare bundle. Infine è stato aperto lo sportello del forno a microonde per emulare il momento di contatto tra i due dispositivi. Dall'analisi delle numerose catture effettuate, si evince che mediamente il tempo necessario a due dispositivi per instaurare il link fisico e successivamente effettuare la reciproca scoperta a livello di bundle layer si aggira nell'ordine dei 2,5 secondi con un'incertezza di circa $\pm$ 0,5s su un campione di 16 catture.

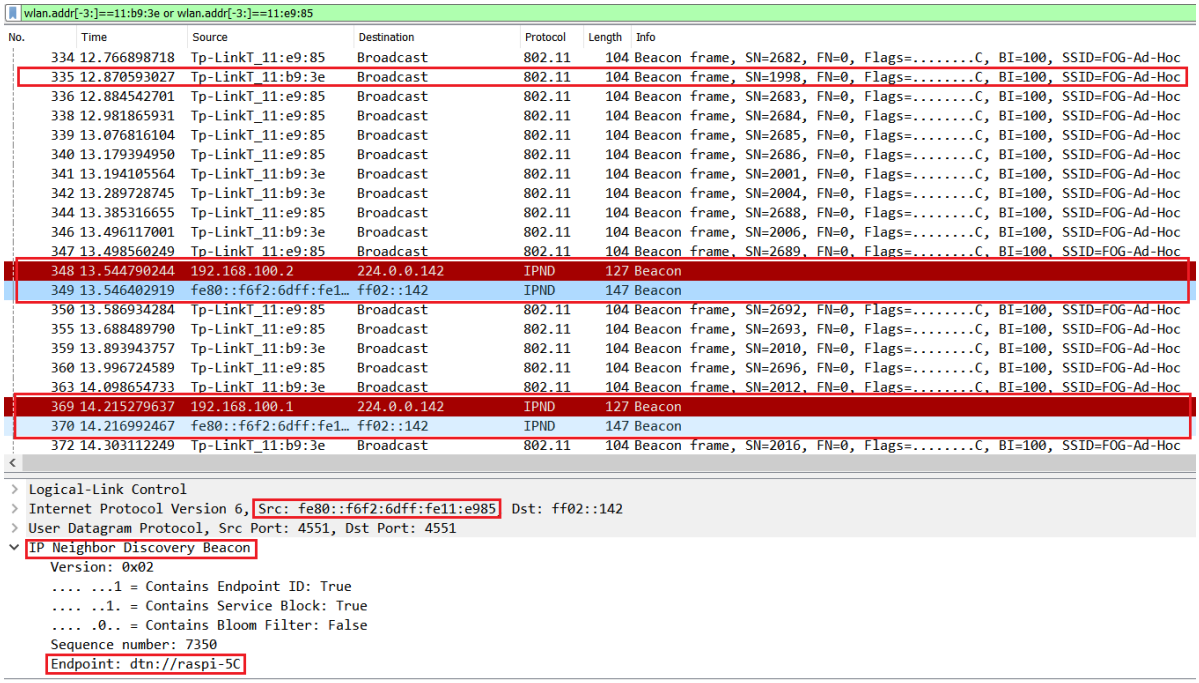


Figura 8.1. Cattura del traffico tra due Sensing Edge Device: i due dispositivi effettuano dapprima il processo di IBSS merge per stabilire il link fisico e successivamente scambiano i beacon IPND per annunciarsi a vicenda come nodi della DTN

In Figura 8.1 è mostrata una delle catture effettuate. Il frame 335 è il primo proveniente dal dispositivo precedentemente isolato: si può approssimativamente assumere come l'istante di apertura dello sportello. Da quel momento i due dispositivi

si scambiano vari *IEEE 802.11 Beacon Frame* che annunciano la stessa rete Ad-Hoc, avviene dunque il cosidetto *IBSS merge* durante il quale concordano il BSSID da utilizzare per la trasmissione. Si noti che i due dispositivi non inviano frame IEEE 802.11 di *Probe Request/Response* in quanto le loro interfacce di rete erano già attive prima del "contatto".

Successivamente i due dispositivi iniziano a scambiarsi i *beacon IPND*, da non confondersi con i Beacon Frame: si tratta di pacchetti UDP inviati in multicast al fine di annunciare la presenza del nodo nella DTN e permette ai nodi in ascolto di effettuare il *binding IP-EID*. In fondo alla Figura 8.1 sono evidenziati i dettagli contenuti in un beacon IPND: l'indirizzo IPv4/IPv6 del nodo che si annuncia può essere ricavato dall'intestazione del pacchetto IP che incapsula il beacon, mentre l'EID è contenuto nel beacon stesso. Nella cattura in questione il primo frame fisico inviato dal nodo isolato è all'istante *12,870*, mentre il processo di discovery a livello di bundle layer può essere considerato concluso all'istante *14,216* (frame 370); sono stati necessari approssimativamente 2 secondi prima che i due dispositivi siano stati messi in condizione di potersi scambiare dei bundle.

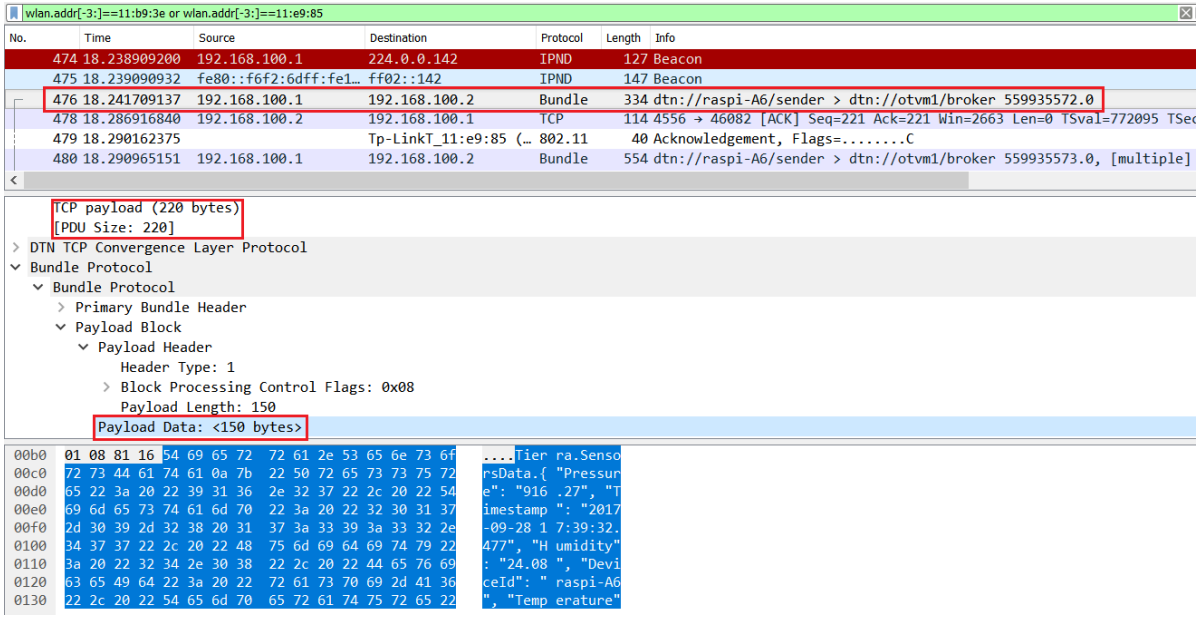


Figura 8.2. Cattura del traffico tra due Sensing Edge Device: i due dispositivi iniziano lo scambio dei bundle successivamente alla fase di discovery del protocollo IPND

In Figura 8.2 viene mostrato il seguito della cattura precedente: viene scambiato il primo bundle all'istante *18,241* della cattura, circa 6 secondi dopo il contatto. Il non immediato invio di bundle dopo la reciproca scoperta dei nodi è imputabile

alle logiche di routing del Bundle Protocol Agent, in quanto i bundle scambiati non sono direttamente destinati al reciproco vicino ma al fogNode; si tratta di bundle inoltrati ai nodi vicini nella speranza che viaggino verso la destinazione. Dalla figura si può inoltre notare come l'overhead dovuto all'inserimento del Bundle Layer per questo messaggio è di 70 byte.

8.2 Test con sistema di orchestrazione Universal Node

Al fine di eseguire dei test in modo controllato e con un numero maggiore di nodi è stato sviluppato uno script di simulazione basato sul sistema di orchestrazione Universal Node (Sezione 5.4). Il fogNode e i Sensing Edge Device sono stati dunque modellizzati come *Virtual Network Functions* (*VNF*), mentre i contatti tra i nodi vengono gestiti attraverso delle regole *OpenFlow* specificate per mezzo di un *Forwarding Graph* (*NF-FG*) istanziato sullo Universal Node. Questo grafo viene continuamente aggiornato dallo script di simulazione per controllare i collegamenti tra i nodi. I sorgenti dello script e le immagini Docker cui si fa riferimento in questa sezione si trovano nel repository del progetto: MadLuigi/fog-over-dtn.

8.2.1 Ambiente di simulazione

La simulazione è stata eseguita su una workstation rack del Netgroup le cui specifiche sono riportate di seguito:

Workstation Winter
CPU: 2x Octa-core Intel Xeon E5-2660 @ 2.2Ghz
RAM: 128GB (8x 16GB RDIMM @1866MT/s)
OS: Ubuntu 14.04 LTS (kernel 3.13.0-112)
Hypervisor: KVM 2.0.0

All'interno della workstation è stata istanziata una macchina virtuale su cui è stato installato lo Universal Node. Di seguito vengono riportate le specifiche della VM:

VM Universal Node
CPU: 16 V-CPU
RAM: 12GB
OS: Ubuntu 16.04.2 LTS (kernel 4.4.0-96)
Network controller: Open vSwitch v2.6.0
Compute controller: Docker v17.09.0-ce

Poiché lo Universal Node è stato configurato per utilizzare Docker come *compute controller*, le VNF che modellizzano fogNode e Sensing Edge Device non sono altro che immagini Docker appositamente create per eseguire in maniera automatizzata tutto lo stack software presente nei nodi fisici. È stata creata un'immagine chiamata "*base-dtn-node*" che contiene il Bundle Protocol Agent IBR-DTN; quest'immagine è stata poi estesa andando a specializzare quelle del fogNode e dei dispositivi, ciascuna con il software sviluppato per la realizzazione del prototipo.

Normalmente i container istanziati dallo Universal Node non sono associati a nessuna rete. È lo stesso orchestratore che per mezzo del network controller si occupa di creare i collegamenti tra le VNF. Lo Universal Node, però, offre la possibilità di specificare nel *template* che descrive una VNF l'opzione "`unify-control`", che crea all'interno del container un'interfaccia collegata alla rete "*bridge*" di Docker. Quest'opzione è stata utilizzata all'interno del template che modellizza il fogNode per farlo comunicare con i container di RabbitMQ e dello stack ELK, che sono stati istanziati separatamente.

L'immagine che modellizza il fogNode, dunque, contiene al suo interno il software IBR-DTN, il gateway DTN-to-MQTT e comunica direttamente con il container di RabbitMQ. L'immagine che modellizza i Sesnsing Edge Device, invece, contiene il software IBR-DTN, il gateway MQTT-to-DTN e due versioni del publisher: una prima versione è quella che pubblica i messaggi MQTT attraverso il gateway, la seconda versione invece utilizza l'indirizzo del container di RabbitMQ per pubblicare direttamente i messaggi attraverso il protocollo MQTT. Ciò permette di eseguire dei test comparativi tra l'inoltro attraverso il Bundle Protocol e la pubblicazione diretta nel caso in cui vi sia una topologia connessa.

8.2.2 Script di simulazione

Lo script di simulazione, realizzato in Python, permette di gestire la creazione e l'aggiornamento del *Forwarding Graph* (NF-FG), che specifica i nodi e i collegamenti tra essi, attraverso le REST API esposte dallo Universal Node. Grazie a questo script viene dunque gestito l'intero ciclo di vita delle VNF e l'evoluzione della topologia nel tempo.

Lo script di simulazione mette a disposizione diversi parametri per configurare il test. Di seguito vengono riportati quelli principali:

- ***RUNNING_TIME***: indica la durata della simulazione in secondi;
- ***GENERATION_TIME***: indica l'intervallo, in secondi, di generazione dei messaggi da parte di ciascun Sensing Edge Device;
- ***NUM_RASPI***: indica il numero di Sensing Edge Device da istanziare per la simulazione;

- ***S***: indica la probabilità che esista un collegamento tra due Sensing Edge Device durante l'iterazione *i-esima*;
- ***T***: indica la probabilità che esista un collegamento tra il fogNode ed un Sensing Edge Device durante l'iterazione *i-esima*;
- ***RANDOM_SEED***: specifica il seme da utilizzare per il generatore di numeri pseudo-casuali in modo da poter rendere i test ripetibili;
- ***MQTT_DIRECT***: determina se i nodi devono pubblicare direttamente i messaggi per mezzo del protocollo MQTT (*True*) o se utilizzare il meccanismo di inoltro per mezzo del gateway MQTT-to-DTN (*False*).

Prima dell'inizio della simulazione lo script istanzia un NF-FG con tutte le VNF necessarie, ovvero un fogNode e tanti Sensing Edge Device quanti specificati dal parametro "`NUM_RASPI`". I container Docker che implementano le VNF non sono inizialmente connessi tra loro ed i publisher all'interno dei Sensing Edge Device vengono avviati dallo script l'istante prima dell'inizio della simulazione. Di seguito si riporta lo pseudo codice del ciclo che la implementa:

Require: $S, T \in [0,1),\ S < T$
  repeat
    for each (i, j) **do**
      $x_{ij} \Leftarrow random() \in [0,1)$
      if $i \vee j$ **is** fogNode **then**
        $P \Leftarrow T$
      else
        $P \Leftarrow S$
      end if
      if $x_{ij} < P$ **and** $\exists!\ link(i, j)$ **then**
        $add\_link\_between\_vnfs(i, j)$
      else if $x_{ij} \geq P$ **and** $\exists\ link(i, j)$ **then**
        $delete\_link\_between\_vnfs(i, j)$
      end if
    end for
    $update\_nffg()$
  until *simulation end*

Ad ogni iterazione viene generato per ogni coppia di nodi un numero pseudo-casuale x_{ij} che determina se il collegamento tra i due nodi in questione deve essere creato o eliminato. Le funzioni "`add_link_between_vnfs`" e "`delete_link_between_vnfs`" implementano la logica per aggiungere o eliminare i collegamenti tra le VNF manipolando il Forwarding Graph. Alla fine dell'iterazione lo script aggiorna il grafo

istanziato sullo Universal Node che modificherà opportunamente i collegamenti per mezzo del network controller.

8.2.3 Tempo medio di consegna di un bundle

Le statistiche riguardanti il tempo di consegna dei bundle vengono calcolate direttamente dai messaggi ricevuti dallo stack ELK. Ogni messaggio pubblicato, infatti, contiene un campo "`Timestamp`" che indica l'istante in cui è stato generato. Poiché Elasticsearch aggiunge ad ogni messaggio elaborato il metacampo "`@timestamp`" contenente l'istante di ricezione, questo viene utilizzato per determinare il tempo di consegna di ogni singolo messaggio calcolando l'offset tra questi due campi.

Sono state effettuate tre diverse simulazioni della durata di 30 minuti, in cui i nodi sono stati configurati per utilizzare l'algoritmo d'inoltro epidemico ed una frequenza di generazione dei messaggi pari ad 1 secondo. La prima simulazione è stata effettuata con 5 nodi e con le probabilità $S = 0{,}1$ e $T = 0{,}3$; la seconda è stata eseguita con le stesse probabilità ma con 15 nodi; la terza con 15 nodi e con una probabilità $S = 0{,}2$ e T invariata.

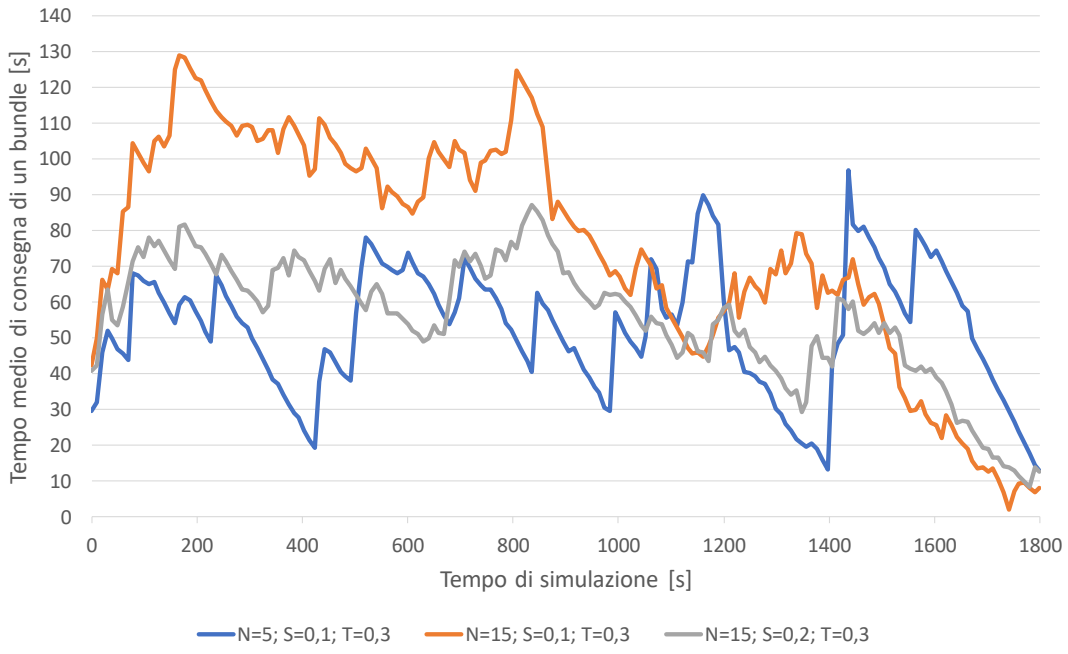


Figura 8.3. Tempo medio di consegna di un bundle

A causa dell'intermittenza dei collegamenti non tutti i messaggi sono stati recapitati durante la simulazione, specie quelli generati negli istanti finali. Questi messaggi

chiaramente non sono persi e in uno scenario reale sarebbero consegnati successivamente. Nel primo caso sono stati recapitati 7644 messaggi su 9000 generati, nel secondo 23379 su 27000 e nell'ultimo caso 24748 su 27000.

Il grafico in Figura 8.3 mostra l'andamento medio del tempo di consegna dei bundle per i tre casi citati. Per ogni istante della simulazione, sulle ordinate è indicato il tempo medio di consegna considerando tutti i nodi partecipanti ed esclusivamente i messaggi recapitati. Si può notare come, a parità dei parametri S e T, tra il primo ed il secondo caso il numero maggiore di nodi peggiori i tempi di consegna. Sebbene un maggior numero di nodi potrebbe garantire una frequenza d'incontri maggiore, aumenta anche il numero di messaggi generati che devono essere consegnati. Se si paragonano il secondo e il terzo caso, a parità di numero di nodi, la maggiore frequenza di contatti tra Sensing Edge Device porta ad un logico miglioramento sia nel numero di messaggi consegnati sia sui tempi medi di consegna.

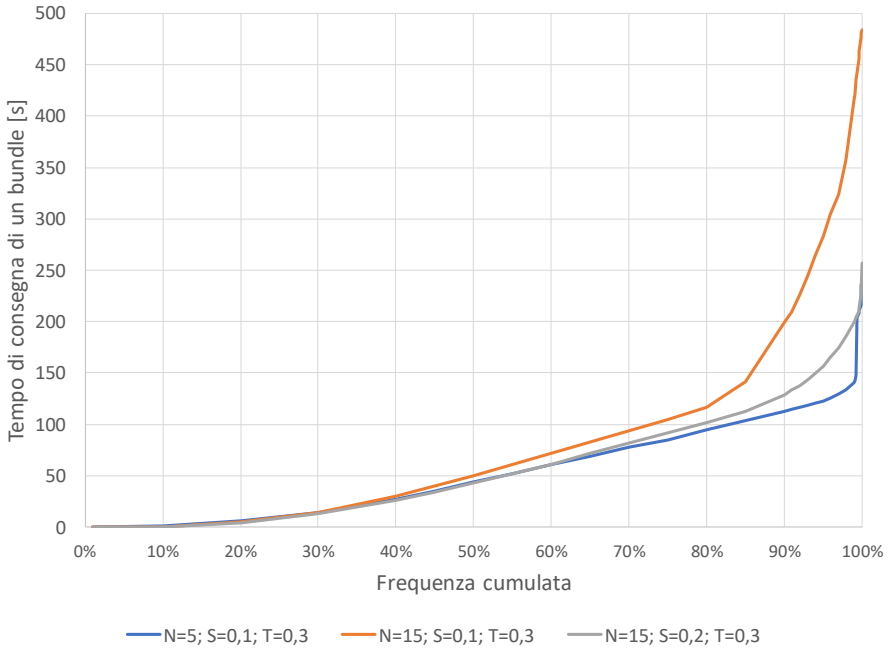


Figura 8.4. Distribuzione di frequenza cumulata dei tempi di consegna di un bundle

La Figura 8.4 mostra invece la distribuzione di frequenza cumulata dei tempi di consegna per gli stessi tre casi. Questo grafico denota una differenza ancora più marcata nei tempi di consegna: se per tutti e tre i casi il 70% dei messaggi è stato consegnato in meno di 100 secondi, nella seconda simulazione il 5% dei messaggi viene consegnato con ritardi superiori ai 300 secondi mentre negli altri due casi nessun messaggio impiega così tanto a raggiungere la destinazione.

8.2.4 Spazio occupato su ciascun nodo

Lo script di simulazione è in grado di calcolare il numero di bundle memorizzati su ciascun nodo[1] durante l'esecuzione della simulazione. Questa funzionalità è implementata da un thread che interroga periodicamente il filesystem di ogni singolo container e produce in output un file CSV. Questo permette di quantificare l'overhead causato dall'algoritmo di inoltro epidemico in termini di spazio occupato per memorizzare le repliche dei bundle su ogni singolo nodo della DTN.

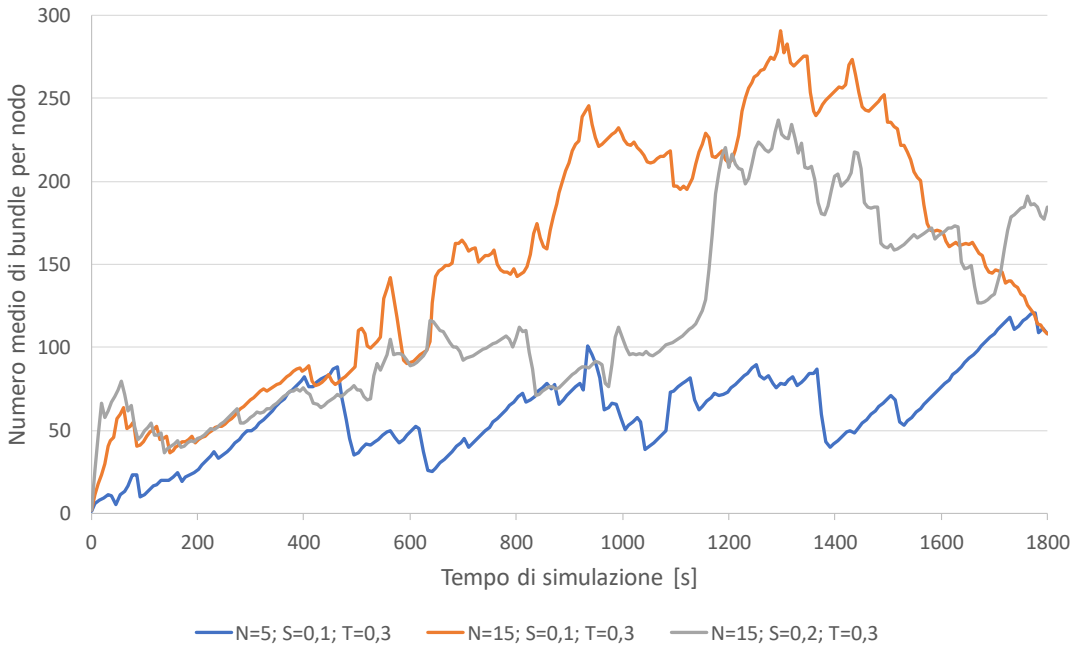


Figura 8.5. Numero medio di bundle memorizzati su ciascun nodo durante la simulazione

In Figura 8.5 è mostrato l'andamento del numero di bundle mediamente presenti su un nodo per gli stessi tre casi testati in precedenza. È evidente come nella simulazione effettuata con soli 5 nodi il numero di bundle mediamente conservati da un nodo sia nettamente inferiore. Le due simulazioni effettuate con 15 nodi presentano un overhead maggiore causato da un numero più elevato di messaggi generati, tuttavia nella simulazione effettuata con una probabilità S maggiore questo overhead risulta essere inferiore. In questo caso la percentuale più elevata di incontri

[1]Si considerano sia i bundle generati dal nodo stesso che copie di bundle da inoltrare per conto dei nodi vicini.

tra Sensing Edge Device oltre a favorire un più rapido smaltimento del traffico ha conseguentemente permesso ai nodi di liberare una maggiore quantità di risorse.

8.2.5 Latenza in caso di topologia connessa

Per calcolare la latenza di consegna in caso di topologia connessa, è stata utilizzata una variante dello script di simulazione che nella fase di inizializzazione istanzia un NF-FG con i collegamenti definiti per ogni coppia di nodi. In questa variante il ciclo che implementa la simulazione si riduce ad una *sleep* della durata specificata dal parametro "`RUNNING_TIME`". Inoltre, specificando opportunamente il parametro booleano "`MQTT_DIRECT`", lo script permette di eseguire una simulazione in cui i nodi utilizzano direttamente il protocollo MQTT per la pubblicazione dei messaggi, piuttosto che l'inoltro tramite Bundle Protocol.

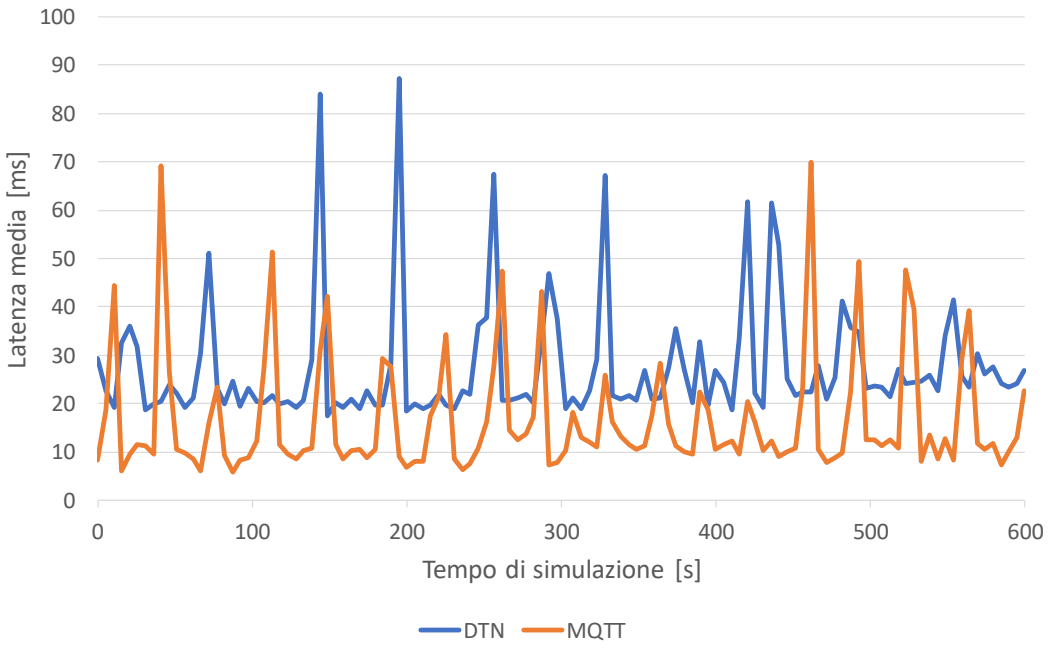


Figura 8.6. Latenzia media a confronto in caso di topologia connessa: DTN vs. MQTT

È stato dunque eseguito un test comparativo per valutare quanto il Bundle Protocol e l'elaborazione aggiuntiva causata dalla coppia di gateway incida nei tempi di consegna nel caso di topologia connessa. Si ricorda che il demone IBR-DTN è stato configurato per preferire l'inoltro diretto verso la destinazione quando questa è presente tra i nodi correntemente raggiungibili. Pertanto, in questo test in cui la

topologia è completamente connessa, non c'è nessun overhead dovuto all'algoritmo di inoltro epidemico ed è quindi possibile effettuare una comparazione.

Il grafico in Figura 8.6 mostra le latenze medie nel caso di pubblicazione diretta tramite MQTT e nel caso di utilizzo del Bundle Protocol. Le due simulazioni in questione sono state eseguite per 10 minuti con 15 nodi ed una frequenza di pubblicazione di 1 messaggio al secondo per ogni Sensing Edge Device. Sebbene la pubblicazione diretta con il protocollo MQTT presenti latenze lievemente inferiori, anche nel caso di utilizzo del Bundle Protocol le latenze si mantengono nell'ordine delle decine di millisecondi.

Capitolo 9

Conclusioni e sviluppi futuri

In questa tesi è stata definita un'architettura che consente di sfruttare le peculiarità delle Delay/Disruption Tolerant Networks per garantire una comunicazione affidabile anche in ambienti sfidanti per la connettività quali possono essere quelli nell'ambito dell'Industrial Internet Of Things. In tali contesti può essere molto vantaggioso introdurre il paradigma del Fog Computing andando ad integrare le attuali infrastrutture con nodi di elaborazione distribuiti all'edge della rete con l'obiettivo di una convergenza tra il mondo delle Information Technologies con quello delle Operations Technologies. L'architettura proposta permette di sfruttare i vantaggi derivanti dall'introduzione del Fog Computing anche nei suddetti contesti in cui è difficile garantire una comunicazione continua ed affidabile tra i nodi che compongono la rete.

Quest'architettura è inoltre trasparente nei confronti degli applicativi esistenti grazie a dei gateway che si occupano di fare da ponte tra il protocollo applicativo e l'astrazione introdotta dal bundle layer. La validazione preliminare condotta sul prototipo realizzato ha messo in evidenza quanto lo sfruttamento di un meccanismo d'inoltro store-carry-and-forward, unitamente alla comunicazione device-to-device e alla mobilità dei dispositivi, possa favorire una maggiore affidabilità ed una più rapida fruizione dell'informazione. Tuttavia sono stati messi in luce alcuni potenziali vincoli all'adozione di una simile architettura, in particolare in termini di risorse di storage di cui devono essere dotati i dispositivi.

La validazione condotta in questa tesi è piuttosto generica e non incentrata su un particolare caso d'uso. È necessario quindi valutare le performance di un simile sistema di comunicazione caso per caso, considerando parametri quali numero di nodi, la loro densità spaziale rispetto all'area operativa, eventuali vincoli sulle risorse di elaborazione e di storage disponibili su ogni dispositivo, le tecnologie utilizzabili a livello fisico per interconnettere i dispositivi e sopratutto la valutazione della variazione nel tempo della topologia di rete.

Se l'infrastruttura di rete è tale da garantire per lo più una topologia connessa, grazie ad esempio all'utilizzo di robuste Wireless Mesh Networks, l'overhead dovuto

all'inoltro basato sull'algoritmo epidemico o su altri algoritmi basati su replica dei bundle potrebbe essere accettabile. In caso di nodi fortemente isolati per lunghi periodi di tempo ed in grado di comunicare solo attraverso contatti occasionali, è bene valutare l'adozione di un algoritmo di routing più efficiente di quello epidemico. Ad esempio, se fosse possibile definire un modello dei contatti, allora l'adozione di un algoritmo di routing stocastico ed adattativo come il PRoPHET potrebbe risultare più parco di risorse garantendo comunque un throughput accettabile.

Un ulteriore aspetto che potrebbe essere fonte di sviluppi futuri è la generalizzazione dell'implementazione dell'architettura definita. Al momento, infatti, si tratta di un prototipo che supporta unicamente il protocollo MQTT. Si potrebbe generalizzare l'implementazione dei gateway in modo da disaccoppiare il parsing dei messaggi applicativi dall'interazione con il Bundle Protocol Agent. Una struttura modulare favorirebbe dunque eventuali sviluppatori intenzionati ad estendere il supporto dell'architettura a nuovi protocolli applicativi.

Bibliografia

[1] James Manyika et al. *Unlocking the potential of the Internet of Things*. McKinsey Global Institute, giu. 2015. URL: http://www.mckinsey.com/business-functions/digital-mckinsey/our-insights/the-internet-of-things-the-value-of-digitizing-the-physical-world.

[2] Peter Middleton. *Forecast Alert: Internet of Things - Endpoints and Associated Services, Worldwide, 2016*. Gartner, Inc., gen. 2017. URL: https://www.gartner.com/doc/3559634/forecast-alert-internet-things-.

[3] Jörg Ott. *Challenged Networks - The Mobile Internet is Not Connected*. 2008. URL: http://www.future-internet.eu/fileadmin/documents/consultation_meeting_31_Jan_08/ott_fp7-fi_2008-01.pdf.

[4] Interplanetary Internet Special Interest Group. *Delay and Disruption Tolerant Networks (DTNs): A Tutorial*. Lug. 2012. URL: http://ipnsig.org/wp-content/uploads/2012/07/DTN_Tutorial_v2.04.pdf.

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

[6] OpenFog Consortium. *Definition of Fog Computing*. URL: https://www.openfogconsortium.org/resources/#definition-of-fog-computing.

[7] OpenFog Consortium. *Fog Computing use cases*. URL: https://www.openfogconsortium.org/resources/#use-cases.

[8] OpenFog Consortium. *Out of the Fog: High-Scale Drone Package Delivery*. Lug. 2016. URL: https://www.openfogconsortium.org/wp-content/uploads/OpenFog-Transportation-Drone-Delivery-Use-Case.pdf.

[9] Nebbiolo Technologies. *Bridging the intelligence gap between the cloud and IoT and end-points*. URL: https://www.nebbiolo.tech/platform/.

[10] Nebbiolo Technologies. *fogNode Overview*. 2017. URL: https://www.nebbiolo.tech/wp-content/uploads/fogNode-OVERVIEW-rev3.pdf.

[11] Nebbiolo Technologies. *Nebbiolo NFN-300 Series fogNode - Datasheet Ver. 1.4*. 2016. URL: https://www.nebbiolo.tech/wp-content/uploads/NFN-300-datasheet-v1.4-2.pdf.

[12] Nebbiolo Technologies. *fogOS Overview*. 2017. URL: https://www.nebbiolo.tech/wp-content/uploads/fogOS-OVERVIEW-rev3.pdf.

[13] OPC Foundation. *OPC Uni ed Architecture*. URL: https://opcfoundation.org/wp-content/uploads/2014/05/OPC-UA_Overview_IT.pdf.

[14] V. Cerf et al. *Delay-Tolerant Networking Architecture*. RFC 4838. IETF, apr. 2007. URL: https://tools.ietf.org/html/rfc4838.

[15] Kevin Fall e Stephen Farrell. «DTN: an architectural retrospective». In: *IEEE Journal on Selected Areas in Communications* (2008). URL: https://doi.org/10.1109/JSAC.2008.080609.

[16] Davide Pallotti. «Estensione dell'Abstraction Layer di DTNperf alle API di IBR-DTN». Tesi. Università di Bologna. URL: http://amslaurea.unibo.it/11329/1/davide_pallotti_tesi.pdf.

[17] Sushant Jain, Kevin Fall e Rabin Patra. «Routing in a delay tolerant network». In: *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*. 2004. URL: https://doi.org/10.1145/1015467.1015484.

[18] Zhensheng Zhang. «Routing In Intermittently Connected Mobile Ad Hoc Networks And Delay Tolerant Networks: Overview And Challenges». In: *IEEE Communications Surveys & Tutorials* (2006). URL: https://doi.org/10.1109/COMST.2006.323440.

[19] Osman Khalid, Rao Naveed Bin Rais e Sajjad A. Madani. «Benchmarking and Modeling of Routing Protocols for Delay Tolerant Networks». In: *Wireless Personal Communications* (2017). URL: https://doi.org/10.1007/s11277-016-3654-5.

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

[21] S. Symington et al. *Bundle Security Protocol Specification*. RFC 6257. IETF, mag. 2011. URL: https://tools.ietf.org/html/rfc6257.

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

[23] D. Ellard et al. *DTN IP Neighbor Discovery (IPND)*. I-D. IETF, apr. 2015. URL: https://tools.ietf.org/html/draft-irtf-dtnrg-ipnd-03.

[24] Dominik Schürmann. *Install IBR-DTN*. URL: https://github.com/ibrdtn/ibrdtn/wiki/Install-IBR-DTN.

[25] *Docker website*. URL: https://www.docker.com/what-docker.

[26] FreeBSD. *Capitolo 15. Jail*. URL: https://www.freebsd.org/doc/it_IT.ISO8859-15/books/handbook/jails-intro.html.

[27] Docker Inc. *Docker container networking*. URL: https://docs.docker.com/engine/userguide/networking/.

[28] CloudAMQP. *What is RabbitMQ?* Ago. 2015. URL: https://www.cloudamqp.com/blog/2015-05-18-part1-rabbitmq-for-beginners-what-is-rabbitmq.html.

[29] RabbitMQ. *Finding bottlenecks with RabbitMQ 3.3*. 2014. URL: http://www.rabbitmq.com/blog/2014/04/14/finding-bottlenecks-with-rabbitmq-3-3/.

[30] Wi-Fi Alliance. *Wi-Fi Peer-to-Peer (P2P) Technical Specification v1.7*. Rapp. tecn. Wi-Fi Alliance, 2016. URL: https://www.wi-fi.org/download.php?file=/sites/default/files/private/Wi-Fi%20P2P%20Technical%20Specification%20v1.7.pdf.

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

[32] *NF-FG examples*. URL: https://github.com/netgroup-polito/un-orchestrator/blob/master/orchestrator/README_NF-FG.md.

[33] OASIS. *MQTT Version 3.1.1*. URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html.

[34] *Cannot receive external multicast inside container - Issue #23659*. URL: https://github.com/moby/moby/issues/23659.