Ingegneria di Internet e del Web

Progetto B1 A.A. 2019/2020

TRASFERIMENTO FILE SU UDP

0253822

Fabio Buracchi

Indice

1. Requisiti

Lo scopo del progetto è quello di progettare ed implementare in linguaggi C usando l'API del socket di Berkeley un'applicazione client-server per il trasferimento di file che impieghi il servizio di rete senza connessione (socket tipo SOCK_DGRAM, ovvero UDP come protocollo di strato di trasporto.) Il software deve permettere:

- Connessione client-server senza autenticazione;
- La visualizzazione sul client dei file disponibili sul server (comando list);
- Il download di un file dal server (comando get);
- L'upload di un file sul server (comando put);
- Il trasferimento file in modo affidabile.

La comunicazione tra client e server deve avvenire tramite un opportuno protocollo. Il protocollo di comunicazione deve prevedere lo scambio di due tipi di messaggi:

1. messaggi di comando: vengono inviati dal client al server per richiedere l'esecuzione delle diverse operazioni
2. messaggi di risposta: vengono inviati dal server al client in risposta ad un comando con l'esito dell'operazione.

1.1 Funzionalità del server

Il server, di tipo concorrente, deve fornire le seguenti funzionalità:

- L'invio del messaggio di risposta al comando list al client richiedente; il messaggio di risposta contiene la filelist, ovvero la lista dei nomi dei file disponibili per la condivisione;
- L'invio del messaggio di risposta al comando get contenente il file richiesto, se presente, o un opportuno messaggio di errore;
- La ricezione di un messaggio put contenente il file da caricare sul server e l'invio di un messaggio di risposta con l'esito dell'operazione.

1.2 Funzionalità del client

I client, di tipo concorrente, devono fornire le seguenti funzionalità:

- L'invio del messaggio list per richiedere la lista dei nomi dei file disponibili;
- L'invio del messaggio get per ottenere un file
- La ricezione di un file richiesta tramite il messaggio di get o la gestione dell'eventuale errore
- L'invio del messaggio put per effettuare l'upload di un file sul server e la ricezione del messaggio di risposta con l'esito dell'operazione.

1.3 Trasmissione affidabile

Lo scambio di messaggi avviene usando un servizio di comunicazione non affidabile. Al fine di garantire la corretta spedizione/ricezione dei messaggi e dei file sia i client che il server implementano a livello applicativo il protocollo Go-Back N (cfr. Kurose & Ross "Reti di Calcolatori e Internet", 7° Edizione).

Per simulare la perdita dei messaggi in rete (evento alquanto improbabile in una rete locale per non parlare di quando client e server sono eseguiti sullo stesso host), si assume che ogni messaggio sia scartato dal mittente con probabilità p.

La dimensione della finestra di spedizione N, la probabilità di perdita dei messaggi p, e la durata del timeout T, sono tre costanti configurabili ed uguali per tutti i processi. Oltre all'uso di un timeout fisso, deve essere possibile scegliere l'uso di un valore per il timeout adattivo calcolato dinamicamente in base alla evoluzione dei ritardi di rete osservati.

I client ed il server devono essere eseguiti nello spazio utente senza richiedere privilegi di root. Il server deve essere in ascolto su una porta di default (configurabile).

2. Architettura

2.1 Protocollo TFTP

Il "Trivial File Transfer Protocol" (TFTP) è un protocollo semplice per il trasferimento di file, definito nell'RFC 1350, che consente ad un client di ottenere o inviare file verso un host remoto. Tale protocollo è progettato per essere implementato su UDP ed è equivalente al protocollo Stop-and-wait ARQ in termini di modalità di trasferimento dei pacchetti.

TFTP non include alcun meccanismo di controllo per l'autenticazione, per l'accesso, la riservatezza o la verifica dell'integrità dei dati. È necessario prestare attenzione ai diritti concessi a un processo server TFTP in modo da non violare la sicurezza del file system dell'host su cui è eseguito il processo. I server TFTP vengono generalmente configurati con privilegi tali da permettere il solo accesso in lettura dei file. A causa di tali considerazioni sulla sicurezza, il protocollo TFTP non trova generalmente applicazione nelle reti internet, mentre è principalmente utilizzato nelle reti locali (LAN) per operazioni come il trasferimento di immagini di avvio per nodi senza disco.

Un aspetto rilevante riguardo alle estensioni del protocollo, descritto nell'RFC 7440, riguarda l'introduzione di un'opzione che permette a client e server di negoziare una finestra di blocchi consecutivi da inviare, anziché utilizzare il tradizionale schema lockstep, al fine di migliorare il throughput complessivo. Tale estensione rende l'operazione di trasferimento di pacchetti equivalente a quella descritta dal protocollo Go-Back-N ARQ.

La scelta di adottare TFTP in questo progetto si fonda quindi su due motivazioni principali:

- **Semplicità:** TFTP garantisce una semplice implementazione rispettando i requisiti progettuali.
- **Utilità:** Rispetto allo sviluppo di un protocollo ex novo, utilizzare un protocollo standardizzato e popolare come TFTP massimizza la possibilità di riutilizzo del sistema sviluppato, oltre ad offrire una maggiore interoperabilità con sistemi e strumenti esistenti.

In sintesi, TFTP rappresenta una soluzione concreta e realizzabile per lo sviluppo di un'applicazione di trasferimento file utile, adatta a contesti reali e soddisfacente i requisiti progettuali.

2.1.1 Estensioni al protocollo TFTP

Il protocollo TFTP non prevede meccanismi di timeout adattivo o un comando per ottenere l'elenco dei file disponibili su cui eseguire operazioni di lettura. È stata dunque implementata un'estensione del protocollo TFTP con le seguenti modifiche:

- **Timeout adattivo:** È stata introdotta la possibilità di richiedere l'uso di un timeout adattivo per le richieste di lettura (RRQ). Ciò avviene specificando il valore `adaptive` per l'opzione `timeout` definita nell'RFC 2349. In questo modo, il server utilizzerà un algoritmo di timeout adattivo basato su stime dinamiche del Round-Trip Time (RTT).
- **Richiesta elenco file:** Per ottenere la lista dei file scaricabili, è possibile inviare una richiesta RRQ con l'opzione chiave-valore case insensitive `type:directory`. Il server, in risposta, restituirà l'elenco dei file disponibili nel percorso relativo specificato nel campo `filename`.

2.2 Server

Il server TFTP è un'applicazione single-process multi-threaded che gestisce in maniera isolata e concorrente le sessioni di comunicazione con i client.

Il server, al momento della sua inizializzazione, crea un numero configurabile di *worker thread* gestiti in una *worker pool*. Ciascun worker thread gestisce simultaneamente un numero massimo configurabile di sessioni. Le richieste dei client in ingresso vengono assegnate ad un worker utilizzando un algoritmo di bilanciamento del carico, in modo tale da ottimizzare l'utilizzo delle risorse.

Il thread principale si occupa di gestire le richieste in arrivo, creando una nuova sessione per ogni richiesta valida. La versione del protocollo Internet utilizzata dalla socket, impiegata dal thread principale per la ricezione dei pacchetti dai client, è configurabile. Assegnando alla socket un indirizzo IPv4, il server gestirà esclusivamente richieste IPv4. Al contrario, se viene assegnato un indirizzo IPv6, il server sarà in grado di gestire sia richieste IPv6 che IPv4, creando in quest'ultimo caso sessioni basate su connessioni IPv4 per supportare eventuali client che non supportano il protocollo IPv6.

Ciascun worker thread gestisce un ciclo di eventi basato sul multiplexing dell'I/O, che consente l'esecuzione di operazioni non bloccanti. In questo contesto, il ciclo non si limita a monitorare simultaneamente più canali di I/O, ma integra anche la gestione dei timeout.

È possibile abilitare il supporto, disabilitato per default per ragioni di sicurezza, alle:

- richieste di scrittura
- richieste di lettura della lista dei file di una directory
- richieste di lettura utilizzando un timeout adattivo

Il server integra un sistema di monitoraggio in grado di raccogliere dati statistici sia a livello di processo che di singola sessione. Questa funzionalità permette di analizzare in dettaglio e registrare l'andamento delle comunicazioni, agevolando il controllo del corretto funzionamento e la diagnosi di eventuali anomalie e facilitando l'integrazione del prodotto con applicazioni di terze parti.

2.3 Client

Il client TFTP è un'applicazione single-process e single-threaded. È supportata la configurazione della versione del protocollo Internet (IPv4 o IPv6) utilizzata per la comunicazione, garantendo così la compatibilità con diversi ambienti di rete. Poiché il protocollo TFTP non consente di effettuare trasferimenti multipli in una singola sessione, il client è stato progettato per operare in modalità non interattiva. Questa scelta riduce il consumo delle risorse, ad esempio minimizzando il tempo di occupazione del canale di comunicazione del server.

2.4 Timeout adattivo

Il timeout adattivo del server è stato progettato ispirandosi all'RFC 6298, che definisce l'algoritmo standard per il calcolo del timer di ritrasmissione nel protocollo TCP. Tuttavia, considerando l'impiego in una rete locale e l'uso con applicazioni TFTP, dove la congestione non rappresenta un problema significativo, sono state adottate durate iniziali e massime inferiori per il timeout. Il sistema utilizza una media esponenziale per stimare il Round-Trip Time (RTT), come descritto da Van Jacobson nel suo lavoro sul controllo della congestione, e applica un backoff esponenziale in caso di ritrasmissioni successive. Inoltre, l'algoritmo di Karn viene implementato per evitare che le misurazioni del RTT siano influenzate da ritrasmissioni, garantendo così una stima più accurata dei tempi di risposta.

3. Implementazione

Il prodotto finale si articola nei seguenti componenti:

- Un eseguibile, dotato di interfaccia a riga di comando, che implementa il server TFTP.
- Un eseguibile, anch'esso con interfaccia a riga di comando, destinato a svolgere il ruolo di client TFTP.
- Una libreria statica, denominata `tftp`, che raccoglie la logica del protocollo. Essa è distribuita insieme a file di intestazione in linguaggio C costituendo un'API completa per lo sviluppo di soluzioni basate sul protocollo TFTP.
- Una libreria statica, `logger` progettata per gestire il logging degli eventi.

Tutti i componenti sono stati sviluppati integralmente in linguaggio ISO C23 per piattaforme Linux, ad eccezione degli eseguibili, che includono anche sorgenti in linguaggio ISO C++23 costituenti un livello di compatibilità necessario a sfruttare la libreria `CLI11` per la realizzazione dell'interfaccia a riga di comando, in quanto le più popolari librerie C per il parsing degli argomenti si sono dimostrate inadeguate.

È notevole osservare che, oltre ad introdurre utili costrutti come l'inizializzatore vuoto (`={}`), gli attributi, il supporto per etichette seguite da dichiarazioni e la nuova keyword `nullptr`, lo standard ISO/IEC 9899:2024 corregge finalmente un importante difetto del linguaggio C, consentendo la dichiarazione del tipo sottostante di un tipo enumerazione, permettendo così la generazione di ABI stabili e coerenti, senza dover ricorrere ad abusi del preprocessore durante l'implementazione di protocolli di rete.

La generazione del progetto è stata gestita tramite CMake, mentre la gestione delle dipendenze è stata realizzata attraverso il gestore di pacchetti vcpkg. Per lo unit testing è stata impiegata la libreria `buracchi-cutest` da me sviluppata in assenza di alternative adeguate.

I sorgenti dei due eseguibili contengono l'implementazione dell'interfaccia a riga di comando e definiscono alcune interazioni con il file system di poco interesse, delegando invece alla libreria `tftp` la logica relativa all'implementazione del protocollo. Per tale motivo, anziché l'implementazione degli eseguibili, verrà discussa in seguito l'implementazione dei moduli server e client esposti dalla sopra citata libreria.

3.1 Server

3.1.1 Gestione della Socket di Ascolto

Il thread principale del server TFTP monitora costantemente una socket di tipo datagramma (`SOCK_DGRAM`) che funge da interfaccia passiva per l'accettazione delle richieste in ingresso. La configurazione della socket si adatta alla versione del protocollo IP associato: se legata a un indirizzo IPv4, il server gestisce esclusivamente richieste IPv4; se, invece, è associata a un indirizzo IPv6, opera in modalità dual-stack, accettando sia richieste IPv6 che IPv4. In quest'ultimo caso, le connessioni IPv4 sono trattate separatamente per garantire la compatibilità con client che non supportano IPv6.

Per assicurare tale flessibilità, sulla socket vengono attivate le opzioni `IP_RECVORIGDSTADDR` per IPv4 e `IPV6_RECVORIGDSTADDR` insieme a `IPV6_V6ONLY` per IPv6, permettendo il recupero dell'indirizzo di destinazione originale. I pacchetti vengono ricevuti mediante la funzione `recvmsg`, che fornisce sia il contenuto dei messaggi sia i metadati di controllo (dati ausiliari contenuti in `msg_control` della struttura `msghdr`), in particolare è verificata la presenza dell'attributo `IP_ORIGDSTADDR`. Tale meccanismo consente di distinguere le richieste IPv4 da quelle IPv6, adattando il comportamento del server di conseguenza.

3.1.2 Monitoraggio e Callback

Oltre alla gestione della socket sopra citata, il thread principale esegue periodicamente una callback configurabile a cui fornisce le statistiche sull'utilizzo del server. Inoltre, al termine di ciascuna sessione viene invocata un'altra callback, anch'essa configurabile, per notificare il completamento e le statistiche della sessione.

Questi meccanismi consentono una raccolta sistematica delle metriche e facilitano l'integrazione con sistemi di monitoraggio esterni, consentendo un'analisi dettagliata delle prestazioni operative.

3.1.3 I/O Multiplexing con io_uring

Per massimizzare l'efficienza delle operazioni di I/O, il server impiega un dispatcher di eventi asincroni basato sull'interfaccia `io_uring` del kernel Linux. Viene configurata una coda di completamento che consente l'invio simultaneo di richieste asincrone al sistema operativo; ad ogni operazione viene associato un contesto utile per la gestione del callback al completamento.

Questo approccio, rispetto a meccanismi tradizionali come `epoll` che si limitano a notificare la disponibilità dei descrittori, permette di inoltrare direttamente le operazioni alla coda del kernel, riducendo il numero di chiamate di sistema e minimizzando i cambi di contesto.

3.1.4 Pool di Worker Multi-Threaded

La pool di worker, implementata utilizzando le definizioni contenute nell'header `threads.h` della libreria standard C, prevede l'allocazione di un array di dimensione configurabile di worker thread. Ogni worker viene inizializzato con il proprio dispatcher e con un semaforo (anch'esso di valore iniziale configurabile) per monitorare i job attivi, ognuno di essi relativo ad una singola sessione TFTP. Il bilanciamento del carico avviene assegnando al worker con il minor numero di job attivi il lavoro corrispondente alla gestione di una nuova sessione.

Durante il ciclo operativo, ciascun worker attende eventi dalla coda del proprio dispatcher, inoltrando ciascun evento al job a cui è registrato. Al completamento di una sessione, il worker rilascia la risorsa e incrementa il semaforo, segnalando la sua disponibilità a gestire nuove sessioni.

3.1.5 Virtualizzazione della Lista dei File

Per supportare il comando `list`, il server interpreta una RRQ (read request) contenente l'opzione case-insensitive `type:directory` come richiesta di elenco file. In tal caso, anziché aprire un file tradizionale, viene gestita la directory indicata utilizzando le funzioni definite nell'header `dirent.h` dello standard POSIX.

La lista dei file viene elaborata come una stringa e trascritta in una pipe, che il server tratta come un file regolare. In questo modo la lista dei file della directory viene virtualizzata come un file stesso, consentendo l'utilizzo di un'unica logica per la gestione di qualunque richiesta di lettura.

3.2 Client

Per semplicità, il client non utilizza tecniche di I/O multiplexing o di I/O asincrono. Questa scelta progettuale è stata adottata in quanto si è ritenuto che il guadagno prestazionale non sarebbe stato sufficiente a giustificare l'aumento di complessità dell'implementazione. Di conseguenza, l'algoritmo Go-Back-N, descritto in Kurose & Ross "Reti di Calcolatori e Internet", 7° Edizione, viene sostituito da una corrispondente versione sincrona. In questo modello, la gestione degli eventi (ricezione di un ACK o scadenza del timeout) avviene in maniera sequenziale, effettuando alcune considerazioni per il corretto controllo dei timeout.

L'originale algoritmo `Sender` e la sua variante `Sender_sync` sono in seguito riportati:

Algorithm 1 Algoritmo `Sender`

```
1:  window_begin ← 1
2:  next_seq_num ← 1
3:  while true do
4:      if next_seq_num < window_begin + window_size then
5:          fetch data
6:      end if
7:
8:      event data ready
9:          make packet next_seq_num from data
10:         send packet next_seq_num
11:         if window_begin = next_seq_num then
12:             start timer
13:         end if
14:         next_seq_num ← next_seq_num + 1
15:
16:     event timer timeout
17:         start timer
18:         for i ← window_begin to next_seq_num − 1 do
19:             send packet i
20:         end for
21:
22:     event received ACK n
23:         window_begin ← n + 1
24:         stop timer
25:         if window_begin ≠ next_seq_num then
26:             start timer
27:         end if
28:
29: end while
```

Algorithm 2 Algoritmo `Sender_sync`

```
1:  window_begin ← 1
2:  next_seq_num ← 1
3:  start_time ← undefined
4:  while true do
5:      while next_seq_num < window_begin + window_size do
6:          fetch data
7:          make packet next_seq_num from data
8:          send packet next_seq_num
9:          if window_begin = next_seq_num then
10:             start_time ← current time
11:         end if
12:         next_seq_num ← next_seq_num + 1
13:     end while
14:     remaining_time ← TIMEOUT − (current time −start_time)
15:     if remaining_time > 0 then
16:         set recv timeout to remaining_time
17:         event ← recv
18:     else
19:         event ← timeout
20:     end if
21:     if event = timeout then
22:         start_time ← current time
23:         for i ← window_begin to next_seq_num − 1 do
24:             send packet i
25:         end for
26:     else if event = ACK n then
27:         window_begin ← n + 1
28:         if window_begin ≠ next_seq_num then
29:             start_time ← current time
30:         end if
31:     end if
32: end while
```

Dove `recv` può restituire un messaggio oppure un segnale di timeout e `TIMEOUT` rappresenta il valore del timeout.

3.3 Simulazione della perdita dei pacchetti

La simulazione della perdita dei pacchetti è stata implementata mediante il wrapping a linking time delle funzioni di invio e ricezione (e.g. `sendto` e `recvfrom`). Non è stato utilizzato il BPF o l'EBPF a causa delle recenti politiche sulla sicurezza che hanno reso i privilegi di root sempre necessari. Viene generato un valore pseudocasuale normalizzato nell'intervallo [0,1] e si verifica se esso è inferiore ad una soglia configurabile, in tal caso il pacchetto viene considerato "perso": in fase di invio, il pacchetto non sarà trasmesso, mentre in fase di ricezione verrà scartato. Lato server, inoltre, tale logica è integrata con un thread dedicato e un dispatcher; in caso di simulazione di perdita, la richiesta di ricezione sarà ridirezionata verso il dispatcher del thread, che scarterà il pacchetto una volta disponibile.

4. Limitazioni e criticità

Nel corso dello sviluppo del sistema sono emerse numerose limitazioni, principalmente riconducibili alla scarsità di risorse, in particolare di tempo, a disposizione per lo sviluppo. Tale vincolo ha imposto una serie di compromessi progettuali, i quali hanno inciso su diversi aspetti implementativi e architetturali.

In primo luogo, la scelta di utilizzare l'header `threads.h` della libreria standard C, invece di affidarsi all'API `pthread` dello standard POSIX, ha determinato alcune restrizioni importanti. In particolare, tale scelta impedisce di eseguire ottimizzazioni mirate come l'impostazione dell'affinità del processore, rende dubbia la correttezza della gestione dei segnali costringendo ad utilizzare entrambe le librerie e limita l'utilizzo di strumenti di validazione del codice come il Thread Sanitizer.

Non è stato possibile implementare una terza libreria intermedia che, tramite callback, avrebbe disaccoppiato la logica di logging dalla libreria `tftp`. Il sistema ha accorpato tali funzionalità, generando una dipendenza indesiderata e riducendo la modularità complessiva.

L'interfaccia del dispatcher, risulta grezza e costringe il programmatore a gestire dettagli implementativi che sarebbero potuti essere opportunamente astratti, rendendo il codice più difficile da leggere e manutenere. Ciò ha inoltre causato una complessità maggiore relativa ai problemi di concorrenza, comportando l'eliminazione di alcune ottimizzazioni come la vettorizzazione dell'I/O, al fine di mitigare condizioni di corsa irrisolte.

La duplicazione del codice tra le componenti client e server, anziché una sua efficace condivisione, rappresenta un ulteriore obiettivo per una futura operazione di refactoring del codice. Anche l'interfaccia per le astrazioni del protocollo TFTP, sebbene funzionale, non raggiunge il livello di eleganza e chiarezza auspicabile e necessita pertanto di una ridefinizione.

Il supporto per la modalità `netascii` e per l'opzione `tsize` risulta parziale, e la copertura del codice mediante unit test non è sufficiente a garantire una validazione completa del comportamento del prodotto e del rispetto dei requisiti contenuti negli RFC considerati.

Infine, anche la raccolta dei dati relativi alle metriche risulta incompleta, limitando i possibili casi d'uso inerenti al monitoraggio del sistema.

5. Ambiente di sviluppo e di testing

Per lo sviluppo e la validazione del sistema sono state adottate diverse soluzioni tecnologiche, che hanno consentito di creare un ambiente flessibile e replicabile sia per le attività di coding che per quelle di testing. Di seguito viene presentata una panoramica dettagliata dell'ambiente impiegato.

Il progetto è stato interamente sviluppato su piattaforme Linux, utilizzando Ubuntu 22.04 LTS su Windows 10 x86_64 con kernel 6.6.36.3-microsoft-standard-WSL2 come ambiente di riferimento.

Le scelte tecnologiche principali includono:

- **Compilatori:** Il codice sorgente è stato compilato utilizzando GCC 14.2 per garantire la conformità agli standard ISO C23 e C++23. In alcune fasi di sviluppo, è stato testato anche il compilatore Clang 19 per verificare la portabilità.

- **Tool di Build e Gestione delle Dipendenze:** La generazione del progetto è stata gestita tramite CMake, mentre le dipendenze esterne sono state integrate attraverso il package manager `vcpkg`.
- **Strumenti di Debug e Profiling:** Per il controllo e l'analisi delle performance, sono stati impiegati GDB e Valgrind, utili per l'identificazione di eventuali memory leak e per il tracciamento di anomalie nel comportamento delle applicazioni.
- **Strumenti del compilatore:** sono stati utilizzati strumenti del compilatore come l'Address Sanitizer, il Leak Sanitizer e l'Undefined Behavior Sanitizer per l'individuazione di errori di memoria e comportamenti indefiniti.

L'approccio al testing del sistema ha previsto una combinazione di test unitari, test integrati e simulazioni in condizioni reali, al fine di garantire una copertura completa delle funzionalità implementate.

- **Testing Unitario:** I test unitari sono stati realizzati con la libreria `buracchi-cutest`, sviluppata ad hoc, che ha permesso di validare le funzionalità critiche della logica TFTP.
- **Coverage e Analisi Statica:** Strumenti come gcov sono stati impiegati per monitorare la copertura del codice durante l'esecuzione dei test, evidenziando eventuali aree non coperte e guidando ulteriori sviluppi. Inoltre, l'integrazione di analizzatori statici ha contribuito a identificare e correggere potenziali criticità prima della fase di integrazione finale.
- **Continuous Integration:** L'intero ciclo di test è stato automatizzato mediante una pipeline di Continuous Integration, realizzata con GitHub CI/CD. Ogni commit attiva una serie di job che eseguono i test unitari e integrati, garantendo così una rapida individuazione di regressioni e il mantenimento della qualità del codice.

Complessivamente, l'ambiente di sviluppo e di testing adottato ha permesso di ottenere una elevata affidabilità del sistema, supportando il lavoro di debugging e facilitando l'integrazione di nuove funzionalità in maniera modulare.

6. Esempi di funzionamento

Di seguito vengono riportati alcuni esempi di invocazione dei programmi, client e server, utili per comprendere il corretto funzionamento delle applicazioni TFTP.

6.1 Avvio del TFTP Server

- Avvio con directory specifica e abilitazione delle richieste di scrittura:

```
./server --enable-write-requests /home/utente/tftp_root
```

- Avvio con configurazione personalizzata dei thread e dei parametri di rete:

```
./server -w 10 -m 50 -H 2001:db8::1 -p 6969 /home/utente/tftp_root
```

6.2 Utilizzo del TFTP Client

- Scaricare un file dal server:

```
./client 192.0.2.0 get documento.txt
```

- Caricare un file sul server:

```
./client 192.0.2.0 put aggiornamento.bin
```

- Ottenere l'elenco dei file presenti in una directory sul server:

```
./client 192.0.2.0 list /home/utente/tftp_root
```

Questi esempi dimostrano la flessibilità e la facilità di configurazione offerte dalle applicazioni, permettendo di adattare il sistema a differenti scenari di rete e requisiti operativi. La lista dettagliata dei comandi e delle opzioni riconosciute dagli eseguibili è contenuta nella sezione Manuale d'uso.

7. Valutazione delle prestazioni

L'analisi delle prestazioni del sistema è stata condotta considerando quattro parametri principali: la dimensione dei file, la dimensione della finestra di spedizione, la probabilità di perdita dei messaggi e la modalità di timeout (costante o adattiva).

I benchmark sono stati eseguiti con l'obiettivo di valutare l'impatto di tali parametri sul tempo di trasferimento durante il download dei file.

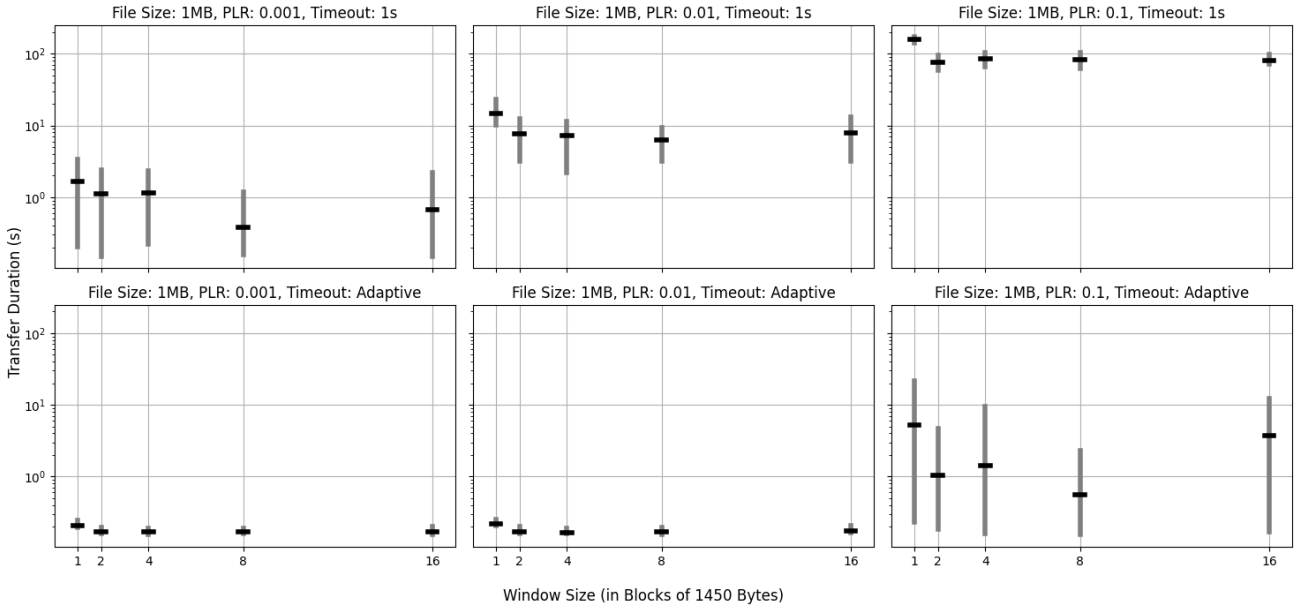
Al fine di garantire l'affidabilità dei dati raccolti, sono state adottate misure volte a mitigare gli effetti della memorizzazione dei file nella cache del sistema operativo. Inoltre, per ciascun valore della probabilità di perdita, è stata mantenuta costante la sequenza di numeri pseudocasuali utilizzata nella simulazione della perdita dei pacchetti, assicurando così la stessa distribuzione degli eventi di perdita tra le diverse esecuzioni. L'analisi sottostante è il risultato di 10 esecuzioni di ciascun caso di interesse, effettuate secondo le modalità sopra descritte, ad eccezione dei test che hanno richiesto un tempo maggiore di due ore per essere completati, in tal caso si è considerato un numero inferiore di campioni.

I test sono stati eseguiti considerando le seguenti variabili sperimentali:

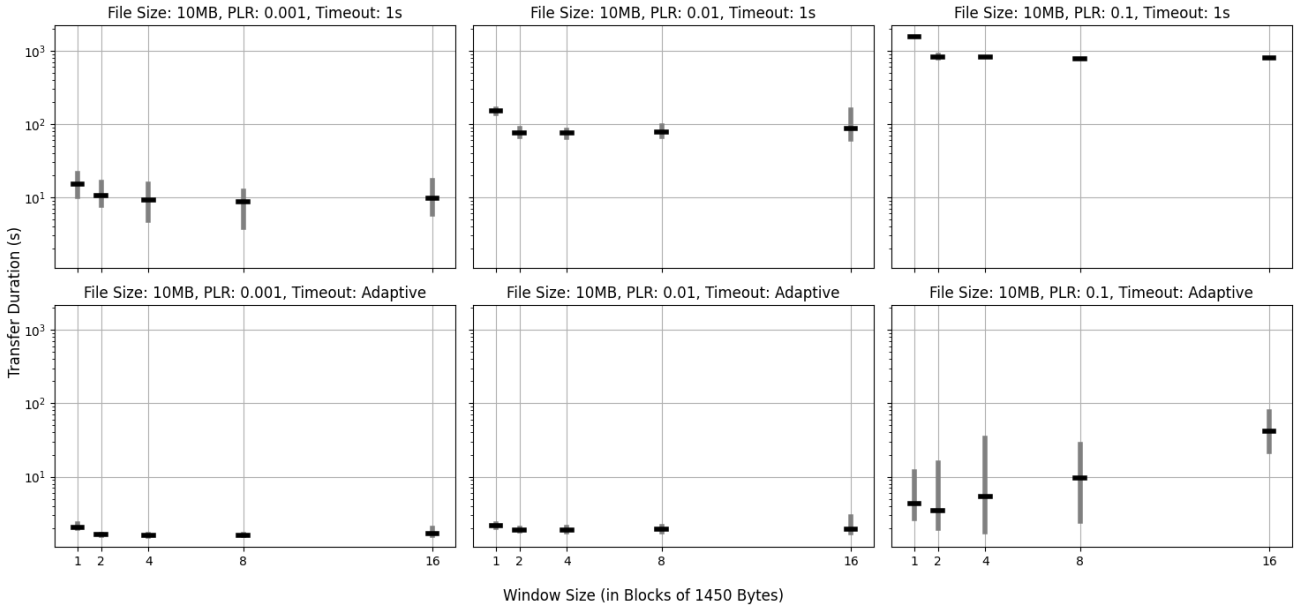
- le dimensioni dei file da trasferire, utilizzando file di dimensioni 1MB, 10MB e 100MB;
- la dimensione della finestra di spedizione N, con $N \in 1, 2, 4, 8, 16$;
- la modalità di timeout T, confrontando il valore di 1s con quello dell'RTO stimato in modalità adattiva (indicato in seguito con il simbolo A);
- la probabilità di perdita di pacchetto p, con $p \in 0.001, 0.01, 0.1$.

Per evitare la frammentazione a livello IP, la dimensione massima dei blocchi di dati trasmessi è stata limitata a 1450 byte. Tale valore è stato scelto considerando come livello di accesso alla rete il protocollo Ethernet, la cui Unità Massima di Trasmissione (MTU) è pari a 1500 byte (escluso l'overhead dato dall'intestazione del protocollo).

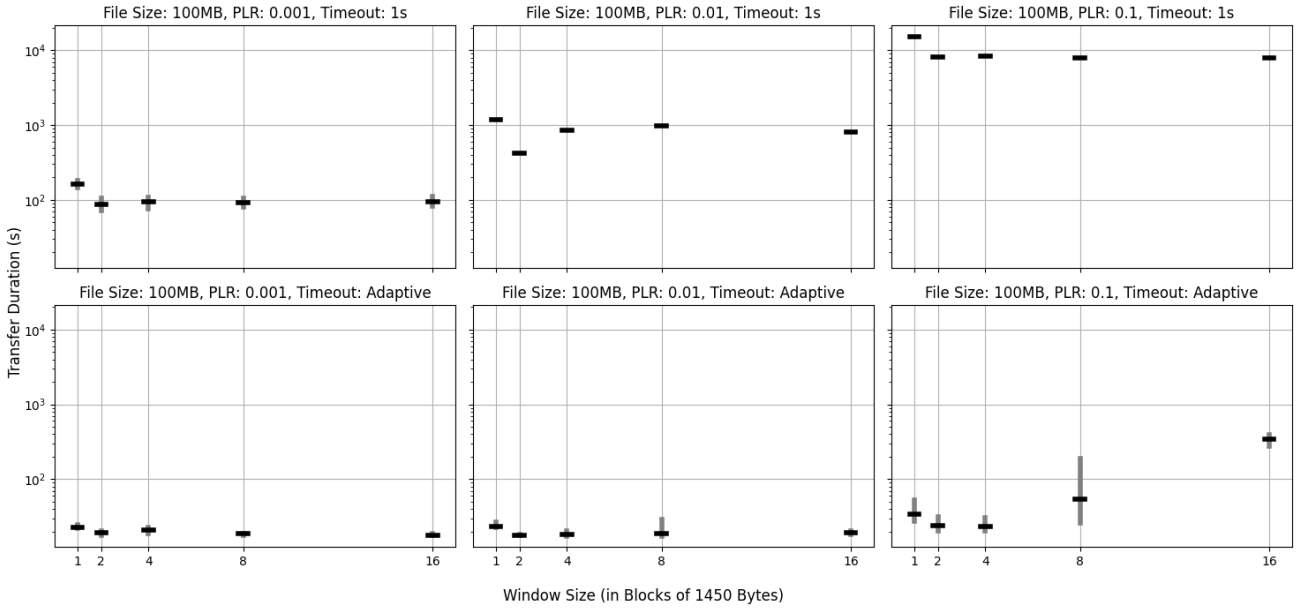
Dimensione File	T	N	p	**Durata (min)**	**Durata (avg)**	**Durata (max)**
1MB	1s	1	0.001	0.205	1.663	3.412
1MB	1s	2	0.001	0.149	1.126	2.364
1MB	1s	4	0.001	0.220	1.150	2.348
1MB	1s	8	0.001	0.157	0.377	1.163
1MB	1s	16	0.001	0.148	0.667	2.185
1MB	1s	1	0.01	10.292	14.636	23.365
1MB	1s	2	0.01	3.179	7.625	12.252
1MB	1s	4	0.01	2.201	7.325	11.260
1MB	1s	8	0.01	3.196	6.264	9.249
1MB	1s	16	0.01	3.150	7.976	13.223
1MB	1s	1	0.1	142.606	156.518	173.751
1MB	1s	2	0.1	59.291	76.405	94.287
1MB	1s	4	0.1	66.393	85.471	102.370
1MB	1s	8	0.1	62.286	83.243	104.192
1MB	1s	16	0.1	71.147	80.963	97.161
1MB	A	1	0.001	0.197	0.209	0.246
1MB	A	2	0.001	0.158	0.169	0.197
1MB	A	4	0.001	0.155	0.172	0.188
1MB	A	8	0.001	0.161	0.167	0.188
1MB	A	16	0.001	0.155	0.169	0.201
1MB	A	1	0.01	0.206	0.217	0.253
1MB	A	2	0.01	0.159	0.168	0.200
1MB	A	4	0.01	0.159	0.167	0.189
1MB	A	8	0.01	0.156	0.168	0.193
1MB	A	16	0.01	0.164	0.174	0.209
1MB	A	1	0.1	0.234	5.173	21.388
1MB	A	2	0.1	0.186	1.042	4.700
1MB	A	4	0.1	0.160	1.436	9.393
1MB	A	8	0.1	0.154	0.561	2.321
1MB	A	16	0.1	0.168	3.719	12.326



Dimensione File	T	N	p	**Durata (min)**	**Durata (avg)**	**Durata (max)**
10MB	1s	1	0.001	10.274	15.329	21.297
10MB	1s	2	0.001	7.677	10.437	16.011
10MB	1s	4	0.001	4.923	9.092	15.043
10MB	1s	8	0.001	3.888	8.690	12.273
10MB	1s	16	0.001	5.871	9.728	17.017
10MB	1s	1	0.01	137.437	149.161	158.904
10MB	1s	2	0.01	66.831	75.273	85.625
10MB	1s	4	0.01	65.319	76.243	83.083
10MB	1s	8	0.01	67.230	76.943	94.385
10MB	1s	16	0.01	63.062	86.293	155.677
10MB	1s	1	0.1	1558.596	1558.596	1558.596
10MB	1s	2	0.1	793.479	821.831	863.758
10MB	1s	4	0.1	827.607	827.607	827.607
10MB	1s	8	0.1	785.314	785.314	785.314
10MB	1s	16	0.1	802.164	802.164	802.164
10MB	A	1	0.001	1.963	2.038	2.263
10MB	A	2	0.001	1.587	1.611	1.639
10MB	A	4	0.001	1.548	1.584	1.619
10MB	A	8	0.001	1.576	1.602	1.624
10MB	A	16	0.001	1.596	1.684	1.949
10MB	A	1	0.01	2.031	2.139	2.250
10MB	A	2	0.01	1.807	1.867	1.988
10MB	A	4	0.01	1.764	1.865	2.059
10MB	A	8	0.01	1.782	1.899	2.092
10MB	A	16	0.01	1.719	1.913	2.818
10MB	A	1	0.1	2.691	4.341	11.706
10MB	A	2	0.1	1.981	3.415	15.418
10MB	A	4	0.1	1.774	5.376	33.431
10MB	A	8	0.1	2.476	9.603	27.438
10MB	A	16	0.1	22.161	41.741	76.662



Dimensione File	T	N	p	**Durata (min)**	**Durata (avg)**	**Durata (max)**
100MB	1s	1	0.001	145.600	162.794	180.984
100MB	1s	2	0.001	71.801	87.836	105.588
100MB	1s	4	0.001	74.965	93.232	108.449
100MB	1s	8	0.001	79.147	90.768	105.707
100MB	1s	16	0.001	82.060	93.611	112.438
100MB	1s	1	0.01	1193.140	1193.140	1193.140
100MB	1s	2	0.01	424.000	424.000	424.000
100MB	1s	4	0.01	862.400	862.400	862.400
100MB	1s	8	0.01	987.700	987.700	987.700
100MB	1s	16	0.01	804.600	804.600	804.600
100MB	1s	1	0.1	15265.835	15265.835	15265.835
100MB	1s	2	0.1	8218.300	8218.300	8218.300
100MB	1s	4	0.1	8276.070	8276.070	8276.070
100MB	1s	8	0.1	7853.140	7853.140	7853.140
100MB	1s	16	0.1	7994.111	7994.111	7994.111
100MB	A	1	0.001	22.148	22.514	24.328
100MB	A	2	0.001	17.789	19.183	20.526
100MB	A	4	0.001	18.938	20.990	22.852
100MB	A	8	0.001	17.758	18.540	18.791
100MB	A	16	0.001	17.753	18.016	18.643
100MB	A	1	0.01	22.497	23.497	26.472
100MB	A	2	0.01	17.754	18.008	18.316
100MB	A	4	0.01	17.391	18.220	20.440
100MB	A	8	0.01	17.500	18.907	28.858
100MB	A	16	0.01	18.497	19.436	20.598
100MB	A	1	0.1	27.755	33.784	53.118
100MB	A	2	0.1	20.190	23.729	31.599
100MB	A	4	0.1	20.461	23.204	30.730
100MB	A	8	0.1	25.975	53.874	187.643
100MB	A	16	0.1	274.613	343.539	395.516



Si osserva che, nel protocollo TFTP come definito nell'RFC 2349, essendo il valore minimo per il timeout pari a 1 secondo, l'impiego di un timeout adattivo può migliorare in modo significativo le prestazioni in reti soggette a elevati tassi di errore, come le reti wireless conformi allo standard IEEE 802.11 (Wi-Fi).

Notiamo inoltre come, variando le dimensioni della finestra di spedizione, non si riscontrano variazioni significative nella durata delle trasmissioni. Tale risultato è riconducibile alla modalità con cui sono stati condotti i test: utilizzando un unico nodo di rete sia come client sia come server, il ritardo di propagazione è stato completamente virtualizzato, consentendo il raggiungimento rapido della condizione di trasmissione continua. I risultati dei test effettuati non sono dunque rappresentativi delle prestazioni percepite su una rete ad alta latenza.

8. Installazione, configurazione ed esecuzione

8.1 Requisiti di Sistema

Per garantire il corretto funzionamento del sistema, è necessario soddisfare i seguenti requisiti minimi:

- **Sistema operativo:** distribuzione Linux compatibile.
- **Spazio su disco:** almeno 4 MB di spazio disponibile su disco.
- **Versione del kernel:** il server richiede un kernel Linux versione 6.1 o successiva per poter essere eseguito correttamente.

Il sistema di compilazione supporta ufficialmente la sola architettura `amd64`. L'utilizzo di architetture diverse richiede azioni aggiuntive per la generazione di un preset valido per la compilazione.

8.2 Procedura di Compilazione e Build

Per poter compilare il progetto è necessario aver installato i seguenti programmi: `gdb`, `make`, `ninja-build`, `rsync`, `zip`, `cmake`, `g++` e `gcc` assicurandosi che la versione di gcc installata supporti C23.

Dopo aver scaricato i file sorgente del progetto clonando la relativa repository github, ed essersi spostati nella directory generata eseguendo da terminale i comandi

```
git clone --recurse-submodules https://github.com/buracchi/tftp.git
cd tftp
```

è possibile compilare la soluzione eseguendo sempre da terminale i comandi

```
cmake -S . --preset x64-linux-release
cmake --build --preset x64-linux-release -j $(nproc)
```

I file compilati saranno contenuti nella sottodirectory `build`.

8.3 Installazione

Nel caso in cui non si desideri compilare manualmente il software, è possibile scaricare semplicemente il tarball online contenente i binari precompilati eseguendo da terminale i comandi

```
wget https://github.com/buracchi/tftp/releases/download/v1.0/tftp-1.0-x64.tar.gz
tar -xzf tftp-1.0-x64.tar.gz
```

Gli eseguibili saranno contenuti all'interno della directory `bin`.

8.4 Manuale d'uso

8.4.1 TFTP Server

Il TFTP Server espone numerose opzioni che consentono di configurare il comportamento operativo in base alle esigenze specifiche. Il comando di esecuzione di base è il seguente:

`./server [OPTIONS] [directory]`

dove:

- `directory ROOT`: Specifica la directory radice per lo storage dei file (default: directory corrente).
- `--enable-write-requests`: Abilita le richieste di scrittura.
- `--enable-list-requests`: Abilita le richieste di elenco file.
- `--enable-adaptive-timeout`: Abilita il timeout adattivo, basato sui ritardi di rete.
- `-w, --workers COUNT`: Imposta il numero di thread lavoratori (default: 7).
- `-m, --max-sessions-per-worker SESSIONS`: Definisce il numero massimo di sessioni per ciascun worker (default: 32).

Le impostazioni di rete possono essere personalizzate tramite:

- `-H, --host ADDRESS`: Specifica l'indirizzo a cui il server si deve associare (default: ::).
- `-p, --port PORT`: Indica la porta su cui il server attende le connessioni (default: 6969).
- `-r, --retries RETRIES`: Definisce il numero di tentativi di ritrasmissione prima di interrompere la comunicazione (default: 5).
- `-t, --timeout SECONDS`: Imposta il timeout in secondi prima di una ritrasmissione (default: 2).

Per scopi di debug e simulazione, il server offre ulteriori opzioni:

- `-l, --loss-probability PROBABILITY`: Configura la probabilità di simulare la perdita di pacchetti (default: 0.0).
- `--disable-fixed-seed`: Disabilita la scelta di un seme fisso per la generazione di numeri pseudocasuali durante la simulazione della perdita di pacchetti.
- `-v, --log-level LEVEL`: Specifica il livello di verbosità dei log (default: info).

8.4.2 TFTP Client

Il TFTP Client, progettato per interagire con il server, supporta diverse modalità operative accessibili tramite specifici suottocomandi. Il comando base è:

`./client [OPTIONS] host SUBCOMMAND`

dove **host** rappresenta l'indirizzo IP o il nome dell'host del server.

I sotto comandi supportati sono:

- `get [OPTIONS] filename`: Scarica un file dal server. Le opzioni includono:
  - `-m, --mode MODE`: Specifica la modalità di trasferimento (default: octet).
  - `-o, --output OUTPUT_FILE`: Imposta il nome del file di output.
- `put [OPTIONS] filename`: Carica un file sul server. È disponibile l'opzione:
  - `-m, --mode MODE`: Imposta la modalità di trasferimento (default: octet).
- `list [OPTIONS] [directory]`: Richiede l'elenco dei file presenti in una directory sul server. Il parametro `directory` è opzionale (default: directory corrente) e può essere accompagnato dall'opzione:
  - `-m, --mode MODE`: Specifica la modalità di trasferimento (default: octet).

Altre opzioni comuni del client includono:

- `-p, --port PORT`: Imposta la porta per la connessione (default: 6969).
- `-r, --retries RETRIES`: Definisce il numero di tentativi prima di interrompere l'operazione (default: 3).
- `-t, --timeout SECONDS`: Specifica il timeout in secondi per l'attesa di una risposta.
- `-b, --block-size BLOCK_SIZE`: Imposta la dimensione del blocco dati per il trasferimento.
- `-w, --window-size WINDOW_SIZE`: Configura la dimensione della finestra di dispatch.
- `-a, --adaptive-timeout`: Abilita il timeout adattivo, basato sui ritardi di rete.
- `--use-tsize`: Richiede la dimensione del file dal server.
- `-l, --loss-probability PROBABILITY`: Simula una determinata probabilità di perdita dei pacchetti (default: 0.0).
- `--disable-fixed-seed`: Disabilita la scelta di un seme fisso per la generazione di numeri pseudocasuali durante la simulazione della perdita di pacchetti.
- `-v, --log-level LEVEL`: Imposta il livello di dettaglio dei log (default: info).