ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica – Scienza e Ingegneria

Corso di Laurea Triennale in Ingegneria Informatica

TESI DI LAUREA
in
Fondamenti di Telecomunicazioni T

Sviluppo di un'applicazione di sincronizzazione file per reti challenged

CANDIDATO:
Nicolò Castellazzi

RELATORE:
Prof. Carlo Caini

Anno Accademico 2015/2016

Sessione I

Indice generale

Prefazione

Questa tesi si pone l'obiettivo di implementare in ambiente Linux un'applicazione di sincronizzazione, chiamata DTNbox, che permetta lo scambio di file tra due nodi di una rete classificabile come Delay-/Disruption-Tolerant Network (DTN), ossia una rete in cui a causa di ritardi, interruzioni, partizionamento, non sia possibile utilizzare l' usuale architettura di rete TCP/IP.

E' evidente che i problemi menzionati rendono estremamente più complessa la sincronizzazione fra cartelle rispetto ad Internet, da cui le peculiarità di DTNbox rispetto ad altre applicazioni in rete visto che, ad esempio, non è possibile la sincronizzazione tramite un nodo centrale, come in Dropbox e similari, ma occorre basarsi su comunicazioni peer-to-peer.

Ad oggi è presente una controparte di questo programma su ambiente Android. Con una tesi precedente è stato redatto un progetto ex novo dell'applicazione per realizzarne una versione Linux ma, date le caratteristiche dell'ambiente Android, la versione Linux è da considerarsi un progetto indipendente, anche se le finalità sono le stesse. Vista la complessità, la tesi precedente aveva definito progetto e struttura del programma, ma solo alcuni blocchi di codice erano stati compiutamente scritti e testati.

L'oggetto della mia tesi si è quindi sviluppato principalmente su tre direzioni:

- Implementare, utilizzando il linguaggio di programmazione C, le funzionalità previste dal nuovo progetto per Linux
- Integrarne e modificarne le parti ritenute carenti, man mano che i test parziali ne hanno mostrato la necessità
- Testarne il suo corretto funzionamento

Si è deciso pertanto di dare precedenza alla scrittura delle parti

fondamentali del programma quali i moduli di controllo, la struttura e gestione del database e lo scambio di messaggi tra due nodi appartenenti ad una rete DTN per poter arrivare ad una prima versione funzionante del programma stesso, in modo che eventuali future tesi possano concentrarsi sullo sviluppo di una interfaccia grafica e sull'aggiunta di nuovi comandi e funzionalità accessorie. Il programma realizzato è stato poi testato su macchine virtuali grazie all'uso dello strumento *Virtualbricks*, per poter avere un ambiente di test uniforme e facilmente replicabile, utilizzando l'implementazione del *bundle protocol* DTN2. Anche se i test sono stati eseguiti su DTN2, grazie alla scelta di utilizzare la libreria *Abstraction Layer* per la comunicazione con il *bundle protocol*, il programma risulta essere indipendente da una sua implementazione specifica.

Capitolo 1: Delay-/Disruption-Tolerant Networking

1. Cenni sul Delay-/Disruption-Tolerant Networking

Ogni giorno sempre più dispositivi presentano la necessità di essere connessi tra loro e potersi scambiare informazioni in maniera affidabile. Tipicamente questa necessità viene soddisfatta attraverso l'utilizzo di Internet.

Internet è l'unico strumento che ad oggi consente di collegare globalmente tra loro milioni di dispositivi, e basa il suo funzionamento sull'utilizzo di una suite di protocolli di comunicazione chiamata suite TCP/IP, che deve il suo nome ai due protocolli principali, il *Transmission Control Protocol* (TCP) e l'*Internet Protocol* (IP). Ogni nodo della rete deve quindi avere un indirizzo IP e deve utilizzare la suddetta suite per poter comunicare con gli altri nodi.

Nel progetto del TCP/IP (fine anni '70) sono state fatte delle assunzioni, molto ragionevoli in un contesto di interconnessioni terrestri, secondo le quali per ogni coppia di nodi esiste sempre una connessione bidirezionale end-to-end, il tasso di errore per bit sui canali di trasmissione è basso e quindi di conseguenza è trascurabile la perdita di pacchetti dati dovuta a violazione dei controlli di parità (un pacchetto con uno o più bit errati viene scartato). Infine si assume che i tempi di *roundtrip* (RTT, Round Trip Time, ovvero tempo trascorso fra l'invio di un pacchetto e la ricezione di una sua conferma) siano bassi (di norma su Internet sono minori di 200 ms, escluse le comunicazioni via satellite geostazionario, per le quali RTT=600ms).

Se però in una rete una o più di queste condizioni viene a mancare, la suite di protocolli TCP/IP presenta forti limitazioni e risulta inadatta allo scambio di messaggi.

Vengono quindi definite come reti *challenged* tutte quelle reti che presentano:

- Connettività intermittente: non è sempre possibile avere una connessione end-to-end tra due nodi (*network partitioning*)
- Ritardi significativi e variabili: possono passare anche molte ore o giorni tra l'invio di un pacchetto e la ricezione della sua conferma di trasmissione
- Canale di comunicazione asimmetrico: se c'è una grossa differenza tra la quantità di dati che è possibile trasmettere in un verso del canale rispetto al verso opposto non è possibile utilizzare protocolli discorsivi come il TCP
- Alta percentuale di errore: a causa di disturbi o altri problemi sul canale di trasmissione è probabile che i pacchetti arrivino a destinazione presentando errori, ed è quindi necessaria la loro ritrasmissione da parte della sorgente

E' inoltre a volte necessario dover connettere tra loro reti eterogenee, che non utilizzano la stessa suite di protocolli di comunicazione e che hanno caratteristiche intrinseche molto differenti.

Per trovare una soluzione a tutti questi problemi Kevin Fall introdusse nel 2003 il concetto di Delay-/Disruption-Tolerant Networking (DTN), estendendo l'ambito di applicazione degli studi sulla Interplanatery networking (IPN), iniziati poco prima su impulso di Vinton Cerf, uno dei due autori del TCP/IP.

L'architettura DTN [RFC4838] risolve i problemi presentati dalle reti *challenged* utilizzando lo *store-(carry-)and-forward message switching.*

Rispetto alla suite di protocolli TCP/IP dove ogni pacchetto viene tenuto in memoria da ogni nodo per brevissimo tempo, cioè quello strettamente necessario alla ricezione ed immediato successivo invio, nello *store-and-forward message switching* ogni messaggio (spesso molto più grande di un pacchetto IP) viene spostato (*forward*) dalla memoria (*store*) di un nodo all'altro fino a quando non viene consegnato al destinatario. Il messaggio

non verrà tenuto in memoria per qualche millisecondo all'interno di un *buffer* ma rimane memorizzato su di una memoria (a richiesta non volatile, per aumentare la robustezza) fino a quando non è stato trasmesso con successo al nodo successivo per raggiungere il destinatario. Nel caso in cui il collegamento verso il destinatario sia interrotto, il messaggio può rimanere memorizzato per lungo tempo (anche ore), in attesa che il collegamento sia ripristinato.

L'architettura DTN implementa lo *store-and-forward message switching* creando un nuovo protocollo di trasmissione, detto *bundle protocol* [RFC5050], che si va a inserire tra il livello *Application* e il livello *Transport* dello stack TCP/IP (Figura 1).

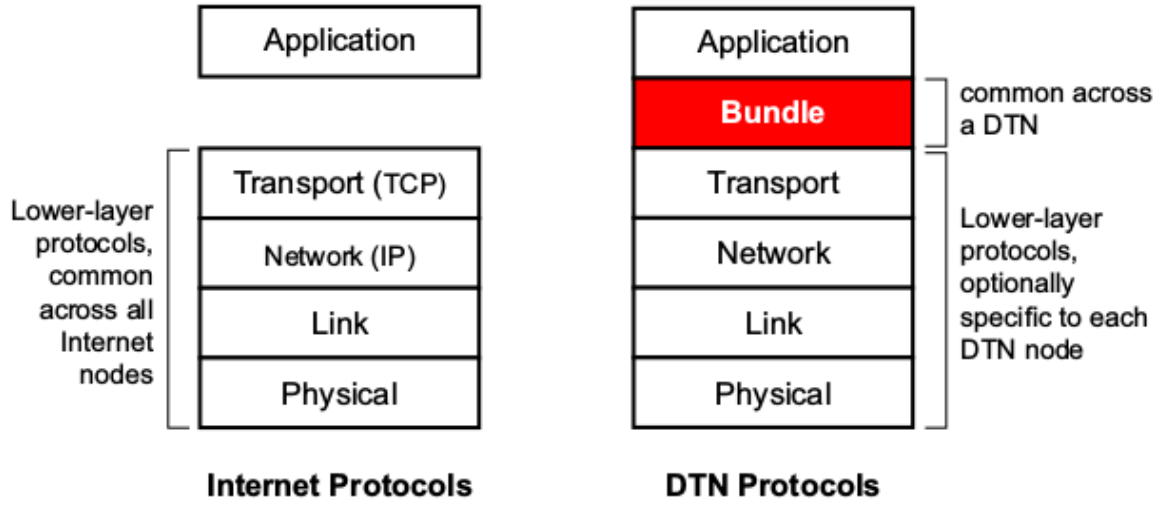

Figura 1: Bundle Protocol nello stack TCP/IP [Warthman]

Come accennato precedentemente, e come si può vedere in Figura 1, può essere necessario dover mettere in comunicazione tra loro reti che utilizzano protocolli di comunicazione di basso livello diversi tra loro.

Andando un po' più in dettaglio nell'architettura DTN, questo viene realizzato introducendo il *convergence layer adapter* (CLA), che ha il compito di rendere trasparente al *bundle protocol* quali protocolli di basso livello vengono effettivamente utilizzati, consentendo così la

comunicazione anche tra reti che sono molto eterogenee per caratteristiche e protocolli utilizzati (Figura 2).

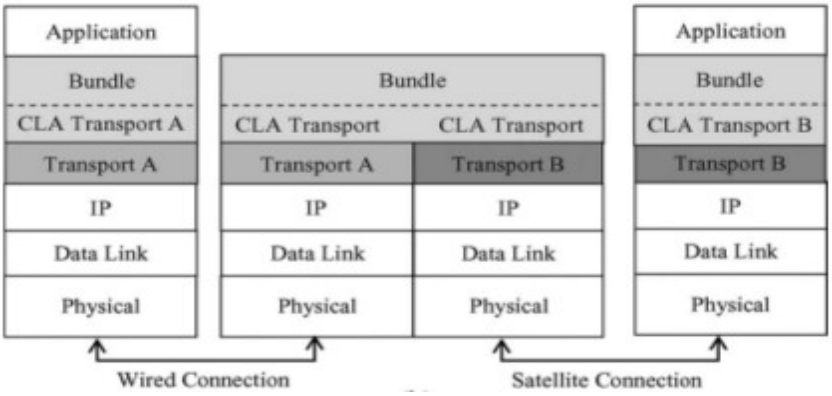

Figura 2: Utilizzo del convercence layer adapter per una comunicazione prima via cavo e poi via satellite [Caini]

Ogni nodo di una DTN deve essere univocamente identificabile per poter essere raggiunto (si noti che i nomi DTN non sono indirizzi, cioè non hanno un valore topologico).

Al momento sono presenti due tipi di *Uniform Resource Identifier* (URI) che sono utilizzati come *Endpoint Identifiers* (EID) e che rispettano il seguente formato:

<nome schema>:<parte relativa allo schema>

Questi sono utilizzati dalle implementazioni del *bundle protocol*: sono lo schema “dtn” e lo schema “ipn”.

Di seguito si evidenziano le loro caratteristiche, in riferimento all'applicazione DTNbox.

- Schema “dtn”, sintassi: “dtn://nodo/DTNbox”

Con riferimento a quanto sopra esplicitato la stringa “nodo” identifica l' *endpoint* “dtn://nodo” mentre la stringa “DTNbox” funziona come *demux token* che specifica a quale applicazione è destinato il messaggio o quale applicazione l'ha generato all'interno del nodo (in pratica l'equivalente della porta nel TCP o nell'UDP).

- Schema “ipn”, sintassi: “ipn:17.1”

In questo caso la stringa “17” identifica univocamente il nodo mentre l'”1” ha la funzione di *demux token* come descritto sopra. Si noti che a differenza dello schema “dtn”, l'EID del nodo nello schema “ipn” non può essere scritto senza indicare anche il *demux token*, per cui si assume 0 come default, ossia nel caso in esempio avremmo “ipn:17.0” (ad esempio per traffico amministrativo del BP).

Grazie quindi all'introduzione del *bundle protocol* e ai *convergence layer adapter* si è riusciti a risolvere gran parte dei problemi presentati dalle reti *challenged* che ora, grazie ad essi, sono in grado di poter comunicare.

Alcuni esempi di utilizzo dell'architettura DTN sono:

- InterPlanetary Networking *(*IPN), dove le enormi distanze e collegamenti end-to-end non sempre disponibili a causa del moto dei pianeti rendono inutilizzabile la suite di protocolli TCP/IP
- Vehicular Ad hoc Networks (VANETs), per la comunicazione tra veicoli e strumenti posti a bordo strada, in quanto i nodi sono in continuo movimento
- Reti di sensori radio
- Unmanned Aerial Vehicle (UAV), per la comunicazione e controllo di veicoli senza pilota
- Comunicazioni sottomarine
- Comunicazioni tattiche militari
- Comunicazioni in assenza di infrastruttura (regioni remote)
- Comunicazioni di emergenza (catastrofi naturali)

2. Implementazioni attuali

Attualmente esistono tre implementazioni principali del *bundle protocol* su piattaforma Linux, che sono nate con scopi leggermente diversi; di una esiste anche la versione per Android.

Le implementazioni sono interoperabili essendo tutte conformi con [RFC5050].

i. DTN2 [DTN2]

Originariamente sviluppata da Kevin Fall presso l'Intel Labs a Berkley, è ora mantenuta da Elwyn Davies, in associazione con Stephen Farrell del Trinity College di Dublino. E' realizzata in C e C++ ed ha lo scopo di fornire una piattaforma di semplice utilizzo per condurre esperimenti sulle DTN, ma può essere utilizzata anche in applicazioni reali. Possiede un buon parco di programmi utili al suo utilizzo quali dtnd, dtnsend, dtnping ecc. Le informazioni di configurazione di ogni nodo sono tutte all'interno di un unico file (es. /etc/dtn.conf). Utilizza preferenzialmente lo schema “dtn”, ma supporta anche quello “ipn”.

ii. ION [ION]

L'Interplanetary Overlay Network (ION) è l'implementazione del *bundle protocol* creato dal Jet Propulsion Laboratory (JPL) della NASA. Ha come scopo la comunicazione interplanetaria e le comunicazioni spaziali ed è stata progettata per funzionare su hardware dalle prestazioni limitate e con limiti di consumo energetico. E' più difficile da utilizzare rispetto a DTN2 in quanto per la configurazione di ogni nodo sono necessari più script e gli strumenti messi a disposizione sono meno intuitivi, ma viene aggiornata e mantenuta con molta più costanza. Supporta sia lo schema “dtn” che lo schema “ipn”, preferendo quando possibile l'utilizzo del secondo.

iii. IBR-DTN [IBR_DTN]

Questa implementazione nasce per sistemi *embedded* e Android, viene attualmente sviluppata presso il Technische Universität Braunschweig. Può essere utilizzata come *framework* per lo sviluppo di applicazioni DTN. Per identificare i nodi possono essere utilizzati sia lo schema “dtn” o quello “ipn”.

Capitolo 2: Descrizione funzionamento DTNbox

1. Descrizione dell'applicazione

Come accennato nell'introduzione, lo scopo principale dell'applicazione DTNbox è quello di permettere la condivisione e la sincronizzazione di file tra due o più nodi appartenenti ad una rete DTN.

Le difficoltà introdotte dalle reti DTN hanno portato ad una discreta complessità del programma.

Le caratteristiche chiave dell'applicazione sono:

- Protocollo di comunicazione peer-to-peer: non è necessaria la presenza di un server sempre raggiungibile, due nodi che hanno una sincronizzazione attiva comunicano direttamente
- Non è richiesta la presenza di un percorso continuo end-to-end tra due nodi in quanto, attraverso l'utilizzo del *bundle protocol*, si riesce a ovviare ad interruzioni anche di lunga durata di parti del percorso
- Sempre grazie al *bundle protocol* e grazie al fatto che il programma prevede ritrasmissioni in caso di mancata ricevuta di conferma entro tempi prestabiliti, il protocollo di comunicazione utilizzato è molto robusto a fronte di interruzione dei link, partizionamento della rete, disconnessioni, riavvii, errori e terminazioni involontarie
- Ogni scambio di messaggi tra due nodi è stato ottimizzato, raggruppando i comandi quando possibile e filtrando a monte l'invio degli stessi, per evitare il numero di trasmissioni necessarie in vista di RTT significativi

Tutto questo ha portato alla definizione di un protocollo di comunicazione robusto e alla necessità di utilizzare un database per mantenere una serie di informazioni, quali lo stato delle sincronizzazioni attive, con quali nodi è possibile comunicare, lo stato dei comandi e lo stato dei file.

Il suo uso è quindi adatto a tutti gli scenari di reti *challenged* visti nel

capitolo precedente, ma non è necessariamente limitato a questi (se la rete non è *challenged* l'applicazione funziona ovviamente lo stesso).

2. Protocollo di comunicazione

Il programma prevede due tipi di sincronizzazione di cartelle, una sincronizzazione bilaterale che prevede lo scambio di file solo tra due nodi (caso più semplice) e una sincronizzazione multilaterale che prevede lo scambio di file tra più nodi. La seconda è un'estensione della prima.

i. Sincronizzazione bilaterale

La sincronizzazione bilaterale coinvolge esclusivamente due nodi e può essere di tre tipi:

1. Sincronizzazione di tipo PUSH:

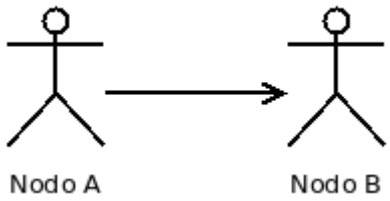

Figura 3: Sincronizzazione di tipo PUSH

La sincronizzazione di tipo PUSH è asimmetrica (o unidirezionale).

Supponiamo che il nodo A abbia una cartella, ad esempio “foto”, di cui è il proprietario, e decida condividerla con il nodo B.

Il nodo A invierà la richiesta di sincronizzazione PUSH al nodo B e, se accettata, verrà creata per prima cosa in B una cartella “foto” sotto la cartella “nodo A” (“~/DTNbox/foldersToSync/nodoA/foto”, vedi in seguito) quindi A inizierà a trasferire i file all'interno della sua cartella “foto” verso il nodo B, il quale potrà unicamente leggere i file che gli verranno inviati dal nodo A. A è il proprietario dei file sincronizzati, in quanto è lui che ha iniziato e messo a disposizione di B i suoi file. L'eventuale creazione, modifica o cancellazione di un file all'interno della cartella “foto” sul file system del nodo A verranno quindi automaticamente

propagate al nodo B mentre non sarà vero il contrario in quanto il nodo B sarà sempre un ricevitore passivo.

Questo metodo di sincronizzazione può risultare utile ad esempio per avere il backup di una cartella su di un altro nodo o per la distribuzione di contenuti dal nodo A al nodo B.

2. Sincronizzazione di tipo PULL:

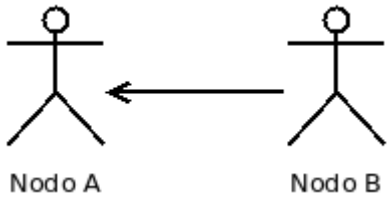

Figura 4: Sincronizzazione di tipo PULL

La sincronizzazione di tipo PULL è anch'essa asimmetrica (o unidirezionale), ma in senso inverso. E' la duale della PUSH, in questo caso il nodo A è il nodo passivo che riceve i file dal nodo B.

Il nodo A può chiedere la sincronizzazione in PULL di una cartella di B, che resta il proprietario dei file sincronizzati, ed eventuali modifiche o cancellazioni del file verranno solo propagate dal nodo B verso il nodo A.

3. Sincronizzazione di tipo PUSH&PULL:

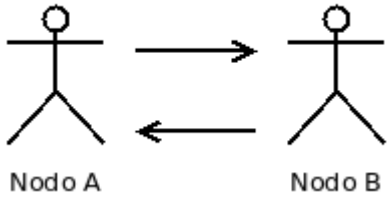

Figura 5: Sincronizzazione di tipo PUSH&PULL

La sincronizzazione di tipo PUSH&PULL è simmetrica (o bidirezionale).

Quando due nodi hanno una sincronizzazione di tipo PUSH&PULL le modifiche effettuate su di un file all'interno della cartella oggetto della sincronizzazione vengono propagate da un nodo all'altro

indipendentemente da chi sia l'autore della modifica ma tenendo conto solo dell'ultima data di modifica del file.

Ogni tipo di sincronizzazione per motivi di sicurezza prevede l'utilizzo opzionale di una password in lettura in scrittura e, in caso di richiesta di sincronizzazione, ogni nodo dovrà inviare al nodo mittente una conferma o un rifiuto della suddetta richiesta.

ii. Sincronizzazione multilaterale

Partendo da una sincronizzazione bilaterale è possibile che un altro utente chieda di aggiungersi o venga invitato da uno dei partecipanti , creando così una catena arbitrariamente lunga di sincronizzazioni. Non è necessario che il nuovo nodo chieda la sincronizzazione direttamente al proprietario della cartella ma può anche chiedere la sincronizzazione a qualsiasi nodo che sia già in sincronizzazione con il proprietario.

iii. Terminazione di una sincronizzazione

Ogni nodo può decidere in qualsiasi momento di terminare una sincronizzazione di cui fa parte.

Se chi termina la sincronizzazione è anche il proprietario della cartella sincronizzata, DTNbox provvederà a terminare la sincronizzazione su ogni nodo che ha una sincronizzazione anche indiretta su quella cartella, eliminandone anche i file sul relativo file system. Ad esempio, se A chiede la terminazione della sincronizzazione sulla cartella foto, sul nodo B verrà cancellato il contenuto della cartella “~/DTNbox/foldersToSync/nodeA/foto”. Questo per evitare di lasciare su B cartelle relative a sincronizzazioni terminate. Di conseguenza, se B vuole essere certo di poter mantenere una copia dei file sincronizzati di cui non è il proprietario, deve copiare i file in una sua cartella differente.

Se il nodo che chiede la terminazione della sincronizzazione non è il proprietario, esso semplicemente cancellerà i file dal proprio file system e notificherà all'altro nodo il suo desiderio di non essere più parte della sincronizzazione.

iv. Gestione dei nodi

Ogni nodo può decidere se accettare o meno la richiesta di sincronizzazione proveniente da un altro nodo.

L'accettazione può essere estesa, per semplicità, anche a tutte le successive richieste (per evitare troppe interazioni).

Ogni nodo mantiene quindi memorizzato sul database all'interno delle seguenti liste il comportamento da tenere verso l'altro nodo:

1. Whitelist: Se un nodo è all'interno di questa lista tutte le richieste da lui provenienti verranno sempre accettate
2. Blacklist: Ogni richiesta proveniente da un nodo in questa lista viene automaticamente scartata ignorandone il contenuto
3. BlockedByList: Ogni nodo che è stato inserito in Blacklist aggiungerà il nodo che l'ha bloccato in BlockedByList e non gli invierà più nessun messaggio fino a quando non verrà sbloccato. Questo porta ad un notevole risparmio di risorse

E' prevista inoltre un altro tipo di lista, denominata Frozenlist, che ha lo scopo di memorizzare tutti i nodi che sono nello stato *frozen*, nel quale le sincronizzazioni da e verso quel nodo sono momentaneamente bloccate (ad esempio perché è stato spento il terminale).

v. Scambio di messaggi

Due nodi comunicano tra loro scambiandosi un file compresso in formato *Tape Archive* (TAR).

Il nome del file sarà composto dal nome del nodo ricavato dall'EID del mittente e dalla data di creazione in millisecondi del tar stesso a partire da EPOCH (mezzanotte UTC del 1 gennaio 1970).

Se ad esempio, utilizzando lo schema “dtn”, il nodo “dtn://nodoA.dtn/DTNbox” vuole inviare un tar all'istante 1463068193 il nome risultante del tar inviato sarà: “nodoA-1463068193.tar”.

All'interno di questo tar saranno presenti uno o più file, in particolare:

- Un file di segnalazione, con estensione “.dbmc” e che conterrà, in formato testuale e uno per riga, tutti i comandi destinati a quel nodo. In maniera simile al tar il nome di questo file sarà composto dalla data di creazione e dal nome del nodo che l'ha creato (seguendo l'esempio precedente: “nodoA-1463068193.dbcm”)
- Eventuali file dati (da creare ex novo o da aggiornare nel nodo ricevente)

Si è deciso di raggruppare tutti i comandi all'interno di un unico file in quanto, come già menzionato, le reti DTN hanno un RTT che può essere molto alto e così facendo è possibile minimizzare il tempo necessario per lo scambio di messaggi, inoltre è necessario limitare il numero di iterazioni al minimo, a causa dei possibili elevati ritardi. Tuttavia, in alcuni casi, è necessario che al mittente arrivi conferma dell'esecuzione dei comandi inviati. DTNbox controllerà periodicamente se vi sono dei comandi che richiedono risposta non confermati alla scadenza di un tempo limite; in questo caso verranno ritrasmessi per un numero prefissato di volte. La sincronizzazione tiene quindi traccia per ogni file che deve essere sincronizzato e per ogni comando, in apposite tabelle del database, del loro stato di corretta ricezione da parte del destinatario (più dettagli a riguardo verranno forniti in seguito).

vi. Comandi

Ogni comando inviato e ricevuto è composto rispettando il seguente scherma:

nome comando \t timestamp \t dettagli specifici \n

Le stringhe sono quindi separate tra loro dal carattere “\t” e ogni comando viene terminato dal carattere “\n”.

Il campo “timestamp” è la data di creazione del comando a partire da EPOCH.

Un comando, in ogni determinato istante di tempo, può essere in uno dei 5 possibili stati previsti dal protocollo a seconda della fase di elaborazione in cui esso si trova.

Gli stati possibili sono: PROCESSING, DESYNCHRONIZED, PENDING, CONFIRMED, FAILED (Figura 6 e Figura 7).

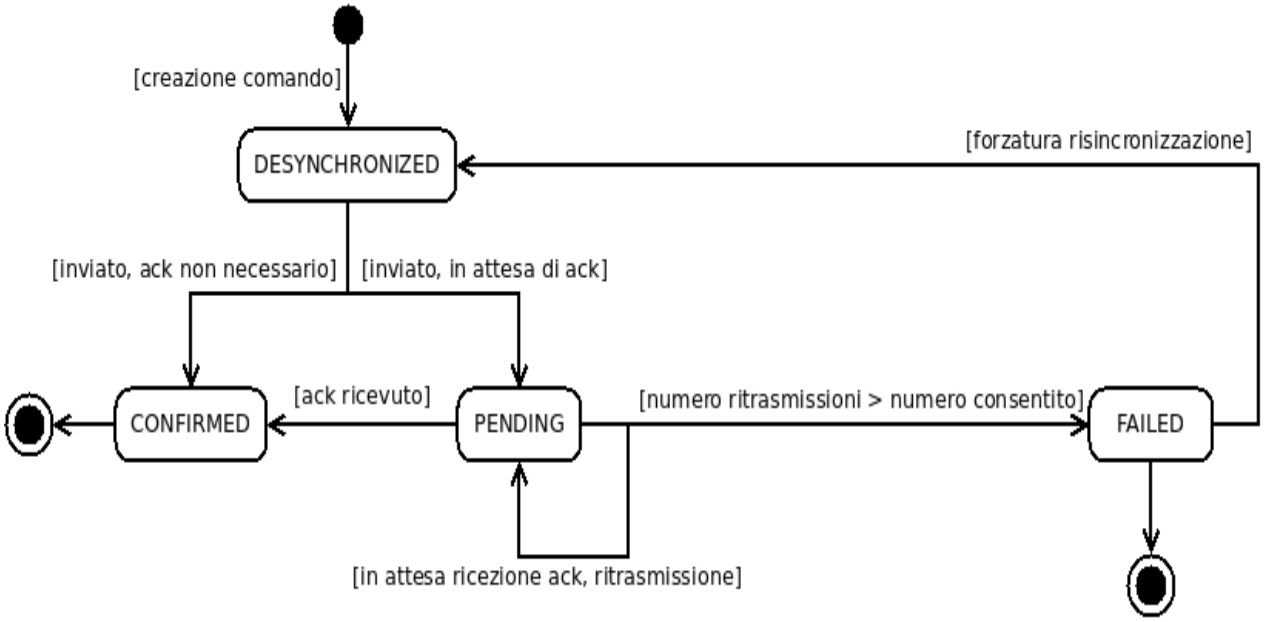

Figura 6: Diagramma degli stati di un comando creato

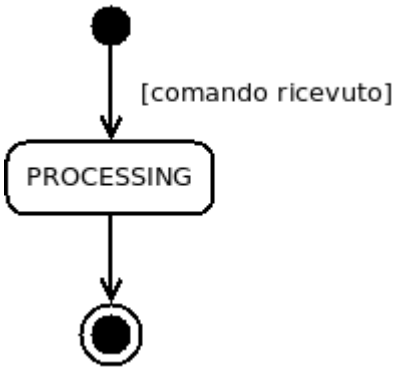

Figura 7: Diagramma degli stati di un comando ricevuto

Come possiamo quindi vedere dai diagrammi degli stati, alcuni comandi prevedono un comando di risposta (*ack*) e verranno ritrasmessi finché il programma non riceve il suddetto *ack* o non raggiunge il numero massimo di trasmissioni verso quel nodo.

Segue una lista di comandi previsti dal protocollo per la comunicazione tra

due nodi:

1. Inizio sincronizzazione

   Sintassi:

   Sync timestamp owner folderName mode nochat lifetime pwdRead pwdWrite

   Significato campi:

   owner = Proprietario della cartella di cui si vuole chiedere la sincronizzazione. Se si offre una cartella il proprietario sarà il nodo chiamante

   folderName = Nome della cartella che si vuole sincronizzare

   mode = Modo in cui si intende gestire la sincronizzazione. Può essere PUSH, PULL o PUSH&PULL come descritto precedentemente nei possibili modi di sincronizzazione

   lifetime = Campo che specifica il *time to live* in secondi da assegnare ai pacchetti diretti verso il nodo che ha effettuato la richiesta di sincronizzazione, da utilizzare in caso il nodo ricevente non conosca il mittente a priori

   nochat = Campo presente nella versione precedente che specificava se era possibile inviare file insieme ai comandi. Può essere 0 (falso) o 1 (vero). Nella versione attuale viene messo di default e non viene processato in quanto l'accorpamento di file e comandi è previsto dal protocollo

   pwdRead = Password di lettura, opzionale

   pwdWrite = Password di scrittura, opzionale

   Questo comando prevede una risposta.

2. Risposta a inizio di sincronizzazione

   Sintassi:

SyncAck timestamp owner foldetName mode nochat lifetime pwdRead pwdWrite response

Significato campi:

Questo comando è la risposta che viene inviata da un nodo quando gli viene richiesta una sincronizzazione. Tutti i campi sono una copia del comando *Sync* ad eccezione del campo *response*, che può avere valore “OK” in caso la sincronizzazione venga accettata o “ERROR” se la richiesta di sincronizzazione viene rifiutata. A seguito di “ERROR” va specificato il motivo del rifiuto.

3. Fine sincronizzazione

   Sintassi:

   Fin timestamp owner folderName fromOwner

   Significato campi:

   owner = Nome del proprietario della cartella su cui terminare la sincronizzazione

   folderName = Nome della cartella

   fromOwner = Campo che specifica se la richiesta di fine sincronizzazione viene dal proprietario della cartella o meno, può quindi assumere i valori 0 (falso) o 1 (vero). In caso questo campo sia impostato a 1 il ricevente deve eliminare la cartella specificata ed inoltrare la richiesta di sincronizzazione a tutti i nodi che avevano una sincronizzazione sulla cartella stessa.

   Questo comando prevede una risposta.

4. Risposta a fine sincronizzazione

   Sintassi:

   FinAck timestamp owner folderName fromOwner response

   Significato campi:

Ogni campo del comando è la copia dei campi del comando di fine sincronizzazione che l'ha generato, ad eccezione del campo *response* che può avere valore “OK” in caso la fine della sincronizzazione sia andata a buon fine oppure “ERROR” in caso di errore, seguito dal motivo che ha generato l'errore.

5. Aggiornamento file

   Sintassi:

   Update timestamp owner folderName fileName isDeleted lastModified … fileName isDeleted lastModified

   Significato campi:

   owner = Proprietario della cartella in cui andranno i file da aggiornare

   folderName = Nome della cartella oggetto della sincronizzazione

   A seguire, per ogni file da aggiornare, saranno presente i campi

   fileName = Nome del file da aggiornare

   isDeleted = Campo che può assumere i valori 0 (falso) o 1 (vero) e ci indica se un file è stato eliminato dal file system

   lastModified = Data di ultima modifica del file. Se la data specificata è più recente della data di modifica del file presente sul file system del nodo ricevente il file verrà sostituito, altrimenti verrà ignorato l'aggiornamento

   Questo comando prevede una risposta.

6. Risposta aggiornamento file

   Sintassi:

   UpdateAck timestamp owner folderName response

   Significato campi:

   Ogni campo è identico al comando di *Update* che l'ha generato, a

differenza di *response* in cui sarà indicato “OK” in caso l'aggiornamento sia andato a buon fine oppure “ERROR” in caso di problemi. A seguito di “ERROR” saranno presenti i motivi del mancato aggiornamento.

7. Forza aggiornamento file

   Sintassi:

   CheckUpdate timestamp owner folderName fileName isDeleted lastModified … fileName isDeleted lastModified

   Significato campi:

   Questo comando è identico al comando di *Update* e serve a forzare la sincronizzazione di una cartella anche in caso non ci siano state modifiche sui file presenti al suo interno. Può risultare utile in caso di guasti, problemi o più in generale in ogni caso in cui non si è sicuri che due nodi abbiano la stessa versione dei file e quindi non siano correttamente sincronizzati.

   Questo comando prevede una risposta.

8. Risposta forzatura aggiornamento file

   Sintassi:

   CheckUpdateAck timestamp owner folderName response

   Significato campi:

   Ogni campo è la copia dei campi del comando *CheckUpdate* che l'ha generato a differenza del campo *response* che può assumere valore “OK” in caso di corretta ricezione o “ERROR” in caso di errore, seguito dai motivi che hanno scatenato l'errore.

9. Sospensione di una sincronizzazione

   Sintassi:

   Freeze timestamp owner folderName voluntary

Significato campi:

owner = Proprietario cartella oggetto della sincronizzazione

folderName = Nome della cartella

voluntary = Campo che può assumere valore 0 (falso) o 1 (vero). Ci dice se la richiesta di congelamento della sincronizzazione è stata richiesta dall'utente o se invece nasce a fronte di errori. Se è a 1 questo comando prevede una risposta, non necessaria altrimenti in quanto si suppone che il nodo sia in uno stato di scorretto funzionamento e non può quindi processare un comando a lui diretto.

10. Risposta a sospensione di sincronizzazione

Sintassi:

FreezeAck timestamp owner folderName response

Significato campi:

Ogni campo è la copia del comando di *Freeze* che l'ha generato a differenza del campo *response* che può assumere i valori “OK” od “ERROR”, in modo simile ad ogni altro comando di *ack*.

11. Ripresa di una sincronizzazione

Sintassi:

Unfreeze timestamp owner folderName

Significato campi:

owner = Nome proprietario della cartella su cui riprendere la sincronizzazione

folderName = Nome cartella

Questo comando viene inviato quando si vuole sbloccare una sincronizzazione precedentemente bloccata da un comando *Freeze.* Prevede una risposta.

12. Risposta a richiesta ripresa di sincronizzazione

Sintassi:

UnfreezeAck timestamp owner folderName response

Significato campi:

Ogni campo è la copia del comando di *Unfreeze* che l'ha generato a differenza del campo *response* che può assumere i valori “OK” od “ERROR”, in modo simile ad ogni altro comando di *ack*.

13. Blocco di un nodo

Sintassi:

Block timestamp

Significato campi:

Questo comando viene inviato in automatico dal programma per notificare a un nodo che è stato inserito nella Blacklist del nodo con cui cercava di comunicare. Non prevede risposta in quanto ogni ulteriore comunicazione verso quel nodo verrà bloccata in automatico dal programma.

14. Sblocco di un nodo

Sintassi:

Unblock timestamp

Significato campi:

Questo comando viene generato in maniera automatica dal programma quando un nodo viene rimosso dalla Blacklist per notificargli che può tornare a comunicare con il nodo che lo aveva bloccato. Prevede un comando di risposta.

15. Risposta allo sblocco di un nodo

Sintassi:

UnblockAck timestamp response

Significato campi:

Il campo *response* può assumere i valori “OK” in caso di corretta ricezione o “ERROR” in caso di errore, seguito dai motivi dell'errore stesso.

3. Organizzazione del File System

Ad ogni avvio il programma controlla che siano presenti le cartelle e i file necessari al proprio funzionamento e, in caso negativo, provvede a crearli.

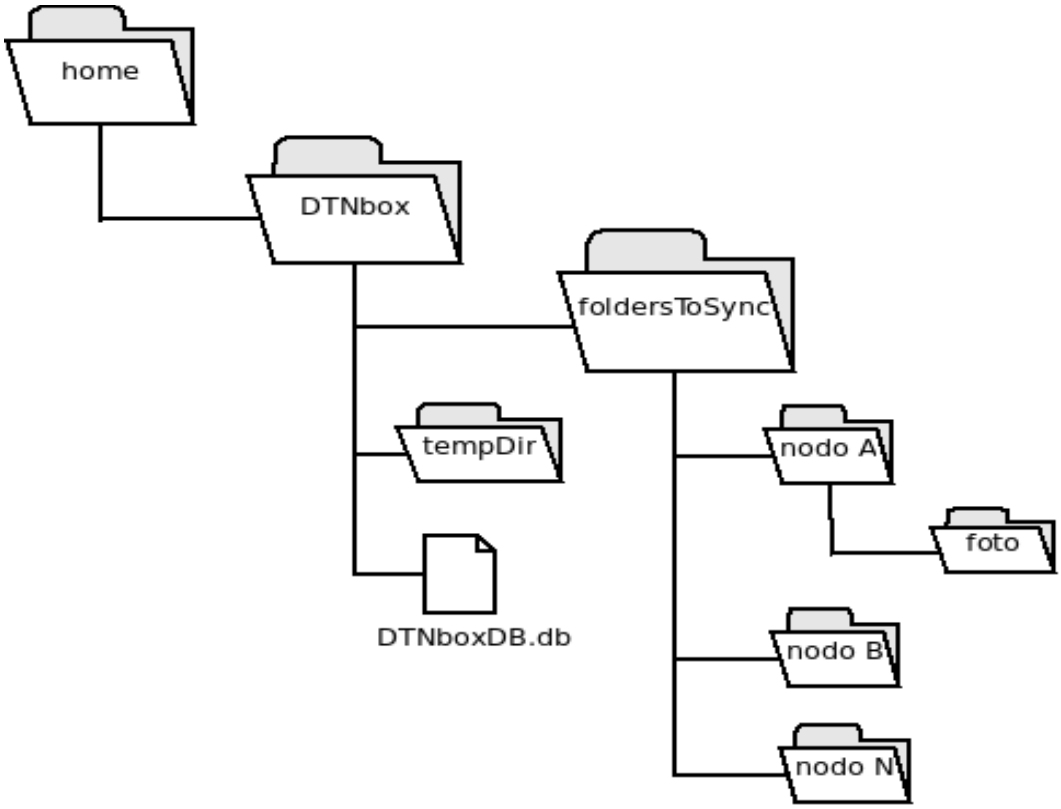

Figura 8: Rappresentazione cartelle e file del programma DTNbox sul file system

La cartella principale è denominata “DTNbox” e viene creata all'interno della cartella “home” dell'utente che ha avviato il programma. Eseguendolo come utente *root* avremo ad esempio il path “/root/DTNbox” o, più in generale, il path sarà “~/DTNbox”.

Dentro alla cartella “DTNbox” è presente il file “DTNboxDB.db” che ne rappresenta il database, la cartella nascosta “tempDir” in cui verranno scompattati e creati i tar ricevuti e destinati agli altri nodi e la cartella

“foldersToSync”, che conterrà tutte le cartelle che si intende condividere con altri nodi e che verranno monitorate in automatico per la rilevazione delle modifiche sui file contenuti al loro interno.

In futuro è previsto che, sempre dentro alla cartella “DTNbox”, venga creato un file di log ed un file di configurazione per facilitare l'uso del programma.

I nomi delle cartelle all'interno di “foldersToSync” per convenzione saranno uguali ai nomi dei proprietari delle cartelle e verranno rispettivamente ricavati a partire dal loro EID. Sarà quindi sempre presente almeno una cartella avente il nome del nodo che sta eseguendo il programma.

Ogni cartella che si vorrà sincronizzare andrà quindi creata all'interno della cartella “foldersToSync” e posizionata nella sottocartella relativa al nodo proprietario. Se ad esempio il nodo A vuole creare e sincronizzare una cartella chiamata “foto”, essa avrà path “~/DTNbox/foldersToSync/nodo A/foto” e, una volta iniziata correttamente una sincronizzazione verso un altro nodo, ogni file (comprese sottocartelle) contenuto al suo interno sarà monitorato e inviato ai vari nodi. Se il nodo ricevente, ad es. il nodo B, accetta una richiesta di sincronizzazione dal nodo A per la cartella “foto” il programma creerà in automatico, all'interno della cartella “foldersToSync” sul nodo B, la cartella “nodo A” e la sottocartella “foto” in cui andrà ad inserire i file da sincronizzare.

Capitolo 3: Architettura e implementazione

1. Architettura dell'applicazione

L'applicazione è stata sviluppata utilizzando il linguaggio di programmazione C, per poter utilizzare al meglio le API messe a disposizione dalle varie implementazioni del *bundle protocol* ed in particolare per poter usufruire della libreria *Abstraction Layer* [al_bp_API].

Si è cercato di rendere l'architettura il quanto più possibile modulare in modo da favorire il mantenimento futuro del programma in caso fossero necessarie modifiche, disaccoppiando la logica applicativa dalla logica di presentazione attraverso l'utilizzo del *pattern* architetturale Model-View-Controller (MVC).

Tutto il codice relativo alla modellazione dati è stato quindi posizionato nella cartella “Model” e verrà descritto nella sezione “Strutture dati”, mentre il codice responsabile della logica di *business* è stato inserito nella cartella “Controller” e verrà descritto nella sezione “Processi di controllo”.

In futuro tutti i moduli di gestione dell'interfaccia grafica e della logica di presentazione andranno posizionati nella cartella “View”.

La comunicazione tra i vari processi (*thread*) del programma è limitata, in quanto viene utilizzato un database per il mantenimento dello stato dell'applicazione, al quale ci si interfaccia attraverso il modulo “DBInterface”.

L'unica struttura dati condivisa (“synchronizationCmdList”) viene utilizzata per accorpare i comandi verso ogni nodo con cui si vuole comunicare, per i motivi descritti nella sezione “Protocollo di comunicazione”. Ogni *thread* che genera un comando andrà quindi ad inserirlo nella suddetta coda, che verrà periodicamente svuotata dal *thread* di controllo “scanAndSendThread”.

2. *Processi di controllo*

i. *mainDTNbox*

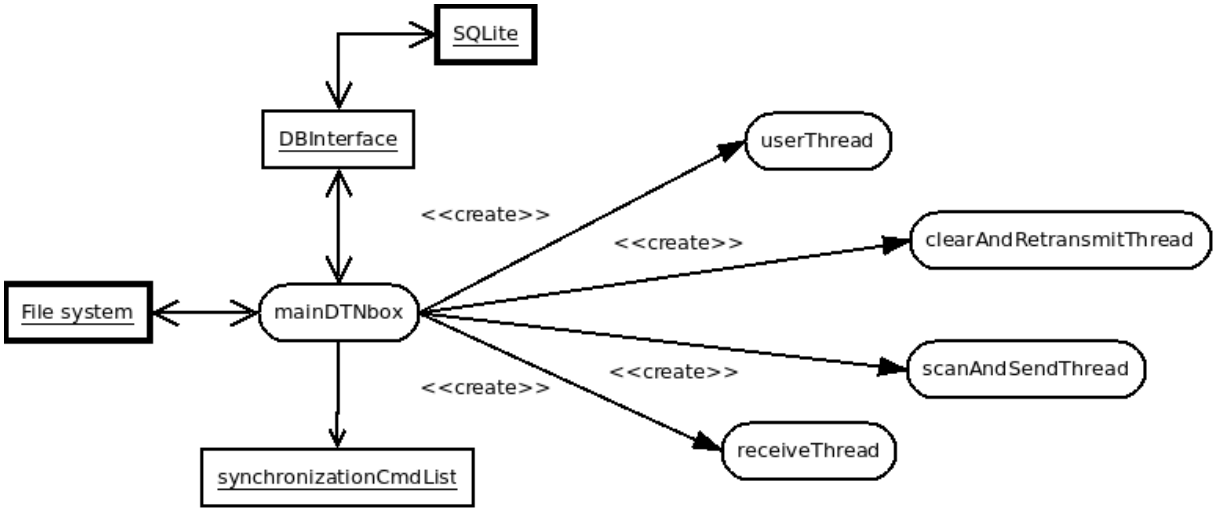

Figura 9: Schema della creazione dei vari thread di controllo

Il *thread* “mainDTNbox”, come suggerisce il nome stesso, è il primo *thread* che viene chiamato all'avvio del programma ed ha il compito di creare e passare gli argomenti a tutti i processi che compongono DTNbox, oltre ad occuparsi della corretta inizializzazione e/o controllo delle risorse utilizzate. Implementa il metodo:

- *int main();*

All'avvio del programma vengono come prima cosa recuperate dal demone DTN le informazioni relative al nodo locale, in particolare viene letto l'EID del nodo su cui si sta eseguendo il programma e in seguito vengono chiamati i metodi del modulo “utils” per leggere i valori di default riguardo al *lifetime* da assegnare ai *bundle* destinati a questo nodo e al numero massimo di tentativi d'invio dei comandi in caso di mancata ricezione dell'ack.

Recuperate queste informazioni vengono create, in caso non siano già presenti, tutte le cartelle necessarie al funzionamento (Figura 8).

Viene poi controllata l'esistenza del file “DTNboxDB.db”, che rappresenta il database, e, se non presente, viene creato e correttamente inizializzato generando ex novo le tabelle previste. In seguito si esegue una scansione del file system che controlla tutte le cartelle all'interno della cartella di proprietà dell'utente, ad esempio per il nodo A tale cartella sarà: “~/DTNbox/foldersToSync/nodo A/” e, in caso siano presenti delle cartelle non registrate sul database, esse vengono aggiunte. Questa operazione consente all'utente di creare delle nuove cartelle al di fuori di DTNbox e ad avere la garanzia che il file system ed il database siano allineati all'avvio del programma.

Se sono presenti dei comandi in stato DESYNCHRONIZED (ad esempio a causa di una terminazione involontaria) questi, per ogni nodo destinatario, vengono aggiunti alla coda di comandi in uscita “synchronizationCmdList” che viene quindi inizializzata ed eventualmente popolata.

Terminate le operazioni di inizializzazione il main procederà quindi a creare tutti i *thread* di controllo e gestione di DTNbox, aspettando la terminazione per ognuno di essi (Figura 9).

ii. al_bp_wrapper

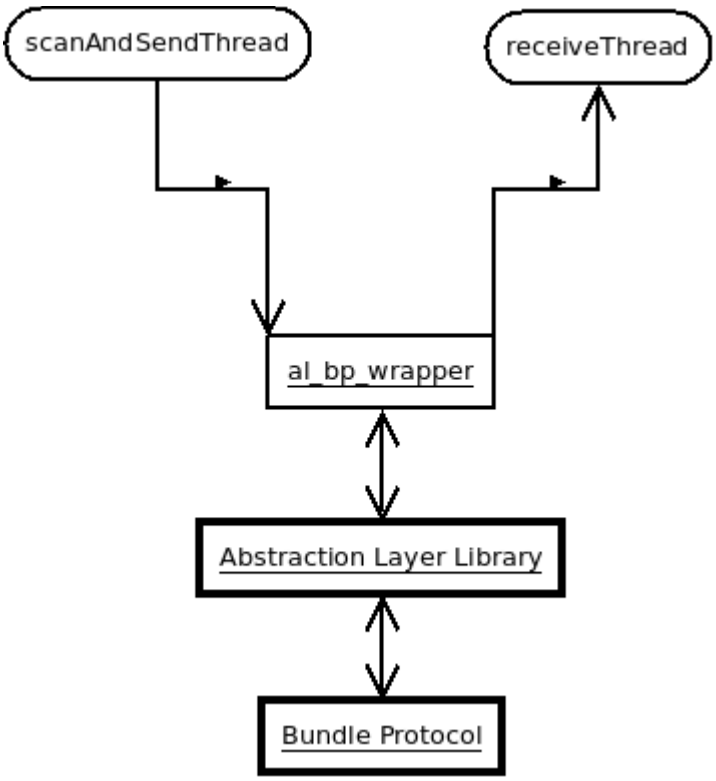

Figura 10: Schema utilizzo modulo al_bp_wrapper

Implementato nei file “al_bp_wrapper.c” e “al_bp_wrapper.h”, questo modulo ha lo scopo di definire tutti i metodi che si interfacciano all'*Abstraction Layer*.

L'*Abstraction Layer* rappresenta una libreria, sviluppata in C, che ci consente di astrarre dall'implementazione del *bundle protocol* offrendo delle API uniformi, evitando così di dover scrivere diverse versioni del programma a seconda dell'implementazione che si intende utilizzare.

Al momento sono supportati DTN2 (al_bp_vDTN2), ION (al_bp_vION) e IBR-DTN (al_bp_vIBR). E' inoltre possibile utilizzare la libreria generica (al_bp) che decide a *runtime* quale implementazione del *bundle protocol* utilizzare.

Questa modulo viene utilizzato unicamente dai *thread* “scanAndSendThread” per l'invio dei *bundle* e “receiveThread” per la ricezione degli stessi (Figura 10).

Se in futuro la sintassi e le API offerte dall'*Abstraction Layer* verranno modificate sarà sufficiente riscrivere solo questo modulo del programma.

Metodi implementati:

- *int dtnbox_openConnection(dtnBoxConnection** conn);*

Metodo che prende in input una struttura “dtnBoxConnection”, che incapsula le variabili contenenti le informazioni fondamentali per una connessione con il *bundle protocol*, ne alloca la memoria e apre una connessione al demone DTN.

- *int dtnbox_send(dtnBoxConnection* conn, char* tarName, dtnNode dest);*

Metodo per l'invio di un file (nel nostro caso sempre di tipo tar) verso un nodo DTN. Richiede una connessione correttamente inizializzata, il path assoluto del file tar che si vuole inviare e le informazioni riguardanti il nodo destinatario, in particolare il suo EID ed il *lifetime* in secondi da impostare nel *bundle*.

- *int dtnbox_receive(dtnBoxConnection* conn, char* tarName, char* sourceEID);*

Metodo dalla semantica bloccante, rimane in attesa dal demone DTN di un *bundle* e, in caso di corretta ricezione, scrive rispettivamente all'interno delle variabili “tarName” e “sourceEID” il path assoluto del file tar ricevuto e l'EID del nodo mittente.

- *int dtnbox_closeConnection(dtnBoxConnection* conn);*

Metodo che ha lo scopo di chiudere una connessione DTN e de-allocare la memoria della struttura *wrapper* “dtnBoxConnection”.

iii. clearAndRetransmitThread

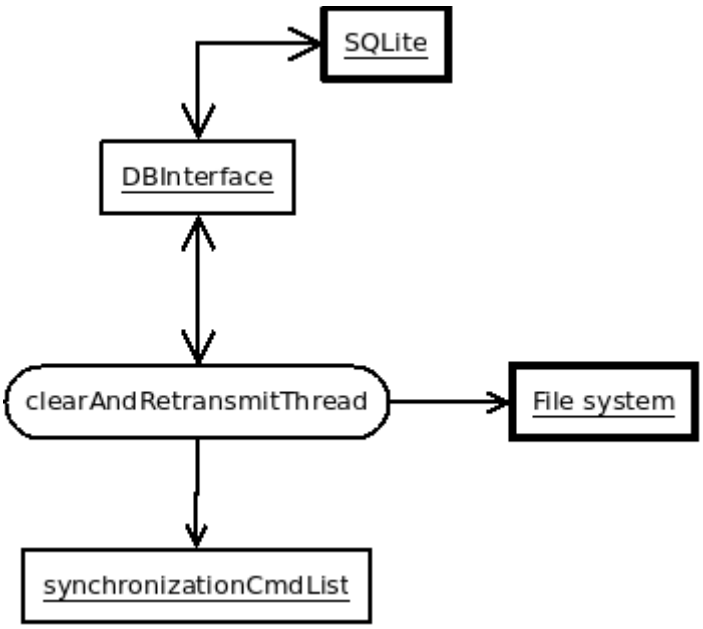

Figura 11: Schema clearAndRetransmitThread

Implementato nei file “clearAndRetransmitThread.c” e “clearAndRetransmitThread.h”, rappresenta uno dei tre *thread* sempre presenti dell'applicazione.

- *void* clearAndRetransmitDaemon(void* args);*

Processo sempre attivo che ciclicamente chiede al database quali comandi devono essere ritrasmessi, ad esempio tutti i comandi per cui non è arrivato l'ack nei tempi previsti oppure perché sono in stato DESYNCHRONIZED, e li aggiunge alla coda condivisa di comandi “synchronizationCmdList” per la ritrasmissione.

Terminato ciò procede ad eliminare tutti i comandi più vecchi di un certo valore di soglia (attualmente 5 minuti) che sono in stato CONFIRMED (ho ricevuto l'ack) o PROCESSING (comando ricevuto dal destinatario e già elaborato). Questa operazione risulta necessaria per limitare la crescita delle dimensioni del database. La cancellazione non immediata può servire a scopo di *debug*.

Infine il processo controlla che tutte le cartelle presenti all'interno del database abbiano una corrispondenza sul file system. In caso una cartella

non sia presente viene creata, mantenendo così la corrispondenza tra file system e database necessaria al corretto funzionamento del programma.

iv. commandParser

Utilizzato per ottenere una lista di comandi a partire da un file in formato ".dbcm". Implementa il seguente metodo:

- *cmdList cmdParser_getCommands(char* commandFile);*

Apre in lettura il file ".dbcm" al path assoluto specificato nella variabile "commandFile", legge tutti i comandi presenti al suo interno e chiama su di essi il metodo "newCommand" (vedere in seguito sezione "Strutture dati"). Restituisce una lista contenente tutti i comandi letti e correttamente istanziati.

v. parseBundleThread

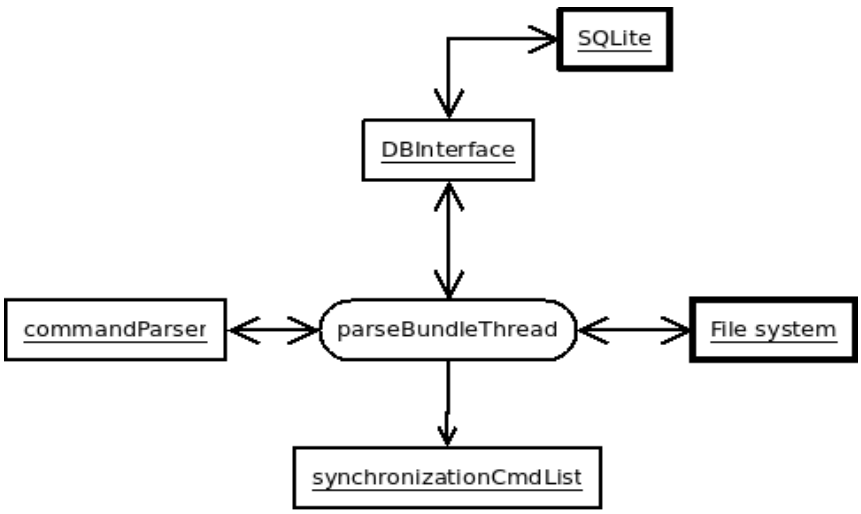

Figura 12: Schema parseBundleThread

Thread che viene creato ad ogni ricezione di un *bundle*. Il suo compito è quello di elaborare i messaggi ricevuti ed eseguire i comandi che vengono richiesti dal mittente.

Metodi implementati:

- *void* parseBundle(void* args);*

Ricevuto un file tar esso viene spostato all'interno della cartella temporanea “tempDir” dove viene scompattato. In seguito viene chiesto al modulo “cmdParser” di leggere tutti i comandi all'interno del file “.dbmc”. Per ogni comando presente verrà quindi chiamata la rispettiva funzione “receiveCommand”, dopo averlo aggiunto sul database in stato PROCESSING. In caso venga generata una risposta (ack) essa verrà inserita nella coda di comandi (“synchronizationCmdList”) da inviare verso il nodo mittente. Una volta eseguiti tutti i comandi richiesti vengono de-allocate le risorse utilizzate ed il *thread* termina la sua esecuzione. In caso di errore i file ricevuti vengono eliminati dal file system e ne viene notificato il motivo all'utente.

vi. receiveThread

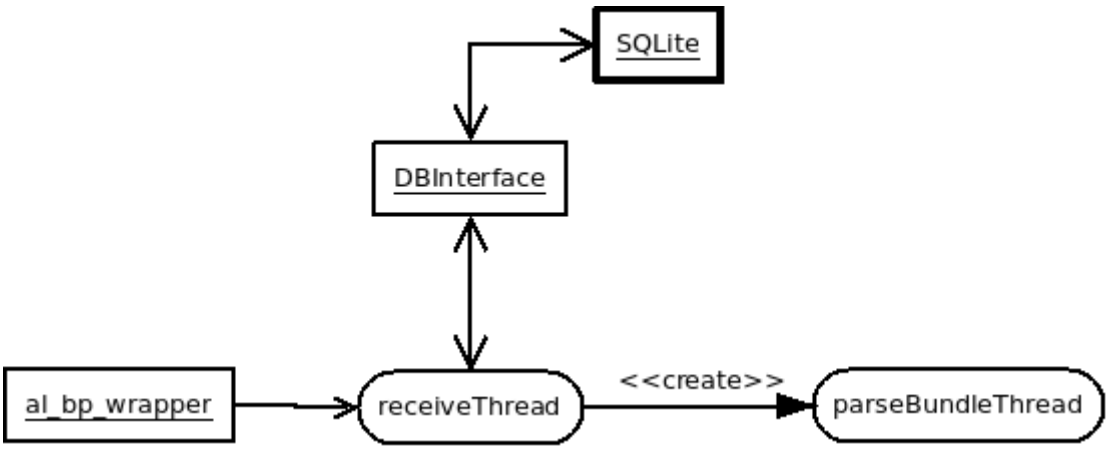

Figura 13: Schema receiveThread

Thread che ha il compito di rimanere in attesa di messaggi provenienti dal *bundle protocol.*

Metodi implementati:

- *void* receiveDaemon(void* args);*

Processo sempre attivo che, dopo aver aperto una connessione DTN, chiama la funzione bloccante “dtnbox_receive()”.

Alla ricezione di un *bundle* viene per prima cosa controllato se il *demux token* coincide con quello di DTNbox e, in caso affermativo, viene controllato se il nodo mittente è stato inserito all'interno della Blacklist o

della Whitelist.

In caso non sia in nessuna delle due liste viene chiesto all'utente se accettare o meno i *bundle* provenienti da quel nodo. Al momento, vista la mancanza di un interfaccia grafica per poter chiedere in modo chiaro all'utente come comportarsi, tutti i pacchetti in ingresso vengono elaborati ad esclusione del caso in cui l'utente abbia precedentemente inserito il nodo in Blacklist.

Una volta deciso che il *bundle* va elaborato viene chiamata la funzione “spawnBundleParserThread”.

- *int spawnBundleParserThread(dtnNode sender, dtnNode myNode, synchronizationCmdListArray* outArray, char* bundleLocation, char* tempDir, char* dbName);*

Questa funzione ha il compito di creare un nuovo “parseBundleThread” al quale passerà tutti i parametri necessari al suo corretto funzionamento quali il path in cui trovare il file tar ricevuto e informazioni accessorie sul nodo mittente. Non viene aspettata la terminazione del “parseBundleThread” permettendo così al processo chiamante di rimettersi immediatamente in attesa di ricezione dei *bundle*.

vii. scanAndSendThread

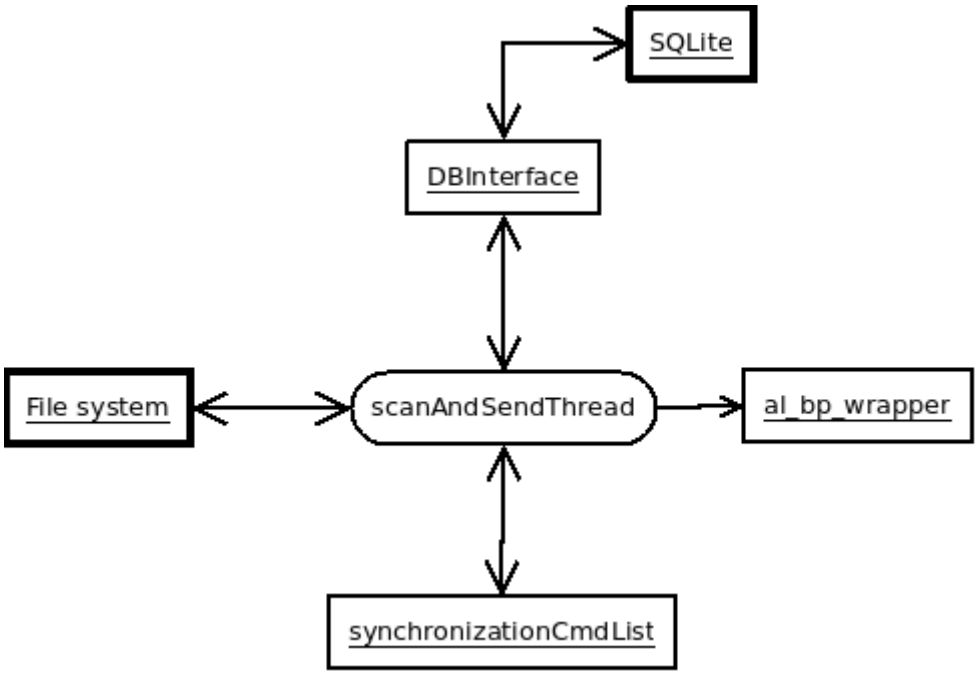

Figura 14: Schema scanAndSendThread

Processo sempre attivo che ha lo scopo di controllare se ci sono state modifiche sul file system (ad esempio modifica, creazione o eliminazione di file) e svuotare periodicamente la struttura “synchronizationCmdList” generando, per ogni nodo a cui si vogliono inviare i comandi, il rispettivo file tar, come definito dal protocollo di comunicazione.

Metodi implementati:

- *int updateFilesFromFS(sqlite3* dbConn, folderToSync folder, char* folderPath);*

Ricevuto in ingresso il path (“folderPath”) assoluto di una cartella da monitorare e le informazioni attualmente presenti sul database, contenute nella struttura “folder”, per ogni file della cartella vengono controllate le differenze tra le informazioni presenti sul database e le informazioni presenti sul file system.

Se un file è presente sul file system ma non sul database significa che è un nuovo file, viene quindi aggiunto sul database e viene propagata l'informazione a ogni sincronizzazione presente sulla cartella in modo che

ogni nodo sincronizzato ne riceva una copia.

Se invece il file è già presente sul database ma ha una data di modifica posteriore a quella memorizzata lo stato viene cambiato a FILE_DESYNCHRONIZED e viene propagata la modifica a tutte le sincronizzazioni precedentemente presenti su quel file aggiornando sul database la data di ultima modifica.

Infine, se il file è presente sul database ma non è presente sul file system significa che è stato eliminato dall'utente, il metodo provvede quindi ad aggiornare lo stato del file a FILE_DESYNCHRONIZED e ad impostare il flag di FILE_DELETED, notificando così a tutti i nodi che lo possiedono di procedere alla sua rimozione dal file system. Nella sezioni successive verranno descritti più dettagliatamente gli stati possibili per un file su cui è stata creata una sincronizzazione.

- *void* scanAndSendDaemon(void* args);*

Processo sempre attivo. Periodicamente controlla tutte le cartelle che devono essere monitorate, ossia le cartelle che fanno parte di una sincronizzazione in PUSH o PUSH&PULL verso un altro nodo, e chiede le differenze rispetto all'istantanea presente sul database di quella cartella per quella sincronizzazione alla funzione “updateFilesFromFS”. In seguito, se sono state trovate delle modifiche, vengono creati i relativi comandi di “Update” ed inseriti nella coda di comandi “synchronizationCmdList”.

Terminata l'operazione di controllo del file system viene svuotata la “synchronizationCmdList”, creando per ogni nodo il relativo file tar contenente il file “.dbcm” con i comandi da eseguire e tutti i file da aggiornare. Terminata la creazione del file tar viene invocata la funzione “dtnbox_send()” che provvederà a creare ed inviare il relativo *bundle*. Infine vengono de-allocate tutte le risorse precedentemente utilizzate.

viii. userThread

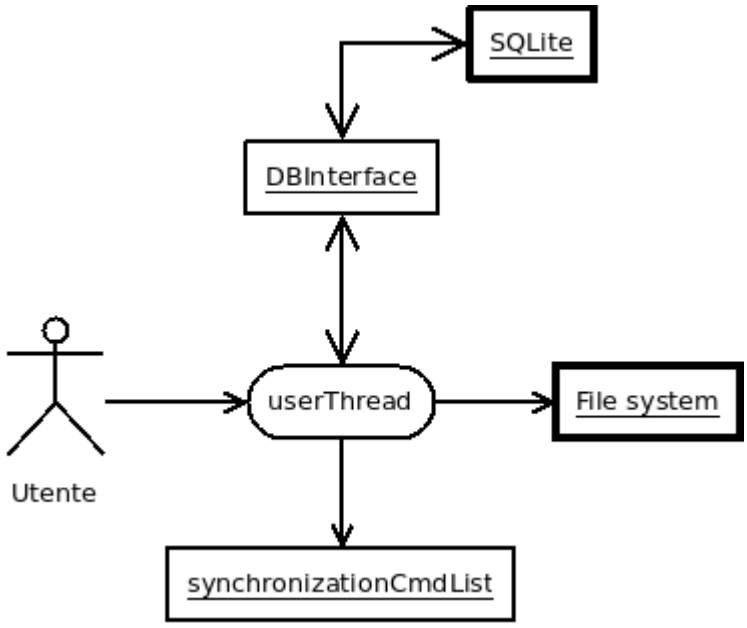

Figura 14: Schema userThread

Lo scopo di questo *thread* è di offrire un'interfaccia all'utente per poter utilizzare il programma. Al momento, sia per poter procedere più velocemente verso una fase di test, sia per permetterne l'impiego su dispositivi facenti parte di una rete *Internet of Things* (IoT), è stato implementato come *Command Line Interface* (CLI). Il processo resta in continua attesa di comandi fino a quando non viene inserito dall'utente il carattere di *End Of File* (EOF) terminando effettivamente l'esecuzione di DTNbox.

In futuro questo *thread*, per rendere più semplice e gradevole l'utilizzo del programma, andrà integrato da un interfaccia grafica opzionale. Implementa il metodo:

- void* userThread(void* args);

Questo metodo consente all'utente di inserire tramite CLI diversi comandi che gli consentono di interagire con DTNbox. I comandi attualmente implementati sono:

- “help”: Stampa a video i comandi attualmente presenti.

- “add node”: Vengono chieste in maniera interattiva all'utente tutte le informazioni relative a un nodo con cui si intende comunicare, quali il suo EID, il *lifetime* in secondi da assegnare ai *bundle* ed il numero di invii da eseguire in caso di mancata ricezione di ack. Terminata la fase di input viene aggiunto il nodo sul database e viene inserito all'interno della Whitelist.
- “add folder”: Viene chiesto all'utente il nome (relativo) della cartella che si vorrà creare sul file system, all'interno della sottocartella di proprietà dell'utente che sta eseguendo il programma (ad esempio per il nodo A sarà “~/DTNbox/foldersToSync/nodo A”). La funzione provvederà quindi a crearla sia sul file system che ad inserirla nel database. In caso la cartella fosse già presente nel file system ci si limiterà al suo inserimento all'interno del database.
- “add sync”: Comando di inizio di sincronizzazione. Vengono chieste all'utente tutte le informazioni necessarie per l'inizio di una sincronizzazione quali l'EID del nodo destinatario, il modo della sincronizzazione, la cartella che si vuole condividere ed eventuali password di lettura e scrittura. Terminata la fase di input viene creato il comando di “Sync” e tutte le informazioni necessarie vengono salvate all'interno del database. Il comando di “Sync” appena creato verrà poi inserito all'interno della “synchronizationCmdList” per il futuro invio.
- “reset database”: Comando che distrugge e ricrea tutte le tabelle presenti sul database. Utilizzato prevalentemente in fase di *debug*.
- “dump database”: Comando che crea il file “DTNboxDB.txt”, il quale rappresenta un istantanea del database in quel determinato momento, stampando tutti i dati presenti al suo interno in un file di testo. Utile per controllare lo stato dei campi all'interno del database, può essere importato come foglio di calcolo, ricordandosi di specificare come delimitatore dati il carattere “|”.

- “force update”: Comando che consente all'utente di forzare la ritrasmissione di tutti i file presenti sul file system all'interno delle cartelle monitorate, indipendentemente dallo stato della loro sincronizzazione, generando i relativi comandi di “Update” per ogni sincronizzazione presente su di essi.
- “delete sync”: Comando di fine sincronizzazione. All'utente viene chiesto su quale cartella e per quale nodo terminare la sincronizzazione, ed eventualmente se eliminare definitivamente la cartella ed i file contenuti al suo interno. In caso l'utente decida di cancellare la cartella il comando di fine sincronizzazione verrà propagato a tutti i nodi con cui attualmente si era in sincronizzazione sulla cartella appena eliminata. Verranno quindi creati uno o più comandi di “Fin” che saranno inseriti nella coda “synchronizationCmdList” per il futuro invio.

3. Struttura e gestione del database

Il database è un componente necessario e fondamentale al funzionamento di DTNbox per il mantenimento persistente del suo stato.

Si è deciso di utilizzare la libreria software open source offerta da SQLite [SQLITE3] che implementa un *Database Management System* (DBMS) estremamente veloce e di ridotte dimensioni, largamente utilizzato nei sistemi *embedded* e nell'ambiente Android. SQLite non presenta un architettura di tipo client-server ma il database è interamente contenuto all'interno di un file (nel nostro caso il file “DTNboxDB.db”), rendendo così superfluo l'utilizzo di un demone sempre attivo che svolga la funzione di DBMS. E' inoltre molto robusto garantendo che le transazioni siano atomiche, consistenti, isolate e durabili (ACID) anche in caso di malfunzionamento.

La compilazione di DTNbox richiede quindi l'installazione sul sistema della libreria “libsqlite3-dev” avente versione di SQLite maggiore o uguale a “3.5.0”.

Per la gestione dell'accesso concorrente al database, dato che come abbiamo visto il programma ha almeno tre *thread* sempre attivi che necessitano contemporaneamente di una connessione ad esso, sono state valutate diverse possibili soluzioni.

SQLite offre una gestione di tipo *lock* delle risorse ma, in caso di connessione condivisa, è compito del programma che lo utilizza quello di farsi carico della corretta gestione della risorsa. Le possibili soluzioni considerate sono state:

1. Utilizzare un unica connessione condivisa per tutti i processi.
2. Creare un *pool* di connessioni e, ad ogni creazione di un nuovo *thread*, assegnare ad esso una connessione se disponibile.
3. Assegnare una connessione locale ad ogni *thread* che intende utilizzare il database.

Si è deciso di optare per la terza soluzione visto il numero ridotto e predicibile di *thread* che necessitano di accesso al database nel nostro programma, gestendo così il problema dell'accesso concorrente nella maniera più semplice ed efficace possibile fornendo ad ogni processo la propria connessione locale al database, eliminando l'*overhead* che si genererebbe creando un *pool* di connessioni che rimarrebbero largamente inutilizzate.

Tutti i metodi che si interfacciano al database sono stati implementati all'interno del modulo “DBInterface.c” e “DBInterface.h”, in modo che se in futuro si decidesse di cambiare DBMS sarà sufficiente sostituire solo questo modulo. Visto il numero considerevole di funzioni presenti, di seguito verrà data una breve descrizione del comportamento dei metodi implementati e testati per ogni tabella, si rimanda alla lettura del codice e dei commenti in esso contenuti per una descrizione più dettagliata.

La maggior parte delle funzioni segue il seguente *pattern* di esecuzione:

1. Controllo che i parametri passati in input al metodo abbiano valori

significativi.

2. Compilazione della *query* da eseguire sul database e *bind* dei parametri forniti in input.
3. Esecuzione della *query*.
4. Eventuale lettura dei dati richiesti.
5. De-allocazione delle risorse utilizzate.

Nonostante la maggior parte delle funzioni siano chiamate in automatico dai processi del programma, alcune di esse sono direttamente accessibili dall'utente. Per evitare attacchi di tipo *SQL Injection* e per garantire maggior robustezza all'applicazione, tutti i metodi che richiedono parametri di input utilizzano l'API *prepared statement* offerta da SQLite, che permette di eseguire un controllo a monte sulla correttezza dei dati eseguendo quando necessario l'*escape* dei caratteri sensibili e garantendo l'ottimizzazione della *query* stessa.

Le uniche funzioni che non ne fanno utilizzo sono quelle che eseguono *query* non modificabili e senza parametri di input, quali ad esempio quelle di creazione o distruzione tabelle. Ogni metodo restituirà poi al chiamante un intero che segnalerà il successo dell'operazione richiesta o notificherà i motivi dell'errore che ne hanno impedito la corretta esecuzione.

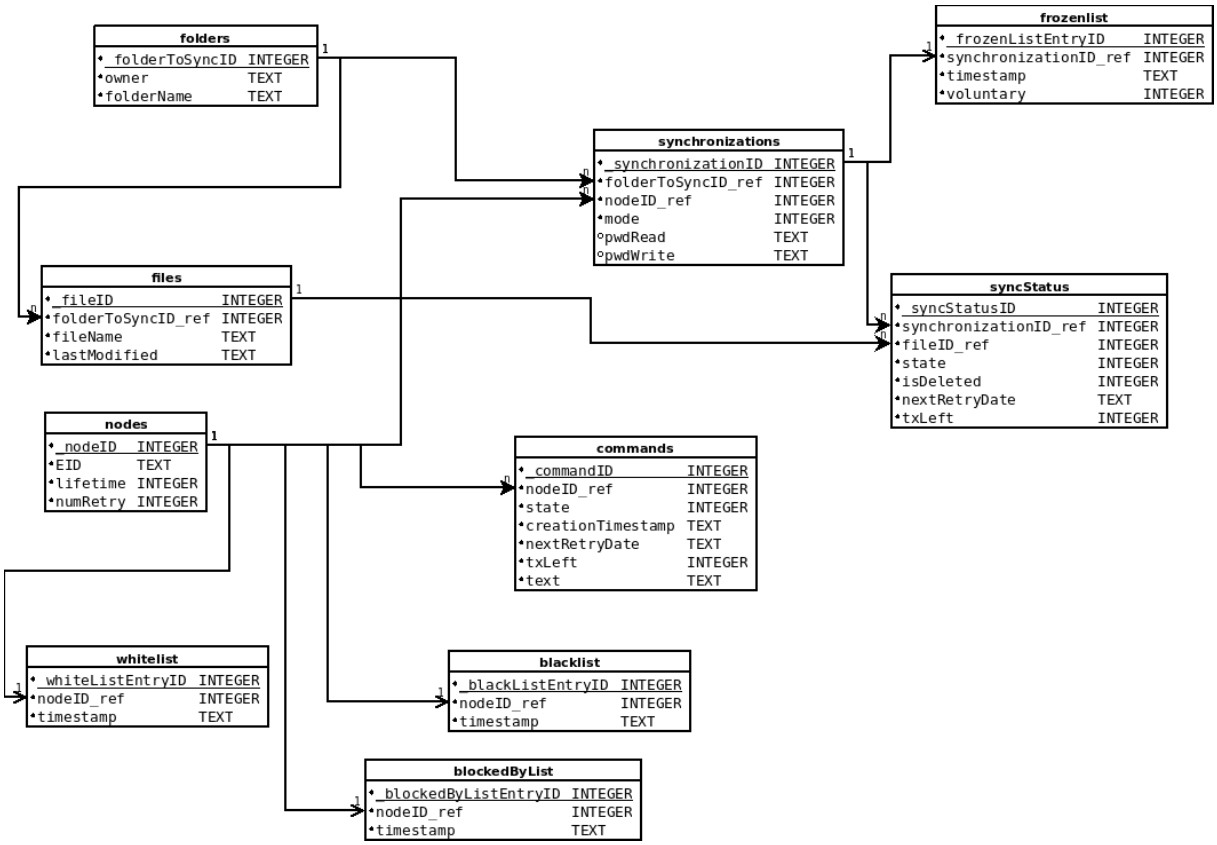

Figura 15: Modello entità-relazione rappresentante la struttura del database

Ogni tabella ha lo scopo di rappresentare e rendere persistente un entità (ed una struttura dati) utilizzata all'interno di DTNbox. Alcuni campi sono stati definiti come TEXT anche se rappresentano dati numerici in quanto, a causa delle limitazioni di SQLite, l'unico tipo di dato numerico supportato è INTEGER. All'interno di DTNbox, per aver una maggior precisione, le date sono state ricavate come *long*, per evitare quindi una conversione ed un conseguente troncamento di informazioni significative i dati vengono salvati come stringhe e convertiti in maniera opportuna alla loro lettura.

Tutti i campi che rappresentano l'identificativo univoco di un *record*, che per convenzione iniziano col carattere “_” e terminano con la stringa “ID”, vengono automaticamente auto-incrementati dal DBMS ad ogni nuovo inserimento e sono di tipo INTEGER.

Segue una lista dettagliata delle tabelle implementate e dei relativi metodi

di Create, Read, Update e Delete (CRUD):

i. folders

Nome campo	Tipo dato	Significato
_folderToSyncID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.
owner	TEXT	Stringa rappresentante il proprietario della cartella, viene ricavata a partire dall'EID del nodo. Al momento solo lo schema “dtn” è supportato.
folderName	TEXT	Nome della cartella.

Questa tabella ha lo scopo di rappresentare una cartella presente sul file system, in particolare verranno mappate tutte le cartelle presenti all'interno della cartella “~/DTNbox/foldersToSync”.

Vincoli:

Nessun campo può essere vuoto (NULL), e ogni coppia “owner”, “folderName” deve essere univoca.

Metodi di gestione:

- *int DBConnection_addFolderToSync(sqlite3* conn, folderToSync folder);*

Aggiunta sul database di una cartella presente sul file system.

- *int DBConnection_getFolderToSyncID(sqlite3* conn, folderToSync folder, int* folderToSyncID);*

Funzione che restituisce l'identificativo univoco della cartella richiesta in input.

- *int DBConnection_getSynchronizationsOnFolder(sqlite3* conn, folderToSync folder, synchronization** syncs, int* numSyncs);*

Passando in input una struttura “folderToSync” contenente tutte le informazioni relative ad una cartella, questo metodo restituisce un vettore

contenente tutte le sincronizzazioni attualmente presenti sulla cartella stessa.

- *int DBConnection_getAllFoldersToSync(sqlite3* conn, folderToSync** folders, int* numFolders);*

Funzione che restituisce tutte le cartelle da monitorare, ossia le cartelle sulle quali al momento esiste almeno una sincronizzazione attiva.

- *int DBConnection_isFolderOnDb(sqlite3* conn, folderToSync folderToCheck, int* result);*

Funzione che controlla se una cartella è presente o meno sul database, restituendo il risultato in logica positiva all'interno della variabile “result”.

- *int DBConnection_getAllFolders(sqlite3* conn, folderToSync** folders, int* numFolders);*

Metodo che restituisce tutte le cartelle attualmente mappate all'interno del database.

- *int DBConnection_deleteFolder(sqlite3* conn, folderToSync folder);*

Metodo che ha lo scopo di eliminare definitivamente un *record* rappresentante una cartella all'interno del database. Viene chiamato alla cancellazione di una cartella dal file system (va quindi invocato dopo l'eventuale eliminazione di sincronizzazioni presenti su di essa), e provvede prima ad eliminare tutti i record nella tabella *files* chiamando la funzione specifica che fanno riferimento alla cartella, poi viene eliminata la cartella stessa.

ii. files

Nome campo	Tipo dato	Significato
_fileID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.
folderToSyncID_ref	INTEGER	Intero rappresentate la chiave primaria della tabella “folders” che ci permette di risalire a quale cartella

		appartiene il file.
fileName	TEXT	Nome del file.
lastModified	TEXT	Data di ultima modifica del file, aggiornata all'ultimo controllo eseguito dal thread “scanAndSendThread”.

Tutti i file nelle sottocartelle da monitorare vengono memorizzati all'interno di questa tabella. Il campo “lastModified” verrà ciclicamente confrontato con la data di ultima modifica del file che il *record* rappresenta e, in caso sia più recente della data salvata sul database, il campo verrà aggiornato e verranno creati i comandi di “Update” necessari.

Vincoli:

Nessun campo può essere NULL e la coppia “folderToSyncID_ref”, “fileName” deve essere univoca per ogni record, in quanto non possono esistere sul file system due file con lo stesso nome all'interno della stessa cartella.

Metodi di gestione:

- *int DBConnection_addFileToSync(sqlite3* conn, fileToSync file, int folderToSyncID);*

Passata in input una struttura “filesToSync” ne aggiunge il contenuto sul database.

- *int DBConnection_isFileOnDb(sqlite3* conn, fileToSync fileToCheck, int folderToSyncID, int* result);*

Funzione che controlla se un file è attualmente presente sul database, il valore di ritorno sarà passato all'interno della variabile “result” in logica positiva (1 se il file è presente, 0 altrimenti).

- *int DBConnection_getFilesToSyncFromFolder(sqlite3* conn, fileToSyncList* files, int folderToSyncID);*

Funzione che, per la cartella passata in input, restituisce una lista

contenente tutti i file appartenenti a tale cartella.

- *int DBConnection_getFileToSyncID(sqlite3* conn, fileToSync file, int folderToSyncID, int* fileID);*

Metodo che passato in input una struttura “fileToSync” e l'identificativo della cartella a cui appartiene, ne restituisce l'identificativo univoco del record presente nel database.

- *int DBConnection_getFileToSyncLastModified(sqlite3* conn, fileToSync file, int folderToSyncID, unsigned long long* lastModified);*

Funzione che restituisce la data di ultima modifica di un file presente all'interno del database. Viene utilizzata per controllare se ci sono state modifiche sul file dall'ultimo controllo eseguito.

- *int DBConnection_updateFileToSyncLastModified(sqlite3* conn, fileToSync file, int folderToSyncID, unsigned long long lastModified);*

Se un file è stato modificato dall'ultimo controllo, attraverso questa funzione è possibile aggiornare la sua data di ultima modifica presente all'interno del database.

- *int DBConnection_deleteFilesForFolder(sqlite3* conn, int folderToSyncID);*

Funzione che consente l'eliminazione di tutti i file che fanno riferimento alla cartella con l'identificativo corrispondente al valore passato in input “ folderToSyncID”. Viene chiamata prima dell'eliminazione di una cartella per rispettare i vincoli imposti.

iii. nodes

Nome campo	Tipo dato	Significato
_nodeID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.
EID	TEXT	Stringa rappresentate l'EID del nodo, completo di schema e *demux token* (es. “dtn://test.dtn/DTNbox”).

lifetime	INTEGER	Lifetime in secondi da assegnare ai *bundle* diretti verso questo nodo.
numRetry	INTEGER	Numero di tentativi di invio de eseguire in caso di mancata ricezione dell'ack di un comando inviato a questo nodo.

Rappresenta un nodo DTN con cui si intende comunicare.

Vincoli:

Nessun campo può essere impostato a NULL e l'EID è univoco come da specifica DTN.

Metodi di gestione:

- *int DBConnection_addDtnNode(sqlite3* conn, dtnNode node);*

L'utente fornisce in input una struttura di tipo “dtnNode” contenente tutte le informazioni di un nodo con cui si intende comunicare ed essa viene memorizzata all'interno del database nella tabella “nodes”.

- *int DBConnection_getDtnNodeID(sqlite3* conn, dtnNode node, int* nodeID);*

A partire da una struttura di tipo “dtnNode” viene ricavato il suo identificativo univoco sul database (campo “_nodeID” tabella “nodes”).

- *int DBConnection_getDtnNodeFromEID(sqlite3* conn, dtnNode* node, char* EIDToGet);*

A partire da una stringa contenente l'EID del nodo viene restituita la struttura dati “dtnNode” con tutte le informazioni sul nodo attualmente presenti sul database.

- *int DBConnection_getDtnNodeFromID(sqlite3* conn, dtnNode* node, int nodeIDToGet);*

Passando in input un intero contenente l'identificativo univoco di un nodo (campo “_nodeID”) viene restituita la struttura “dtnNode” contenente le

informazioni relative ad esso.

- *int DBConnection_getAllDestNodesWithSync(sqlite3* conn, dtnNode** dtnNodes, int* numDtnNodes);*

Funzione che restituisce un vettore contenente tutti i nodi con cui si è stabilita una sincronizzazione.

- *int DBConnection_updateDTNnodeLifetime(sqlite3* conn, dtnNode node, int lifetime);*

Funzione che, dato in input un intero contenente il *lifetime* in secondi per i *bundle* destinati a quel nodo ne aggiorna il valore attualmente presente sul database.

iv. synchronizations

Nome campo	Tipo dato	Significato
_synchronizationID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.
folderToSyncID_ref	INTEGER	Intero che identifica univocamente la cartella oggetto della sincronizzazione, in riferimento alla tabella “folders”.
nodeID_ref	INTEGER	Intero che identifica univocamente il nodo con cui si desidera entrare in sincronizzazione, in riferimento alla tabella “nodes”.
mode	INTEGER	Uno dei tre possibili modi della sincronizzazione ossia PUSH&PULL (0), PULL (1) o PUSH (2). Viene salvato come intero in quanto all'interno del programma questo dato viene gestito come enumerativo.
pwdRead	TEXT	Password di lettura opzionale.
pwdWrite	TEXT	Password di scrittura

		opzionale.

Tabella su cui vengono memorizzate tutte le sincronizzazioni da e verso un nodo.

Vincoli:

Solo i campi pwdRead e pwdWrite possono essere impostati a NULL, ogni altro campo deve contenere valori significativi. La coppia “folderToSyncID_ref”, “nodeID_ref” deve essere univoca per ogni record della tabella in quanto una sincronizzazione si riferisce sempre in modo univoco ad una cartella e ad un nodo di destinazione.

Metodi di gestione:

- *int DBConnection_addSynchronization(sqlite3* conn, synchronization sync);*

Funzione che passata una struttura di tipo “synchronization” ne salva le informazioni all'interno del database.

- *int DBConnection_getSynchronizationID(sqlite3* conn, synchronization sync, int* synchronizationID);*

Funzione che, passato in input una struttura di tipo “synchronization” ne restituisce l'identificativo univoco sul database (campo “_synchronizationID”).

- *int DBConnection_deleteSynchronization(sqlite3* conn, synchronization sync);*

Funzione per l'eliminazione di una sincronizzazione dal database. Come prima cosa, dopo il controllo dei parametri, vengono invocati i metodi di eliminazione dei record sulla tabella “syncStatus” e delle eventuali *entry* all'interno della tabella “frozenlist” utilizzando gli appositi metodi e, in caso positivo, si procede all'eliminazione della sincronizzazione passata all'interno della struttura “sync”.

v. syncStatus

Nome campo	Tipo dato	Significato
_syncStatusID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.
synchronizationID_ref	INTEGER	Intero che rappresenta una sincronizzazione in riferimento alla tabella “synchronizations”.
fileID_ref	INTEGER	Intero che rappresenta un file parte della sincronizzazione in relazione alla tabella “files”.
state	INTEGER	Stato in cui si trova un file durante una sincronizzazione. E' un intero in quanto gli stati vengono gestiti come enumerativi (Figura 16) e sono definiti all'interno del modulo dati “fileToSync.h”.
isDeleted	INTEGER	Intero che ci descrive se un file è stato eliminato (FILE_DELETED, 1) oppure no (FILE_NOTDELETED, 0).
nextRetryDate	TEXT	Stringa rappresentante la data di prossima trasmissione dell'aggiornamento del file in caso non sia FILE_DELETED e non si sia ricevuto l'ack.
txLeft	INTEGER	Numero di ritrasmissioni ancora disponibili per il file.

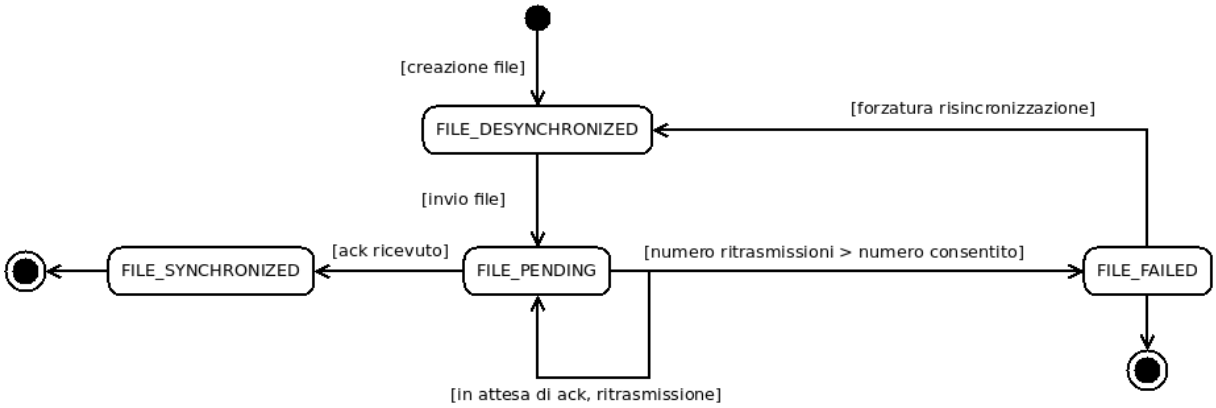

Figura 16: Possibili stati del file

Questa tabella rappresenta il legame tra i file e le sincronizzazioni. Il protocollo di comunicazione specifica che un file può essere parte di più sincronizzazioni, grazie a questa tabella è possibile tenere traccia del suo stato, in riferimento ad ogni sincronizzazione, in modo granulare.

Vincoli:

Nessun campo può essere impostato a NULL e ogni coppia “synchronizationID_ref”, “fileID_ref” deve essere univoca.

Metodi di gestione:

- *int DBConnection_addSyncOnFile(sqlite3* conn, synchronization sync, fileToSync file, int folderToSyncID);*

Funzione per l'aggiunta di una sincronizzazione su di un file. Viene chiamata a seguito della creazione con successo di una sincronizzazione sulla cartella che contiene il file specificato in input.

- *int DBConnection_updateSyncOnFileState(sqlite3* conn, synchronization sync, fileToSync file, int folderToSyncID, FileState newState);*

Funzione che aggiorna lo stato del file (Figura 16) sul database a seconda del suo stato di elaborazione e sincronizzazione.

- *int DBConnection_updateSyncOnFileTxLeftAndRetryDate(sqlite3* conn, synchronization sync, fileToSync file, int folderToSyncID, int txLeft,*

unsigned long long nextRetryDate);

Funzione che, dato in input un file e la sincronizzazione a cui fa riferimento, ne aggiorna il numero di trasmissioni disponibili rimaste e la prossima data di ritrasmissione del file stesso.

- *int DBConnection_updateSyncOnFileDeleted(sqlite3* conn, synchronization sync, fileToSync file, int folderToSyncID, FileDeletionState newState);*

Metodo che permette di cambiare lo stato di un file a DELETED sulla sincronizzazione.

- *int DBConnection_deleteSyncOnFile(sqlite3* conn, synchronization sync, fileToSync file, int folderToSyncID);*

Funzione che, specificato in input un file ed una sincronizzazione sulla cartella che lo contiene, elimina la sincronizzazione attualmente presente sul file.

- *int DBConnection_deleteSyncOnAllFiles(sqlite3* conn, synchronization sync);*

Funzione per l'eliminazione dei *record* relativi a tutti i file facenti parte della sincronizzazione passata all'interno della struttura “sync”.

- *int DBConnection_getSyncOnFileTxLeft(sqlite3* conn, synchronization sync, fileToSync file, int folderToSyncID, int* txLeft);*

Funzione che restituisce il numero di ritrasmissioni disponibili attualmente su quel file.

- *int DBConnection_getAllFilesToUpdate(sqlite3* conn, synchronization sync, folderToSync folder, fileToSync** files, int* numFiles);*

Metodo che, passata in input una sincronizzazione, restituisce un array contenente tutti i files che hanno lo stato DESYNCHRONIZED e che quindi devono essere trasmessi al nodo destinatario.

- *int DBConnection_hasSyncStatusOnFile(sqlite3* conn, synchronization*

sync, fileToSync fileToCheck, int folderToSyncID, int result, int* isPending);*

Funzione che ci consente di sapere se attualmente, dato un file ed una sincronizzazione, esiste una sincronizzazione su quel file e se il suo stato corrisponde a PENDING. I risultati sono scritti nelle variabili “result” e “isPending” in logica positiva.

- *int DBConnection_forceUpdate(sqlite3* conn);*

Funzione invocata all'inserimento da parte dell'utente del comando “force update”. Come descritto imposta lo stato di tutti i file non eliminati a DESYNCHRONIZED forzandone così l'effettiva ritrasmissione da parte del *thread* “scanAndSendThread”.

vi. frozenlist

Nome campo	Tipo dato	Significato
_frozenListEntryID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.
synchronizationID_ref	INTEGER	Intero rappresentante la sincronizzazione che si intende mettere in stato “frozen”.
timestamp	TEXT	Data di inserimento del record.
voluntary	INTEGER	Intero che specifica se l'interruzione è volontaria (1) oppure è dovuta a guasti (0).

Ogni volta che una sincronizzazione entrerà nello stato di “frozen” su un nodo, verrà inviato un comando all'altro nodo con cui si è in comunicazione; esso creerà a sua volta una entry all'interno della sua tabella frozen.

Vincoli:

Nessun campo può essere NULL ed il campo “synchronizationID_ref” deve essere univoco.

Metodi di gestione:

- *int DBConnection_addSyncToFrozenlist(sqlite3* conn, synchronization sync, int voluntary);*

Funzione di aggiunta di un nodo alla Frozenlist. L'intero passato all'interno della variabile “voluntary” ci specifica se il comando è stato richiesto dall'utente (1) oppure se è nato a fronte di malfunzionamenti (0) nel nodo attivo, mentre nel nodo passivo sarà sempre (0) e dovrà quindi attendere la ricezione di un comando di sblocco.

- *int DBConnection_deleteSyncFromFrozenList(sqlite3* conn, synchronization sync)*;

Funzione per eliminare tutti i record dalla tabella “frozenlist” che fanno riferimento alla sincronizzazione passata nella struttura “sync”.

vii. commands

Nome campo	Tipo dato	Significato
_commandID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.
nodeID_ref	INTEGER	Intero che identifica univocamente un nodo in riferimento alla tabella “nodes”.
state	INTEGER	Enumerativo che definisce in che stato si trova un comando come definito da protocollo di comunicazione.
creationTimestamp	TEXT	Data di creazione del comando.
nextRetryDate	TEXT	Data di prossima ritrasmissione del comando.
txLeft	INTEGER	Numero di ritrasmissioni disponibili rimaste per il comando. Ad ogni trasmissione questo numero verrà decrementato fino a raggiungere lo 0, portando lo stato del comando a FAILED.

text	TEXT	Testo completo del comando, verrà scritto all'interno del file ".dbcm".

Ogni comando inviato o ricevuto viene memorizzato all'interno di questa tabella secondo le specifiche definite nel Capitolo 2, sezione 2.

Vincoli:

Nessun campo può essere impostato a NULL e la coppia di campi "text", "nodeID_ref" deve essere univoca.

Metodi di gestione:

- *int DBConnection_addCommand(sqlite3* conn, command cmd);*

Aggiunta di un comando all'interno del database.

- *int DBConnection_isCommandOnDb(sqlite3* conn, command cmd, int* result);*

Funzione che controlla se un comando è attualmente presente all'interno del database, restituisce il risultato all'interno della variabile "result" in logica positiva.

- *int DBConnection_getAllCommandsDesync(sqlite3* conn, dtnNode localNode, dtnNode node, command*** commands, int* numCommands);*

Metodo che, passato in input un nodo DTN all'interno della struttura dati "dtnNode", restituisce un array contenente tutti i comandi in stato DESYNCHRONIZED da inviare al nodo.

- *int DBConnection_updateCommandState(sqlite3* conn, dtnNode node, char* text, CmdState newState);*

Funzione che consente di aggiornare sul database lo stato in cui si trova un comando.

- *int DBConnection_getCommandTxLeft(sqlite3* conn, dtnNode node, char* text, int* txLeft);*

Funzione che restituisce il numero di ritrasmissioni ancora disponibili per il comando passato in input.

- *int DBConnection_updateCommandTxLeftAndRetryDate(sqlite3* conn, dtnNode node, char* text, int txLeft, unsigned long long nextRetryDate);*

Metodo che, specificato un comando, ne aggiorna il numero di trasmissioni disponibili e la data di futura ritrasmissione.

- *int DBConnection_getAllCommandsRetransmit(sqlite3* conn, dtnNode localNode, cmdList* cmdListReturn, unsigned long long currentDate);*

Funzione che restituisce una lista contenente tutti i comandi che hanno il valore della data di ritrasmissione inferiore alla data passata nella variabile "currentDate" e che hanno un numero di ritrasmissioni disponibili maggiore di zero.

- *int DBConnection_clearCommandsOlderThan(sqlite3* conn, unsigned long long limitDate);*

Metodo che elimina tutti i comandi in stato CONFIRMED o PROCESSING che hanno la data di creazione inferiore alla data passata in input nella variabile "limitDate".

- *int DBConnection_deleteCommand(sqlite3* conn, int commandID);*

Funzione che, fornendo in input l'identificativo univoco di un comando, ne elimina il record dal database.

- *int DBConnection_setPreviusPendingAsFailed(sqlite3* conn, dtnNode sourceNode, unsigned long long ackDate);*

Metodo, invocato ad ogni ricezione di un comando di tipo ack, che imposta lo stato di FAILED per tutti i comandi ancora in stato PENDING con data di creazione inferiore alla data di creazione dell'ack ricevuto, evitando così la ritrasmissione fuori ordine di comandi a fronte di malfunzionamenti o di perdita di *bundle*.

viii. whitelist

Nome campo	Tipo dato	Significato
_whiteListEntryID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.
nodeID_ref	INTEGER	Campo che identifica univocamente un nodo in riferimento alla tabella “nodes”.
timestamp	TEXT	Data di inserimento del record.

Tabella in cui vengono inseriti tutti i nodi dai quali processare sempre i comandi ricevuti.

Vincoli:

Nessun campo può essere impostato a NULL, il campo “nodeID_ref” deve essere univoco.

Metodi di gestione:

- *int DBConnection_addDtnNodeToWhitelist(sqlite3* conn, dtnNode node);*

Funzione di aggiunta di un nodo alla Whitelist. Ogni *bundle* proveniente da questo nodo sarà quindi automaticamente processato.

- *int DBConnection_isInWhitelist(sqlite3* conn, char* EIDToCheck, int* result);*

Funzione che, passata in input una stringa contenente l'EID del nodo da controllare, ci restituisce all'interno della variabile “result” in logica positiva se suddetto nodo è presente o meno all'interno della Whitelist.

ix. blacklist

Nome campo	Tipo dato	Significato
_blackListEntryID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.

nodeID_ref	INTEGER	Campo che identifica univocamente un nodo in riferimento alla tabella “nodes”.
timestamp	TEXT	Data di inserimento del record.

Tabella in cui vengono inseriti tutti i nodi di cui verranno scartati i *bundle*. All'inserimento di un record in questa tabella in automatico verrà generato un comando di “Block”.

Vincoli:

Nessun campo può essere impostato a NULL, il campo “nodeID_ref” deve essere univoco.

Metodi di gestione:

- *int DBConnection_addDtnNodeToBlacklist(sqlite3* conn, dtnNode node);*

Funzione per l'aggiunta di un nodo all'interno della tabella Blacklist.

- *int DBConnection_isInBlacklist(sqlite3* conn, char* EIDToCheck, int* result);*

Funzione che, passata in input una stringa contenente l'EID del nodo, restituisce all'interno della variabile “result” in logica positiva se il nodo è presente all'interno della Blacklist o meno.

x. blockedByList

Nome campo	Tipo dato	Significato
_blockedByListEntryID	INTEGER	Chiave primaria che identifica univocamente un record all'interno della tabella.
nodeID_ref	INTEGER	Campo che identifica univocamente un nodo in riferimento alla tabella “nodes”.
timestamp	TEXT	Data di inserimento del record.

Tabella in cui vengono inseriti tutti i nodi verso cui è inutile mandare *bundle* in quanto ci hanno inserito nella loro Blacklist.

Vincoli:

Nessun campo può essere impostato a NULL, il campo “nodeID_ref” deve essere univoco.

Metodi di gestione:

- *int DBConnection_addDtnNodeToBlockedByList(sqlite3* conn, dtnNode node);*

Metodo che aggiunge un nodo alla BlockedByList.

- *int DBConnection_amIBlocked(sqlite3* conn, char* EIDToCheck, int* result);*

Funzione che ci consente di sapere se un nodo è inserito nella blockedByList e quindi di conoscere se esso ci ha inseriti all'interno della sua Blacklist. Viene invocata prima di generare un *bundle* in modo da evitarne un inutile invio (se il nostro nodo risulta nella Blacklist del destinatario ogni *bundle* a lui diretto verrà scartato).

xi. Funzioni di utility / gestione del database

- *int DBConnection_createTables(sqlite3* conn);*

Questa funzione ha lo scopo di creare sul database tutte le tabelle precedentemente descritte con i vincoli richiesti.

- *int DBConnection_dropTables(sqlite3* conn);*

Funzione con lo scopo di eliminare, rispettando i vincoli di referenza, tutte le tabelle all'interno del database.

- *int DBConnection_dumpDBData(sqlite3* conn);*

Funzione che crea un istantanea del database scrivendo il contenuto di tutte le tabelle all'interno del file “DTNboxDB.txt”, separando i campi tra loro attraverso il carattere “|”. Utilizzata in fase di test per poter controllare in

che stato si trovano i record senza dover utilizzare programmi accessori. E' possibile importare questo file come foglio di calcolo.

4. Strutture Dati

I moduli di questa sezione hanno il compito di definire tutti i tipi di dato che rappresentano astrazioni degli elementi necessari all'utilizzo del programma, quali ad esempio i nodi DTN o le sincronizzazioni che intercorrono tra essi. Tali strutture verranno poi rese persistenti sul database all'interno di apposite tabelle, come definito nella sezione precedente, con corrispondenza diretta tra tabelle e strutture, salvo qualche rara eccezione. A fronte di questa corrispondenza di seguito verranno fornite solo le descrizioni delle strutture dati di maggior importanza e che non trovano una rappresentazione identica nel database, escludendo dalla trattazione le strutture dati di appoggio, quali ad esempio le strutture necessarie alla gestione delle liste di dati, dando rilievo alle strutture che implementano i comandi previsti dal protocollo di comunicazione e alla struttura rappresentante la coda di comandi in uscita verso un nodo.

Tutti i comandi previsti dal protocollo che non compaiono nelle seguenti sottosezioni non sono attualmente implementati.

i. command

Struttura dati implementata nei file “command.h” e “command.c”, rappresenta quella che, in un linguaggio a oggetti, verrebbe definita come classe astratta. Ha lo scopo di rappresentare ogni comando che viene scambiato tra due nodi DTN definendone gli attributi e le funzioni comuni, consentendo così ai moduli che processano i comandi di poter utilizzare un interfaccia uniforme indipendente dal comando specifico che si sta attualmente eseguendo. Per ogni comando che si intende utilizzare in DTNbox sarà poi necessario implementare il relativo modulo che ne definisce il comportamento specifico, secondo quanto definito dal protocollo di comunicazione.

All'interno della struttura “command” sono presenti i campi:

- text: stringa che rappresenta il testo completo del comando, tipicamente letta dal file “.dbcm”.
- state: enumerativo rappresentate uno degli stati possibili in cui si può trovare un comando (Figura 8).
- message: struttura contenente delle informazioni aggiuntive sul comando quali il nodo sorgente, il nodo destinatario, la prossima data di ritrasmissione del comando e il numero di trasmissioni disponibili attualmente rimaste.
- functionTable: struttura contenente i puntatori a tutte le funzioni che ogni comando specifico dovrà implementare, ossia le funzioni di “createCommand”, “destroyCommand”, “receiveCommand” e “sendCommand”.

Questi campi saranno quindi comuni a ogni comando che viene utilizzato all'interno del programma (che conterrà di conseguenza una struttura di tipo “command”), insieme alla lista di funzioni presenti nella struttura “functionTable” che dovranno essere implementare e specializzate.

Segue una descrizione delle funzioni di “createCommand”, “destroyCommand” e “sendCommand”, essendo simili come logica indipendentemente dal comando specifico, a differenza della funzione “receiveCommand” che invece implementa le operazioni da eseguire in base al comando ricevuto e verrà descritta dettagliatamente in seguito per ogni comando implementato.

- *void createCommand(command** cmd);*

Funzione che ha lo scopo di inizializzare correttamente un comando leggendone tutti gli attributi dal testo che lo compone, convertendoli nel formato opportuno e inserendoli all'interno della struttura specifica del comando.

- *void destroyCommand(command* cmd);*

Funzione che ha il compito di de-allocare la memoria istanziata dalla

funzione “createCommand”.

- *int sendCommand(command* cmd);*

Funzione che imposta lo stato del comando a PENDING in caso sia un comando che richiede risposta o a CONFIRMED in caso sia un comando di ack, viene invocata prima dell'invio effettivo del file tar.

Il modulo “command” implementata poi i seguenti metodi necessari alla creazione e distruzione di un comando generico:

- *void newCommand(command** cmd, char* text);*

Funzione che ha il compito di istanziare un comando ed allocarne la memoria discriminando il tipo in base al contenuto della variabile “text”, ed in seguito invocare la funzione “createCommand” specifica del comando che rappresenta.

- *void destroyCommand(command* cmd);*

Funzione che ha il compito di chiamare il metodo “destroyCommand” specifico del comando e de-allocare la memoria istanziata dalla “newCommand”.

ii. syncCommand

Implementato nei file “syncCommand.c” e “syncCommand.h”, questo modulo rappresenta il comando di “Sync”, avendo al suo interno tutti i campi che lo costituiscono come definito nel Capitolo 2, sezione Comandi.

Il metodo “receiveCommand” è implementato come:

- *int syncCommand_receiveCommand(command* cmd, sqlite3* dbConn, char* ackCommandText);*

Metodo invocato all'arrivo della ricezione di un comando di “Sync”. Come prima cosa viene preparata la base del comando di risposta “SyncAck”.

Da protocollo il comando dovrebbe chiedere all'utente se accettare o meno la richiesta di sincronizzazione ma, a causa della natura della *command line interface* attualmente utilizzata per l'input di comandi da parte

dell'utente, non è possibile notificare questa richiesta in maniera chiara. Ogni richiesta di sincronizzazione viene quindi automaticamente accettata, appena sarà presente l'interfaccia grafica questa parte del programma andrà modificata inserendo, ad esempio tramite comparsa di un “pop-up”, la possibilità all'utente di effettuare la scelta.

Se l'utente accetta la sincronizzazione (sempre vero nel nostro caso), il programma controlla il tipo di sincronizzazione offerta (PUSH, PULL o PUSH&PULL) e, a seconda del tipo, esegue i controlli di legittimità della richiesta, in quanto ad esempio un nodo non può chiedere la PUSH sulla cartella di proprietà di un altro nodo. La funzione procede in seguito a creare o ad offrire la cartella oggetto della sincronizzazione, invocando i relativi metodi di accesso al database per l'aggiunta delle nuove informazioni ricevute.

In caso di rifiuto o violazione di vincoli ciò verrà notificato all'utente e al nodo mittente attraverso il comando “SyncAck”, come da specifiche.

iii. syncAckCommand

Implementata nei file “syncAckCommand.c” e “syncAckCommand.h” questa struttura dati ha lo scopo di rappresentare all'interno di DTNbox il comando “SyncAck” definito dal protocollo di comunicazione . Il metodo “receiveCommand” è implementato come:

- *int syncAckCommand_receiveCommand(command* cmd, sqlite3* dbConn, char* ackCommandText);*

Funzione invocata alla ricezione di un comando di “SyncAck”. Come prima cosa viene ricostruito il testo comando di “Sync” che l'ha generato, che verrà quindi aggiornato allo stato di CONFIRMED. In caso la richiesta sia stata accettata la sincronizzazione viene aggiunta sul database ed eventualmente, se non già presente, viene creata la cartella oggetto della sincronizzazione. In caso di errore o rifiuto sul destinatario ne vengono notificati all'utente i motivi. Infine viene chiamato il metodo “DBConnection_setPreviusPendingAsFailed” che ha lo scopo di mettere a

FAILED tutti i comandi ancora in attesa di ack che sono stati inviati prima di questa richiesta di sincronizzazione.

iv. updateCommand

Modulo relativo al comando di “Update”, implementato nei file “updateCommand.c” e “updateCommand.h”. Viene generato dal *thread* “scanAndSendThread” ogni volta che viene rilevata una modifica del file system su di una cartella oggetto di una sincronizzazione.

- *int updateCommand_receiveCommand(command* cmd, sqlite3* dbConn, char* ackCommandText);*

Metodo invocato alla ricezione di un comando di “Update”. Alla sua esecuzione viene da prima creata la base del comando di “UpdateAck” che verrà generato alla terminazione della funzione, ed in seguito viene ricavato l'identificativo univoco della cartella nella quale andranno modificati i file per poter recuperare le informazioni necessarie su di essa. Viene poi elaborato ogni file presente nella lista di file del comando, a seconda dei seguenti casi:

- Eliminando il file in caso il *flag* “isDeleted” sia stato settato ed il file sia attualmente presente nel file system, cancellandone inoltre i *record* relativi all'interno del database quando opportuno.
- Se il file è stato modificato, e la data di ultima modifica del file inviato è più recente della data di ultima modifica del file presente sul nodo che riceve il comando di “Update”, il file viene sostituito con quello presente all'interno del file tar, aggiornandone di conseguenza tutte le informazioni sul database. Per evitare che due nodi si inviino costantemente tra loro il file, in quanto a causa dei tempi di trasmissione non istantanei quando il file viene copiato dal tar avrà sempre una data di modifica più recente di quella specificata all'interno del comando di “Update”, tale data viene modificata allineandola con quella del mittente, in modo da avere corrispondenza esatta tra i due file system ed evitare un ciclo infinito

di ritrasmissioni.

- Se il file è nuovo, e di conseguenza non è presente nel file system, esso viene aggiunto copiando il file dal tar inviato e inserendo tutti i *record* necessari all'interno del database.

In caso di corretta elaborazione viene poi terminato il comando di “UpdateAck” con la stringa “OK” per notificare il successo dell'operazione al mittente.

v. updateAckCommand

Modulo rappresentante il comando “UpdateAck”, implementato nei file “updateAckCommand.c” e “updateAckCommand.h”. Viene inviato dal programma dopo aver processato un comando di “Update”. Il metodo eseguito alla ricezione ha il seguente comportamento:

- *int updateAckCommand_receiveCommand(command* cmd, sqlite3* dbConn, char* ackCommandText);*

Appena invocata questa funzione si ricostruisce il testo del comando di “Update” che l'ha generato, aggiornandone lo stato a CONFIRMED. In caso il mittente non abbia segnalato errori, viene poi aggiornata la tabella “syncStatus” per ogni file inviato impostandone lo stato a FILE_SYNCHRONIZED (Figura 16) o provvedendo ad eliminarne il suo record dalla tabella in caso il *flag* “isDeleted” sia stato impostato, in quanto ora il programma ha la certezza che il file sia stato eliminato da entrambi i file system. Infine viene invocato il metodo “DBConnection_setPreviusPendingAsFailed” per impostare lo stato di tutti i comandi che aspettano un ack inviati precedentemente a questo a FAILED, per evitare ritrasmissione di comandi fuori ordine.

vi. finCommand

Modulo implementato nei file “finCommand.h” e “finCommand.c”, rappresenta il comando di “Fin” da generare per terminare una sincronizzazione attiva. Alla ricezione di tale comando viene invocato il metodo:

- *int finCommand_receiveCommand(command* cmd, sqlite3* dbConn, char* ackCommandText);*

La prima operazione eseguita da questa funzione è quella di costruirsi la base del comando di “FinAck” da inviare al termine dell'elaborazione. In seguito viene cancellata dal database la sincronizzazione. Il programma inoltre, in caso la richiesta di “Fin” provenga dal nodo proprietario della cartella, provvede a notificare tutti gli eventuali nodi con cui era in sincronizzazione su quella cartella dell'avvenuta terminazione generando tanti comandi di “Fin” quanti sono i nodi. Infine viene eliminata la cartella in oggetto definitivamente dal file system, come da specifica di protocollo.

vii. finAckCommand

Modulo rappresentante il comando “FinAck”, implementato nei file “finAckCommand.c” e “finAckCommand.h”. Il metodo di “receiveCommand” ha il seguente comportamento:

- *int finAckCommand_receiveCommand(command* cmd, sqlite3* dbConn, char* ackCommandText);*

Alla ricezione di un comando di “FinAck” viene ricostruito il comando di “Fin” che l'ha generato e ne viene aggiornato lo stato a CONFIRMED. In caso di corretta elaborazione viene poi notificato all'utente la corretta terminazione della sincronizzazione e viene invocato il metodo “DBConnection_setPreviusPendingAsFailed” per evitare l'invio di comandi fuori ordine.

viii. synchronizationCmdList

Conclude la sezione l'ultima struttura dati di particolare importanza. Come anticipato precedentemente questa struttura ha il compito di accorpare tutti i comandi in uscita verso un nodo destinatario. Viene inizializzata dal processo “mainDTNbox” all'avvio dell'applicazione e viene passata come argomento a tutti i successivi *thread* generati. Per la corretta gestione dell'accesso concorrente alla risorsa, tutte le funzioni che questo modulo implementa sono state messe in mutua esclusione attraverso l'utilizzo di

una particolare variabile di tipo semaforo. Segue un esempio di utilizzo, si rimanda alla lettura del codice per un approfondimento sui vari metodi implementati, in quanto ben documentati.

Nel caso si voglia inviare uno o più comandi ad un nodo, sarà innanzitutto necessario creare la lista relativa ad esso attraverso la funzione “synchronizationCmdList_createList”, operazione idempotente in quanto se il nodo è già presente non viene ricreata la lista. In seguito tramite l'invocazione della “synchronizationCmdList_add” ogni comando verrà aggiunto alla lista relativa al nodo di destinazione, ma solo se esso non è presente nella “blockedByList”, e contemporaneamente verrà aggiunto al database. Infine il *thread* “scanAndSendThread” provvederà periodicamente a svuotare tali liste chiamando il metodo “synchronizationCmdList_createCmdFile” che creerà il file “.dbcm” per ogni nodo, contenente tutti i comandi ad esso destinati, invocando su di essi la funzione “sendCommand” e decrementandone il numero di trasmissioni disponibili, aggiornando di conseguenza il database con tutte le nuove informazioni.

5. Metodi di Utility

I metodi di *utility,* inseriti all'interno dei file “utils.c” e “utils.h”, rappresentano tutti quei metodi invocati in più parti del programma che si è deciso di accorpare in quanto non relativi ad un modulo specifico. In questo modo si evita la replicazione di codice in parti diverse di DTNbox favorendone la manutenibilità e consentendo di impostare determinati comportamenti di default al programma, che in caso di nuove esigenze possono essere semplicemente modificati. Segue una lista non esaustiva dei metodi implementati:

- *unsigned long long getCurrentTime();*

Funzione che restituisce il tempo attuale in millisecondi a partire da EPOCH, tipicamente viene chiamata per generare i *timestamp* di creazione dei comandi.

- *int checkAndCreateFolder(char* folderName);*

Metodo che controlla se la cartella al path assoluto passato in input esista sul file system e, in caso negativo, la crea. Utilizzata all'inizio del programma per creare le cartelle necessarie o all'inizio di una sincronizzazione.

- *void getOwnerFromEID(char* owner, char* EID);*

Funzione per ricavare il proprietario di una cartella a partire dal suo EID. Al momento è supportato solo lo schema “dtn”, in futuro andrà aggiunto il supporto allo schema “ipn” per garantire interoperabilità con le varie implementazioni del *bundle protocol*. Se ad esempio il nodo avesse EID “ipn:2.3” il nome del nodo risulterebbe “2”.

- *int getHomeDir(char* homeFolder);*

Metodo che ricava la cartella “home” dell'utente che sta eseguendo il programma. Utilizzato per creare la struttura di cartelle (Figura 8) o per ricostruirsi il path assoluto di una cartella a partire dal nome relativo presente sul database.

- *Metodi per il controllo delle strutture*

Funzioni utilizzate principalmente dal modulo “DBInterface” per controllare se la strutture a cui fanno riferimento contengono valori significativi. Restituiscono il risultato in logica positiva. Sono state implementate “isDtnNodeValid”, “isFolderToSyncValid”, “isFileToSyncValid”, “isSynchronizationValid”, “isCommandValid”.

- *int tarFilesInDir(char* dirName, char* tarName, int numFiles, char** fileNames);*

Metodo che, data in input una cartella, crea un file tar contenente tutti i file presenti al suo interno. Viene utilizzato dal processo “scanAndSendThread” per la creazione dei file tar da inviare ai vari nodi con cui è in comunicazione.

- *int tarToFiles(char* tarName, char* destDir);*

Estrae il contenuto di un file tar all'interno della cartella specificata al path assoluto indicato nella variabile “destDir”. Viene utilizzato dal processo “parseBundle” per l'elaborazione di un messaggio.

- *unsigned long long getNextRetryDate(dtnNode node);*

Funzione per il calcolo di data di prossima ritrasmissione di un comando diretto verso il nodo specificato nella struttura “dtnNode” in base al *lifetime* del nodo stesso.

- *int getDefaultLifetime();*

Funzione che restituisce un intero contenente il valore di default del *lifetime* in secondi da impostare nel *bundle* in caso di comunicazione con nodo sconosciuto.

- *int getDefaultNumRetry();*

Metodo che restituisce il numero di default di ritrasmissioni di un comando verso un nodo sconosciuto.

Capitolo 4: Test eseguiti

Per verificare il corretto funzionamento dei moduli implementati ci si è avvalsi del programma Virtualbricks [Virtualbricks], che realizza un *frontend* per macchine virtuali Qemu/KVM (VMs) e dispositivi di rete virtuali (VDE), il quale permette di creare *testbed* composti da più VM interconnesse tra loro da elementi VDE. Grazie all'utilizzo di questo strumento è stato possibile avere un ambiente di test uniforme in quanto più utenti possono utilizzare lo stesso *testbed*, avendo quindi la garanzia di avere una situazione di partenza nota per tutti gli utilizzatori. Lo stesso progetto è inoltre facilmente replicabile in quanto il *testbed* è rappresentato come un file condivisibile e solleva l'utente dal dover configurare in maniera opportuna le varie macchine virtuali ad ogni nuova installazione, processo che può risultare abbastanza lungo e difficoltoso in particolare per quanto riguarda l'installazione delle varie implementazioni del *bundle protocol*. Risulta inoltre molto utile in fase di sviluppo in quanto la macchina *host*, sulla quale tipicamente viene svolto lo sviluppo del programma, viene vista dalle altre VM come una macchina virtuale alla quale esse possono connettersi ed è quindi possibile effettuare modifiche al codice e controllarne il risultato in tempo reale.

E' stata quindi effettuata l'installazione di DTNbox all'interno delle varie macchine virtuali per simulare un possibile scenario di utilizzo reale, sfruttando il demone DTN2 per semplicità d'uso e completezza degli strumenti forniti, provando a ricreare per quanto possibile ad esempio una possibile interruzione della rete per verificare l'effettivo funzionamento del meccanismo di ritrasmissione o provando a creare ed eliminare sincronizzazioni tra le varie VM per verificare la correttezza dei dati scambiati. Tutti i comandi ed i moduli implementati sono stati testati e presentano il comportamento desiderato; l'interfaccia utente potrebbe essere, in una futura versione, resa più robusta da un controllo più stretto sulla validità e correttezza degli input forniti dall'utente.

Conclusioni

Come specificato in premessa con questa tesi ci si è posti l'obiettivo di implementare in ambiente Linux un'applicazione di sincronizzazione che permetta lo scambio di file tra due nodi di una rete *challenged*.

Sono partito da un lavoro svolto in una tesi precedente che aveva definito le specifiche di base del programma.

Il mio obiettivo è stato quello di perfezionarne l'architettura e integrare le parti ritenute carenti a fronte di nuove necessità sorte in fase di sviluppo.

Ho quindi:

- Sviluppato tutti i moduli che compongono la struttura base del sistema, ossia i processi di controllo.
- Implementato i comandi di inizio, aggiornamento e fine di una sincronizzazione secondo specifiche, per avere una prima versione utilizzabile del programma, comprese alcune funzioni accessorie utili in fase di test.
- Testato su macchine virtuali il corretto funzionamento dell'intero programma DTNbox.

Così facendo ho portato quella che era una prima stesura del programma ad un livello applicativo concreto ed utilizzabile.

Bibliografia

[al_bp_API]	A. D'Amico, “The Abstraction Layer”, Internal Document, https://github.com/annettaws/al_bp/
[Caini]	C. Caini, H. Cruickshank, S. Farrell, M. Marchese, “Delay- and Disruption-Tolerant Networking (DTN): An Alternative Solution for Future Satellite Networking Applications”, *Proc. IEEE*, vol. 99, no. 11, pp. 1980-1997
[DTN2]	DTN2 project: https://sites.google.com/site/dtnresgroup/home/code/dtn2documentation
[DTNBOX]	C. Caini, M.Nani, I. Spagnuolo, “DTNbox for Android: a DTN Application for Peer-to-Peer Directory Synchronization”, Internal Document
[IBR_DTN]	IBR-DTN project: https://www.ibr.cs.tu-bs.de/projects/ibr-dtn/
[ION]	S.Burleigh, “Interplanetary Ovelay Network (ION) an Implementation of the DTN Bundle Protocol”, In the Proc. of 4th IEEE Consumer Communications and Networking Conference, 2007, pp. 222-226 ION-DTN project: http://sourceforge.net/projects/ion-dtn/
[RFC4838]	V. Cerf, S. Burleigh, A. Hooke, L. Torgerson, R. Durst, K. Scott, K. Fall, H. Weiss, “Delay-Tolerant Networking Architecture”, Internet RFC 4838, April 2007, https://tools.ietf.org/html/rfc4838
[RFC5050]	K. Scott, S. Burleigh, “Bundle Protocol Specification”, Internet RFC 5050, Nov. 2007, https://tools.ietf.org/html/rfc5050
[SQLITE3]	SQLITE3 project: http://www.sqlite.org/
[Virtualbricks]	P. Apollonio, C. Caini, M. Giusti and D. Lacamera, "Virtualbricks for DTN satellite communications research and education”, in Proc. of PSATS 2014, Genoa, Italy, July 2014, pp. 1-14.
[Warthman]	F. Warthman, “Delay-Tolerant Networks (DTNs). A tutorial. Version 2.0”, July2012, http://ipnsig.org/wp-content/uploads/2012/07/DTN_Tutorial_v2.05.pdf