Università degli Studi di Pisa

Facoltà di Scienze Matematiche Fisiche e Naturali
Corso di Laurea Specialistica in Tecnologie Informatiche

Tesi di Laurea

Analisi dell'avvio del TCP su canali satellitari a larga banda

Candidato
Giovanni Verrecchia

Relatore
Dott. Francesco Potortì

Controrelatore
Prof. Maurizio Bonuccelli

Anno Accademico 2004/2005

Alla mia famiglia,
che mi è stata sempre vicina
in questa esperienza

Sommario

I satelliti hanno un ruolo molto importante nelle comunicazioni mobili, nelle comunicazioni Internet, nella navigazione marittima, nelle trasmissioni televisive, nel controllo e nel comando militare. Inoltre, in molte situazioni costituiscono l'unico modo per comunicare.

Questa tesi si inserisce all'interno del progetto SatNEx, "European Satellite Communications Network of Excellence", il quale ha come obiettivo primario quello di raggiungere un'integrazione duratura della ricerca europea nell'ambito delle comunicazioni satellitari e di sviluppare una base di conoscenza comune.

Scopo della tesi è l'analisi simulativa del comportamento iniziale di una connessione TCP con diverse varianti del protocollo TCP, lo studio della letteratura sull'argomento, la descrizione del fenomeno, la verifica utilizzando un canale satellitare reale.

Indice

Capitolo 1 Introduzione

I satelliti hanno un ruolo molto importante nelle comunicazioni mobili, nelle comunicazioni Internet, nella navigazione marittima, nelle trasmissioni televisive, nel controllo e nel comando militare. Inoltre, in molte situazioni costituiscono l'unico modo per comunicare.

L'Europa, in questo settore, è stata sempre molto attiva con i programmi di ricerca e sviluppo dell'European Space Agency (ESA), i programmi quadro dell'Unione Europea (EU) e le azioni di cooperazione europea nel campo della ricerca scientifica e tecnologica (COST).

Il progetto SatNEx, "European Satellite Communications Network of Excellence", nato nel 2004, ha come obiettivo primario quello di raggiungere un'integrazione duratura della ricerca europea nell'ambito delle comunicazioni satellitari e di sviluppare una base di conoscenza comune. Attraverso la cooperazione delle università e degli istituti di ricerca, SatNEx costruirà un centro virtuale europeo di eccellenza nelle comunicazioni satellitari e contribuirà alla realizzazione dell'area di ricerca europea (ERA). Partecipano attivamente al progetto 22 partner, tra i quali i membri del gruppo Wireless Network Laboratory dell'istituto ISTI del CNR di Pisa.

Questa tesi si inserisce all'interno di tale progetto e pone l'attenzione sullo studio del protocollo TCP nelle comunicazioni satellitari.

1.1 Problematiche affrontate

L'argomento centrale di questo lavoro è l'analisi del comportamento iniziale (slow start) di una connessione TCP su un canale satellitare geostazionario a larga banda. Per sfruttare al meglio la banda di questo canale, è necessario che la finestra di trasmissione sia pari almeno al prodotto banda*ritardo, dove per ritardo si intende il tempo intercorrente fra la trasmissione di un pacchetto e la ricezione dell'Ack relativo. Su un canale geostazionario, tale ritardo è dell'ordine di mezzo secondo. Nella maggior parte dei sistemi, la dimensione della finestra consente velocità di trasmissione inferiori a 1 Mbit/s. E' possibile ovviare a ciò ricorrendo all'opzione TCP

window scale sui due estremi della connessione e configurando le applicazioni affinché utilizzino finestre sufficientemente grandi.

Operando in condizioni ideali, cioè senza errori e con finestre sufficientemente grandi, il comportamento del TCP durante la fase di avvio è talvolta disastroso e può entrare in uno stato, che può protrarsi per diversi minuti, in cui la velocità di trasmissione è ridotta ad un pacchetto per RTT. Questo si verifica in corrispondenza della perdita di un numero elevato di segmenti: ad una brevissima fase nella quale sono trasmessi sia segmenti vecchi che nuovi (grazie alla notevole quantità di Ack duplicati ricevuti), ne segue un'altra nella quale il trasmettitore invia un solo segmento per RTT quando usa l'algoritmo NewReno, o un blocco di segmenti perduti quando usa il Sack. Gli unici Ack ricevuti, infatti, sono quelli parziali, che informano ciascuno della perdita di un solo segmento (o di un blocco di segmenti). Secondo il principio di conservazione del pacchetto, il trasmettitore può inviare in risposta all'Ack parziale (o al blocco Sack) ricevuto un solo segmento (o un blocco di segmenti) senza aggravare la situazione di congestione della rete. Questo comportamento anomalo del TCP è analizzato in prima istanza con analisi simulative e successivamente mediante prove dirette su di un canale satellitare reale.

Il simulatore ns-2, ricreando uno scenario molto simile al caso reale, rende possibile un'analisi simulativa completa e dettagliata dei principali algoritmi di controllo della congestione del TCP: Reno, NewReno e Sack. Circa il secondo algoritmo, il documento di riferimento ([RFC3782]) introduce due diverse varianti: la prima, chiamata Slow but Steady, in cui ad ogni Ack parziale si riarma il timer di ritrasmissione; la seconda, chiamata Impatient, in cui il riarmo avviene solo per il primo Ack parziale.

Nelle classi del simulatore che implementano un trasferimento dati bidirezionale (classe `FullTcpAgent` e le sue sottoclassi) è presente soltanto l'implementazione della prima variante, la quale è causa del comportamento anomalo accennato poc'anzi. L'assenza della variante Impatient richiede l'intervento sul codice sorgente del simulatore e la sua modifica. La validazione dei cambiamenti apportati al codice è condotta mediante la costruzione di scenari ad hoc nei quali si confrontano la versione originale del simulatore e quella modificata.

Contestualmente a questa modifica, è considerato anche il problema dell'aggiornamento della finestra di congestione in seguito ad un timeout. RFC 3782 e simulatore ns-2 operano in modo diverso: nel primo, l'arrivo di Ack duplicati non concorre all'aumento della finestra di congestione; nel secondo, per ogni Ack duplicato successivo al terzo si opera l'incremento. Anche qui è proposta una modifica del simulatore, per aggiungere la modalità di aggiornamento dettata da RFC 3782.

Come nel caso precedente, i cambiamenti apportati al simulatore sono soggetti ad un successivo controllo mediante la creazione di scenari ad hoc e la verifica dei risultati ottenuti.

Gli ultimi due punti trattati in questa tesi riguardano lo studio di alcuni stack TCP e la verifica dei risultati dell'analisi simulativa mediante esperimenti su un canale satellitare reale. Visto che FreeBSD e Linux sono open source, è possibile ispezionare il codice sorgente dei rispettivi kernel al fine di individuare quali sono i meccanismi di controllo della congestione adottati. Gli esperimenti su un canale satellitare sono svolti in ambiente Linux, utilizzando un sistema di accesso al satellite chiamato Skyplex. Presso l'istituto ISTI del CNR di Pisa è disponibile, a scopo di sperimentazione, un canale satellitare la cui larghezza di banda nominale è pari a 2 Mbit/s e il ritardo è di circa 250 ms. I due algoritmi di controllo della congestione sottoposti al confronto sono il NewReno e il Sack.

1.2 Organizzazione della relazione

La presente tesi è organizzata come segue. Nel capitolo 2 sono illustrate le nozioni base sul protocollo TCP e le sue principali varianti. Nel capitolo 3 è introdotto il simulatore ns-2 utilizzato per l'analisi simulativa. Il capitolo 4 discute e giustifica le modifiche apportate al codice del simulatore relative all'algoritmo NewReno. Nel capitolo 5 sono commentati i risultati delle simulazioni del comportamento iniziale del TCP con le principali varianti del protocollo (Reno, NewReno e Sack). Il capitolo 6 descrive i meccanismi di controllo della congestione utilizzati da FreeBSD e Linux, importanti ai fini della comprensione dei risultati degli esperimenti su un canale satellitare riportati nel capitolo 7. Nell'ultimo capitolo, infine, vengono tratte le conclusioni del lavoro di tesi.

Capitolo 2 Il protocollo TCP

2.1 Introduzione

Il TCP (Transmission Control Protocol) è un protocollo del livello trasporto progettato per fornire un flusso di byte affidabile, da sorgente a destinazione, su una rete non affidabile.

Il protocollo TCP è definito formalmente in RFC 793 [RFC793]. Con il passare degli anni, sono stati scoperti errori ed incoerenze, eliminati con RFC successivi: una versione sostanzialmente definitiva è contenuta in RFC 1122 [RFC1122] e alcune estensioni sono definite in RFC 1323 [RFC1323] e in altri successivi.

Anche se TCP e UDP usano lo stesso network layer (IP), il primo protocollo offre al livello applicativo un servizio totalmente differente dall'altro. Il TCP infatti offre un servizio "connection-oriented" e affidabile tra coppie di processi. Con il termine "connection-oriented" si intende che due applicazioni che fanno uso del TCP (denominate usualmente client e server) devono stabilire una connessione prima che possa avvenire lo scambio di dati.

L'affidabilità nel trasporto di dati è garantita dal TCP nel seguente modo:

- I dati provenienti dal livello applicativo sono suddivisi in blocchi di dimensione stabilita dal protocollo TCP. L'unità di informazione trasmessa al protocollo IP è chiamata segmento. Questa pacchettizzazione è invisibile all'applicazione, che vede la connessione come un flusso di byte non strutturato.
- Quando il TCP manda un segmento, mantiene un timer, in attesa che l'altro lato invii una conferma di ricezione ("positive Ack") del segmento. Se l'Ack non è ricevuto prima che scada il timer, viene ritrasmesso il segmento.
- Quando il TCP riceve un segmento dall'altro lato della connessione, invia un Ack. Di solito l'Ack non è inviato immediatamente, ma è ritardato di una certa frazione di secondi.
- Viene calcolato e trasmesso un checksum per ogni segmento. Lo scopo è di individuare eventuali modifiche dei dati che transitano sulla rete. Se arriva a de-

stinazione un segmento con checksum non valido, il TCP lo scarta e non invia alcun Ack.

- Poiché i segmenti TCP sono incapsulati nei datagram IP e poiché questi possono arrivare non in ordine, anche i segmenti TCP possono arrivare fuori ordine. Il lato ricevente del TCP deve riordinare i dati ricevuti, in modo da inoltrare al livello superiore i dati nell'ordine corretto.
- Poiché i datagram IP possono essere duplicati, il ricevitore TCP deve scartare i segmenti duplicati.
- Il TCP effettua il recupero di dati danneggiati, persi, duplicati o consegnati fuori ordine assegnando un numero di sequenza ad ogni byte trasmesso. I numeri di sequenza sono usati dal ricevitore per ordinare correttamente i segmenti fuori ordine ed eliminare gli eventuali duplicati.
- Il TCP definisce anche un meccanismo di controllo del flusso. Entrambi i lati di una connessione TCP hanno un buffer di dimensioni finite. Il ricevitore ha la possibilità di controllare la quantità di dati mandati dal trasmettitore. Questo è ottenuto restituendo con ogni Ack una “finestra” che indica il numero di byte che il ricevitore è in grado di memorizzare.

Per permettere a più processi nello stesso host di utilizzare il protocollo TCP, il TCP introduce il concetto di porta. La concatenazione di un indirizzo IP e di un numero di porta costituisce un socket. Una coppia di socket identifica univocamente una connessione.

I meccanismi di affidabilità e controllo del flusso appena descritti richiedono che il TCP inizializzi e mantenga alcune informazioni di stato per ogni stream di dati. La combinazione di queste informazioni, dei socket, dei numero di sequenza e delle dimensioni delle finestre è chiamata connessione. Ogni connessione è specificata dalla coppia di socket che identificano i due lati.

2.2 Struttura dello header TCP

La figura 2.1 mostra il formato dello header TCP. Lo header è di 20 byte, se non sono presenti opzioni, come frequentemente avviene.

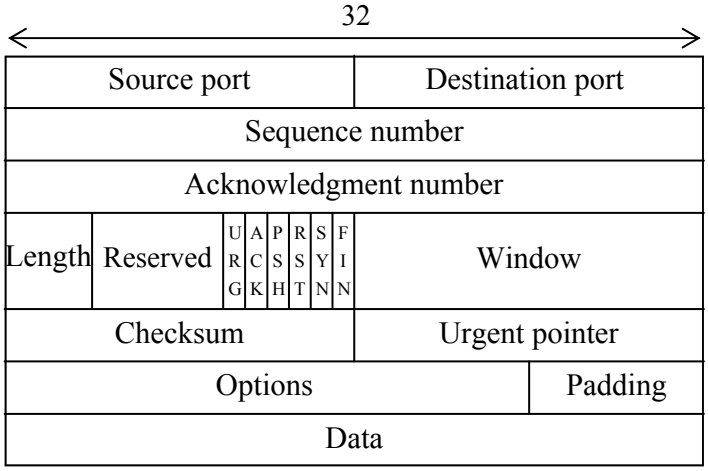

Figura 2.1. Struttura header TCP.

Ogni segmento TCP contiene il numero di porta sorgente e destinazione che sono usati per il multiplexing/demultiplexing dei dati da/verso le applicazioni del livello superiore. Avendo a disposizione 16 bit per indicare il numero di porta, si possono avere 65536 possibilità: i numeri di porta minori di 1024 corrispondono alle cosiddette "well-known port" e sono associati a particolari servizi, alcuni dei quali sono elencati in tabella 2.1:

Numero di porta	**Servizio**
20	FTP (Data)
21	FTP (Control)
53	DNS
80	HTTP
389	LDAP
443	HTTPS

Tabella 2.1. Corrispondenza tra numeri di porta e servizi.

Il numero di sequenza identifica i byte nello stream di dati tra il trasmettitore e il ricevitore. Concettualmente ad ogni ottetto di dati è assegnato un numero di sequenza. Il numero di sequenza del primo ottetto di dati in un segmento è riportato nel campo Sequence Number dello header TCP ed è chiamato numero di sequenza del segmento.

Poiché ogni byte di dati scambiato è numerato, il campo Acknowledgment Number contiene il prossimo numero di sequenza che il mittente dell'Ack si aspetta di ricevere. In effetti questo è il numero di sequenza dell'ultimo byte di dati ricevuto correttamente, più uno. Il TCP fornisce un servizio full-duplex al livello applicativo. Questo significa che i dati possono fluire in entrambe le direzioni e i due estremi di

una connessione devono mantenere entrambi i numeri di sequenza dei dati che fluiscono nelle due direzioni.

Il campo Length specifica la lunghezza dello header TCP in parole di 32 bit.

Il campo Flag contiene 6 bit.

- Il bit URG è usato per indicare che nel segmento ci sono dati che l'entità del livello superiore ha contrassegnato come "urgenti". La dislocazione dell'ultimo byte di questi dati urgenti può essere individuato sommando al campo Sequence Number il valore contenuto nel campo "urgent pointer". La modalità "urgent mode" è una tecnica che ha il trasmettitore per inviare dati urgenti all'altro lato.
- Il bit ACK è usato per indicare se il valore riportato nel campo Acknowledgment Number è valido. E' sempre attivato, tranne che nel primo segmento usato per aprire una connessione.
- Il bit PSH indica la presenza di dati che devono essere consegnati all'applicazione destinataria senza aspettare che si riempia il buffer del ricevitore.
- Il bit RST indica la richiesta del mittente di abbattimento della connessione.
- Il bit SYN è usato per aprire una connessione. Se tale bit è uguale a 1, nel campo Sequence Number è presente il numero di sequenza iniziale.
- Il bit FIN è usato per chiudere una connessione, in quanto il mittente non ha ulteriori dati da spedire.

Il controllo del flusso è di tipo "a finestra scorrevole": il campo Window indica il numero di byte, a partire da quello specificato nel campo Acknowledgment Number, che il ricevitore è disposto ad accettare. E' un campo di 16 bit, che limita la finestra a 65535 byte.

Il campo Checksum è utilizzato per verificare dal lato ricevitore la presenza di errori nel segmento. Per il calcolo del valore del campo checksum, si considera innanzitutto la lunghezza del payload (espressa in numero di byte): se è un numero dispari, si aggiungono in coda 8 bit posti a 0. Successivamente si calcola il complemento a 1 della somma di tutte le parole di 16 bit del segmento. Il calcolo del checksum include anche uno pseudoheader di 96 bit, la cui struttura è mostrata in figura 2.2.

Lo pseudoheader non è qualcosa che viene trasmesso unitamente al segmento TCP. I suoi campi hanno le seguenti funzioni:

- Source IP address, destination IP address: indirizzi IP sorgente e destinazione;
- Protocol: il codice numerico del protocollo TCP (=6);
- TCP Length: il numero di byte del segmento TCP, header incluso.

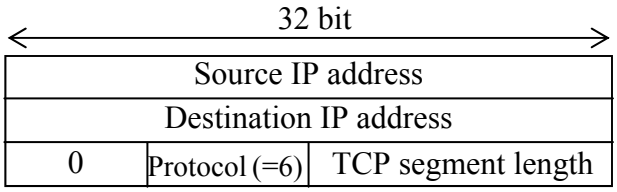

Figura 2.2. Struttura pseudoheader TCP.

L'inclusione dello pseudoheader nel calcolo del checksum è utile per individuare segmenti consegnati alla destinazione errata (misrouted). Per verificare la correttezza dei dati ricevuti, il ricevitore calcola il checksum sull'intero segmento. Se il risultato è diverso da 0, il segmento è corrotto e viene scartato.

Il campo Options, facoltativo e di lunghezza variabile, è utilizzato per specificare servizi aggiuntivi non compresi nello header TCP. Ad esempio l'opzione MSS (Maximum Segment Size), opzione che permette di stabilire la dimensione massima dei segmenti durante la fase di apertura di una connessione. Le opzioni possono essere specificate in due diversi formati:

1. un singolo ottetto per indicare il tipo dell'opzione;
2. un ottetto per indicare il tipo dell'opzione, un ottetto per la lunghezza dell'opzione e un certo numero di ottetti per i dati dell'opzione.

Tipicamente il campo Options è vuoto, dunque la lunghezza dello header è di 20 byte.

Il campo Padding è usato per assicurare che lo header abbia una dimensione che sia un multiplo di 32 bit. Tale campo è composto da bit uguali a 0.

Infine il campo Data (payload) può anche essere vuoto. Questo tipicamente avviene per i segmenti scambiati durante l'apertura e la chiusura di una connessione TCP e per i cosiddetti "Ack puri".

2.3 Apertura e chiusura di una connessione TCP

Il TCP è un protocollo "connection-oriented". Prima che le due entità possano scambiarsi dati, deve essere creata una connessione. Per stabilire una connessione TCP:

- Passo 1. Il TCP del lato client per primo invia un segmento al TCP del lato server. Questo segmento non ha alcun payload e ha il bit SYN posto a 1. Per questo motivo tale segmento è comunemente noto come segmento SYN. Il client, inoltre, sceglie un initial sequence number (client_ISN) e lo inserisce nel campo Sequence Number del segmento SYN.

- Passo 2. Quando il TCP del lato server riceve un segmento SYN, costruisce ed invia un segmento contenente tre informazioni importanti: il bit SYN settato a 1, il campo Acknowledgment Number posto a client_ISN +1 e il campo Sequence Number posto a server_ISN, ovvero l'initial sequence number scelto dal server. Tale segmento è noto come segmento SYNACK.
- Passo 3. Quando il TCP del lato client riceve un segmento SYNACK, costruisce ed invia un segmento nel quale il campo Sequence Number è uguale a client_ISN+1 e il campo Acknowledgment Number è posto a server_ISN+1. Il bit SYN è posto a 0, in quanto la connessione è stabilita. A seconda delle implementazioni, questo segmento può contenere o meno dati.

Questi tre segmenti completano l'apertura di una connessione. Questa procedura è nota come "three-way handshake" ed è rappresentata graficamente nella figura 2.3. L'entità che manda il segmento SYN effettua una "active open". L'altra, la quale riceve il segmento SYN e manda il segmento SYNACK, effettua una "passive open".

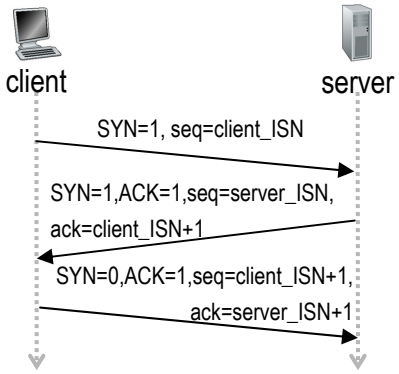

Figura 2.3. Three-way handshake: scambio di segmenti.

Un aspetto molto importante del three-way handshake riguarda la scelta dell'initial sequence number. Quando nuove connessioni sono create, un generatore di ISN è impiegato per selezionare l'ISN di 32 bit. Il generatore è legato ad un clock a 32 bit in cui il bit di ordine più basso è incrementato ogni 4 microsecondi. In questo modo si ha un ciclo completo dell'ISN ogni 4,55 ore. Poiché si può assumere che i segmenti permangono nella rete per un intervallo di tempo non superiore al Maximum Segment Lifetime (MSL) e che MSL è minore di 4,55 ore, si può concludere che l'ISN è unico per ogni quadrupla srcIP, srcPort, dstIP, dstPort. Lo scopo di questi ISN è impedire ai pacchetti che sono ritardati nella rete di essere consegnati successivamente e interpretati erroneamente come parte di una connessione esistente.

E' possibile, sebbene improbabile, che due applicazioni contemporaneamente eseguano una active open. Questo è noto come open simultanea. Il protocollo TCP è

stato progettato per gestire le open simultanee e la regola generale è che venga creata una sola connessione e non due.

Mentre l'apertura di una connessione richiede tre segmenti, la chiusura ne richiede quattro. Questo è causato dallo half-close del TCP[1]. Poiché una connessione TCP è full-duplex (i dati, cioè, possono fluire indipendentemente in entrambe le direzioni), entrambe le direzioni devono essere chiuse. Solitamente ogni lato della connessione invia un segmento FIN (in cui il bit FIN è uguale a 1) quando ha terminato di mandare dati. Quando un TCP riceve un segmento FIN, deve inviare una notifica all'applicazione che l'altro estremo ha terminato il flusso di dati in quella direzione. L'invio di un segmento FIN è di solito il risultato dell'invocazione della primitiva `close`.

La ricezione di un segmento FIN indica solamente che non ci sono altri dati che fluiscono in quella direzione. Un'entità può ancora mandare dati dopo aver ricevuto un segmento FIN. Il lato che per primo invoca la primitiva close effettua una active close mentre l'altro lato effettua una passive close.

La procedura di chiusura di una connessione ha inizio nel momento in cui una delle due entità invia un segmento FIN. L'altra entità, ricevuto tale segmento, invia indietro un Ack, informando il livello superiore della chiusura della connessione. La connessione viene chiusa dall'altro lato inviando un segmento FIN, al quale seguirà un Ack.

Possono presentarsi tre diversi casi:

1. il TCP locale effettua la chiusura: in questo caso è costruito un segmento FIN e posto nella coda dei segmenti di uscita. L'invio di nuovi segmenti non sarà più possibile. Tutti e soli i segmenti che precedono il segmento FIN saranno ritrasmessi finché non si riceverà un Ack. Il TCP locale entra nello stato FIN-WAIT-1 in attesa di ricevere il segmento FIN dall'altro lato, così come di un Ack per il FIN inviato. La figura 2.4 mostra la sequenza di segmenti scambiati.
2. Il TCP locale riceve un segmento FIN: invia in risposta un Ack e notifica al livello superiore che la connessione sta per essere chiusa. Quando il livello superiore risponde con una `close`, viene inviato anche il segmento FIN. Il TCP locale a questo punto resta in attesa di un Ack dall'altro lato.

[1] Il TCP prevede la possibilità per una delle due entità di terminare l'invio di dati, continuando a ricevere comunque dati dall'altra entità.

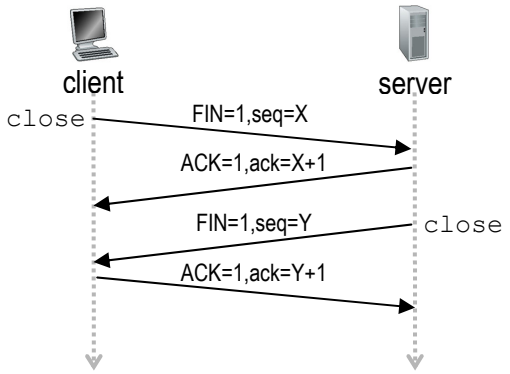

Figura 2.4. Chiusura di una connessione (caso 1).

3. Entrambe le entità chiudono la connessione simultaneamente: questo causa l'invio per entrambi di un segmento FIN. Quando tutti i segmenti che precedono il segmento FIN sono stati processati e riscontrati, entrambe le entità possono inviare un Ack per il segmento FIN ricevuto. Ricevuti questi Ack, la connessione viene cancellata. La figura 2.5 descrive questo caso.

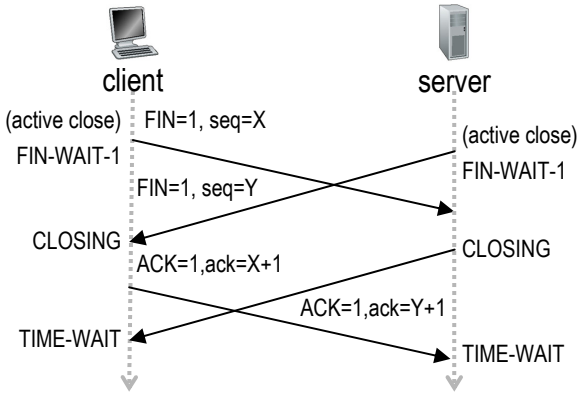

Figura 2.5. Chiusura di una connessione (caso 3).

2.4 Diagramma di transizione di stato del TCP

Una connessione TCP avanza attraverso una serie di stati nell'arco della sua vita. Gli stati sono: LISTEN, SYN-SENT, SYN-RECEIVED, ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT e lo stato fittizio CLOSED. Lo stato CLOSED è fittizio perchè rappresenta lo stato nel quale non esiste la connessione.

Le transizioni di stato sono riportate nel diagramma 2.1. La prima cosa da notare in questo diagramma è che un sottoinsieme delle transizioni di stato è "tipico". Nel

diagramma le transizioni del lato client sono rappresentate mediante una linea continua, le transizioni del lato server con una linea tratteggiata.

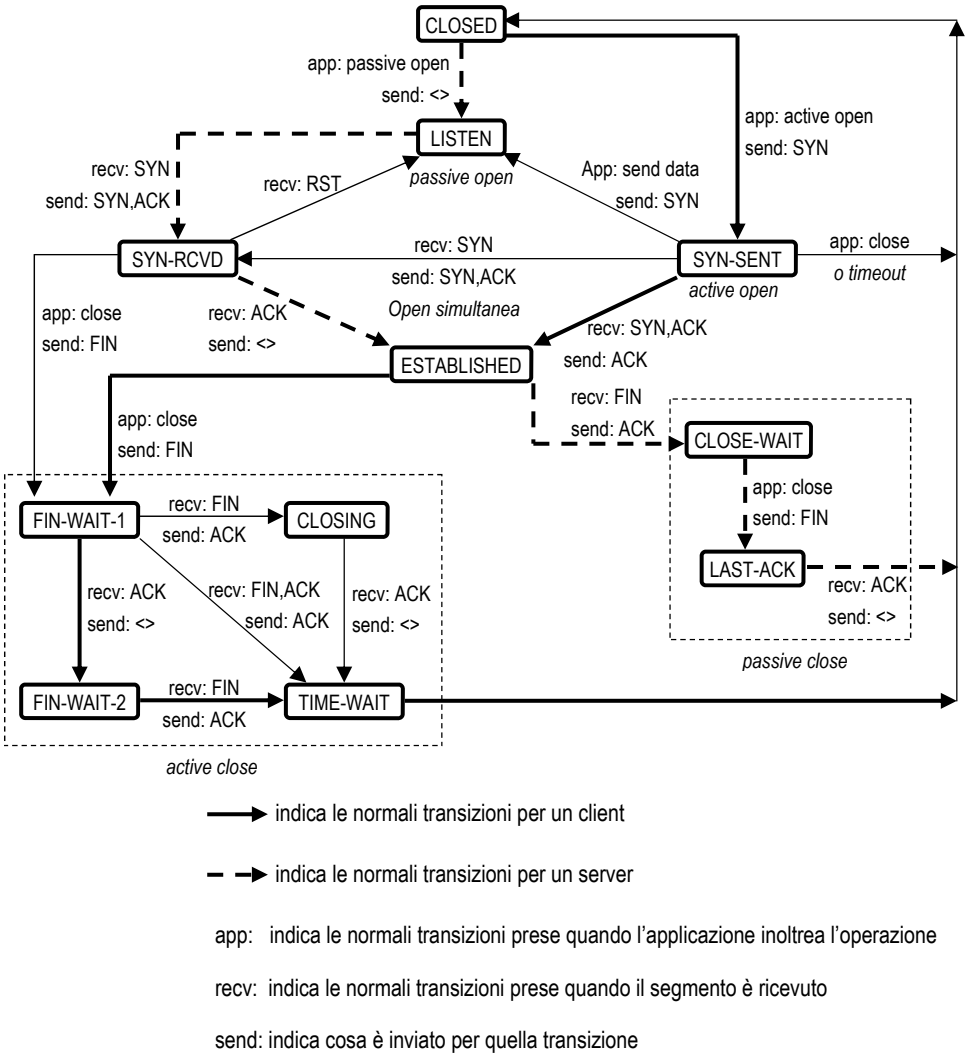

Diagramma 2.1. Transizioni di stato del TCP.

Le due transizioni che portano allo stato ESTABLISHED corrispondono all'apertura di una connessione, mentre le due transizioni che partono da questo stato sono per la chiusura di una connessione. Tale stato rappresenta una connessione aperta, nella quale può avvenire il trasferimento di dati tra le due entità e in modalità full-duplex.

I quattro stati riportati nell'angolo in basso a sinistra del diagramma sono raggruppati assieme ed etichettati come "active close". I due stati nell'angolo a destra (CLOSE-WAIT e LAST-ACK) sono raggruppati assieme ed etichettati come "passive close".

La transizione dallo stato SYN-RECEIVED allo stato LISTEN è valida solo se in precedenza si era effettuata la transizione dal secondo al primo stato (scenario normale) e non a partire dallo stato SYN-SENT (open simultanea). Questo significa che

se si effettua una passive open (si entra nello stato LISTEN), si riceve un segmento SYN, si invia un segmento SYNACK (entrando nello stato SYN-RECEIVED), e poi si riceve un segmento RST invece di un Ack, l'entità ritorna allo stato LISTEN e attende l'arrivo di un'altra richiesta di connessione.

Lo stato TIME-WAIT è anche chiamato stato di attesa di 2*MSL secondi. Ogni implementazione del TCP sceglie un valore per il Maximum Segment Lifetime, cioè l'intervallo massimo di tempo per cui un qualsiasi segmento può rimanere nella rete prima di essere scartato.

La figura 2.6 mostra la normale apertura e chiusura di una connessione TCP, riportando in dettaglio i differenti stati attraverso i quali i lati client e server passano e assumendo che il client (lato sinistro) effettui una active open mentre il server (lato destro) faccia una passive open.

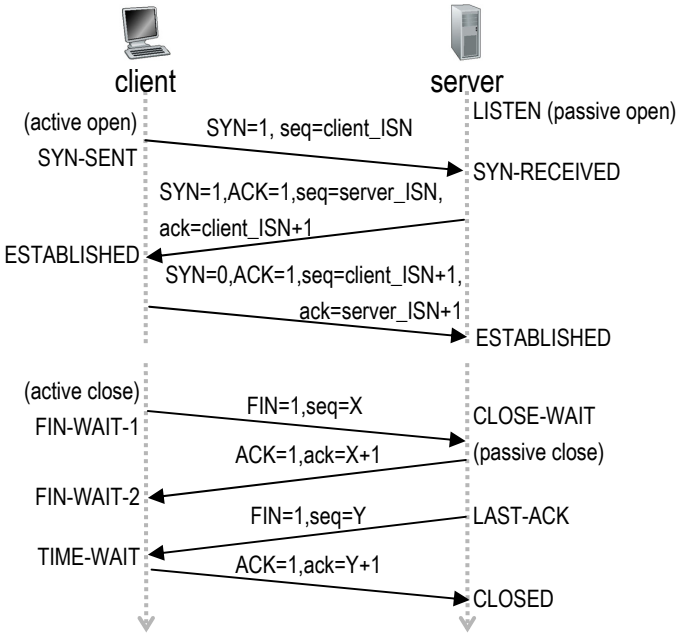

Figura 2.6. Apertura e chiusura di una connessione con indicazione degli stati corrispondenti.

Nel caso in cui sia il client che il server effettuino nello stesso istante una active open (open simultanea), le transizioni di stato differiscono da quelle mostrate nella figura precedente. Le due entità inviano un segmento SYN contemporaneamente, entrando nella stato SYN-SENT. Quando ognuno riceve un segmento SYN, lo stato cambia in SYN-RECEIVED e ognuno manda un segmento SYNACK. Ricevuto quest'ultimo segmento, lo stato cambia in ESTABLISHED. Questi cambiamenti di stato sono mostrati in figura 2.7.

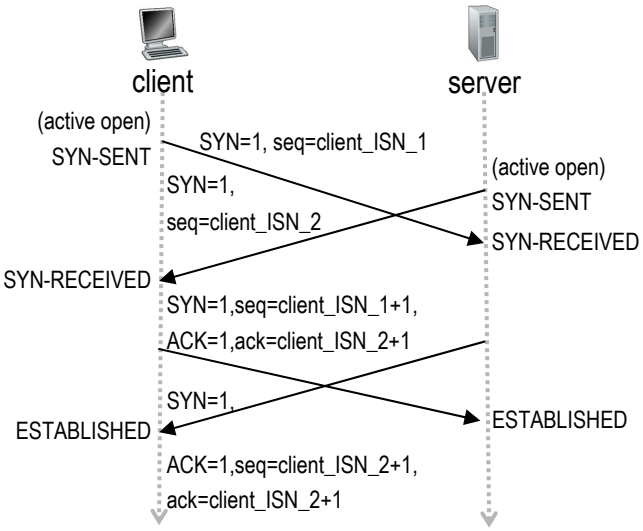

Figura 2.7. Scambio di segmenti in una open simultanea.

2.5 Il controllo del flusso nel TCP

Il protocollo TCP usa una forma particolare di controllo del flusso chiamata protocollo a finestra scorrevole (sliding window). Permette al trasmettitore di inviare contemporaneamente più segmenti prima di fermarsi e aspettare un Ack. Questo implica un trasferimento dati più veloce, in quanto il trasmettitore non deve bloccarsi e attendere un Ack per ogni segmento inviato. Il protocollo sliding window può essere visualizzato nella figura 2.8.

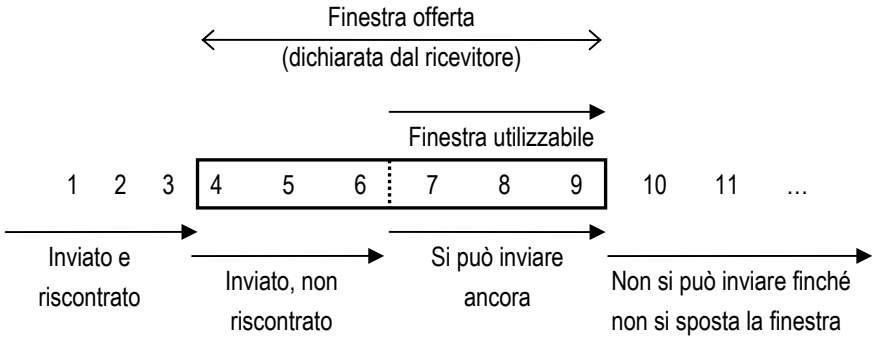

Figura 2.8. Il protocollo sliding window.

In questa figura i byte sono numerati da 1 a 11. La finestra dichiarata dal ricevitore è chiamata finestra offerta (offered window) e include i byte dal 4 al 9, indicando con questo che il ricevitore ha riscontrato tutti i byte fino a e incluso il byte numero 3, notificando una finestra di dimensione 6. Il trasmettitore calcola la sua finestra utilizzabile, la quale indica quanti dati può inviare immediatamente.

Questa finestra scorrevole si sposta verso destra quando il ricevitore invia un Ack. Il movimento relativo delle due estremità della finestra fa aumentare o diminuire la dimensione della finestra stessa. Tre termini sono usati per descrivere il movimento delle estremità destra e sinistra della finestra.

1. La finestra si chiude quando l'estremità sinistra si sposta verso destra. Questo accade quando il dato è inviato e subito riscontrato.
2. La finestra si apre quando l'estremità destra si sposta verso destra, permettendo ad altri dati di essere trasmessi. Questo accade quando il processo ricevente legge i dati riscontrati, liberando spazio nel suo buffer di ricezione.
3. La finestra si riduce quando l'estremità destra si sposta verso sinistra. Questo è un comportamento anomalo e scoraggiato, ma il TCP deve far fronte ad una simile situazione.

Se l'estremità sinistra raggiunge quella destra, si ha la cosiddetta "zero window". Questa impedisce al trasmettitore di inviare dati.

Si consideri, per esempio, un trasferimento di 5 kbyte dallo host H1 allo host H2 (come mostrato in figura 2.9).

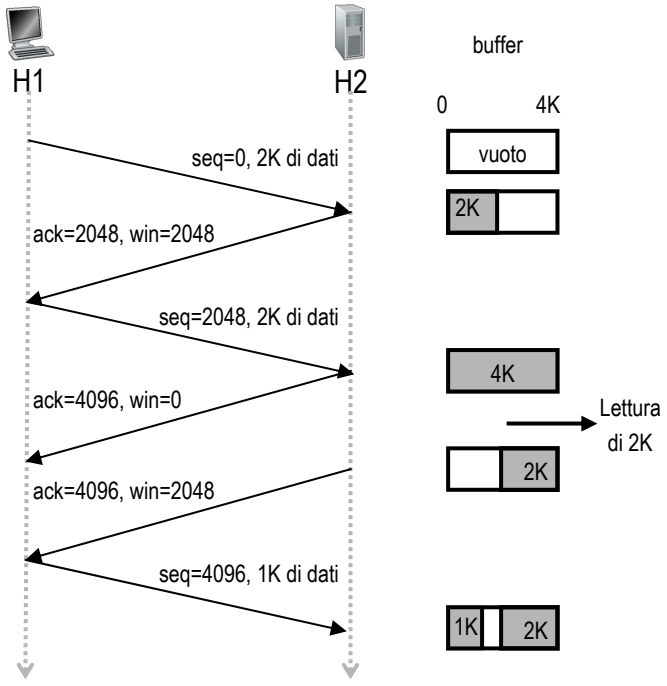

Figura 2.9. Esempio di controllo del flusso TCP.

Inizialmente lo host H2 ha un buffer di 4 Kbyte completamente vuoto (lato destro della figura precedente). Se il mittente trasmette un segmento di dimensione 2 Kbyte e questo viene correttamente ricevuto, il ricevente invia un Ack (con numero di Ack pari a 2048) e, dato che il suo buffer possiede ancora 2 Kbyte liberi, dichiara una

finestra di ricezione di 2 Kbyte (segmento 2). Il mittente può trasmettere al più 2 Kbyte di nuovi dati: il buffer del ricevitore diventa pieno e, insieme alla conferma di ricezione, dichiara anche una zero window (segmento 4). In questo caso il mittente è costretto a fermarsi, in attesa di ricevere dall'altro lato un segmento di aggiornamento della finestra (in cui il campo Window Size è diverso da 0).

Ad ogni modo, anche se il mittente riceve un segmento con il campo Window Size nullo, può comunque inviare dati urgenti. Potrebbe capitare che il segmento nel quale è presente un aggiornamento della finestra vada perso, per cui il mittente sia erroneamente obbligato ad attendere. Questo potrebbe portare ad una situazione di stallo. Per evitarla, parte un timeout dopo l'invio del segmento con window size diverso da 0 e si ricorre ad un segmento chiamato sonda.

2.6 Timeout e ritrasmissioni

Il protocollo TCP fornisce un livello di trasporto affidabile. Ogni entità che comunica mediante il TCP deve inviare un riscontro per i dati ricevuti dall'altro lato. Ma segmenti e Ack possono andare persi. Il TCP gestisce questa situazione impostando un timer quando sono inviati i dati; se un segmento non è riscontrato prima che scada il timer, viene ritrasmesso.

A causa della variabilità delle reti e dei numerosi possibili usi di una connessione TCP, il timeout di ritrasmissione (RTO) deve essere dinamicamente determinato. Tutte le procedure di determinazione del RTO si basano sul valore del RTT[2] misurato.

Una prima semplice procedura, riportata in RFC 793, fa uso di una quantità chiamata "Smoothed Round Trip Time" (SRTT), calcolata nel seguente modo:

```
SRTT = (ALPHA*SRTT) + (1 – ALPHA)* RTT
```

dove `ALPHA` è un fattore di smoothing con un valore tipico pari a 0,9.
`SRTT` è poi usato per calcolare il RTO:

```
RTO = BETA*SRTT
```

dove `BETA` è un fattore di sicurezza con un valore pari a 2.

Questa procedura non tiene conto delle oscillazioni del RTT, causando inutili ritrasmissioni. Nel novembre del 2000 è stato pubblicato RFC 2988 [RFC2988], "Computing TCP's Retransmission Timer", nel quale è proposta una procedura di-

[2] Il Round-Trip Time è l'intervallo di tempo che intercorre tra l'invio di un byte di dati e la ricezione di un riscontro per quei dati.

versa di calcolo, alla base della quale vi sono gli studi compiuti da Van Jacobson nel 1988 [Jac88]. Per calcolare RTO, il trasmettitore TCP ha bisogno di due variabili di stato, SRTT (Smoothed Round-Trip Time) e RTTVAR (RTT Variation). Si assume, inoltre, una granularità del clock pari a G secondi. Alla prima misurazione R del RTT, il trasmettitore imposta

```
SRTT = R
RTTVAR = R/2
RTO = SRTT + max(G, K * RTTVAR)
```

dove K = 4

Per la generica misurazione R' del RTT, il trasmettitore effettua i seguenti calcoli:

```
RTTVAR = (1 - beta) * RTTVAR + beta * |SRTT - R'|
SRTT = (1 - alpha) * SRTT + alpha * R'
```

Dove alpha = 1/8 e beta = 1/4

Il valore di SRTT usato nel calcolo di RTTVAR è il valore che ha al passo precedente. Il valore di RTO è aggiornato nel modo seguente:

```
RTO = SRTT + max (G, K*RTTVAR)
```

Nel caso in cui RTO sia minore di 1 s, viene arrotondato a 1 s.

2.7 Il controllo della congestione nel TCP

Effettuato il three-way handshake, due entità possono inserire più segmenti nella rete, fino alla dimensione della finestra dichiarata dal ricevitore. Questo è corretto se le due entità appartengono alla stessa rete; se ci sono dei router e dei link più lenti tra trasmettitore e ricevitore, si possono avere dei problemi. Per questo motivo, il protocollo TCP implementa due algoritmi: Slow Start e Congestion Avoidance.

L'implementazione di questi due algoritmi richiede due variabili che vanno aggiunte a quelle necessarie per il mantenimento dello stato di una connessione. La finestra di congestione (cwnd) costituisce un limite dal lato trasmettitore della quantità di dati che il trasmettitore può inviare prima di ricevere un Ack, mentre la finestra dichiarata dal ricevitore (rwnd) costituisce un limite dal lato ricevitore del numero di dati ricevibili. Il minimo tra queste due quantità governa la trasmissione dei dati. Vedremo nel seguito che il prodotto bandwidth-delay costituisce una quantità significa-

tiva nel meccanismo di controllo di congestione del TCP. In particolare, il lato ricevitore deve disporre di un buffer la cui dimensione sia almeno pari al prodotto bandwidth-delay per poter sfruttare appieno la banda disponibile sul link. Un'altra variabile di stato, slow start threshold (ssthresh), è necessaria per determinare se è utilizzato l'algoritmo Slow Start oppure Congestion Avoidance per controllare la trasmissione di dati.

Per poter avviare una trasmissione di dati in una rete dalle caratteristiche sconosciute, il TCP deve determinare la capacità disponibile della rete, al fine di evitare un congestionamento dovuto ad un'eccessiva quantità di dati inviati. L'algoritmo Slow Start è adottato all'inizio di un trasferimento oppure dopo aver recuperato una perdita di segmenti evidenziata dal timer di ritrasmissione.

Secondo RFC 2581 [RFC2581], il valore iniziale della finestra di congestione deve essere non superiore a 2 segmenti (vedi sezione 3.1 di [RFC2581]). Il valore di slow start threshold può essere arbitrariamente elevato, ma può essere ridotto in risposta alla congestione. L'algoritmo Slow Start è usato quando `cwnd < ssthresh`, mentre l'algoritmo Congestion Avoidance è usato quando `cwnd > ssthresh`. Quando ssthresh è uguale a cwnd, il trasmettitore può usare indistintamente uno dei due algoritmi.

Durante lo Slow Start, il TCP incrementa la finestra di congestione di un segmento per ogni Ack ricevuto che riscontra nuovi dati. Lo Slow Start termina quando la finestra di congestione supera ssthresh o quando si osserva una situazione di congestione.

Durante la Congestion Avoidance, la finestra di congestione è incrementata di un segmento per round-trip time (RTT). La Congestion Avoidance continua finché non è individuata una situazione di congestione. Una formula comunemente usata per aggiornare il valore della finestra di congestione è la seguente:

```
cwnd += 1/cwnd
```

Si tratta di un incremento lineare nel tempo della finestra di congestione, se confrontato all'incremento esponenziale fatto nell'algoritmo Slow Start.

Quando il trasmettitore TCP, mediante il timer di ritrasmissione, si accorge della perdita di segmenti, il valore di ssthresh deve essere impostato alla metà del valore corrente della finestra di congestione, e la finestra di congestione ad un segmento. Dopo aver ritrasmesso il segmento perso, il trasmettitore TCP usa l'algoritmo Slow Start.

La figura 2.10 è una descrizione visuale degli algoritmi Slow Start e Congestion Avoidance. Per comodità, cwnd e ssthresh sono mostrate in unità di segmenti, mentre in realtà sono espresse in byte.

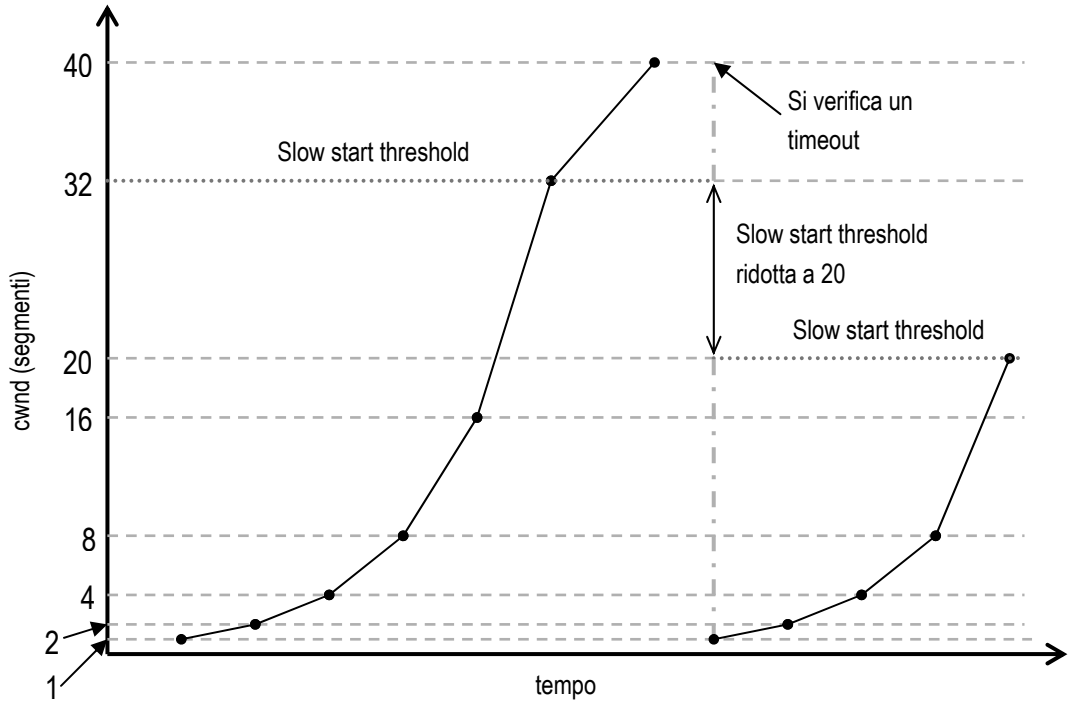

Figura 2.10. Esempio di controllo della congestione TCP.

Nella figura si assume che la congestione si sia avuta quando cwnd era di 40 segmenti. ssthresh è dunque settata a 20 segmenti e cwnd a un segmento. Per ogni Ack successivo ricevuto, cwnd è incrementata di un segmento fino al raggiungimento di ssthresh. Da quel punto in poi l'incremento della finestra di congestione è lineare nel tempo.

2.8 Alcune varianti del protocollo TCP

Le prime versioni del protocollo TCP commercializzate da Berkeley Software Distribution (la 4.2 BSD nel 1983 e la 4.3 BSD nel 1986) si sono dimostrate affidabili, ma incapaci di fornire prestazioni accettabili se utilizzate in reti di grandi dimensioni. Sono stati apportati notevoli miglioramenti e ad oggi sono note diverse varianti del protocollo TCP: Reno, NewReno, Sack e Westwood.

2.8.1 TCP Reno

In generale, quando arriva a destinazione un segmento fuori ordine, il ricevitore TCP deve inviare un Ack duplicato. Lo scopo di questo Ack è di informare il trasmettitore che è stato ricevuto un segmento fuori ordine e quale numero di sequenza ci si aspettava. Dalla prospettiva del trasmettitore, la ricezione di Ack duplicati può essere dovuta a numerosi problemi di rete. Primo fra tutti, la perdita di segmenti: in questo caso, tutti i segmenti che arrivano dopo quello perso provocano l'invio di Ack

duplicati. Secondo, gli Ack duplicati possono essere dovuti al riordinamento dei segmenti nella rete. Terzo ed ultimo, gli Ack duplicati possono essere dovuti alla replicazione di Ack o segmenti dati nella rete.

La variante Reno del TCP è descritta in dettaglio nella sezione 3.2 di RFC 2581 ed è stata implementata per la prima volta nel 1990 nella release BSD Reno.

Nel TCP Reno, il trasmettitore TCP fa uso di un algoritmo noto con il nome di "Fast Retransmit" per individuare le perdite di segmenti e porvi rimedio, basandosi sugli Ack duplicati in arrivo. L'algoritmo di Fast Retransmit fa ricorso all'arrivo di tre Ack duplicati (cioè quattro Ack identici) come indicazione della perdita di uno o più segmenti. Dopo aver ricevuto tre Ack duplicati, il trasmettitore TCP effettua la ritrasmissione di ciò che sembra essere il segmento mancante, senza attendere che scada il timer di ritrasmissione.

Una volta inviato il segmento mancante, l'algoritmo di "Fast Recovery" controlla la trasmissione di nuovi dati fino alla ricezione del primo Ack non duplicato. La ragione per cui non si utilizza l'algoritmo Slow Start è insita nel doppio significato degli Ack duplicati: questi non solo indicano che un segmento si è perso, ma anche che altri segmenti sono stati inseriti nel buffer del ricevitore e quindi non si trovano più nella rete. Avendo lasciato la rete, non consumano più spazio nei buffer dei router, e si può quindi trasmettere ancora senza aggravare la situazione di congestione della rete.

Gli algoritmi di Fast Retransmit e Fast Recovery sono implementati insieme come segue.

1. Quando il trasmettitore riceve il terzo Ack duplicato, ssthresh è posta alla metà del valore corrente della finestra di congestione.
2. Si ritrasmette il segmento perso e cwnd è posta a ssthresh + 3*SMSS. Questo aumenta la finestra di congestione del numero di segmenti (tre) che hanno lasciato la rete e sono stati bufferizzati dal ricevitore.
3. Per ogni ulteriore Ack duplicato ricevuto, cwnd è incrementa di SMSS byte, in modo da tener conto del segmento addizionale che ha lasciato la rete.
4. Si trasmette un nuovo segmento, se permesso dal nuovo valore di cwnd e della finestra dichiarata dal ricevitore (rwnd).
5. Ricevuto un Ack non duplicato (un Ack che riscontra nuovi dati), cwnd è posta pari al valore di ssthresh.

L'Ack appena ricevuto non è solamente il riscontro per il segmento ritrasmesso al punto 2, un round-trip time dopo la sua ritrasmissione, ma anche il riscontro per tutti i segmenti inviati dopo quello perso e prima della ricezione del terzo Ack duplicato.

Questa variante non ha buone prestazioni in scenari nei quali all'interno di una finestra di dati si perdono più segmenti. Alcune modifiche per risolvere questo pro-

blema sono parte integrante della variante NewReno e sono discusse in RFC 3782 [RFC3782].

2.8.2 TCP NewReno

Nella variante Reno descritta nel paragrafo precedente, si hanno dei problemi quando si perdono più segmenti all'interno di una finestra di dati e sono invocati gli algoritmi di Fast Retransmit e Fast Recovery.

In questo caso il trasmettitore dispone di poche informazioni sui segmenti persi. Dalla ricezione di tre Ack duplicati, deduce che un segmento è perso e lo ritrasmette. Il trasmettitore potrebbe, poi, ricevere ulteriori Ack duplicati, poiché il ricevitore riscontra dei segmenti che erano nella rete quando il trasmettitore è entrato nella procedura di Fast Retransmit. La prima nuova informazione disponibile al trasmettitore è costituita dalla ricezione dell'Ack per il segmento ritrasmesso. Se si è verificata la perdita di un singolo segmento, allora l'Ack ricevuto riscontra tutti i segmenti trasmessi prima di entrare nel Fast Retransmit. Nel caso di perdita di più segmenti, l'Ack ricevuto riscontra solo alcuni dei segmenti trasmessi e non tutti. Questo Ack è chiamato "Ack parziale".

La variante NewReno, descritta in RFC 3782, propone alcune modifiche rispetto alla variante precedente, riguardanti essenzialmente la procedura di Fast Recovery, che ha inizio con la ricezione di tre Ack duplicati e termina o a causa di un timeout o con la ricezione di un Ack che riscontra tutti i segmenti pendenti quando è iniziata la procedura stessa.

NewReno differisce dall'implementazione presente in RFC 2581 (vedi paragrafo 2.8.1) nell'introduzione della variabile `recover_` nel passo 1, nella risposta agli Ack parziali e agli Ack completi nel passo 5, nelle modifiche al passo 1 e nell'aggiunta del passo 6 per evitare ritrasmissioni veloci causate dalla ritrasmissione di segmenti già ricevuti dal ricevitore.

I passi dell'algoritmo sono:

1. Quando il trasmettitore riceve tre Ack duplicati e non è già nella procedura di Fast Recovery, controlla se il campo Acknowledgment Number è maggiore di `recover_`. Se sì, salta al passo 1A, altrimenti al passo 1B.

1A. Il valore di ssthresh è posto alla metà del valore corrente della finestra di congestione. Si registra il più alto numero di sequenza trasmesso nella variabile `recover_`, continuando al passo 2.

1B. Non si entra nella procedura di Fast Retransmit e Fast Recovery. In particolare, non si modifica il valore di ssthresh, non si va al passo 2 per ritrasmettere il segmento perso, non si va al passo 3 in seguito alla ricezione di altri Ack duplicati.

2. Si ritrasmette il segmento perso e si pone cwnd a ssthresh + 3*SMSS. Questo aumenta la finestra di congestione del numero di segmenti (tre) che hanno lasciato la rete e sono stati bufferizzati dal ricevitore.
3. Per ogni ulteriore Ack duplicato ricevuto durante la procedura di Fast Recovery, si incrementa cwnd di SMSS byte, per tener conto del segmento addizionale che ha lasciato la rete.
4. Si trasmette un nuovo segmento, se permesso dal nuovo valore della finestra di congestione e della finestra dichiarata dal ricevitore (rwnd).
5. Ack completo
   Se l'Ack riscontra tutti i segmenti fino a e incluso quello con numero di sequenza pari al valore della variabile `recover_`, allora si pone cwnd pari al valore di ssthresh. Si esce dalla procedura di Fast Recovery.

   Ack parziale
   Se questo Ack non è un riscontro per i segmenti con numero di sequenza pari al valore della variabile `recover_`, allora è un Ack parziale. Si pone cwnd uguale al valore di ssthresh e si trasmette il primo segmento non riscontrato. Se il nuovo valore di cwnd lo permette, si può trasmettere un nuovo segmento. Non si esce dalla procedura di Fast Recovery.
6. Dopo un timeout di ritrasmissione, si registra nella variabile `recover_` il più alto numero di sequenza trasmesso e si esce dalla procedura di Fast Recovery. In questo modo, se dopo il timeout il trasmettitore riceve degli Ack duplicati con numero di Ack minore di `recover_`, non entra nuovamente in Fast Recovery.

La variante NewReno descrive due differenti comportamenti in risposta ad un Ack parziale: nel primo (variante Slow but Steady) il riarmo del timer di ritrasmissione è fatto per ogni Ack parziale; nel secondo (variante Impatient) è fatto solo per il primo Ack parziale. Nella variante Impatient, se all'interno di una finestra di dati sono andati persi molti segmenti, il timer di ritrasmissione ad un certo punto scade e il trasmettitore TCP invoca l'algoritmo Slow Start. Nella variante Slow but Steady se N segmenti sono andati persi in una finestra di dati, il trasmettitore resta in Fast Recovery per N round-trip time, ritrasmettendo un solo segmento per ogni round-trip time.

Le implicazioni relative alla scelta di una delle due varianti sopra menzionate sono descritte nel capitolo 4.

2.8.3 TCP Sack

La perdita di molti segmenti all'interno di una finestra di dati può avere un effetto catastrofico sul throughput del TCP. Il TCP adotta uno schema di Ack cumulativo, nel quale i segmenti ricevuti che non sono all'estremità sinistra della finestra di ricezione non sono stati riscontrati. Nell'algoritmo NewReno, questo costringe il trasmettitore o ad attendere un round-trip time per ottenere informazioni su ciascun segmento perso, o a ritrasmettere inutilmente segmenti che in realtà sono stati ricevuti correttamente.

Selective Acknowledgment (Sack) è una tecnica che risolve questo problema. Il ricevitore può informare il trasmettitore su quali segmenti sono stati ricevuti con successo, in modo tale che quest'ultimo invia soltanto quelli persi.

L'estensione Sack fa uso di due opzioni TCP.

1. Opzione "Sack-permitted": può essere inclusa solo in un segmento SYN dal ricevitore, indicando che questo è abilitato a ricevere e processare opportunamente l'opzione Sack e indica che tale opzione può essere usata una volta che la connessione è creata. Il formato di questa opzione consiste di soli due byte: il primo indica il tipo (=4), il secondo la lunghezza in byte (=2).
2. Opzione "Sack": è utilizzata per trasmettere informazioni estese di Ack dal ricevitore al trasmettitore. Più precisamente, per informare il trasmettitore di blocchi di dati contigui che sono stati ricevuti e accodati. Ogni blocco contiguo di dati è definito in questa opzione mediante due interi di 32 bit: il primo (`Left Edge of Block`) indica il primo numero di sequenza del blocco, il secondo (`Right Edge of Block`) è il numero di sequenza immediatamente successivo all'ultimo di quel blocco. I byte immediatamente precedenti al blocco (`Left Edge of Block - 1`) e quelli oltre il blocco (`Right Edge of Block`) non sono stati ricevuti.

Quando il trasmettitore riceve un Ack contenente un'opzione Sack, memorizza le informazioni in esso contenute. Si può assumere che il trasmettitore abbia una coda di ritrasmissione che contiene i segmenti che sono stati inviati ma non ancora riscontrati, ordinati per numero di sequenza. Per ognuno di questi segmenti vi è un flag, "Sacked", usato per indicare se quel segmento è stato riportato in un'opzione Sack. Il trasmettitore imposta ad uno i flag dei segmenti che sono stati selettivamente riscontrati. I segmenti in cui il flag non è ad uno saranno ritrasmessi.

Per chiarire quanto detto poc'anzi, si considerino tre scenari nei quali il trasmettitore invia otto segmenti di 500 byte di dati ciascuno e l'estremo sinistro della finestra è uguale a 5000.

Caso 1: i primi quattro segmenti sono ricevuti e gli ultimi quattro sono persi. Il

ricevitore invia un Ack con numero di Ack pari a 7000 senza far uso di opzioni Sack.

Caso 2: il primo segmento è perso mentre gli altri sette sono ricevuti. Alla ricezione degli ultimi sette segmenti, il ricevitore invia un Ack con numero di Ack pari a 5000 e contenente l'opzione Sack come riportato in tabella 2.2:

Arrivo del segmento	**Ack**	**Left Edge**	**Right Edge**
5000	(perso)		
5500	5000	5500	6000
6000	5000	5500	6500
6500	5000	5500	7000
7000	5000	5500	7500
7500	5000	5500	8000
8000	5000	5500	8500
8500	5000	5500	9000

Tabella 2.2. Blocchi Sack riportati nello scenario 2.

Caso 3: il secondo, il quarto, il sesto e l'ottavo segmento sono persi. Il ricevitore riscontra il primo segmento normalmente. Il terzo, il quinto e il settimo segmento causano l'invio di un'opzione Sack come indicato nella tabella 2.3:

		Primo blocco		**Secondo blocco**		**Terzo blocco**	
Arrivo del segmento	**Ack**	**Left Edge**	**Right Edge**	**Left Edge**	**Right Edge**	**Left Edge**	**Right Edge**
5000	5500						
5500	(perso)						
6000	5500	6000	6500				
6500	(perso)						
7000	5500	7000	7500	6000	6500		
7500	(perso)						
8000	5500	8000	8500	7000	7500	6000	6500
8500	(perso)						

Tabella 2.3. Blocchi Sack riportati nello scenario 3.

Per i dettagli implementativi dell'algoritmo Sack, si faccia riferimento al paragrafo 5.4.

Capitolo 3 Il simulatore ns-2

3.1 Introduzione

ns-2 è un simulatore di reti ad eventi discreti, il quale supporta simulazioni dei protocolli UDP e TCP (in tutte le sue varianti), routing e protocolli di multicast per reti wired e wireless.

E' nato nel 1989 come variante del simulatore REAL, ma negli ultimi anni ha subito numerose modifiche ed è tuttora in fase di sviluppo. Dello sviluppo si occupano i ricercatori del VINT Project, un progetto che vede la collaborazione di UC Berkeley, LBL, USC/ISI, e Xerox PARC, e supportato da DARPA (Defense Advanced Research Projects Agency).

3.2 Descrizione del simulatore

ns-2 è un simulatore object oriented, scritto in C++, con un interprete OTcl[3] come frontend. Il simulatore supporta una gerarchia di classi in C++ (chiamata anche gerarchia compilata) e una gerarchia simile di classi nell'interprete OTcl (chiamata anche gerarchia interpretata). Le due gerarchie sono correlate l'una all'altra e dalla prospettiva dell'utente c'è una corrispondenza uno a uno tra una classe nella gerarchia interpretata e una nella gerarchia compilata. La radice di questa gerarchia è la classe TclObject. Gli utenti creano nuovi oggetti del simulatore attraverso l'interprete; questi oggetti sono instanziati nell'interprete e associati ad un corrispondente oggetto nella gerarchia compilata. La gerarchia di classi interpretata è instaurata automaticamente mediante appositi metodi definiti nella classe `TclClass`. Gli oggetti instanziati dall'utente sono associati mediante metodi definiti nella classe `TclObject`.

Perchè due linguaggi? ns-2 usa due linguaggi perchè il simulatore deve svolgere due compiti molto diversi. Da una parte, le simulazioni dettagliate dei protocolli richiedono un linguaggio di programmazione di sistema il quale possa in maniera efficiente manipolare byte, pacchetti, header e implementare algoritmi che operano su un grande insieme di dati. Per questi compiti, la velocità a run-time è importante, mentre

[3] versione object oriented di Tcl, Tool Command Language.

il tempo di sviluppo è meno importante. Dall'altra parte, la ricerca sulle reti riguarda soprattutto la variazione di parametri e configurazioni e l'esplorazione di un certo numero di scenari. In questi casi, il tempo di iterazione (modifica del modello e nuova esecuzione della simulazione) è più importante.

ns-2 è in grado di svolgere entrambi i compiti sopra menzionati con due linguaggi, C++ e OTcl. C++ è veloce nell'esecuzione ma più lento da modificare, adatto per l'implementazione dettagliata dei protocolli. OTcl è molto più lento nell'esecuzione ma può essere modificato molto velocemente (e interattivamente), particolarmente indicato per la configurazione delle simulazioni. Attraverso tclcl (Tcl with classes), un'interfaccia Tcl/C++, ns-2 realizza il legame tra gli oggetti e le variabili che compaiono nei due linguaggi.

Avendo a disposizione due linguaggi, sorge la domanda su quale linguaggio debba essere utilizzato. Il suggerimento offerto dagli sviluppatori di ns-2 è di usare OTcl per configurazioni e setup e di usare C++ nel caso si voglia fare qualcosa che richieda il processing dei pacchetti in un flusso o se si voglia cambiare il comportamento di una classe C++. Ad esempio se si vuol costruire una nuova disciplina di accodamento o un nuovo modello di perdita dei pacchetti, occorre usare C++.

3.3 Definizione di uno scenario di simulazione

Lo scenario di una simulazione è descritto in ns-2 mediante uno script OTcl. I passi per la definizione di uno scenario sono i seguenti:

- creazione dello scheduler degli eventi;
- creazione della topologia della rete;
- definizione del traffico;
- inserimento di eventuali moduli di errore;
- definizione della traccia della simulazione.

3.3.1 Scheduler degli eventi

Il simulatore nel suo complesso è descritto dalla classe OTcl `Simulator`. Essa fornisce un insieme di interfacce per configurare una simulazione e per scegliere il tipo di scheduler degli eventi. Uno script di simulazione in genere inizia creando un'istanza di questa classe e chiamando vari metodi per aggiungere nodi, definire topologie e configurare altri aspetti della simulazione.

Quando un nuovo oggetto `Simulator` è creato in Tcl, la procedura di inizializzazione si preoccupa di inizializzare il formato dei pacchetti, di creare uno scheduler, di creare un "null agent" (utile come sink per pacchetti persi o come destinazione per pacchetti che non sono registrati).

Attualmente in ns-2 sono disponibili quattro diversi tipi di scheduler, ognuno implementato usando una differente struttura dati: una linked-list semplice, uno heap, una calendar queue (default) e un tipo speciale chiamato “real time”. Lo scheduler seleziona il primo degli eventi, lo esegue fino al completamento e ritorna ad eseguire il prossimo. L’unità di tempo usata dallo scheduler è il secondo. Il simulatore è single-threaded e si può avere un solo evento in esecuzione in un dato istante. Se sono schedulati più eventi da eseguire nello stesso istante, la loro esecuzione è fatta secondo la modalità “first scheduled-first dispatched”.

Qui di seguito è riportata una lista dei comandi del simulatore comunemente usati negli script di simulazione:

`set ns_ [new Simulator]`
crea un’istanza dell'oggetto `Simulator` e la assegna alla variabile `ns_`.

`set now [$ns_ now]`
Lo scheduler tiene traccia del tempo in una simulazione nella variabile `Simulator now`. Questa istruzione assegna il tempo alla variabile `now`.

`$ns_ halt`
Ferma lo scheduler.

`$ns_ run`
Fa partire lo scheduler.

`$ns_ at <time> <event>`
Schedula un `<event>` (il quale è una porzione di codice) per essere eseguita all’istante `<time>`.

`$ns_ after <delay> <event>`
Schedula un `<event>` per essere eseguito dopo un lasso di tempo pari a `<delay>`.

3.3.2 Definizione della topologia della rete

La topologia della rete, in ns-2, è definita indicando nodi, link e code che costituiscono lo scenario di simulazione. La primitiva di base per creare un nodo e assegnargli un identificativo è:

```
set node1 [$ns_ node]
```

La struttura tipica di un nodo (unicast) è mostrata in figura 3.1. Questa semplice struttura consiste di due oggetti Tcl: un classificatore dell'indirizzo (`classifier_`)

e un classificatore di porta (`dmux_`). La funzione di questi classificatori è di distribuire i pacchetti in ingresso agli agenti appropriati o ai link di uscita.

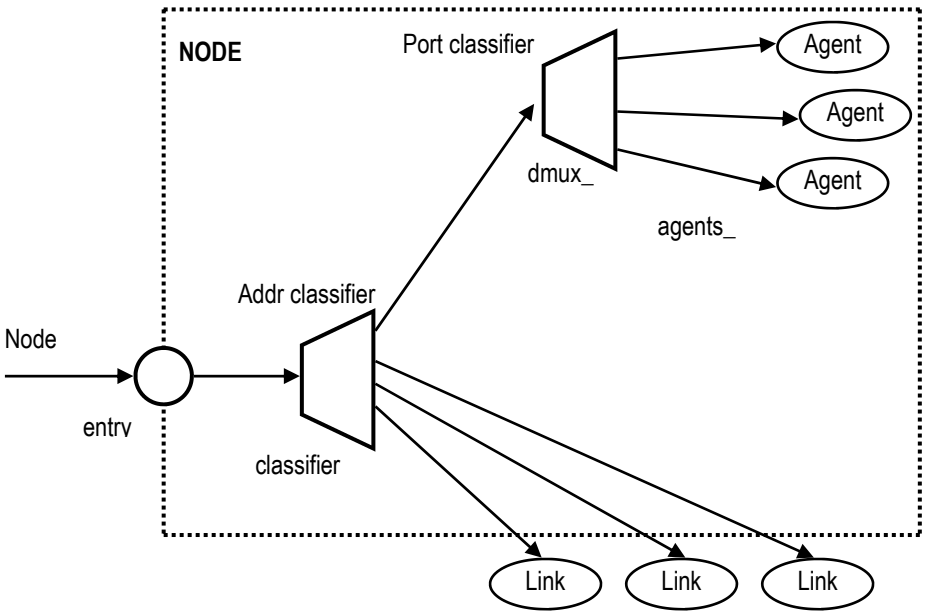

Figura 3.1. Struttura di un nodo unicast. `entry_` è solo una etichetta anziché un oggetto reale, come ad esempio `classifier_`.

Per default, i nodi in ns-2 sono costruiti per simulazioni unicast. Per effettuare simulazioni multicast, è necessario creare un'istanza dell'oggetto `Simulator` con l'opzione `-multicast on`.

Ad ogni nodo viene assegnato un indirizzo di 16 bit, in cui gli 8 bit più significativi definiscono l'identificatore di nodo, mentre gli altri 8 bit servono per distinguere i vari agenti che si possono porre su un nodo. Dunque è possibile creare una topologia che contiene al massimo $2^8 = 256$ nodi. In realtà è possibile creare anche una topologia più ampia, espandendo lo spazio degli indirizzi con il comando `Node expandaddr`. Nelle simulazioni multicast, il bit più significativo di un indirizzo è posto a 0.

Due nodi possono essere collegati mediante un link. In ns-2, la classe `Link` mette a disposizione un insieme di semplici primitive per connettere due nodi con link punto punto unidirezionali e bidirezionali.
La sintassi per creare un semplice link bidirezionale è:

```
$ns_ duplex-link node0 node1 bandwidth delay queue_type
```

Il comando crea un link dal nodo node0 al nodo node1, con banda `bandwidth` e ritardo `delay`. Il link usa una coda di tipo `queue_type`. (Le code saranno approfondite più avanti). La procedura aggiunge anche un controllore del Time-To-Live (TTL) al link. Cinque variabili di istanza definiscono il link:

- `head_`: entry point al link, punta al primo oggetto nel link;
- `queue_`: riferimento all'elemento coda del link;
- `link_`: riferimento all'elemento che modella il link, in termini delle caratteristiche di ritardo e larghezza di banda del link;
- `ttl_`: riferimento all'elemento che manipola il ttl in ogni pacchetto;
- `drophead_`: riferimento ad un oggetto che è la testa di una coda di elementi che processano le perdite del link.

Nella figura 3.2 è riportata la struttura di un link unidirezionale.

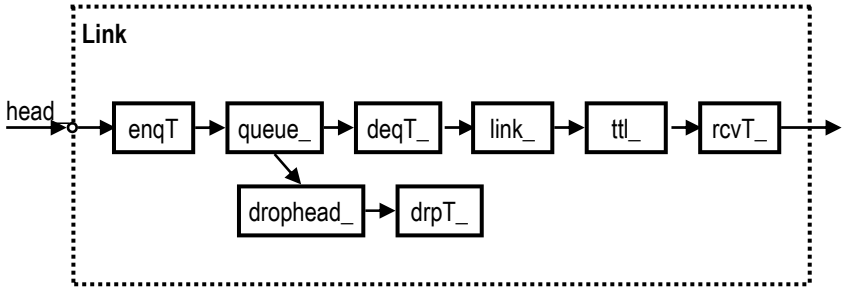

Figura 3.2. Struttura di un link unidirezionale.

Nella creazione di un link (sia unidirezionale che bidirezionale), possono essere adottate varie discipline di gestione della coda. In ns-2 è incluso il supporto per numerosi tipi di coda: Drop-Tail (che implementa una semplice disciplina FIFO), RED (Random Early Detection), CBQ (che include uno scheduler a priorità e round-robin), Fair Queueing, Stochastic Fair Queueing e Deficit Round-Robin. Le code rappresentano le locazioni dove i pacchetti possono essere mantenuti (o persi). É possibile specificare anche la dimensione della coda, intesa come il numero massimo di pacchetti che può mantenere al suo interno mediante il comando

```
$ns_ queue-limit <node1> <node2> <limit>
```

3.3.3 Definizione del traffico

Oltre a nodi e link, altre componenti fondamentali per definire uno scenario di simulazione sono gli Agenti e le Applicazioni.

Gli Agenti sono usati nell'implementazione di protocolli a vari livelli e rappresentano gli endpoint dove i pacchetti del livello network sono costruiti o consumati. La classe `Agent` ha parte dell'implementazione in OTcl e parte in C++. Ogni agente ha un proprio stato interno, composto dalle seguenti variabili:

- `addr_`: indirizzo della sorgente dei pacchetti;
- `dst_`: destinazione dei pacchetti;

- `size_`: dimensione del pacchetto in byte;
- `type_`: tipo del pacchetto;
- `fid_`: identificatore del flusso IP;
- `prio_`: campo priorità IP;
- `flags_`: flag del pacchetto;
- `defttl_`: valore di default del Time-To-Live IP.

Queste variabili possono essere modificate da una delle sottoclassi della classe `Agent`, sebbene solo alcune di queste sono richieste da un particolare agente.

Gli agenti più importanti sono quelli che implementano i due protocolli più usati nel livello trasporto: TCP e UDP. Gli agenti TCP possono essere di due tipi: one-way e two-way. I primi sono ulteriormente suddivisi in trasmettitore TCP (ad esempio Agent/TCP/Reno) e ricevitore TCP (ad esempio Agent/TCP/Sink) e cercano di simulare il più fedelmente possibile il controllo della congestione e degli errori, senza essere, però, repliche esatte delle implementazioni reali del protocollo TCP. I secondi sono simmetrici, nel senso che è implementato al loro interno un trasferimento dati bidirezionale, e costituiscono una replica molto fedele dell'implementazione del TCP nei sistemi BSD.

La creazione degli agenti (TCP o UDP) e la successiva assegnazione ai nodi può essere fatta mediante i seguenti comandi:

```
set tsrc [new Agent/TCP]
```
Crea un'istanza dell'agente TCP che implementa l'algoritmo Reno e la assegna alla variabile `tsrc`.

```
set tdst [new Agent/TCPSink]
```
Crea un'istanza dell'agente TCPSink, responsabile dell'invio di Ack al trasmettitore TCP e la assegna alla variabile `tdst`.

```
$ns_ attach-agent $node0 $tsrc
```
Assegna l'agente TCP `tsrc` al nodo `node0`.

```
$ns_ attach-agent $node1 $tdst
```
Assegna l'agente TCPSink `tdst` al nodo `node1`.

```
$ns_ connect $tsrc $tdst
```
Stabilisce una connessione end-to-end tra i due agenti `tsrc` e `tdst` a livello trasporto.

Ogni tipo di agente può essere configurato ulteriormente mediante opportune variabili. Per un agente TCP è possibile specificare, ad esempio, la dimensione della finestra di congestione, di slow start threshold e dei pacchetti.

Le applicazioni si collocano al di sopra degli agenti del livello trasporto. La figura 3.3 illustra due esempi di composizione delle applicazioni e connessione agli agenti di trasporto.

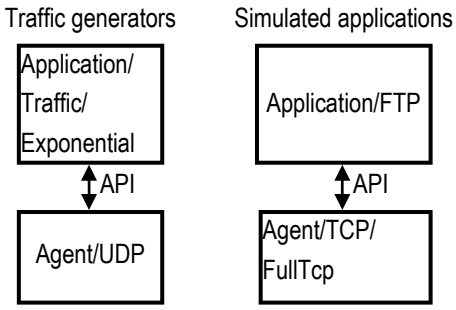

Figura 3.3. Esempio di composizione di applicazioni.

Nei sistemi reali, le applicazioni tipicamente accedono ai servizi di rete attraverso una Application Programming Interface (API). La più popolare di queste API è nota con il nome di socket. Nel simulatore, viene riprodotto il comportamento dei socket mediante un insieme ben definito di funzioni API. Queste funzioni sono poi mappate nelle funzioni interne dell'agente (ad esempio una chiamata al metodo `send(numBytes)` fa sì che il TCP incrementi il suo buffer di trasmissione del numero di byte corrispondente).

In ns-2 sono definiti due tipi di applicazioni: i generatori di traffico e le applicazioni simulate. I generatori di traffico implementati nel simulatore sono quattro:

1. EXPOO_Traffic: genera traffico secondo una distribuzione Esponenziale On/Off. I pacchetti sono inviati ad una velocità fissata durante il periodo On e nessun pacchetto è inviato durante il periodo Off.
2. POO_Traffic: genera traffico in accordo alla distribuzione Pareto On/Off. E' identico al precedente, eccetto che i periodi di On e Off sono presi da una distribuzione Pareto.
3. CBR_Traffic: genera traffico ad una velocità costante.
4. TrafficTrace: genera traffico in accordo ad un file di traccia.

Ogni generatore può essere configurato mediante opportune variabili.

Le applicazioni simulate attualmente implementate in ns-2 sono FTP e Telnet. Il primo è utile per simulare un "bulk data transfer", mentre il secondo un trasferimento dati interattivo.

La creazione delle applicazioni e la successiva connessione all'agente è possibile mediante i seguenti comandi:

`set ftp [new Application/FTP]`
Crea un'istanza dell'applicazione FTP e la assegna alla variabile `ftp`.

`$ftp attach-agent $tsrc`
Assegna l'applicazione `ftp` all'agente `tsrc`.

`$ns at <time> $ftp start`
Fa partire il trasferimento dati.

`$ns at <time> $ftp stop`
Interrompe il trasferimento dati.

3.3.4 Moduli di errore

I moduli di errore in ns-2 simulano errori al livello link o perdite di pacchetti. Nelle simulazioni, gli errori possono essere generati a partire da un semplice modello come la velocità di perdita del pacchetto, o a partire da modelli empirici e statistici più complessi. L'unità di errore, considerata la grande varietà di modelli, può essere specificata in termini di pacchetti, bit o istanti di tempo.

Alcuni dei più significativi modelli di errore sono:

- ErrorModel/List: specifica una lista di pacchetti/byte da perdere;
- ErrorModel/Trace: modello di errore che legge da un file di traccia;
- ErrorModel/Periodic: modella perdite periodiche di pacchetti (perde un pacchetto dopo averne visti N).

Qui di seguito è riportato un semplice esempio di creazione di un modello di errore:

`set LossyLink [$ns_ link $node0 $node1]`
Crea un link unidirezionale fra i nodi `node0` e `node1` con ritardo nullo e senza limitazione di banda e lo assegna alla variabile `LossyLink`.

`set em [new ErrorModel/List]`
Crea un'istanza del modello di errore e la assegna alla variabile `em`.

`$em droplist {11 20 30 40}`
Specifica la lista dei pacchetti da perdere.

`$LossyLink errormodule $em`
Associa il modulo di errore `em` al link `LossyLink`.

`add-error $LossyLink`
Attiva il modello di errore nel link `LossyLink`.

3.3.5 Traccia della simulazione

ns-2 offre la possibilità di raccogliere l'output di una simulazione in vari modi. Generalmente, l'output può essere visualizzato direttamente durante l'esecuzione della simulazione, oppure può essere memorizzato in un file per successive analisi e post-processing. Un modo semplice per generare una traccia è l'uso del comando `trace-all <file>` che crea un file con il formato mostrato in figura 3.4.

event	time	from node	to node	pkt type	pkt size	flags	fid	src addr	dst addr	seq num	pkt id

```
r 1.3556 3 2 ack 40 ------- 1 3.0 0.0 15 201
+ 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
- 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
d 1.35576 2 3 tcp 1000 ------- 1 0.0 3.0 29 199
```

dove il primo campo, "event", ha il seguente significato:

```
 r : receive (at to_node)          - : dequeue (at queue)
 + : enqueue (at queue)            d : drop    (at queue)
```

e i campi *addr hanno il seguente formato:

```
src_addr : node.port (3.0)
dst_addr : node.port (0.0)
```

Figura 3.4. Esempio di formato della traccia.

Ogni riga della traccia ha come primo campo il descrittore dell'evento (+, -, d, r) seguito dall'istante temporale (in secondi) di quell'evento, dai nodi mittente e destinatario, i quali identificano il link nel quale l'evento è occorso. Segue il tipo del pacchetto, la dimensione (in byte) e i flags usati. I campi successivi sono il flow id (fid) dell'IPv6 che può essere impostato per ogni flusso nello script OTcl, l'indirizzo sorgente e destinazione nella forma `nodo.porta`. Gli ultimi due campi riportano rispettivamente il numero di sequenza del pacchetto e il suo identificativo. Anche se le implementazioni del protocollo UDP non usano il numero di sequenza, il simulatore tiene traccia del numero di sequenza per scopi di analisi.

3.4 Strumenti accessori: Nam e Xgraph

La distribuzione dei sorgenti di ns-2 contiene, oltre al codice sorgente del simulatore, alcuni strumenti che possono essere utilizzati per analizzare le tracce di una simulazione. I principali sono Nam e Xgraph.

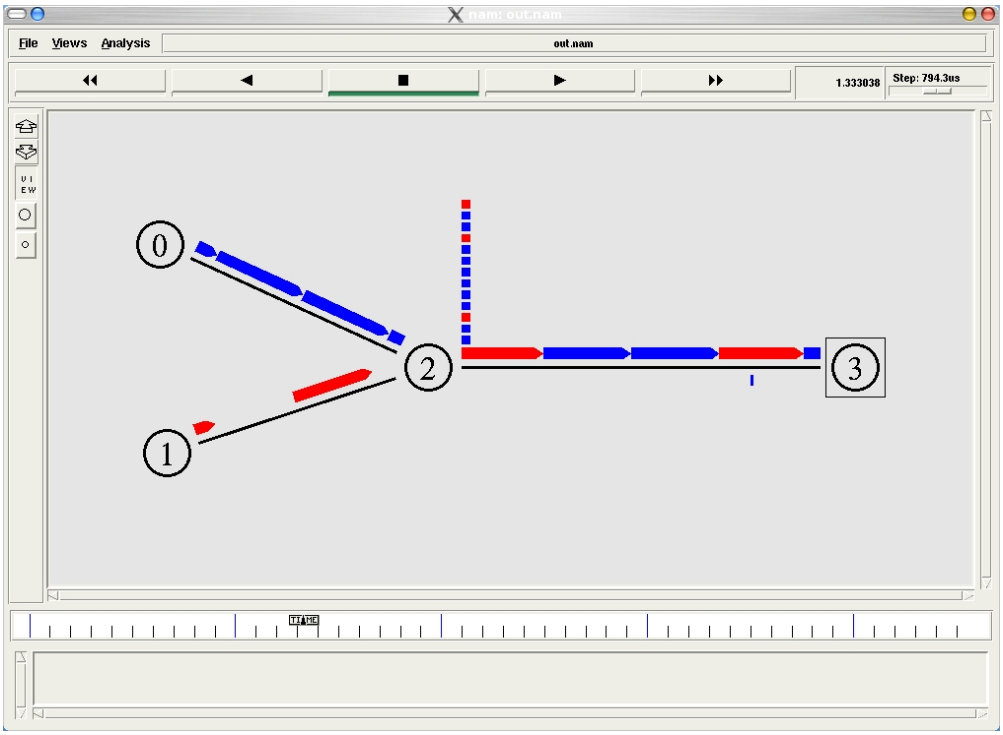

Figura 3.5. Un schermata di Nam.

Il primo, Nam (The Network Animator), è un tool che permette di ricostruire lo scenario di simulazione attraverso un'animazione. Necessita di un file di traccia particolare (tipicamente con estensione .nam) in cui vengono registrati tutti gli eventi significativi durante la simulazione, ed in particolare gli istanti di generazione e ricezione di ogni singolo pacchetto ed il loro percorso dettagliato attraverso i vari nodi. Il file di traccia nam è creato nello script OTcl mediante la funzione `namtrace-all`. In questo modo è possibile avere un riscontro visivo dell'interazione tra i vari componenti; inoltre nel corso dell'animazione è possibile realizzare dei fermoimmagine, aumentare o diminuire la velocità di avanzamento e, cliccando sui vari elementi grafici che compongono la simulazione, ottenere informazioni specifiche (ad esempio, cliccando su un link si ottengono i grafici di utilizzo e di perdita). Un esempio molto semplice di animazione Nam è riportato in figura 3.5.

Il secondo, Xgraph, è un programma che traccia semplici grafici. L'insieme di punti da tracciare possono essere in un file nel quale ogni riga contiene le coordinate x e y separate da uno spazio, oppure possono essere letti dallo standard input. Tale strumento si rivela di fondamentale importanza nella valutazione di vari aspetti di uno scenario di simulazione (ad esempio l'andamento della finestra di congestione e di slow start threshold). La figura 3.6 mostra una finestra di Xgraph.

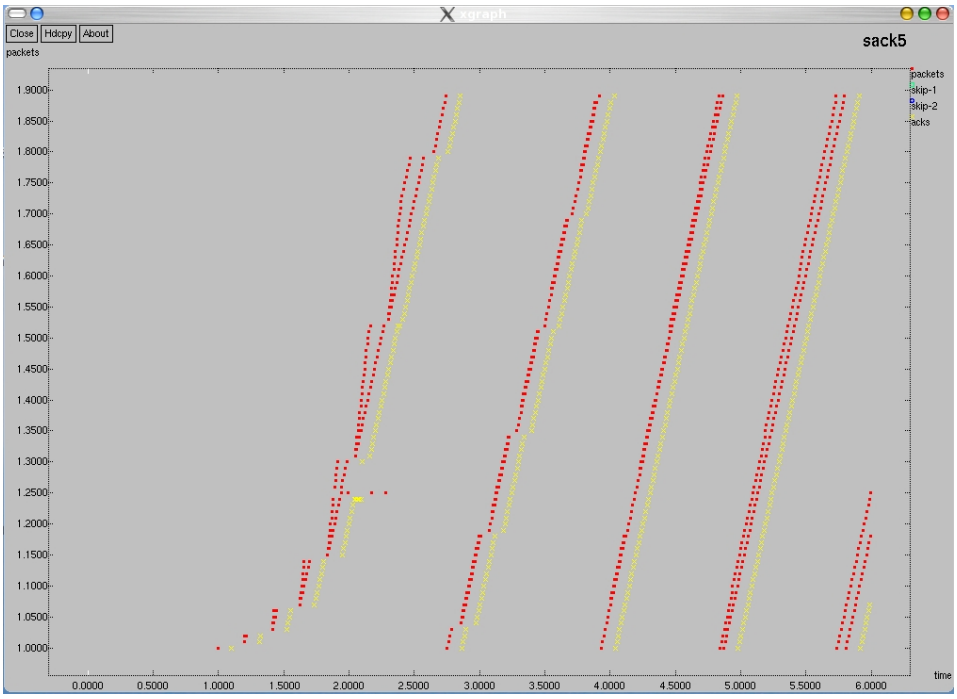

Figura 3.6. Una schermata di Xgraph.

Capitolo 4 Modifiche al codice del simulatore

4.1 Introduzione

RFC 3782, "The NewReno modification to TCP's Fast Recovery Algorithm", ([RFC3782]), definisce una variante dell'algoritmo Fast Retransmit/Fast Recovery nota con il nome di NewReno. Come già descritto nel paragrafo 2.8.2, le modifiche essenziali di tale algoritmo rispetto a quello riportato in RFC 2581 consistono:

- nell'aggiunta di alcuni test da effettuare alla ricezione di tre Ack duplicati;
- nell'introduzione di una variabile di controllo, `recover_`;
- nella risposta agli Ack parziali (variante Slow but Steady e variante Impatient);
- nell'inserimento di un ulteriore passo (passo 6) per evitare Fast Retransmit multipli.

In questo capitolo si è analizzato e studiato il comportamento del simulatore ns-2, focalizzando l'attenzione su due punti in particolare:

- l'aggiornamento della finestra di congestione in seguito ad un Retransmit Timeout (vedi paragrafo 4.2);
- la risposta agli Ack parziali (vedi paragrafo 4.3).

4.2 Controllo della finestra di congestione in seguito ad un timeout

RFC 3782 specifica che, alla ricezione del terzo Ack duplicato, il trasmettitore TCP entra in Fast Recovery solo se il campo Acknowledgment Number dell'Ack è maggiore del valore della variabile `recover_`. La ragione è che, dopo un timeout avvenuto durante un Fast Recovery, il ricevente continua a ricevere Ack duplicati fino a quando non viene ritrasmesso il segmento con numero di sequenza pari al valore della variabile `recover_`.

Dall'esame del comportamento del simulatore ns-2, si nota che dopo un timeout, in presenza di più di tre Ack duplicati, il trasmettitore, per ognuno di questi, aumenta la finestra di congestione di un'unità, permettendo a nuovi segmenti di essere tra-

smessi. Anche se il trasmettitore non è in Fast Recovery, il suo comportamento, almeno per quanto riguarda l'aggiornamento della finestra di congestione, è del tutto analogo a quello che avrebbe se fosse entrato nella procedura di recupero. Questo non è conforme a quanto stabilito da RFC 3782, nel quale cwnd non deve essere soggetta ad alcun aumento per via degli Ack duplicati ricevuti.

In prima istanza entrambi i comportamenti risultano corretti: l'approccio di ns-2 è più aggressivo, se si tiene conto del numero di segmenti inviati; l'altro, invece, si dimostra più cauto nell'invio di segmenti.

Uno dei passi importanti di questa fase della tesi è la modifica dei sorgenti del simulatore ns-2, in modo tale da offrire la possibilità all'utente di poter scegliere quale tra i due approcci adottare di volta in volta.

4.2.1 Dettagli implementativi

Come già accennato nel paragrafo 3.2, il simulatore ns-2 fa uso di due linguaggi di programmazione: C++ per il backend e OTcl per il frontend. Il linguaggio OTcl si presta maggiormente alla variazione di parametri e di configurazioni all'interno di uno scenario, poiché non occorre la ricompilazione dell'intero simulatore.

Nei sorgenti di ns-2, all'interno della directory `~ns/tcl/lib`, è presente un file (`ns-default.tcl`) nel quale sono elencate le variabili membri delle classi di ns-2, sia quelle visibili solo in OTcl che quelle legate con un bind tra OTcl e C++, con i rispettivi valori di default.

Al fine di consentire all'utente la scelta tra i due approcci (quello predefinito in ns-2 e quello di RFC 3782), si è aggiunto in questo file una variabile di nome `cwnd_inflation_after_timeout_`, come indicato qui di seguito:

```
--- ns-default.tcl.2.27 2005-05-07 09:59:46.000000000 +0000
+++ ns-default.tcl      2005-05-07 10:10:49.000000000 +0000
@@ -1040,6 +1040,8 @@ if [TclObject is-class Agent/TCP/FullTcp

  Agent/TCP/FullTcp/Newreno set recov_maxburst_ 2;
+ Agent/TCP/FullTcp/Newreno set newreno_changes1_ 1;
+ Agent/TCP/FullTcp/Newreno set cwnd_inflation_after_timeout_ 1;
  Agent/TCP/FullTcp/Sack set sack_block_size_ 8;
  Agent/TCP/FullTcp/Sack set sack_option_size_ 2;
  Agent/TCP/FullTcp/Sack set max_sack_blocks_ 3;
```

La variabile `cwnd_inflation_after_timeout_` è legata, mediante il metodo `bind`, alla variabile C++ omonima. Il valore di default è 1, quindi è effettuato l'incremento di cwnd in presenza di Ack duplicati anche al di fuori della procedura di recupero.

Le altre modifiche al codice interessano esclusivamente i file `tcp-full.h` e `tcp-full.cc`. Questi file simulano il comportamento di un two-way TCP per le varianti più conosciute del protocollo: Reno, NewReno e Sack.

Alla ricezione di un segmento, è invocato il metodo `recv`, il quale, al terzo Ack duplicato consecutivo ricevuto dal trasmettitore, invoca il metodo `dupack_action`, ridefinito opportunamente per l'algoritmo NewReno, il quale individua l'azione opportuna da effettuare in risposta a quanto ricevuto. Nel caso in cui la disequazione `highest_ack_ > recover_` risulti non verificata, si salta al ramo else e si controlla il valore della variabile `cwnd_inflation_after_timeout_`. Se tale valore è 0, si avrà un comportamento conforme a quanto stabilito dal passo 1 dell'algoritmo NewReno. Al contrario, invece, la variabile che mantiene il numero di Ack duplicati (`dupacks_`) non subisce alcuna variazione e, in presenza di ulteriori Ack duplicati, il trasmettitore entra nel metodo `extra_ack` (definito nel file `tcp-full.h`) dove si procede all'incremento di cwnd.

```
--- tcp-full.cc.2.27    2005-05-09 21:21:57.584520792 +0200
+++ tcp-full.cc         2005-05-09 21:21:23.963631944 +0200
@@ -2668,6 +2670,56 @@ dupack_action:
+void
+NewRenoFullTcpAgent::dupack_action()
+{
+     int recovered = (highest_ack_ > recover_);
+     if (recovered) {
+           firstpartial_ = TRUE;
+           FullTcpAgent::dupack_action();
+     } else {
+           // After a timeout, we may receive dupacks before we get
+           // to retransmit the highest seqno we transmitted before
+           // the timeout (stored in recover_). RFC 3782 says
+           // (3.1B) not to enter Fast Retransmit. By reducing the
+           // dupack count by 1 we remember that we received
+           // dupacks.
+           if (!cwnd_inflation_after_timeout_)
+                 dupacks_ -= 1;
+     }
+}
```

4.2.2 Scenario di prova

Per mostrare i differenti comportamenti in base al valore assunto dalla variabile `cwnd_inflation_after_timeout_`, si è costruito un semplice scenario in ns-2 costituito da due nodi connessi mediante un link con larghezza di banda pari a 1Mbit/s e ritardo pari a 100 ms (vedi appendice). La dimensione del buffer associato al link non è essenziale ai fini dello scenario qui considerato ed è pari a 48 segmenti, considerando segmenti di 256 byte. Mediante un modello di errore chiamato `ErrorModel/List` è possibile specificare i segmenti che devono essere persi. In questo scenario i segmenti persi sono quello con id 62 (numero di sequenza 13177) e la sua ritrasmissione. Quando il trasmettitore riceve il terzo Ack duplicato relativo al

segmento 62, lo ritrasmette. La ritrasmissione del segmento con id 62, che avviene all'istante t = 1.42 s, va persa, il che causa un timeout, per cui il trasmettitore è costretto ad attendere un RTO prima di poter ritrasmettere quel segmento. Inviatolo, il trasmettitore riceve 30 Ack duplicati. Ogni Ack, dal 4° al 30°, fa aumentare cwnd di una unità, portando quest'ultima a una dimensione pari a 28. Per ogni incremento di cwnd, inoltre, può essere inviato un nuovo segmento, 27 segmenti in totale (vedi fig. 4.1).

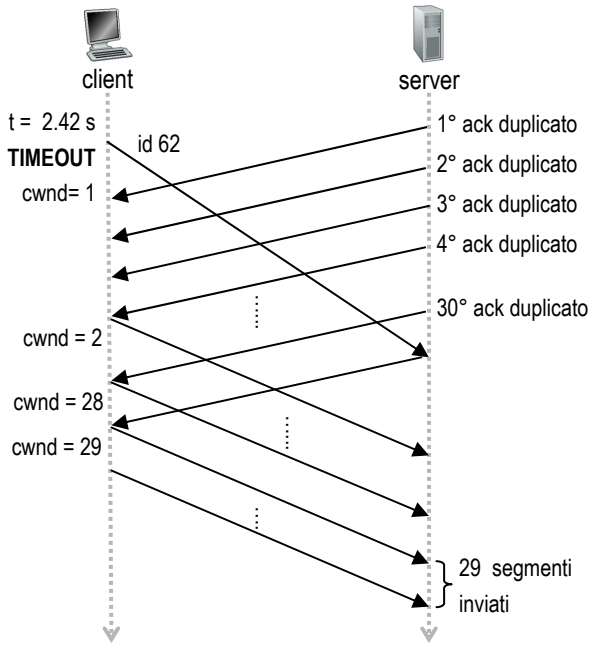

Figura 4.1. Esempio di comportamento del simulatore.

Alla ricezione dell'Ack relativo al segmento ritrasmesso al timeout (quello con id 62), cwnd è incrementata di un'unità, raggiungendo il valore 29. Dato che per tutti i segmenti inviati in precedenza si è ricevuto anche un riscontro, il trasmettitore ha la possibilità di inviare contemporaneamente tanti segmenti quanti ne contiene la finestra di congestione, cioè 29 (vedi fig. 4.3).

Questo considerevole aumento della finestra di congestione non avrebbe avuto luogo nel caso di un'implementazione dell'algoritmo NewReno conforme a RFC 3782. In tal caso, l'unico incremento possibile (di cwnd) si sarebbe avuto con la ricezione dell'Ack corrispondente al segmento perso (vedi fig 4.2). Questo perché ogni volta che la variabile `dupacks_` raggiunge il valore 3, il trasmettitore effettua un decremento della variabile di un'unità (vedi fig. 4.4).

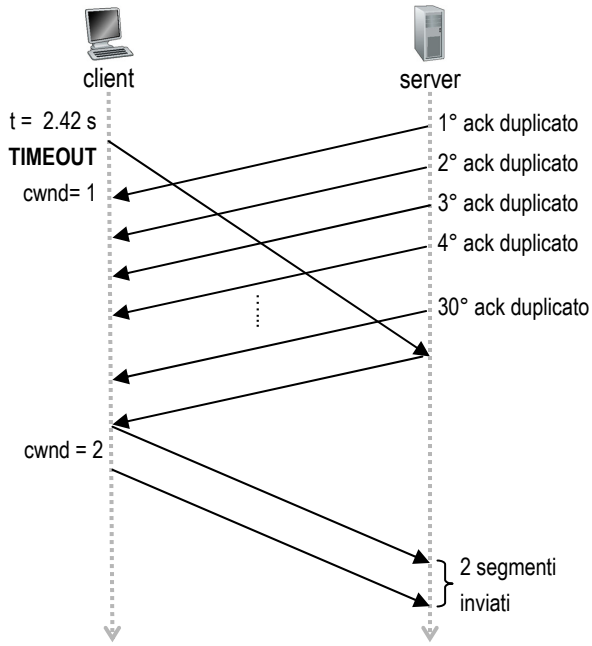

Figura 4.2. Esempio di comportamento conforme a RFC 3782.

Nello scenario preso in esame, l'incremento di cwnd, dovuto alla ricezione di Ack duplicati, favorisce l'invio di un numero di segmenti maggiore rispetto a quelli inviati adottando l'altro approccio. Dal punto di vista delle prestazioni, l'approccio del simulatore ns-2 risulta vincente rispetto a quello indicato da RFC 3782. Nello scenario qui esaminato, in una simulazione di durata pari a 5 secondi, con il primo approccio sono inviati circa 1400 segmenti contro i circa 1000 segmenti dell'altro approccio.

La ratio dietro questo approccio è il cosiddetto principio di conservazione dei pacchetti, cioè la considerazione che, ogni volta che si riceve un Ack, un segmento ha lasciato la rete, e quindi immettere un nuovo pacchetto non peggiora le condizioni di congestione preesistenti.

Vi è comunque un caso nel quale l'approccio suggerito da RFC 3782 permette di ottenere prestazioni migliori: si supponga che, all'arrivo dell'Ack relativo al segmento ritrasmesso, cwnd abbia un valore superiore alla dimensione del buffer associato al link. Il trasmettitore invia contemporaneamente più segmenti di quanti ne può contenere il buffer e quindi inevitabilmente alcuni si perdono. Questa perdita di segmenti non si sarebbe avuta se il trasmettitore avesse adottato l'approccio più cauto dettato da RFC. In questo caso, infatti, il principio di conservazione dei pacchetti è perdente, in quanto le condizioni di congestione preesistenti sono eccessive per la rete.

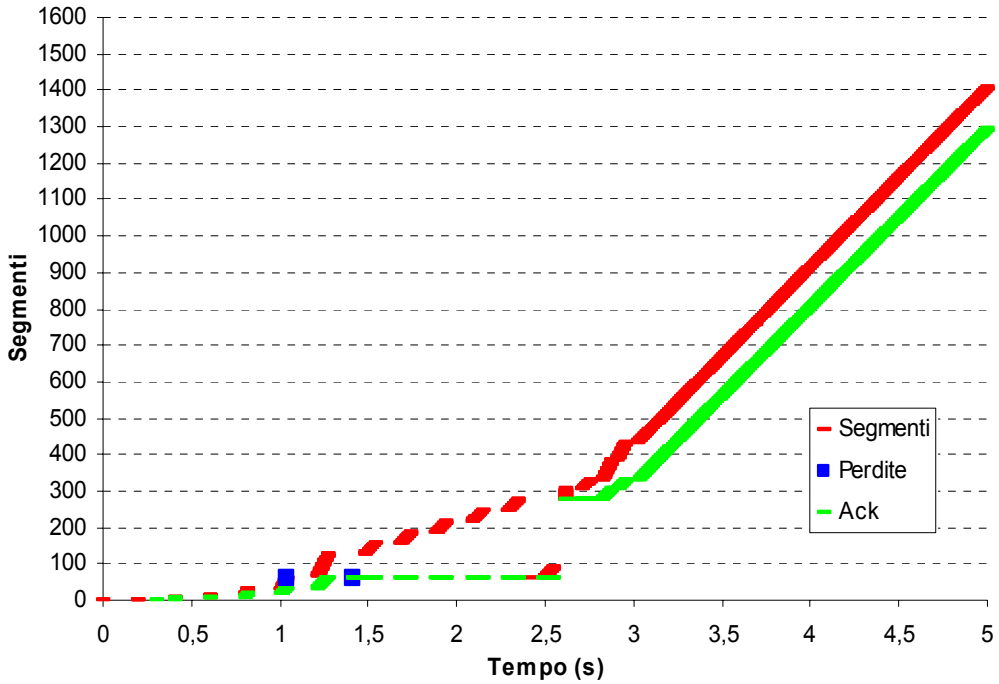

Figura 4.3. Esempio di comportamento del simulatore ns-2.
Lo scenario è stato creato con il file Tcl `cwnd_inflation.tcl`.

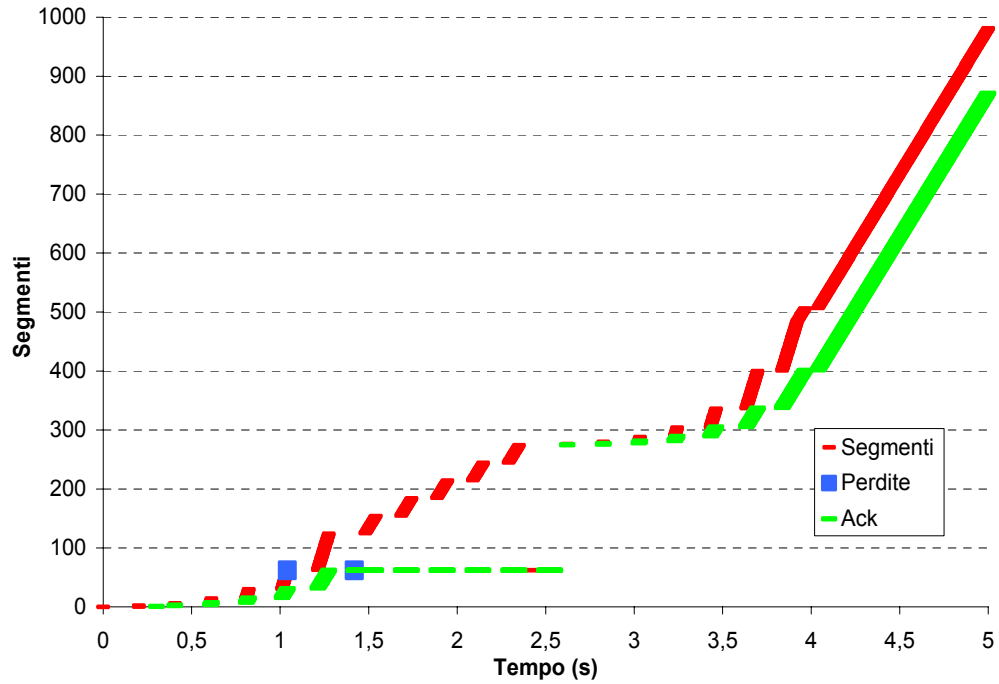

Figura 4.4. Esempio di comportamento di RFC 3782.
Lo scenario è stato creato con il file Tcl `cwnd_inflation.tcl`.

4.3 Motivazioni dell'introduzione della variante Impatient

L'algoritmo di Fast Recovery del NewReno descrive due comportamenti diversi in risposta ad un Ack parziale (vedi sezione 4 di [RFC3782]): nel primo, noto come variante Slow but Steady, il riarmo del timer di ritrasmissione è fatto per ogni Ack parziale; nel secondo, noto come variante Impatient, è fatto solo per il primo Ack parziale. Ci sono casi in cui all'interno di una finestra di dati si perdono molti segmenti e l'adozione della variante Impatient comporta un recupero più veloce e prestazioni migliori. Sono possibili, però, anche casi in cui la variante Slow but Steady offre prestazioni più alte rispetto all'altra variante. Ad esempio, quando un numero ridotto di segmenti sono persi e il RTO è sufficientemente piccolo da far scattare il timer di ritrasmissione. Nel paragrafo 4.3.2 sono descritti scenari che dettagliano entrambi i casi.

Il simulatore ns-2 contiene delle classi per simulare sia il comportamento di un one-way TCP (classe Tcl `Agent/TCP/Newreno`) che di un two-way TCP (classe Tcl `Agent/TCP/FullTcp/Newreno`) con l'algoritmo NewReno. Mentre per il primo tipo sono implementate tutte e due le varianti (offrendo la possibilità all'utente di scegliere una delle due varianti mediante un'apposita variabile OTcl), per il secondo è implementata solamente la variante Slow but Steady. Questo significa che la classe Tcl `Agent/TCP/FullTcp/Newreno` può essere usata solo con la variante Slow but Steady, che ha prestazioni molto scadenti in alcune situazioni, come accennato in precedenza.

Uno dei risultati presentati in questa tesi è la modifica del codice del simulatore per introdurre la variante Impatient nel two-way TCP, consentendo all'utente di poter scegliere la variante adatta ad ogni scenario.

4.3.1 Dettagli implementativi

L'introduzione della variante Impatient nel simulatore ha richiesto la modifica dei valori di default di ns-2 contenuti nel file `ns-default.tcl` (presente nella directory `~ns/tcl/lib)` e delle classi che descrivono il comportamento del two-way TCP.

Il file `ns-default.tcl`, come già detto nel paragrafo 4.2.1, elenca i valori di default sia per le variabili visibili solo in OTcl che per quelle legate con un bind tra OTcl e C++. In questo file si è aggiunta la variabile `newreno_changes1_` per consentire all'utente di poter scegliere se usare o meno la variante Impatient.

```
--- ns-default.tcl.2.27 2005-05-07 09:59:46.000000000 +0000
+++ ns-default.tcl      2005-05-07 10:10:49.000000000 +0000
@@ -1040,6 +1040,8 @@ if [TclObject is-class Agent/TCP/FullTcp
```

```
  Agent/TCP/FullTcp/Newreno set recov_maxburst_ 2;
+ Agent/TCP/FullTcp/Newreno set newreno_changes1_ 1;
+ Agent/TCP/FullTcp/Newreno set cwnd_inflation_after_timeout_ 1;
  Agent/TCP/FullTcp/Sack set sack_block_size_ 8;
  Agent/TCP/FullTcp/Sack set sack_option_size_ 2;
  Agent/TCP/FullTcp/Sack set max_sack_blocks_ 3;
```

La variabile `newreno_changes1_` è legata, mediante il metodo `bind`, alla variabile C++ omonima. Se posta a 1, indica che è adottata la variante Impatient. La scelta di un nome non molto esplicativo per questa variabile è motivata dalla presenza nel one-way TCP (classe Tcl `Agent/TCP/Newreno`) di un'omonima variabile con lo stesso significato.

Le altre modifiche apportate al simulatore riguardano i file `tcp-full.h` e `tcp-full.cc`. Questi due file descrivono le varianti più conosciute del protocollo TCP: Reno, NewReno e Sack.

Alla ricezione di un segmento (dati o Ack), viene invocato il metodo `recv`, il quale si occupa della ricezione di un segmento. In presenza di tre Ack duplicati, si entra nel metodo `dupack_action`, il quale individua l'azione opportuna da effettuare in risposta agli Ack ricevuti. Nel caso in cui il trasmettitore non sia in Fast Recovery e il valore della variabile `recovered_` sia vero, la variabile `firstpartial_` assume valore true. Questa variabile è di fondamentale importanza per il riarmo del timer di ritrasmissione, come discusso in seguito.

```
--- tcp-full.cc.2.27    2005-05-09 21:21:57.584520792 +0200
+++ tcp-full.cc         2005-05-09 21:21:23.963631944 +0200
@@ -2668,6 +2670,56 @@ dupack_action:
+void
+NewRenoFullTcpAgent::dupack_action()
+{
+     int recovered = (highest_ack_ > recover_);
+     if (recovered) {
+           firstpartial_ = TRUE;
+           FullTcpAgent::dupack_action();
+     } else {
+           // After a timeout, we may receive dupacks before we get
+           // to retransmit the highest seqno we transmitted before
+           // the timeout (stored in recover_). RFC 3782 says
+           // (3.1B) not to enter Fast Retransmit. By reducing the
+           // dupack count by 1 we remember that we received
+           // dupacks.
+           if (!cwnd_inflation_after_timeout_)
+                 dupacks_ -= 1;
+     }
+}
```

Superato il controllo sugli Ack duplicati, nel metodo `recv`, si controlla se un Ack è parziale utilizzando il metodo `pack`. Ma un Ack può essere considerato parziale solo se il trasmettitore è in Fast Recovery, come indicato nelle righe seguenti:

```
--- tcp-full.cc.2.27    2005-05-09 21:21:57.584520792 +0200
+++ tcp-full.cc         2005-05-09 21:21:23.963631944 +0200
@@ -2108,7 +2108,7 @@ process_ACK:

-           int partial = pack(pkt);
+           int partial = fastrecov_ && pack(pkt);

            if (partial)
            pack_action(pkt);
```

Questo evita che venga invocato il metodo `pack_action`, il quale ritrasmette il primo segmento non ancora riscontrato anche se non si è in Fast Recovery.

Il metodo `newack`, il quale si occupa della gestione dei timer di ritrasmissione e dell'avanzamento del numero di Ack, è stato modificato per poter implementare le varianti Slow but Steady e Impatient, facendo ricorso ad un metodo accessorio chiamato `reset_rtx_timer_after_ack`.

```
--- tcp-full.cc.2.27    2005-05-09 21:21:57.584520792 +0200
+++ tcp-full.cc         2005-05-09 21:21:23.963631944 +0200
@@ -1236,7 +1236,7 @@ FullTcpAgent::newack(Packet* pkt)
      if (ackno == maxseq_) {
            cancel_rtx_timer();     // all data ACKd
      } else if (progress) {
-           set_rtx_timer();
+           reset_rtx_timer_after_ack(pkt);
      }
```

Tale metodo, nel caso in cui è adottata la variante Impatient, riarma il timer di ritrasmissione soltanto per il primo Ack parziale, in quanto la variabile `firstpartial_` ha valore true; per tutti gli altri Ack parziali non si ha alcun riarmo del timer poiché `firstpartial_` ha valore falso. Se è adottata la variante Slow but Steady o è utilizzato un diverso algoritmo, il riarmo del timer di ritrasmissione avviene per ogni Ack parziale.

```
--- tcp-full.cc.2.27    2005-05-09 21:21:57.584520792 +0200
+++ tcp-full.cc         2005-05-09 21:21:23.963631944 +0200
@@ -2668,6 +2670,56 @@ reset_rtx_timer_after_ack:
+/*
+ * do a reset of the RTO timer after receiving an ACK
+*/
+
+void
+FullTcpAgent::reset_rtx_timer_after_ack(Packet* pkt)
+{
+     set_rtx_timer();
+}
+
+/*
+ * Newreno, when the impatient variant is used (RFC 3782), does not
+ * reset the RTO timer in Fast Recovery after the first partial
+ * ACK.
+ */
+
```

```
+void
+NewRenoFullTcpAgent::reset_rtx_timer_after_ack(Packet* pkt)
+{
+     if (newreno_changes1_) {
+           hdr_tcp *tcph = hdr_tcp::access(pkt);
+           register int ackno = tcph->ackno();
+           // This is the "impatient" version of NewReno
+           int partial = (fastrecov_ && ((ackno > highest_ack_) &&
+            (ackno < recover_)));
+           int do_not_reset_timer = (partial && !firstpartial_);
+           firstpartial_ = FALSE;
+           if (do_not_reset_timer) {
+                 return;
+           }
+     }
+     FullTcpAgent::reset_rtx_timer_after_ack(pkt);
+}
```

4.3.2 Scenari di esempio

Per sottolineare le differenze tra le due varianti, si sono considerati due casi, in ognuno dei quali una diversa variante ottiene le migliori prestazioni:

1. la variante Impatient ottiene prestazioni migliori rispetto alla variante Slow but Steady: a tal proposito si sono costruiti due scenari ad hoc (vedi paragrafi 4.3.2.1 e 4.3.2.2);
2. la variante Slow but Steady prevale sull'altra: qui si è fatto riferimento alla suite di test fornita a corredo con i sorgenti del simulatore ns-2 (vedi paragrafo 4.3.2.3).

I due scenari ad hoc sono costituiti da due nodi, n1 e n2, connessi mediante un link bidirezionale da 1 Mbit/s con 100 ms di ritardo, come mostrato in figura 4.5. La dimensione del buffer associato al link è posta pari a 48 segmenti, considerando segmenti di 256 byte.

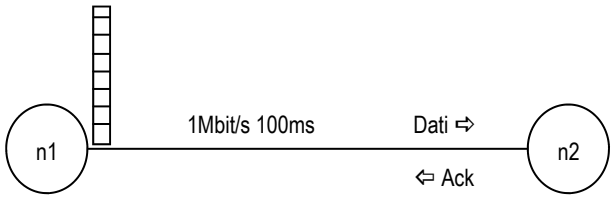

Figura 4.5. Scenario per il confronto tra le varianti Slow but Steady e Impatient.

La perdita di segmenti in ns-2 può essere impostata utilizzando opportuni modelli di errore. Uno dei più semplici è chiamato `ErrorModel/List` e permette di indicare esplicitamente quali segmenti devono essere persi, fornendone l'id (si veda il paragrafo 2.3.4 per maggiori dettagli sui modelli di errore).

4.3.2.1 Scenario ad hoc senza delayed Ack

Il primo scenario non adotta la tecnica del delayed Ack. Per ogni segmento ricevuto, il ricevitore invia un Ack. Questo risulta in un aumento più rapido della dimensione della finestra di congestione rispetto al metodo descritto in RFC 1122. E' da notare che questo scenario è proposto solo in forza della sua maggior semplicità, ma una simulazione realistica non può prescindere dai delayed Ack, che sono implementati in tutti i sistemi reali.

I segmenti perduti, in questo scenario, sono il 30 e tutti quelli a partire dal 34 fino al 47 incluso, in tutto 15 segmenti. All'istante t = 0.82 s, si perde il primo segmento. cwnd in quel momento è pari a 31 segmenti. Ricevuti i tre Ack duplicati, il trasmettitore deduce che qualche segmento si è perso e ritrasmette il primo segmento non ancora riscontrato (seg. n. 30). Alla ricezione di ogni Ack parziale, il trasmettitore viene informato della perdita di un solo segmento. Per questo motivo resta nella procedura di Fast Recovery per quasi tre secondi. All'istante t = 4.25 s, il trasmettitore esce dal Fast Recovery ricevendo un Ack completo relativo al segmento n.61. Il grafico sottostante mostra questa situazione.

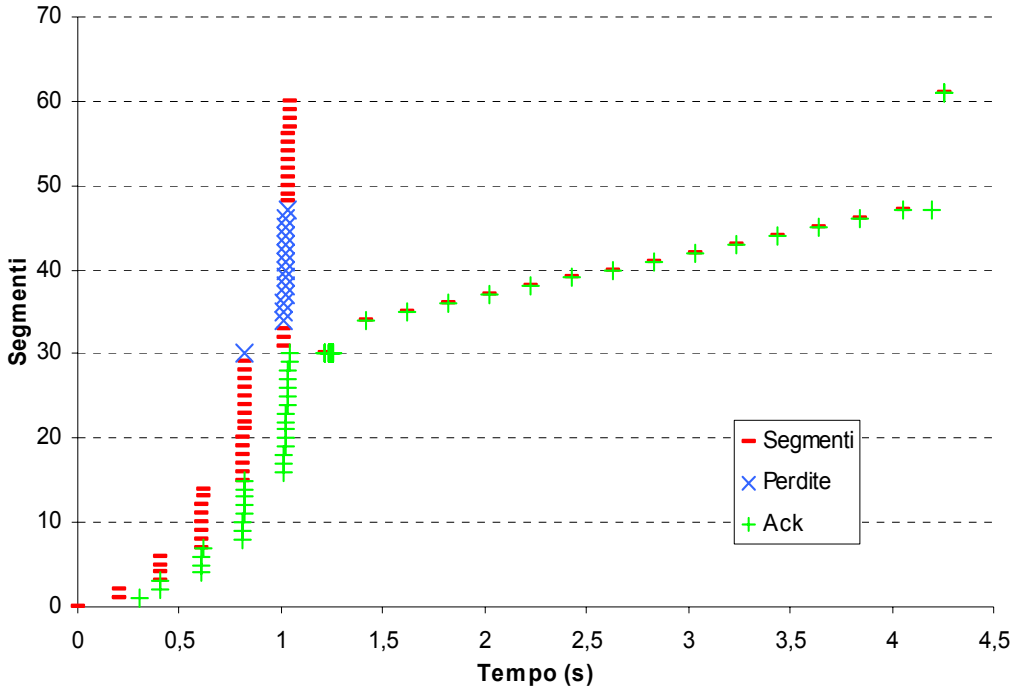

Figura 4.6. Comportamento della variante Slow but Steady.
Lo scenario è stato creato con il file Tcl `NewReno_without_del_ack.tcl`.

In questo scenario si può notare un aumento di prestazioni se si adotta la variante Impatient. Trascorso un secondo dalla ricezione del primo Ack parziale (t = 1.42 s), si ha un timeout (t = 2.42 s) poiché il trasmettitore non è riuscito a ritrasmettere tutti i segmenti persi. Il timeout favorisce un recupero più veloce dei segmenti persi, come evidenziato nel grafico riportato in figura 4.7. All'istante t = 3.04 s, il trasmettitore

riceve un Ack il quale indica che tutti i 15 segmenti persi sono arrivati correttamente a destinazione. Nella precedente variante, invece, occorre circa un secondo in più per poter trasmettere nuovi segmenti.

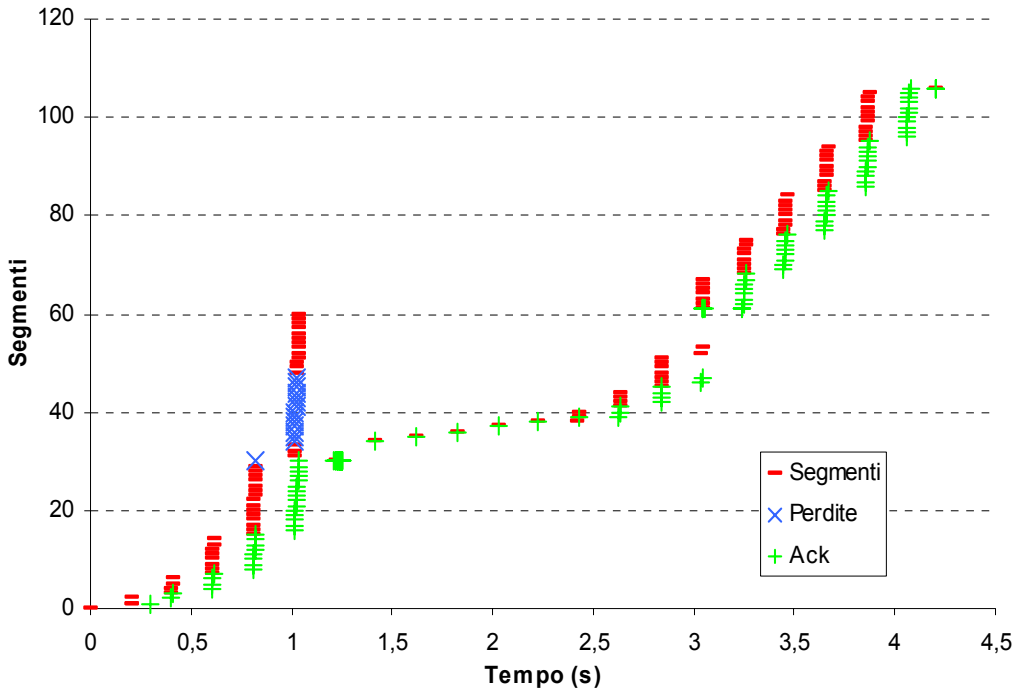

Figura 4.7. Comportamento della variante Impatient.
Lo scenario è stato creato con il file Tcl `NewReno_without_del_ack.tcl`.

4.3.2.2 Scenario ad hoc con delayed Ack

Il secondo scenario adotta, come nella realtà, la tecnica del delayed Ack. Quando il ricevitore TCP riceve un segmento, non invia immediatamente un Ack, ma attende una certa frazione di secondi (solitamente 100 ms). Se entro tale intervallo di tempo riceve un altro segmento, allora invia un Ack complessivo. Altrimenti, scaduto il timer, invia ugualmente un Ack. Questo si traduce essenzialmente in una riduzione del numero di Ack inviati dal ricevitore al trasmettitore.

In questo scenario si perdono 19 segmenti: il 37 e quelli a partire dal 41 al 58 incluso. Il primo di questi (id = 37) ha numero di sequenza 7777. Appena arriva il terzo Ack duplicato, il quale indica che il segmento n. 37 è andato perso, si entra in Fast Recovery. cwnd in quell'istante è uguale a 23, quindi ssthresh assume un valore pari a 11. Si ritrasmette il segmento perso e cwnd è posta al valore 14. Quando il segmento ritrasmesso arriva a destinazione, il ricevitore invia un Ack con numero di Ack pari a 8641, perché il segmento con quel numero di sequenza (seg. n. 41) è stato perso. Alla ricezione del primo Ack parziale, il trasmettitore riarma il timer di ritrasmissione senza uscire dal Fast Recovery e ritrasmette il primo segmento non ri-

scontrato (in questo caso il seg. n. 41). I segmenti successivi al seg. n.41 sono perduti fino al 57 incluso.

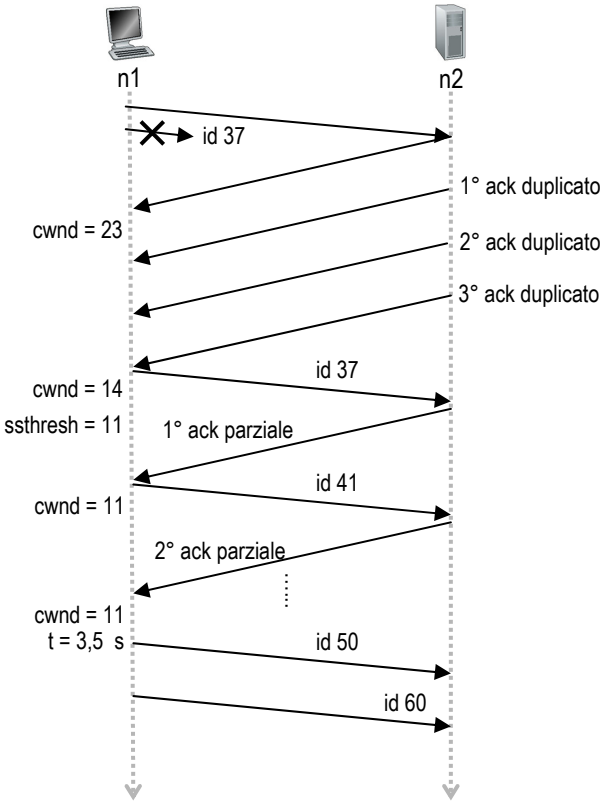

Figura 4.8. Sequenza dei segmenti scambiati adottando la variante Slow but Steady.

La perdita di un numero elevato di segmenti all'interno della finestra (19 segmenti in questo scenario) fa sì che il trasmettitore rimanga nella procedura di Fast Recovery per alcuni secondi, venendo a conoscenza, ad ogni RTT, della perdita di un solo segmento. Questo risulta evidente dal grafico riportato in figura 4.9 e dalla sequenza temporale di segmenti scambiati tra trasmettitore e ricevitore mostrata in figura 4.8. Si noti a tal proposito l'insieme di segmenti trasmesso nell'intervallo di tempo compreso tra t_1 = 1.71 s e t_2 = 5.15 s: questi sono i segmenti trasmessi ognuno in risposta ad un Ack parziale. Il trasmettitore invia più segmenti in un RTT se lo permettono le dimensioni della finestra di congestione, come ad esempio all'istante t=3.53 s, dove sono inviati due segmenti, anziché uno solo.

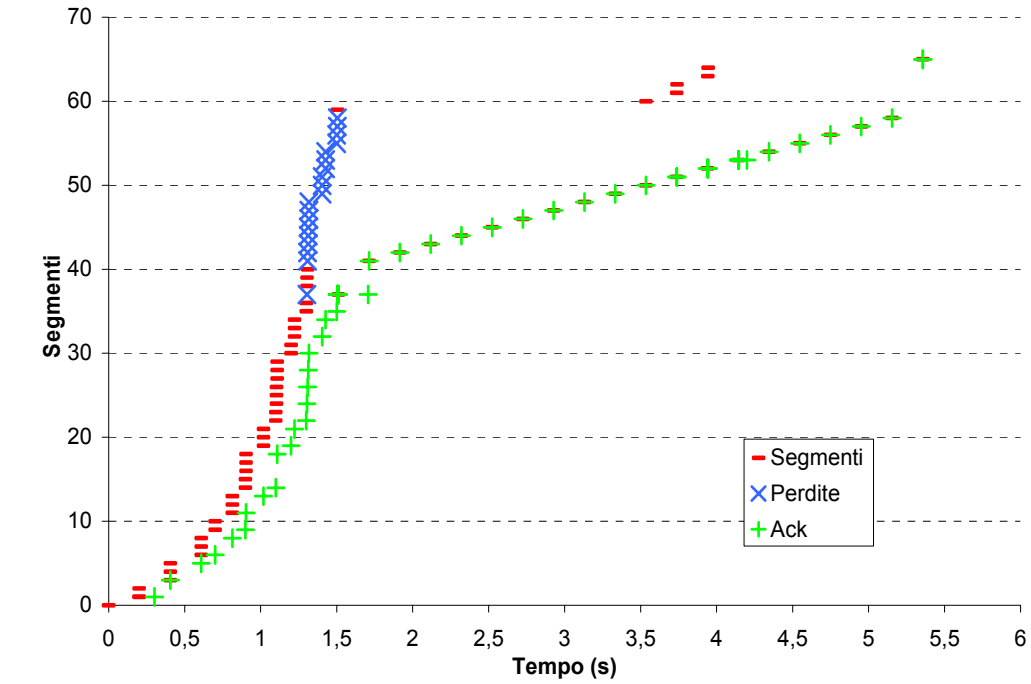

Figura 4.9. Comportamento della variante Slow but Steady.
Lo scenario è stato creato con il file Tcl `NewReno_del_ack.tcl`.

L'introduzione della variante Impatient, per questo scenario, darebbe luogo ad un recupero più veloce, come risulta dal grafico in figura 4.10.

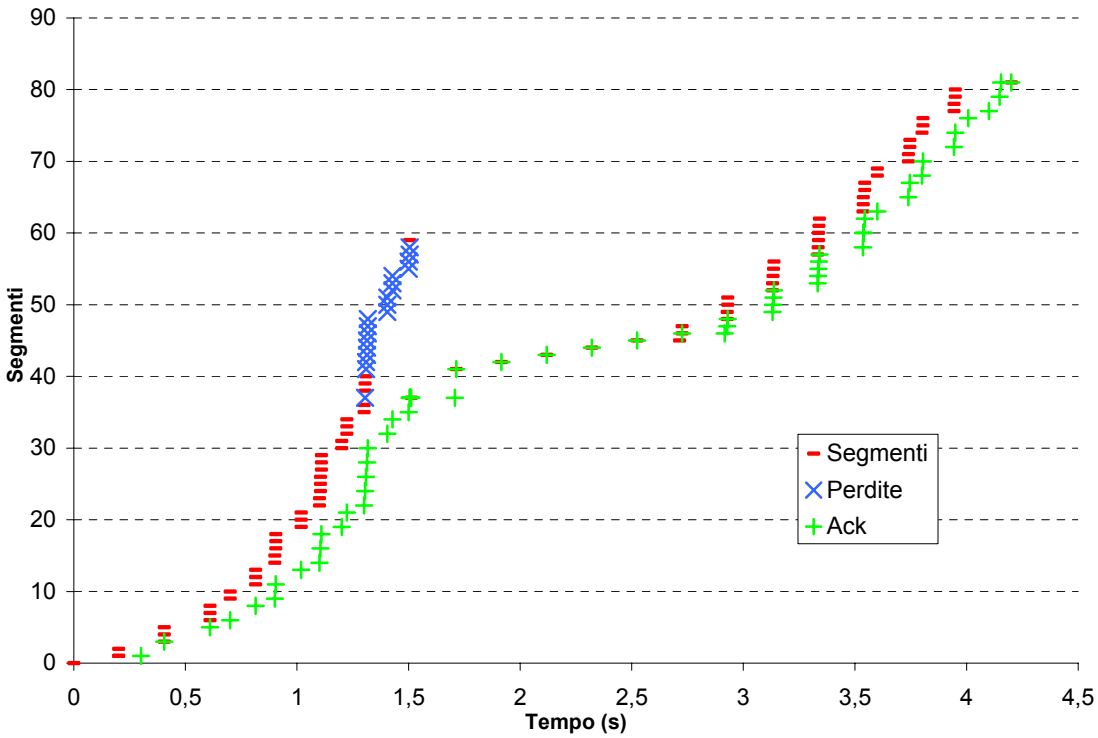

Figura 4.10. Comportamento della variante Impatient.
Lo scenario è stato creato con il file Tcl `NewReno_del_ack.tcl`.

Il trasmettitore, come indicato nella sezione 4 di RFC 3782, riarma il timer di ritrasmissione solo per il primo Ack parziale ricevuto. In quel momento, il valore di RTO (Retransmit TimeOut) è pari a 1 s. A causa del numero elevato di segmenti persi, il trasmettitore non riesce a ritrasmetterli tutti e a ricevere un riscontro entro il RTO. Di conseguenza, trascorso un secondo dalla ricezione del primo Ack parziale (t=1.71 s), si ha un timeout (t = 2.71 s). Per effetto del timeout, il trasmettitore esce dalla procedura di Fast Recovery, ritrasmette il primo segmento non riscontrato (seg. n. 45) ed entra in Slow Start. In un breve intervallo di tempo (meno di un secondo) tutti i 19 segmenti persi sono ritrasmessi e riscontrati dal ricevitore. La figura 4.11 mostra la sequenza di segmenti scambiati.

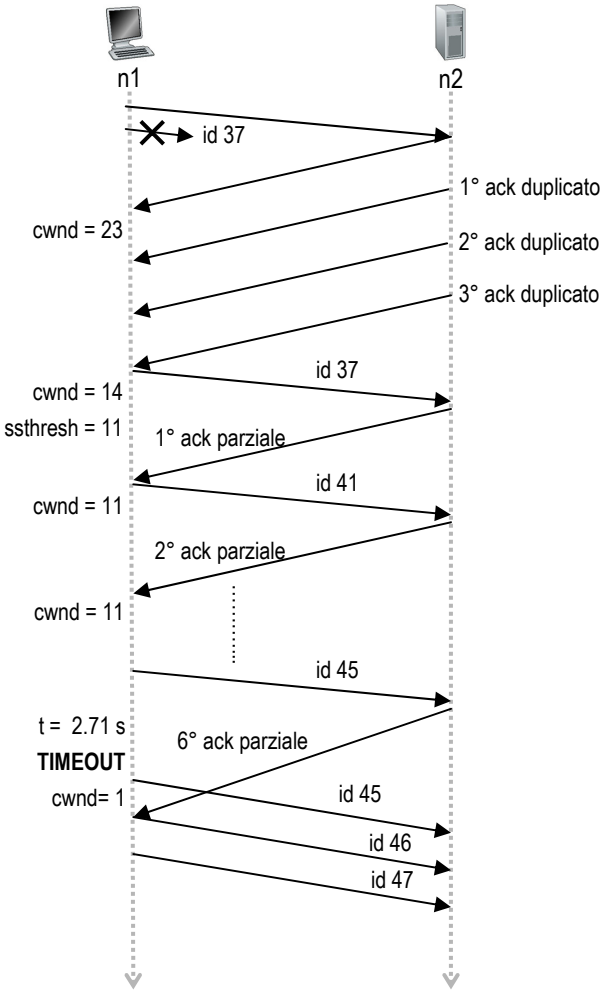

Figura 4.11. Sequenza dei segmenti scambiati adottando la variante Impatient.

4.3.2.3 Scenario della suite di test di ns-2

All'interno della directory dei sorgenti del simulatore, è incluso un insieme di scenari di comportamento del one-way TCP. Tra questi ve ne sono alcuni che consentono di confrontare le due varianti dell'algoritmo NewReno.

I test ‘`ns test-suite-newreno.tcl slow2`’ e ‘`ns test-suite-newreno.tcl impatient2`’ illustrano un caso nel quale, adottando la variante Slow but Steady, si ottengono prestazioni più elevate rispetto all'altra variante. Si ha quindi il caso duale rispetto agli scenari visti finora, nei quali la variante Impatient ha migliori prestazioni.

Lo scenario è costituito da quattro nodi con la seguente topologia:

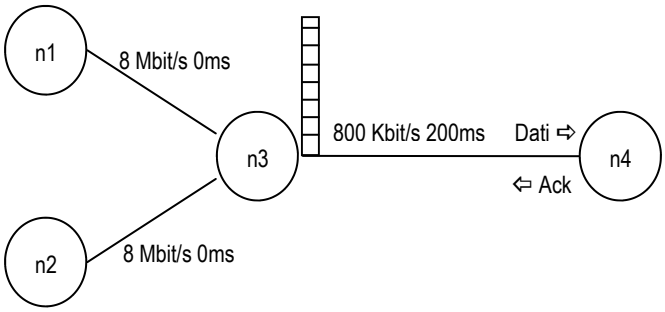

Figura 4.12: Scenario della suite di test di ns-2.

Il nodo n1 è il nodo mittente, il nodo n4 è il nodo ricevente. Il buffer associato al link che unisce i nodi n3 e n4 ha dimensione 8 segmenti. A tale link è anche associato un modello di errore, chiamato `ErrorModel/Periodic`. Come già accennato nel paragrafo 2.3.4, questo modello perde K segmenti dopo averne visti N ed è unito ad un classificatore flow-based per ottenere delle perdite in flussi particolari. In questo scenario, N = 14 e K = 5.

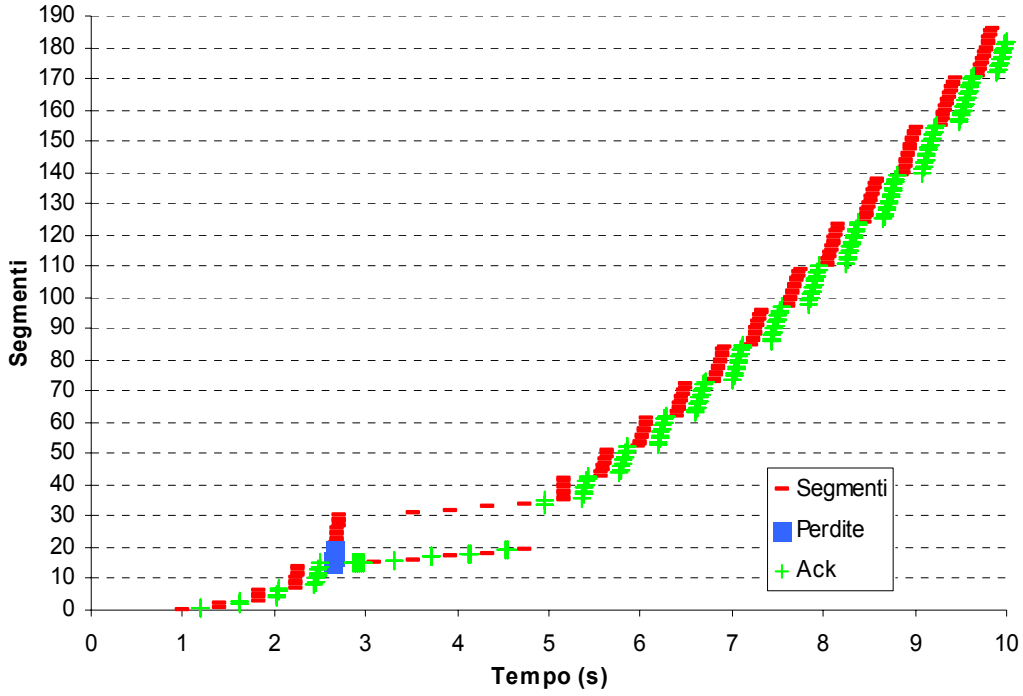

Figura 4.13. Comportamento della variante Slow but Steady.
Lo scenario è stato creato con il file Tcl `test-suite-newreno.tcl`.

Quando cwnd raggiunge il valore 9, si verificano delle perdite di segmenti. Precisamente, si perdono i segmenti con id 15, 16, 17, 18, 19. Il trasmettitore entra nella procedura di Fast Recovery dopo aver ricevuto tre Ack duplicati, ritrasmette il segmento perso (segmento con id. 15) e imposta ssthresh al valore 8 e cwnd al valore 11. Il trasmettitore resta in Fast Recovery per circa 2 secondi, ritrasmettendo ad ogni RTT solamente un segmento. Quando all'istante t = 5.15 s il trasmettitore riceve il primo Ack completo, esce dalla procedura di Fast Recovery con cwnd = 8. Quindi invia 8 segmenti, come si vede in figura 4.13.

Anche se la procedura di Fast Recovery ha una durata di circa 2 secondi, le prestazioni ottenute sono decisamente più elevate di quelle che si ottengono adottando la variante Impatient (vedi figura 4.14).

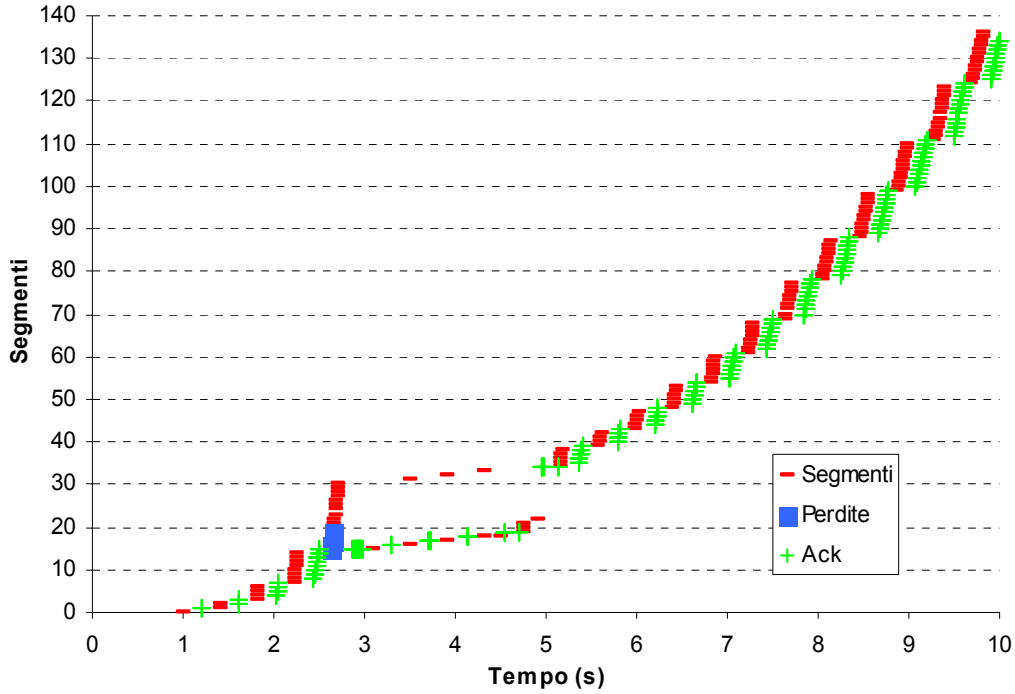

Figura 4.14. Comportamento della variante Impatient.
Lo scenario è stato creato con il file Tcl `test-suite-newreno.tcl`.

Usando questa seconda variante, infatti, in corrispondenza della prima perdita, il trasmettitore ritrasmette il segmento con id. 15. Ricevuto il primo Ack parziale, riarma il timer di ritrasmissione. Risultano ancora da recuperare 4 segmenti: il trasmettitore, però, non ha la possibilità di inviarli tutti e ricevere un riscontro prima che scada il timer di ritrasmissione (pari a 1 s). Scaduto il timer di ritrasmissione all'istante t = 4.5 s, viene ritrasmesso il primo segmento non ancora riscontrato, cioè il segmento con id 18, riducendo cwnd a 1 segmento e ssthresh a 4 segmenti. A causa del timeout, sono ritrasmessi anche alcuni segmenti che erano già arrivati a destinazione (ad esempio i segmenti con id 20, 21, 22).

Si può concludere che in questo scenario il timeout rallenta notevolmente il trasmettitore (vedi grafico in figura 4.14). Questo avviene perché c'è un bilanciamento possibile tra aspettare, come fa Slow but Steady, in modo da recuperare tutti i segmenti mancanti mantenendo aperta una finestra già grande, o non aspettare, come fa Impatient, rinunciando alla finestra ampia. La seconda strategia è premiante quando il numero dei segmenti persi è grande o la finestra non è abbastanza grande che possa convenire attendere.

Capitolo 5 Esperimenti in ns-2

5.1 Introduzione

In questo studio siamo interessati al comportamento di una connessione TCP su canali satellitari geostazionari a banda larga e privi di errore, per valutare se le comuni implementazioni del TCP siano in grado di sfruttare appieno un simile link. Lo studio utilizza il simulatore ns-2, descritto nel capitolo 3 per analizzare le principali varianti del protocollo TCP (Reno, NewReno e Sack).

Lo scenario costruito (vedi appendice) si compone di un nodo a_1 e dei nodi n3 e n4 (vedi fig. 5.1) e simula un "bulk data transfer" tra il nodo a_1 e il nodo n4. Il nodo a_1 è collegato al nodo n3 mediante un link con bw = 100 Mbit/s e delay = 1 ms. Il collegamento tra i nodi n3 e n4 simula il canale satellitare a larga banda. Questo link è il cosiddetto "bottleneck link" (strozzatura), in quanto i valori possibili per bw sono compresi tra 1 Mbit/s e 8 Mbit/s, mentre per il delay il valore tipico è di 250 ms. La dimensione del buffer associato al link è pari al prodotto 2*delay*bw*β, dove β è un parametro i cui valori tipici sono compresi tra 0.5 e 2. I due estremi della connessione utilizzano finestre sufficientemente grandi, in modo da non costituire un limite superiore per il numero massimo di segmenti da inviare.

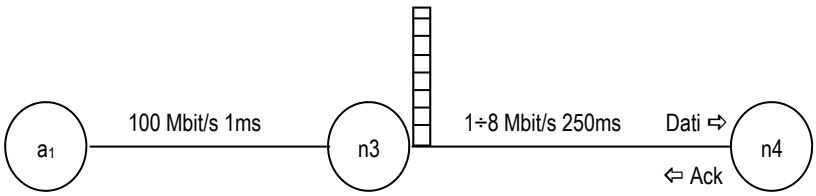

Figura 5.1. Scenario per il confronto tra gli algoritmi Reno, NewReno e Sack.

Si è posta particolare attenzione alla perdita del primo burst di dati e alle reazioni delle principali varianti del protocollo TCP: Reno, NewReno e Sack.

5.2 Studio del comportamento dell'algoritmo Reno

Il comportamento dell'algoritmo Reno è stato analizzato nel caso in cui il bottleneck link ha le seguenti caratteristiche: larghezza di banda pari a 2 Mbit/s, ritardo di 250ms e fattore beta uguale a 1. In questo modo la dimensione del buffer associato a tale link è pari al prodotto bw*RTT*beta, ovvero 125000 byte.

La perdita dei primi segmenti si ha quando cwnd diventa maggiore di 2*bw*RTT*beta. In questo specifico scenario, ciò avviene quando cwnd è maggiore di 250 segmenti, considerati segmenti di 1000 byte.

A causa della saturazione del buffer associato alla strozzatura, si perde un numero molto elevato di segmenti. Come si può notare dal grafico 5.2 e dalla sequenza di segmenti scambiati (vedi fig. 5.3), il primo segmento si perde all'istante t = 6.76 s. Ricevuti i segmenti successivi a quello perso, il ricevitore si accorge di un salto nello spazio di sequenza e quindi invia degli Ack duplicati. Alla ricezione del terzo Ack duplicato, il trasmettitore deduce che uno o più segmenti si sono persi ed entra nella procedura di Fast Retransmit e Fast Recovery. ssthresh è posta al valore corrente di cwnd (190 segmenti), si ritrasmette il segmento perso (segmento con id 726) e cwnd è ridotta a 193 segmenti. Nel corso della procedura di Fast Recovery, il trasmettitore riceve ulteriori Ack duplicati, che fanno aumentare cwnd fino ad un massimo di 440 segmenti e permettono l'invio di nuovi segmenti (i segmenti con id compreso tra 1107 e 1165).

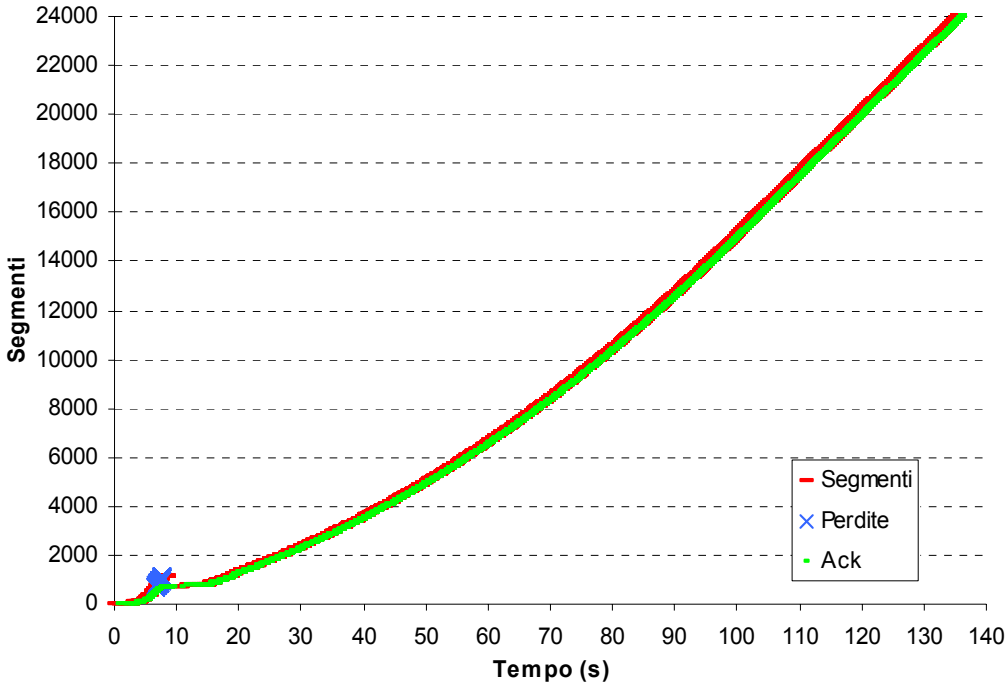

Figura 5.2. Comportamento dell'algoritmo Reno.
Lo scenario è stato creato con il file Tcl `satellite.tcl`.

La ricezione del primo Ack non duplicato indica al trasmettitore che il segmento in precedenza ritrasmesso è arrivato a destinazione; esce, dunque, dalla procedura di Fast Recovery, impostando cwnd al valore di ssthresh.

La ritrasmissione degli altri segmenti persi può avvenire solamente se si ricevono almeno tre Ack duplicati. Ad esempio, il secondo segmento perso (id 729) è ritrasmesso non appena si ricevono tre Ack duplicati in cui il campo Acknowledgment Number è pari al numero di sequenza del segmento perso (cioè 698881). Come prima, ssthresh è posta alla metà del valore corrente di cwnd (95 segmenti) e cwnd al valore di ssthresh più tre segmenti (98 segmenti). Il trasmettitore è di nuovo nella procedura di Fast Retransmit e Fast Recovery. Gli Ack duplicati ricevuti non permettono l'invio di nuovi segmenti, poiché cwnd raggiunge la dimensione massima di 154 segmenti e tutti i segmenti all'interno della finestra di congestione risultano essere stati già inviati. Alla ricezione del primo Ack non duplicato, il trasmettitore esce dal recupero con cwnd uguale a 95 segmenti.

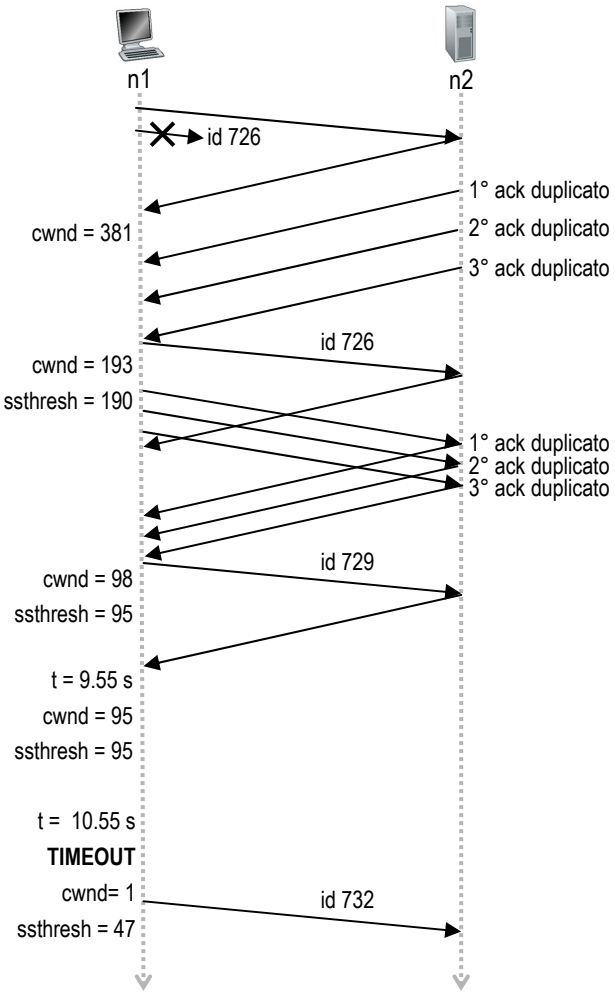

Figura 5.3. Sequenza dei segmenti scambiati con l'algoritmo Reno.

L'assenza di Ack duplicati successivi non consente al trasmettitore di ritrasmettere ulteriori segmenti persi. Scaduto il timer di ritrasmissione (pari a 1 secondo), il

trasmettitore ritrasmette il primo segmento non riscontrato, quello con id 732. Il timeout provoca la riduzione di cwnd a 1 segmento e di ssthresh a 47 segmenti. Molti dei segmenti trasmessi in seguito al timeout risultano essere già stati ricevuti dal ricevitore ma, non avendo un riscontro per questi, sono nuovamente inviati. Il recupero del primo burst di segmenti persi si può considerare concluso all'istante t = 18.6 s, dove viene registrato l'arrivo a destinazione dell'ultimo segmento perso.

Essendo `cwnd < ssthresh`, in seguito al timeout si utilizza l'algoritmo Slow Start per regolare l'aumento di cwnd in risposta agli Ack ricevuti. Come già descritto nel paragrafo 2.7, l'algoritmo Slow Start aumenta cwnd di un segmento per ogni Ack non duplicato ricevuto. Questo è un incremento esponenziale nel tempo, diversamente dall'incremento lineare nel tempo dell'algoritmo Congestion Avoidance. Si veda a tal proposito il grafico 5.4 che traccia l'andamento di cwnd nel tempo per tutte le varianti del protocollo TCP analizzate in questo capitolo.

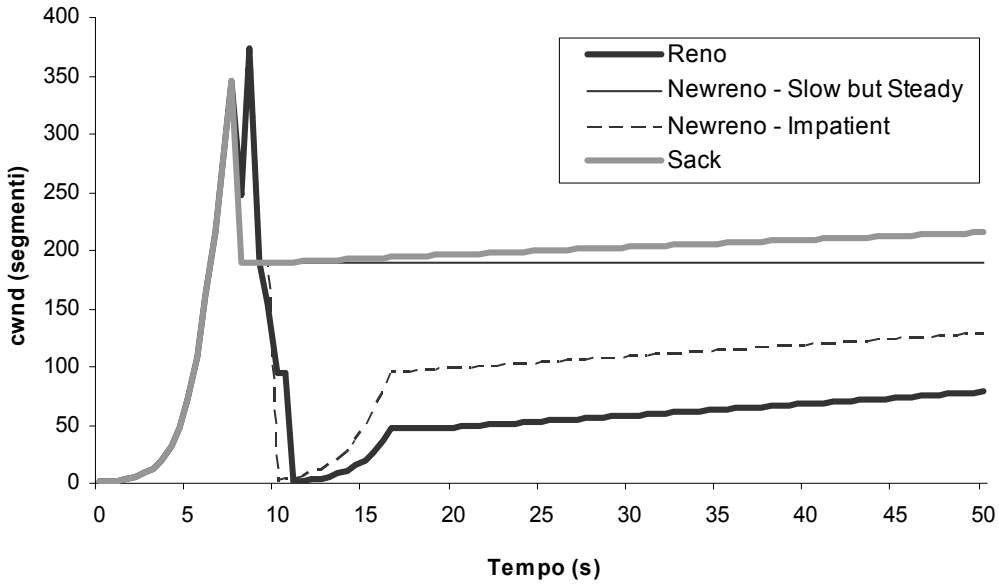

Figura 5.4. Grafico comparativo dell'andamento della finestra di congestione in uno scenario in cui bw = 2 Mbit/s, delay = 250 ms e beta = 1.

5.3 Studio del comportamento dell'algoritmo NewReno

L'algoritmo NewReno, come descritto nel paragrafo 2.8.2, definisce due diverse varianti: Slow but Steady e Impatient. La prima riarma il timer di ritrasmissione alla ricezione di ogni Ack parziale, la seconda soltanto alla ricezione del primo Ack parziale. Il comportamento di una connessione TCP con le due varianti è discusso nei due sottoparagrafi seguenti.

5.3.1 Variante Slow but Steady

Lo scenario considerato è uguale al precedente, tranne che per l'utilizzo di un diverso algoritmo.

La perdita dei primi segmenti si verifica all'istante t = 6.76 s. Il trasmettitore, mediante la ricezione di tre Ack duplicati, deduce che alcuni segmenti si sono persi. Ritrasmette quindi il primo andato perso (quello con id 726), riducendo cwnd a 193 segmenti e ssthresh a 190 segmenti. L'arrivo di ulteriori 247 Ack duplicati durante la procedura di Fast Recovery porta cwnd ad un valore massimo pari a 440 segmenti e all'invio dei segmenti con numero di sequenza compreso tra 1107 e 1165.

La ricezione del primo Ack non duplicato indica al trasmettitore che vi sono altri segmenti da ritrasmettere. cwnd è ridotta a 190 segmenti, il valore attuale di ssthresh. Viene ritrasmesso il secondo segmento perso (quello con id 729) e, prima di ricevere un Ack parziale, cwnd risulta pari a 249 segmenti. Questo sostanziale incremento, però, non permette di inviare nuovi segmenti durante la procedura di Fast Recovery, poiché il più alto numero di sequenza in cwnd è quello con id 977 e in precedenza si erano trasmessi segmenti con numero di sequenza maggiore.

L'invio, in questo RTT e nei successivi, di un solo segmento (ovvero la ritrasmissione di quello perso) è dovuto in parte alle dimensioni ridotte della finestra di congestione (soli 190 segmenti) e in parte alla non ricezione di ulteriori Ack duplicati. Questo fenomeno si protrae per un numero di secondi che è direttamente proporzionale alla larghezza di banda. Nel caso di bw = 2 Mbit/s e delay = 250 ms, il fenomeno sopra descritto ha una durata di 43 secondi: bisogna attendere, infatti, la ritrasmissione del segmento con id 978 avvenuta all'istante t = 51.8 s perchè siano inviati due nuovi segmenti.

Per dimostrare che tale fenomeno ha durata crescente al crescere della dimensione del canale, si è considerato uno scenario con larghezza di banda maggiore, ad esempio 8 Mbit/s. Si è potuto constatare che, anche in questo caso, tale fenomeno si verifica a partire dalla ritrasmissione del secondo segmento perso e ha una durata di 167 secondi, 4 volte più grande di quella registrata con bw = 2 Mbit/s.

I grafici qui riportati mostrano il comportamento disastroso dell'algoritmo NewReno nel caso di perdita di un numero elevato di segmenti.

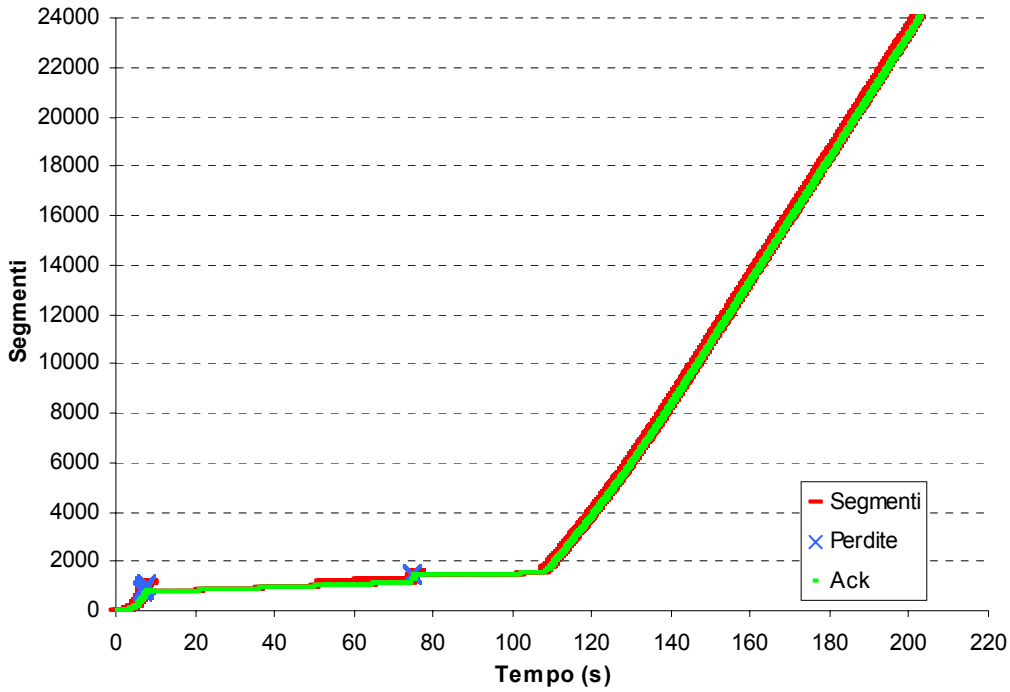

Figura 5.5. Comportamento dell'algoritmo NewReno (variante Slow but Steady) con bw = 2 Mbit/s. Lo scenario è stato creato con il file Tcl `satellite.tcl`.

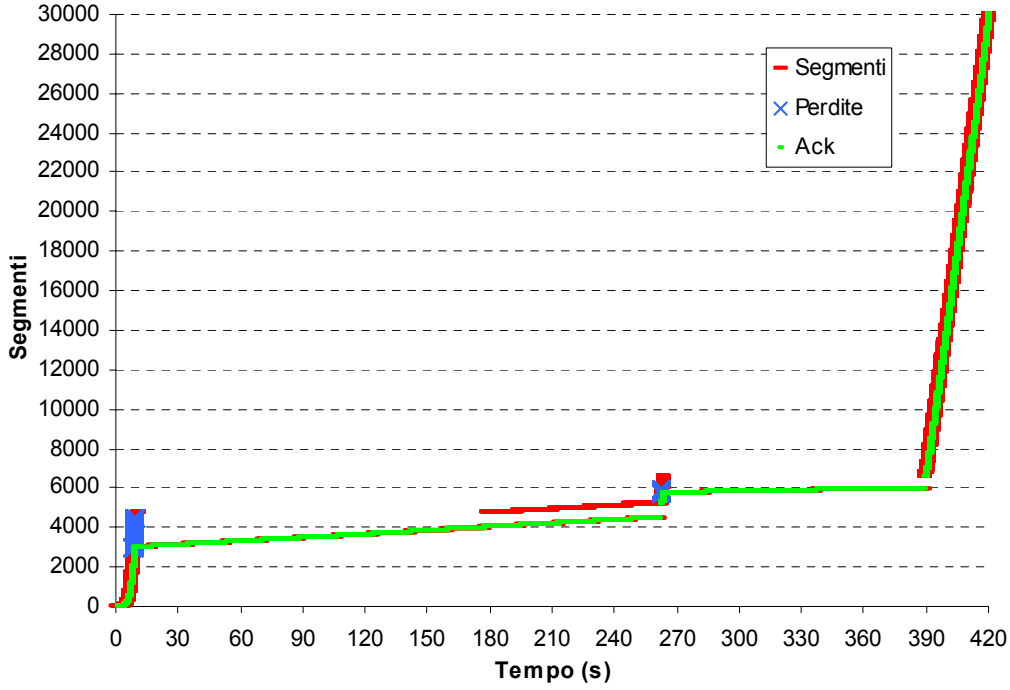

Figura 5.6. Comportamento dell'algoritmo NewReno (variante Slow but Steady) con bw = 8 Mbit/s. Lo scenario è stato creato con il file Tcl `satellite.tcl`.

5.3.2 Variante Impatient

La variante Impatient effettua il riarmo del timer di ritrasmissione solo per il primo Ack parziale ricevuto. Se entro il retransmit timeout (1 secondo) non si riceve un Ack completo (vedi paragrafo 2.8.2), si verifica un timeout con conseguente uscita dalla procedura di Fast Recovery. Anche se il trasmettitore riduce la finestra di congestione a un segmento, dimezzando ssthresh, il recupero del primo burst di segmenti persi avviene molto più velocemente rispetto alla variante Slow but Steady.

Infatti, all'istante t = 16.74 s, l'ultimo segmento perso arriva a destinazione. Nella variante Slow but Steady, l'invio di un solo segmento per RTT rallenta notevolmente la fase di recupero dei segmenti persi, spostando l'istante di arrivo dell'ultimo segmento perso addirittura a t = 74.33 s.

Come si può notare dal grafico della finestra di congestione riportato in figura 5.4, all'istante t = 9.77 s si ha un timeout. ssthresh è ridotta a 95 segmenti, il segmento inviato è il primo non riscontrato (quello con id 735) e si utilizza l'algoritmo Slow Start fino al raggiungimento della soglia.

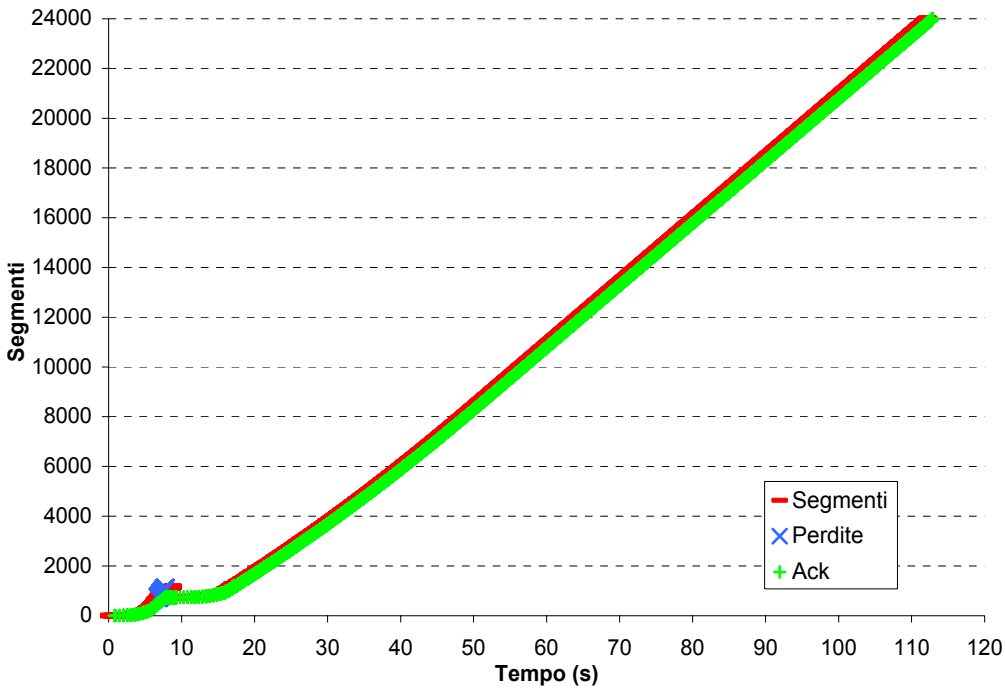

Figura 5.7. Comportamento dell'algoritmo NewReno (variante Impatient) con bw=2Mbit/s. Lo scenario è stato creato con il file Tcl `satellite.tcl`.

5.4 Studio del comportamento dell'algoritmo Sack

Il documento di riferimento dell'algoritmo Sack ([RFC2018]), a differenza di quelli relativi agli algoritmi Reno e NewReno, non fornisce alcun dettaglio implementativo dell'algoritmo stesso.

Ai fini dello studio del comportamento del Sack nello scenario descritto nel paragrafo 5.1, si sono perseguiti i seguenti due obiettivi:

- approfondire lo studio dell'implementazione dell'algoritmo adottata dal simulatore (vedi paragrafo 5.4.1);
- analizzare il comportamento dell'algoritmo variando opportunamente alcuni parametri quali la larghezza di banda, il ritardo, il fattore beta, per mettere in luce aspetti significativi (vedi paragrafo 5.4.2).

5.4.1 Implementazione dell'algoritmo in ns-2

Da quanto riportato nel paragrafo 2.8.3, un blocco Sack riporta un insieme di dati non contigui che sono stati ricevuti e accodati dal ricevitore. Il primo blocco deve necessariamente contenere il segmento ricevuto più di recente, mentre gli altri devono riportare i blocchi Sack più recenti. Nel simulatore, l'opzione Sack ha a disposizione soltanto 26 byte, che permettono di indicare tre blocchi di dati non contigui.

Gli algoritmi di controllo della congestione implementati nel Sack costituiscono un'estensione conservativa di quelli usati nel Reno, poiché sono utilizzati gli stessi algoritmi per l'incremento e il decremento della finestra di congestione. La principale differenza tra le due varianti è da ricercarsi nelle azioni effettuate quando sono persi più segmenti all'interno di una finestra di dati.

Come negli algoritmi Reno e NewReno, l'implementazione del Sack entra nella procedura di Fast Recovery quando il trasmettitore riceve almeno tre Ack duplicati. Si ritrasmette il segmento perso e si dimezza la finestra di congestione. Durante il Fast Recovery, il Sack mantiene una variabile chiamata `pipe` che rappresenta il numero stimato di segmenti pendenti nella rete. Il trasmettitore invia nuovi segmenti o segmenti ritrasmessi soltanto quando il numero stimato di segmenti nel percorso risulta minore della finestra di congestione (`pipe < cwnd`). La variabile `pipe` è incrementata di un'unità dopo l'invio di un nuovo segmento o la ritrasmissione di uno vecchio ed è decrementata di un'unità per ogni Ack duplicato ricevuto contenente blocchi Sack.

Con l'uso della variabile `pipe`, si vogliono mantenere distinti due concetti: quando inviare segmenti e quali segmenti inviare. Il trasmettitore mantiene una struttura dati, chiamata scoreboard, che memorizza tutti gli Ack ricevuti in precedenza. Quando il trasmettitore ha la possibilità di inviare segmenti, ritrasmette il primo segmento estratto dalla lista di quelli che si suppone siano persi. Se lo score-

board è vuoto e la finestra dichiarata dal ricevitore è sufficientemente grande, invia un nuovo segmento.

Se un segmento ritrasmesso va perso, l'implementazione del Sack si accorge della perdita mediante un timeout di ritrasmissione, ritrasmettendo il segmento perso e adottando l'algoritmo Slow Start.

Il trasmettitore esce dalla procedura di Fast Recovery solo dopo aver ricevuto un Ack completo, un Ack che riscontra tutti i segmenti che erano pendenti nel momento in cui è entrato in Fast Recovery.

Il Sack riserva un trattamento particolare agli Ack parziali (gli Ack ricevuti durante il Fast Recovery che permettono un avanzamento del campo Acknowledgment Number, ma che non portano il trasmettitore al di fuori della procedura di Fast Recovery). Per tali Ack, il trasmettitore decrementa la variabile `pipe` di due unità, anziché una. L'arrivo di un Ack parziale rappresenta, infatti, l'uscita dal pipe di due segmenti: quello perso e quello appena ritrasmesso.

5.4.2 Simulazioni

Si è studiato in prima istanza il comportamento dell'algoritmo Sack in un contesto piuttosto ampio, ottenuto variando alcuni parametri quali la larghezza di banda, il ritardo, il fattore beta. Per la larghezza di banda si sono considerati i valori 2, 4, 8 Mbit/s; per il ritardo i valori 250, 300, 400 ms; per il fattore beta i valori 1 e 2.
Fissando il valore di beta a 2, si è notato che in alcuni casi si verificano dei timeout, come riportato in tabella 5.1:

Bandwidth	Delay	Beta	Timeout?
2Mb/s	250ms	2	X
2Mb/s	300ms	2	
2Mb/s	400ms	2	X
4Mb/s	250ms	2	X
4Mb/s	300ms	2	
4Mb/s	400ms	2	X
8Mb/s	250ms	2	
8Mb/s	300ms	2	X
8Mb/s	400ms	2	

Tabella 5.1. Fenomeno dei timeout al variare dei parametri dello scenario.

In cinque dei nove casi considerati, si verifica un timeout. Per via del timeout, il trasmettitore perde tutte le informazioni contenute nello scoreboard. Si ritrasmettono, dunque, tutti i segmenti persi ma anche (e purtroppo) segmenti che erano già stati ri-

cevuti correttamente a destinazione. Questo può essere impedito se viene posta a false la variabile OTcl `clear_on_timeout`.

Dalla tabella 5.1 si può evincere un comportamento anomalo del timeout: il suo verificarsi, cioè, non dipende in maniera monotona, a parità di larghezza di banda, dall'aumento o dalla riduzione del ritardo.

Si è dunque posta l'attenzione principalmente al problema dell'aggiornamento del valore del Retransmit TimeOut. Tale quantità è calcolata a partire dal valore del Round Trip Time misurato, come specificato nelle equazioni del paragrafo 2.6. Il calcolo del valore del RTT non è fatto per tutti i segmenti inviati. Ad esempio con bw = 2 Mbit/s, delay = 250 ms e beta = 2, il calcolo del RTT e l'aggiornamento del valore del RTO avviene solo per i segmenti 1, 2, 4, 7 come in fig. 5.8.

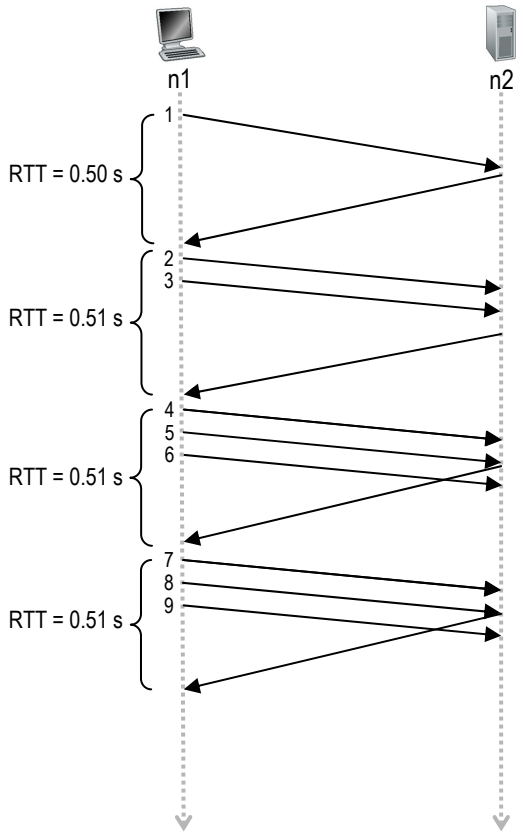

Figura 5.8. Misura del RTT nel caso di bw = 2 Mbit/s, delay = 250 ms, beta = 2.

Nei casi in cui si verifica il timeout, il segmento impiegato per l'ultimo aggiornamento del valore del RTO prima del timeout stesso è inserito nel buffer associato al bottleneck link quando questo risulta pieno al 50% circa. L'aumento di RTO rispetto al valore precedentemente calcolato non risulta sufficientemente consistente da evitare il timeout in seguito alla ritrasmissione del primo segmento perso. Nei casi in cui non si verifica il timeout, invece, il segmento impiegato per la misurazione del RTT è inserito nel buffer quando risulta pieno all'80% circa. Il valore più elevato del RTT

misurato provoca un considerevole aumento di RTO e l'assenza di un timeout successivo.
Il verificarsi o meno del timeout, in conclusione, non sembra dipendere da particolari valori della larghezza di banda, del ritardo, del fattore beta, bensì dai segmenti impiegati per il calcolo del RTT e l'aggiornamento successivo di RTO. Si supponga di essere nel caso in cui bw = 2 Mbit/s, delay = 400 ms e beta = 2. Se prima di ricevere un Ack duplicato si fosse tenuto conto, nel calcolo del RTT, di un diverso segmento, si sarebbe avuto un valore di RTO abbastanza alto da impedire il timeout in seguito alla ritrasmissione del primo segmento perso.

5.5 Conclusioni

Lo studio qui svolto sul comportamento di una connessione TCP su canali satellitari geostazionari a larga banda ha cercato di mostrare, per ogni variante del protocollo, vantaggi e svantaggi. Individuare quale sia in assoluto la variante con prestazioni migliori non è fattibile per numerosi fattori in gioco: larghezza di banda, ritardo, fattore beta.

Limitatamente al caso più frequente analizzato, il caso in cui bw = 2 Mbit/s, delay = 250 ms e beta = 1, si è costruito il grafico riportato in figura 5.9.

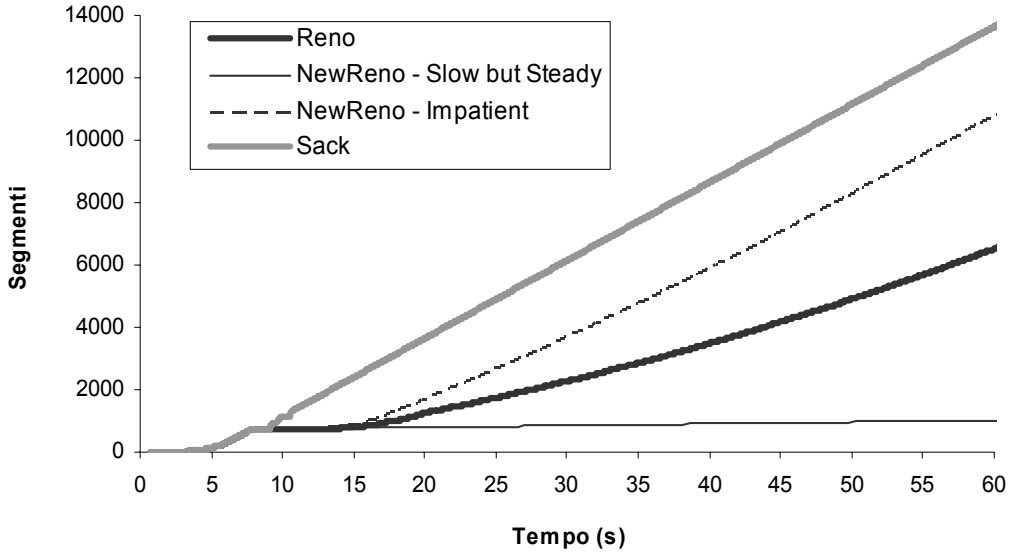

Figura 5.9. Grafico comparativo degli Ack ricevuti in uno scenario in cui bw = 2 Mbit/s, delay = 250 ms e beta = 1.

Tale grafico confronta le varianti Reno, NewReno (Slow but Steady e Impatient), Sack dal punto di vista degli Ack ricevuti. Le prestazioni più scadenti si registrano per la variante Slow but Steady del NewReno, a causa dell'elevato numero di segmenti persi (131 in totale), come già discusso nel paragrafo 5.3.1. Il riarmo del timer

di ritrasmissione soltanto per il primo Ack parziale (variante Impatient del NewReno) aiuta ad ottenere prestazioni più elevate. L'algoritmo Sack, in questo scenario, si dimostra la soluzione vincente.

Capitolo 6 Controllo della congestione negli stack TCP FreeBSD e Linux

6.1 Introduzione

In questo capitolo sono discusse le scelte implementative riguardanti il controllo della congestione adottate da due stack TCP reali: FreeBSD e Linux. Questa trattazione è basata sull'ispezione del codice sorgente.

6.2 FreeBSD

L'implementazione nel simulatore ns-2 degli algoritmi di controllo della congestione segue abbastanza fedelmente quanto descritto nel kernel presente in FreeBSD 5.3.

Nei sorgenti del kernel, il file `tcp_input.c` contenuto nella directory `sys/netinet` descrive tutte le operazioni da effettuare nel caso della ricezione di un segmento.

Qui è definita, e posta uguale a 3, una variabile chiamata `tcprexmtthresh`, che indica quanti Ack duplicati sono richiesti per entrare nella procedura di Fast Recovery.

Alla ricezione di tre Ack duplicati, nello stato ESTABLISHED, se è adottato l'algoritmo NewReno, si controlla se `ackno <= recover`. In tal caso è azzerata la variabile che mantiene il numero degli Ack duplicati e non si effettuano altre operazioni. La stessa cosa accade se è adottato l'algoritmo Sack e si è già in Fast Recovery.

```
if (tp->sack_enable) {
      if (IN_FASTRECOVERY(tp)) {
            tp->t_dupacks = 0;
            break;
      }
} else if (tcp_do_newreno) {
      if (SEQ_LEQ(th->th_ack,
          tp->snd_recover)) {
            tp->t_dupacks = 0;
            break;
      }
```

```
}
```

Se, invece, le condizioni sopra riportate non risultano verificate, si entra nella procedura di Fast Recovery, aggiornando opportunamente il valore della finestra di congestione.

Quando il ricevitore riceve un Ack parziale, tale per cui `ackno > snd_una` e `ack < snd_recover`, è invocato un metodo diverso a seconda dell'algoritmo utilizzato:

- `tcp_sack_partial_ack` nel caso del Sack;
- `tcp_newreno_partial_ack` nel caso del NewReno.

Nella definizione di quest'ultimo metodo, è riportato il seguente commento:

```
/*
 * On a partial ack arrives, force the retransmission of the
 * next unacknowledged segment.  Do not clear tp->t_dupacks.
 * By setting snd_nxt to ti_ack, this forces retransmission
 * timer to be started again.
 */
```

Da questo commento e dal codice del metodo, si può dedurre che l'unica variante dell'algoritmo NewReno implementata in un kernel FreeBSD è la variante Slow but Steady, che comporta il riarmo del timer di ritrasmissione ad ogni Ack parziale.

La scelta dell'algoritmo di controllo della congestione è possibile grazie alla chiamata di sistema sysctl. sysctl è un'interessante funzionalità offerta dal mondo Unix per modificare a runtime i parametri del kernel (parametri della rete, della memoria virtuale, del filesystem) senza dover ricompilare l'intero kernel. I parametri sono circa 500 e possono essere elencati mediante l'opzione `-a`. Gli algoritmi NewReno e Sack possono essere abilitati mediante due variabili: `net.inet.tcp.newreno` e `net.inet.tcp.sack`.

6.3 Linux

L'implementazione del protocollo TCP in Linux, sebbene conforme ai principi di controllo della congestione dettati dagli RFC, adotta un differente approccio (come descritto ampiamente in [SK02]).

6.3.1 Controllo della congestione: l'approccio di Linux

Le specifiche del TCP ma anche alcune sue implementazioni esprimono la finestra di congestione in byte; in Linux, invece, viene espressa in unità di full-sized

segment. Nel caso siano utilizzati segmenti di dimensioni ridotte, in un'implementazione byte-based possono essere trasmessi più segmenti per ogni full-sized segment nella finestra di congestione; Linux, invece, consente ad un solo segmento di essere inviato. Questo risulta in un approccio più conservativo, se rapportato all'approccio byte-based.

Nel prendere decisioni sul numero dei segmenti da inviare, il trasmettitore Linux confronta la finestra di congestione con il numero di segmenti pendenti, anziché confrontarla con la differenza tra `SND.NXT` e `SND.UNA`. Si utilizza la seguente formula per determinare il numero di segmenti pendenti nella rete:

```
in_flight = packets_out - left_out + retrans_out
```

dove `left_out = sacked_out + lost_out`

`packets_out` è il numero dei segmenti trasmessi (cioè la differenza tra `SND_NXT` e `SND_UNA` espressa in numero di segmenti). `sacked_out` è il numero di segmenti riscontrati dai blocchi Sack (in assenza di informazioni Sack, tale variabile è incrementata di un'unità per ogni Ack duplicato ricevuto). `lost_out` rappresenta una stima del numero di segmenti persi nella rete, mentre `retrans_out` è il numero di segmenti ritrasmessi. La determinazione del valore della variabile `lost_out` dipende dal metodo di recupero adottato.

I contatori necessari per tracciare il numero di segmenti pendenti, riscontrati, persi o ritrasmessi richiedono delle strutture dati aggiuntive. In Linux lo stato di ogni segmento pendente è mantenuto in una struttura dati chiamata scoreboard. Mediante questa struttura dati, il trasmettitore Linux sa quali sono i segmenti che devono essere ritrasmessi.

Indicare quando un segmento è pendente, riscontrato o ritrasmesso è semplice. Lo stesso non si può dire per un segmento perso. Questo dipende dal metodo di recupero usato. Ad esempio, con l'algoritmo NewReno, il primo segmento non riscontrato è marcato come perso quando il trasmettitore entra nella procedura di Fast Recovery. In effetti, ciò corrisponde alle specifiche date da RFC. Se durante il Fast Recovery arriva un Ack parziale, il primo segmento non riscontrato va marcato come perso. Se è usato l'algoritmo Sack, invece, più di un segmento alla volta può essere marcato come perso.

Le azioni intraprese dal trasmettitore Linux in risposta agli Ack ricevuti sono governate da una macchina a stati con i seguenti stati:

- **Open**. Questo è lo stato nel quale il trasmettitore TCP segue il normale percorso di esecuzione per il processing degli Ack in entrata. Quando arriva un Ack, il trasmettitore incrementa la finestra di congestione a seconda dell'algoritmo in uso (Slow Start o Congestion Avoidance).

- **Disorder**. Quando il trasmettitore individua degli Ack duplicati o Ack selettivi, si sposta nello stato Disorder. La finestra di congestione non viene modificata, ma ogni segmento in entrata fa scattare la trasmissione di un nuovo segmento. Il trasmettitore segue il cosiddetto principio di conservazione del segmento [Jac88], il quale stabilisce che un nuovo segmento non è emesso finché uno vecchio ha lasciato la rete.
- **CWR** (Congestion Window Reduced). Il trasmettitore può ricevere notifiche esplicite di congestione (ECN). In questo caso non riduce la finestra di congestione in un sol colpo, ma di un segmento per ogni due Ack ricevuti finché la sua dimensione risulta dimezzata. Questo stato può essere interrotto dagli stati Recovery e Loss descritti qui di seguito.
- **Recovery**. In seguito alla ricezione di un numero sufficiente di Ack duplicati consecutivi, il trasmettitore ritrasmette il primo segmento non ancora riscontrato ed entra nello stato Recovery. Per default, la soglia per entrare in questo stato è di tre Ack duplicati, un valore raccomandato dalle specifiche del protocollo TCP. Durante questo stato, la dimensione della finestra di congestione è ridotta di un segmento per ogni due Ack ricevuti, similmente allo stato precedente. La riduzione della finestra termina quando la sua dimensione risulta pari a quella di slow start threshold (la metà del valore della finestra di congestione quando il trasmettitore è entrato in questo stato). La finestra di congestione non subisce alcun aumento e il trasmettitore o invia i segmenti marcati come persi, oppure ne invia di nuovi, rispettando sempre il principio di conservazione del segmento. Il trasmettitore resta nello stato Recovery fino a quando tutti i segmenti che erano pendenti quando è entrato in questo stato sono riscontrati. Successivamente torna nello stato Open.
- **Loss**. Quando si ha un timeout di ritrasmissione, il trasmettitore entra nello stato Loss. Tutti i segmenti pendenti sono marcati come persi e la finestra di congestione è ridotta ad un segmento. A differenza dello stato precedente, la finestra di congestione può essere incrementata. Lo stato Loss non può essere interrotto da nessuno degli altri stati; il trasmettitore ritorna nello stato Open solo dopo aver ritrasmesso tutti i segmenti che in precedenza sono stati marcati come persi.

6.3.2 Conformità alle specifiche

Poiché Linux combina le caratteristiche specificate in differenti RFC con le scelte di progetto descritte nel precedente paragrafo, alcuni algoritmi presentati negli RFC non sono interamente implementati nel kernel Linux. La tabella 6.1 mostra quali indicazioni degli RFC relative al controllo della congestione del TCP sono implemen-

tate in Linux. Alcune di queste sono contrassegnate da un asterisco in quanto implementate, ma con alcune differenze.

La procedura di Fast Recovery di Linux, ad esempio, non segue con precisione il comportamento descritto in RFC 3782. Innanzitutto il trasmettitore regola la soglia per far scattare il Fast Retransmit dinamicamente, basandosi sul riordinamento osservato nella rete. Quindi, è possibile che il terzo Ack duplicato non dia inizio, in tutti i casi, ad una ritrasmissione veloce. Inoltre il trasmettitore Linux non incrementa artificialmente la finestra di congestione durante la procedura di Fast Recovery per ogni Ack duplicato ricevuto. Il differente approccio da solo non dovrebbe causare effetti significativi sulla performance del TCP. Quando si entra in Fast Recovery, il trasmettitore non riduce la finestra di congestione in un sol colpo, come suggerisce RFC 3782: la finestra di congestione è ridotta di un segmento per ogni due Ack ricevuti, fino al raggiungimento della metà del valore iniziale. Questa tecnica è nota con il nome di "Rate-halving".

Specifiche	**Stato**
RFC 2018 (Sack)	+
RFC 2581 (Controllo della congestione)	*
RFC 3782 (NewReno)	*
RFC 2988 (RTO)	*

Tabella 6.1. Specifiche del controllo della congestione implementate in Linux.
+ = implementate, * = implementate, ma i dettagli differiscono dalle specifiche.

La stima del RTT e il calcolo del RTO in Linux differiscono da quanto stabilito negli RFC. Linux segue i modelli di base dati in RFC 2988 ([RFC2988]), ma l'implementazione differisce nella modifica della variabile `rttvar`. Una significativa differenza tra l'implementazione di Linux e quella di RFC 2988 risiede nell'uso in Linux di un valore minimo del RTO pari a 200ms anziché di 1000ms.

Capitolo 7 Esperimenti sul satellite

7.1 Introduzione

La parte conclusiva di questa tesi è consistita nella verifica del comportamento delle principali varianti del protocollo TCP su un canale satellitare reale e nel confronto con quanto ottenuto in precedenza mediante simulazione (vedi capitolo 5).

7.2 Caratteristiche del canale satellitare

Nell'ambito del progetto di ricerca SatNEx, è stato messo a disposizione presso l'ISTI del CNR di Pisa, a scopo di sperimentazione, un sistema di accesso al satellite denominato Skyplex ([Esa]). Skyplex è un multiplexer digitale: può ricevere dati in ingresso da più sorgenti, combinarli e ritrasmetterli come segnale combinato. Lo standard su cui si basa è lo standard europeo DVB (Digital Video Broadcasting), che da una decina di anni ha lo scopo di armonizzare le tecniche di trasmissione e ricezione dati delle telecomunicazioni.

Le unità Skyplex ([Sat04]) sono composte da canali configurabili secondo due modalità: Low Rate (2.112 Mbps) o High Rate (6.226 Mbps). Il metodo di accesso al canale è il TDMA (Time Division Multiple Access), il quale presenta una struttura a frame: ogni frame è composto da N slot di tempo, ognuno contenente M pacchetti, quindi N*M pacchetti per frame. In una simile struttura, N rappresenta il numero di stazioni trasmittenti. Ogni stazione ha a disposizione una banda di almeno 44 kbps. Ad intervalli regolari di 820 ms, viene effettuato nuovamente l'assegnamento dei frame ai terminali che vogliono trasmettere.

L'assegnamento della banda può essere dinamico (Bandwidth on Demand), statico o misto. Nel primo caso, la banda è richiesta periodicamente da ogni terminale sulla base del proprio bisogno istantaneo e gli slot di tempo sono assegnati in modo best-effort. Questa modalità di assegnamento non garantisce alcuna priorità o qualità del servizio tra i terminali. Nel caso di allocazione statica, un numero fissato di slot di tempo è preconfigurato e assegnato ad un certo terminale.

Nelle nostre sperimentazioni, ci è stato assegnato un canale di tipo Low Rate il cui ritardo è di circa 250 ms, con assegnamento dinamico della banda.

7.3 Scenario dei test

Gli esperimenti sono stati svolti in ambiente Linux. Sono state predisposte due macchine desktop, una presso l'istituto ISTI del CNR di Pisa con funzione di trasmettitore e l'altra presso il Laboratorio di Comunicazioni Multimediali del CNIT di Napoli con funzione di ricevitore, entrambe collegate direttamente ad un modem satellitare, come si può vedere dalla figura 7.1.

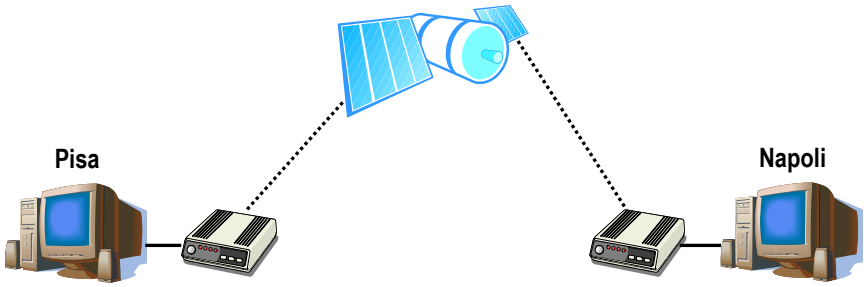

Figura 7.1. Scenario degli esperimenti sul satellite.

I valori sopra riportati della larghezza di banda e del ritardo, in realtà, sono nominali: prima di valutare il comportamento del protocollo TCP, occorre calcolare i valori effettivi per i due parametri.

Per il calcolo del ritardo, si è fatto ricorso al programma mtr, un tool di diagnostica della rete, già incluso nella maggior parte delle distribuzioni Linux. La sua particolarità sta nel fornire in output statistiche dettagliate sulla raggiungibilità di un host in internet. Mediante mtr, si è registrato un tempo medio di risposta di 840 ms e uno minimo di 620 ms.

La larghezza di banda effettiva del canale satellitare è stata misurata usando il programma hping2, inviando un numero via via crescente di datagram UDP al secondo e controllando quanti ne arrivano a destinazione. I tentativi fatti hanno mostrato che Skyplex è in grado di trasmettere circa 130 datagrammi al secondo, ciascuno di 1500 byte. Da ciò segue che la larghezza di banda effettiva è di circa 1600 kbit/s. Questo valore, circa il 20% in meno della banda dichiarata, è dovuto in parte all'overhead del DVB, ma soprattutto al metodo di accesso multiplo.

Un ultimo parametro importante ai fini degli esperimenti è la dimensione del buffer associato al bottleneck link, il link satellitare. Anche in questo caso, tale valore è stato determinato mediante il programma hping2. Il valore trovato è di circa 1500 Kbyte, eccessivo se si pensa che sia usato solo per consentire il regolare funzionamento del TCP (per il quale, in questa configurazione, basterebbero circa 200 Kbyte), ma ragionevole se il buffer è usato come parte dell'algoritmo di accesso multiplo al mezzo, per calcolare la richiesta di banda inviata dalla stazione all'allocatore.

7.4 Configurazione degli endpoint

Perché una connessione TCP possa sfruttare appieno la banda di un canale, la finestra di trasmissione deve essere pari almeno al prodotto banda*RTT. Con le impostazioni predefinite sulla maggior parte dei sistemi Linux, la dimensione della finestra consente velocità di trasmissione minori di 1 Mbit/s. Per ovviare a ciò bisogna definire l'uso dell'opzione TCP window scale sui due estremi della connessione e configurare le applicazioni perché utilizzino una finestra sufficientemente grande.

In Linux, i parametri riguardanti il comportamento della rete possono essere modificati mediante il comando `sysctl`, come già accennato nel paragrafo 6.2. In alternativa, è sufficiente usare lo pseudo file system `/proc`, sotto l'albero di directory `/proc/sys`. Ad esempio l'opzione window scale può essere attivata mediante `sysctl`:

```
sysctl –w net.ipv4.tcp_window_scaling=1
```

oppure usando il comando `echo`:

```
echo 1 > /proc/sys/net/ipv4/tcp_window_scaling
```

Nella maggior parte dei sistemi Linux, l'opzione sopra citata è attivata per default. Restano, quindi, da modificare solo i parametri relativi alle finestre di trasmissione e di ricezione qui riportati:

- `net.ipv4.tcp_mem`: dimensione totale dei buffer TCP (min, default, max);
- `net.ipv4.tcp_rmem`: memoria riservata per i buffer di ricezione del TCP (min, default, max);
- `net.ipv4.tcp_wmem`: memoria riservata per i buffer di trasmissione del TCP (min, default, max);
- `net.core.rmem_default`: dimensione standard della finestra di ricezione;
- `net.core.rmem_max`: dimensione massima della finestra di ricezione;
- `net.core.wmem_default`: dimensione standard della finestra di trasmissione;
- `net.core.wmem_max`: dimensione massima della finestra di trasmissione.

Negli esperimenti di questo capitolo, le variabili sono state impostate ai valori seguenti:

```
net.ipv4.tcp_rmem="8388608 8388608 8388608"
net.ipv4.tcp_wmem="8388608 8388608 8388608"
net.ipv4.tcp_mem="8388608 8388608 8388608"
net.core.rmem_default=8388608
net.core.wmem_default=8388608
```

```
net.core.rmem_max=8388608
net.core.wmem_max=8388608
```

7.5 Descrizione degli esperimenti

Il comportamento del protocollo TCP sul canale satellitare è stato analizzato mettendo a confronto due algoritmi di controllo della congestione: NewReno e Sack. Il primo algoritmo, in Linux, può essere attivato semplicemente disabilitando il secondo, ovvero ponendo a zero la variabile `net.ipv4.tcp_sack`.

7.5.1 Comportamento dell'algoritmo NewReno

Il grafico riportato in figura 7.2 mostra l'andamento dei segmenti e degli Ack in una trasmissione dati di durata pari a 300 secondi.

L'asse temporale è stato suddiviso in più parti per distinguere le fasi rilevanti dell'esperimento.

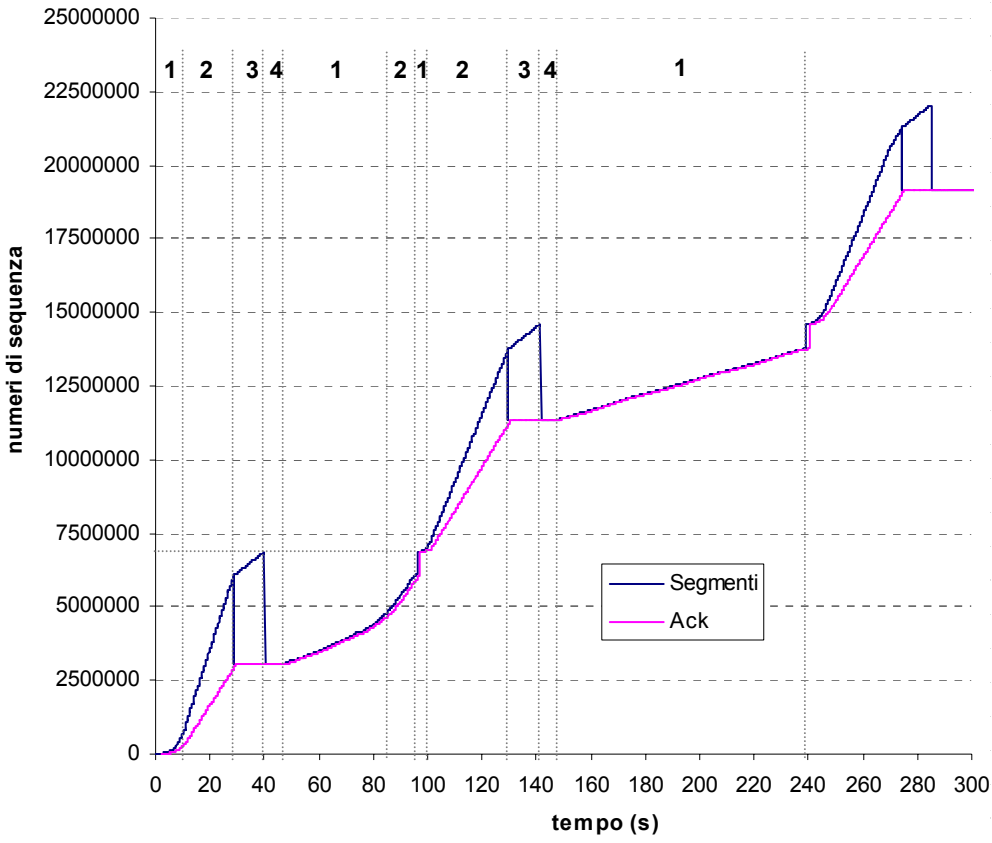

Figura 7.2. Comportamento dell'algoritmo NewReno.

Inizialmente l'algoritmo Slow Start regola l'aumento della finestra di congestione, incrementandola di un segmento per ogni Ack ricevuto che riscontra nuovi dati. Le velocità alle quali sono trasmessi segmenti e Ack qui coincidono. Queste

velocità progressivamente si discostano nella fase successiva: dal confronto delle pendenze della retta dei segmenti e di quella degli Ack si può affermare che i segmenti sono trasmessi a velocità doppia rispetto a quella degli Ack (2.4 Mbit/s contro 1.2 Mbit/s). Il divario tra le due velocità comporta un rapido riempimento del buffer e in breve la perdita di un elevato numero di segmenti che si manifesta nella seconda metà della fase 2.

Il primo segmento perso ha numero di sequenza 3029501. Quando il trasmettitore riceve tre Ack duplicati recanti quel numero di Ack, la finestra di congestione è di 2074 segmenti, valore calcolato a partire dalle tracce prodotte dal tcpdump, contando tutti gli Ack non duplicati ricevuti ed escludendo gli eventuali aggiornamenti della finestra di ricezione. Secondo la macchina a stati di Linux descritta nel paragrafo 6.3.1, il trasmettitore entra nello stato Recovery, ritrasmettendo il segmento perso e riducendo gradualmente la finestra di congestione fino a raggiungere la metà del valore iniziale (cioè 1037 segmenti). In questo stato, anche se la finestra di congestione non subisce alcun aumento, il trasmettitore invia ugualmente segmenti vecchi o nuovi rispettando il principio di conservazione del pacchetto. Ogni Ack duplicato ricevuto sta ad indicare che un segmento ha lasciato la rete; è possibile, quindi, trasmettere nuovi segmenti senza aggravare ulteriormente la situazione attuale di congestione della rete. In questo caso, sono trasmessi i segmenti con numero di sequenza compreso tra 6064841 e 6872221 (fase 3).

Se il trasmettitore avesse ricevuto un Ack parziale relativo al primo segmento ritrasmesso, la fase di recupero delle perdite si sarebbe protratta per diversi minuti mostrando il comportamento anomalo del TCP NewReno, ovvero la trasmissione di un solo segmento per RTT. A causa dell'eccessivo riempimento del buffer, invece, la ritrasmissione del primo segmento perso non va a buon fine. Non avendo ricevuto prima dello scadere del timer di ritrasmissione un Ack relativo a tale segmento, all'inizio della fase 4 (t = 40.12 s) si ha un timeout, con conseguente uscita dalla procedura di Fast Recovery. Il trasmettitore si sposta nello stato Loss: tutti i segmenti pendenti, in questo stato, sono marcati come persi, la finestra di congestione riparte da un segmento e l'algoritmo Slow Start si occupa dell'incremento di cwnd. La transizione allo stato Open può avvenire solo quando tutti i segmenti marcati come persi sono riscontrati con successo.

La curva degli Ack compresa tra gli istanti $t_1 = 47$ s e $t_2 = 85$ s, come indicato dal grafico, ha un andamento meno veloce del solito poiché, in seguito al timeout, sono inviati sia segmenti realmente mancanti che segmenti già arrivati a destinazione. Questi ultimi, abbastanza numerosi, causano l'invio di Ack duplicati da parte del ricevitore, i quali non concorrono all'aumento di cwnd.

Nell'intervallo di tempo compreso tra $t_1 = 85$ s e $t_2 = 96$ s, la curva degli Ack torna ad avere un andamento molto simile a quello registrato nella fase 2. Il salto del numero di Ack visibile all'istante t = 96.26 s indica il riempimento di tutti i buchi nei

numeri di sequenza provocati dalla perdita del primo burst di dati, come si vede dalla linea orizzontale nel grafico.

Le quattro fasi descritte si ripresentano nel corso di questa trasmissione dati. Anche in corrispondenza del secondo e del terzo gruppo di perdite, la ritrasmissione del primo segmento perso non va a buon fine, a causa del numero eccessivo di segmenti nel buffer. La perdita di questo segmento è indicata dal timeout, avutosi circa 11 secondi dopo la sua ritrasmissione. Il trasmettitore è di nuovo nello stato Loss, con cwnd ridotta ad un solo segmento. Rispetto al caso precedente, durante l'algoritmo Slow Start, l'incremento di cwnd è più ridotto, per via dell'elevato numero di Ack relativi a segmenti ricevuti correttamente durante la procedura di Fast Recovery.

L'adozione dell'algoritmo NewReno fa registrare un throughput piuttosto basso: il programma tcptrace riporta un valore medio di circa 0.9 Mbit/s (vedi grafico comparativo del throughput in figura 7.4).

7.5.2 Comportamento dell'algoritmo Sack

Prestazioni sicuramente più elevate si possono ottenere nel caso in cui venga utilizzato l'algoritmo Sack, come mostrato nel grafico riportato in figura 7.3.

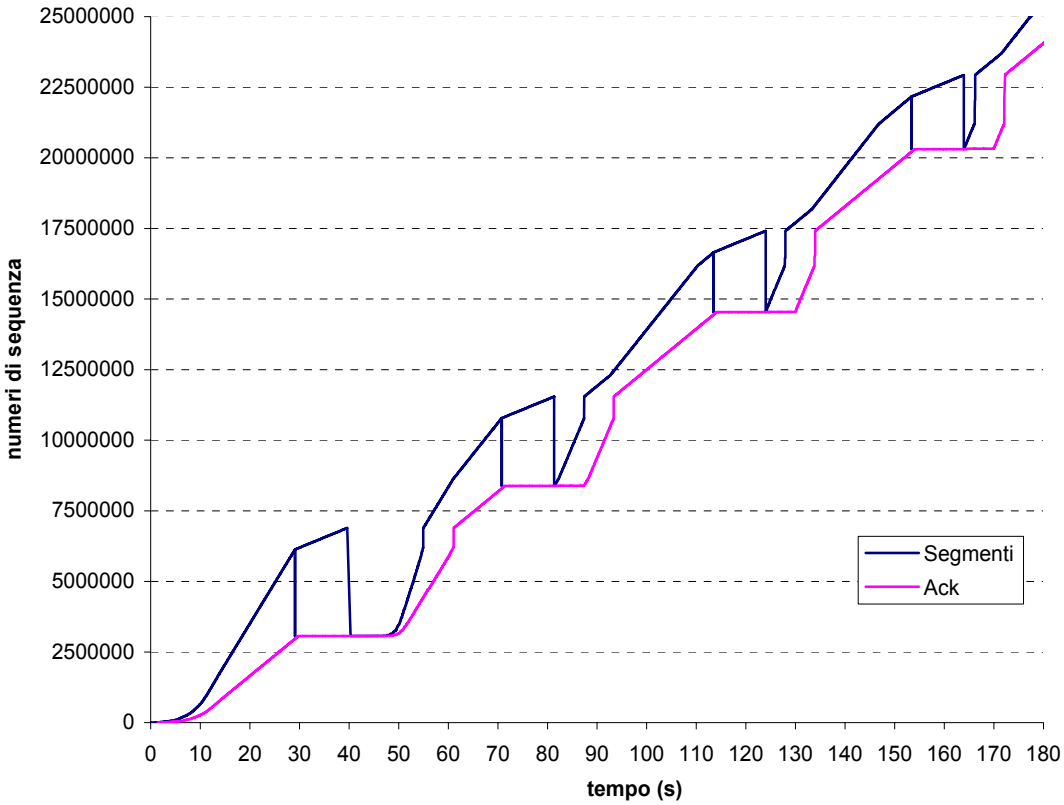

Figura 7.3. Comportamento dell'algoritmo Sack.

La novità più interessante dell'algoritmo Sack è l'introduzione della nozione di Ack selettivo, il quale offre al trasmettitore tutta una serie di informazioni dettagliate sui segmenti realmente persi. Questo impedisce delle inutili ritrasmissioni di segmenti già arrivati a destinazione.

Da una prima analisi, si può notare che, in circa 300 secondi di trasmissione, si verificano sei gruppi di perdite. Queste risultano abbastanza regolari nel tempo: ognuna avviene circa 40 secondi dopo la precedente.

La perdita del primo segmento si ha dopo circa 20 secondi. In quell'istante, il valore corrente della finestra di congestione è pari a circa 2100 segmenti. Come nel precedente paragrafo, la dimensione di cwnd è calcolata a partire dalle tracce del tcpdump. Il segmento perso ha numero di sequenza 3066001 ed è ritrasmesso non appena il trasmettitore riceve il primo Ack duplicato contenente un blocco Sack. Rispetto all'implementazione dell'algoritmo nel simulatore ns-2 e rispetto all'implementazione del NewReno in Linux, il trasmettitore non attende l'arrivo di tre Ack duplicati per dedurre che uno o più segmenti si sono persi, ma ritrasmette appena ha notizia della perdita dal blocco Sack. Questo sicuramente rende più breve la fase di recupero delle perdite.

Nella procedura di recupero, che va dall'istante t_1=29.1 s all'istante t_2=40.3 s, oltre a ritrasmettere il primo segmento perso, sono inviati i segmenti con numero di sequenza compreso tra 6134921 e 6895581. La ritrasmissione del primo segmento non va a buon fine per via dell'eccessiva congestione della rete. Soltanto in seguito ad un timeout si scopre che quel segmento non è arrivato a destinazione e lo si ritrasmette immediatamente, uscendo prematuramente dalla procedura di Fast Recovery.

La fase di Slow Start, successiva all'arrivo dell'Ack relativo al segmento ritrasmesso al timeout, non appare così lenta come è accaduto nel caso dell'algoritmo NewReno, perchè il trasmettitore, in virtù dei numerosi Ack duplicati ricevuti contenenti blocchi Sack, ritrasmette solo ed esclusivamente i segmenti che risultano essere mancanti al destinatario. Nel grafico si osserva che la curva degli Ack, limitatamente agli istanti successivi al timeout, ha l'andamento tipico della fase di Slow Start.

Il primo gruppo di perdite si differenzia dagli altri cinque perchè in questi ultimi non avviene la perdita del primo segmento ritrasmesso. Ad esempio, la porzione di grafico compresa tra t_1 = 72.45 s e t_2 = 82.14 s, relativa al recupero del secondo burst di dati, sembra apparentemente uguale a quella relativa al primo burst. Si tratta, invece, di una falsa similarità dovuta alla scala del grafico. Nessuno dei segmenti ritrasmessi va nuovamente perso per l'eccessivo riempimento del buffer, per cui la procedura di Fast Recovery si conclude con l'arrivo a destinazione di tutti i segmenti persi.

In conclusione, la tecnica degli Ack selettivi è vincente nelle prove qui effettuate. Questo è evidente sia nel numero di segmenti inviati (34321 segmenti contro i 15096 del NewReno), sia nel throughput istantaneo. Il valore medio del throghput istanta-

neo ottenuto con l'algoritmo Sack è di 1.3 Mbit/s, circa il 50% in più rispetto a quello calcolato per il NewReno. Il grafico in figura 7.4 mostra l'andamento del throughput per il NewReno e per il Sack.

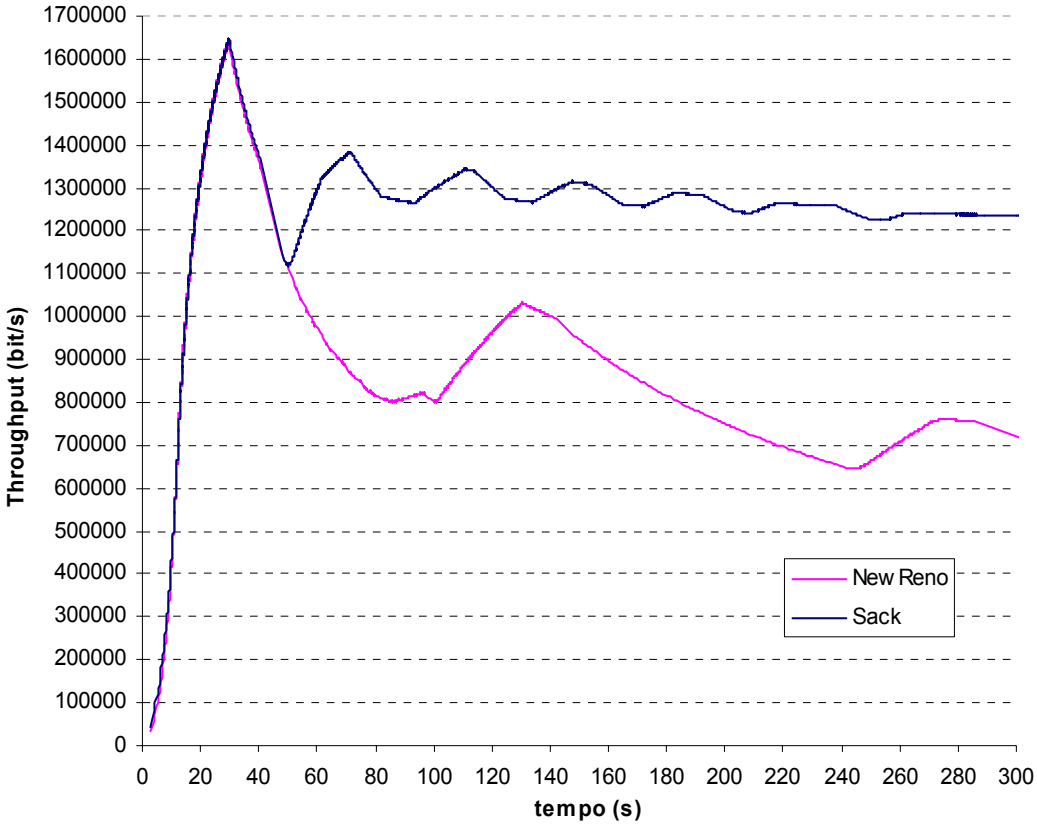

Figura 7.4. Grafico comparativo del throughput registrato per il NewReno e il Sack.

La stabilizzazione del throughput a valori significativamente inferiori alla banda disponibile (~ 1.6 Mbit/s) è probabilmente dovuta al funzionamento dell'algoritmo di allocazione dinamica di Skyplex, i cui dettagli sono purtroppo segreti.

Capitolo 8 Conclusioni

In questa tesi è stato analizzato il comportamento iniziale di una connessione TCP con diverse varianti del protocollo sia mediante analisi simulativa sia mediante verifica su un canale satellitare reale.

In assenza di errori, il TCP NewReno, durante la fase di avvio, ha un comportamento anomalo: la velocità di trasmissione è ridotta ad un solo segmento per RTT. La causa di ciò, come è stato accertato, è la variante del NewReno chiamata Slow but Steady.

Nel contesto delle modifiche al simulatore ns-2, si è introdotta, nelle classi che implementano un trasferimento dati bidirezionale, la variante Impatient dell'algoritmo NewReno. Tale variante si rivela efficace in tutti gli scenari in cui si verifica la perdita di un numero elevato di segmenti. In questo caso, la fase di recupero dalle perdite non si conclude prima dello scadere del timer di ritrasmissione. Il timeout, anche se riduce la finestra di congestione ad un segmento, consente un recupero molto veloce dei segmenti persi. La variante Slow but Steady, invece, impiega anche diversi minuti per il recupero, riducendo notevolmente il valore del throughput. Vi sono comunque degli scenari in cui è preferibile adottare quest'ultima variante, in quanto il timeout penalizzerebbe fortemente il trasmettitore. Entrambi i casi sono stati analizzati simulativamente con l'uso di ns-2.

Conclusioni analoghe possono essere tratte per quanto riguarda le modifiche relative all'aggiornamento di cwnd in seguito ad un timeout. Entrambi gli approcci (quello di ns-2 e quello di RFC 3782) sono corretti. Quello di ns-2 risulta più aggressivo nel numero di segmenti inviati, a differenza dell'altro che risulta più conservativo. Non si può concludere, quindi, che un approccio è in assoluto vincente rispetto all'altro. La scelta tra i due va fatta caso per caso, considerando i vantaggi e gli svantaggi di entrambi. Anche in questo caso è stata effettuata un'analisi simulativa.

L'analisi simulativa ha messo in luce alcuni aspetti interessanti: restringendoci al caso reale in cui il bottleneck link ha una larghezza di banda di 2 Mbit/s e ritardo di 250 ms, l'algoritmo Reno ottiene prestazioni più elevate della variante Slow but Steady. Il motivo è rintracciabile nella modalità di recupero dei segmenti e di riarmo del timer di ritrasmissione. L'algoritmo Sack risulta la soluzione vincente: il trasmet-

titore è a conoscenza di quali sono i segmenti realmente persi e si occupa soltanto della loro ritrasmissione. Questo porterebbe a concludere che il Sack è da preferirsi agli altri: in realtà non è così. Infatti, variando parametri come la larghezza di banda, il ritardo e il fattore beta, possono verificarsi dei timeout che penalizzano fortemente le prestazioni complessive dell'algoritmo.

Le buone prestazioni del Sack ottenute con l'analisi simulativa hanno avuto un riscontro negli esperimenti fatti sul canale satellitare reale: per una trasmissione di 300 secondi, la quantità di dati trasferita è doppia usando il Sack.

8.1 Sviluppi futuri

Il lavoro svolto in questa tesi non può dirsi né completo né esaustivo in tutte le sue parti. Una sezione che può essere sicuramente estesa ed approfondita riguarda gli esperimenti sul canale satellitare. Il comportamento dell'avvio del TCP andrebbe analizzato con altri stack (Mac OS, FreeBSD e Windows), in modo da avere un quadro più chiaro e completo del fenomeno qui discusso.

Un'altra direzione possibile di sviluppo potrebbe essere quella di studiare e analizzare il comportamento di altre varianti del protocollo TCP: ad esempio Westwood e le estensioni del Sack (Fack e D-Sack).

Appendice

cwnd_inflation.tcl

Impostando a 1 (0) il valore della variabile `cwnd_inflation_after_timeout_`, si adotta l'approccio di ns-2 (dettato da RFC 3782) per quanto riguarda l'aggiornamento della finestra di congestione in seguito ad un timeout.

```
set ns [new Simulator]; # initialise ns

set nf [open out.nam w]
$ns namtrace-all $nf
set node1 [$ns node]; set node2 [$ns node]

set src [new Agent/TCP/FullTcp/Newreno]
set sink [new Agent/TCP/FullTcp/Newreno]
$src set window_ 9999; $sink set window_ 9999
$src set segsize_ 472; $sink set segsize_ 472

$ns attach-agent $node1 $src; $ns attach-agent $node2 $sink
set ftp [new Application/FTP]
$ftp attach-agent $src
$ns connect $src $sink
$sink listen

$src set newreno_changes1_ 0; #to use Slow but Steady
$src set cwnd_inflation_after_timeout_ 0;

$ns duplex-link $node1 $node2 1e6 100ms DropTail
$ns queue-limit $node1 $node2 48

proc finish {} {
    global ns nf
    $ns flush-trace
    close $nf
    exit 0
}

proc add-error {LossyLink} {
    set em [new ErrorModel/List]
    $em droplist {63 126}
    $LossyLink errormodule $em
}

set LossyLink [$ns link $node1 $node2]
add-error $LossyLink

set startime   0
set stoptime   5
$ns at $startime "$ftp start"
$ns at $stoptime "$ftp stop"
$ns at $stoptime "$sink close"

$ns at 0 "record"
$ns at $stoptime "finish"
$ns run
```

NewReno_without_del_ack.tcl

Impostando a 1 (0) il valore della variabile `newreno_changes1_`, si sceglie la variante Impatient (Slow but Steady) del NewReno.

```
set ns [new Simulator]; # initialise ns

set nf [open out.nam w]
$ns namtrace-all $nf
set node1 [$ns node]; set node2 [$ns node]

set src [new Agent/TCP/FullTcp/Newreno]
set sink [new Agent/TCP/FullTcp/Newreno]
$src set window_ 9999; $sink set window_ 9999
$src set segsize_ 216; $sink set segsize_ 216

$ns attach-agent $node1 $src
set ftp [new Application/FTP]
$ftp attach-agent $src

#newreno_changes1_ is 0 to use Slow but Steady, 1 to use Impatient.
$src set newreno_changes1_ 0;
$src set cwnd_inflation_after_timeout_ 0;

$ns attach-agent $node2 $sink
$ns connect $src $sink
$sink listen

$ns duplex-link $node1 $node2 1e6 100ms DropTail
$ns queue-limit $node1 $node2 48;

proc finish {} {
    global ns nf
    $ns flush-trace
    close $nf
    exit 0
}

proc add-error {LossyLink} {
    set em [new ErrorModel/List]
    $em droplist {31 35 36 37 38 39 40 41 42 43 44 45 46 47 48}
    $LossyLink errormodule $em
}

set LossyLink [$ns link $node1 $node2]
add-error $LossyLink

set startime   0
set stoptime   4
set closetime  7
set finishtime 10

$ns at $startime "$ftp start"
$ns at $stoptime "$ftp stop"
$ns at $closetime "$sink close"

$ns at 0 "record"
$ns at $finishtime "finish"

$ns run
```

NewReno_del_ack.tcl

Impostando a 1 (0) il valore della variabile `newreno_changes1_`, si sceglie la variante Impatient (Slow but Steady) del NewReno.

```
set ns [new Simulator]; # initialise ns

set nf [open out.nam w]
$ns namtrace-all $nf
set node1 [$ns node]; set node2 [$ns node]

set src [new Agent/TCP/FullTcp/Newreno]
set sink [new Agent/TCP/FullTcp/Newreno]
$src set window_ 9999; $sink set window_ 9999
$src set segsize_ 216; $sink set segsize_ 216
$src set segsperack_ 2; #generate one ack every least 2 segments
$sink set segsperack_ 2;

$ns attach-agent $node1 $src
set ftp [new Application/FTP]
$ftp attach-agent $src
$ns attach-agent $node2 $sink
$ns connect $src $sink
$sink listen

#newreno_changes1_ is 0 to use Slow but Steady, 1 to use Impatient.
$src set newreno_changes1_ 0;
$src set cwnd_inflation_after_timeout 0;

$ns duplex-link $node1 $node2 1e6 100ms DropTail
$ns queue-limit $node1 $node2 48;

proc finish {} {
    global ns nf
    $ns flush-trace
    close $nf;
    exit 0
}

proc add-error {LossyLink} {
    set em [new ErrorModel/List]
    $em droplist {38 42 43 44 45 46 47 48 49 50 51 52 53 54
      55 56 57 58 59}
    $LossyLink errormodule $em
}

set LossyLink [$ns link $node1 $node2]
add-error $LossyLink

set startime   0
set stoptime   4
set closetime  7
set finishtime 10

$ns at $startime "$ftp start"
$ns at $stoptime "$ftp stop"
$ns at $closetime "$sink close"

$ns at 0 "record"
$ns at $finishtime "finish"
$ns run
```

satellite.tcl

Le variabili `duration`, `bandwidth`, `delay` e `variant` possono essere modificate dalla linea di comando nel modo seguente:

```
ns satellite.tcl "set duration 10" "set bandwidth 1Mb" \
"set delay 100ms" "set variant {}"
```

```
set duration 1000; # simulation length in seconds
set pktlen 1000; # IP datagram length (MTU [byte])
set bandwidth 8Mb; # bottleneck bandwidth [bit/s]
set delay 250ms; # one-way bottleneck delay
set beta 1; # buffer/(delay*bandwidth)
set variant /Newreno; # TCP variant

# bw_parse function is in ns-2.27/tcl/lib/ns-lib.tcl
set bw [bw_parse $bandwidth]
#delay_parse function is in ns-2.27/tcl/lib/ns-lib.tcl
set del [delay_parse $delay]
# minimum RTT (round trip latency)
set dtime [expr 2 * $del]
# buffer size in segments
set bufsize [expr $beta*$bw/(8*$pktlen)*$dtime]
# TCP segment size (payload [byte])
set segsize [expr $pktlen - 40]

set ns [new Simulator]; # initialise ns

set f0 [open trace.tr w]
$ns trace-all $f0

set nf [open out.nam w]
$ns namtrace-all $nf
set f_pktn [open pktn.tr w]
set rto_file [open rto.tr w]

set a1 [$ns node]; set n3 [$ns node]; set n4 [$ns node]

set src [new Agent/TCP/FullTcp$variant]
set sink [new Agent/TCP/FullTcp$variant]
$src set window_ 9999; $sink set window_ 9999
$src set segsize_ $segsize; $sink set segsize_ $segsize

#generate one ack every least 2 segments
$src set segsperack_ 2; $sink set segsperack_ 2

#$src set clear_on_timeout_ false
#$sink set clear_on_timeout_ false
$src set newreno_changes1_ 0
$src set cwnd_inflation_after_timeout_ 0

$ns attach-agent $a1 $src
set ftp [new Application/FTP]
$ftp attach-agent $src

$ns attach-agent $n4 $sink
$ns connect $src $sink
$sink listen

$ns duplex-link $a1 $n3 100e6 1ms DropTail
$ns queue-limit $a1 $n3 $bufsize
$ns duplex-link $n3 $n4 $bw $del DropTail
$ns queue-limit $n3 $n4 $bufsize
```

```
$ns duplex-link-op $n3 $n4 queuePos 0.5

set qmon_a1n3 [$ns monitor-queue $a1 $n3 /dev/null]
set qmon34 [$ns monitor-queue $n3 $n4 /dev/null]

proc print-data {} {
    global stations ns qmon34
    global qmon_a1n3 src

    puts "Results obtained using qmon "
    puts "Packets arrived-a1-n3  =
      [$qmon_a1n3 set   parrivals_]\n \
    departed = [$qmon_a1n3 set pdepartures_]"
    puts ""
    puts "Packets arrived-3-4 (bottleneck link) = [$qmon34 set
      parrivals_]\n \
    lost = [$qmon34 set pdrops_] \n \
    departed = [$qmon34 set pdepartures_]"
    puts "Approximate Number of cycles for source a1 =
      [$src set ncwndcuts_]\n \
    Number of retransmission timeouts for source a1 =
      [$src set nrexmit_]"
}

set dep34 0
puts $f_pktn {$ DATA=COLUMN}
puts $f_pktn "% sidelabel=False subtitle=\"bandwidth=$bandwidth,
  delay=$delay, variant=$variant\""
puts $f_pktn {"time" "cumulative packets" "utilization" "cwnd"
  "ssthreshold"}

proc record {} {
    global ns stations dtime bw pktlen
    global src
    global f_pktn qmon34 dep34 rto_file
    set now [$ns now]
    set newdep34 "[$qmon34 set pdepartures_]"
    #Values needed by RTO calculation
    set rttvar [$src set rttvar_]
    set rttvar_exp [$src set rttvar_exp_]
    set SRTT_BITS [$src set T_SRTT_BITS]
    set RTTVAR_BITS [$src set T_RTTVAR_BITS]
    set srtt [$src set srtt_]
    set tcp_tick [$src set tcpTick_]
    set rto  [expr (((($rttvar << ($rttvar_exp + ($SRTT_BITS -
      $RTTVAR_BITS))) \
      + $srtt) >> $SRTT_BITS) * $tcp_tick];#Retransmit Timeout

    puts $f_pktn [format "%7.3f %7d %.4f %7.2f %10.1f" \
                      $now \
                      $newdep34 \
                      [expr ( $newdep34 - $dep34 ) / $dtime /
                        ($bw/(8*$pktlen))] \
                      [$src set cwnd_] \
                      [$src set ssthresh_] \
                  ]
    puts $rto_file [format "%7.3f %6.3f %6.3f %d" \
                  $now \
                  $rto \
                  [expr $tcp_tick * [$src set rtt_]] \
                  [$src set ack_] \
                   ]
    set dep34 $newdep34
```

```
    $ns at [expr $now + $dtime] "record"
}

proc finish {} {
    global stations command ns
    global nf f0
    global f_pktn rto_file
    $ns flush-trace
    close $nf; close $rto_file; close $f0
    close $f_pktn
    print-data
    exit 0
}

set startime   0
set stoptime   $duration
set closetime  [expr $duration + 10]
set finishtime [expr $duration + 20]

$ns at $startime "$ftp start"
$ns at $stoptime "$ftp stop"
$ns at $closetime "$sink close"

$ns at 0 "record"
$ns at $finishtime "finish"

$ns run
```

Riferimenti bibliografici

[CP02] Cussotto E., Pera S., "Protocolli a livello trasporto in sistemi radio di terza generazione", Luglio 2002. http://www.tlc-networks.polito.it/UPS3/tesi_cussotto_pera/TesiCussottoPera.ps

[Eck00] Eckel B., "Thinking in C++, Second edition, Volume 1: Introduction to standard C++", Prentice Hall, Gennaio 2000. http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html

[Esa] Sito ufficiale Agenzia Spaziale Europea, Notizie Locali, http://www.esa.int/esaCP/SEMKI967ESD_Italy_0.html

[FF96] Fall K. and Floyd S., "Simulation-based Comparisons of Tahoe, Reno and SACK TCP", Computer Communication Review, Luglio 1996. ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z

[Jac88] Jacobson V., "Congestion Avoidance and Control", Computer Communication Review, vol. 18, no. 4, pp. 314-329, Agosto 1988. ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z

[KR01] Kurose James F., Ross Keith W., "Computer Networking: A Top-Down Approach Featuring the Internet", Addison-Wesley, Ottobre 2001.

[Ns05] The VINT Project, "The ns Manual (formerly ns Notes and Documentation)", Marzo 2005.

[Ns2] Sito ufficiale Network Simulator, http://www.isi.edu/nsnam/ns/

[Pet01] Petrizzelli S., "Simulazioni di algoritmi di compressione dell'header per sistemi wireless basati su piattaforma TCP/IP", Febbraio 2001. http://users.libero.it/sandry/UMTS/tesi.htm

[RFC793] Postel J., "Transmission Control Protocol", STD 7, RFC 793, Settembre 1981.

[RFC1122] Internet Engineering Task Force, R. Braden editor, "Requirements for Internet Hosts - Communication Layers", RFC 1122, Ottobre 1989.

[RFC1323] Jacobson V., Braden R., Borman D., "TCP Extensions for High Performance", RFC 1323, Maggio 1992.

[RFC2018] Mathis M., Mahdavi J., Floyd S. and Romanow A., "TCP Selective Acknowledgment Options", RFC 2018, Ottobre 1996.

[RFC2581] Stevens W., Allman M. and Paxson V., "TCP Congestion Control", RFC 2581, Aprile 1999.

[RFC2988] V. Paxson and M. Allman. "Computing TCP's Retransmission Timer". RFC 2988, November 2000.

[RFC3782] Floyd S., Henderson T. and Gurtov A., “The NewReno Modification to TCP’s Fast Recovery Algorithm”, RFC 3782, Aprile 2004.

[Rub97] Rubini A., “Chiamate di sistema sysctl()”, Ottobre 1997. http://www.pluto.it/journal/pj9710/index.html

[Sat04] Feltrin E., Weller E., Martin E., Zamani K., “Design, Implementation and Performances Analysis of an On Board Processor – Based Satellite Network”, 2004. http://ieeexplore.ieee.org/Xplore/guesthome.jsp

[SatNEx] Sito ufficiale del progetto SatNEx, http://www.satnex.org

[SK02] Sarolahti P., Kuznetsov A., “Congestion control in Linux TCP”, Giugno 2002. http://www.cs.helsinki.fi/research/iwtcp/papers/linuxtcp.pdf

[Ste94] Stevens W., “TCP/IP Illustrated, Volume 1: The Protocols”, Addison-Wesley, 1994.