Interazione tra HTTP e TCP

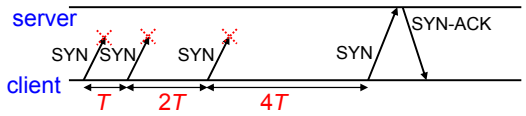
Valeria Cardellini
Università di Roma Tor Vergata

Timer del protocollo TCP

- Il protocollo TCP usa alcuni timer per attivare alcune operazioni
- Ritrasmissione di pacchetti persi
  - Attivata dal mittente TCP allo scadere di un timer
- Ripetizione della fase di slow-start
  - In alcune implementazioni del TCP il mittente TCP ripete lo slow start dopo un periodo di inattività
- Recupero dello stato da una connessione terminata
  - Il mittente TCP che ha iniziato la chiusura della connessione rimuove lo stato della connessione allo scadere di un timer

Timer di ritrasmissione

- Ritardo nello stabilire una connessione TCP
- ReTransmission Timeout (**RTO**) usato dal mittente TCP per individuare la perdita di un pacchetto
- Tradeoff nella scelta dell'RTO
  - RTO grande: latenza considerevole per gestire la perdita
  - RTO piccolo: ritrasmissioni inutili
  - Soluzione: il mittente TCP sceglie l'RTO sulla base del Round Trip Time (*RTT*) misurato verso il destinatario
- Problema: come selezionare l'RTO all'inizio della connessione TCP?
  - Valore iniziale dell'RTO pari a 3 secondi (*T*)
  - Backoff esponenziale in caso di perdite multiple
    - Se il mittente non riceve il SYN-ACK dal destinatario entro l'RTO, raddoppia il valore corrente dell'RTO



Timer di ritrasmissione (2)

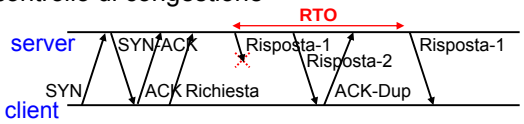
- La perdita di pacchetti dipende dal traffico sulla rete e dal carico sul server
  - SO del server: coda delle connessioni TCP pendenti (TCP SYN scartati quando la coda è piena)
- Comportamento dell'utente di tipo interrompi-ricarica: equivale ad una "ritrasmissione" immediata del SYN da parte del client
  - Riduce la latenza percepita dall'utente
  - … ma aumenta il carico sulla rete e sul server

Timer di ritrasmissione (3)

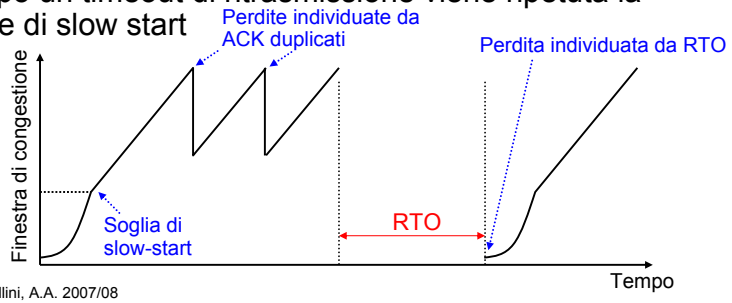
- Timeout di ritrasmissioni elevati nel mezzo di un trasferimento Web sono meno frequenti a causa di:
  - Valore minore dell'RTO
    - A regime il valore dell'RTO stabilito dinamicamente dal mittente TCP si avvicina al valore dell'RTT tra mittente e destinatario
  - Acknowledgment duplicati
    - E' l'altro meccanismo del TCP per gestire la perdita di pacchetti (ACK duplicati e RTO sono tra loro indipendenti)
    - In caso di mancata ricezione di un pacchetto, il destinatario non incrementa il numero di acknowledgment
    - Il mittente riceve pacchetti con ACK duplicati
      - *Es.*: il mittente invia i pacchetti 1, 2, 3, 4, 5 ma 2 si perde
      - Alla ricezione di 1, il destinatario invia ACK usando il primo byte di 2; dopo aver ricevuto 3 invia ACK usando il primo byte di 2…
    - Alla ricezione di 3 ACK duplicati il mittente ritrasmette il pacchetto mancante senza aspettare lo scadere dell'RTO
      - Il mittente ritrasmette 1 solo dopo aver ricevuto 3 ACK duplicati per tale pacchetto (quindi solo dopo aver inviato 5 ed aver ricevuto 4 ACK identici)

Timer di ritrasmissione (4)

- Ma risposte Web caratterizzate da una dimensione media di 10 KB (anche se distribuzione *heavy-tailed*)
  - Maggior parte del trasferimento durante la fase di slow start del controllo di congestione



- Dopo un timeout di ritrasmissione viene ripetuta la fase di slow start



Slow-start restart

- Le connessioni persistenti aiutano ad evitare lo slow-start
  - Aumenta la dimensione della finestra di congestione
- Tuttavia, traffico Web caratterizzato da una raffica di richieste seguiti da periodi idle
  - Periodi idle: inattività dovuta al *think-time* dell'utente
- Il mittente TCP ripete la fase di slow-start dopo un periodo di inattività (*slow-start restart*)
  - Obiettivo: evitare di sovraccaricare la rete alla ripresa del trasferimento
  - *Es.*: 5 connessioni con un throughput pari al 20% della capacità del collegamento
  - 1 connessione inattiva: le altre 4 connessioni inviano dati con un throughput pari al 25%
  - Se la connessione inattiva riprende ad inviare pacchetti con un throughput pari al 20%, si ha 20% di traffico in più rispetto alla capacità del collegamento → perdita di pacchetti

Slow-start restart (2)

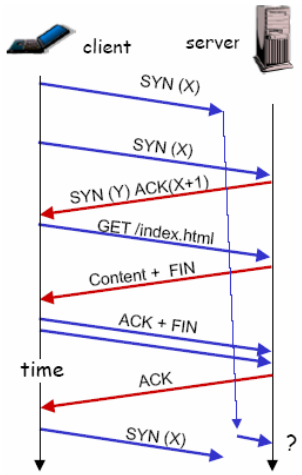
- Ripetizione della fase di slow-start dopo un periodo di inattività
  - Inizio del periodo di inattività
    - Dopo la ricezione dell'ultimo ACK
    - Timer basato sul valore corrente dell'RTO
  - La dimensione della finestra di congestione viene rinizializzata allo scadere del timer
    - Valore iniziale della finestra di congestione: uno o due pacchetti di dimensione massima
- Il meccanismo dello slow-start restart riduce il beneficio delle connessioni persistenti

Slow-start restart (3)

- Per ridurre il degrado delle prestazioni causato dallo slow-start restart sono state proposte diverse tecniche:
  - Disabilitare lo slow-start restart sul server
    - Si limita il rischio di una raffica inaspettata di richieste se il server chiude la connessione dopo un breve periodo di inattività
  - Usare un timeout maggiore per lo slow-start restart
    - Possibile inconsistenza tra dimensione della finestra di congestione e stato corrente della rete
  - Diminuire gradualmente la finestra di congestione
    - In proporzione alla durata del periodo di inattività
  - Mitigare la trasmissione dei pacchetti
    - Per evitare la generazione di raffiche di pacchetti

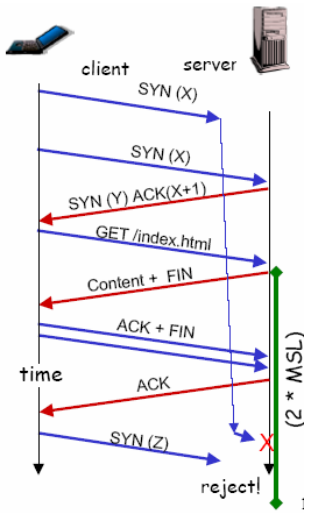
Problema dei duplicati vecchi

- In Internet:
  - Pacchetti duplicati arbitrariamente, ritardati e riordinati
  - Eventi non frequenti, ma che devono essere considerati (ad es. 5% di perdite)
- Esempio:
  - Connessioni tra due host, trasferimento di dati, chiusura della connessione
  - Il client inizia la nuova connessione usando la stessa tupla (indirizzi IP, numeri di porta)
  - Arriva un pacchetto duplicato dalla prima connessione
  - La prima connessione è stata chiusa, il suo stato perso
  - Come si può distinguere il pacchetto duplicato?



Ruolo dello stato TIME_WAIT

- Soluzione: per evitare di usare la stessa tupla, uno dei due host tiene traccia dell'esistenza della connessione precedente
  - Compito assegnato dal TCP all'host che inizia la chiusura della connessione (il server Web!)
  - Stato **TIME_WAIT** della connessione TCP di durata pari a 2 volte l'*MSL* (Maximum Segment Lifetime)
  - MSL pari a 2 minuti nell'RFC 1122
    - Quindi 4 minuti come durata del TIME_WAIT
  - In realtà, nelle varie implementazioni MSL pari a 30 sec, 1 o 2 minuti

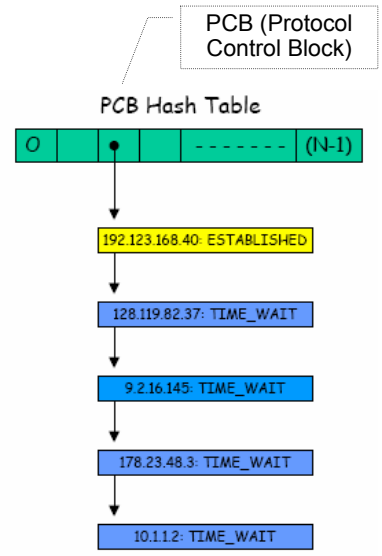


Problemi del TIME_WAIT sul server

- Nel Web è tipicamente il server a chiudere la connessione
  - Asimmetria del modello client-server: molti client
  - Maggior consumo di risorse sul server: incentivo a chiudere la connessione
  - Ma il server deve mantenere traccia della connessione per la durata dello stato TIME_WAIT
- Problema esistente anche sui proxy
- L'uso delle connessioni persistenti riduce l'entità del problema
  - Riduzione del numero di connessioni in TIME_WAIT
- Per visualizzare lo stato TIME_WAIT usare il comando **`netstat`**

Riduzione del TIME_WAIT sul server

- Due tecniche proposte per attenuare il problema del TIME_WAIT sul server
- Prima soluzione: ridurre la quantità di risorse di sistema usate per tenere traccia delle connessioni in TIME_WAIT
  - Il SO mantiene la minima quantità possibile di informazioni
  - Controllo periodico dei timer scaduti scandendo la lista delle connessioni TCP: connessioni in TIME_WAIT poste alla fine della lista
  - Si migliorano sensibilmente le prestazioni dei server soggetti ad un traffico elevato



Riduzione del TIME_WAIT sul server (2)

- Seconda soluzione: modificare le specifiche del protocollo HTTP
  - Forzare il client a chiudere la connessione
    - Ad es., introducendo un nuovo header di risposta inviato dal server al client
    - Obiettivo: spostare sul client la responsabilità di mantenere lo stato TIME_WAIT
  - Problema: il client non può avere incentivi a chiudere la connessione o ad assumersi la responsabilità dello stato TIME_WAIT

Stratificazione HTTP/TCP

- Funzioni implementate a livello del protocollo di trasporto possono avere un impatto significativo sulle prestazioni Web
- Interruzione di trasferimenti HTTP
  - Richiede la chiusura della connessione TCP sottostante
- Algoritmo di Nagle
  - Limita il numero di pacchetti piccoli trasmessi dal mittente TCP, ritardando il trasferimento dell'ultimo pacchetto del messaggio HTTP
- Acknowldegment ritardati
  - Il destinatario TCP può ritardare l'invio di un acknowldegment sperando di farne il piggybacking in un pacchetto di dati in uscita

Interruzione di trasferimenti HTTP

- Assenza di un meccanismo di interruzione precoce nell'HTTP
  - Richiede la terminazione della connessione TCP
- Effetto dell'operazione di interruzione sulle prestazioni Web
  - Terminare il trasferimento evita di caricare il resto (ad es. le immagini embedded nella pagina)
  - Il client deve ristabilire la connessione TCP per servire la prossima richiesta
  - Quali sono gli effetti collaterali?
  - Accoppiamento stretto tra richieste in pipeline sulla stessa connessione TCP
    - Interrompendo una richiesta, si interrompono tutte le richieste inviate in pipeline
  - I proxy richiedono di mantenere uno stato dettagliato
  - L'interruzione causata dall'utente non blocca immediatamente il trasferimento dei dati
  - Accoppiamento dei dati trasferiti tra proxy e client e tra proxy e server

Dettagli sull'interruzione: FIN

- Il client invia un segmento con impostato il flag FIN mediante la chiamata di sistema close()
- Alla ricezione del FIN, il SO invia l'EOF all'applicazione server
- Il SO continua ad inviare dati prelevandoli dal buffer di trasmissione
- L'EOF comporta che il server smetta di scrivere nuovi dati
- I dati nel buffer di invio saranno inviati al client

Dettagli sull'interruzione: RST

- Il client invia un segmento con impostato il flag RST (*reset*) mediante la close(), avendo prima impostato sul socket l'opzione SO_LINGER con setsockopt()
- Il SO scarta ogni dato in uscita rimanente per la connessione
  - Inclusi i dati nel buffer di invio
- Si evita il trasferimento di dati addizionali
- L'RST comporta che il SO scarti tutti i dati nel buffer di ricezione
- Evita al server il bisogno di gestire le richieste in pipeline
- Ma non è una chiusura pulita della connessione
- Non vi è ritrasmissione dei pacchetti persi

Algoritmo di Nagle

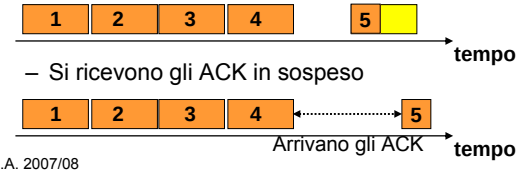
- TCP prevede un meccanismo di bufferizzazione dei dati in uscita per evitare la trasmissione di tanti piccoli segmenti con un utilizzo non ottimale della banda disponibile; inviando tanti segmenti piccoli:
  - Maggior consumo di banda (aumenta il rapporto header/ payload)
  - Maggior uso del processore (molti costi sono per pacchetto, non per byte)
- L'algoritmo di Nagle implementa questo meccanismo
  - I dati vengono accumulati fino a che non si raggiunge una dimensione sufficiente per eseguire la trasmissione di un singolo segmento (536 o 1460 B)

Algoritmo di Nagle (2)

- L'algoritmo prevede che il mittente non invii il prossimo segmento piccolo se è ancora in attesa di ricevere l'ACK di un segmento piccolo già inviato
- Conseguenza: il mittente deve accumulare i dati in spedizione fino a che sia soddisfatta una delle due seguenti condizioni:
  - La dimensione dei dati pronti per l'invio ha superato l'MSS
  - E' stato ricevuto l'ACK per tutti i segmenti piccoli in sospeso

Impatto sulle prestazioni Web

- Può avere un effetto negativo sulle connessioni persistenti se i trasferimenti Web richiedono il trasferimento di segmenti piccoli
  - Es.: il server trasmette la risposta HTTP scrivendo i dati sul socket con due chiamate di sistema separate per header e dati
  - Es: messaggio di 6000 B su una connessione con MSS pari a 1460 B
    - 4 segmenti da 1460 B e 1 segmento da 160 B
    - Il SO ritarda l'invio dell'ultimo segmento piccolo finché:
      - Si raggiunge la dimensione dell'MSS scrivendo dati addizionali



  - Si ricevono gli ACK in sospeso

Come disabilitare l'algoritmo di Nagle

- L'algoritmo di Nagle può essere disabilitato a livello di applicazione
- Si usa la funzione setsockopt() per settare l'opportuna opzione del socket (TCP_NODELAY)

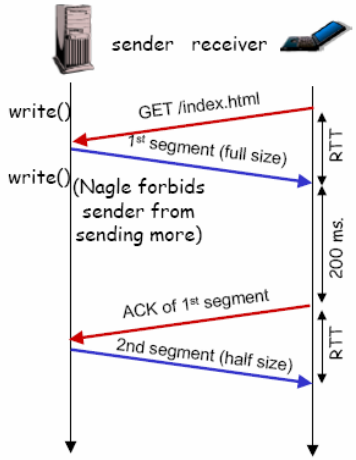
```
int one = 1;
setsockopt(sock, IPPROTO_TCP, TCP_NODELAY, &one, sizeof(one));
```


ACK ritardati

- Ritardando la trasmissione di un ACK, aumenta la probabilità di poterne fare il piggybacking in un pacchetto di dati
- Algoritmo delayed ACK
  - ACK inviati
    - Ogni due segmenti ricevuti
    - Oppure dopo 200 ms dalla ricezione di un singolo segmento
  - L'invio immediato di un ACK si ha solo per segmenti fuori sequenza

Interazione tra Nagle e ACK ritardato

- Nagle e ACK ritardato causano una situazione di deadlock temporaneo
  - Il mittente vuole inviare 1,5 segmenti, manda il primo segmento completo
  - Nagle impedisce l'invio del secondo segmento (non ha dimensione completa e il mittente attende di ricevere l'ACK del primo segmento)
  - Il mittente attente l'ACK ritardato dal destinatario
  - Il destinatario aspetta il secondo segmento per inviare l'ACK
- Soluzione: disabilitare Nagle



Overhead sul server Web

- Le operazioni effettuate dal server Web consumano risorse
  - Molte di queste operazioni sono legate al TCP
- Per ridurre in parte l'overhead sul server si possono combinare o eliminare alcune operazioni
- Stesse osservazioni valide anche per i proxy

- Consideriamo le operazioni per servire nello spazio utente una richiesta HTTP con metodo GET

```
initialize;
forever do {
  get request;
  process;
  send response;
  log request;
}
```

server in a nutshell

Preparare il server

```
s = socket();        /* allocate listen socket */
bind(s, 80);         /* bind to TCP port 80    */
listen(s);           /* indicate willingness to accept */
while (1) {
   newconn = accept(s);    /* accept new connection */
```

- Il server notifica al SO di essere interessato a servire richieste Web (HTTP sulla porta 80)
- Alloca il socket di ascolto e ne effettua il bind() alla porta 80
- Chiama listen() sul socket per indicare la sua volontà di ricevere richieste
- Può usare setsockopt() per disabilitare Nagle
- Chiama accept() per attendere una richiesta (e si blocca)
- Quando accept() ritorna, il nuovo socket di connessione rappresenta la nuova connessione di un client

Processare una richiesta

```
remoteIP = getsockname(newconn);
remoteHost = gethostbyname(remoteIP);
gettimeofday(currentTime);
read(newconn, reqBuffer, sizeof(reqBuffer));
reqInfo = serverParse(reqBuffer);
```

- getsockname() per ottenere il nome dell'host remoto
  - Per scopi di logging (opzionale)
- gethostbyname() per ottenere il nome dell'altro endpoint della connessione
  - Ancora per scopi di logging
- gettimeofday() per prendere l'ora della richiesta
  - Per l'header Date e per il logging
- read() chiamata sul socket di connessione per ottenere la richiesta
- La richiesta è individuata effettuando il parsing dei dati letti

Processare una richiesta (2)

```
fileName = parseOutFileName(requestBuffer);
fileAttr = stat(fileName);
serverCheckFileStuff(fileName, fileAttr);
open(fileName);
```

- stat() per controllare il path della risorsa
  - Per verificare se il file esiste ed è accessibile
- stat() usata anche per ottenere meta-dati sul file
  - Ad es., dimensione del file, last modified time
  - Può essere necessario stat() su file multipli
- Assumendo che sia tutto OK, open() per aprire il file

Rispondere alla richiesta

```
read(fileName, fileBuffer);
headerBuffer = serverFigureHeaders(fileName, reqInfo);
write(newSock, headerBuffer);
write(newSock, fileBuffer);
close(newSock);
close(fileName);
write(logFile, requestInfo);
```

- read() per leggere il file nello spazio utente
- write() per inviare gli header HTTP sul socket
  - I primi server chiamavano write() per *ogni header*!
- write() per scrivere il file sul socket
- close() per chiudere il socket
- close() per chiudere il descrittore del file aperto
- write() per scrivere sul file di log

Ottimizzazioni del server

- Esiste un elevato grado di località nelle richieste Web e nel traffico Web
- Una buona parte del lavoro visto per il server non deve essere eseguito *necessariamente* per ogni richiesta
- Due categorie di ottimizzazioni
  - Caching
  - Uso di primitive del SO che consentono di combinare più operazioni

Ottimizzazioni: caching

- Idea: sfruttare la località delle richieste
  - Molti file sono richiesti spesso (ad es., index.html)

- Perché aprire e chiudere gli stessi file? Conviene mettere in cache i descrittori dei file aperti e gestirli con politica LRU
- Perché chiamare stat() sugli stessi file? Meglio caching del path e delle caratteristiche del file
- Caching anche delle informazioni per costruire gli header HTTP, piuttosto che generare le stesse identiche informazioni per gli stessi file

```
fileDescriptor =
  lookInFDCache(fileName);
metaInfo =
  lookInMetaInfoCache(fileName);
headerBuffer =
  lookInHTTPHeaderCache(fileName);
```


Ottimizzazioni: caching (2)

- Invece di leggere e scrivere i dati, caching dei dati e dei meta-dati nello spazio utente

- In modo migliore: mmap() del file in modo tale che non esistano due copie (sia nello spazio utente sia nello spazio kernel)

```
fileData =
  lookInFileDataCache(fileName);
fileData =
  lookInMMapCache(fileName);
remoteHostName =
  lookRemoteHostCache(fileName);
```

- Anche caching delle risoluzioni inverse degli indirizzi IP dei client negli hostname corrispondenti
  - Ancora meglio: no risoluzione inversa ma logging dei soli indirizzi IP dei client

Ottimizzazioni: chiamate SO

- Invece di chiamare accept(), getsockname() e read(), usare, se disponibile, la nuova chiamata acceptExtended(), che combina le 3 chiamate

```
acceptExtended(listenSock,
        &newSock, readBuffer,
        &remoteInfo);
```

- Invece di chiamare gettimeofday(), usare un contatore mappato in memoria, che richiede un accesso più economico (alcune istruzioni invece di una chiamata di sistema)

```
currentTime = *mappedTimePointer;
```

- Invece di chiamare write() molte volte, usare writev()

  int writev(int fd, const struct iovec *vector, int count);

  – Permette l'implementazione dell'I/O vettorizzato, scrivendo più buffer con una sola chiamata

```
buffer[0] = firstHTTPHeader;
buffer[1] = secondHTTPHeader;
buffer[2] = fileDataBuffer;
writev(newSock, buffer, 3);
```


Ottimizzazioni: chiamate SO (2)

- Invece di chiamare read() e write() (o write() con un file mappato in memoria), usare la chiamata di sistema sendfile() (o transmitfile()) per copiare direttamente da un descrittore di file ad un altro
  - I byte rimangono nello spazio kernel, anziché essere copiati nello spazio utente per poi essere copiati nel buffer di invio
  - zerocopy: trasferimento diretto via DMA dal controller del disco alla scheda di rete
- Se possibile (no Linux), aggiungere l'opzione per chiudere la connessione
  - Si evita di dover chiamare close() esplicitamente

```
httpInfo = cacheLookup(reqBuffer);
sendfile(newConn,
        httpInfo->headers,
        httpInfo->fileDescriptor,
        OPT_CLOSE_WHEN_DONE);
```

- L'uso di queste chiamate richiede un **adeguato supporto** da parte del SO
  - Attualmente fornito da diversi SO (almeno in parte), tra cui Linux