Relazione Progetto Programmazione di Reti

Traccia 2

Stefano Furi

30 giugno 2022

Indice

Capitolo 1

Analisi

Si è realizzata la traccia numero 2, ovvero la creazione di un'architettura *Client-Server* `UDP` per il trasferimento di file. Devono essere possibili, quindi, lo scambio di due tipi di messaggio: messaggi di *comando* e messaggi di *risposta*. Questi messaggi vengono inviati tramite un opportuno protocollo di trasporto. I messaggi avranno ognuno la medesima struttura definita da un *header* del segmento inviato tramite il socket, affinché sia il client che il server possano inviarsi informazioni dettagliate riguardo l'operazione in corso in modo standard e predefinito. Infine il server invia messaggi di risposta in base alle operazioni richieste dal client per segnalare il successo di queste ultime, oppure il fallimento. In entrambi i casi il client ha il compito di mostrare all'utente il successo/fallimento dell'operazione da esso richiesta. Le funzioni richieste sono:

- **LIST**: *files* contenuti all'interno del server.
- **GET**: scaricare un determinato file dal server.
- **PUT**: *upload* di un file sul server.

Capitolo 2

Desgin

2.1 Panoramica

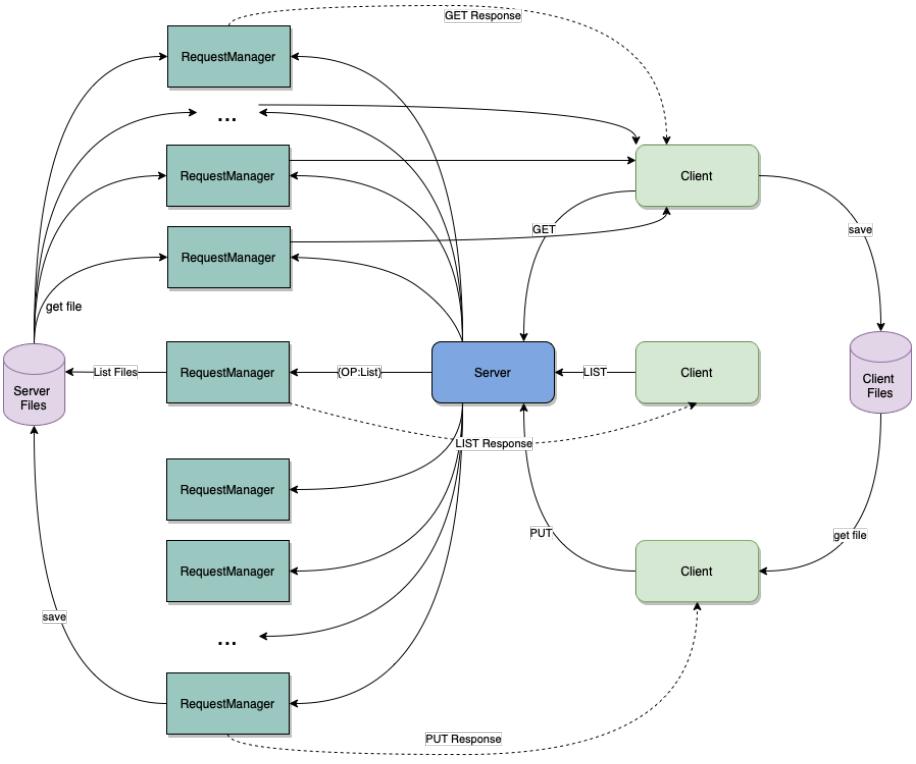


Figura 2.1: Semplificazione del Funzionamento dell'applicativo

L'architettura è composta da due moduli: il modulo lato `client` e il modulo lato `server`. Per quanto riguarda il lato *client*, esiste una classe `client`, la

quale fornisce ed espone all'esterno le funzionalità richieste, ovvero la funzione di *LIST*, *GET* e *PUT*. All'interno di questa classe sono stati impostati dei parametri di default che possono essere modificati a piacimento, come le dimensioni dei *buffer* di spedizione/ricezione, il tempo di default di *timeout* o le *directory* dove il client salva i file tramite la funzione *GET* o seleziona il file da spedire tramite la funzione *PUT*.

A differenza del client, il *server* si compone di due classi: la classe `server` si occupa di ricevere le richieste spedite dal client, verificarne la correttezza, prelevare i dati essenziali dal pacchetto e assegnare la gestione all'altra classe del modulo: `request_manager`. La classe `server` funge così da accettatore di richieste, potendo rimanere sempre in ascolto verso nuovi client o nuove richieste dallo stesso, poiché delega l'avvenuta delle operazioni alla classe `request_manager`. Quest'ultima è quindi un *thread*, e si occupa di soddisfare la richiesta del client, assegnatagli dalla classe `server`. Come la classe `client`, vengono assegnati parametri di default per le dimensioni dei *buffer* o la locazione della *directory* dei file contenuti sul server. Questa classe, oltre ad effettuare le operazioni richieste dal client, ha il compito di spedire a quest'ultimo uno speciale messaggio dove viene esplicitato il successo/fallimento dell'operazione (operazione "*FIN*", in fig. 2.1 è indicato dalla linea tratteggiata).

2.2 Design dettagliato

Il protocollo permette lo scambio di messaggi tramite un semplice segmento composto da un *header* e i dati. L'*header* comprende i seguenti campi:

- `Operation`: tipo di operazione richiesta (*LIST*, *GET*, *PUT*, *FIN*);
- `Sequence Number` del pacchetto corrente;
- `Checksum` calcolato sull'intero segmento. Viene utilizzata la funzione di *hashing* `SHA1` producendo un valore a 160 bit;

Nel segmento finale sarà aggiunto il campo `Payload` contente i dati da spedire. Nelle operazioni possibili è anche presente il tipo di operazione *FIN*, ovvero il messaggio di risposta del server verso il client, contente nel *paylaod* l'esito dell'operazione (0 fallimento, 1 successo). La struttura dati utilizzata è quindi un dizionario, e tramite la libreria `json`, è possibile serializzare il dizionario e spedirlo tramite il socket.

2.2.1 Client

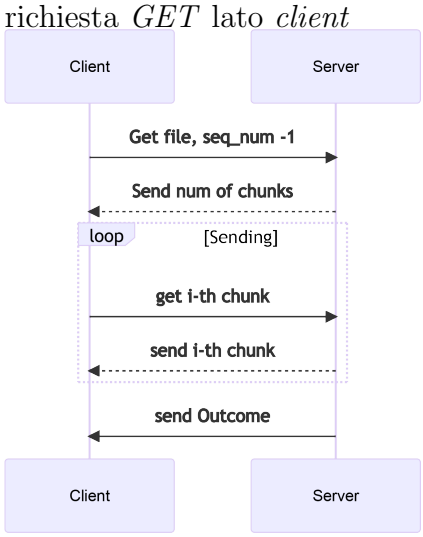
Lato client, `client_runner` permette l'attivazione del client e la scelta dell'operazione da far eseguire. Controlla inoltre se l'operazione è andata a buon fine, e visualizza a schermo l'output dei comandi eseguiti. La classe `client` come già detto, è il cuore della richiesta delle operazioni e permette la spedizione/ricezione di messaggi tramite un socket di tipo `SOCK_DGRAM`. Il socket utilizzato dal client possiede un *timeout* dal valore di default di 5 secondi, utilizzato in tutti quei casi in cui il server dovesse fallire a spedire risposte o sequenze di file. In caso di *timout*, il client termina l'operazione mostrando a video un messaggio di errore. Se fosse stato effettuato un *GET* e a metà dell'operazione si verifica un timeout, prima di terminare viene eliminato il file su cui si stava scrivendo. Per quanto riguarda la funzione *LIST*, viene inviata la richiesta della lista dei file al server, il quale pone nel *payload* del segmento di risposta, una stringa con i nomi dei file presenti separati da uno spazio. A questo punto è sufficiente leggere tutti i nomi tramite la funzione `split()` e mostrarli a video. In questa funzione specifica quindi, alla ricezione del messaggio contente la lista dei file, è implicitamente contenuto il messaggio di successo dell'operazione. In caso di fallimento il controllo sul contenuto del pacchetto (`self._check_incoming_package(data)`) fa in modo che se il server dovesse spedire un messaggio di tipo *FIN* con esito negativo, il client fallisce e ritorna mostrando a video un errore.

Le funzioni che prevedono il vero e proprio scambio di file (*GET*, *PUT*) procedono dividendo il contenuto in blocchi di `self.sending_rate` (valore di default 2048 byte) *bytes*, e per ognuno di essi viene assegnato un *sequence number* crescente, mano a mano che si legge il file. Nella funzione *GET*, quest'operazione di suddivisione viene effettuata dal server, mentre dal client per la funzione *PUT*.

Considerando la funzione *GET* lato client, il primo messaggio che viene spedito è la richiesta di *GET*, contente nel *payload* il nome del file e con il campo *sequence_number* pari a -1, per indicare al server che il primo messaggio che deve spedire deve contenere nel *payload* il numero di pacchetti che il client si deve aspettare per quel determinato file.

Una volta ricevuto il numero di *chunks* nei quali verrà suddiviso il file, il client procede a creare il nuovo file e a mano a mano, spedisce al server la richiesta del *chunk* i-esimo, lo riceve e lo scrive sul nuovo file (vedi digramma sezione 2.2.1). Ogni pacchetto ricevuto porta con se il proprio *seqence_number*, il quale viene comparato con quello atteso dal client. In caso i due non combaciassero, un errore viene mostrato a video e l'operazione fallisce terminando. L'operazione di *GET* va a buon fine se vengono ricevuti tutti i pacchetti attesi, e viene ricevuto infine il messaggio dal server *FIN* che specifica il successo del trasferimento del file. Nota: per permettere il trasferimento di file non testuali (immagni, video, ...) è stato necessario aprire i file in lettura/scrittura in modalità binaria, e siccome la libreria `json` non permette di mantenere *bytes* nella struttura dati, è stato necessario convertire il contenuto dei pacchetti tramite la libreria `base64` per effettuare un *binary-to-text-encoding*.

Figura 2.2: Semplificazione richiesta *GET* lato *client*



Infine la funzione *PUT* prevede l'invio di 3 tipi di messaggi verso il server, contraddistinti da un diverso valore del *sequence_number*:

- `-1`: Il client comunica al server di inziare un'operazione di *PUT*.
- `i=0...N`: Il client informa il server della spedizione dell'i-esimo *chunk*.
- `-2`: Il client notifica il server di aver terminato la spedizione dei pacchetti, ed è pronto a ricevere l'esito dell'operazione

Il client quindi opera in modo simile al server per la funzione *GET*, ovvero legge il file in input, lo divide in *chunks* e spedisce ognuno di questi insieme al relativo *sequence_number*. Una volta terminato l'invio di tutti i pacchetti, il client avvisa il server, rimane in attesa di un responso e mostra a video il risultato dell'operazione.

2.2.2 Server

Come già detto, il server si compone di due classi: `server` e `request_manager`. Il primo di questi si occupa della ricezione delle richieste da parte del client, analizza l'integrità e il contenuto del pacchetto, e infine delega a `request_manager` il compito di effettuare l'operazione, affinché il server possa sempre rimanere in ascolto per nuovi messaggi in arrivo anche da client differenti, non dovendosi preoccupare dell'operazione da svolgere o del suo esito. Il cuore del server risiede quindi in `request_manager`, il quale esegue le operazioni richieste dal client. Come già accennato, la funzione *GET* (fig. 2.4) risulta per certi versi simile alla funzione *PUT* eseguita dal client, in particolare il concetto di suddivisione del file in chunks in `sending_rate` *bytes*. La grande differenza è che ogni chunk è gestito e spedito da un diverso *thread*, e per capire quale sezione di file leggere, preparare e spedire, il client effettua una richiesta di uno specifico chunk, indicato dal *sequence number* all'interno del pacchetto. Questo numero, permette di far capire al `request_manager` quali *bytes* del file leggere e spedire. Tramite la funzione di python `seek(offset, whence)`, è infatti possibile determinare un *offset* nella lettura di un file, e di conseguenza sarà sufficiente leggere una porzione di `sending_rate`*`sequence_number` *bytes* di ques'utimo. All'arrivo dell'ultimo *sequence number* e dopo aver mandato l'ultimo *chunk*, viene spedito al client l'esito dell'operazione tramite il messaggio *FIN*. Per "attivare" la funzionalità di get e farsi spedire la dimensione del file, il client deve spedire al server un messaggio di *GET* con il nome del file nel *payload* e *sequence number* pari a -1, affinché il *thread* in carico (il quale non conosce lo stato dell'operazione globale, cioè non conosce a che punto del download si trova l'operazione) possa agire correttamente e far proseguire il client con l'avanzamento delle richieste dei vari *chunks*.

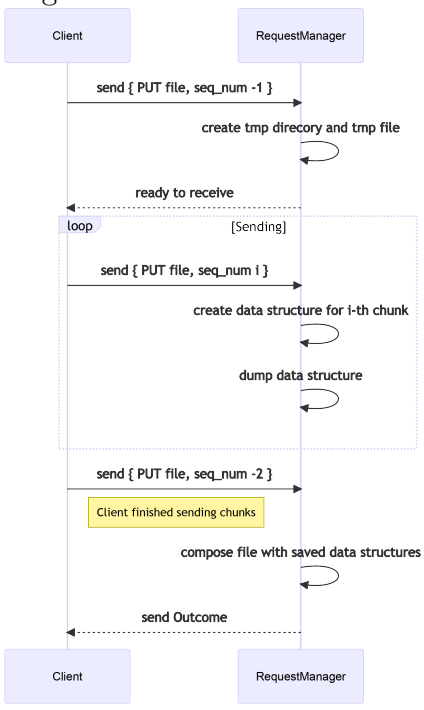
Questo uso di *sequence number* negativi permette inoltre di notificare i *threads* incaricati sullo stato dell'operazione *PUT*. Come per la funzione *GET*, per inizializzare l'operazione viene spedito un pacchetto con *sequence number* pari a -1 e il nome del file all'interno del campo *payload*. Il thread ricevente questo pacchetto, creerà in una *directory* temporanea il file (inizialmente vuoto). Successivamente, quando il client inzierà a spedire i vari *chunks* del file, ogni *thread* crea una struttura dati (dizionario) dove mantiene il *sequence number* di quel determinato *chunk* e il contenuto della porzione di

file arrivatagli. Dopodiché procede a salvare la struttura dati in un altro file temporaneo (*dump*). Sotto è mostrato una porzione del codice della funzione *PUT*, dove viene illustrato come viene creata la struttura dati e salvata su file.

```
# seqno = sequence number
# payload = binary content of file's slice
chunk = {
    "seqno" : seqno,
    "payload" : payload
}
w = open(dump_path, "a")
w.write(json.dumps(chunk))
w.close()
```

Il motivo dell'utilizzo di questa struttura dati è dato dal fatto che durante l'invio dei pacchetti da parte del client, le scritture sul file da parte dei diversi *threads* possono non avvenire nell'ordine giusto poiché dipendono dalle politiche di *scheduling*, causando così una corruzione del file stesso. Perciò salvando le porzioni del file numerate tramite il *sequence number*, al termine dell'invio dei *chunks* da parte del client sarà possibile riordinarle e salvarle su file. Nel momento in cui il client spedisce il pacchetto con *sequence number* pari a -2, il thread ricevente si occuperà della creazione e salvataggio del file vero e proprio: ciò avviene mediante la lettura del *dump* contenente tutte le strutture dati che compongono il file, le riordina in base al *sequence number*, e sequenzialmente salva sul file precedentemente creato i valori contenuti nel secondo campo della struttura dati. Una volta terminata la scrittura, viene spostato il file nella *directory* dei file sul server ed elimata la *directory* temporanea creata precedentemente. Il comportamento delle varie componenti è illustrato nella figura fig. 2.3.

Figura 2.3: Funzione PUT



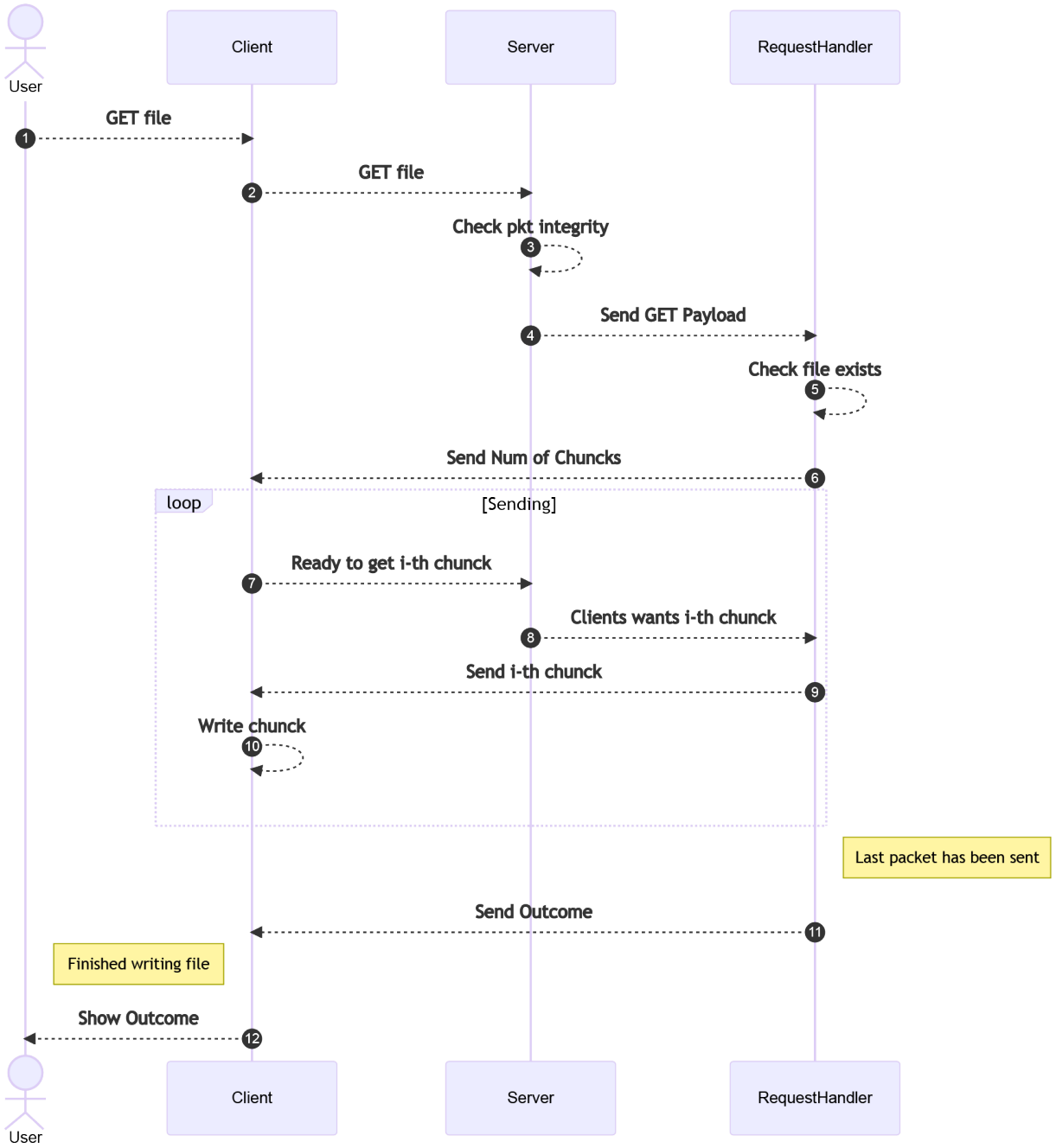


Figura 2.4: Diagramma di sequenza completo della funzione GET

Schema Generale dei *Threads*

Come si può evincere dallo schema sottostante, l'applicativo funziona attraverso una serie di *threads* lato server, incaricati di varie mansioni, a seconda dell'operazione richiesta dal client, e sono tutti istanze della stessa classe, ovvero `request_manager`. Per l'operazione *GET*, il primo thread incaricato della gestione del primo pacchetto spedito dal client, ha il compito di controllare l'esistenza del file richiesto, e in caso di esito positivo spedisce il numero di pacchetti che verranno spediti al client. Per quanto riguarda il resto dell'operazione *GET*, i thread incaricati spediranno *chunks* del file file selezionato, fino a quando l'ultimo pacchetto non viene spedito: in questo caso il thread incaricato, spedirà un messagio di esito al client, il quale terminerà la procedura di *GET*.

Per la funzione *PUT* invece, il compito del primo thread è quello di creare l'ambiente di lavoro per il salvataggio del file e successivamente avvisare il client che è tutto pronto per il ricevimento. I *threads* nella fase intermedia dell'operazione salvano i *chunks* ricevuti sul file temporaneo che, come detto precedentemente, può avvenire fuori ordine per via delle politiche di scheduling. Proprio per questo fattore l'ultimo thread incaricato ha il compito di ricomporre il file ordinando ogni *chunk* presente nel file temporaneo, e successivamente procede al salvataggio del file nella *directory* del server, concludendo l'operazione spedendo al client il messaggio di avvenuto successo/fallimento dell'operazione.