Comunicazione Peer-to-Peer su TCP

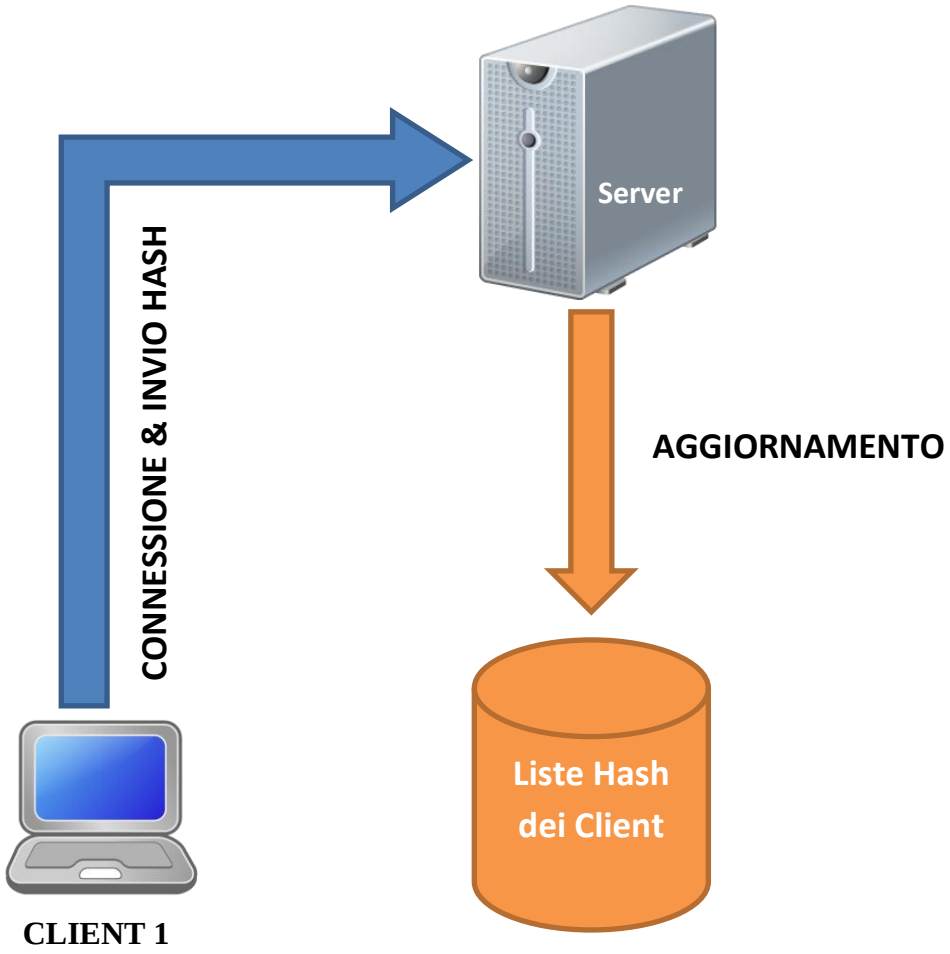
Giacomo Persichini

Laboratorio di Sistemi Operativi ‘11/’12
Linguaggio utilizzato: C

Funzionamento

Il progetto simula il funzionamento di una rete *Peer-to-Peer* che ha lo scopo di rendere disponibili dei file, identificati da un hash univoco, ai partecipanti. I tipi di componenti che fanno parte del sistema sono due: *client* (o peer) e *server*.

Segue una dimostrazione grafica delle interazioni progettate tra client e server.



Un client ha una o più cartelle pubbliche con dei file che vuole **condividere** ed una lista contenente i relativi valori **hash** dei file. Non importa il nome del file o la dimensione, **l'hash è calcolato sul contenuto** ed è **univoco** (es: SHA1, MD5, ecc…).

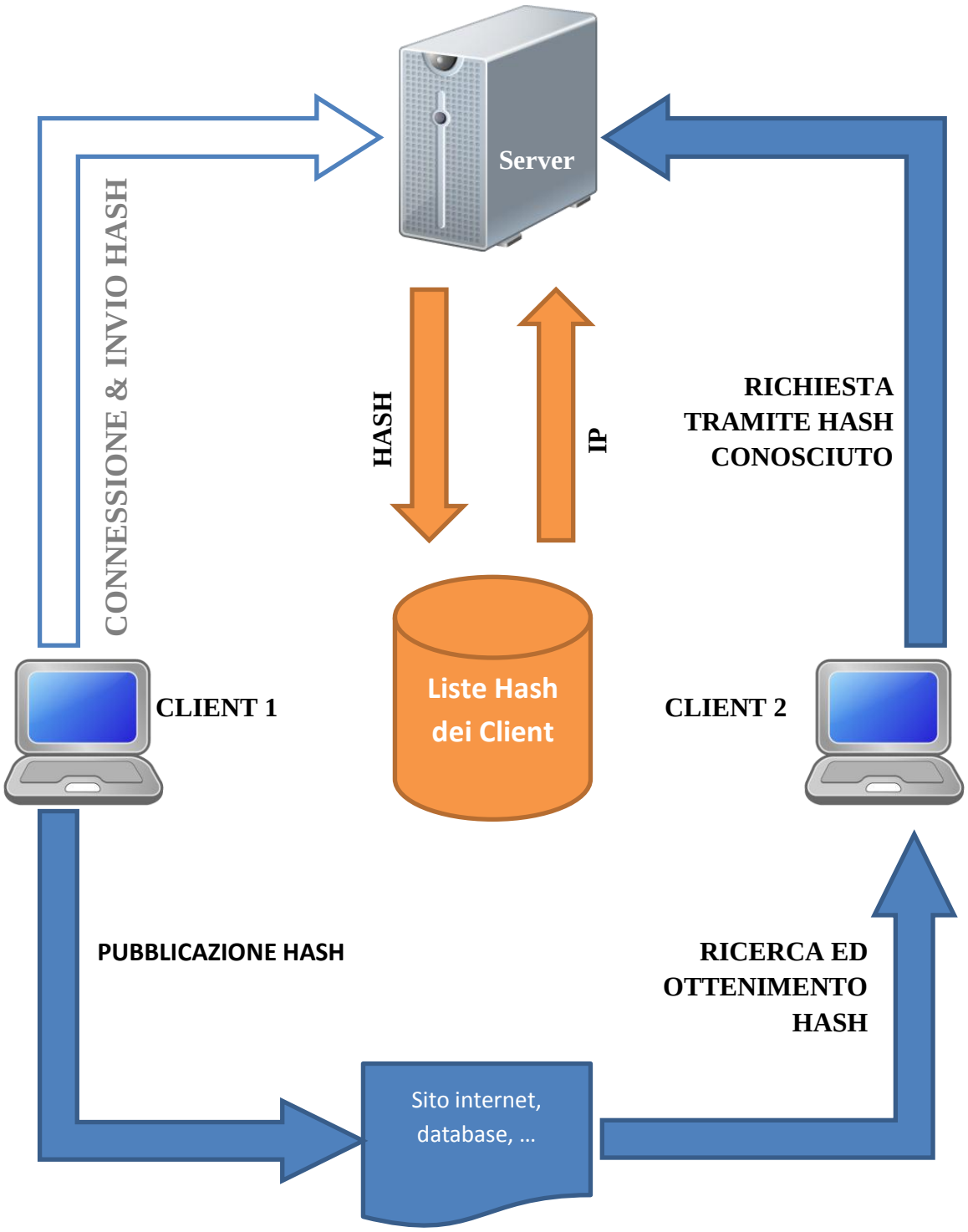
Quando un client si collega al server, invia la sua lista con tutti gli hash dei file che è **disponibile a condividere**. Il server avrà una propria lista con tutti gli hash inviati dai client, associati agli IP dei client stessi. In questo modo conoscendo l'hash di un file è possibile risalire al client che è disponibile a inviarlo, se quest'ultimo è online.

Con un server principale al centro della rete che conosce *chi ha cosa* a questo punto è semplice introdurre il **download** di un file.

È stato necessario risolvere il problema di associazione *file che si vuole scaricare -> hash*, non è semplice sapere che per scaricare il file contenente "gli appunti di Sistemi Operativi" occorre richiedere l'hash "A94A8F5". Per questo motivo la distribuzione degli hash con una descrizione del contenuto a loro associato è fatta su canali esterni a questo sistema, ad esempio su *siti web, database, email, ecc…*

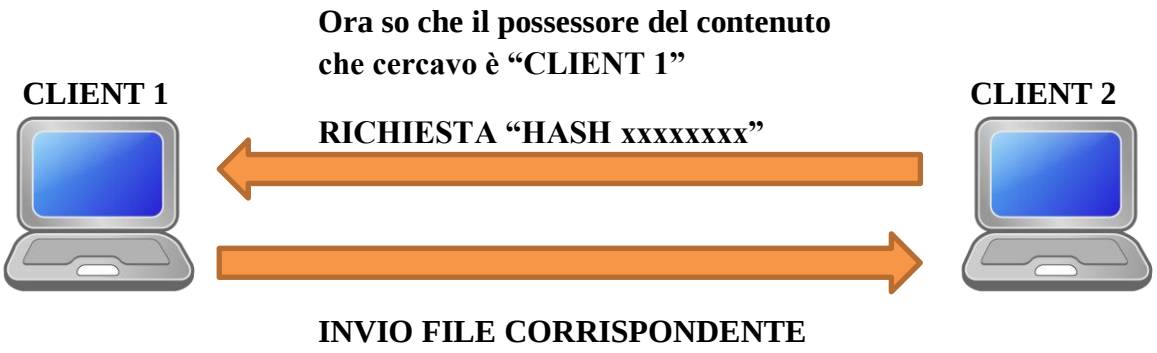
In questo modo il sistema può concentrarsi sul puro scambio dei file, lasciando invece la ricerca, l'organizzazione e la distribuzione degli hash identificativi ad altri sistemi più specializzati. Questa

metodologia è ispirata al funzionamento del protocollo **Bittorrent**, in cui i file *.torrent* che fanno da identificatori di un contenuto sono distribuiti all'esterno del sistema stesso.



Una volta che un client interessato al download di un contenuto trova **l'hash corrispondente** lo comunica al server. Quest'ultimo interroga la propria lista hash che, dato un hash, restituisce l'IP del client che possiede il contenuto condiviso. Il vantaggio degli hash è che il client che condivide una risorsa **non è necessariamente il creatore o il primo condivisore**. Gli hash di un certo contenuto saranno sempre identici (sempre che qualcuno non ne modifichi, appunto, il contenuto).

Una volta che il server ha ricavato l'IP del **client possessore** del contenuto, lo comunica prontamente al **client richiedente** ed ecco che si stabilisce una **connessione diretta tra Client 1 e Client 2** (nell'esempio delle immagini) e il file è trasferito.



A questo punto il server non serve più, il servizio è gestito interamente dai client interessati visto che **si conoscono a vicenda** e termina quando il richiedente ha ricevuto il file dal condivisore. Il server si preoccupa semplicemente di tenere traccia dei client condivisori in tutta la rete per aiutare i richiedenti a trovare chi possiede un certo contenuto.

Gli aspetti principali del Server

Il server è il programma che crea una rete di condivisione e accetta le connessioni dei client in ingresso. È stato strutturato in due thread per consentire all'utente di inviare il comando di chiusura mentre il programma è in un ciclo infinito in attesa di connessioni. Inizialmente era stato progettato con un ciclo infinito al cui interno c'era il controllo in attesa di una connessione e in seguito la funzione **fork()**, ma questa tecnica non permetteva la terminazione del programma in modo corretto: non era possibile uscire dal ciclo in modo pulito visto che, in caso di client non connessi, il programma rimaneva in attesa di una nuova connessione senza poter fare nulla.

La funzione **select()** ha invece un timeout, in questo caso è stato possibile creare una variabile globale condivisa tra i due thread: il thread che controlla l'input dell'utente la imposta ad 1 se si vuole terminare il programma, il thread che controlla le connessioni la legge ogni volta che la select() va in timeout, se il valore è 0 ritorna alla select() in attesa di connessioni, altrimenti comincia la procedura d'uscita.

```
/* Set the timeout to 1 second */
timeout.tv_sec = 1;
timeout.tv_usec = 0;
selectval = select(fdmax+1, &read_fds, NULL, NULL, &timeout);
if (selectval < 0) {
        perror("[ERROR] Listener: select() call failed");
        pthread_exit(NULL);
}
else if (selectval == 0) {
        /* timeout */
        if (quit) {
                break;
        }
        else
                continue;
}
```

Nello spezzone di codice sopra si mostra la gestione della select(): se il valore di ritorno è uguale a 0 vuol dire che è avvenuto il timeout e si può quindi controllare il valore della funzione globale *quit*.

Un altro punto chiave da chiarire è l'utilizzo di una variabile condivisa tra due thread. In questo caso non è stato utilizzato un **mutex** (funzioni *pthread_mutex_**): la lettura di una variabile di tipo intero è, nella maggior parte delle architetture, un'**istruzione atomica**, per questo è stato possibile evitare l'inserimento di un *lock* in fase di scrittura e lettura da parte dei thread.

Il server è programmato per comunicare con i client in un determinato modo, ad esempio la funzione **handshake()** si assicura che chi si sta connettendo è realmente un client che conosce il funzionamento del sistema. La stessa funzione poi sarà anche sul programma client in una versione leggermente differente.

Un aspetto importante dell'accettazione delle connessioni è stato anche il controllo delle connessioni massime. La funzione **listen()** richiede comunque di impostare un limite massimo di connessioni, ma questo numero non garantisce che i nuovi client che cercano di connettersi oltre il limite ricevano un errore.

Altri punti degni di nota sono i controlli sui permessi di file e cartelle nelle funzioni che lavorano sul file system come ad esempio **receive_file()** o **create_config_file()**.

Struttura file e configurazione del Server

La cartella in cui si trova l'eseguibile del server è strutturata come segue:

```
.
..
db/
config
Server
```

Nella cartella *db* il server salva le liste di hash che riceve dai client connessi. Le liste potrebbero essere anche di grandi dimensioni, a seconda di quanti file un client condivide, e potrebbero essere numerose a seconda di quanti client sono connessi in un dato periodo di tempo, per questo è stato preferito spostare tutto dalla memoria centrale al disco.

Il **file di configurazione** del server è riportato di seguito:

```
server-ip=192.168.204.128
server-port=1313
max-connections=50
```

Il primo parametro indica al programma su quale IP mettersi in ascolto, in questo modo è possibile avere più server aperti su una macchina che ha più schede di rete.

Il secondo parametro indica la porta di ascolto.

Il terzo parametro imposta il limite massimo di connessioni che verranno accettate.

Gli aspetti principali del Client

Il client è, come il server, strutturato su due thread che condividono una variabile globale. Un thread è adibito allo user input mentre l'altro è un *"mini-server"* che si mette in ascolto delle connessioni di altri client.

Anche qui le interazioni con il filesystem sono importanti e più numerose rispetto al server, è stato quindi necessario gestire tutti i possibili casi, specialmente nei permessi utente. Ad esempio, la funzione **write_hash_list()** che si occupa di creare gli hash di tutti i file condivisi nelle **shared-folder** (separate da un punto e virgola nel file di configurazione se più di una), prova anche a creare le cartelle condivise se queste dovessero essere inesistenti, mentre le ignora totalmente se risultassero non accessibili per una questione di permessi. Per fare queste distinzioni è stata fondamentale la libreria **errno.h**. Inoltre, il file chiamato "hash" in cui vengono salvati tutti gli hash dei file condivisi, viene creato (se inesistente) con particolari permessi utente specificati grazie alla system call **open()**: il file è accessibile in lettura e scrittura dall'utente che avvia il programma e dal suo relativo gruppo. Pubblicamente, il file è di sola lettura mentre nessuno ha permessi di esecuzione.

L'hashing del contenuto dei file è stato possibile grazie ad una libreria esterna, la **gcrypt.h**, al cui interno ha funzioni specifiche per creare hash data una sequenza di byte. La funzione implementata per semplificare l'hashing è stata chiamata **sha1_hash()**, che utilizza a sua volta **gcry_md_hash_buffer()** di gcrypt.

Un aspetto particolare del client è l'invio di file, implementato tramite la funzione **send_file()**. Questa funzione prima invia la dimensione in byte del file da inviare, poi invia direttamente il contenuto del file, in questo modo il ricevente (il server nel caso dell'invio degli hash oppure un altro client nel caso dell'invio di contenuti richiesti) sa quando lo streaming dati relativo al file finisce e può chiudere il file, dichiarandolo ricevuto. Chi riceve utilizza la funzione **receive_file()**.

Una funzione che può considerarsi "chiave" della parte client è quella che implementa il **download** di un contenuto conoscendone l'hash. La funzione **download_file()** chiede all'utente di inserire l'hash del contenuto richiesto ed il nome del file con cui salvarlo nella cartella dei download una volta ricevuto, successivamente viene inviato un comando particolare al server con in allegato l'hash richiesto. Il server può rispondere con l'IP del possessore in caso positivo, altrimenti può dire che nessun client della rete è disponibile a condividere il contenuto richiesto. In caso di risposta positiva, il client avvia il collegamento con l'IP ricevuto, questo collegamento è gestito dalla funzione **peer_listener()** da chi ascolta. Una volta che il collegamento è stato confermato con un handshake tra i client, il richiedente invia l'hash del contenuto interessato e il possessore del contenuto utilizza la funzione **send_file()** per inviarlo. Una volta completato il trasferimento, il collegamento è chiuso e **peer_listener()** torna in ascolto di un nuovo client da servire.

Anche qui, come nel server, c'è il controllo periodico della variabile condivisa tra i due thread: se l'utente volesse chiudere il programma, è necessario che **peer_listener()** interrompa il proprio ascolto e chiuda i socket per poi terminare il thread.

Il programma termina solo quando entrambi i thread sono chiusi.

Struttura file e configurazione del Client

La cartella in cui si trova l'eseguibile del client è strutturata come segue:

```
.
..
downloads/
config
hash
Peer
```

La cartella *downloads* contiene tutti i file scaricati dalla rete di condivisione, l'utente che avvia il programma deve ovviamente avere i permessi necessari per scrivere i file e leggere il contenuto della cartella.

Il file *hash* è quello che contiene la lista di hash di tutti i file condivisi, questo file viene inviato al server in fase di connessione.

Il **file di configurazione** del client è riportato di seguito:

```
server-ip=192.168.204.128
server-port=1313
shared-folder=/home/user/shared;/home/user/public
```

Il primo parametro specifica l'indirizzo IP del server a cui il client si collegherà.

Il secondo parametro indica ovviamente la porta, deve coincidere con quella di ascolto del server.

Il terzo parametro indica le cartelle contenenti i file da condividere. Queste cartelle possono avere percorsi sia relativi che assoluti e devono essere separate da un punto e virgola.

Aspetti tecnici generali

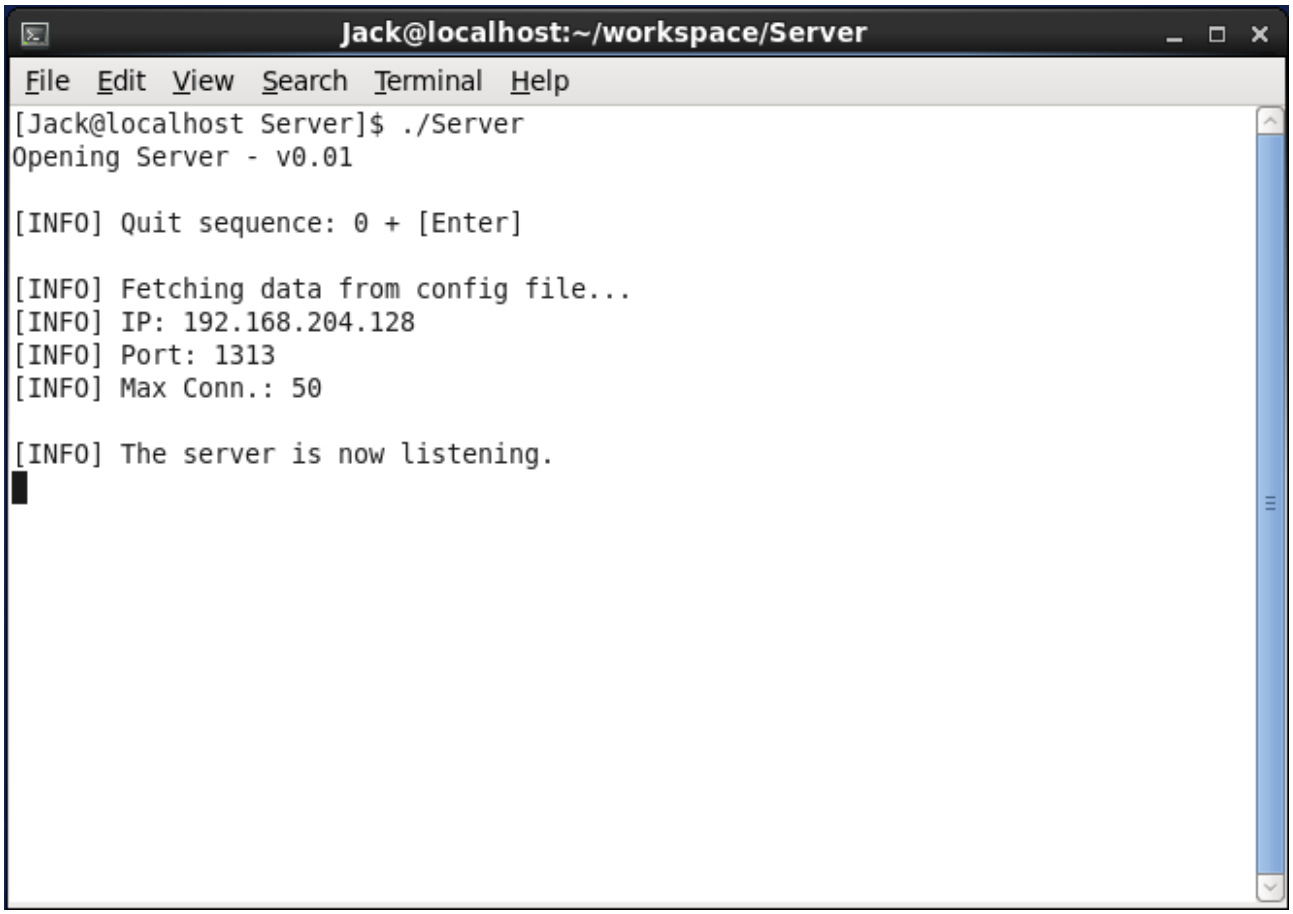
In entrambi i programmi, la variabile globale *"quit"* è stata definita come **volatile**. Questo perché in fase di ottimizzazione (compilazione come Release) il codice è migliorato automaticamente e molte dinamiche introdotte dal programmatore potrebbero venire mal interpretate. Ad esempio, se una variabile dichiarata non volatile è modificata e valutata all'interno di una funzione, la valutazione non sarà più fatta sul contenuto della variabile ma direttamente sul valore cui è impostata, pensando quindi di ottimizzare un codice mal scritto. Se la variabile è modificata da un altro thread, il nuovo valore non sarà mai valutato, per cui è necessario dichiarare la variabile volatile per evitare situazioni di questo tipo.

In alcune funzioni, inoltre, sono anche presenti variabili dichiarate come **register**. Questo di solito avviene con valori utilizzati in un loop (ad esempio il for), una variabile dichiarata in questo modo è salvata in un registro della CPU, ottimizzandone quindi l'accesso. È impossibile utilizzare questa tecnica quando bisogna lavorare con l'*indirizzo* della variabile, in quel caso trovandosi in un registro non è possibile ottenerne l'indirizzo.

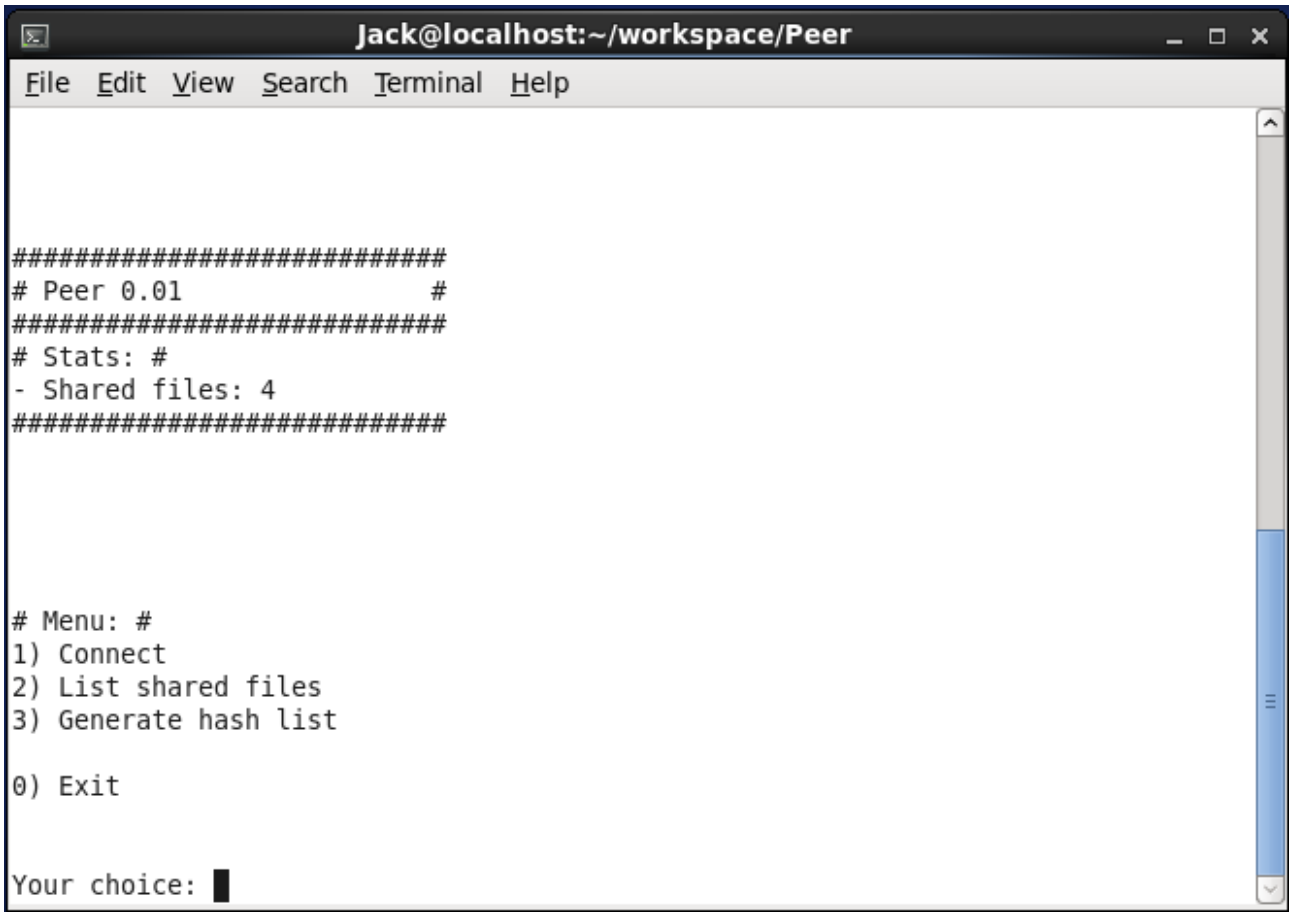
La compilazione finale avviene inserendo anche alcuni valori con il parametro -l, quello che include le librerie. Le librerie da includere necessariamente nel comando di compilazione per il client sono: **gcrypt**, **gpg-error** e **pthread**, mentre basta solamente pthread per il server.

Il progetto funziona solamente con client che hanno IP diversi tra loro, per via dell'aspetto didattico del programma. Un metodo per sopperire a questa mancanza potrebbe essere quello di inserire nell'handshake il passaggio di una *struct*, in cui sono salvati alcuni dati dell'utente e un codice identificativo univoco.

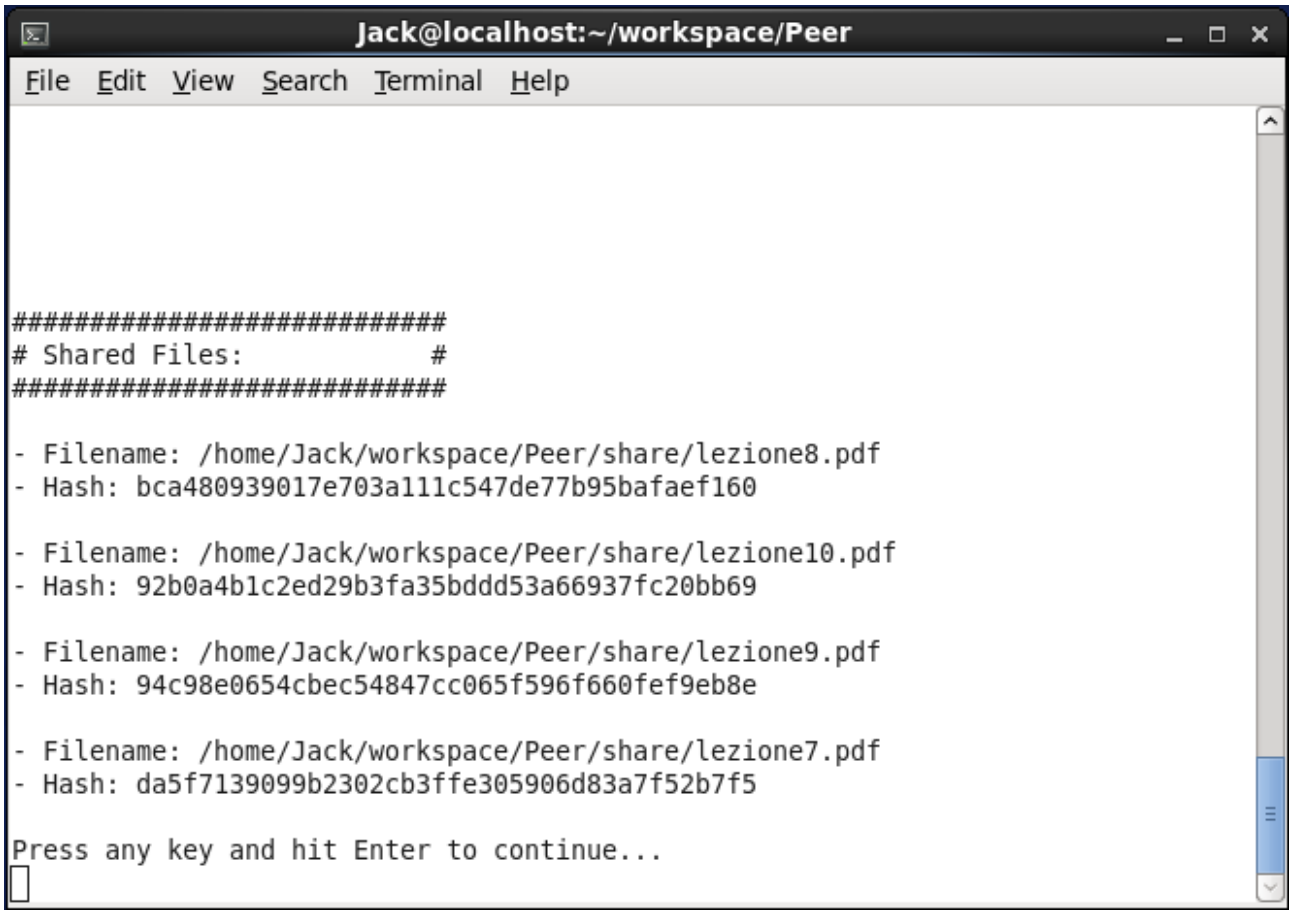
Dimostrazione pratica



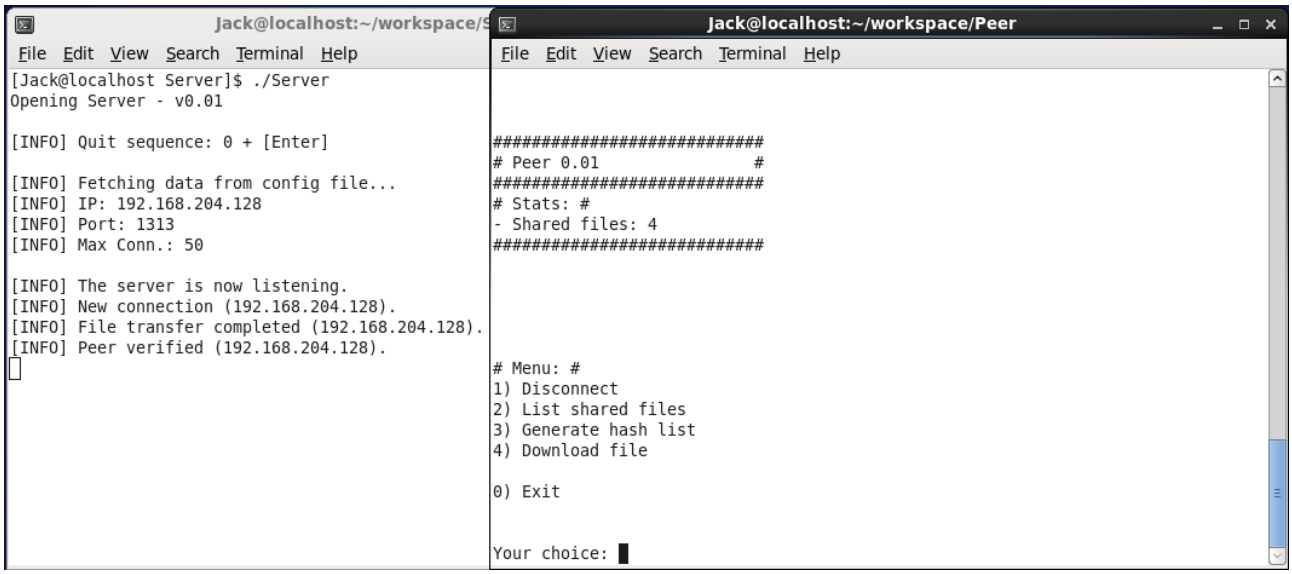
Avvio del server



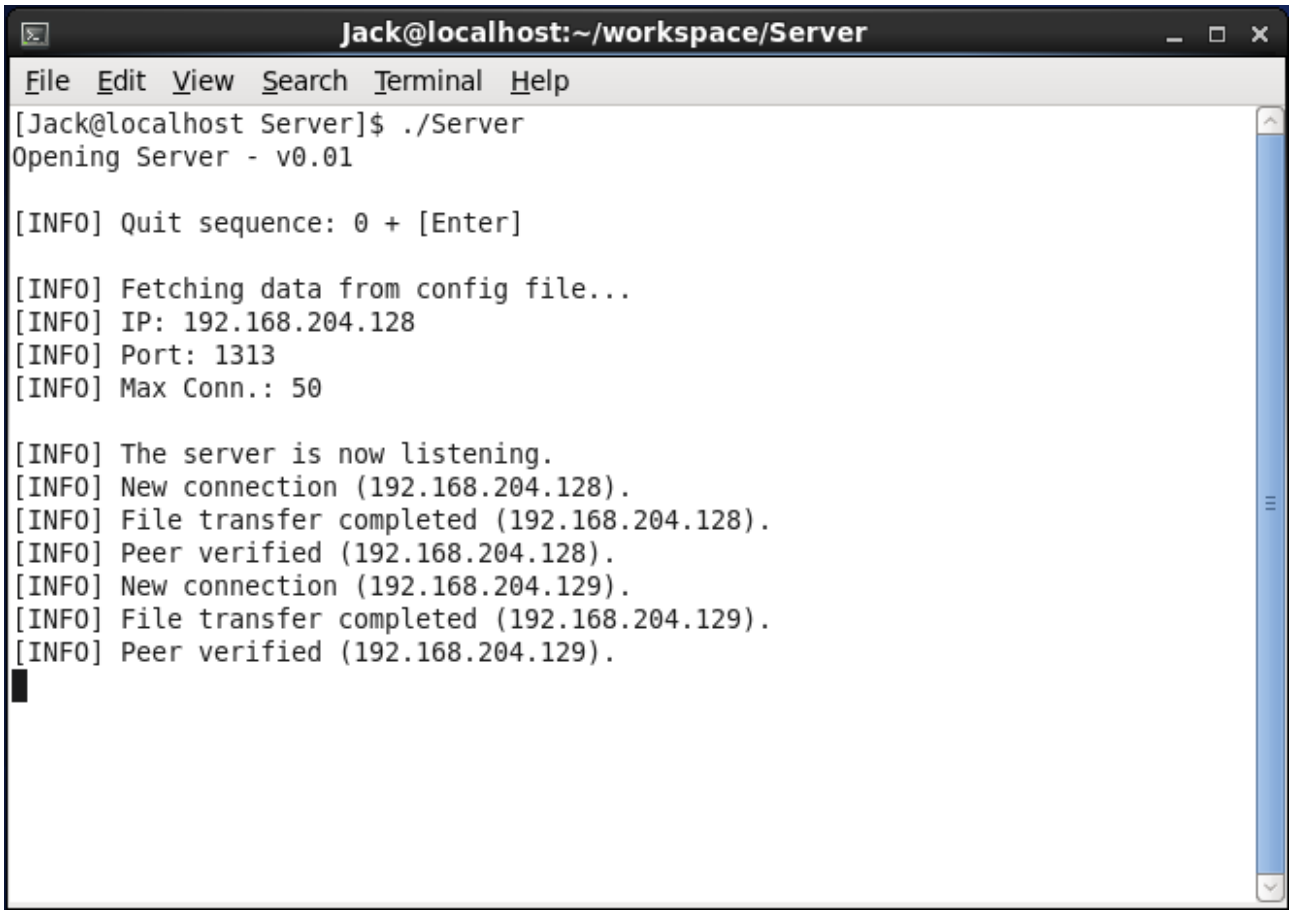
Avvio del client



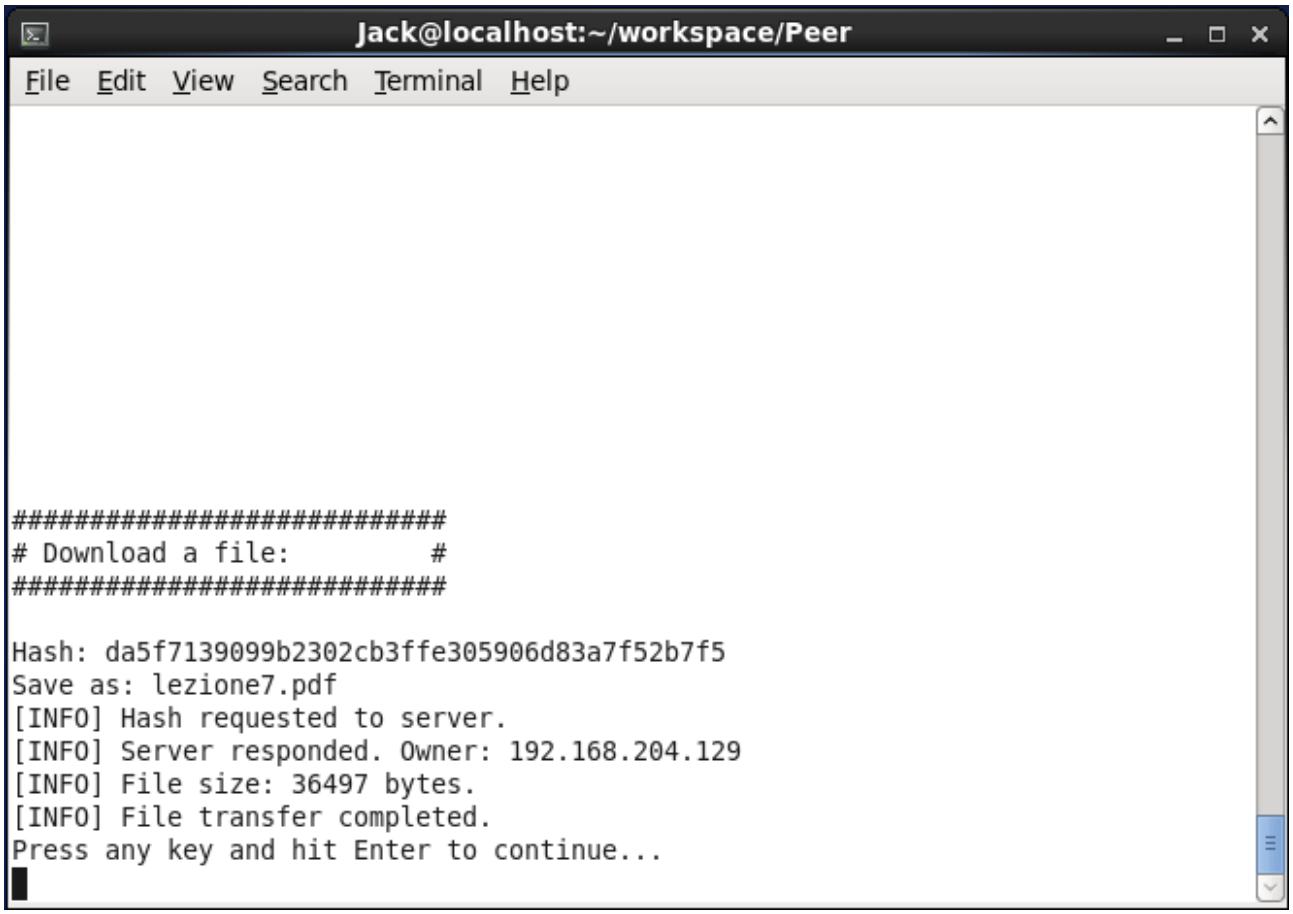
Il client visualizza i propri file condivisi



Il client si collega al server premendo 1 + Invio. Nel server, con "File transfer completed" si riferisce al trasferimento degli hash avvenuto con successo



Un altro client si collega alla rete di condivisione



Il primo client richiede un hash, specifica il nome con cui salvare l'eventuale file ricevuto, e ottiene una risposta positiva dal server. Infine, si collega direttamente al client che condivide il contenuto e riceve il file, che sarà poi salvato nella cartella "downloads"