UNIVERSITÀ DI PADOVA **FACOLTÀ DI INGEGNERIA**

TESI DI LAUREA

DRIVEFARM: UN CLIENT PER LA SINCRONIZZAZIONE IN TEMPO REALE DI FILE IN AMBITO CLOUD

DriveFarm: a client for real-time synchronization in Cloud environment

Laureando: Stefano Mandruzzato

Relatore: Carlo Ferrari

Corso di Laurea Magistrale in Ingegneria Informatica

23 Aprile 2013

Anno Accademico 2012-2013

Ad Elisa,
Ai miei genitori.

Prefazione

L'obiettivo del tirocinio svolto c/o **zero12 s.r.l.** è quello di progettare l'applicazione Client del servizio *DriveFarm* il cui scopo è quello di mantenere sincronizzati i files presenti nel PC dell'utente con quelli presenti nel suo profilo Cloud.
Lo scopo secondario della tesi è quello di progettare un algoritmo in grado di identificare in modo univoco i documenti e le successive versioni che si creano di esso nel tempo all'interno di una rete locale.

Il Capitolo 1 ha lo scopo di introdurre il servizio *DriveFarm* e spiegare nel dettaglio lo scopo della tesi.

Il secondo capitolo invece spieghierà i vari algoritmi in letteratura per l'identificazione dei file univoci e l'implementazione utilizzata nel servizio DriveFam.

Il Capitolo 3 invece descrive le tecniche trattate in letteratura per la condivisione dei file in una rete *Peer To Peer*, e l'implementazione scelta per il Client al fine di condividere le risorse all'interno della rete locale.

Nel Capitolo 4 affronteremo le tecniche utilizzate per la sincronizzazione *Client-Server*, il servizio di notifiche, i vari casi di conflitto di sincronizzazione e i modi per risolverli.

Nell'ultimo capitolo daremo una paranomica di tutte le classi presenti nel nostro Client, l'interfaccia grafica presente, e le varie funzionalità che l'utente può disporre per personalizzare a suo piacimento il servizio offerto.

Nell'appendice A sono presenti tabelle e pseudocodici rappresentanti l'algoritmo SHA2 spiegato in maniera dettagliata nel capitolo 1.

Nell'appendice B invece sono presenti le varie chiamate API del servizio DriveFarm offerto dall'azienda **zero12 s.r.l.**.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Cos'è DriveFarm

DRIVEFARM è un nuovo file *manager as a server* progettato e sviluppato per assecondare le esigenze delle Aziende nella gestione del proprio patrimonio documentale (progetti, foto, contratti, cataloghi, ecc...) eliminando tutte le problematiche relative alle infrastrutture hardware fisiche.
DRIVEFARM consente di soddisfare tutte le esigenze di gestione documentale grazie a:

- Repository unico, sempre accessibile;
- Profilazione degli utenti con gestione delle regole di accesso sui file;
- Un'applicazione che facilita il lavoro in mobilità degli utenti;
- Gestione dei permessi per la manipolazione delle informazioni;
- Sistemi di controllo per individuare eventuali abusi sulla gestione documentale;

1.1.1 Profili Utente

A differenza degli altri file manager Cloud disponibili sul Web, **DRIVEFARM** consente al responsabile aziendale o all'IT Manager di gestire ogni singolo profilo utente ed i relativi privilegi. In particolare l'amministratore potrà:

- decidere le caratteristiche di storage dell' utente;
- decidere se consentire all'utente di condividere informazioni con contatti esterni all'azienda;
- decidere se abilitare o disabilitare la funzione di accesso al file server da dispositivi mobili;
- decidere se disabilitare l'accesso web dall'esterno della rete aziendale;

1.1.2 Storage Sicuro e Replicato

DRIVEFARM garantisce un'elevata sicurezza dei dati del'utente grazie alla replica, in tempo reale, su *datacenter* dislocati geograficamente in luoghi diversi.
Drivefarm è stato progettato per trasferire i file attraverso connessioni sicure (HTTPS).
Il sistema garantisce che i dati siano accessibili dal proprietario del profilo, o dall'amministratore aziendale in caso di necessità.

1.1.3 Share and Track Files

Con **DRIVEFARM** l'utente in pochissimi passi può condividere cartelle e documenti direttamente dalla *Web App o Mobile App* con utenti interni ed esterni all'azienda. Il *plus* aggiuntivo che distingue DriveFarm da tutti gli altri file manager Cloud è la capacità di tracciare ogni singola attività di condivisione ed invio dei files. Tramite questa procedura, realizzata nel rispetto delle norme della privacy,il responsabile aziendale o l'IT Manager può comprendere l'uso, fatto dagli utenti, del patrimonio documentale ed identificare eventuali comportamenti non conformi alle policy aziendali.

1.1.4 Monitoring

Documenti, progetti, immagini e altri tipi di file hanno differenti gradi di sicurezza. Quindi, è estremamente importante, per gli amministratori, tener traccia di ogni evento di modifica e condivisione dei file dello *storage* aziendale. Drivefarm, perfettamente conforme alle norme di privacy aziendale, permette un sistema di *logging* delle condivisioni, invii, e download dei file; in questo modo, ogni uso impropio dei documenti sensibili viene identificato.

1.1.5 Mobile

Gli utenti privati e le aziende utilizzati sempre più device mobili e quindi è importante offrire degli strumenti e servizi adeguati per l'utente in modo che questi sia al centro del sistema e che abbia le informazioni delle *cerchie* di utenti intorno a lui. In accordo con le regole definite dell'amministratore, DriveFarm permette agli utenti di accedere ai propri documenti in qualsiasi momento e in qualsiasi luogo si trovino attraverso *device mobili* (Tablet e smartphone) e altri PC dotati di connnessione.

1.1.6 Client per PC

Per poter usufruire del servizio di DriveFarm, con architetture PC e MAC, è necessario installare un software dedicato che consenta all'utente di utilizzare il servizio in maniera completamente trasparente e immediato.
Il client deve consentire quindi, le seguenti funzionalità:

1. Definire quali cartelle del PC mantenere sincronizzate.

2. Identificare in modo univoco tutti i dati che devono essere sincronizzati sia localmente che a livello mondiale.
3. Se viene caricato un nuovo file tramite *Web App o Mobile App* in una delle cartelle sincronizzate questo deve essere riportato subito anche sul PC / Mac.
4. Sincronizzare file mediante la rete locale se possibile senza interrogare direttamente il server di DriveFarm.
5. Visionare lo stato dello spazio di *storage*.
6. Possibilità di draggare i file spot da conservare direttamente all'interno di DriveFarm.
7. Tracciare l'eventuale invio di file .

Inoltre il Client deve dare la possibilità all'utente, non solo di sincronizzare i file con il Server ma, di avere la possibilità di aggiornamento e sincronizzazione dei file direttamente all'interno della rete locale; in modo da diminuire i tempi di latenza di *Upload/Download* di file di grosse dimensioni.
Questa opportunità gioca molto a favore alle imprese i cui dipendenti lavorano agli stessi file ed hanno l'esigenza di lavorare tutti sulla versione più aggiornata dei file in questione. In questo modo è possibile avere una sincronizzazione più efficiente e veloce mantenendo allo stesso tempo la sicurezza dei file utilizzati mediante lo *storage* nel server di DriveFarm.

1.2 Scopo della tesi

Lo scopo della tesi è quello di progettare un algoritmo in grado di identificare in modo univoco i documenti e le successive versioni che si creano di esso nel tempo all'interno di una rete locale.
La fase successiva prevede quindi di sviluppare il Client, il cui scopo è quello di mantenere sincronizzati i files presenti nel PC dell'utente con quelli presenti nel suo profilo Cloud attraverso le API del servizio DriveFarm sviluppato da **zero12 s.r.l**.
Il servizio DriveFarm permette di gestire delle cartelle condivise tra gli utenti di un'organizzazione e, per questo motivo, il Client sviluppato dovrà integrare l'algoritmo di identificazione dei contenuti per consentire di identificare un documento condiviso su cloud all'interno della rete, e poterlo reperire dalla rete locale senza effettuare attività di download direttamente dall'ambiente Cloud.

Capitolo 2

Identificazione univoca dei file condivisi

Per gestire al meglio la sincronizzazione dei file all'interno dell'ecosistema DriveFarm è risultato opportuno cercare di classificare i file presenti in maniera univoca. Questo risulta molto interessante soprattutto per evitare una ridondanza inutile dei file ed evitare un'inutile occupazione di banda (sia in upload che in download) in fase di download e upload delle risorse.
Per esempio, all'interno di una azienda, è molto probabile che due o più dipendenti carichino lo stesso file su DriveFarm in momenti differenti. Risulta quindi utile che il Server di DriveFarm, dopo il primo Upload, si accorga che la risorsa sia già presente e quindi riesca ad evitare i successivi Upload.
In questo modo, non solo il Server risparmierà spazio in memoria salvando il file in questione una sola volta, ma si eviteranno occupazione di banda in Upload (nel caso in cui il file sia di dimensione relativamente grandi), traducibile quindi in risparmio di costo di gestione.

Per affrontare questo problema ci vengono in aiuto le funzioni hash, le quali riescono a rappresentare un messaggio *M* (quindi una sequenza di bit) di lunghezza arbitraria con un valore *h* di lunghezza fissata ovvero:

$$h = H(M)$$

Le funzioni di hash hanno le seguenti particolari proprietà:

- Sono deterministiche: ad un messaggio M deve restituire sempre lo stesso valore *h*.
- Dato *M* è statisticamente impossibile calcolare *h*.
- Dato *h* è statisticamente impossibile calcolare *M* tale che $H(M) = h$
- Dato *M* è statisticamente impossibile calcolare un altro messaggio *M'* tale che $h = H(M) = H(M')$
- E' statisticamente impossibile calcolare due messaggio M e M' tali che $H(M) = H(M')$

2.1 Sicurezza

Il *NIST (National Institute of Tecnhology)* ha pubblicato, nel documento [1], una valutazione della sicurezza necessaria per evitare una compromissione degli algoritmi di hash SHA (Secure hash Algoritm). Essi sono infatti costituiti da un insieme di algoritmi, con una lunghezza del codice hash variabile da 160 bit a 512 bit. Chiaramente, maggiore è la lunghezza del codice prodotto, minore è la possibilità che si verifichi una collisione. Gli algoritmi SHA sono:

- Beta test: SHA-0 (mai pubblicato per una vulnerabilità intrinseca)
- Prima generazione: **SHA1** (160 bit)
- Seconda generazione: **SHA2**: **SHA224** (224 bit) **SHA256** (256 bit) **SHA 384** (384 bit) **SHA512** (512 bit)
- Terza generazione (famiglia **SHA3**): attualmente in fase di standardizzazione.

L'algoritmo SHA-0 venne pubblicato nel documento FIPS PUB 180 nel 1993, e fu ritirato poco dopo la pubblicazione per essere sostituito dall'algoritmo SHA-1, pubblicato nel documento FIPS PUB 180-1 nel 1995. La sostituzione avvenne a causa di una falla che ne minava la sicurezza. Sia SHA-0 che SHA-1 producono un valore di hash di 160 bit. Successivamente, la nuova versione dello standard chiamata FIPS PUB 180-2 e pubblicata nel 2002 ha introdotto la famiglia detta SHA-2, composta inizialmente dalle versioni SHA-256, SHA-384 e SHA-512. L'ultima versione dello standard, FIPS PUB 180-3 del 2008, ha introdotto la versione SHA-224. La differenza tra le versioni della famiglia SHA-2 è dovuta alla diversa lunghezza in bit del codice hash. Quando il messaggio o il file ha dimensione inferiore a 264 bit si utilizzano gli algoritmi SHA-1, SHA-224 o SHA-256, mentre se ha una dimensione superiore, ma inferiore a 2^{128} bit, allora si usano gli algoritmi SHA-384 e SHA-512. La famiglia SHA-3 è stata presentata nell'ottobre 2012 definita con il nome *Keccak* e in attesa di essere standardizzata.

La valutazione sulla sicurezza è riassunta nella seguente tabella:

	2010	2010-2030	2030
SHA-1	insufficiente	insufficiente	insufficiente
SHA-224	buona	sufficiente	insufficiente
SHA-256	buona	buona	sufficiente
SHA-384	buona	buona	buona
SHA-512	buona	buona	buona
Keccak	buona	buona	buona

La National Security Agency statunitense ha stabilito l'impiego dell'algoritmo SHA per proteggere documeti classificati [2]:

	CONFIDENTIAL (livello 1)	SECRET (livello 2)	TOP SECRET (livello 3)
SHA-256	protezione adeguata	protezione adeguata	protezione non adeguata
SHA-384	protezione adeguata	protezione adeguata	protezione adeguata
SHA-512	protezione adeguata	protezione adeguata	protezione adeguata

2.2 algoritmo SHA 512

L'algoritmo SHA 512 calcola un valore di hash lungo 512 bit suddividendo ed elaborando il messaggio, ovvero una sequenza di bit, in blocchi da 1024 bit. Ogni blocco viene convertito in una sequenza di 16 word da 64 bit ed elaborato attraverso una serie di operazioni. L'algoritmo prevede 64 cicli di operazioni da effettuare per ogni blocco.

2.2.1 Padding

Sia *L* la lunghezza in bit del messaggio, il padding viene effettuato aggiungendo un bit **1** seguito da un certo numero di bit **0** (10000...0). In seguito viene aggiunto il valore di L espresso in numero binario da 128 bit. Viene richiesto che *L* non sia più grande di 2^{128} bit. I dati di padding che vengono aggiunti sono tali da rendere la lunghezza del messaggio un multiplo di 1024 bit, quindi essendo gli ultimi 128 bit utilizzati per inserire il valore di L, allora la stringa composta dal bit **1** seguito dai bit **0** dovrà essere tale da rendere il messaggio 128 bit inferiore ad un multiplo di 1024 bit, quindi la lunghezza della stringa modulo 1024 deve essere uguale a 896 (1024 - 128 = 896).

2.2.2 Operazioni

Le operazioni di addizione, rotazione e spostamento sono analoghe a quelle definite nell'algoritmo SHA1, ma operano su word da 64 bit.

2.2.2.1 Addizione

Le addizioni vengono definite modulo 232 , ovvero il risultato viene troncato a 32 bit. In questo modo il risultato non va in overflow e può sempre essere rappresentato da un'altra word da 32 bit. Quindi se $a = 0xF351284C$ e $b = 0xD58AB106$la loro somma non sarà$0x1C8DBD952$ ma invece $0xC8DBD952$.

2.2.2.2 Rotazione

Le rotazioni a destra e a sinistra effettuate su una word da 32 bit vengono definite a partire dalle operazioni di spostamento, o shift, a destra e a sinistra di una word da 32 bit.

Lo spostamento (shift) a sinistra di n bit di una word a 32 bit comporta uno scorrimento a sinistra dei bit che compongono la word. I bit a destra che restano vuoti vengono sostituiti con bit di valore 0 in questo modo: Si considera uno spostamento a sinistra di 10 bit Sia W una word e n il numero di bit da spostare. Allora l'operazione di spostamento a sinistra viene indicata con la notazione: $W << n$

mentre una operazione di spostamento a destra viene indicata in questo modo: $W >> n$

La rotazione a destra o a sinistra è simile allo spostamento, ma a differenza di quest'ultima i bit che vengono persi sono inseriti al posto dei bit 0 di riempimento. Nella rotazione (detto anche spostamento ciclico - cyclic shift) a sinistra vengono inseriti i primi n bit a destra, mentre nella rotazione a destra vengono inseriti gli ultimi n bit a sinistra.
L'operazione di rotazione a destra e a sinistra viene indicata con la seguente notazione: $W <<< n$ e $W >>> n$
Può essere definita convenientemente a partire dalle operazioni di spostamento. Infatti si può pensare che una rotazione di n bit a sinistra sia il risultato di una operazione di or tra la word spostata a sinistra di n bit con la stessa word spostata a destra di 32 - n bit. Allo stesso modo una rotazione di n bit a destra è risultante da una operazione di **OR** tra la word spostata a destra di n bit con la stessa word spostata a sinistra di 32 - n bit.
Spostamento a sinistra di 10 bit:

$$b_{10}b_{11}b_{12}b_{13}b_{14}b_{15}b_{16}b_{17}b_{18}b_{19}b_{20}b_{21}b_{22}b_{23}b_{24}b_{25}b_{26}b_{27}b_{28}b_{29}b_{30}b_{31}0000000000$$

Spostamento a destra di 32 - 10 = 22 bit:

$$0000000000000000000000b_0b_1b_2b_3b_4b_5b_6b_7b_8b_9$$

Quindi si ha:

$$W <<< n = (W << n)OR(W >> (32 - n))$$
$$W >>> n = (W >> n)OR(W << (32 - n))$$

2.2.3 Funzioni

Vengono definite 6 funzioni di cui 2 sono le funzioni *Ch e Maj* utilizzate nell'algoritmo **SHA1**, ma operano su word da 64 bit mentre le altre 4 ricevono come argomento un word e la trasformano attraverso operazioni di xor e di rotazione/spostamento di bit. La differenza rispetto a quelle definite negli altri algoritmi consiste nel numero di bit da spostare.

```
1 Ch(X, Y, Z) = (X and Y) xor ((not)X and Z)
  Maj(X, Y, Z) = (X and Y) xor (X and Z) xor (Y and Z)
3
  S0(X) = (X >>> 28) xor (X >>> 34) xor (X >>> 39)
5 S1(X) = (X >>> 14) xor (X >>> 18) xor (X >>> 41)
  S'0(X) = (X >>> 1) xor (X >>> 8) xor (X >> 7)
7 S'1(X) = (X >>> 19) xor (X >>> 61) xor (X >> 6)
```

2.3 Costanti

Vengono utilizzate 80 word da 64 bit costanti *K0, K1, ..., K79* da utilizzare in ogni ciclo. Sono state ricavate considerando i primi 64 bit della parte frazionale della radice cubica dei primi 80 numeri primi. L'appendice A.1 illustra tutti le 80 word utilizzate in ogni ciclo.

2.3.1 Elaborazione

Il codice hash viene memorizzato in 8 word da 64 bit per un totale di 512 bit *H0, H1, H2, H3, H4, H5, H6, H7* che vengono inizializzate prima dell'elaborazione del primo blocco di dati con i seguenti valori:

```
H0 = 0x6a09e667f3bcc908
H1 = 0xbb67ae8584caa73b
H2 = 0x3c6ef372fe94f82b
H3 = 0xa54ff53a5f1d36f1
H4 = 0x510e527fade682d1
H5 = 0x9b05688c2b3e6c1f
H6 = 0x1f83d9abfb41bd6b
H7 = 0x5be0cd19137e2179
```

Viene utilizzato un insieme di 80 valori *W0, W1, ..., W63* che vengono generati in questo modo:

$$Wt = Mt\ per\ 0 <= t <= 15$$

$$Wt = S'1(Wt-2) + Wt-7 + S'0(Wt-15) + Wt-16\ per\ 16 <= t <= 79$$

I valori da 0 a 15 sono uguali alle 16 word che compongono il blocco da 1024 bit che viene elaborato, mentre i valori rimanenti sono costruiti a partire da quelli precedenti utilizzando le funzioni *S'1 e S'0*.
Si utilizzano 10 word temporanee denominate *a, b, c, d, e, f, g, h, T1, T2*. Le prime 8 vengono inizializzate con il valore delle word che contengono il codice hash temporaneo in questo modo:

```
a = H0
b = H1
c = H2
d = H3
e = H4
f = H5
g = H6
h = H7
```

Il nucleo dell'elaborazione consiste in una serie di operazioni che costituiscono un ciclo. Esistono in totale 80 cicli.
Viene calcolato il valore della variabile T1 come la somma (modulo 264) della funzione **S1** a cui viene passato il valore di *e*, della funzione **Ch** a cui vengono passati come argomenti i valori di *e, f, g,* del valore di *h*, della costante **K** e del valore **W** (anche questi ultimi due dipendono dal ciclo).
Il valore **T2** è dato dalla somma (modulo 232) della funzione **S0** del valore di a e della funzione **Maj** a cui viene passato il valore di *a, b, c*. Vengono ruotati i valori delle variabili in modo da assegnare alla variabile *h* il valore di *g*, a *g* il valore di *f*, a *f* il valore di *e*, ad *e* la somma di *d* con *T1*, a *d* il valore di *c*, a *c* il valore di *b*, a *b* il valore di *a* ed infine ad a il valore della somma di *T1* e *T2*.
Dopo aver effettuato tutti gli 80 cicli, i valori di *a, b, c, d, e, f, g, h* vengono sommati rispettivamente alle variabili *H0, H1, H2, H3, H4, H5, H6, H7*, che ora conterranno il nuovo valore temporaneo di hash. Una volta elaborato l'ultimo blocco da 512 bit, allora il codice hash sarà contenuto proprio nelle variabili *H0, H1, H2, H3, H4, H5, H6, H7*.

```
for i = 1 to N (N numero di blocchi da elaborare)
        a = H0
        b = H1
        c = H2
        d = H3
        e = H4
        f = H5
        g = H6
        h = H7

for t = 0 to 79
        T1 = h + S1(e) + Ch(e, f, g) + K + W
        T2 = S0(a) + Maj(a, b, c)
        h = g
        g = f
        f = e
        e = d + T1
        d = c
        c = b
        b = a
        a = T1 + T2
        H0 = H0 + a
        H1 = H1 + b
        H2 = H2 + c
        H3 = H3 + d
        H4 = H4 + e
        H5 = H5 + f
        H6 = H6 + g
        H7 = H7 + h
```

L'appendice A.1 riassume tutto lo pseudocodice dell'algoritmo, dalla fase di Padding alla sua elaborazione.

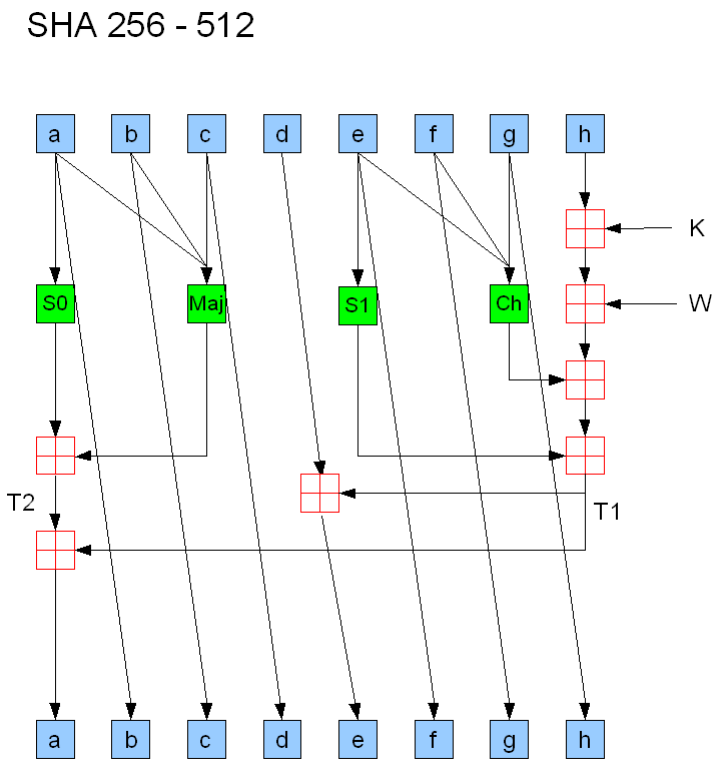

Figura 2.1: *Funzionamento dell'algoritmo SHA a 256 e a 512 bit durante ogni ciclo*

2.4 Keccak

Keccak[3] è stato l'algoritmo vincitore della competizione organizzata dalla *NIST* per selezionare un nuovo algoritmo crittografico di hash [4].
Keccak è stato creato da **Guido Bertoni, Joan Daemen and Gilles Van Assche** provenienti da *STMicroelectronics* and **Michaël Peeters** da *NXP Semiconductors*. Il team riucì ad aver la meglio agli altri 63 candidati al concorso organizzato nel 2007 quando si pensava che SHA2 potesse essere dichiarato vulnerabile in breve tempo.

L'algoritmo Keccak è sostanzialmente una famiglia di *sponge functions*. Una funzione spugna è una generalizzazione del concetto di *cryptographic hash function* con infiniti output e possono eseguire funzioni crittografiche quasi simmetriche, dall'hashing alla generazione di numeri pseudo-casuali e alla l'autenticazione crittografica.

2.4.1 Flessibilità

Come una sponge function, Keccak ha una lunghezza arbitraria di output. Questo fatto riesce a semplificare il modo d'uso dell'algoritmo in quanto riesce ad essere versatile per qualsiasi co-

struzione dedicata.
Esso può essere utilizzato per hashing, *full domain hashing*, hashing randomico, stream encryption e MAC computation. In aggiunta la lunghezza arbitraria dell'output lo rende adatto anche per il *tree hashing*.
Keccak può essere usato in modo efficiente come ad esempio per il *reseedable pseudo-random bit generator* o per l'autentificazione. Questa efficienza deriva dalla mancanza di trasformazione in output.
Keccak inoltre ha una facile versalità per quanto riguarda la sicurezza. Si può avere come obbiettivo un determinato livello di sicurezza scelto a seconda di adeguate capacità. Infatti per una data capacità *c*, Keccak è affermato di respingere ogni attacco fino a complessità $2c/2$ (a meno che genericamente più facile). Questo è simile all'approccio di forza di sicurezza utilizzato in *NIST SP 800-57*.
La sicurezza della codifica hash non dipende dalla lunghezza di uscita. Esiste una lunghezza minima di uscita come conseguenza del livello di sicurezza scelto, per evitare generici *birthday attack*, ma non il contrario: cioè, non è la lunghezza di uscita che determina il livello di sicurezza.
L'istanza proposta per il conoscrso SHA-3 usa una singola permutazione per tutti i livelli di sicurezza. Questo taglia nettamente il costo di implementazione paragonato alle famiglie delle hash function che si avvalgono di due (o più) primitive come l'SHA-2. Inoltre con la stessa permutazione, si può eseguire prestazioni compromesse per quanto riguarda la sicurezza, in modo da scegliere l'appropriato rapporto di capacità.

2.4.2 Design e sicurezza

Keccak ha un grosso margine di sicurezza in quanto si stima che la sponge function di Keccak riesce ad essere sicura anche se il numero di cicli è divisibile per 2.
Il design delle permutazione segue il principio di *Matryoshka*, dove le proprietà di sicurezza delle sette permutazioni sono linkate tra di loro. L'analisi crittografica delle più piccole permutazioni incominciano con il toy *Keccak-f[25]*, il quale è quello più significativo per le più grandi permutazioni, e viceversa. In particolare, percorsi differenziali e lineari in una instanza *Keccak-f* estende a percorsi simmetrici in grandi istanze.
Le *sponge e duplex construction* utilizzate da Keccak sono dimostrabilmente sicure dagli attacchi generici.
Diversamente da SHA-1 e SHA-2, Keccak non ha debolezza della *length-extension* e quindi non necessita di una costruzione HMAC nidificata.
In conclusione le scelte di design di keccak sono abbastanza diverse da quelle di SHA-1 e SHA-2 e anche di *AES (Advanced Encryption Standard)*. Keccak quindi prevede la diversità rispetto agli standard esistenti.

2.4.3 Implementazione

L'algoritmo Keccak eccelle in hardware performance e ed piú efficiente ad SHA-2 in un ordine di un grandezza. Keccak ha complessivamente buone prestazioni software. Anche se si potreb-

be dire che altri finalisti del concorso SHA-3 (come ad esempio *Blake e Skein*) hanno migliori prestazioni software su PC: Keccak risulta piú efficiente quando viene utilizzato con hardware e software che sfruttano i diversi gradi di parallelismo del calcolatore. Su *AMD Bulldozer*, con 128-bit e 256-bit di output hash arrivano 4.8 e 5.9 cicli/byte, rispettivamente.
Per mezzo di un processore con tecnologia *Intel Sany Bridge*, con le stesse funzioni, essi raggiungono rispettivamente 5,4 4 6.9 cicli/byte. Sulle piattaforme vincolate, Keccak ha medie dimensioni del codice e necessità di consumo di RAM.
Per le modalità che coinvolgono una chiave, la protezione dell'implementazione contro *side-channel attack* è voluta.
Le operazioni utilizzate in Keccak consentono contromisure efficaci contro questi attacchi.

Concludendo l'algoritmo Keccak inoltre, ha un vantaggio aggiuntivo: esso non potrà essere vulnerabile nella stessa maniera di SHA-2 per differenza di algoritmica utilizzata. In questo modo, qualora SHA2 (o Keccak) possa essere attaccato, molto probabilmente Keccak (o SHA2) potrebbe resiste ancora, in quanto i due algoritmi sono progettati in maniera differente.

2.5 implementazione utilizzata

La nostra implementazione porta a scegliere l'algoritmo hash SHA-2 a 512 bit contro il nuovo algoritmo Keccak per via della sua non completa standardizzazione nei vari linguaggi di programmazione, in quanto uscito pochi mesi fa durante lo svolgimento della tesi.
Infatti, all'inizio del tirocinio, ci accorgemmo subito del risultato del concorso di *NIST* ma, non avendo le librerie necessarie per implementare il nuovo agoritmo, ci affidammo all' SHA-2.
Dato che, come menzionato in precendenza, la NIST dichiara che SHA risulti un algoritmo affidabile per ancora molto tempo, la nostra scelta non comporta nessun aggravio in termini di sicurezza e neanche in temrini di prestazioni in quanto, secondo il capitolo precende, il Keccak eccelle in termini di prestazioni sono il condizioni hardware specifiche e non *general-purpose*.
Tuttavia, c'è sempre la possibilità di cambiare algoritmo di hashing nelle prossime versioni del client, in quanto è stato implementato in classi indipendenti dal resto del codice e quindi non desta particolari problemialla sostituizione con un nuovo algoritmo di hashing.

Tornando alla nostra implementazione, abbiamo pensato di aumentare la sicurezza, trasferendo, durante la verifica dell'identificazione del file, alcune informazioni aggiuntive.
Sebbene infatti, non vi sia la certezza matematica di non aver collisioni, abbiamo pensato di utilizzare altri tipi di confronto, nel caso due file avessero la stessa chiave hash (e quindi la possibilitá di aver due file identici).

Per garantire un l'identità tra due file, nel caso in cui abbiano la stessa chiave hash:

- E' possibile verificare se i due file abbiano la stessa estensione
- Verificare l'ugualianza di 3 intervalli di bit presi a random dai due file

Nel caso i vari test risultassero positivi, potremmo garantire l'ugualianza tra i due file, aumentando solo l'informazione trasmessa di pochi bit oltre alla chiava hash utilizzata (512 bit).

Capitolo 3

Sincronizzazione dei file nella rete locale

Il client sviluppato offre la possiblità non solo di sincronizzare i file presenti nella cartella locale con il Server, ma anche quello di sfruttare l'alta velocità della rete locale al fine di sincronizzare tra client (anche di utenti diversi) file e cartelle condivise, in modo da diminuire la latenza di ogni aggiornamento o modifica.
Si è pensato quindi di realizzare un protocollo ad hoc che mira a rispettare le seguenti specifiche:

- **funzionalità**: obiettivo principale è quello di ottenere un protocollo di comunicazione funzionante che gestisca i nodi all'interno della rete, dando la possibilità di mettere a disposizione delle risorse e di poterle reperire.

- **scalabilità**: il modulo deve essere progettato il più scalabile possibile per renderlo facilmente espandibile.

- **prestazioni**: si possono valutare le prestazioni a livello locale, come ad esempio le risorse in termini di disco, memoria e cicli macchina, oppure le prestazioni distribuite comprendenti i costi, in termini di numero di messaggi scambiati e dimensione di ciascun messaggio, per il mantenimento della rete e per effettuare le varie operazioni richieste. Per la natura di una protocollo di rete è più importante quest'ultimo aspetto.

- **sicurezza**: La condivisione delle risorse in questo caso non utilizza il paradigma "tutto a tutti" ma il "tutto a pochi": ogni risorsa infatti è condivisa solo con utenti che possono accedere alla cartella condivisa e quindi con gli utenti che hanno i permessi necessari a modificare o accedere a quella risorsa. Noteremo infatti che nella letteratura, le reti p2p non fanno nessuna distinzione a livello di permessi di accesso alle risorse, ma considerano ogni *peer* uguale a tutti gli altri, e tutti possono richiede e scaricare tutte le risorse presenti nella rete p2p.

3.1 Tipi di Reti Peer 2 Peer

Un sistema peer to peer è un insieme di entità autonome ed eterogenee, denominate *peer* o *nodi*, capaci di auto-organizzarsi in una rete non gerarchica di livello applicativo (denominata *overlay network*), che può essere completamente diversa dalla rete fisica.
Il paradigma p2p è l'antitesi della classica architettura client-server: ogni coppia di peer può potenzialmente avviare e completare una transazione perché ogni nodo assume il ruolo sia di client che di server. Il vantaggio principale è quello di offrire un servizio distrubuito più scalabile ed affidabile anche in condizione di elevato *churn rate*, ovvero elevato tasso di aggiunta o rimozione di nodi dalla rete.
Tutte le varie reti Peer 2 Peer si possono complessivamente categorizzare in due insiemi ben distinti: le reti strutturate e le reti non strutturate.

Le **reti non strutturate** non prevedono vincoli strutturali sulla topologia della rete e permettono dunque ad un peer che si unisce alla rete di stabilire connessioni logiche con peer già attivi arbitrariamente selezionati. La crescita disordinata di tali sistemi vincola all'adozione di procedure di *flooding* per la ricerca delle risorse che risultano però inefficienti e poco scalabili. Non è inoltra garantito che, a fronte dell'inserimento di una risorsa sulla rete, essa sia reperibile da qualsiasi peer.

Le **reti strutturate** invece sono proprio l'opposto: infatti ad ogni peer che si connette alla rete viene assegnato un ID in base al quale esso selezionerà i nodi con i quali stabilire un "vicinato", occupando perciò una posizione *weel-defined* sull'overlay. I sistemi strutturati garantiscono dunque un'elevata scalabilità, in quanto ad un sostanzioso aumento in termini di nodi del sistema non corrisponde un eccessivo aumento del traffico generato, e quindi del consumo di banda disponibile.
Inoltre in questa tipologia di reti, anche ogni risorsa viene assegnata in identificativo univoco (utilizzando i metodi parlati nel capitolo 2), e viene "affidata" al nodo con l'ID più "simile" (vedremo nel dettaglio nel paragrafo 3.3). In questo modo viola i principi di sicurezza e di privacy delle risorse condivise, appongiando a sua volta pienamente la totalità del concetto di *file sharing* e *open source*.

3.2 Modello Gnutella

Essendo una rete non strutturata, i peer che entrano nella rete Gnutella non hanno nessun ID che li identifica. Un nodo appena arrivato, per formare delle relazioni con i peer connessi, si affida o alla conoscenza di un nodo già connesso alla rete, oppure ad una web chache *Gnutella Web Caches* in cui è presente una lista di nodi attivi.
Una volta ricevuta una lista di nodi, il peer incomincia ad cercare di formare il suo "vicinato" utilizzando il *ping-pong protocol*.
Ogni nodo conosce solo i suoi vicini e, essendo una rete non strutturata, i contenuti vengono

memorizzati nei nodi ovunque capiti (ossia ogni nodo tiene memorizzato i file che vuole renedere disponibile alla rete) e non in maniera ordinata. L'impressione è quella di avere un grande archivio disordinato dove qualunque cosa può trovarsi ovunque.

La ricerca di una risorsa avviene in questo modo:

- Alla richiesta di un oggetto il nodo invia ai propri vicini un messaggio (Query, PING) per l'oggetto stesso.
- Se qualcuno di loro possiede l'oggetto risponde al nodo che ha inviato la richiesta con un messaggio di successo (Query response, PONG) contentente, per esempio, un indirizzo IP e un numero di porta TCP, che specificano dove può essere trovato l'oggetto.
- In questo modo il nodo che aveva richiesto l'informazione può accedervi attraverso i comandi GET e PUT.
- Se un nodo non e' in grado di soddisfare la richiesta procede solo con l'inoltro del messaggio (Query) a tutti i suoi vicini, tranne quello che ha inviato la richiesta, altrimenti si arriva alla formazioni di loop (infiniti) nella rete.
- Si procede in questo modo finchè non è stata percorsa tutta la rete o, per evitare un flooding nell'intera rete, si può utilizzare un TTL (time to live) definito in precedenza, che stabilisce il numero massimo di hop che la *Query* può effettuare.

Si possono inoltre attuare altre varianti che aumentano le performance di una richiesta di una risorsa.
Una di queste stabilisce un identificativo univoco (QID, query identifier) in ogni *Query*, che lo contraddistingue dagli altri messaggi. Ogni nodo ha un archivio di messaggi Query che ha recentemente elaborato in modo tale che, se il nodo riceve un messaggio con un QID uguale ad uno visto di recente non lo inoltra; in questa maniera vengono interrotti eventuali cicli di secondo o terzo grado.
Ogni volta che la ricerca è andata a buon fine il messaggio di risposta viene inoltrato al mittente di ogni *hop*, arrivando quindi al mittente originario del messaggio.

3.3 Modello DHT e Kademlia

Gli algoritmi DHT vengono molto spesso utilizzati in quanto risolvono egregiamente il problema di scalabilità nelle reti P2P. Nelle reti che utilizzano l'algoritmo DHT ogni file viene associato ad una *key*, che è prodotta dall'hashing del file stesso e ogni nodo nel sistema è responsabile nel salvare un certo range di key.
Ogni peer quindi è identificato da un ID a 160 bit. Possono essere presenti quindi un massimo di 2^{160} peers. Lo spazio degli indirizzi è definito molto grande appunto perché in questo modo

risulta altamente improbabile che due macchine diverse siano mappate con lo stesso ID. La garanzia di avere nodi con ID univoci è molto alta se la rete è formata da un numero massimo di macchine che si aggira sui 260 nodi. Per dimostrare che sia altamente improbabile che due peers vengano mappati con lo stesso ID, basta far riferimento al paradosso del compleanno.
Nella rete Kademlia, con uno spazio di indirizzamento molto grande, ci sarà dunque una bassissima probabilità che due peer vengano mappati con lo stesso ID.
La scalibilità di questi algoritmi dipende dall'efficienza dei suoi algoritmi di routing. Ogni systema DHT proposta utilizza algoritmi di routing con dettagli tra loro differenti. L'aspetto in comune è che un nodo mantiene $O(logn)$ vicini e il routing di una risorsa può avvenire cone $O(logn)$ *hops*.

Kademlia definisce un concetto di distanza che permette di stabilire la lontananza tra due peers. Il metodo usato è lo **XOR**. Per capire se un nodo X sia più vicino ad un nodo Y o ad un nodo Z basta effettuare lo XOR tra XY e XZ. Il risultato minore indicherà il nodo più vicino a X.

Ogni nodo possiede una tabella dei contatti formata da $logn$ righe e tre colonne. In ogni riga è presente l'ID,l'IP e la UDPport di altri peers. I peers inseriti in tabella non sono però scelti a caso ma in base ad un semplice ed ingegnoso criterio.

1. Il primo nodo, scelto in modo casuale, deve appartenere all'altra metà della rete (in termini di ID).

2. Il secondo nodo interrogato deve appartenere alla mia stessa metà della rete ma all'altro quarto

3. Il terzo contatto appartiene al mio stesso quarto ma all'altro ottavo della rete

4. Il quarto "amico" deve appartenere al mio stesso ottavo della rete ma all'altro sedicesimo

K Il k-esimo nodo che scelgo deve appartenere al mio stesso (1/2K-1) di rete ma all'altro (1/2K).

La rete Kademlia può anche essere vista come un albero binario di ricerca. I peers rappresentano le foglie dell'albero. Per facilitare la comprensione poniamo un esempio di tabella per il peer con id **11011101** (tab. 3.1): In grassetto sono stati evidenziati i bit fissati, gli altri potrebbero anche variare a seconda dell'ID dei reali vicini del nodo.

Nella rete Kademlia reale i nodi contattati non sono $log(n)$ ma $20log(n)$, con la costante 20 che viene moltiplicata al logaritmo per rendere la ricerca più efficiente. In pratica non instauro un collegamento con un solo "amico" per ogni livello ma ne contatto 20. In tale modo posso effettuare più ricerche in parallelo su "amici" appartenenti allo stesso sottoalbero. Contattando più peers in parallelo riesco così a trovare quello più vicino alla risorsa in minor tempo di quello che sarebbe necessario contattando un solo nodo alla volta.

ID	IP	port
01000110	aaa	aaa
10100100	aaa	aaa
11100101	aaa	aaa
11001001	aaa	aaa
11010010	aaa	aaa
11011010	aaa	aaa
11011110	aaa	aaa
11011100	aaa	aaa

Tabella 3.1: *esempio di contact table di un peer di nodo **11011101***

3.3.1 Ricerca di una risorsa

Per trovare una risorsa sfrutto i contatti presenti nella mia tabella: verifico chi tra di essi è il più vicino alla risorsa e, una volta individuato il contatto, lo interrogo e gli chiedo di fornirmi il contatto più vicino alla risorsa che è presente nella sua lista. Una volta ottenuto l'indirizzo del nuovo contatto interrogherò anche quest'ultimo e mi farò dare il contatto presente nella sua tabella che è più vicino alla risorsa. Continuo ad eseguire questa procedura finchè non avrò trovato il peer che è più vicino di tutti alla risorsa e che quindi la possiede.
Essendo una ricerca in un albero binario troverò sicuramente una soluzione (negativa o positiva) in $O(log2n)$ passaggi.

3.3.2 Upload di una risorsa

Se si vuole caricare una risorsa nella rete p2p, dopo aver calcolato l'*hashKey*, viene cercato il nodo con ID più vicino alla risorsa in metrica **XOR**. A quel nodo verrà associata la risorsa che io possiedo. Questo fatto comporta solamente che il nodo scelto possiede le informazioni di localizzazione del nodo che esegue l'Upload (indirizzo IP, porta UDP, ecc...).

3.3.3 Download di una risorsa

La prima cosa da fare è individuare l'ID della risorsa corrispondente al file. Successivamente viene interrogato nella lista dei peer il contatto più vicino alla risorsa, il quale manderà le infomazioni del contatto più vicino rispetto alla sua lista, e così via finchè non giungo al contatto che possiede la risorsa. L'ultimo contatto fornirà al *peer* richiedente l'indirizzo dell'host che possiede realmente il file da scaricare.

3.4 Modello Broadcast

Il modello Broadcast non richiede nessuna particolare struttura della rete, sia in fase di bootstrap che in fase di upload di qualche risorsa. Semplicemente il *peer* richiedente di una risorsa, interrogherà tutti i peer, uno ad uno (senza particolare ordine), finchè non risponderà il nodo possessore della risorsa. Ciò comporta un costo nullo (a livello di numero di messaggi scambiati) in fase *bootstrap* che nell'upload di una risorsa, ma comporta un costo molto oneroso in termini computazionali durante la ricerca. Infatti nel caso pessimo verrano interrogati tutti i peer presenti nella rete con un costo computazionale di $O(n)$.

3.5 Protocollo Drivefarm in una rete locale

Il protocollo utilizzato dal Client DriveFarm è una estensione rivisitata ad hoc del protocollo utilizzato da Gnutella. Ogni nodo viene associato all'username dell'utente che utilizza il client DriveFarm e al suo indirizzo IP all'interno della rete locale.

In questa versione sono implementate le seguenti primitive:

- **IMALIVE**: usato per avvertire gli altri nodi della rete, la sua connessione alla stessa.
- **REQUEST**: chiede al destinatario del messaggio se possiede una specifica risorsa, i parametri utilizzati sono il codice hash della risorsa e il nome utente del richiedente.
- **RESPONSE**: risposta positiva ad un messaggio **REQUEST**, i parametri utilizzati sono il nome utente del mittente, l'hashKey della risorsa e una responseKey, ossia una stringa randomica creata dal mittente e utilizzata per indicare la sessione appena instaurata tra i nodi. La risposta ad una **RESPONSE** (**SENDME**) dovrà avere la stessa *responseKey*.
- **SENDME**: richiesta della risorsa al mittente del messaggio **RESPONSE**.
- **FILE**: invio diretto del file in stream di byte.
- **FAILURE** : risposta negativa ad un determinato messaggio ricevuto: i parametri sono l'username del mittente, l'*hashKey* della risorsa, il *typeMessage* del messaggio ricevuto.

Per facilità di implementazione, i vari parametri dei messaggi, sono spediti tramite una stringa in formato JSON nella quale il primo byte identifica sempre il tipo di messaggio spedito.
Nella seguente tabella sono mostrati i valori di identificazione:

3.5.1 Fase di bootstrap

In fase di bootstrap, come precedentemente descritto, il messaggio di comunicazione è il messaggio **IMALIVE** e con payload il nome utente del mittente.

Primitiva	Valore
REQUEST	0x01
RESPONSE	0x02
SENDME	0x03
FILE	0x04
FAILURE	0x05
IMALIVE	0x06

Appena un utente si collega alla rete LAN spedisce in broadcast nella porta specifica il messaggio IMALIVE. A sua volta, ogni utente che riceve questo messaggio avverte il mittente della sua presenza rispondendo con un altro messaggio *IMALIVE*.

Il listato 3.1 e la figura 3.1 spiegano nel dettaglio i vari passaggi nella fase di bootstrap.

Listing 3.1: *pseudocodice che descrive la fase di bootstrap*

```
(Alice)
send to broadcast address IMALIVE message with username as payload.

for each IAMLIVE message received on broadcast address do:
                    update address table on the database
                    send IMALIVE message to sender's address
```

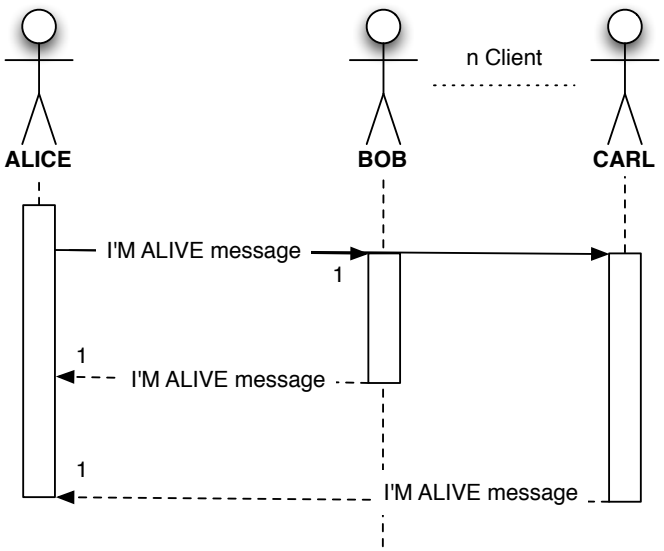

Figura 3.1: *diagramma UML che descrive la fase di bootstrap*

3.5.2 Fase di searching

La fase di searching descrive la ricerca di un file (risorsa) all'interno della rete locale da parte del Client, dopo aver ricevuto una notifica di modifica o creazione della risorsa da parte del Cloud Server.

Appena ricevuto la notifica (con presente l'hashkey della risorsa da cercare), il client interroga i vari nodi della rete avendo la lista degli utenti collegati nel database salvato in RAM. Quindi il client incomincia a mandare massaggi REQUEST scorrendo la lista degli utenti della rete e spedendo il messaggio **SOLO** agli utenti che possono aver accesso alla risorsa. Ad ogni invio, il client controlla se la richiesta della risorsa è stata soddisfatta, altrimenti continua ad interrogare i nodi della lista.

Per quanto riguarda i messaggi ricevuti da altri nodi, il client si comporta in modo diverso a seconda del messaggio ricevuto:

- **REQUEST**: controlla se è in possesso del file richiesto; nel caso positivo controlla se il mittente del messaggio hai i permessi di leggere e/o modificare il file richiesto controllando i permessi del mittente nel suo database; quindi risponde con un **RESPONSE** generando una responseKey e aggiunge alla tabella responseSent presente nel database il messaggio inviato.
- **RESPONSE**: controlla se è stato richiesto il file specificato nel messaggio, quindi invia un messaggio SENDME con la stessa responseKey del messaggio di *RESPONSE* e aspetta il messaggio *FILE* con il bytestream del file richiesto.
- **SENDME**: controlla se il messaggio di SENME ricevuto possa corrispondere ad un messaggio di *RESPONSE* inviato presente nella tabella del database *ResponseSent* (baste verificare che la responseKey e il mittente del file SENDME siano uguali ad una responseKey e ad un destinatario di un messggio *RESPONSE* presente nella *ResponseSent Table*). In caso positivo viene il messaggio *FILE* al mittente contenente il bytestraem nel file richiesto.

Listing 3.2: *pseudocodice che descrive la fase di searching*

```
i=0
for each address in addressList do:
        if i==k:
                if request has been satisfied:
                        break

        if address has permission for file:
                send REQUEST to address
                i++

if not request has been satisfied:
        download file to Cloud Server


for each message received do:

```

```
        case of message:
18
                -"REQUEST":
20                      if the client has the file:
                                check if sender has permission of the file
22                              if not:
                                        send FAILURE to sender
24                              else:
                                        generate responseKey
26                                      send RESPONSE with the reponseKey to sender
                                        add RESPONSE to ResponseSent list
28                      else:
                                send FAILURE to sender
30
                -"RESPONSE":
32                      if the file is requested:

34                              send SENDME with the reponseKey to sender
                                wait the FILE message
36                              save bytestream to the file
                        else:
38                              drop the message

40              -"SENDME":
                        if the client has sent the response message:
42                              open the file
                                send FILE message with bytestrem of file
44                      else:
                                drop the message
```

3.5.3 Calcolo delle prestazioni

Analizziamo le prestazioni del protocollo utilizzato.
Nel caso peggiore un utente A interroga tutti i client connessi alla rete ricevendo risposta positiva soltanto dall'ultimo utente interrogato. Ciò comporta che in questo caso, tutti i peer della rete abbiano accesso alla risorsa.
Supponendo quindi che siano presenti *n* nodi nella rete, vengono generati *n-1* messaggi di *REQUEST*. Il numero di messaggi è quindi direttamente proporzionale al numero dei nodi presenti nella rete, quindi e' un O(n).
Questo è giustificato dal fatto che non esista nessuna rete strutturata all'interno della rete LAN. C'è da considerare però che in una LAN l'infrastuttura presenta una forma molto simile alla figura 3.3 e 3.4, in cui ogni dispositivo è collegato a uno o più router centrali: per cui creare una rete a hoc (ad esempio una rete Gnutella o una DHT) per la comunicazione tra i dispositivi, non ottimizza l'overhead della rete. Infatti l'overhead della rete di un messaggio tra un nodo A e B e il suo forward tra B e C corrisponde allo stesso overhead dello stesso messaggio spedito ad B e C direttamente da A; viene soltanto distribuito il "carico" computazionale su due nodi anzichè su uno.

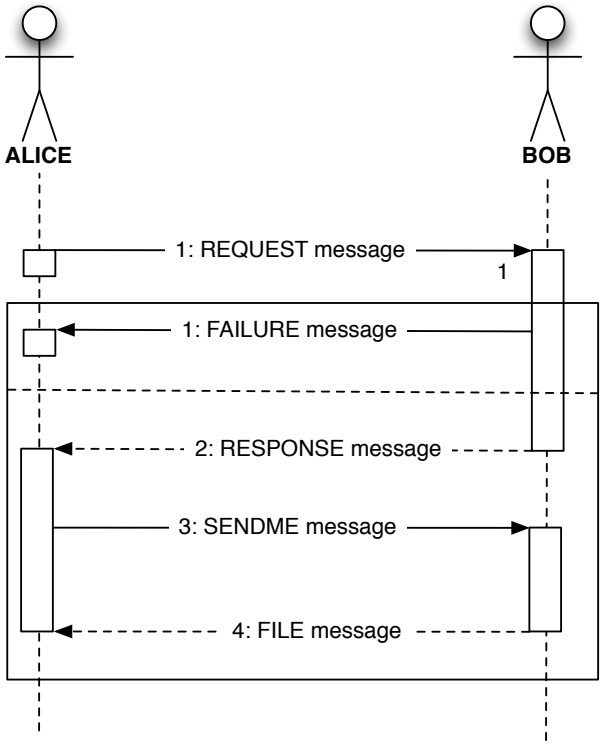

Figura 3.2: *diagramma UML che descrive la fase di searching di un file*

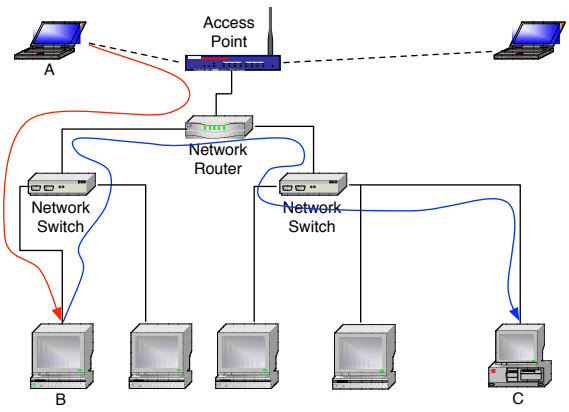

Figura 3.3: *esempio di messaggio tra A a B e il suo forwading verso C*

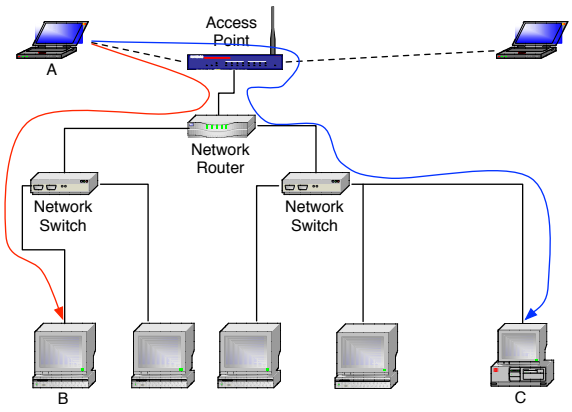

Figura 3.4: *esempio di messaggio sepdito a B e a C direttamente dal nodo A*

Capitolo 4

Sincronizzazione

La sincronizzazione dei file è il fulcro dell'applicazione Client e quindi anche la parte più delicata. La sincronizzazione delle risorse avviene quando queste vengono modificate, cancellate o create sia in lato Client, sia in Lato Server. Bisogna distinguere i vari casi:

- Modifica, creazione e rimozione di file in locale da parte di un utente $\Longrightarrow$ aggiornamento della cartella condivisa nel Server.
- Modifica, creazione e rimozione di file nel Server $\Longrightarrow$ aggiornamento della cartella condivisa nel Client DriveFarm.

4.1 Modifica di un file a lato Client

Le azioni compiute in locale (modifica, creazione, cancellazione) vengono notificate al Server in modo che i file in locale e in remoto siano sincronizzati alla stessa versione (figura 4.1).

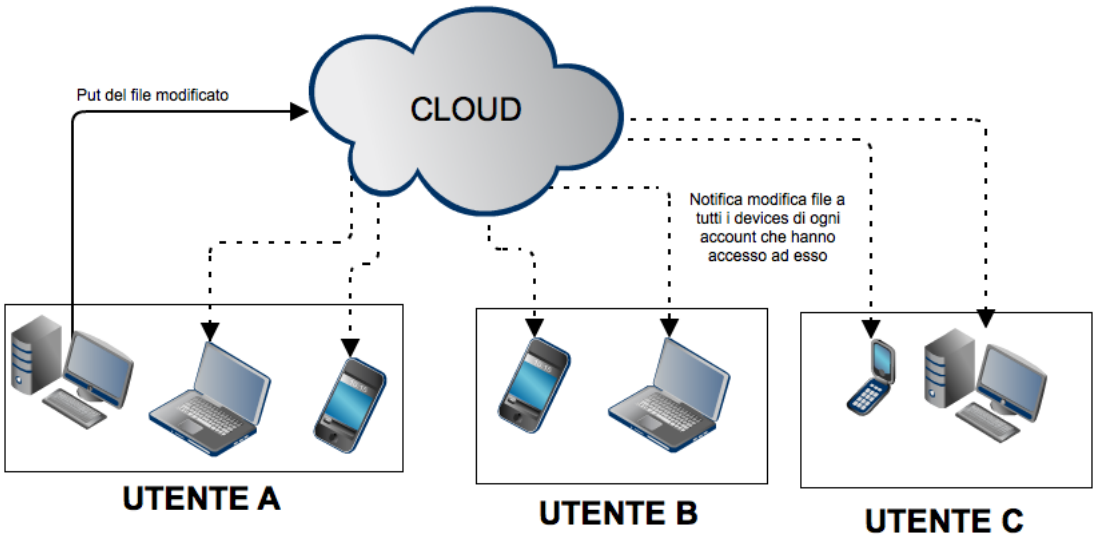

Figura 4.1: *Scenario di modifica di un file da parte di un client*

4.2 Modifica di un file a lato Server

Quando viene modificato a lato Server (o tramite web browser o addirittura dopo un aggiornamento di file per mezzo di un altro Client), il Server ha il compito di notificare ad ogni Client dell'avvenuta modifica.
Il numero di notifiche per ogni file modificato sarà: $\#notificafile = \#user\#devices$.
Per ogni notifica ricevuta, il Client proverà a trovare il file aggiornato nella rete locale utilizzando il protocollo spiegato nel capitolo 3.5 (figura 4.2), altrimenti lo scaricherà dal server utilizzando le API DriveFarm già sviluppate da zero12 (appendice B.1).
L'implementazione delle notifiche é spiegata del dettaglio nella prossimo paragrafo.

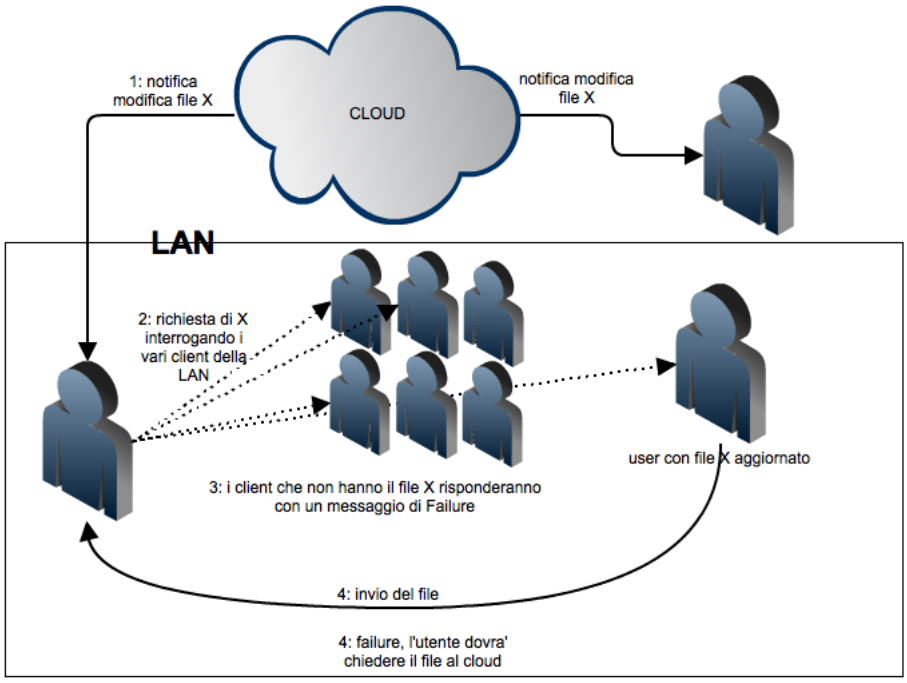

Figura 4.2: *Scenario di richiesta di un file da parte di un client dopo aver ricevuto la notifica da parte del Cloud Server*

4.3 Gestione delle notifiche

Come abbiamo già precedentemente descritto, risulta indispensabile implementare un servizio di notifiche da Server a Client per informare quest'ultimo dell'aggiornamento e della modifica di file nell'ambiente DriveFarm.
Questo servizio di notifiche deve tener conto di queste problematiche:

- Il Server deve saper identificare in maniera univoca ogni Client utilizzato e sapere inoltre a che utente è associato.

- Il Server deve garantire l'invio delle notifiche ad ogni Client coinvolto e quest'ultime devono essere personalizzate per ongi Client coinvolto.

- Le notifiche devono essere persistenti: il Client quindi deve aver la possibilità di ricevere le notifiche in tempo reale, qualora risulti connesso al Server; oppure deve aver la possibilità di ricevere le notifiche generate durante il sua fase offline, una volta che si connette al Server.

Per garantire questi servizi, ci viene in aiuto il sistema fornito da java: JMS (Java Message Service)[5]. Il JMS è un insieme di API, appartenenti al linguaggio Java, che consente ad applicazioni Java di scambiarsi messaggi tra di loro. JMS inoltre è definito dalle specifiche *JSR 914*[6].
Con questa tecnologia è possibile utilizzare la politica *Publish-Subscribe Messaging*. Questo approccio è utilizzato quando molteplici applicazioni devono ricevere messaggi spediti dallo stesso mittente. Il concetto centrale in un sistema di messaggistica *Publish-SubScribe* è il **Topic**. Molteplici *Publisher* possono mandare messaggi al Topic specificato e tutti i *Subscribers* al Topic ricevono tutti i messaggi presenti in quel Topic. In altre parole con il termine *publisher* viene indicato in *producer* dei messaggi, mentre con il termine *subscriber* il *consumer*.Nella figura 4.3 viene illustato il maniera dettagliata questo concetto.

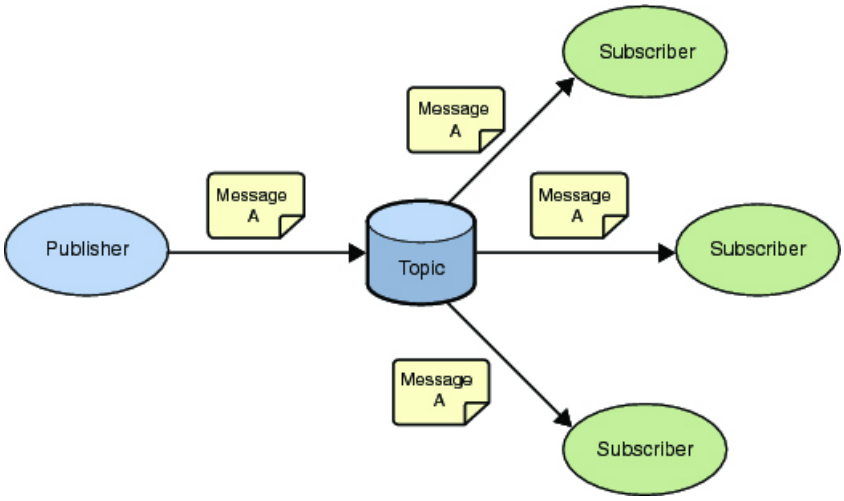

Figura 4.3: *Approcio Publish-Subscribe Messaging secondo la tenologia JMS*

Nel nostro caso, il Server fa le veci del *publisher*, mentre i *subscribes* sono tutti i Client che si affidando al servizio di DriveFarm identificati dal codice univoco *accessToken*, generato nella fase di registrazione.
Il topic generato dal server è denominato ***jms.topic.desktopClient*** in cui viene immesso ogni messaggio indirizzato ad uno specifico Client. Grazie alla concezione dei **subTopic** implementata grazie ad un filtro nel messaggio **subscribe** di JMS, il Client riceverà solo i messaggi indirizzati a lui: il selector utilizzato è: "***rf_push_subtopic = '<accessToken>'***".
JMS offre la possibilità di garantire la ricezione delle notifiche anche quanto il *subscriber* non risulta connesso al server JMS, ricevendo quindi le notifiche alla successiva connessione al server. Per essere sicuri di questo bisogna rendere i messaggi spediti dal *publisher* di tipo *PERSISTENT* e in aggiunta il *subscriber* deve utilizzare una sottoscrizione al topic di tipo *durable*. Una sottoscrizione di tipo *durable* fa in modo di considerare in maniera differente la *subscription* con la connessione del *subscriber* rendendo la prima persistente anche quando il *subscriber* non risulti connesso al server JMS. Le figure 4.4 e 4.5 illustrano in maniera dettagliata tale differenza.

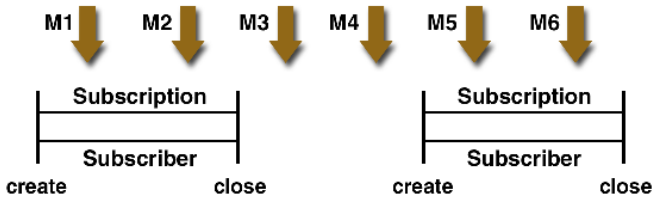

Figura 4.4: *Esempio di Nundurable Subscriber, i messaggi M3 e M4 vengono persi*

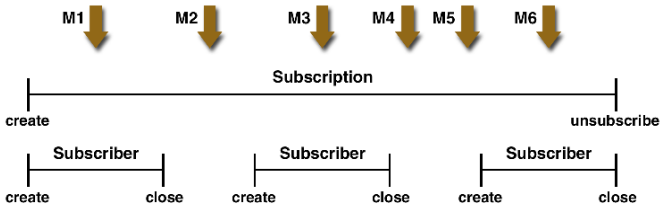

Figura 4.5: *Esempio di Durable Subscriber, viene garantita la ricezione di tutti i messaggi*

Per quanto riguarda la compatibilità con la tecnologica JMS, in lato Server non abbiamo problemi in quanto è implementato da JBoss e nel quale è già disponibile un server JMS *HornetQ*[7]. Il problema rimane lato Client per il fatto che, nel linguaggio Python, non è presente una libreria standard o opensource che implementi un JMS Consumer.

Ci viene in aiuto però **STOMP**(Streaming Text Oriented Messaging Protocol)[8]: esso è un protocollo di facile implementanzione derivante dal design del'HTTP, con il quale è possibile facilmente implementare un client consumer. La maggior parte del server JMS (tra cui anche il nostro HornetQ) supportano il protocollo STOMP e gestiscono in automatico il mapping tra STOMP-JMS protocols.
Avendo un design simile a messaggi HTTP, i messaggi STOMP utilizzano messaggi con *header* e *body*.

Il protocollo STOMP lavora tramite sockets a livello TCP e i comandi che utilizzeremo sono:

- **SEND**: spedisce una messaggio alla destinazione specificata nell'header del messaggio, il body message invece identifica il corpo del messaggio.
- **SUBSCRIBE**: è usato per effettuare una subscription ad una determinata destinazione definita nell'header del messaggio. In questo caso il body messagge è ignorato.
- **UNSUBSCRIBE**: simile al *SUBSCRIBE*, è usato per rimuovere il mittente dal messaggio ad una determinata iscrizione (specificata sempre nell'header del messaggio).

I messaggi di notifica inviati dal Server avranno come body una stringa codificata in JSON con vari parametri a seconda del tipo di notifica, specificata dal parametro *type*. Tutti i messaggi sono identificati a livello temporale del parametro *time* che indica il tempo della pubblicazione della notifica da parte del Server. I tipi di notifica sono:

- **CREATE**: notifica la creazione di un file. I parametri presenti nella stringa JSON sono: *srcPath* che identifica il percorso del file assoluto alla cartella DriveFarm, *hashKey* che identifica la chiave hash del file.
  (ES: {“time”: “123456.8327”,“type”: “create” ,“srcPath”:“/pippo.txt”,
  “hashKey”:“abcdef1234567890”})

- **MODIFY**: notifica di una modifica di un file. I parametri presenti nella stringa JSON sono: *srcPath* che identifica il percorso del file assoluto alla cartella DriveFarm, *hashKey* che identifica la nuova chiave hash del file.
  (ES: {“time”: “123456.8327”, “type”: “modify”, “srcPath”:“/pippo.txt”,
  “hashKey’:“1234567890abcdef” })

- **MOVE**: segnalazione di un move di un file, comprende sia uno spostamento di un file da una cartella all’altra, sia una semplice rinomina. I parametri sono *srcPath* che indica il percorso originario del file, *destPath* indicante il nuovo percorso del file e la sua *hashKey*.
  (ES: {“time”: “123456.8327”,“type”: “move” , “hashKey”:“abcdef1234567890”,
  “srcPath”:“/pippo.txt”, destPath: “/folder/pippo.txt” })

- **COPY**: segnalazione di una copiatura di un file. Questo tipo di notifica presenta gli stessi parametri di una notifica *MOVE*.
  (ES: {“time”: “123456.8327”, “type”: “copy” , “hashKey”: “abcdef1234567890”,
  “srcPath”:“/pippo.txt”, destPath: “/folder/pippo.txt” })

- **DELETE**: cancellazione di un file. L’unico parametro passato oltre il *type* e il *time* è *srcPath*.
  (ES: {“time”: “123456.8327”, “type”: “delete” , “srcPath”:“/pippo.txt”})

- **CREATEFOLDER**: creazione di una nuova cartella. Viene specificato il percorso della nuova cartella tramite il parametro *srcPath*.
  (ES: {“time”: “123456.8327”, “type”: “createFolder” , “srcPath” : “/folder1” })

- **MOVEFOLDER**: spostamento di una cartella in un’altra locazione. Allo stesso modo della notifica di un move per un file, i parametri presenti in questo tipo di notifica sono: *srcPath*, *destPath*.
  (ES: {“time”: “123456.8327”, “type”: “moveFolder” ,
  “srcPath”:“/folder1” , “destPath”:“/folder2” })

- **COPYFOLDER**: copiatatura di una cartella in un’altra locazione. Questo tipo di notifica presenta gli stessi parametri di una notifica *MOVEFOLDER*.
  (ES:“time”: “123456.8327”, “type”: “copyFolder” ,
  “srcPath”: “/folder1” , “destPath”: “/folder2’)

- **DELETEFOLDER**: cancellazione di una cartella. L’unico parametro passato è *srcPath*.
  (ES: “time”: “123456.8327”, “type’: “deleteFolder” , “srcPath”: “/folder1”)

- **ADDTOFOLDER** e **REMOVEFROMFOLDER**: Queste due tipi di notifiche vengono utilizzate per avvertire i Client relativamente a quali sono siano gli utenti che possono accedere alle risorse di una determinata cartella condivisa. Queste informazioni risultano utili per l'invio di notifiche *p2p* quando la rete locale non riesce a raggiungere il Server di DriveFarm (dettagli nel paragrafo 4.4). Le seguenti notifiche hanno come parametri *srcPath* indicante il percorso della cartella condivisa, relativo all'utente mittente del messaggio, e il *codeFolder* il quale identifica in maniera univoca la cartella condivisa nell'ecosistema DriveFarm. Questo codice identificativo è formato dal nome dell'utente owner della cartella, il nome della cartella condivisa relativa all'owner, e il timestamp della prima condivisione di quella cartella.
  Questo codice identificativo serve ai vari Client per identificare in maniera univoca la cartella tra loro condivisa in quanto la path di quella cartella potrebbe essere molto differente tra i vati utenti coinvolti. I prossimo paragrafo spiegherà nel dettaglio questo aspetto con un pratico scenario.

4.3.1 Esempio

In zero12 Stefano Dindo vuole condividere a tutti i dipendenti, collaboratori e stagisti dell'azienda una cartella in cui sono presenti tutte le documentazioni di un progetto. La cartella ha come path su DriveFarm di Dindo "/project/documentation". Questa cartella viene quindi condivisa tra 7 persone: Stefano Dindo, Daniel Sperotto, Simone Maratea, Roberto Contiero, Stefano Mandruzzato, Mattia Peterle, Lorenzo Scorzato.

Quando Stefano Dindo condividerà la cartella, ogni client di ogni utente riceverà 1 notifica addToFolder con parametro username i 6 utenti che condividono la cartella con l'utente destinatario.
Quindi il Server invierà ad ogni utente, per una condivisione di una cartella con n persone, 1 notifica addToFolder nel cui parametro username saranno posti n-1 utenti che sono tutti quelli coinvolti nella condivisione escluso il destinatario della notifica).
Ritornando al nostro esempio, Stefano Dindo riceverà la seguente notifica:

```
{''type'': ''addToFolder'' ,
''srcPath'': ''/project/documentation'',
code: ''s.dindo@zero12.it#documentation#6583028236'' ,
recipients: [''d.sperotto@zero12.it'', ''s.maratea@zero12.it'',
''r.contiero@zero12.it'', ''stefano.mandruzzato@gmail.com'',
''mattia.peterle@gmail.com'', ''lorenzo.scorzato@gmail.com'']}
```

Roberto Contiero, che aveva già presente una cartella "/Shared/documentation" su DF, avrà la nuova cartella condivisa con path "/Shared/documentation_1" e quindi riceverà le seguenti notifiche:

```
''type'': ''addToFolder'' ,
''srcPath'': ''/Shared/documentation_1'',
```

```
code: ''s.dindo@zero12.it#documentation#6583028236'' ,
recipients:[ ''s.dindo@zero12.it'', ''d.sperotto@zero12.it'',
''s.maratea@zero12.it'', ''stefano.mandruzzato@gmail.com'',
''mattia.peterle@gmail.com'', ''lorenzo.scorzato@gmail.com'']
```

Qualche giorno dopo, l'owner (Stefano Dindo) decide di aggiungere un nuovo utente alla condivisione della cartella: *pincopallo@pinco.it*.
Il nuovo utente avra' la nuova cartella condivisa nella path "/Shared/documentation" in quanto non aveva nessuna cartella di nome "documentation" su Shared, e riceverà le seguenti notifiche:

```
{''type'': ''addToFolder'' ,
''srcPath'': ''/Shared/documentation'',
code: ''s.dindo@zero12.it#documentation#6583028236'' ,
recipients: [''s.dindo@zero12.it'', ''d.sperotto@zero12.it'',
''s.maratea@zero12.it'', ''r.contiero@zero12.it'',
''stefano.mandruzzato@gmail.com'', ''mattia.peterle@gmail.com'',
''lorenzo.scorzato@gmail.com''] }
```

Mentre gli altri utenti riceveranno la seguente notifica:

```
{''type'': ''addToFolder'' ,
''srcPath'': ''<pathFolderDellaCartellaCondivisa>'',
code:''s.dindo@zero12.it#documentation#6583028236'' ,
recipients:[''pincopallo@pinco.it''] }
```

Anche in questo caso, come per i messaggi inviati tra i Client nella rete locale, i tipi di notifiche vengono "tradotti" nel seguente modo per un risparmio di byte:

Primitiva	Valore
CREATE	0x11
MODIFY	0x12
DELETE	0x13
MOVE	0x14
COPY	0x15
CREATEFOLDER	0x21
DELTEFOLDER	0x22
MOVEFOLDER	0x23
COPYFOLDER	0x24
ADDTOFOLDER	0x31
REMOVEFROMFOLDER	0x32

4.4 Notifiche tra client nella rete locale in fase OFFLINE

Se la rete locale non è collegata ad internet (oppure semplicemente non riesce a raggiungere il Server) risulta impossibile ,con i protocolli di comunicazione spiegati in precedenza, che utenti della stessa rete locale riescano ad aggiornarsi tra loro files tra loro condivisi. Il motivo è semplice: per ora la sincronizzazione tra Client avviene sempre tramite l'aggiornamento del Server e successivamente tramite le notifiche Server-Client agli altri Client.
Per risolvere questo problema abbiamo quindi implementato altre due tipi di messaggi che vengono utilizzati tra Client della rete locale qualora quest'ultima non riesca ad raggiungere il Server di DriveFarm (fig 4.6).
I due messaggi sono:

- **NOTIFY**: messaggio di notifica qualora il file in questione risulti condivisono con altri utenti. La notifica quindi viene spedita ai Client che possono accedere alla cartella condivisa ma aventi un nome utente DIVERSO dal mittente (la sincronizzazione tra Client con lo stesso username avviene tramite il messaggio *NOTIFYTOME*. I parametri previsti per questo messaggio sono:
  - *Type*: indica il tipo di modifica effettuata al file
  - *CodeFolder*: indica il codice univoco della cartella condivisa in cui è presente il file
  - *relativePath*: indica il percorso del file relativo alla cartella condivisa: con questo parametro e il precedente, il destinatario del file riuscirà ad avere la path corretta del file nel suo ecosistema DriveFarm.
  - *username*: username del mittente. Il destinatario controllerà se il mittente può aver accesso al file in questione e quindi garantire la regolarità della notifica.
  - *hashKey*: codice hash del file
  - *arg*: argomento secondario relativo al tipo di notifica spedita. Può essere la nuova chiave hash del file nel caso in qui la notifica si di tipo "modified", oppure la destinazione relative del file nel caso in cui la notifica sia del tipo "moved".
- **NOTIFYTOME**: messaggio di notifica spedita a Client con lo stesso username del mittente (per cui utenti che scuramente hanno accesso a tutti i file del mittente).
  I parametri previsti per questo messaggio risultano simili a quello precedente ma non è presente il parametro *codeFolder* il quale non più risulta necessario in quanto il percorso del file coinvolto è uguale sia per il mittente che per il destinatario, e il parametro *relativePath* il quale indica, in questo caso, il percorso del file assoluto alla catella DriveFarm.

I byte relativi a questi due messaggi saranno:
Appena uno di questi Client coinvonti riesce ad raggiungere il Server, ha il compito di aggiornarlo utilizzando le API di DriveFarm.

Primitiva	Valore
NOTIFY	0x07
NOTIFYTOME	0x08

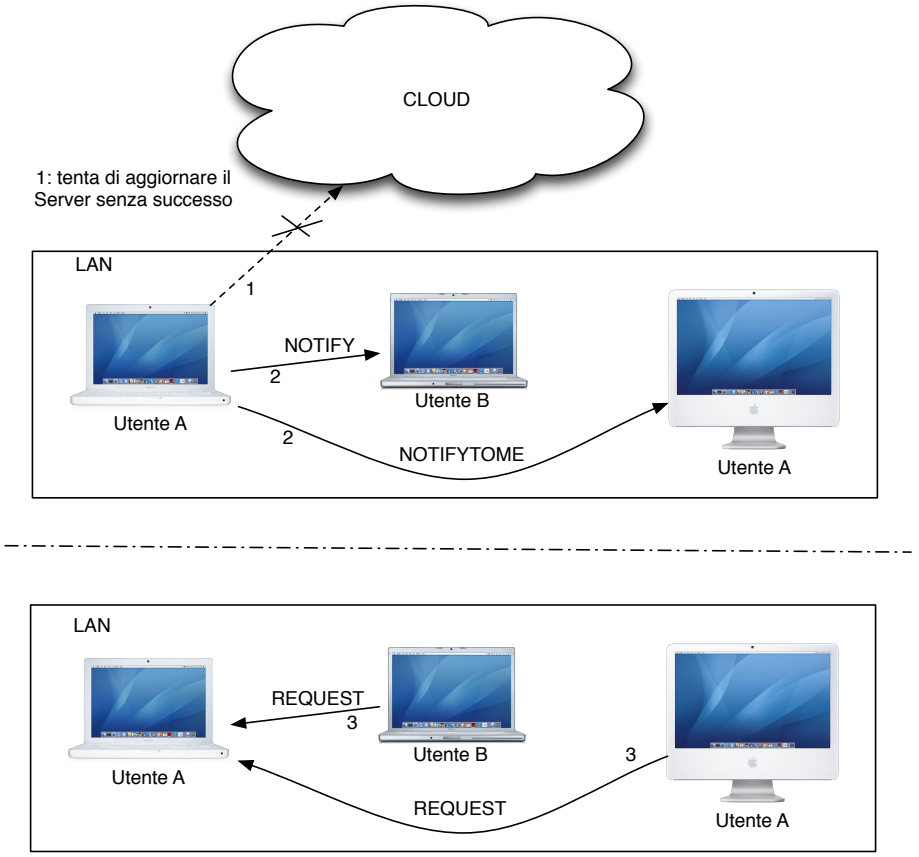

Figura 4.6: *Aggiornamento di un file in una rete locale senza accesso al Server DriveFarm*

4.5 Fase di avvio del Client

Il Client viene avviato a discrezione dell'utente per cui può succedere che la creazione e la modifica di file avvenga quando il Client risulti non funzionante e quindi nemmeno il Server sia aggiornato relativamente a queste modifiche. Il compito di Watcher infatti è quello di tener traccia di tutte le modifiche relative ai file della cartella di DriveFarm sia quando il Client è in funzione sia quando non lo è. Per questo motivo il *Watcher* è stato implementato in un processo a parte e DEVE essere funzionante sin dall'avvio del sistema operativo.
Il *Watcher*, qualora il Client risulti in funzione, lo avviserà ad ogni evento di modifica, cancellazione, creazione, tramite connessione TCP nella rete "localhost". Nel caso in Client non sia in funzione, ogni evento verrà salvato nel database dell'applicativo.

Appena il Client viene avviato eseguirà le seguenti operazioni:

- Scaricherà dal topic JMS tutte le notifiche del Server che devono essere ancora processate dal Client,
- Prenderà tutte gli eventi del *Watcher* che devono essere ancora spediti al Server,
- Eseguirà un parsing delle due liste di eventi verificando se sono presenti dei conflitti, ossia eventi di modifica, successi lato Client, di file che sono stati modificati anche a lato Server mentre il Client era spento.

In caso di conflitto (fig. 4.7), la scelta sul da farsi viene data all'utente, il quale può:

- dare priorità alla modifica lato Server: in questo caso la versione presente nel Client viene caricata al Server come versione obsoleta del file e viene scaricate la versione del Server in locale;
- dare priorità alla modifica lato Client: in questo modo viene considerata come ultima versione quella presente in locale e quindi viene aggiornato il file nel Server eseguendo un upload.

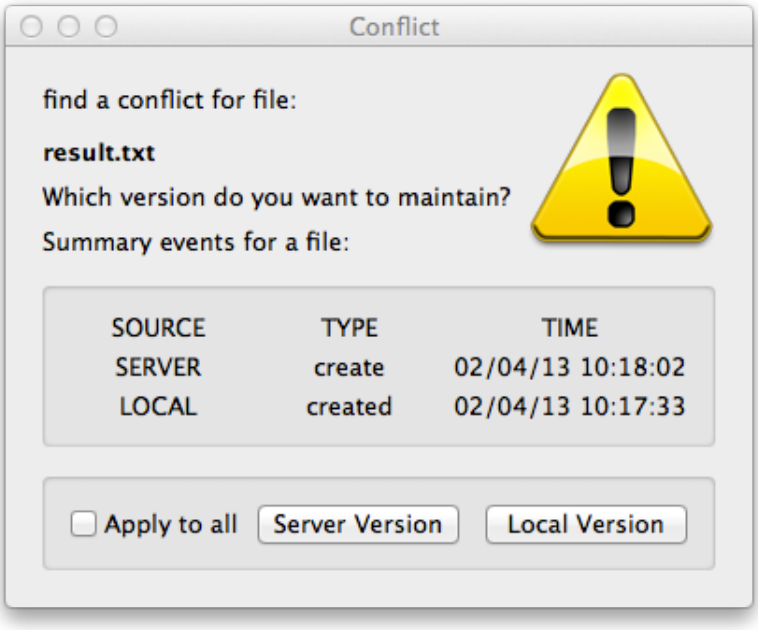

Figura 4.7: *Finestra di conflitto sincronizzazione per un file*

Capitolo 5

Sviluppo del Client

Dopo aver studiato le varie tecniche implementative per l'identificazione univoca dei file, i vari protocolli, presenti nella letteratura, per condivisione di una risorsa in una rete locale, dopo aver progettato la sua implementazione e la gestione delle notifiche per il nostro progetto, abbiamo incominciato lo sviluppo e la programmazione del nostro Client.
Abbiamo deciso di utilizzare il linguaggio **Python**[9] per la sua estrema portabilità su moltissime piattaforme, con ottime performance. Esso ha una caratteristisca di scripting come Perl e la potenza di una linguaggio ad alto livello OOP (Object Oriented Protocol) come Java. Con queste particolarità risulta ideale per creare applicazioni con un alto livello di complessità e multi-piattaforma.

5.1 Struttura

La figura (5.1) rappresenta schematicamente i principali protagonisti del Client. Più nel dettaglio, dall'basso verso l'alto abbiamo:

- *File Database*: il thread che gestisce tutte le query effettuate al database locale in cui sono presenti tutte le informazioni di file e cartelle, informazioni riguardante le condivisioni, ecc...
- *User Database*: il thread che gestisce il database creato il RAM: in questo database vengono salvate le informazioni relative alla sola sessione di avvio del Client; come ad esempio i client connessi nella rete locale, i messaggi REQUEST e RESPONSE inviati.
- *JMS Client*: il Client JMS, colui che riceve le notifiche JMS spiegate nel dettaglio dal capitolo 4.3.
- *File Watcher*: Riceve gli eventi della cartella DriveFarm dei file notificati dal processo *stand-alone Watcher*.
- *Lan Sync*: spiegato nel dettaglio nel prossimo capitolo (cap 5.2).

- *Server Sync*: Thread che gestisce, tramite *ThreadPool*, la sincronizzazione di file con il Server utillizzando le API di DriveFarm.
- *Download Manager*: Thread che gestisce tutto il ciclo di download di file, dalle notifiche ricevuto dal JMS Client, alle richieste spedite in LAN e eventualmente al download del file direttamente dal Server.
- *Upload Manager*: fulcro centrale del Client; processa tutti gli eventi ricevuti dal *File Watcher* e tutti gli update da inviare al Server.
- *GUI*: interfaccia grafica del Client.

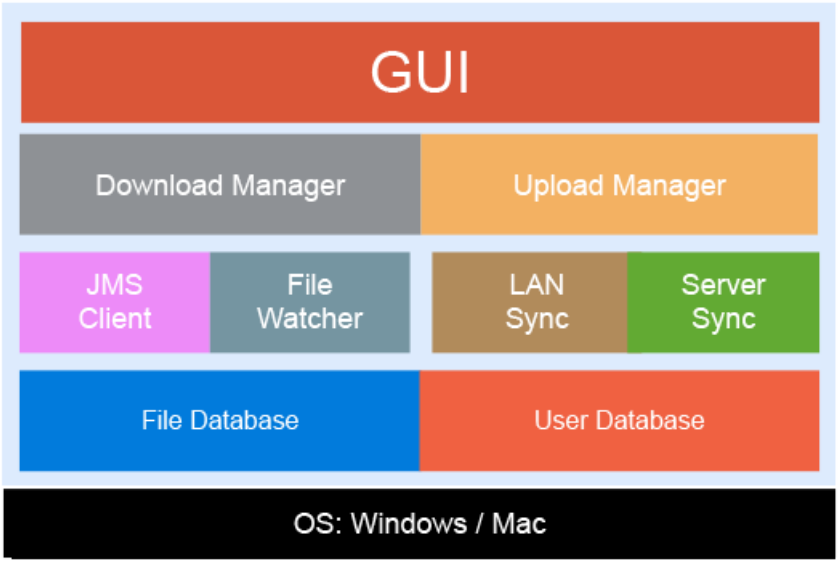

Figura 5.1: *Struttura dell'intero Client DriveFarm*

5.2 Implementazione dell'infrastuttura di rete per la comunicazione LAN

La comunicazione LAN viene quindi gestita da due server:

- un **server TCP**: questo Server da la possibilità al client di ricevere il messaggio di richiesta dei file secondo il protocollo specificato il precendenza. Riceverà quindi i messagi di tipo *REQUEST, RESPONSE, SENDME, FILE*, i messaggi **IMALIVE** e anche **NOTIFY** e **NOTIFYTOME**
- un server UDP in ascolto all'indirizzo di broadcast della rete: questo server ha il compito solamente di ricevere i messaggi *IMALIVE* spediti dai client appena connessi alla rete e,

per ogni messaggio ricevuto, risponderà al server TCP del mittente (quindi in unicast) con un messaggio IMALIVE.

Il server TCP comunica medianti code con i rispettivi due thread che gestiscono i due database del Client. Il primo database **Database DriveFarm** è salvato in locale e viene utilizzato per salvare tutte le informazioni relative ai file e le cartelle presenti in quella di DriveFarm, mentre il secondo, **Database RAM** viene salvato solamente in RAM e viene utilizzato per le informazioni utili solamente nella esecuzione del Client: lista dei vari utenti connessi alla rete LAN e i vari messaggi di request e response inviati. Avrà quindi tre tabelle non relazionali: *Address, Request e ResponseSent*.

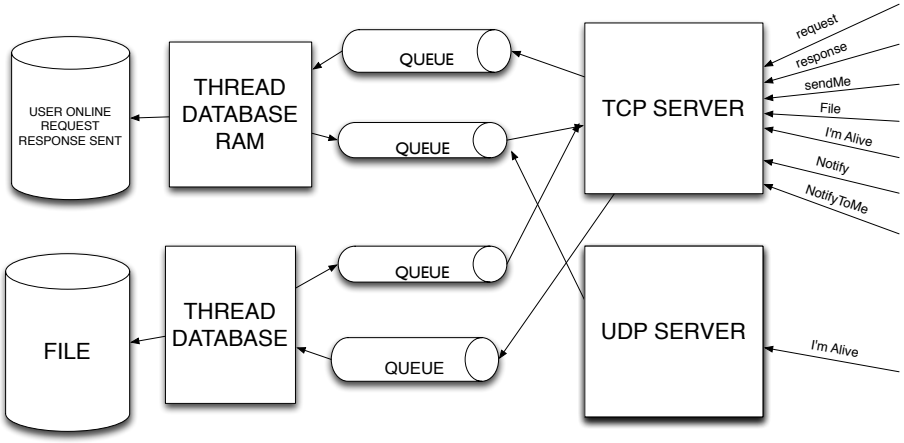

Figura 5.2: *Struttura dell'interfaccia LAN*

5.3 Interfaccia GUI

L'interfaccia GUI, per lo scopo di questo software, presenta solo una minima parte del progetto in quanto le varie funzionalità di esso lavorano in background senza alcuna interazione con l'utente. La parte grafica del progetto propone solamente all'utente di settare alcune impostazioni e ad associare il Client al suo username precedentemente registrato su *drivefarm.com* (al primo avvio del programma).
Mostreremo quindi, le varie finestre presenti nella sezione *Preferenze* facilmente raggiungibile dal menu trayBar (fig 5.3).

5.3.1 Pannello Generale

In questa sezione è possibile abilitare e disabilitare varie funzioni del Client:

- abilitare le notifiche popup; se abilitata, ogni modifica di ogni file nella cartella DriveFarm verrà notificato con una finestra pop-up.

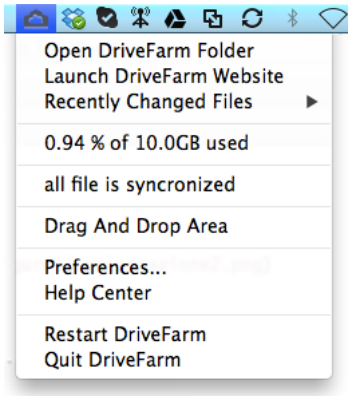

Figura 5.3: *TrayBar Menu*

- abilitare la sincronizzazione in LAN; se abilitata verrà sfruttata la sincronizzazione all'interno della rete locale diminuendo il tempo di download dei file rispetto al download dal Server. Sostanzialmente, abilita e disabilita i server TCP e UDP per la comunicazione in LAN con gi altri Client DriveFarm.

- avvio del Client all'avvio; se abilitata l'applicativo verrà aperto all'avvio del sistema operativo. (È da sottilineare il fatto che il processo *Watcher* deve essere abilitato **SEMPRE** all'avvio del sistema operativo in quanto deve monitorare sempre le varie modifiche effettuate nella cartella DriveFarm).

5.3.2 Pannello Utente

Nel pannello Utente è possibile vedere le informazioni relative all'utente registrato nell'applicativo. Inoltre è possibile scollegare l'utente dal Client, in questo modo il Client non sarà più associato all'username precedentemente registrato, ogni file presente nella cartella DriveFarm locale sarà rimosso e al riavvio del Client verrà mostrato la fase di registrazione dell'utente.

5.3.3 Pannello Network

Nel pannello Network (fig. 5.6) è possibile limitare la banda di Upload e Download da e verso la rete esterna al fine di non occupare totalmente la connessione verso l'esterno della rete locale. Il limite di banda viene effettuato secondo la tecnica del *Token Bucket*.
Bisogna sottolineare però che le connessioni nella rete locale tra i Client (se sono presenti) non possono essere limitate in *runtime* dall'utente ma sono settate con un limite di banda fisso che garantisce la non saturazione della rete, tenendo allo stesso tempo una velocità di download e upload nettamente superiore rispetto a quella Client-Server.

5.3.4 Panello Avanzate

In questa sezione del pannello Preferenze (fig. 5.7) è possibile cambiare la locazione della cartella DriveFarm o il linguaggio dell'applicativo. Qualora venisse cambiata una delle due impostazioni il Client verrà riavviato, e qualora fosse cabiamenta la locazione, verrà anche riavviato anche il processo *Watcher* indicandogli la nuova locazione da "osservare".

5.3.5 Pannello Folder

In questa finestra (fig. 5.8) è possibile aggiungere una o più cartelle a quella di DriveFarm mantenendo traccia del percorso originario della cartella. Infatti nel percorso originario verrà creato un link alla cartella appena spostata. L'utente, quando vuole, può riporla nel percorso precendente togliendo la spunta alla riga che corrisponde la cartella e cliccando su "aggiorna". In questo modo risulta più semplice e intuitivo salvare in un ambiente Cloud una cartella pur mantendo il percorso originario della stessa, senza compromettere quindi l'orginazzazione delle cartelle dell'utente.

5.3.6 Pannello Sync

In questa finestra (fig. 5.9) è possibile selezionare quale sottocartella di DriveFarm voler sincronizzare e quale mantenerla non sincronizzata con il profilo DriveFarm. Questa opzione risulta molto utile quando l'utente non vuole più effettuare download in locale di documenti presenti in una cartella condivisa che viene continuamente aggiornata. Nel caso in cui l'utente abiliterà nuovamente la sincronizzazione di una cartella precedentemene esclusa, verrà allineata con quella di DriveFarm, eventualemente eseguendo anche upload di file qualora fossero stati aggiornati in locale.

5.3.7 Registrazione

Al primo avvio del Client, viene richiesto all'utente le credenziali registrate sul sito **drive-farm.com**. Se esatte, il Client, tramite le API, chiede al Server un nuovo *accessToken* associato al nome utente specificato. L'*accessToken* sarà la chiave d'accesso del Client per le future chiamate dell'API e per la ricezione delle notifiche JMS.

Una volta effettuato l'accesso, viene chiesto all'utente di specificare la locazione della cartella DriveFarm (fig. 5.12), Successivamente verranno scaricati tutti i file dell'utente già presenti nel suo profilo, in modo da aver già sincronizzata con il Server la cartella.
L'ultimo step (fig. 5.13) rappresenta un sommario delle impostazioni specificate dall'utente, il quale può controllare se siano esatte e al più tornare indietro per correggerle.

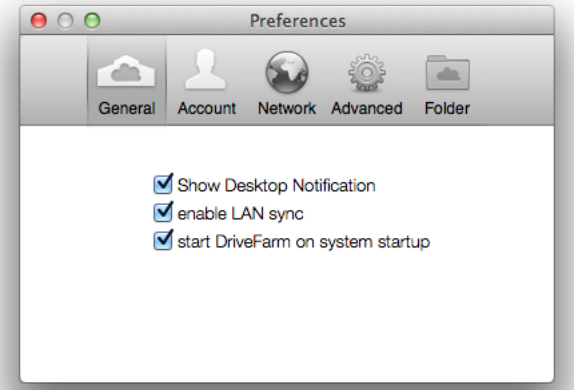

Figura 5.4: *Finestra generale*

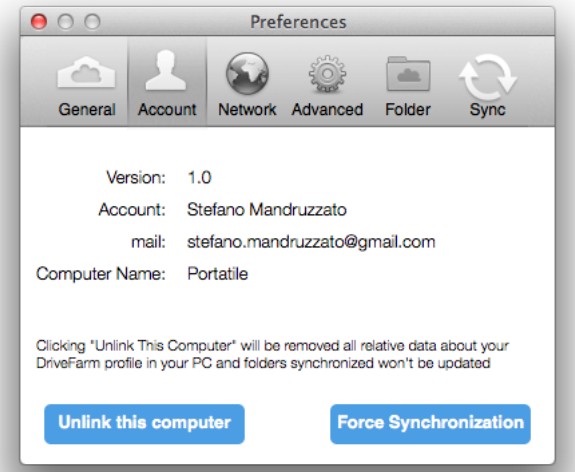

Figura 5.5: *Finestra account*

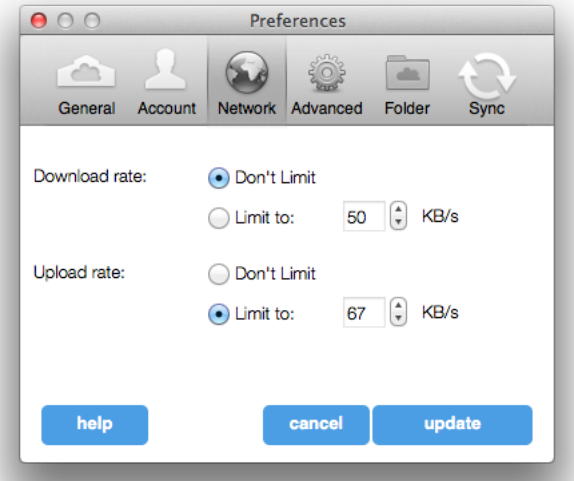

Figura 5.6: *Finestra network*

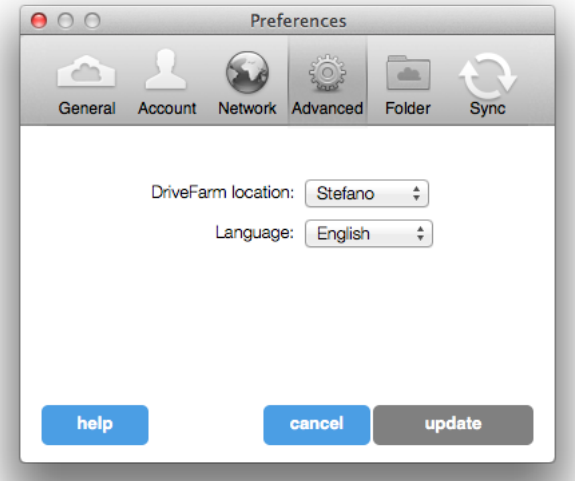

Figura 5.7: *Finestra avanzate*

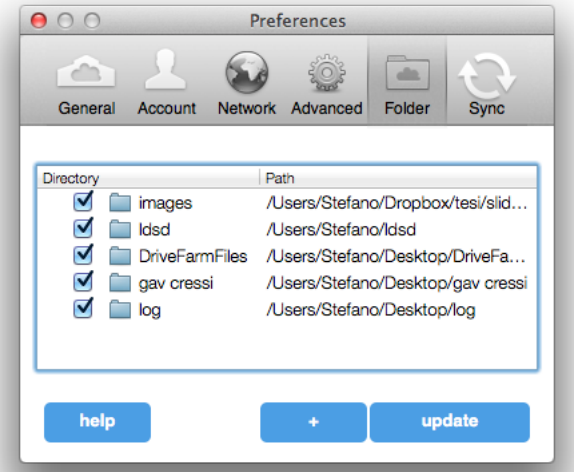

Figura 5.8: *Finestra folder*

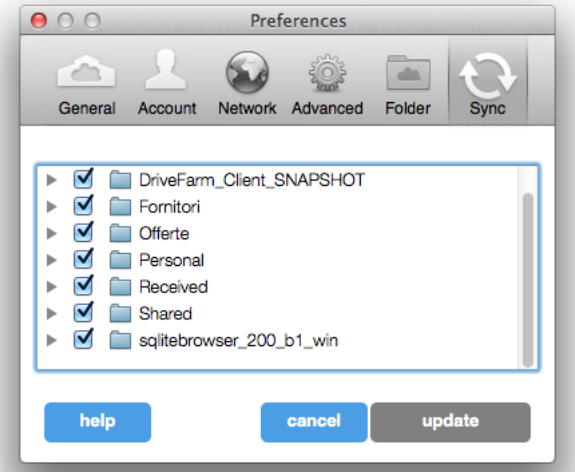

Figura 5.9: *Finestra Sync*

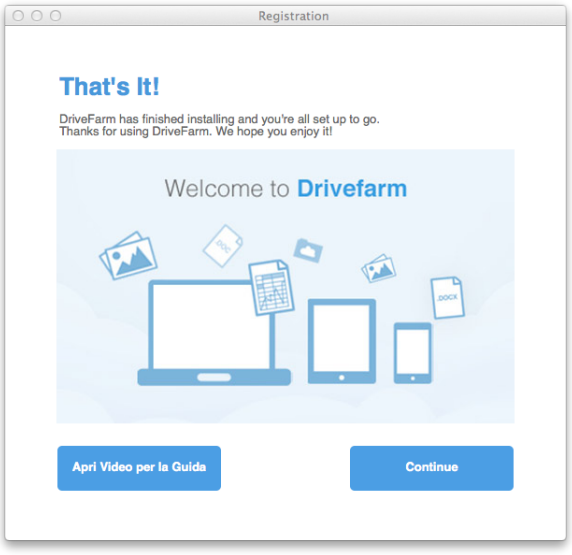

Figura 5.10: *Registrazione - primo step*

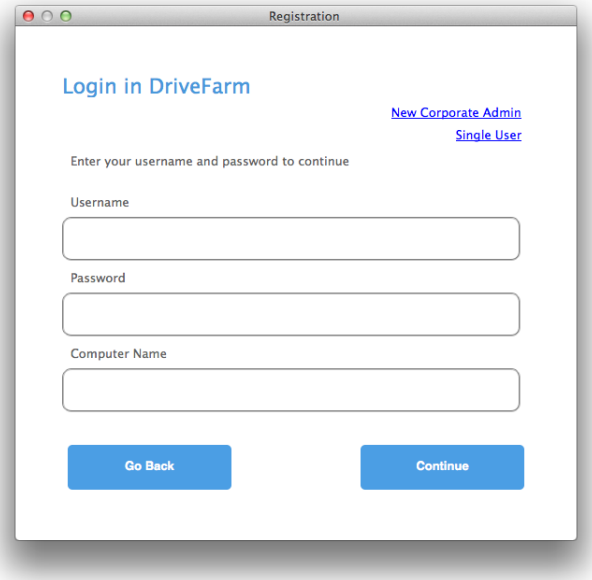

Figura 5.11: *Registrazione - secondo step*

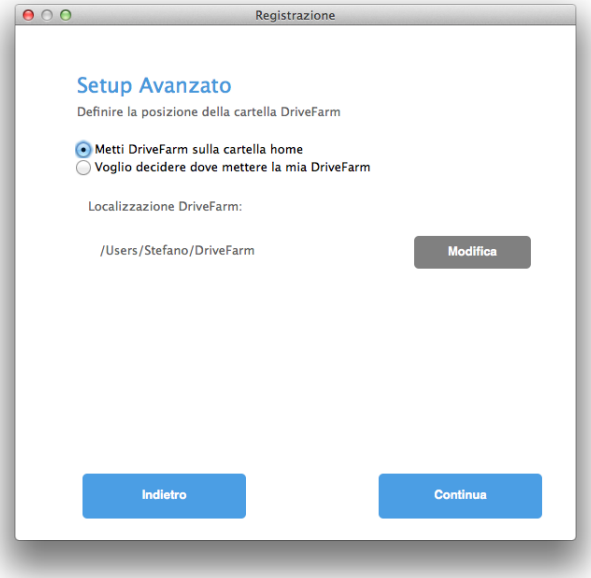

Figura 5.12: *Registrazione - terzo step*

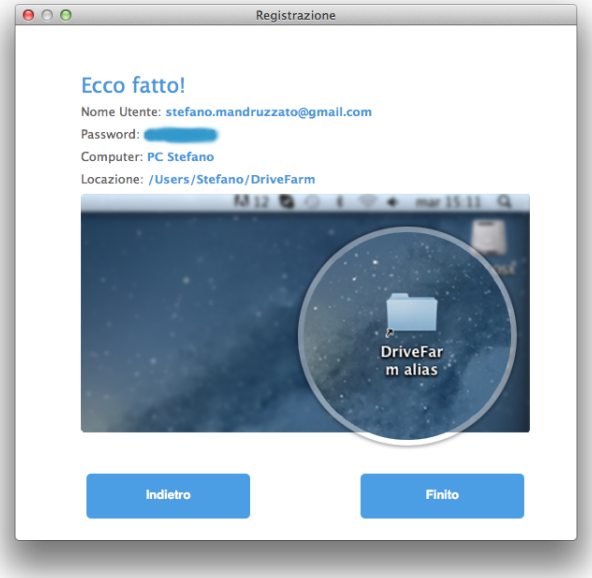

Figura 5.13: *Registrazione - quarto step*

Appendici

Appendice A

algoritmo SHA 2

ciclo	K	ciclo	K	ciclo	K	ciclo	K
1	0x428a2f98d728ae22	2	0x7137449123ef65cd	3	0xb5c0fbcfec4d3b2f	4	0xe9b5dba58189dbbc
5	0x3956c25bf348b538	6	0x59f111f1b605d019	7	0x923f82a4af194f9b	8	0xab1c5ed5da6d8118
9	0xd807aa98a3030242	10	0x12835b0145706fbe	11	0x243185be4ee4b28c	12	0x550c7dc3d5ffb4e2
13	0x72be5d74f27b896f	14	0x80deb1fe3b1696b1	15	0x9bdc06a725c71235	16	0xc19bf174cf692694
17	0xe49b69c19ef14ad2	18	0xefbe4786384f25e3	19	0x0fc19dc68b8cd5b5	20	0x240ca1cc77ac9c65
21	0x2de92c6f592b0275	22	0x4a7484aa6ea6e483	23	0x5cb0a9dcbd41fbd4	24	0x76f988da831153b5
25	0x983e5152ee66dfab	26	0xa831c66d2db43210	27	0xb00327c898fb213f	28	0xbf597fc7beef0ee4
29	0xc6e00bf33da88fc2	30	0xd5a79147930aa725	31	0x06ca6351e003826f	32	0x142929670a0e6e70
33	0x27b70a8546d22ffc	34	0x2e1b21385c26c926	35	0x4d2c6dfc5ac42aed	36	0x53380d139d95b3df
37	0x650a73548baf63de	38	0x766a0abb3c77b2a8	39	0x81c2c92e47edaee6	40	0x92722c851482353b
41	0xa2bfe8a14cf10364	42	0xa81a664bbc423001	43	0xc24b8b70d0f89791	44	0xc76c51a30654be30
45	0xd192e819d6ef5218	46	0xd69906245565a910	49	0xf40e35855771202a	48	0x106aa07032bbd1b8
49	0x19a4c116b8d2d0c8	50	0x1e376c085141ab53	51	0x2748774cdf8eeb99	52	0x34b0bcb5e19b48a8
53	0x391c0cb3c5c95a63	54	0x4ed8aa4ae3418acb	55	0x5b9cca4f7763e373	56	0x682e6ff3d6b2b8a3
57	0x748f82ee5defb2fc	58	0x78a5636f43172f60	59	0x84c87814a1f0ab72	60	0x8cc702081a6439ec
61	0x90befffa23631e28	62	0xa4506cebde82bde9	63	0xbef9a3f7b2c67915	64	0xc67178f2e372532b
65	0xca273eceea26619c	66	0xd186b8c721c0c207	67	0xeada7dd6cde0eb1e	68	0xf57d4f7fee6ed178
69	0x06f067aa72176fba	70	0x0a637dc5a2c898a6	71	0x113f9804bef90dae	72	0x1b710b35131c471b
73	0x28db77f523047d84	74	0x32caab7b40c72493	75	0x3c9ebe0a15c9bebc	76	0x431d67c49c100d4c
77	0x4cc5d4becb3e42b6	78	0x597f299cfc657e2a	79	0x5fcb6fab3ad6faec	80	0x6c44198c4a475817

Tabella A.1: *valori di K durante i vari cicli dell'algoritmo SHA*

Listing A.1: *pesudocodice dell'algoritmo SHA-2 a 512 bit*

```
1
3  Note 1: All variables are unsigned 32 bits and wrap modulo 232 when calculating
   Note 2: All constants in this pseudo code are in big endian
5
   \\Initialize variables
7  \\(first 32 bits of the fractional parts of the square roots of the first 8 primes):
   h[0..7] :=
9     0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a,
      0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19
11
   \\Initialize table of round constants
13 \\(first 32 bits of the fractional parts of the cube roots of the first 80 primes):
   k[0..80] :=
15 Pre-processing:
   append the bit '1' to the message
17 append k bits '0', where k is the minimum number >= 0 such that the resulting message
       length (in bits) is 448 (modulo 512).
19 append length of message (before pre-processing), in bits, as 64-bit big-endian integer

21 Process the message in successive 512-bit chunks:
   break message into 1024-bit chunks
23 for each chunk
       break chunk into sixteen 32-bit big-endian words w[0..15]
25
       Extend the sixteen 32-bit words into sixty-four 32-bit words:
27     for i from 16 to 63
           s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshift 3)
29         s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightshift 10)
           w[i] := w[i-16] + s0 + w[i-7] + s1
31
       Initialize hash value for this chunk:
33     a := h0
       b := h1
35     c := h2
       d := h3
37     e := h4
       f := h5
39     g := h6
       h := h7
41
       Main loop:
43     for i from 0 to 79
           S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
45         ch := (e and f) xor ((not e) and g)
           temp := h + S1 + ch + k[i] + w[i]
47         d := d + temp;
           S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
49         maj := (a and (b xor c)) xor (b and c)
           temp := temp + S0 + maj
51
           h := g
53         g := f
           f := e
55         e := d
           d := c
57         c := b
           b := a
59         a := temp
```

```
    Add this hash of chunk to result so far:
    h0 := h0 + a
    h1 := h1 + b
    h2 := h2 + c
    h3 := h3 + d
    h4 := h4 + e
    h5 := h5 + f
    h6 := h6 + g
    h7 := h7 + h

Produce the final hash value (big-endian):
digest := hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
```

Appendice B

API

Login

Path : /login **Metodo** :GET

Parametri

- accessToken

Risposta

```
{ "login" :
    { "name" : "Nome",
      "surname" : "Cognome",
      "email" : "E-mail",
      "image" : "Stringa codificata in Base64",
      "homeUser" : true,
      "externalShareEnabled" : true,
      "mobileAppUseEnabled" : true
    }
}
```

Folder Content

Path : /folderContent **Metodo**: GET

Parametri

- accessToken
- path: il percorso assoluto della cartella
- deep (opzionale): booleano che indica se l'elenco dei file deve essere profondo. Se non specificato viene considerato "false".

Risposta

```
{ "files" : [
    { "path" : "/path/to/file",
      "directory" : false,
      "size" : Dimensione in byte (0 per le cartelle),
      "readOnly" : false,
      "lastModifiedDate" : timestamp in millisecondi (0 per le cartelle)
    }]
}
```

Create Folder

Path :/createFolder **Metodo** :POST

Parametri

Mime type: application/json

```
{ "accessToken" : "accessToken",
  "path" : "percorso assoluto della cartella da creare"
}
```

Risposta

Array di 1 elemento con le informazioni sulla cartella appena creata.

```
{ "files" : [
    { "path" : "/path/to/created/folder",
      "directory" : true,
      "size" : 0,
```

```
      "readOnly" : false,
      "lastModifiedDate" : timestamp in millisecondi
    }]
}
```

File Upload

Path : /fileUpload **Metodo** : POST

Parametri

Va inviato un contenuto di tipo "multipart/form-data" contenente 2 parti:

La prima parte chiamata **fileUpload** deve contenere una stringa JSON con i seguenti parametri:

```
{ "accessToken" : "accessToken",
  "size" : dimensione del file in bytes,
  "path" : percorso assoluto del file da caricare
}
```

La seconda parte chiamata **uploadedFile** deve contenere i dati effettivi da caricare.

Risposta

Array di 1 elemento con le informazioni sul file appena caricato.

```
{ "files" : [
    { "path" : "/path/to/uploaded/file",
      "directory" : false,
      "size" : dimensione del file in byte,
      "readOnly" : false,
      "lastModifiedDate" : timestamp in millisecondi
    }]
}
```

File Content

Path : /fileContent **Metodo** : GET

Parametri

- accessToken
- path: il percorso assoluto del file da scaricare

Risposta

Flusso di tipo **application/octet-stream** del contenuto del file.

Remove Files

Path : /removeFiles **Metodo** : POST

Parametri

Mime type: application/json

```
{ "accessToken" : "accessToken",
  "paths" : [
```

```
  ... array di percorsi assoluti da cancellare
  ]
}
```

Risposta

In caso non si verifichino errori:

```
{ "response" : "OK" }
```

Remove Folders

Path : /removeFolders **Metodo** : POST

Parametri

Mime type: application/json

```
{ "accessToken" : "accessToken",
  "paths" : [
  ... array di percorsi assoluti da cancellare
  ]
}
```

Risposta

In caso non si verifichino errori:

```
{ "response" : "OK" }
```

Empty Trash

Path : /emptyTrash **Metodo** : POST

Parametri

```
{ "accessToken" : "accessToken" }
```

Risposta

In caso non si verifichino errori:

```
{ "response" : "OK" }
```

Rename File

Path : /renameFile **Metodo** : POST

Parametri

```
{ "accessToken" : "accessToken,
  "path" : percorso assoluto del file da rinominare,
  "newName" : nuovo nome
}
```

Risposta

In caso non si verifichino errori:

```
{ "response" : "OK" }
```

Rename Folder

Path : /renameFolder **Metodo** : POST

Parametri

```
{ "accessToken" : "accessToken,
  "path" : percorso assoluto della cartella da rinominare,
  "newName" : nuovo nome
}
```

Risposta

In caso non si verifichino errori:

```
{ "response" : "OK" }
```

User Stats

Path : /userStats **Metodo** : GET

Parametri

- accessToken

Risposta

```
{ "userStats" :
    { "name" : "Nome",
      "surname" : "Cognome",
      "email" : "E-mail",
      "image" : "Stringa codificata in Base64",
      "homeUser" : true,
      "externalShareEnabled" : true,
      "mobileAppUseEnabled" : true,
      "purchasedSpace" : spazio acquistato in byte,
      "usedSpace" : spazio utilizzato in byte,
      "versioningEnabled" : booleano che indica se l'account dell'utente ha il versionamento abilitato
    }
}
```

Copy File

Path : /copyFile **Method** : POST

Parametri

```
{ "accessToken" : "...",
  "sourcePath" : percorso assoluto del file di partenza,
  "destinationPath" : percorso assoluto del file di destinatione
}
```

Risposta

In caso non si verifichino errori:

```
{ "response" : "OK" }
```

Copy Folder

Path : /copyFolder **Method** : POST

Parametri

```
{ "accessToken" : "...",
  "sourcePath" : percorso assoluto della cartella di partenza,
  "destinationPath" : percorso assoluto della cartella di destinatione
}
```

Risposta

In caso non si verifichino errori:

```
{ "response" : "OK" }
```

Move File

Path : /moveFile **Method** : POST

Parametri

```
{ "accessToken" : "...",
  "sourcePath" : percorso assoluto del file di partenza,
  "destinationPath" : percorso assoluto del file di destinatione
}
```

Risposta

In caso non si verifichino errori:

```
{ "response" : "OK" }
```

Move Folder

Path : /moveFolder **Method** : POST

Parametri

```
{ "accessToken" : "...",
  "sourcePath" : percorso assoluto della cartella di partenza,
  "destinationPath" : percorso assoluto della cartella di destinatione
}
```

Risposta

In caso non si verifichino errori:

```
{ "response" : "OK" }
```

Bibliografia

[1] E. Barker, W. Barker, W. B. andWilliam Polk, and M. Smid. (2007) Recommendation for key march, 2007 management – part 1: General (revised). [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57-Part1-revised2_Mar08-2007.pdf

[2] N. S. Agency. (2009, november) Nsa suite b cryptography. [Online]. Available: http://www.nsa.gov/ia/programs/suiteb_cryptography/index.shtml

[3] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche1. Keccak website. [Online]. Available: http://keccak.noekeon.org

[4] S. jen Chang, R. Perlner, W. E. Burr, M. S. Turan, J. M. Kelsey, S. Paul, and L. E. Bassham. (2012, november) Third-round report of the sha-3 cryptographic hash algorithm competition. [Online]. Available: http://nvlpubs.nist.gov/nistpubs/ir/2012/NIST.IR.7896.pdf

[5] Oracle. Java message service (jms). [Online]. Available: http://www.oracle.com/technetwork/java/jms/index.html

[6] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. (2012, April) Javatm message service specification. [Online]. Available: http://download.oracle.com/otndocs/jcp/7195-jms-1.1-fr-spec-oth-JSpec/

[7] JBoss. Hornetq documentation. [Online]. Available: http://docs.jboss.org/hornetq/2.2.14.Final/user-manual/en/html_single/index.html

[8] STOMP. Stomp. [Online]. Available: http://stomp.github.com/

[9] Python. Python website. [Online]. Available: http://www.python.org