UNIVERSITA' DEGLI STUDI DI PAVIA
FACOLTA' DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica
Sede di Mantova

Simulazione del comportamento prestazionale del protocollo TCP e di alcuni protocolli di Routing

Relatore:
Prof. GIUSEPPE FEDERICO ROSSI

Tesi di laurea di:
ALBERTO SAVIOLI
Matricola: 293117/22

Anno Accademico 2004/2005

Indice

Introduzione

I tempi moderni sono caratterizzati dall'elevato grado di ricerca che viene effettuata per poter migliorare le tecnologie: basti pensare ad esempio come dall'osservazione dei pianeti attraverso telescopi si sia passati all'analisi reale della loro superficie tramite automi. Per arrivare a questi risultati i progettisti hanno dovuto trovare metodi che fossero in grado di prevedere il comportamento di un generico modello, che il più delle volte ha complessità troppo elevate per essere analizzato manualmente, oppure si evolve in tempi che non possono essere riprodotti in laboratorio per le misure sperimentali (l'evoluzione degli strati di roccia sedimentaria richiede tempi lunghissimi, ovvero l'andamento del Big Bang è stato dapprima rapidissima e poi via via più lenta). Il metodo che si è affermato per la maggiore è quello che prevede l'utilizzo di simulatori software: essi sono in grado di rendere gestibili tempi lunghissimi o brevissimi, e di poter analizzare modelli di grande complessità tramite i calcolatori con precisioni elevate e in tempi brevi.

In questo elaborato verrà analizzato il simulatore di reti telematiche Network Simulator 2, tramite il quale saranno simulati i comportamenti di alcune versioni del protocollo di trasporto TCP, e quelli di alcuni protocolli di routing utilizzati oggi in Internet. In particolare verrà analizzato il comportamento dei diversi algoritmi alla base dei protocolli in presenza di errori sulla rete, quali perdite di dati o caduta di un collegamento, e di ciascuno ne verranno quindi messi in luce i pregi ed i limiti.

Capitolo 1

Simulazione ad eventi discreti

1.1 Definizione di simulazione

Una simulazione è la tecnica di imitare il comportamento di alcune situazioni o sistemi (fisici, meccanici, ecc.) tramite un modello, per studiarne le caratteristiche dinamiche o statiche senza però dover passare alla realizzazione, cosa che può essere dispendiosa e difficile da ottenere. La simulazione implica l'elaborazione di un modello che dovrà rispecchiare il più possibile la realtà, in modo da garantire l'attendibilità dei risultati: proprio da questo modello dipende la bontà dei dati ottenuti in uscita dalla simulazione. Purtroppo la maggioranza dei sistemi reali sono caratterizzati da cambiamenti senza soluzione di continuità: preso un intervallo di tempo piccolo a piacere, il sistema evolverà in quell'intervallo assumendo infiniti valori. Sarà quindi impossibile avere un modello che rispecchi completamente la situazione reale, ma si avranno modelli che approssimano la realtà in maniera più o meno precisa, a seconda delle tecniche usate. Nasce così la simulazione ad eventi discreti, che verrà trattata nel corso dell'elaborato. Tipicamente una simulazione viene eseguita tramite un calcolatore, che permette di elaborare un'enorme mole di dati in tempi relativamente brevi, con la possibilità di ottenere in uscita dati immediatamente utilizzabili dagli interessati.

1.2 Sistemi e Modelli in simulazioni ad eventi discreti

Come accennato in precedenza, una simulazione si prefigge di imitare il comportamento di un sistema nel tempo attraverso modelli.
Un sistema è costituito da un'insieme di entità descritte da un certo numero di attributi. Le entità sono quegli elementi che interagiscono tra loro o con il mondo esterno al sistema; ogni entità è caratterizzata da uno o più attributi che ne definiscono le proprietà.
Un modello è la rappresentazione formale di un sistema che mette in relazione attraverso funzioni matematiche o logiche i parametri che caratterizzano il sistema stesso e le sue relazioni con la realtà esterna. Dette funzioni descrivono le regole di funzionamento del sistema da simulare. Lo scopo è capire la realtà nella sua complessità, attraverso l'osservazione del comportamento dinamico di una sua rappresentazione, seppur semplificata: il modello.
Non esiste solo un solo modello per un sistema, ma ne esistono molti, tutti in grado si rispecchiare più o meno esatta il sistema: individuare il modello più adatto è difficile e ci si basa sull'esperienza.

Infatti di norma si raggiunge il modello che meglio rappresenta il sistema solamente dopo averne simulati e modificati diversi.
Una caratteristica importante del sistema è lo stato: esso è definito attraverso un insieme di variabili che contengono le informazioni necessarie a descrivere il sistema in ogni istante di osservazione.
Lo stato di un sistema può variare in modo continuo nel tempo, oppure in modo discreto:

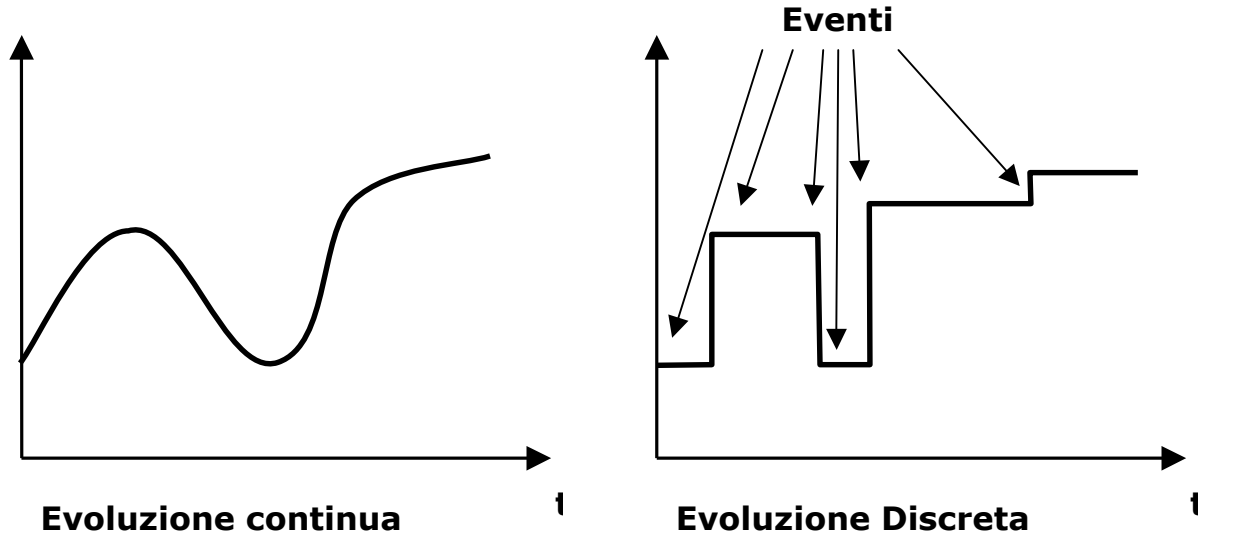
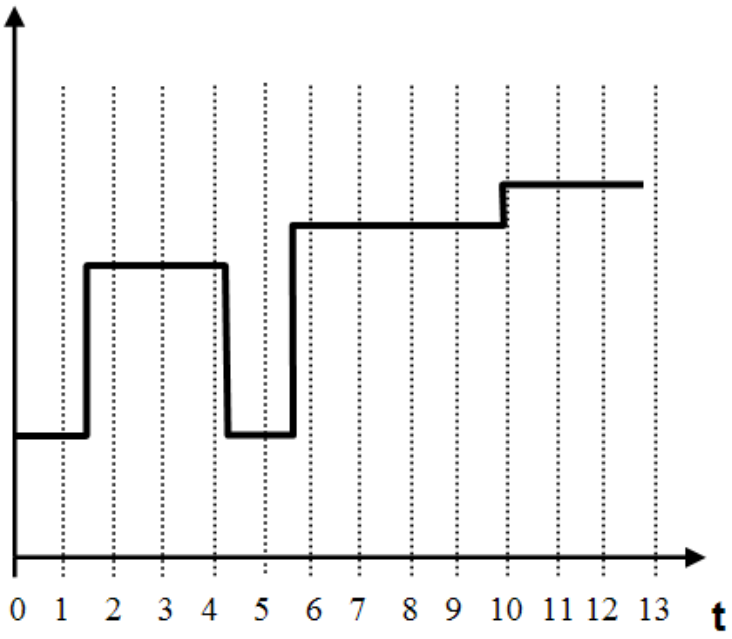


Fig.1

Analizziamo ora l'evoluzione discreta. In questi sistemi il cambiamento dello stato avviene solo in punti ben precisi (lo "scalino") cioè in corrispondenza di eventi: qui si hanno variazioni di stato istantanee da un valore ad un altro. Ogni evento è caratterizzato da un momento di inizio e uno di fine attività. In questo modo abbiamo fatto una semplificazione che sarà tanto più vicina alla realtà quanto maggiore sarà la frequenza degli eventi. Al limite si avrà che l'evoluzione discreta coinciderà con quella continua. Sorge però il problema del costo computazionale della simulazione: aumentando la densità degli eventi aumenterà anche la mole di calcolo necessaria per portarla a termine. Si dovrà allora trovare un compromesso tra precisione e tempo di calcolo, tenendo conto che il tempo di calcolo potrebbe essere molto oneroso, dato che per simulazioni di una certa complessità non basta il semplice pc, ma bisogna ricorrere ai supercomputer dei centri di calcolo.
Molto importante è la scelta di come gestire l'avanzamento temporale della simulazione, infatti una delle funzioni principali di un simulatore è il modo in cui esso fa avanzare il modello nel futuro. Esistono due principali tecniche di avanzamento: una si basa sul *Time Slicing*, mentre l'altra sul *Next Event*.
Nel Time Slicing il tempo è quantizzato a priori ed il modello viene fatto avanzare avanti nel tempo ad intervalli fissi (ad esempio ogni 2 secondi), senza considerare nulla nell'evoluzione del modello.



Time Slicing

Fig.2

Questo metodo però ha molti svantaggi, nel nostro caso:

- È raro che l'inizio di un intervallo di tempo sia contemporaneo ad un evento, quindi si potrebbero verificare dei ritardi nel cambiamento dello stato del modello che potrebbero non essere accettati;
- Anche se non ci sono eventi per molto tempo, ad ogni Time Slice il simulatore dovrà comunque ricalcolare un valore che non è cambiato, sprecando tempo;
- Il Time Slice deve essere scelto in modo corretto poiché se troppo lungo potrebbe portare a perdere eventi distanti meno del valore del Time Slice, viceversa se troppo breve può portare ad un inutile sovraccarico computazionale in quanto il più delle volte ricalcolerà valori che non cambiano.

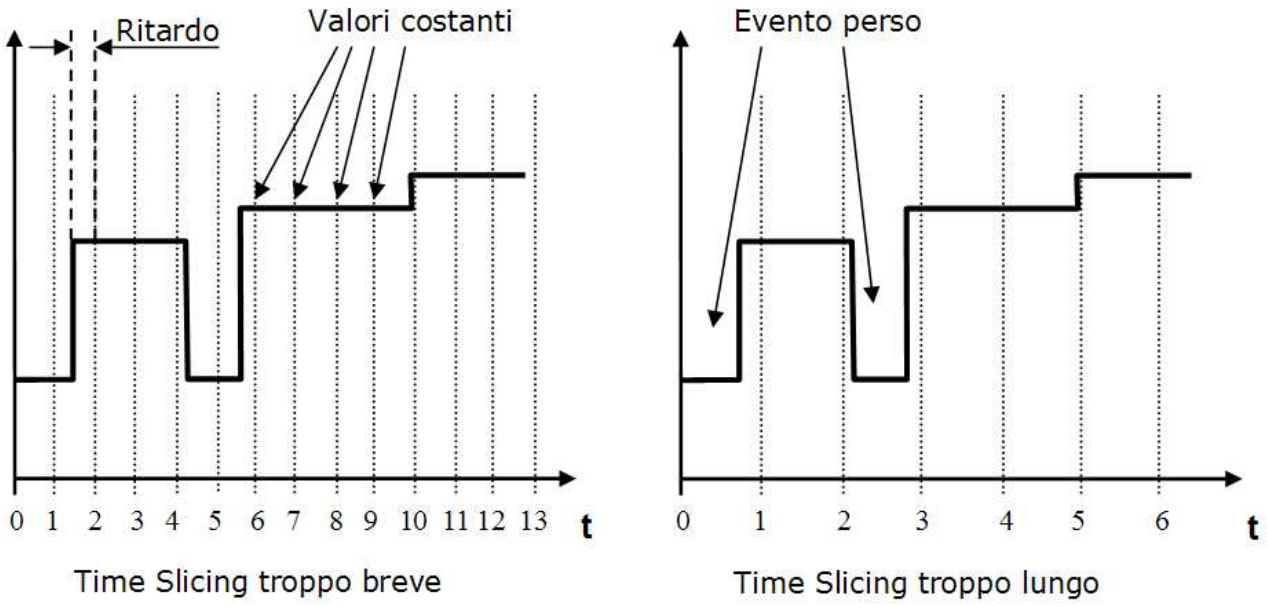


Fig.3

D'altra parte questo metodo si rivela molto utile in caso di analisi di sistemi continui, dove è raro che gli stati rimangano costanti.

In questo caso per avere una precisione maggiore si utilizzerà un Time Slice breve, al contrario più lungo per una minore precisione.

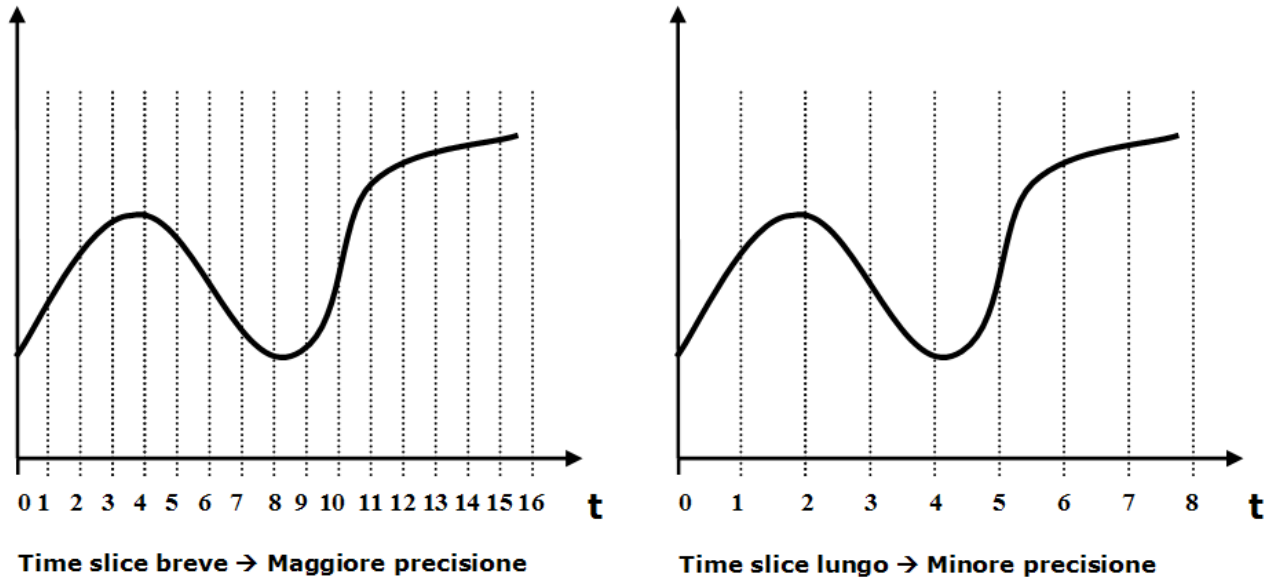


Fig.4

Per la simulazione ad eventi discreti ben si presta la seconda tecnica: il Next Event. Essa si basa sul far avanzare avanti il modello fino al tempo dell'evento futuro, evitando così i problemi visti per il Time Slice:

- Non esistono ritardi nel riconoscimento di un evento: il tempo della simulazione avanzerà esattamente fino ad un evento, lo eseguirà ed una volta terminato avanzerà all'evento successivo;
- Se non ci sono eventi per un lungo periodo di tempo il simulatore andrà direttamente all'evento successivo senza aspettare altro tempo (ad esempio se due eventi sono separati di tre minuti, il simulatore farà un passo nel tempo di tre minuti) rendendo la simulazione molto più efficiente.

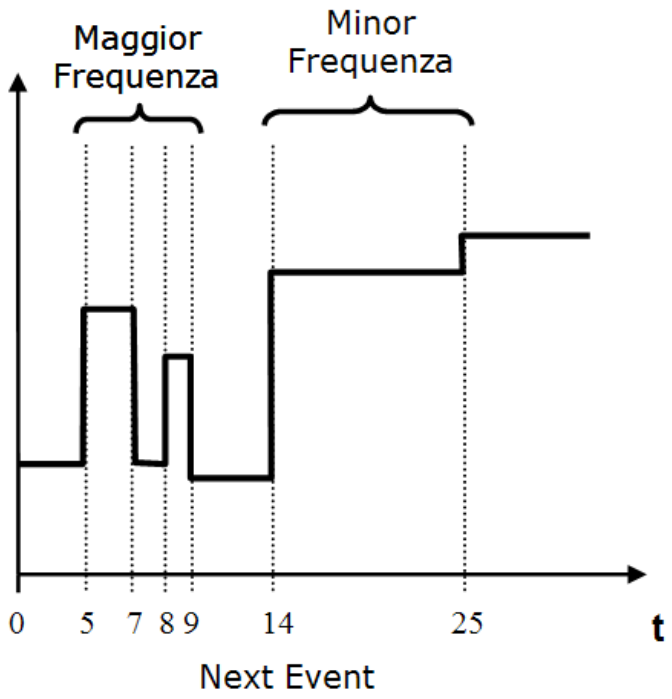


Fig.5

1.3 Modelli deterministici e stocastici

Abbiamo visto in precedenza come un modello sia la rappresentazione di un sistema reale, seppur con alcune semplificazioni dovute in buona parte a causa delle leggi governanti il sistema che a volte sono ben difficili (se non impossibili) da implementare. Nell'universo delle simulazioni esistono almeno due grosse categorie di modelli: deterministici e stocastici.

I modelli deterministici sono i più semplici da descrivere, dato che la loro evoluzione a partire da uno stato iniziale è completamente determinata da leggi matematiche o fisiche che non cambiano. Un esempio è il tempo di percorrenza di un tratto di strada di un'auto: basta conoscere la velocità

iniziale del veicolo, la lunghezza del tratto da percorrere, l'andamento dell'accelerazione e infine la legge che lega le varie grandezze.

Un modello stocastico, invece, è un modello caratterizzato da leggi casuali, cioè non determinabili a priori, che portano a risultati diversi partendo dal medesimo stato iniziale: ad esempio il tempo d'attesa in una fila ad un ufficio postale potrebbe essere una variabile con una data distribuzione nel tempo (es. uniforme tra 0 a 10 min). Anche se si parte in una data posizione nella fila non è detto che si venga serviti sempre nello stesso tempo. L'analisi del modello, in questi casi, viene fatta raccogliendo un campione delle possibili evoluzioni ed applicando tecniche statistiche che porteranno ad una stima delle variabili di interesse.

Un modello viene considerato stocastico o quando al suo interno vi sono componenti realmente stocastiche, oppure quando le leggi che governano il sistema a cui si riferisce sono troppo complesse da essere descritte in modo deterministico con formule matematiche. In questo ultimo caso diviene conveniente una rappresentazione stocastica. Un problema della rappresentazione stocastica è che occorre raccogliere un campione significativo tra tutte le possibili evoluzioni per poter effettuare un'analisi statistica significativa. Infatti un modello stocastico può assumere molti valori all'interno del proprio spazio e bisogna saper interpretare quali tra tutti i valori sono i più probabili e scartare quelli meno probabili che inficerebbero la bontà del modello.

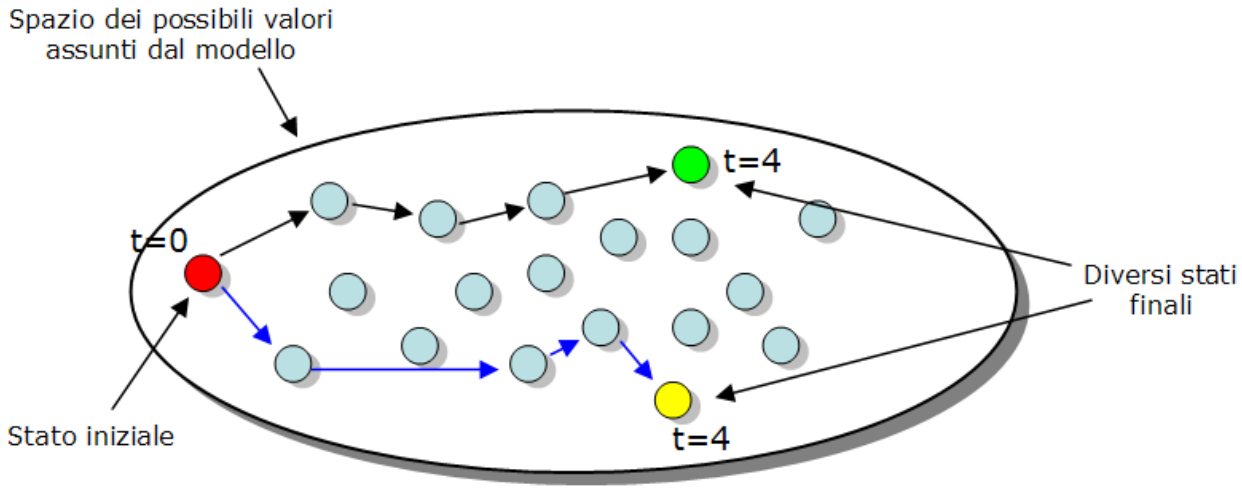


Fig.6: Si nota come da uno stato iniziale si può arrivare a diversi stati finali passando per cammini diversi

In un simulatore solitamente sono implementate delle procedure che sono in grado di generare numeri casuali con diverse distribuzioni (uniforme, gaussiane, poissoniana, ecc.) da poter utilizzare per simulare molte volte lo stesso scenario essendo certi di dare in input al modello valori con la distribuzione desiderata. Questo rende molto più facile la creazione del modello stesso, poiché basta

ricavare l'andamento della distribuzione e far generare al simulatore i valori con l'andamento voluto.

1.4 Raccolta ed analisi dei dati elaborati.

Lo scopo di qualsiasi simulazione è fornire informazioni utili per capire il comportamento del modello analizzato in modo da poter prendere decisioni sicure su come procedere. Una volta terminata, la simulazione può mettere a disposizione dell'utente dati in forme diverse: tabelle, grafici, animazioni e così via. Sta nell'utente scegliere la forma di output che ritiene più utile al fine di studiare il risultato ottenuto nella via più evidente e rapida. Ad esempio se si è simulato il comportamento del controllo di un forno ed in uscita della simulazione abbiamo la temperatura letta ogni secondo, con una simulazione di un'ora, è evidente che sarà più comodo graficare i valori piuttosto di avere una tabella lunghissima. D'altra parte se invece si ha in uscita un solo valore sarà inutile graficarlo.
Dall'analisi di questi dati verrà stabilito se la simulazione ha fornito dati sufficientemente precisi o tali da poterla ritenere conclusa con successo. In caso contrario si dovrà continuare a rielaborare il modello modificando i valori degli attributi che lo compongono, al fine di eseguire altre simulazioni che porteranno a risultati più o meno soddisfacenti, fino a che l'output sarà preso come valido.

1.*5 Vantaggi e svantaggi della simulazione*

Le simulazioni sono uno strumento alquanto utile in praticamente tutti i campi immaginabili: trasporti (simulatori di volo, sviluppo elettronica di bordo delle automobili, ecc.), economia (studio di modelli applicabili su investimenti), sistemi pubblici (ospedali, servizi pubblici), militari, sistemi di telecomunicazione (prestazioni di un protocollo o di un instradamento) e in generale in tutti i casi dove sorgono problemi *What-if*, ovvero dove ci si pone il quesito di cosa (what) succede se (if) si agisce su una determinata variabile senza interrompere il regolare funzionamento di un sistema. Esse sono indispensabili per la verifica dell'efficienza di un nuovo sistema ancora prima di costruirlo, risparmiando in molti casi tempo e denaro. Inoltre dato che la costruzione del modello implica lo studio molto approfondito del sistema può essere che proprio in questa fase si apportino delle significative migliorie al sistema stesso, eliminandone dei punti deboli. D'altra parte però la creazione del modello richiede elevata competenza nelle tecniche di simulazione: una modellizzazione di un sistema e la sua validazione potrebbero necessitare di moltissimo tempo e

fondi per essere portate a termine. Inoltre i risultati della simulazione potrebbero essere molto complessi da analizzare, andando ulteriormente a gravare sul budget del progetto.

Capitolo 2

Il simulatore ad eventi discreti Network Simulator 2 (NS2)

2.1 Caratteristiche del simulatore

Network Simulator 2 è un simulatore di reti ad eventi discreti orientato al livello Network che si basa totalmente sul linguaggio di programmazione C++ / Otcl. Sviluppato dall'Università di Berkley, USC/ISI e da Xerox PARC, il simulatore è distribuito gratuitamente per piattaforme Linux/Unix ma è compilabile anche sotto Windows. I sorgenti da compilare si possono scaricare dal sito `http://www.isi.edu/nsnam/ns`; in particolare il file NS-Allinone, di circa 60 Mb., contiene tutto il codice necessario ad installare e verificare il corretto funzionamento di NS2.
NS2 è un simulatore i cui codici sorgente possono essere modificati a piacere: questo fa si che esso risulti essere il più flessibile possibile, permettendo sia di modificare i protocolli e le loro caratteristiche in base alle necessità dell'utilizzatore, sia di creare nuovi protocolli per ricerca o per particolari scenari di rete (es: una reti *ad Hoc*). Questa libertà di modifica ha permesso al simulatore di essere migliorato col tempo, raggiungendo oggi una versione stabile, semplice da utilizzare e con possibilità quasi infinita di intervento sui parametri caratteristici della rete da esaminare. La fase di analisi post-simulazione può essere inoltre coadiuvata da due applicazioni grafiche: Xgraph, che permette di disegnare grafici dei dati raccolti, e Network Animator (Nam), che consente di visualizzare a video la topologia di rete creata e una animazione del traffico dei pacchetti attraverso essa.

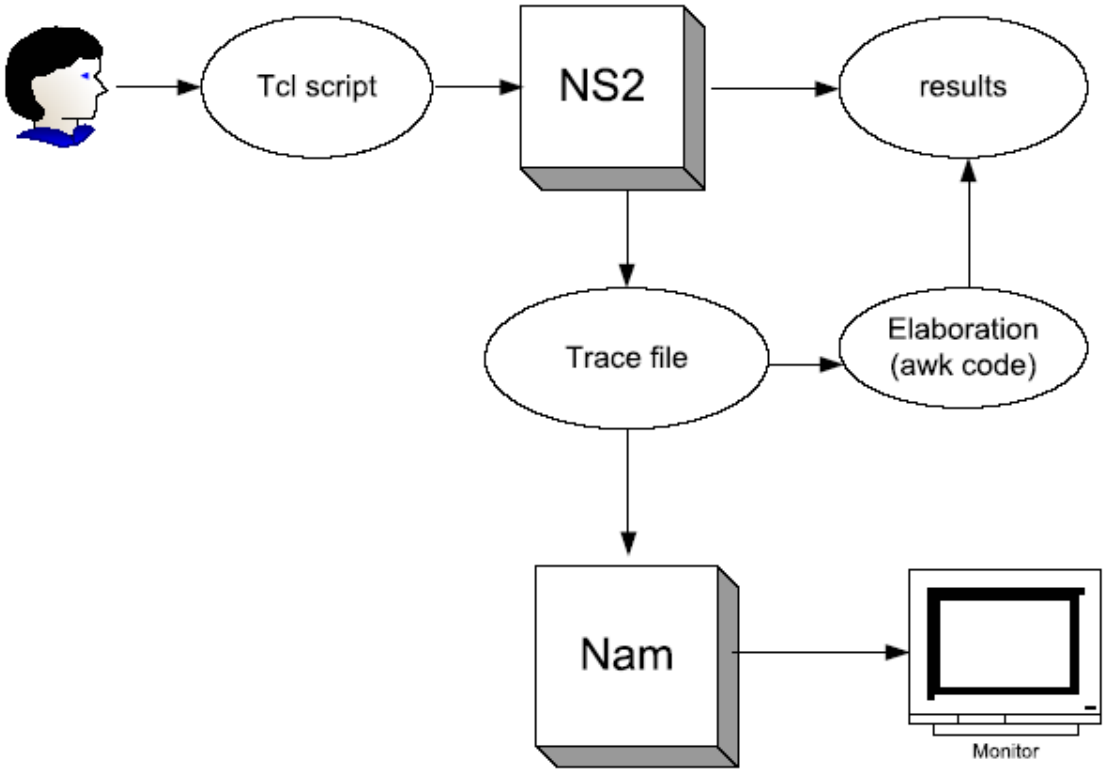


Fig.7: Principio di funzionamento di NS2

2.2 Accenni sul funzionamento del simulatore: C++ ed Otcl

Come già menzionato, NS2 è basato su due diversi linguaggi: il diffusissimo C++ ed Otcl, un linguaggio di scripting estremamente versatile. E' stato utilizzato questo approccio poiché il simulatore deve far fronte fondamentalmente a due richieste principali:

- capacità di elaborare velocemente una gran quantità di dati (bytes, headers dei pacchetti in dettaglio, stato dei componenti del sistema…) e modellare un gran numero di componenti.
- possibilità di modificare agevolmente i parametri di configurazione delle reti simulate, al fine di variare lo scenario e ottimizzare i parametri la rete stessa

Nel primo caso il linguaggio C++ è la scelta ottimale, dato che è orientato agli oggetti, compilato e rapido da eseguire. Non a caso i modelli dei protocolli, all'interno quali vi è la presenza di molte iterazioni e calcoli, sono implementati con questi linguaggio.

Poiché tuttavia C++ è difficile e brigoso da modificare, nel secondo caso invece si tende ad utilizzare un linguaggio interpretato come Object-tcl, dove si trascura l'alto tempo di esecuzione alto e si predilige la rapidità con cui è possibile apportare modifiche al codice.

Le due componenti sono fortemente collegate tra loro poiché l'utente può creare rapidamente nuovi oggetti tramite l'interprete Otcl; all'interno di questi verranno create automaticamente delle

corrispondenze con C++, che penserà alla parte "computazionale" del protocollo. Questa corrispondenza tra linguaggio interpretato e compilato è regolata dalla classe Tcl Object, come mostrato in figura 8.

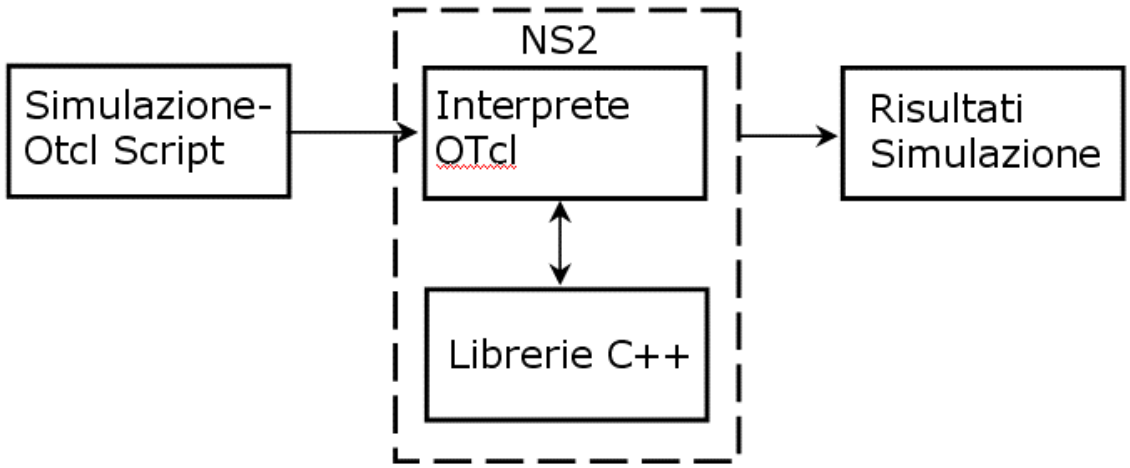


Fig.8 Struttura di Ns: relazione tra librerie C++ e Interprete Otcl

Utilizzando questa combinazione di linguaggi il codice da scrivere per ottenere determinati elementi o azioni si riduce ad alcune righe, divenendo così molto semplice da padroneggiare. Nei prossimi capitoli analizzeremo in dettaglio i componenti costituivi principali di una simulazione e la loro sintassi nel linguaggio tcl.

2.3 Componenti di una simulazione

Gli elementi costitutivi di una simulazione NS2 sono classi di Otcl: quando si dichiara un componente, la classe TCL Object creerà il suo corrispondente Otcl interpretato e successivamente lo tradurrà in linguaggio compilato utilizzabile da C++ per le elaborazioni del caso.
I principali componenti utilizzati per modellazione di una rete di calcolatori sono i nodi, i link, gli agenti e le sorgenti di traffico, le code, i modelli di errore nonché tutti gli eventi in generale associabili ad una simulazione.

I Nodi

Quando si crea un nodo viene costruita da Otcl un'istanza della classe *node* che gestirà il comportamento del nodo stesso tenendo conto delle caratteristiche impostate dall'utente.

Un nodo del simulatore corrisponde agli hosts o router della rete reale. Essi perciò hanno la funzione di inviare o ricevere pacchetti nel caso di host, oppure di inoltrare i pacchetti ricevuti verso altri nodi, nel caso di router. Ad ogni nodo generato nella simulazione viene assegnato un indirizzo sequenziale univoco di 16 bit: 8 bit sono riservati alla numerazione progressiva dei nodi, mentre gli altri 8 bit sono utilizzati per specificare il tipo di *agent* che agisce sul nodo. In questo modo si possono creare al massimo $2^8 = 256$ nodi. Tuttavia se si necessita di topologie più estese, per quanto sia poi fattibile un'analisi, esiste la possibilità di estendere a 32 i bit dell'indirizzo. In questo caso rimarranno gli 8 bit a specificare l'*agent*, ma rimarranno ben 22 bit ($2^{22} = 4194304$ nodi!) per la numerazione progressiva dei nodi. Un nodo viene creato unicast per default dal simulatore; può comunque essere creato un nodo multicast con un apposito comando.

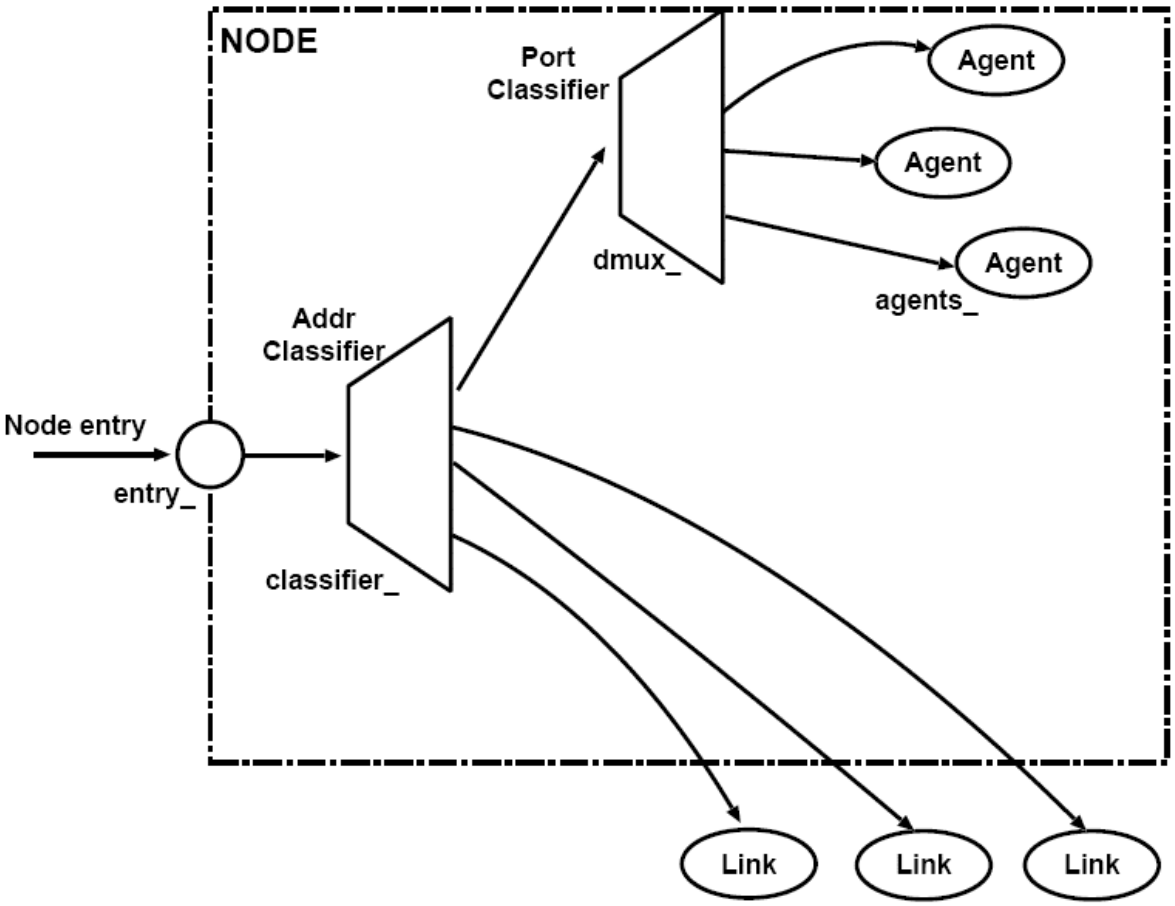


Fig.9: Struttura interna di un nodo

I Link

Anche in questo caso alla creazione di un link verrà istanziato un corrispondente oggetto Otcl *Link* che si preoccuperà della simulazione del canale.

I link sono in pratica i mezzi fisici che collegano tra loro i vari nodi della simulazione. Esistono canali punto a punto o canali condivisi come le Lan (Local Area Network). I link possono essere *half-duplex* (unidirezionali) e *full-duplex* (bidirezionali). Essi sono caratterizzati da un nodo di partenza e uno destinatario, da una capacità, da un ritardo di propagazione del segnale nel canale e dalla politica di gestione delle code. Un link ha anche un costo, che di default è imposto a 1, ma può essere variato a piacimento assegnando due valori, uno per ogni verso di percorrenza del link. Esiste anche lo stato in cui un link può essere, cioè *Up*, se il canale è attivo, oppure *Down*, se il canale non è attivo.

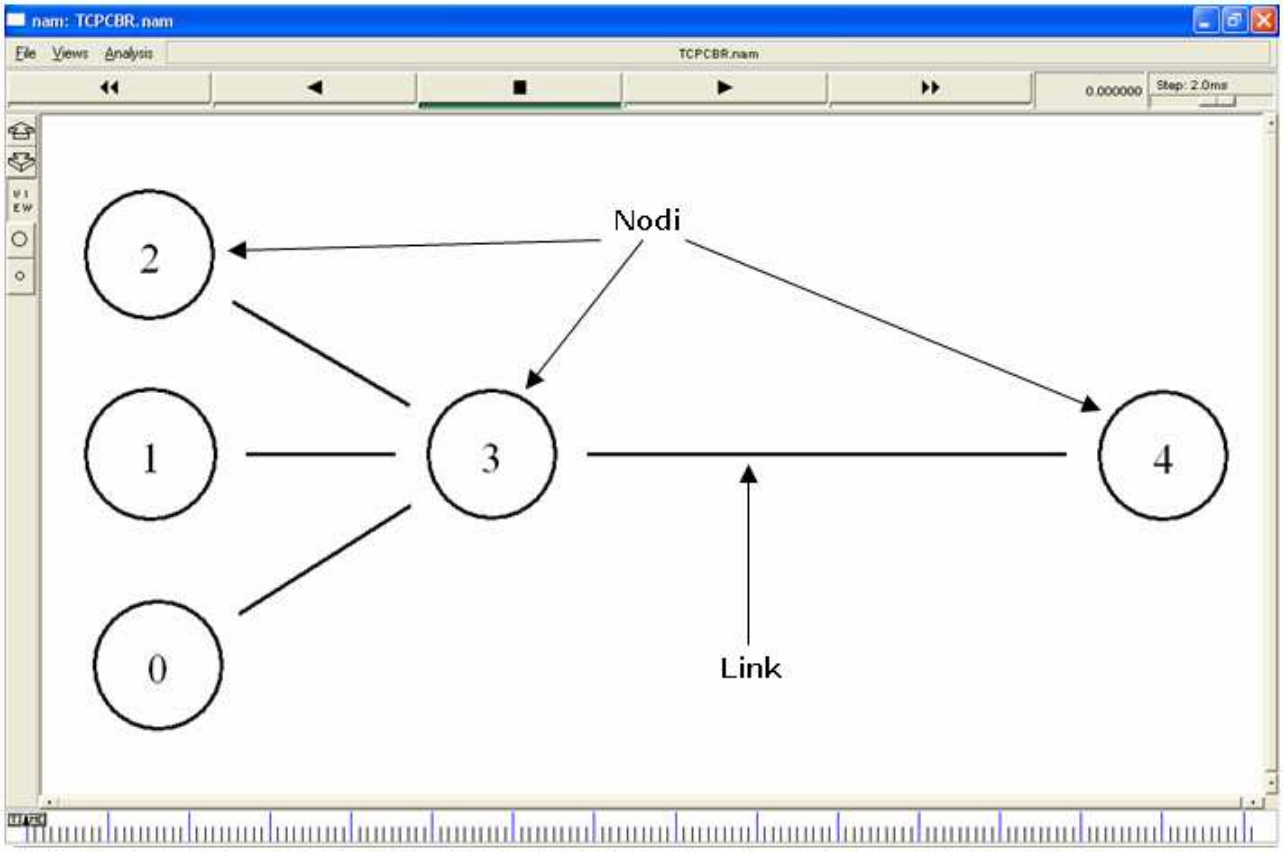


Fig.10: Rappresentazione in ambiente NAM di cinque nodi e quattro Link

All'esecuzione di Nam la topologia desiderata viene a volte visualizzata in un modo disordinato. Esistono dei comandi per direzionare i link ed ottenere così un layout ordinato: si usano i comandi Up, Down, Left, Right, con le varie combinazioni (Left-Up, Right-Down, ecc.).

Nel caso di Lan le caratteristiche principali sono la banda del canale, il delay di propagazione del segnale, il link layer implementato (che si occupa di simulare il protocollo di livello *data link*), la politica di gestione della coda, il tipo di protocollo di accesso al canale (solitamente si usa il CSMA/CD) e il tipo di canale utilizzato; inoltre vanno ovviamente specificati anche i nodi che fanno parte della rete locale in questione.

Gli Agenti

Dopo i nodi e i link, una terza classe di oggetti fondamentale per realizzare una topologia di rete è quella degli agenti (*Agent*). Gli agenti vengono posti sui nodi e sono gli oggetti che rappresentano i processi e/o le entità di trasporto che si trovano su host e router, cioè hanno il compito di simulare sia il livello transport che quello application e di gestire i pacchetti che transitano al livello network. E' importante sottolineare come un agente non sia in grado di creare traffico ma esclusivamente di gestirlo mediante algoritmi opportuni: per la generazione dei dati veri e propri da trasmettere ci si deve appoggiare ad un'opportuna applicazione sorgente, la quale provvederà a generare il traffico necessario e lo passerà quindi all'agente. Tra i principali tipologie di agenti disponibili in NS2 vi sono:

- TCP, un protocollo confermato che gestisce il traffico con controllo della congestione più o meno raffinato a seconda delle versioni, e il calcolo del RTT come i reali protocolli.
- UDP, un protocollo non confermato che solitamente si utilizza per streaming audio o video.
- SINK, da usare come agente ricevente estremamente semplice, che alla ricezione di un pacchetto invia un ACK (acknowledge) al mittente e poi scarta il pacchetto.
- Null è un agente che si limita a ricevere i pacchetti a lui inviati e quindi scartarli.

Ogni agente ha dei parametri configurabili in base alla simulazione che si vuole effettuare: ad esempio per un agente TCP è configurabile la dimensione dei pacchetti, la dimensione della finestra di trasmissione o la porta usata per la trasmissione.

Per poter effettuare lo scambio di dati tra un agente mittente ed uno destinatario è necessario instaurare una connessione tra essi. Va detto che la connessione tra una sorgente ed il destinatario è unica; se si devono quindi mettere in comunicazione due sorgenti con il medesimo destinatario dovranno essere istanziati due agenti mittenti sui nodi trasmettitori e due agenti destinatari sull'unico destinatario. In questo modo quest'ultimo sarà in grado di riconoscere i flussi di dati inviati dalla prima sorgente piuttosto che dalla seconda. Talvolta più agenti trasmettono utilizzando uno stesso link, rendendoli di difficile identificazione quando visualizzati in Nam; per questo motivo esiste la possibilità di assegnare un colore al traffico prodotto da ogni sorgente.

Quando si applica un agente ad un nodo vengono scritti gli 8 bit nell'indirizzo del nodo corrispondenti al codice dell'agente creato.

Un agente crea un'istanza della classe *agent* di Otcl, ed ogni tipo di agente ha la sua istanza: ad esempio il TCP avrà l'istanza *Agent/TCP* in linguaggio Tcl.

Le sorgenti di traffico

Come accennato in precedenza, un agente necessita di un generatore di traffico per poter trasmettere pacchetti. In NS2 le sorgenti di traffico sono denominate *Traffic*, che in Otcl sono delle sottoclassi dell'oggetto *Application*. Una sorgente di traffico fornisce i pacchetti all'agente con certe caratteristiche, proprie di ogni sorgente.

Il più semplice generatore di traffico è il *CBR* (Constant Bitrate), ovvero un trasferimento ad un rate costante dei pacchetti. Le caratteristiche configurabili principali sono la velocità di trasmissione dei pacchetti, la loro dimensione e l'intervallo di tempo che passa tra un pacchetto e l'altro.

Il secondo è il traffico *Exponential* (esponenziale), cioè un generatore di tipo on/off. Quando la sorgente è on il traffico viene generato con rate costante definibile a priori dall'utente, con intervalli di burst (inizio trasmissione) e idle (fine trasmissione) estratti da due distribuzioni esponenziali negative con medie impostate dall'utilizzatore. Viene generato così un traffico con inizio e fine di trasmissione nell'unità di tempo che seguono una distribuzione Poissoniana.

La terza sorgente non è una vera classe di Otcl, in quanto non si basa su algoritmi per la generazione del traffico, bensì si appoggia ad un *Traffic Trace*, ovvero un file di testo dove vengono esplicitamente dichiarati i pacchetti con le loro caratteristiche. Questa opzione è molto utile se si devono analizzare dei traffici con caratteristiche particolari, che non vengono ben simulate da nessuna sorgente di traffico preesistente. Un esempio è la simulazione del traffico ottenuto in una sessione di streaming video o audio tra due terminali: il traffico in questi casi varia in modo completamente casuale, non noto a priori. Può succedere infatti che vengano alternati lunghi momenti di inutilizzo della banda con momenti di elevato carico, senza che esistano delle leggi in grado di prevederle. Questo file deve contenere, per ogni pacchetto appartenente al traffico, un riga costituita da due campi di 32 bit: nel primo campo va inserito il tempo di interarrivo rispetto al pacchetto precedente, mentre nel secondo va inserita la dimensione in byte del pacchetto stesso.

Le code

Ogni nodo possiede un buffer dove accodare i pacchetti che arrivano, in attesa che questi vengano trasmessi oppure scartati. Quest'ultima decisione viene presa dallo scheduler in base alle varie politiche di gestione della coda. La più usata è Drop-Tail, ovvero una coda che risponde alla logica FIFO: il primo pacchetto che entra è anche il primo che esce. La dimensione della coda è configurabile a piacimento impostandone la dimensione o il numero di pacchetti che possono stazionare al suo interno. Lo stato di una coda può essere monitorato in Nam con un'apposita istruzione, che consente anche di visualizzare la perdita dei pacchetti attraverso una semplice animazione.

Le code sono anche un elemento importante per ottenere dati per una successiva analisi: per questo motivo è possibile salvarne in un file lo stato in ogni istante, memorizzando le informazioni sui pacchetti accodati, tolti o scartati. Il formato del file di testo (il *Trace-File*) in questione è il seguente:

```
+ 0.003 4 0 xcp 40 ------- 2 4.0 1.2 0 0
- 0.003 4 0 xcp 40 ------- 2 4.0 1.2 0 0
r 0.013016 4 0 xcp 40 ------- 2 4.0 1.2 0 0
+ 0.013016 0 1 xcp 40 ------- 2 4.0 1.2 0 0
- 0.013016 0 1 xcp 40 ------- 2 4.0 1.2 0 0
r 0.023032 0 1 xcp 40 ------- 2 4.0 1.2 0 0
```

Fig.11: formato del Trace File

La prima colonna rappresenta il tipo di operazione eseguita sulla coda:

- + indica pacchetto accodato
- - pacchetto estratto
- r pacchetto ricevuto
- d pacchetto scartato (*dropped*).

La seconda colonna contiene l'istante della simulazione a cui avviene l'evento.

La terza e la quarta colonna indicano i nodi tra i quale è stata monitorata la coda.

La quinta e la sesta colonna indicano il tipo di trasmissione effettuata e la dimensione del pacchetto.

La settima colonna è riservata a flag di servizio.

L'ottava colonna è utilizzata per reti IPv6, che noi non tratteremo.

La nona e la decima colonna indicano gli indirizzi del nodo sorgente e di quello destinatario.

L'undicesima e dodicesima colonna identificano il sequence number ed una numerazione globale dei pacchetti.

Questi file di traccia, come accennato in precedenza, sono utilizzabili in fase di analisi per verificare il comportamento della rete, estrapolando molti altri parametri oltre a quelli mostrati atti a monitorare particolari evoluzioni o necessari per effettuare calcoli (throughput, quantità di pacchetti presente nel buffer a diversi istanti di tempo etc.); questo fa delle code uno strumento prezioso al quale affidarsi per le analisi post simulazione.

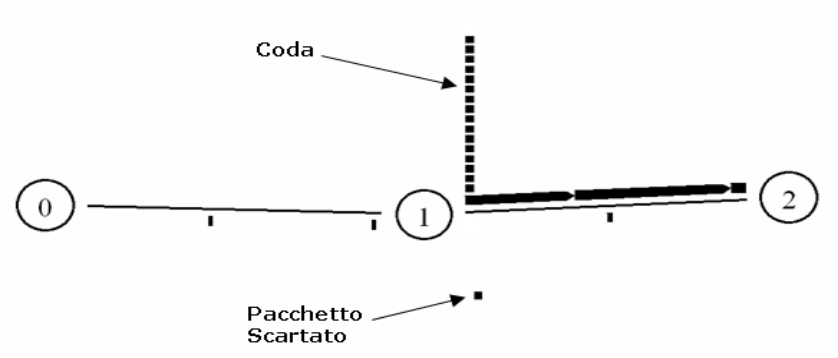


Fig.12: Rappresentazione Nam di una coda e di un pacchetto scartato

I modelli di perdite

Nella realtà in una rete il pacchetto che viaggia non è sicuro né di arrivare a destinazione né di arrivare senza errori per vari motivi, tra i quali la congestione della rete, la caduta di un link o lo scarto da una coda. Questa probabilità di arrivare varia in base al traffico in rete, alla rumorosità del canale e così via. In una simulazione si deve pertanto tenere conto di questa possibilità; tuttavia è proprio la natura casuale delle perdite, che non risulta regolate da leggi vere e proprie, a rendere di difficile gestione la situazione. Fortunatamente esistono dei modelli statistici, costruiti analizzando il comportamento delle reti in determinate situazioni, in grado di descritte soddisfacentemente questi fenomeni. Proprio questi modelli sono stati implementati nel simulatore NS2 e possono essere usati semplicemente settandone alcuni parametri caratteristici.

Un modulo di errore simula gli errori sul livello fisico oppure perdite di pacchetti interi tramite due meccanismi: l'introduzione di errori sul pacchetto, ottenuta impostando l'apposito *error flag* nell'header del pacchetto stesso, e la perdita del pacchetto, simulata consegnando il pacchetto ad una *drop target* invece che al destinatario. In una simulazione gli errori possono essere generati semplicemente definendo una percentuale (probabilità semplice) d'errore su ogni pacchetto, oppure tramite metodi più complessi. Al fine di simulare un'ampia varietà d'errore, la sua unità può essere espressa in termini di pacchetti o bit, oppure basarsi sul tempo. Passiamo ora ad analizzare alcuni modelli d'errore implementati da NS2.

Il primo modello d'errore non si basa su alcuna legge statistica ma semplicemente assegna una *droplist*, cioè un elenco di pacchetti da scartare, al nodo. Gli elementi da scartare vengono identificati in base al proprio *sequence number* e nella simulazione verranno eliminati i pacchetti corrispondenti contenuti nel buffer del nodo interessato. Talvolta le perdite da assegnare possono per comodità essere elencate in un file di testo esterno che verrà poi letto dal simulatore.

Un secondo modello d'errore analizzabile utilizza una distribuzione uniforme nel tempo. Esso può gestire l'errore sia in pacchetti che in bit, dichiarando semplicemente all'inizio della simulazione

l’unità di errore desiderata e la percentuale d’errore voluta. Con questo modello d’errore i pacchetti vengono scartati nel tempo con una probabilità fissa.
Una tipologia di modelli d’errore interessante è la *Two-State*: essa è in grado di comportarsi per alcuni periodi di tempo come un modello senza errori e in altri come un modello d’errore a scelta. Essa è utile in quei casi dove esistono errori solo in un determinato periodo (ad esempio in un momento di congestione della rete) e non in un altro (ad esempio in un momento di basso carico sulla rete).

Gli eventi

Nel caso delle simulazioni ad eventi discreti, ed in particolare in NS2, gli eventi costituisco il “motore” della simulazione, senza i quali non si simulerebbe nulla. Tutte le azioni intraprese dal simulatore sono determinate da eventi: l’inizio o la fine di una simulazione, l’invio di un pacchetto da un nodo, la caduta o il ripristino di un link ed in generale tutti quegli eventi che fanno evolvere lo stato del modello dinamicamente. E’ per questo motivo che viene scelto il metodo Next Event per l’avanzamento temporale della simulazione: si dedica infatti tempo di calcolo solo in presenza di eventi mentre nei tempi di idle essa viene fatta avanzare rapidamente all’evento futuro.
Nel simulatore tutti gli eventi sono oggetti della classe Otcl *at-events* ed in generale sono le procedure la cui esecuzione deve avere luogo ad un determinato istante. L’istante di inizio del tempo nella simulazione è l’istante 0 secondi. Da qui viene fatto partire il tempo per gli eventi successivi.
In generale sono ammesse tutte le procedure che solitamente vengono utilizzate in modo statico, e anche quelle procedure create dall’utente per fare eseguire determinate operazioni al simulatore in determinati istanti. Un esempio può essere la registrazione su un file di testo della routing table ogni n secondi, oppure il calcolo del bitrate di un canale ad intervalli regolari, la caduta dei link ed il loro ripristino etc.

Cap 2.3 Creazione di una nuova simulazione

Fino ad ora abbiamo analizzato i principali componenti di Network Simulator, senza però fare alcun accenno sulla sintassi per invocare gli oggetti o per creare lo scenario di una simulazione. In questo capitolo verrà trattata la sintassi utile alla generazione dei più comuni componenti ed eventi utilizzati. Perciò il capitolo verrà presentato come una guida essenziale ad NS2; tutta la documentazione completa si trova sul sito ufficiale di Network Simulator all'indirizzo *http://www.isi.edu/nsnam/ns/ns-documentation*.

Forniamo prima un accenno sulla forma del codice: generalmente si usa eseguire un comando per riga, salvo particolari segni di punteggiatura che permettono di eseguire più comandi su di una stessa riga. Alla fine del comando, e quindi della riga, non vanno generalmente messi segni di interpunzione (per intenderci, niente punto e virgola in stile Java o C). Le variabili vengono dichiarate col comando *set nomeVariabile* seguite dal valore alle quali devono essere inizializzate e, per richiamare il valore di una variabile, si usa il simbolo '$' seguito dal nome della variabile. Le linee di commento vanno precedute da un cancelletto '#'.

Creazione di un'istanza del simulatore e di Nam

Il principale componente di una simulazione è l'istanza principale di Network Simulator, che a sua volta si incaricherà di gestire tutti gli oggetti e gli eventi creati in seguito. Per questo motivo tutte le simulazioni cominciano con l'invocazione dell'istanza del simulatore:

```
set ns [new Simulator]
```

In questo modo viene creata un'istanza del simulatore battezzata ns: tutte le volte che dovremo invocare il simulatore faremo quindi riferimento alla variabile ns.

Così come tutte le simulazioni iniziano con lo stesso comando, finiscono anche allo stesso modo, più precisamente con l'avvio della simulazione:

```
$ns run
```

Per poter poi visualizzare l'animazione grafica della simulazione è necessario creare un apposito tracciato per Nam, contenente tutti i dati necessari:

```
set nf [open anim.nam w]      #creazione file di animazione
$ns namtrace-all $nf          #tracciato dei dati di animazione
```

Tra queste istruzioni di inizializzazione e terminazione della simulazione verrà poi inserita tutta la sequenza di comandi che daranno origine alla simulazione.
Un'altra operazione che viene sempre effettuata per comodità è la creazione di una procedura *finish* nella quale vengono chiusi tutti i file di traccia e di Nam aperti e si fa uscire il simulatore al sistema operativo:

```
proc finish {} {
     global ns nam
     $ns flush-trace
     close $nam          #chiusura del file di animazione
     exit 0 }
```

Creazione di nodi e link

Le reti sono principalmente formate da nodi collegati tra loro da un canale, sul quale poi viaggeranno le informazioni. Vedremo ora come creare i nodi ed i relativi link che li collegano.
La creazione di un nodo denominato 'nodo1' avviene tramite il comando

```
set nodo1 [$ns node]
```

Di default tutti i nodi sono impostati per simulazioni unicast; se si deve simulare uno scenario multicast bisogna impostare il valore della variabile *EnableMcast_* al valore 1 prima della creazione dei nodi tramite il comando

```
Simulator set EnableMcast_ 1
```

Lavorando in ambiente multicast però il bit più significativo dell'indirizzo dei singoli nodi è riservato e si possono avere quindi al massimo 128 nodi.
Se la simulazione prevede un numero elevato di nodi, per evitare inutili ripetizioni di codice si usa un ciclo *for* tramite il quale si creano N+1 nodi:

```
for { set 1 0 } { $i < N} { incr i} { set n($i) [$ns node]}
```

Per differenziare i vari nodi l'uno dall'altro è possibile assegnare ad ogni nodo un colore diverso dagli altri, in quanto di default sono tutti neri:

```
$nodo1 color "yellow"                    #crea un nodo giallo
```

Dopo la creazione dei nodi si prosegue con la definizione del canale che li collega tra di loro. Esistono due tipologie di link: unidirezionale e bidirezionale. Analizziamo la sintassi per la creazione di un link unidirezionale (*simplex-link*):

```
$ns simplex-link <nodo0> <nodo1> <bandwidth> <delay> <queue_type>
```

Dove node0 e node1 sono rispettivamente i nodi di partenza e di arrivo del link, bandwidth è la capacità del canale espressa di default in bit al secondo (ma è possibile indicarla con i prefissi kB o mB per indicare kiloByte o megaByte), delay il ritardo di propagazione del segnale nel canale espresso in secondi (ma è possibile utilizzare il prefisso ms per indicare i millisecondi) ed infine queue_type indica il tipo di politica di gestione delle code utilizzata. E' possibile fissare anche il numero massimo di pacchetti che possono giacere in coda dichiarando un valore per la variabile *queue_limit*:

```
$ns queue-limit $nodo0 $nodo1 30        #la coda conterrà 30
                                         30 pacchetti
```

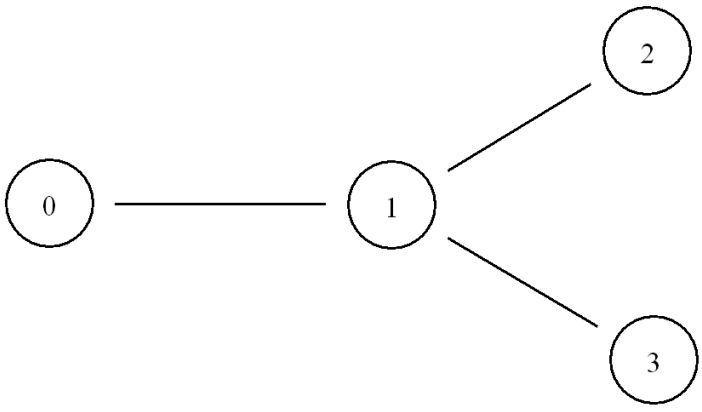
Per la simulazione dei protocolli di routing è importante specificare il costo del link, che di default è impostato a 1:

```
$ns cost $nodo0 $nodo1 10          #il costo del è pari a 10
```

La collocazione spaziale in nam viene effettuata dichiarando la direzione del link:

```
$ns duplex-link-op $nodo0 $nodo1 orient right-down
$ns duplex-link-op $nodo0 $nodo1 orient right
```

Vediamo ora un esempio di script per la creazione dello scenario di rete rappresentato in figura 13:



```
...
#creazione di 4 nodi:
set nodo0 [$ns node]
set nodo1 [$ns node]
set nodo2 [$ns node]
set nodo3 [$ns node]

#creazione link
$ns duplex-link $nodo0 $nodo1 1Mb 10ms DropTail
$ns duplex-link $nodo1 $nodo2 1Mb 10ms DropTail
$ns duplex-link $nodo1 $nodo3 1Mb 10ms DropTail
#direzioniamo i link
$ns duplex-link-op $nodo0 $nodo1 orient right
$ns duplex-link-op $nodo1 $nodo2 orient right-up
$ns duplex-link-op $nodo1 $nodo3 orient right-down
...
```

Creazione di agenti

Dopo aver definito la topologia della rete creiamo ora gli oggetti che si preoccupano di gestire il traffico sul canale, ovvero i protocolli come TCP o UDP che governeranno la trasmissione dei pacchetti sulla rete. Per creare un agente del tipo desiderato si deve creare la corrispondente istanza della classe *Agent*, ad esempio per creare un agente TCP la sintassi sarà:

```
set tcp [new Agent/TCP/FullTcp/Tahoe]  #creazione di un agente TCP Tahoe
```

Una volta creato l'agente, lo si deve porre su di un nodo creato in precedenza:

```
$ns attach-agent $nodo0 $tcp  #posizionamento dell'agente sul nodo0
```

A questo punto il nodo è pronto per supportare un'applicazione per la creazione del traffico.
Ogni agente possiede dei parametri di configurazione che possono essere impostati. Ad esempio per il TCP creato in precedenza alcuni parametri possono essere la dimensione massima del pacchetto

```
$tcp set packetSize_ 500      #il pacchetto sarà di 500 byte
```

oppure l'apertura massima della finestra di trasmissione

```
$tcp set window_ 25           #la finestra massima è 25 pacchetti
```

Occorre infine creare una connessione tra un agente mittente ed uno destinatario affinché possa avvenire lo scambio di pacchetti. Se supponiamo di avere un agente TCP chiamato tcp0 ed uno Null chiamato null0 il comando sarà:

```
$ns connect tcp0 null0
```

Segue una lista dei principali agenti utilizzati:

- TCP/FullTcp/Tahoe: trasmettitore TCP Tahoe
- TCP/ FullTcp: trasmettitore TCP Reno
- TCP/ FullTcp/Newreno: trasmettitore TCP Reno modificato
- TCP/ FullTcp/Sack: trasmettitore TCP SACK che supporta il selective repeat
- UDP: trasmettitore UDP
- TCPSink: ricevitore per Tahoe e Reno
- TCPSink/Sack1: ricevitore per SACK
- Null: ricevitore che scarta i pacchetti

Vi è la possibilità inoltre di differenziare i colori dei pacchetti visualizzati da Nam sulle diverse connessioni per chiarezza in caso di molti traffici:

```
$ns color blu Blue  #assegnamo alla variabile blu il colore blu
$ns color red Red   #assegnamo alla variabile red il colore rosso
$tcp1 set fid_ 1    #assegnamo all'agente tcp1 il colore blu
$tcp2 set fid_ 2    #assegnamo all'agente tcp2 il colore rosso
```

Creazione del traffico tra i nodi

Una volta creati gli agenti voluti e dopo averli posti sui nodi opportuni, bisogna passare alla creazione del traffico che, come già ricordato, deve essere gestita da apposite applicazioni. Le applicazioni che gestiscono il traffico sono raccolte sotto la classe *Application* e le principali sono due: FTP e CBR.

La sorgente di traffico CBR (Constant Bitrate) produce una sequenza di pacchetti con una determinata velocità, che rimane costante per tutto il tempo in cui l'applicazione è attiva. La sintassi per la sua creazione è:

```
set cbr [new Application/Traffic/CBR]
```

Ora è necessario applicare la sorgente ad un agente trasmettitore creato in precedenza, ad esempio all'agente TCP:

```
$cbr attach-agent $tcp
```

In questo modo abbiamo applicato la nostra sorgente all'agente e il nodo è ora operativo. Infine è necessario dichiarare, tramite due eventi, gli istanti di inizio e di fine trasmissione dell'applicazione:

```
$ns at 0.0 "$cbr start"       #la sorgente cbr comincia a
                               trasmettere all'istante 0 s

$ns at 10.0 "$cbr stop"       #la sorgente cbr termina la
                               trasmissione all'istante 10.0 s
```

I parametri configurabili di CBR sono:

- PacketSize_ , ovvero la dimensione costante dei pacchetti generati
- rate_ , la velocità di creazione dei pacchetti
- interval_ , il tempo che trascorre tra la generazione di un pacchetto ed il successivo
- maxpkts_ , il numero massimo di pacchetti da trasmettere

L'altra sorgente analizzata è la sorgente FTP (che solitamente viene usata per creare pacchetti ad agenti TCP), la quale simula come nella realtà la trasmissione di una grande quantità di dati da un nodo ad un altro. Vediamo il comando per la creazione di un traffico FTP:

```
set ftp [new Application/FTP]
```

Come nel caso di CBR, anche FTP deve appoggiarsi ad un agente già esistente e deve avere un istante di inizio e di fine trasmissione.

```
$ftp attach-agent $tcp

$ns at 0.0 "$ftp start"

$ns at 10.0 "$ftp stop"
```

FTP possiede solo un parametro configurabile che è maxpkts_ ovvero il numero massimo di pacchetti da inviare.

Monitoraggio di code e raccolta di dati di interesse

Le code sono dei componenti molto utili dal punto di vista della simulazione, poiché esse possono fornire molte informazione sul traffico che è in transito su un link.
Come prima operazione configuriamo le code. Il tipo di coda viene dichiarata quando si dichiara un link inserendo nel campo <queue_type> la coda desiderata. Creiamo una coda DropTail su un link:

```
$ns duplex-link $nodo0 $nodo1 2Mb 10ms DropTail
```

Passiamo ora alla visualizzazione della coda in Nam per poter tener d'occhio lo stato del buffer con il seguente comando:

```
$ns duplex-link-op n1 n2 queuePos 0.5
```

dove $n1 $n2 sono i nodi tra cui si vuole monitorare la coda e queuePos la posizione della coda.
A questo punto si può configurare dimensione massima della coda, dopo la quale i pacchetti verranno scartati secondo la politica di gestione scelta, dando un valore alla variabile *queue-limit*:

```
$ns queue-limit n1 n2 80            #limite coda a 80 pacchetti
```

Vediamo ora come registrare dati in un file che poi serviranno per successive elaborazioni. Se vogliamo avere un file contenente tutte le partenze, gli arrivi, gli scarti di pacchetti su tutte le code della simulazione bisogna utilizzare il comando

```
$ns trace-all $dataFile
```

dove dataFile è il nome della variabile puntante il file contenente i dati che è stato aperto in precedenza con il comando

```
set dataFile [open nomeFile.tr w]
```

e poi chiuso nella procedura *finish*.
Nel caso in cui si voglia invece monitorare solo una certa coda, è necessario creare delle sonde per il rilevamento dei dati:

```
set mon [$ns monitor-queue $nodo0 $nodo1 [$ns get-ns-traceall]]
```

Queste sonde sono in grado di fornire lo stato di alcune variabile in un determinato istante di tempo e solitamente vengono scritte delle routine che interrogano il valore di suddette variabili regolarmente ogni Δt stabilito, fornendo così l'andamento del valore nel tempo della simulazione. Tramite le sonde è possibile analizzare le seguenti variabili di stato:

- *size_* Dimensione della coda in bytes in un determinato istante
- *pkts_* Dimensione della coda in pacchetti
- *parrivals_* Totale dei pacchetti arrivati fino all'istante t
- *barrivals_* Totale di bytes arrivati fino l''istante t
- *pdepartures_* Totale dei pacchetti partiti fino all'istante t
- *bdepartures_* Totale di bytes partiti fino all'istante t
- *pdrops_* Totale dei pacchetti scartati fino all'istante t
- *bdrops_* Totale di bytes scartati fino all'istante t

Per creare delle procedure in grado di registrare su un file il valore di una variabile in funzione del tempo occorre innanzi tutto creare il file su cui scrivere i dati, come accennato in precedenza. Seguirà poi la creazione della vera e propria procedura, ad esempio per monitorare l'andamento dei pacchetti in coda nel tempo:

```
...
proc coda {} {
     global dataFile coda1_2 #variabili globali
     set ns [Simulator instance]
     set time 0.1   #△t di campionamento
     set coda [$queue1_2 set pkts_]
     set now [$ns now]  #tempo attuale
     puts $dataFile "$now $coda" #scrivo il tempo e la coda
     $ns at [expr $now+$time] "record" #ripete la procedura ogni
     △t
}
...
```

Alla fine della simulazione avremo nel file indicato dalla variabile $dataFile i dati disposti su due colonne: la prima indicherà l'istante di campionamento e la seconda il numero dei pacchetti in coda. Un secondo esempio di interesse è la creazione di una procedura atta a calcolare la velocità di trasmissione dei pacchetti sul canale in funzione del tempo di simulazione. La procedura avrà la seguente sintassi:

```
...
```

```
proc velocita {} {
     global f1 mon ns
     global NumBytePrec       #variabili globali
     set step 0.1             #△t di campionamento
     now [$ns now]       #tempo attuale
     set byte [$mon set barrivals_]    #byte arrivati
     set deltaByte [expr $byte1-$NumBytePrec]
     puts $dataFile "$now [expr $deltaByte/$step]"
     set NumBytePrec1 $Byte1
     $ns at [expr $now+$step] "record"
}
...
```

Nella procedura viene fatta la differenza tra il valore della variabile *byte* e *NumBytePrec*, cioè si ricava la variazione di byte che si ha avuta tra l'istante *now – ⊿t* e *now*, per poi dividerla per lo step di campionamento. Avremo in questo modo il valore della velocità di trasferimento all'istante *now*. La procedura verrà rieseguita ogni *step* secondi e sul file di traccia avremo due colonne rappresentanti l'istante di simulazione considerato ed il corrispondente bitrate in Byte/s.
Le procedure vanno invocate prima della dichiarazione dell'inizio di simulazione tramite l'evento

```
$ns at 0.0 "velocita"
```

Questi file di dati sono utili per la comprensione dell'andamento nel tempo delle variabili analizzate e possono essere analizzate più facilmente grazie a grafici delle variabili considerate. NS2 fornisce nella suite un'utilità grafica chiamata Xgraph che permette di tracciare i grafici dei file di traccia tramite un semplice comando da inserire nella procedura *finish* prima del comando *exit 0*:

```
exec xgraph trace1.tr -geometry 640x400 &
```

Tale tool però è stato sviluppato esclusivamente per ambiente Linux e non per le piattaforme Windows, da noi utilizzate. Useremo perciò strumenti di grafica utilizzabili sotto Windows al fine di poter tracciare i nostri grafici. In figura 13 viene mostrato un grafico rappresentante il Bitrate di un canale in funzione del tempo di simulazione:

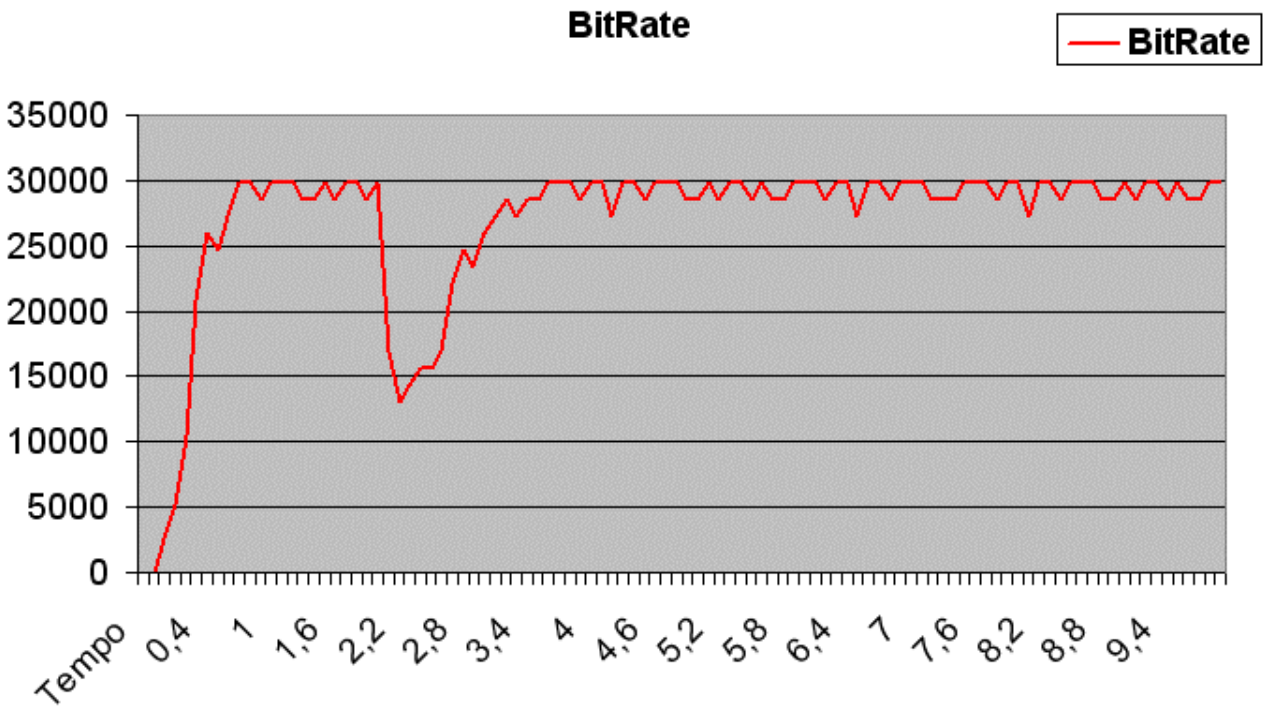


Fig. 13: Grafico BitRate/Tempo

Il vantaggio dei grafici è la possibilità di valutare l'andamento della variabile nel tempo in modo più veloce e diretto che non analizzando una tabella di valori, specialmente in presenza di un elevato numero di dati raccolti.

I LossModel: implementazione dei principali tipi

I LossModel sono dei modelli di perdite dei pacchetti che vengono utilizzati per simulare le perdite di pacchetti come nella realtà; essi agiscono sui pacchetti che transitano attraverso uno specifico link e secondo un determinato algoritmo decidono i pacchetti da scartare.

Il primo LossModel che analizziamo è il più semplice, basato su una *DropList* che scarta gli elementi corrispondenti al *sequence number* dei pacchetti desiderati. In pratica è una lista di numeri che corrispondono ognuno al pacchetto da gettare (la droplist, appunto) la cui implementazione è la seguente:

```
set loss [new ErrorModel/List]          #dichiarazione del modello
$loss unit pkt      #settiamo l'unità di perdita: pacchetto
$loss droplist 50 51 300      #elenco di pacchetti da scartare
```

Dopo aver dichiarato il modello d'errore Droplist e dopo aver elencato i pacchetti da scartare è necessario collegare il modello ad uno specifico link:

```
$ns link-lossmodel $loss $nodo0 $nodo1
```

Così il modello agisce sul link che collega i nodi 0 e 1 scartando i pacchetti con *sequence number* pari a 50, 51, 300.

Questo modello è utile solo per una limitata quantità di pacchetti da perdere, altrimenti la lista diventerebbe molto lunga, ma insostituibile per analizzare il comportamento di un protocollo in presenza di un errore ad un preciso istante.

L'altro modello d'errore che prenderemo in considerazione è quello che si basa su una distribuzione delle perdite uniforme nel tempo, ovvero in ogni istante di tempo ogni pacchetto ha la stessa probabilità degli altri di essere scartato. Analizziamo la sintassi:

```
set loss [new ErrorModel]    #dichiarazione del modello d'errore
$loss unit pkt          #set dell'unità di perdita: pacchetto
$loss set rate_ 0.02    #dichiarazione della percentuale di errore
$loss ranvar [new RandomVariable/Uniform]   #dichiarazione del
                              modello d'errore del tipo uniforme
```

Abbiamo così settato la possibilità di perdita al 2% in ogni momento della simulazione, in quanto la distribuzione dell'errore è uniforme nel tempo. Come in precedenza occorre poi agganciare il modello ad un link:

```
$ns link-lossmodel $loss $nodo0 $nodo2
```

In alternativa si può adottare un modello esponenziale, cioè con una probabilità di errore nei pacchetti che cresce esponenzialmente nel tempo. Anche qui è necessario specificare la percentuale massima di errore nel tempo dei pacchetti. La sintassi è molto simile alla precedente, cambia solo l'ultima riga che diventa

```
$loss ranvar [new RandomVariable/Exponential]
```

Eventi ed esecuzione della simulazione

Dato che NS2 è un simulatore ad eventi discreti, dobbiamo creare degli eventi per fare il modo che la simulazione proceda. In generale sono eventi quelli che fanno intraprendere una certa azione ad un preciso istante di tempo. Solitamente gli eventi sono dichiarati prima della fine della riga di codice ' $ns run' e sono, ad esempio, l'avvio o lo spegnimento di una sorgente di traffico, il lancio di una procedura oppure la caduta o il ripristino di un link:

```
...
$ns at 0.5 "$ftp start"       #ftp comincia a trasmettere
$ns rtmodel-at 1.0 up $nodo0 $nodo1    #disattivazione di un link
$ns rtmodel-at 2.0 down $nodo0 $nodo1  #attivazione di un link
$ns at 4.5 "$ftp stop"        #ftp interrompe la trasmissione
$ns at 4.5 "velocita"         #invocazione procedura “velocità”
$ns at 5.0 "finish"     #invocazione procedura di fine
$ns run             #lancio della simulazione
```

Ogni evento farà cambiare stato al modello e lo farà quindi evolvere in funzione delle operazioni volute.

Una volta creato lo script necessari alla simulazione degli scenari di interesse, il file contenente il codice deve essere salvato in un file di testo con estensione *.tcl*, preferibilmente all’interno di una directory (il simulatore posizionerà gli eventuali file di Nam e di traccia nella stessa cartella dello script). Per compilare si deve digitare quindi dal prompt di sistema il comando:

```
ns nome_file.tcl
```

Il file verrà processato dal compilatore del simulatore e verranno creati i vari file di output del caso. Se la compilazione terminerà senza errori non saranno mostrati particolari messaggi. Nel caso ci fossero invece degli errori nello script, verranno forniti sia il tipo d’errore sia la riga nello script corrispondenti all’errore trovato.

Visualizzazione della simulazione con Network Animator (NAM)

Come accennato in precedenza, Nam è un tool di animazione per reti basato sul linguaggio Tcl/TK in grado di leggere i file di traccia della simulazione e di gestire ingenti quantità di informazioni elevate senza avere la necessità di grandi quantitativi di memoria, dato che i dati di animazione vengono caricati man mano durante la simulazione e non tutti all’inizio.

Il primo passo per l’utilizzazione di nam è la creazione del file di traccia contenenti le informazioni sul layout della rete, sui pacchetti e sui nodi presenti. Il file di traccia deve essere prodotto da Ns e la sintassi per crearlo è la seguente:

```
set namFile [open animazione.nam w]
$ns namtrace-all $namFile
```

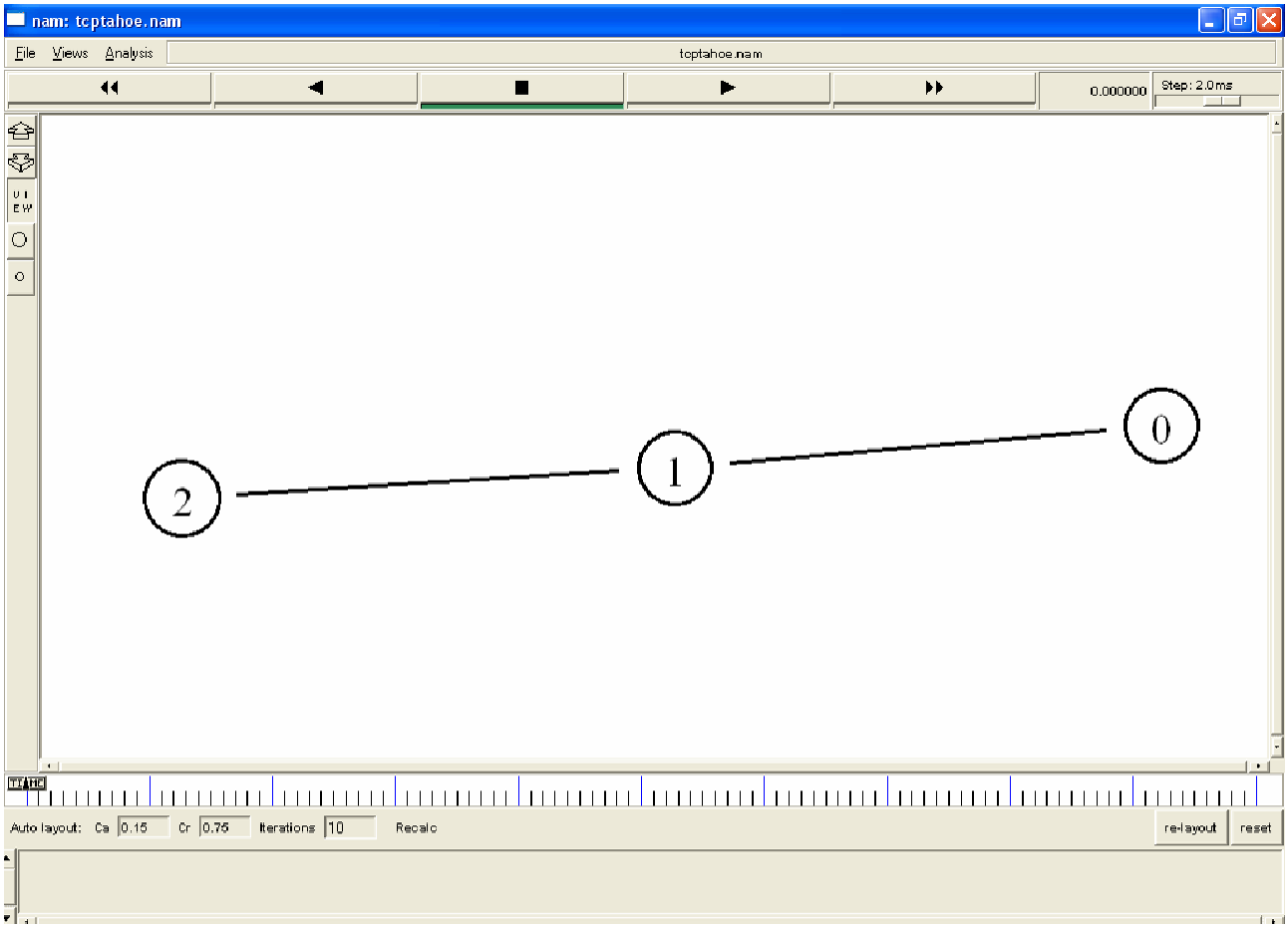
Sarà necessario poi chiudere il file inserendo nella procedura finish la seguente riga:

```
close $namFile
```

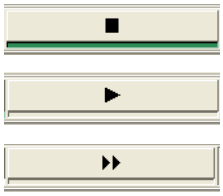
A questo punto alla fine della compilazione dello script Ns verrà creato il file *animazione.nam*, che sarà quello da utilizzare per l'animazione. Per avviare l'animazione desiderata il comando da prompt è il seguente:

```
nam animazione.nam
```

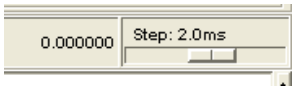
Nam aprirà la finestra di animazione che si presenterà così:



Analizziamo gli strumenti messi a disposizione dal Nam:

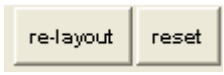


Pulsanti d'azione: fanno partire l'animazione, la fermano o la fanno avanzare velocemente.



Barra di regolazione della granularità del tempo: granularità più bassa equivale a velocità d'animazione minore e viceversa

Strumenti di gestione del layout: zoom in e out, pulsante per spostare i nodi, zoom in e out dei soli nodi.



Comandi per il posizionamento automatico del layout.

Per fare partire l'animazione è sufficiente premere il pulsante play collocato nella barra orizzontale superiore. Si vedranno transitare i pacchetti ad una certa velocità: essa può essere aumentata o diminuita agendo sulla barra di regolazione della granularità aumentandone il valore per avere un'animazione più veloce, o viceversa per averla più lenta, nel caso si debbano osservare eventi di breve durata. All'apertura della finestra di Nam è possibile che la rappresentazione della rete lasci un po' a desiderare: è in questa caso possibile sfruttare la funzione di auto layout premendo il relativo pulsante collocato in basso a destra della finestra, oppure si possono posizionare i nodi manualmente semplicemente trascinandoli con il mouse nelle posizioni volute. L'animazione della rete dovrebbe presentarsi più o meno nel seguente modo:

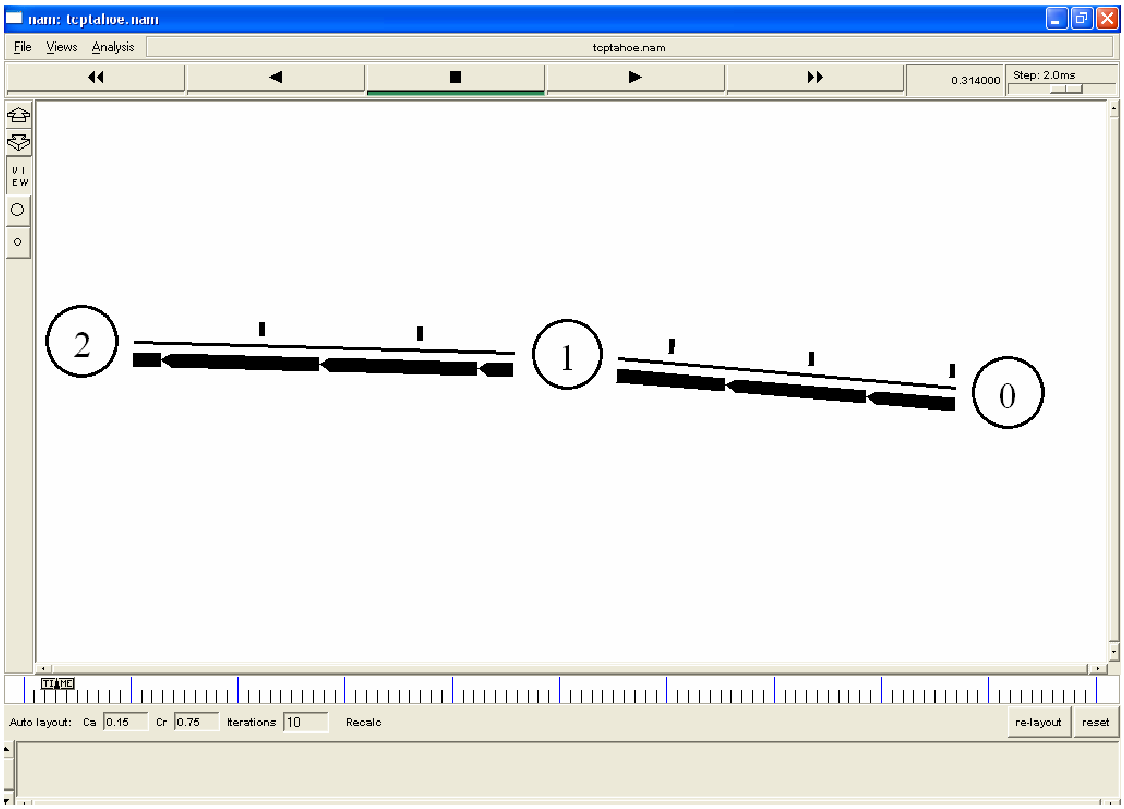


Fig. 14: Rappresentazione della topologia in Nam

Capitolo 3

Simulazione del protocollo TCP tramite NS2

Cap 3.1 Introduzione al protocollo TCP

Una rete di calcolatori è un'insieme di host collegati tra di loro direttamente o tramite dispositivi di interconnessione come router o repeater, in modo che possa avvenire lo scambio di informazioni tra di essi. Tuttavia nella realtà il percorso che deve seguire un'informazione, ad esempio attraverso Internet, è molto impervio e può accadere che l'informazione arrivi a destinazione corrotta o addirittura non arrivi proprio a causa del rumore che si accumula sui canali, per instradamenti errati o situazioni di sovraccarico su alcuni nodi. Questo errore non è sempre accettabile poiché le informazioni errate o incomplete giunte dalla rete potrebbero venir interpretate in maniera non corretta. Per risolvere questa problematica, caratteristica delle rete IO, è stato implementato un protocollo in grado di assicurare che l'informazione trasmessa da un utente venga ricevuta dal destinatario completa e senza errori. Questo protocollo, in grado di garantire il corretto recapito del messaggio su reti IP, è TCP, acronimo di Transfer Control Protocol.

Al giorno d'oggi ci si riferisce genericamente alla suite TCP/IP per indicare quel sistema di protocolli che governano la rete mondiale Internet; in pratica TCP gestisce il controllo della trasmissione con tutti i controlli necessari mentre IP si occupa dell'indirizzamento degli host e dell'instradamento dei dati attraverso la rete. TCP in particolare è un protocollo che si colloca nel quarto strato dello standard OSI/ISO, ovvero nello livello di trasporto, ed è nato per ovviare all'inaffidabilità del protocollo IP, che non garantiva la consegna dell'informazione in maniera corretta e ordinata. Con IP si ha infatti una divisione dei dati inviati in pacchetti, che possono peraltro subire ulteriori frammentazione durante il percorso, potendo così giungere al destinatario in ordine diverso rispetto a quello di trasmissione; TCP fornisce invece una visione dei dati di tipo "a flusso" (*data stream*), cioè il flusso di byte dei dati inviati è ricevuto in sequenza e nello stesso ordine con il quale è stato trasmessi.. TCP poi considera che la trasmissione sia fatta su un canale dedicato in quanto si interessa solo di gestire la trasmissione senza che sia a conoscenza del percorso che dovrà prendere il pacchetto, demandando il compito ad IP. Questo tipo di trasmissione è basata sul concetto di connessione, ovvero sulla comunicazione tra due end-point posti agli estremi della linea, senza tenere conto dei vari passaggi intermedi. Dette connessioni permettono il trasferimento contemporaneo di informazioni in entrambe le direzioni, ovvero un trasferimento *full-duplex*. Si hanno cioè due flussi di dati che scorrono in sensi opposti senza che essi interagiscano tra

loro, permettendo al ricevente di inviare dati di controllo mentre il mittente sta ancora inviando dati. Vediamo ora in che modo il TCP garantisce una certa affidabilità sui dati trasmessi. Il meccanismo di base utilizzato sia dal TCP che da molti altri protocolli cosiddetti "affidabili" è quello della ritrasmissione in caso di mancata conferma. Si tratta di un meccanismo concettualmente semplice: ogni qual volta uno dei due interlocutori di una connessione spedisce dei dati, attende una conferma dell'avvenuta ricezione. Se questa arriva entro un tempo stabilito, chiamato Time-Out, viene spedito il pacchetto successivo, altrimenti l'applicazione rispedisce quello precedente. Questo meccanismo risolve il problema dei pacchetti persi o danneggiati, in quanto TCP prevede un campo di *checksum* per il controllo degli errori, ma può crearne un altro. Supponiamo che a causa di problemi di congestione della rete un pacchetto ci metta molto più tempo del previsto ad arrivare. A questo punto il mittente, non vedendosi arrivare indietro la conferma ne rispedisce una copia. Succede così che il destinatario riceve a una certa distanza l'uno dall'altro due copie dello stesso pacchetto. Il problema della duplicazione dei pacchetti viene risolto facendo numerare in sequenza al mittente tutti i pacchetti da spedire e facendo verificare al destinatario la sequenza ricevuta. Naturalmente questo non vale solo per i messaggi ma anche per le conferme agli stessi. Infatti anche una conferma potrebbe venire erroneamente duplicata. Per evitare questo ogni conferma riporta il numero di sequenza del messaggio a cui si riferisce, permettendo così al mittente di verificare che a ogni messaggio spedito corrisponda una e solo una conferma di ricezione. Naturalmente nella realtà il protocollo implementa altri controlli che variano in base alla versione del protocollo stesso, alcuni dei quali verranno analizzati in seguito.

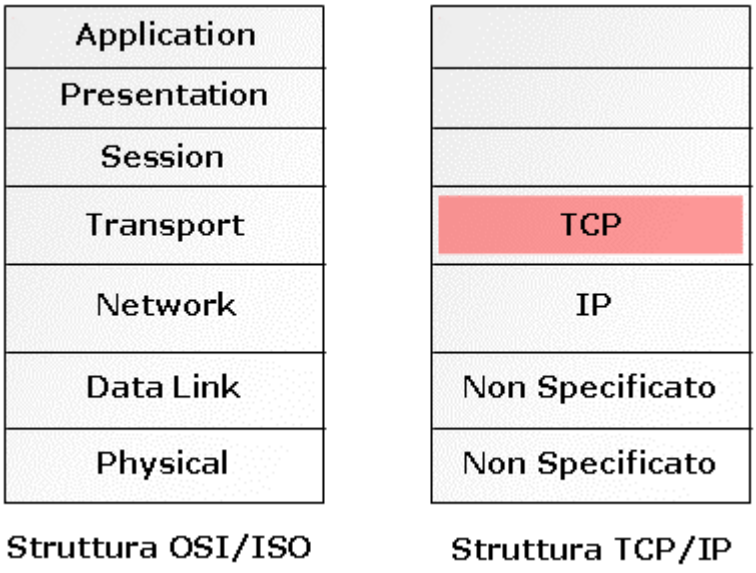


Fig. 15: Corrispondenza OSI/ISO e TCP/IP

Cap 3.2 Il problema della congestione e soluzioni proposte

Abbiamo detto come il problema della perdita dei pacchetti avvenga nella maggior parte dei casi a causa della congestione della rete. La rete può essere congestionata per vari motivi quali la presenza di molto traffico, che porta ad un eccessivo riempimento dei buffer nei nodi intermedi, o più semplicemente per la diversa velocità di trasmissione del mittente e di ricezione del destinatario. TCP implementa tecniche in grado di gestire il controllo della congestione della rete tramite il controllo di flusso dei dati che passano nella connessione. Si tratta delle tecniche che consentono ad una sorgente di adattare il proprio tasso di trasmissione a quello correntemente disponibile dal ricevitore e dalla rete; esse quindi regolano e controllano il livello del traffico introdotto nella rete dal mittente, evitando che avvenga una congestione. Senza questo tipo di controllo si avrebbero degli effetti negativi sulle prestazioni di tutta la rete: infatti al crescere del carico cresce anche il ritardo del canale in quanto le risorse della rete sono limitate e si incombe in ritrasmissione inutili dovute allo scattare dei meccanismi di recupero dell'errore; queste ritrasmissioni non fanno altro che peggiorare la situazione dato che aggiungono traffico inutile ed avviene un sovraccarico. In figura 16 si nota come il Throughput sul canale aumenti con il crescere del carico fino ad un certo valore; da questo punto in poi tuttavia l'elevata presenza in rete di inutili dati duplicati provoca un brusco crollo delle prestazioni.

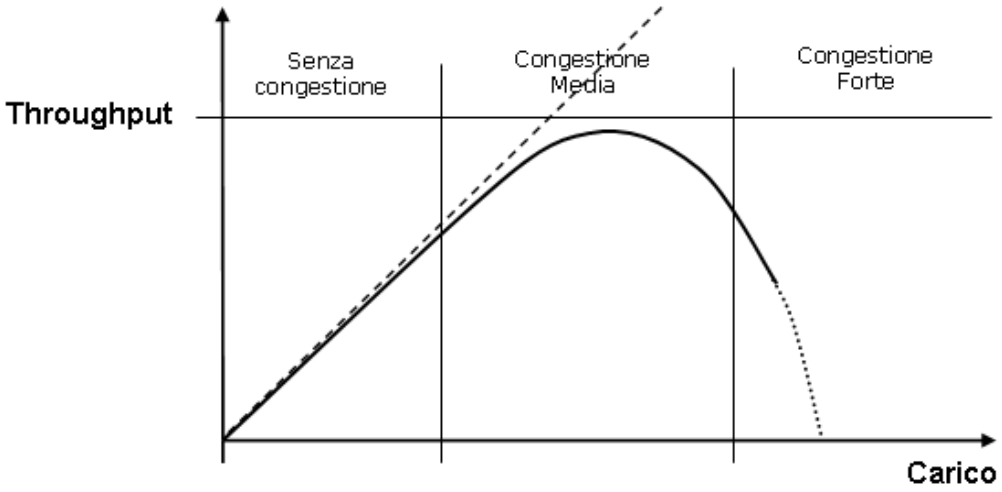


Fig. 16: Rapporto tra Throughput e carico

Il controllo di flusso è quindi uno strumento che si propone di operare un controllo della congestione e, in particolare, di prevenire il riempimento dei buffer sia dei nodi intermedi che del destinatario, garantendo l'effettiva consegna in sequenza dei pacchetti senza errori.

Esistono due tipi di controllo di flusso, uno cosiddetto ad anello aperto e l'altro ad anello chiuso. I controlli ad anello aperto sono preventivi, quindi descrivono le caratteristiche del traffico a priori e costringono a comportarsi di conseguenza il mittente; questo tipo di controlli tuttavia funziona correttamente se tutte le risorse previste risultano poi effettivamente disponibili. Questo lo rende un sistema troppo rigido e quindi poco adatto alle grandi reti che sono molto dinamiche ed inaffidabili. La famiglia di sistemi di controllo ad anello chiuso è stata più sviluppata, implementando protocolli in grado di adattarsi alla situazione incontrata durante la trasmissione e basati sulla retroazione di dati di controllo da parte del sistema. In particolare il meccanismo di regolazione del flusso di dati ad anello chiuso può avvenire secondo due distinte modalità: la prima utilizza esclusivamente le informazioni ricevute dal destinatario, mentre la seconda si basa anche sulle informazioni riguardanti lo stato del canale. Del primo metodo fanno parte sistemi come l'*On-Off*, che consiste nella segnalazione esplicita del destinatario sul blocco o riattivazione della trasmissione, oppure lo *Stop-and-Wait*, che consiste nell'attendere la conferma di avvenuta ricezione del dato dal destinatario per un certo periodo, allo scadere del quale il pacchetto viene considerato perso e quindi ritrasmesso. Al secondo metodo appartengono invece tutti quei controlli che deducono dall'osservazione di particolari eventi rilevati agli estremi della connessione lo stato della rete stessa. Questo verrà ad esempio determinato in base al valore dell'RTT (Round Trip Time), che rappresenta il tempo trascorso dall'istante di trasmissione di un pacchetto all'istante in cui ritorna la conferma di avvenuta ricezione da parte del destinatario: se RTT assume valori più elevati del previsto allora si conclude che la rete è congestionata e TCP mittente si comporterà di conseguenza.

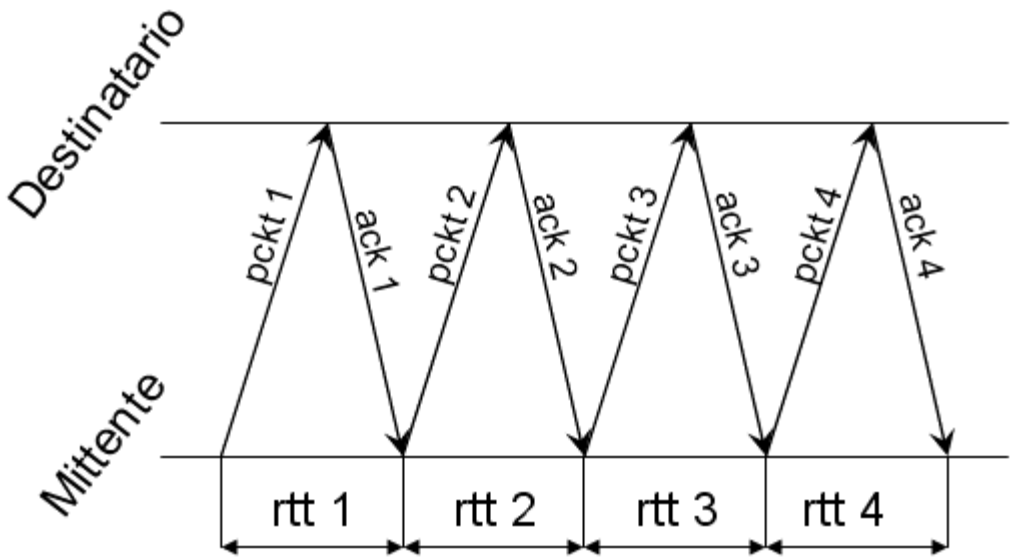


Fig. 17: rtt

Se l'RTT aumenta il proprio valore, il mittente rallenta la trasmissione per non rischiare di alimentare ulteriormente le cause della congestione. Tuttavia se esso assume valori troppo elevati,

in particolare se supera il tempo di *time-out*, il pacchetto verrà considerato perso e ritrasmesso. TCP implementa un altro metodo per il rilevamento della congestione in rete che si basa sull'analisi dei messaggi di conferma che gli arrivano dal destinatario. Se la rete è congestionata i buffer dei nodi intermedi potrebbero essere pieni e alcuni pacchetti potrebbero quindi venire scartati; in questo caso il ricevente notificherà al mittente che non gli è stato recapitato un determinato messaggio inviandogli un ACK con il numero di pacchetto mancante. Supponiamo ad esempio che il mittente invii quattro pacchetti e durante il tragitto vada perso il secondo pacchetto: il destinatario riceverà il primo pacchetto aspettandosi che il successivo sia il secondo, ma gli arriva il terzo e poi il quarto. Esso reagisce scartando il terzo ed il quarto pacchetto ed invierà due ACK contenenti entrambi il codice del messaggio non recapitato, cioè il secondo. A questo punto il mittente, vedendo gli ACK duplicati, ne deduce che la rete è congestionata e, oltre a ritrasmettere il pacchetto perso alla scadenza del time-out, cambierà il valore di alcune variabili che analizzeremo in seguito. Questo sistema necessita di migliorie in quanto ad ogni perdita la trasmissione rimane inattiva per un certo periodo di tempo, andando ad abbassare il throughput efficace della trasmissione; introdurremo in seguito alcune migliorie a questo modello per risolvere questo problema ed evitare di aggravare lo stato di una rete nel caso in cui questa sia già congestionata.

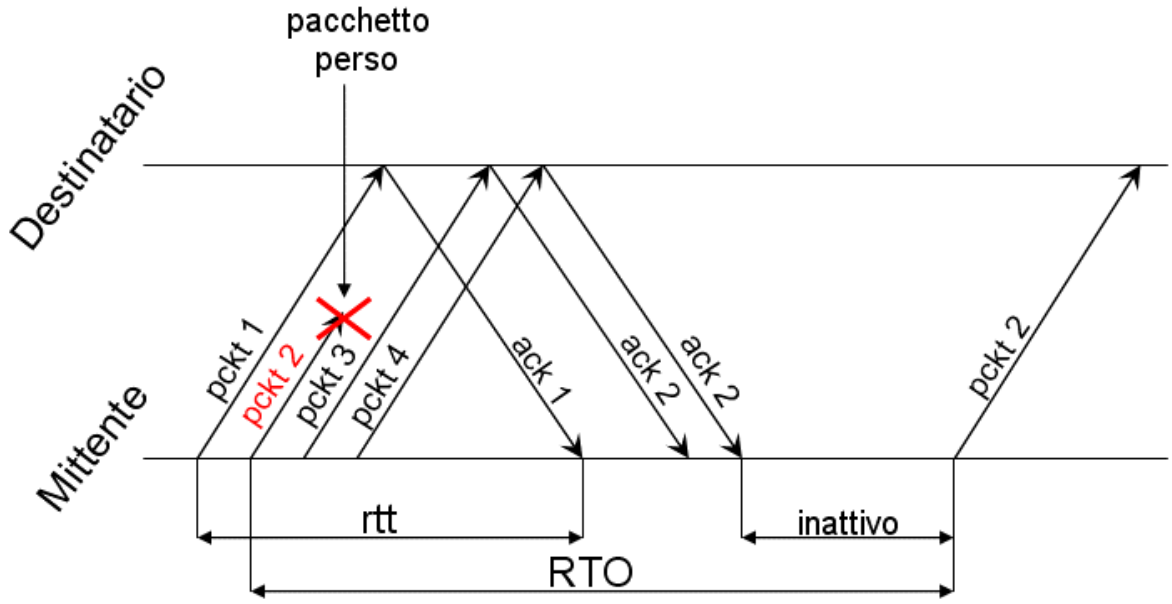
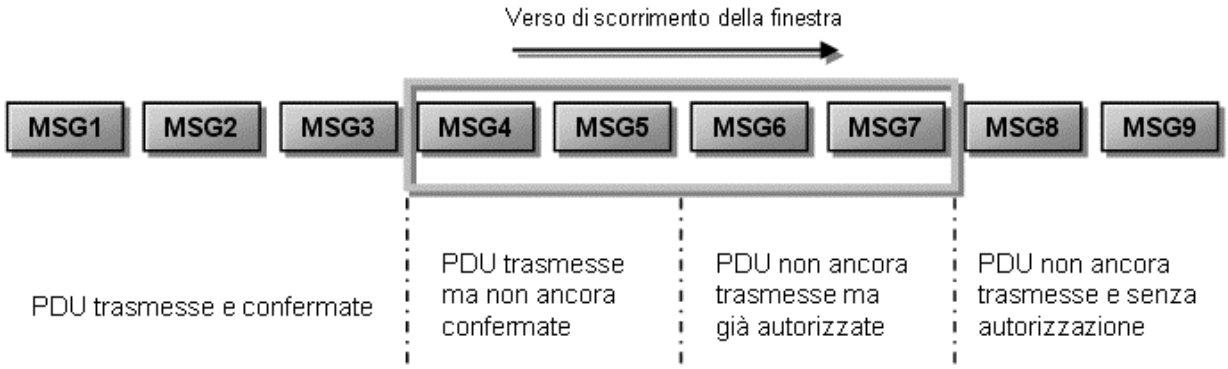


Fig. 18: Reazione di TCP ad una perdita. Si noti il tempo di inattività di trasmissione

Analizziamo ora gli algoritmi di Congestion Control sviluppati per migliorare l'efficienza del TCP, utili per migliorare la gestione dei pacchetti persi nei casi in cui la rete risulti già congestionata e prevenendo la congestione stessa. L'aspetto fondamentale del TCP è l'utilizza di una trasmissione dei pacchetti a finestra scorrevole (la *sliding window*), con dimensione della finestra stessa di

trasmissione variabile in funzione di alcuni parametri ricavati dal ricevente e dalla rete. Altra aspetto importante del protocollo è che il sistema di *acknowledgement* è di tipo cumulativo; ogni conferma non si riferisce cioè alle singole PDU bensì alle sequenze complete e corrette ricevute fino all'ultimo pacchetto ricevuto. Questo nella realtà significa che l'ack di TCP è del tipo "prossimo messaggio atteso": ad esempio se fino al messaggio MSG4 non si sono verificati degli errori, il ricevente informerà il mittente che il messaggio atteso futuro sarà il messaggio MSG5. Inoltre TCP imposta la dimensione massima di un *segment* (MSS) esprimendola in Byte.



La dimensione della finestra (Wtrasm) viene stabilita grazie alla comunicazione da parte dell'unità ricevente della dimensione della finestra sopportabile (Wrec) in base allo stato dei propri buffer di ricezione, il valore della quale viene assegnato a Wtrasm: in questo modo si effettua un controllo di flusso *end-to-end* che impedisce ad un mittente troppo veloce di intasare il destinatario più lento. Si è poi introdotta anche una finestra di congestione (Wcong) con il compito di limitare il ritmo di invio in funzione dello stato della rete intermedia; essa agisce sulla Wtrasm tramite la regola Wtrasm=min(Wrec, Wcong). Wrec è comunicata direttamente dal destinatario con una notifica esplicita inserita nei messaggi di ACK, mentre Wcong viene calcolata dal mittente in base ad osservazione fatte dall'esterno della rete. La stima di Wcong viene effettuata quando viene perso uno o più pacchetti, ovvero quando non ne viene ricevuta la notifica di ricezione dal destinatario entro il time-out, attraverso due algoritmi: *Slow-Start* e *Congestion Avoidance*.

Slow Start

Slow-Start è un algoritmo che viene utilizzato per il calcolo della Wcong all'avvio della connection e ad ogni scadenza di un time-out. Esso si basa su due regole principali:

- il valore di Wcong viene impostato ad 1 ad ogni nuova connection o tutte le volte che scatta la ritrasmissione di un pacchetto a causa di un time-out scaduto
- la Wcong viene incrementata di un MSS per ogni ACK ricevuto

Questo algoritmo si propone di arrivare rapidamente ad ottenere una finestra di trasmissione ampia grazie all'incremento esponenziale di Wcong, per poter avere un throughput elevato sin dalle prime fasi. Tuttavia esso non può essere utilizzato troppo a lungo altrimenti si raggiungerebbero delle finestre con valori troppo elevati. Perciò si imposta una soglia di Wcong, detta *Slow-Start Threshold* (*SS-thresh*), raggiunta la quale TCP cambia comportamento e utilizza come algoritmo Congestion Avoidance.

Congestion Avoidance

L'algoritmo di Congestion Avoidance viene utilizzato dopo il raggiungimento di SS-thresh da parte di Slow-Start ed è governato dalle seguenti regole:

- ogni volta che viene rilevato un time-out e si deve ripartire, la Wcong viene dimezzata
- Wcong viene incrementata di un MSS solo all'accertamento che tutte i segment della corrente finestra di trasmissione siano stati ricevuti dal mittente, ovvero all'arrivo di tutti gliACK dei pacchetti appartenenti alla finestra.

Esso fa crescere la finestra lentamente fino al raggiungimento di una soglia superiore imposta per evitare valori critici di Wcong per la rete. Normalmente quando scatta un time-out la Wcong viene dimezzata ed è frequente che il nuovo valore di Wcong sia inferiore a SS-thresh, quindi il sistema ritorna il fase di Slow-Start.

Nei sistemi usati nella realtà i due algoritmi vengono usati in simbiosi e normalmente si utilizza la fase di Slow-Start all'avvio di una nuova connection sino al valore di SS-thresh impostato di default, proseguendo quindi da questo punto in poi con Congestion Avoidance; allo scadere di ogni time-out inoltre si dimezza la Wcong e si imposta SS-thresh a tale valore, facendo ripartire lo Slow-Start sino a quando Wcong risulta uguale a SS-thresh.

In figura 19 viene mostrato l'andamento di Wcong nel caso in cui venga perso un pacchetto all'istante RTT = 8 e il valore iniziale di SS-thresh sia pari a 16. Dall'istante 0 all'istante 6 la Wcong segue un andamento esponenziale dovuto alla fase di Slow Start, all'istante 6 si arriva a SS-thresh (16 MSS) e l'andamento diventa lineare a causa dell'algoritmo di Congestion Avoidance e continua sino all'istante 8 dove avviene la scadenza di un time-out. A questo punto la Wcong viene resettata ad un MSS e riparte l'algoritmo Slow Start sino a SS-thresh, che stavolta è pari alla metà della precedente (ovvero 8MSS), e dall'istante 12 riparte l'algoritmo di Congestion Avoidance.

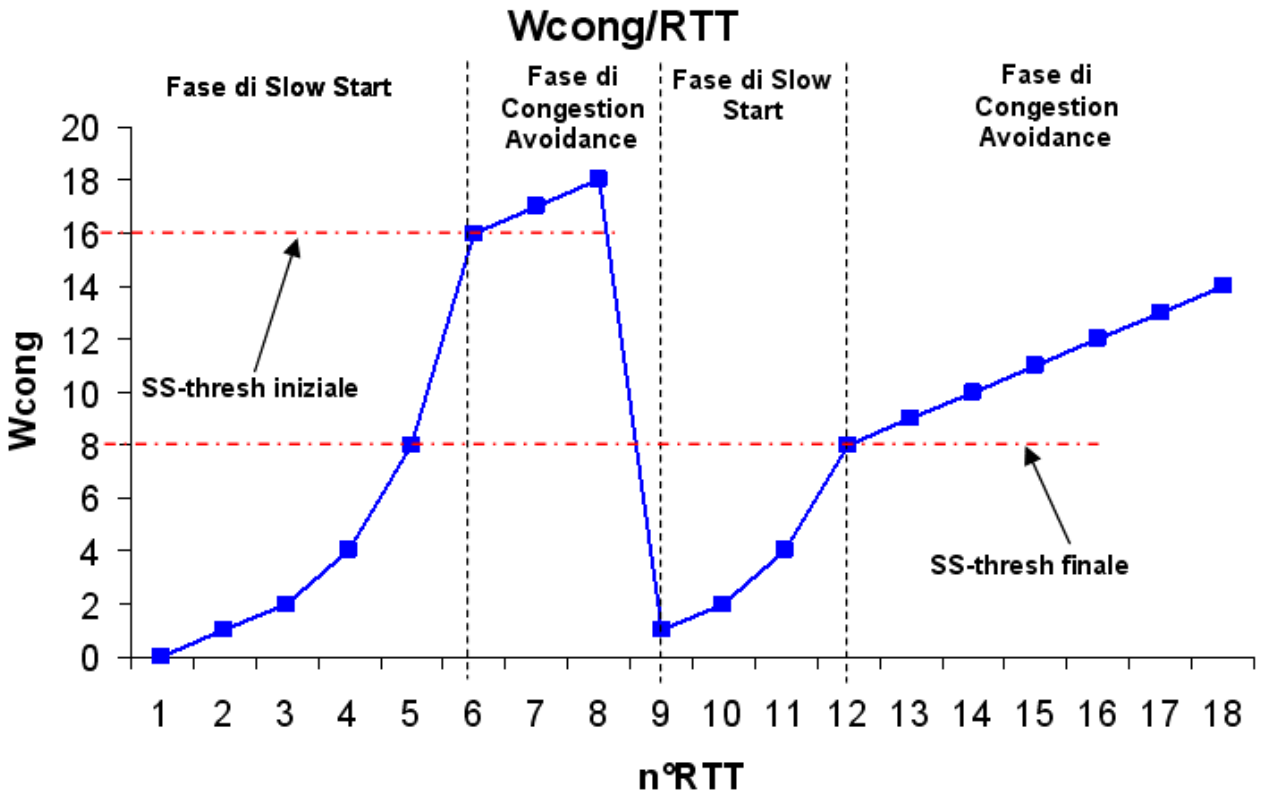


Fig. 19: andamento della Wcong al variare degli algoritmi di Slow Start e Congestion Avoidance

Questo tipo di sistema di controllo degli errori e della congestione ha il difetto di far ripartire da 1MSS la Wcong ogni qualvolta ci si imbatta in un time-out, cosa che porta ad un sottosfruttamento della banda a disposizione. Per questo in alcuni tipi di TCP, al fine di mantenere alto il throughput efficace, non si resetta la Wcong al valore iniziale ma ad un valore opportuno che varia in base alle misurazioni fatte sulla rete.

Esistono inoltre altri algoritmi che reagiscono alla perdita dei pacchetti molto rapidamente rispetto ai due algoritmi visti: essi sono gli algoritmi di Fast Recovery e Fast Retransmit. Vediamoli in dettaglio.

Fast Retransmit

Fino ad ora abbiamo detto che quando un pacchetto viene perso, prima che ne venga effettuata la ritrasmissione deve scadere il time-out corrispondente; questo comporta però ad un considerevole peggioramento delle prestazioni. Per ovviare a questo inconveniente è stato allora pensato un algoritmo che, alla ricezione di un certo numero di ACK duplicati, provochi la ritrasmissione immediata del messaggio richiesto senza attendere il time-out. Questo algoritmo viene chiamato Fast Retransmit e prevede in sostanza che:

- alla ricezione di tre o più ACK duplicati il mittente deduca che il destinatario sta ricevendo pacchetti fuori ordine e che probabilmente il pacchetto richiesto è andato perso
- il mittente ritrasmetta immediatamente il pacchetto corrispondente agli ACK duplicati senza aspettare il time-out

In questo modo, eliminando i tempi di inattività del mittente causati dall'attesa dei time-out out, si ha uno sfruttamento migliore della banda. L'algoritmo è stato implementato per la prima volta in UNIX BSD 4.3 TAHOE Release.

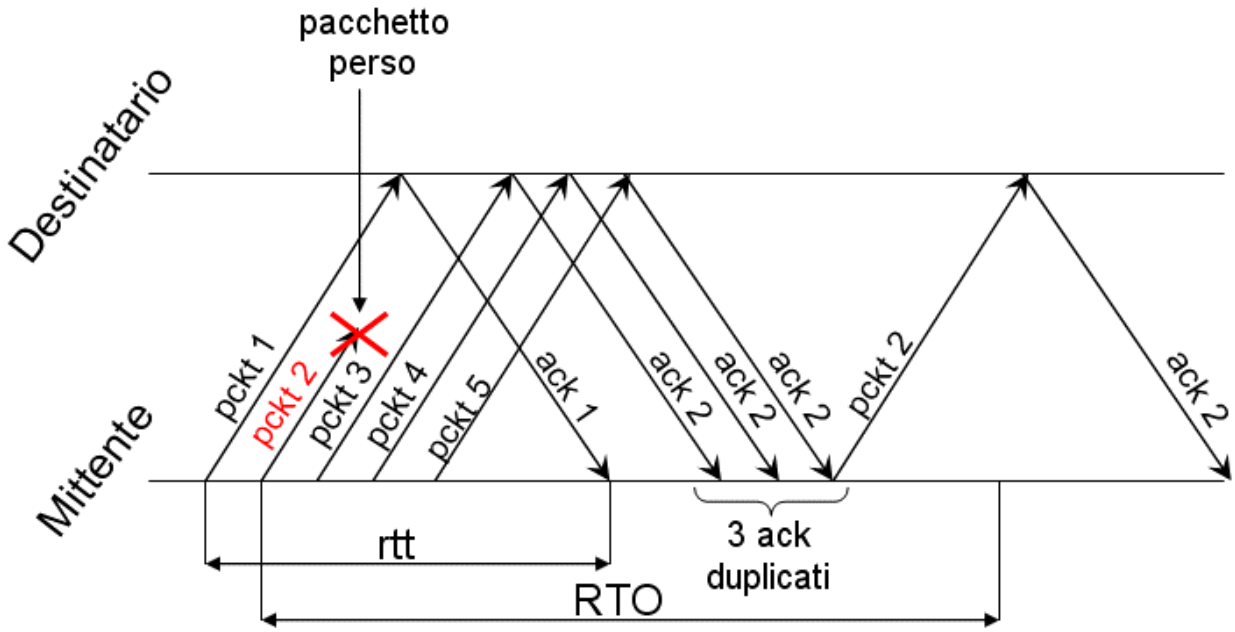


Fig. 20: Funzionamento algoritmo di Fast Retransmit

Fast Recovery

L'algoritmo di Fast Recovery si pone l'obiettivo di eliminare lo svantaggio di dover ripartire da Wcong = 1 ogni volta che viene rilevata la perdita di un pacchetto, sia tramite ACK duplicati che tramite la scadenza di un time-out. I due eventi vengono però interpretati in due modi diversi da Fast Recovery: in caso di ACK duplicati si comprende che il destinatario sta ricevendo messaggi fuori sequenza, sinonimo di rete moderatamente congestionate. Nel caso invece di scadenza di time-out la rete deve per forza essere molto congestionata in quanto i pacchetti non arrivano nemmeno a destinazione. Fast Recovery si comporta nel seguente modo:

- All'inizio di una nuova connessione e tutte le volte che scatta un time-out si implementa il meccanismo di Slow Start, in quanto nel caso di scadenza di time-out la rete è congestionata e la situazione necessita di attenzione
- ogni volta che vengono ricevuti tre o più ACK duplicati scatta, se implementato, il meccanismo di Fast Recovery o comunque la ritrasmissione del pacchetto. A wcong viene

assegnato un valore pari alla metà del suo valore precedente e si continua con l'algoritmo di Congestion Avoidance

Questo algoritmo è stato implementato per la prima volta nella versione di UNIX BSD 4.3 RENO Release.

Tuttavia gli algoritmi visti sono ad ora vengono talvolta impiegati congiuntamente come nel caso del TCP RFC2581 che implementa Slow-Start, Congestion Avoidance, Fast Retransmit e Fast Recovery. Ecco il principio di funzionamento di tale versione:

- Ogni volta che scade un time-out e all'apertura di una nuova connection si utilizzano gli algoritmi di Slow-Start e Congestion Avoidance
- all'arrivo del terzo ACK duplicato si imposta SS-thresh = Wtrasm/2
- il segmento perso viene ritrasmesso e si imposta Wcong = SS-thresh + 3MSS
- si incrementa di un MSS la Wcong per ogni ulteriore ACK duplicato ricevuto e trasmettere tutti i segment permessi da Wcong
- alla ricezione del primo ACK non duplicato si imposta Wcong = SS-thresh e si procede con Congestion Avoidance, in quanto tutti i segment persi sono stati ricevuti.

3.3 Analisi di alcune versioni di TCP tramite NS2

Vediamo ora come poter simulare il comportamento di TCP nelle versioni Tahoe, Reno, SACK, in particolare studiandone le caratteristiche e descrivendone un modello utile alla comprensione del funzionamento. Obiettivo del capitolo è esaminare la simulazione del modello esaminato in base alle specifiche teoriche per poter poi verificare quanto il simulatore fornisca elaborazioni corrette.

TCP Tahoe

La versione Tahoe di TCP implementata nel simulatore esegue il controllo della congestione ispirandosi al TCP della versione Unix 4.3BSD "Tahoe" rilasciato dall'università del Berkley.

Tahoe è una versione del protocollo che implementa i seguenti algoritmi:

- Congestion Avoidance
- Slow Start
- Fast Retransmit

La finestra di congestione viene incrementata di un MSS alla volta per ogni ACK corretto ricevuto durante la fase di Slow-Start, ovvero quando Wcong < SS-thresh, mentre viene aumentata di 1/Wcong ad ogni ACK ricevuto nella fase di Congestion Avoidance, ovvero quando è verificata la condizione Wcong > SS-thresh. TCP si accorge della perdita di un pacchetto dovuta alla congestione della rete quando vede un certo numero di ACK duplicati, che di default è 3, oppure quando scatta un time-out. In entrambi i casi esso reagisce impostando la SS-thresh alla metà del valore della finestra di trasmissione corrente, che a sua volta è calcolata come il minimo tra la Wcong e la Wrec; inoltre viene riportato il valore della Wcong al suo valore iniziale, che normalmente fa rientrare l'algoritmo in fase di Slow-Start.
Vediamo ora come fare a verificare se effettivamente Tahoe si comporta in questo modo grazie ad NS2. Dobbiamo per prima cosa dichiarare la versione di TCP che vogliamo utilizzare e dobbiamo inizializzarne l'agente. Per prima cosa creeremo però la topologia della rete: due nodi estremi avranno le funzioni di mittente (Nodo0) e destinatario (Nodo2), mentre il comportamento della rete intermedia verrà simulato da un terzo nodo, al quale faremo scartare i pacchetti per simulare una congestione. Entrambi i link avranno la stessa velocità per evitare problemi di accodamento dei messaggi, in modo tale da non scartare pacchetti per code piene ed inficiare così la bontà della simulazione.

```
...
#Creazione dei nodi
set Node0 [$ns node]
set Node1 [$ns node]
set Node2 [$ns node]
#Creazione dei link
$ns duplex-link $Node0 $Node1 2Mb 10ms DropTail
$ns duplex-link $Node1 $Node2 2Mb 10ms DropTail
#Creazione dell'agente
set TCP0 [new Agent/TCP/FullTcp/Tahoe]
$ns attach-agent $Node0 $TCP0
set sink0 [new Agent/TCP/FullTcp/Tahoe]
$ns attach-agent $Node2 $sink0
$sink0 listen
```

```
$ns connect $TCP0 $sink0
...
```

In questo modo abbiamo posizionato il mittente sul Nodo0 ed il ricevitore sul Nodo2; sink0 (ovvero il destinatario) viene messo in ascolto in quanto deve solo confermare i pacchetti ricevuti e non deve trasmettere altro. Impostiamo poi il valore massimo che la finestra di trasmissione può raggiungere per evitare che diventi eccessivamente grossa:

```
$TCP0 set window_ 40
```

La decisione di impostare la soglia a 40 pacchetti è stata fatta arbitrariamente per verificare se effettivamente la finestra rimanga stabile al valore impostato una volta raggiunto, essendo peraltro sicuri che in quel momento la trasmissione sia a regime e non in istanti di transitorio. Successivamente possiamo inserire la sorgente di traffico che produrrà i pacchetti da far gestire a TCP. La scelta è caduta su una applicazione FTP in quanto produce pacchetti uguali tra di loro, cosa che aiuta nella successiva analisi dei dati raccolti:

```
set ftp01 [new Application/FTP]
$ftp01 attach-agent $TCP0
```

L’applicazione FTP è stata assegnata al TCP creato in precedenza.
A questo punto la simulazione potrebbe avere inizio, ma la trasmissione sarebbe esente da errori e quindi il TCP non incontrerebbe problemi di trasmissione. Dobbiamo quindi inserire delle perdite, in modo da simulare la congestione della rete. Esse potrebbero essere simulate da modelli stocastici con distribuzione uniforme od esponenziale, ma sarebbe poi difficile analizzare i fenomeni derivanti in quanto non saremmo a conoscenza del preciso pacchetto perso. Per questo utilizzeremo un metodo semplice ma estremamente efficace nel nostro caso, ovvero il modello deterministico, nel quale i pacchetti da scartare sono scelti a priori durante la stesura del codice tramite la *droplist*. Questo ci serve per analizzare il comportamento in caso di errori avvenuti in particolari punti di interesse e non in punti casuali; è utile poi questo approccio nel caso in cui si voglia esaminare il comportamento di TCP in presenza di più errori consecutivi. L’implementazione del modello descritto è semplice ed è la seguente:

```
 ...
#Inserimento di errori deterministici sul link
set loss [new ErrorModel/List]
$loss unit pkt
```

```
$loss droplist 400
#Applicazione del lossmodel su un link
$ns link-lossmodel $loss $Node1 $Node2
...
```

La scelta di scartare il quattrocentesimo pacchetto è stata fatta per avere la perdita quando la trasmissione è a regime e non in istanti di transitorio.
Dopo aver completato la creazione del modello da simulare passiamo alla stesura di procedure atte a registrare i dati che più ci interessano: analizzeremo l'andamento del bitrate sul Link1, l'andamento della finestra di congestione e della soglia SS-thresh. La registrazione di questi dati viene fatta tramite una sonda della coda posizionata sul nodo Nodo1, la quale è così descritta:

```
# sonde per monitorare i rate delle sorgenti TCP
set mon1 [$ns  monitor-queue $Node0 $Node1 [$ns get-ns-traceall]]
```

In questo modo la sonda mon1 potrà essere interrogata dalle procedure al fine di fornire i dati da raccogliere ed elaborare successivamente.
Il bitrate invece non è un parametro esplicitamente fornito dal simulatore e pertanto va calcolato manualmente attraverso la seguente procedura denominata "*rate*"

```
...
# procedura per calcolare il bitrate del canale
proc rate {step} {
     global f1 mon1 ns
     global NumBytePrec
     set now [$ns now]
     set Byte [$mon1 set barrivals_]
     set DeltaByte [expr $Byte-$NumBytePrec]
     puts $f1 "$now [expr $DeltaByte/($step)]"
     set NumBytePrec $Byte
     $ns at [expr $now+$step] "rate $step"
}
...
```

dove *NumBytePrec* è il numero di byte trasmessi dal mittente fino alla precedente iterazione della procedura, *now* è l'istante di simulazione attuale, *step* è il passo con cui la procedura viene eseguita e *Byte* è l'attuale ammontare di byte trasmessi. Il funzionamento della procedura è semplice: viene calcolata la quantità di byte scambiati nel tempo di uno step, ovvero il bitrate del canale. La quantità di byte scambiati è memorizzata in *DeltaByte* facendo la differenza tra *Byte* e *NumBytePrec* ed essa è rapportata all'unità di tempo che è lo *step*: si ha così il rapporto tra una variazione di byte nel tempo, cioè la velocità di trasmissione del canale. La procedura viene ripetuta ogni *step* secondi grazie all'invocazione ricorsiva della procedura e nel file di output si avranno due colonne: la prima conterrà l'istante di calcolo del bitrate, mentre la seconda il valore del bitrate.
L'andamento della dimensione della Wcong e di SS-thresh è monitorato dalla procedura *cwnd,* la quale non fa altro che leggerne i rispettivi valori dal monitor della coda in quanto esse sono grandezze fornite esplicitamente dal simulatore. Ecco la procedura:

```
...
#Procedura per monitorare la Wcong
proc cwnd { step } {
     global ns TCP0 cwnfile
     set now [$ns now]
     set cwn [$TCP0 set cwnd_]
     set ssth [$TCP0 set ssthresh_]
     puts $cwnfile "$now $cwn $ssth"
     $ns at [expr $now+$step] "cwnd $step"
}
...
```

Il file di output sarà composto da tre colonne dove la prima sarà l'istante di registrazione, la seconda sarà il valore della Wcong e l'ultima quello della SS-thresh.

Ora la tutti i componenti della simulazione sono stati definiti ed essa può partire. Il codice completo dalla simulazione in esame è inserita nella sezione 1 dell'appendice A.

Cerchiamo ora di analizzare quello che in teoria sarà l'output della simulazione. Innanzi tutto la Wcong iniziale varrà 1 come da default, per salire esponenzialmente nella fase di Slow-Start sino

alla soglia SS-thresh, raggiunta la quale essa salirà di 1/Wcong per ogni ACK arrivato sino diventare stabile al valore impostato (nell'esempio 40 pacchetti). La SS-thresh resterà fissa al valore di default sino all'apprendimento da parte di TCP della perdita di un pacchetto tramite tre ACK duplicati. Il bitrate avrà un andamento proporzionale a quello della Wcong ovvero esponenziale prima e lineare dopo il superamento di SS-thresh. A questo punto dovrebbe verificarsi la perdita del pacchetto e il destinatario comincerà a mandare al mittente ACK duplicati. Dopo la ricezione di tre ACK duplicati il mittente assumerà la rete come congestionata e farà scattare i meccanismi del caso: la SS-thresh sarà dimezzata e la Wcong resettata al suo valore iniziale. Inoltre grazie al Fast Retransmit il pacchetto perso verrà ritrasmesso senza attendere lo scadere del time-out. Il bitrate avrà una diminuzione dovuta al restringimento della finestra. Da qui poi l'andamento delle variabili analizzate sarà identico a quello descritto in precedenza per la fase iniziale della simulazione. I grafici corrispondenti a tali considerazioni dovrebbero risultare più o meno come quelli in figura 21. Dopo aver dato inizio alla simulazione, e dopo aver atteso pochi istanti, vengono creati i file di output che dobbiamo analizzare. Innanzi tutto lanciamo nam per visualizzare l'animazione e verificare il comportamento così ottenuto. Si nota come la finestra di trasmissione iniziale parte da uno e aumenta in modo esponenziale come previsto e dopo pochi ACK la trasmissione è continua, ovvero il mittente non ferma mai la trasmissione per attendere gli ACK; la velocità di trasmissione è massima. Ad un certo istante si nota la perdita del pacchetto: il mittente dopo svariati ACK duplicati ricevuti si accorge della congestione della rete, ferma la trasmissione e, dato che è implementato il Fast Retransmit, ritrasmette il pacchetto perso. Si nota che la Wcong riparte dal valore iniziale e varia secondo Slow-Start, ma da qui non è possibile capire se poi varia in modo esatto, ovvero secondo Congestion Avoidance sino al valore massimo consentito. Per ampliare gli orizzonti analizzabili è necessario utilizzare i file di traccia creati appositamente in precedenza dal simulatore. Data la notevole quantità di dati da analizzare è necessario l'utilizzo di grafici. Tracciamo quindi i grafici di SS-thresh, Wcong e del Bitrate.

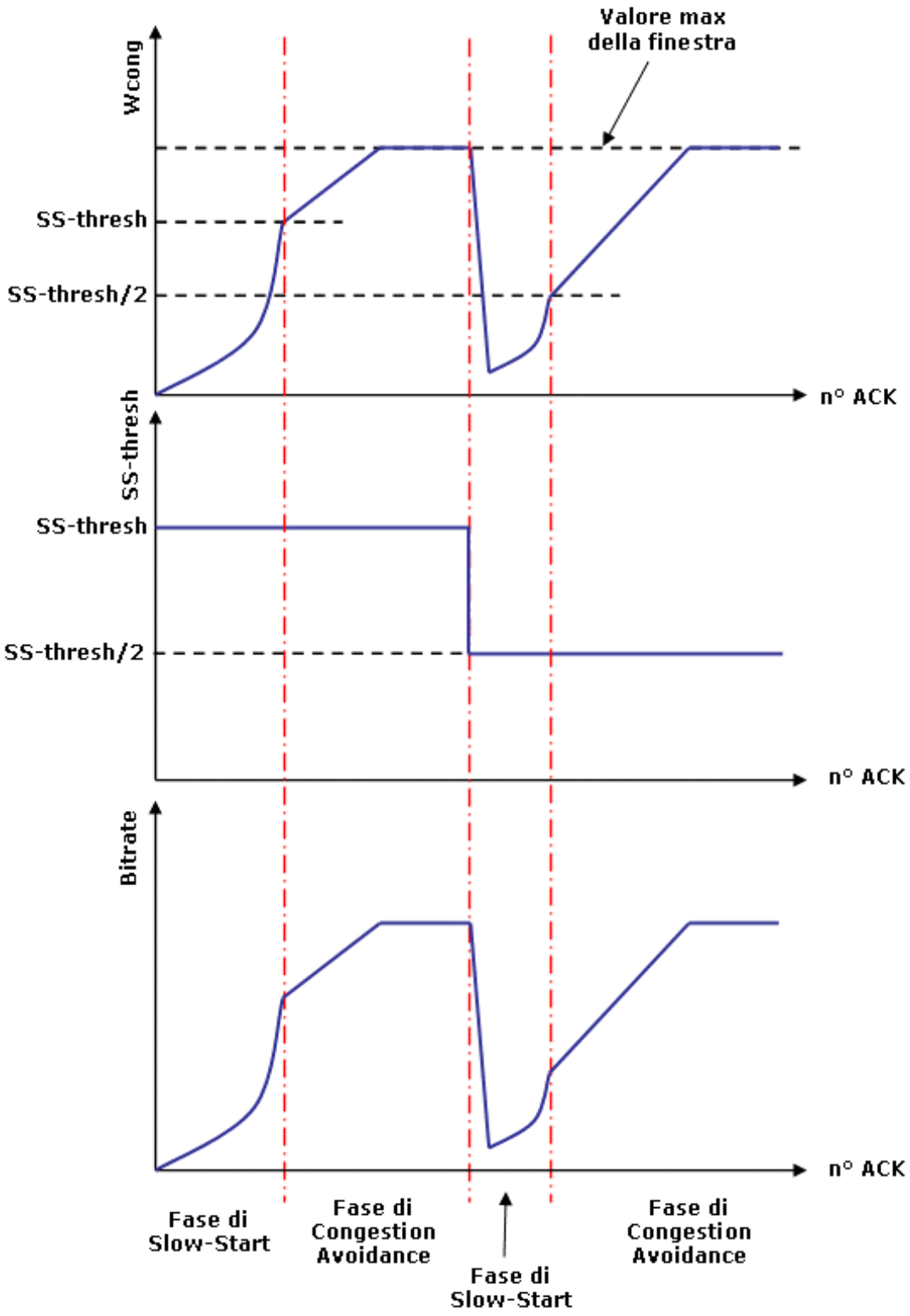


Fig. 21: Rappresentazione dell'andamento della Wcong,
SS-thresh e del bitrate teorico

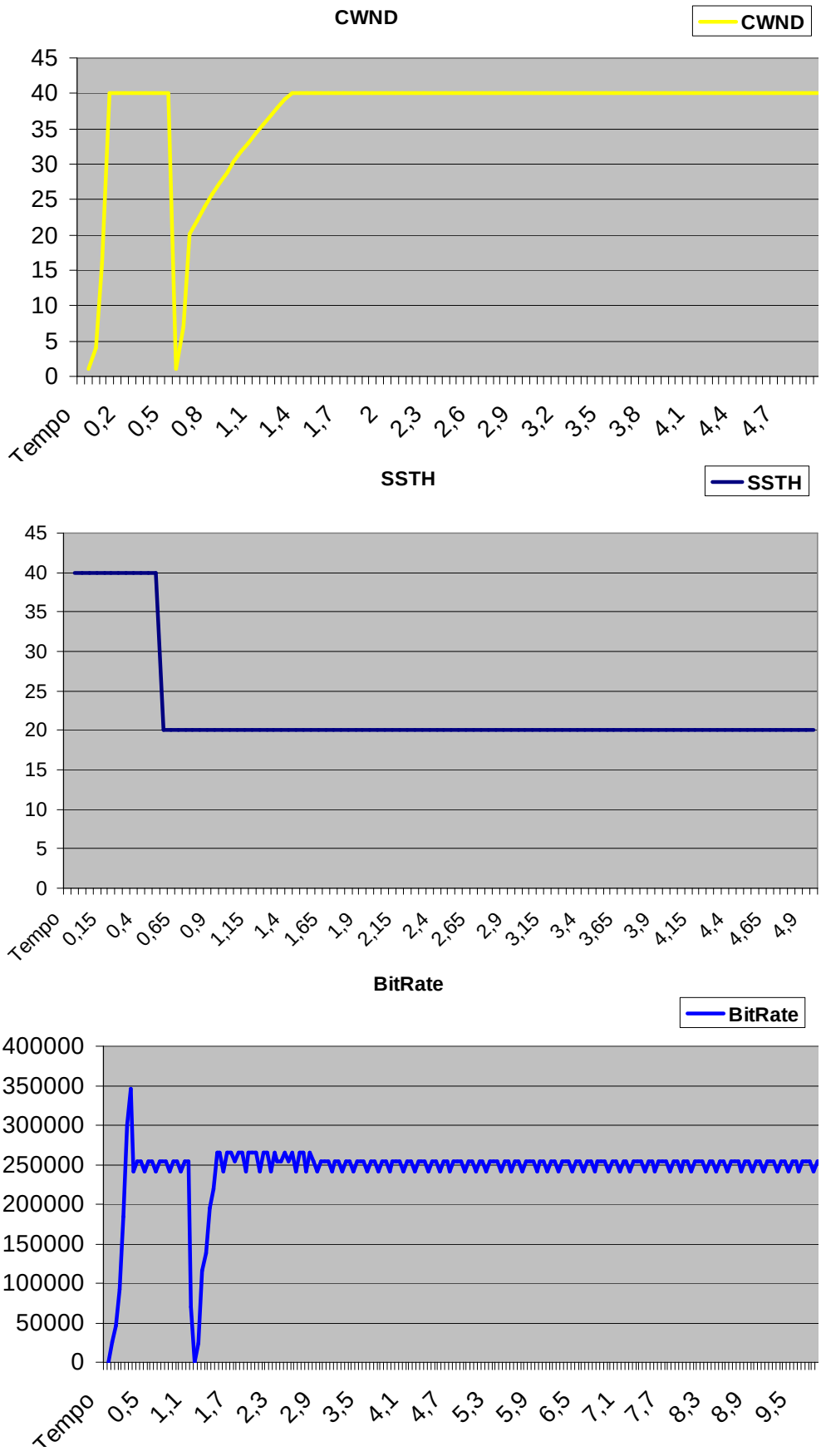


Fig. 22: Rappresentazione dell'andamento della Wcong,
SS-thresh e del bitrate simulato

Come si può notare i grafici della Wcong e della SS-thresh simulati hanno l'andamento previsto, a parte il fatto che nella Wcong all'inizio della trasmissione non si esce dalla fase di Slow-Start perché la SS-thresh è uguale alla dimensione massima raggiungibile dalla finestra e quindi, dopo la crescita esponenziale, non si ha quella lineare ma si ha un appiattimento della curva. Il simulatore si comporta così in quanto tenta di arrivare il prima possibile a regime per avere il throughput più elevato possibile. Il grafico del bitrate risente di questa considerazione e presenta lo stesso andamento, fatta eccezione per il picco che si nota all'istante 0,25s che è probabilmente causato dalle approssimazioni fatte dal simulatore durante il calcolo del bitrate; presumibilmente sono stati presi in considerazione dei pacchetti che erano in minima parte all'interno dello step per il calcolo della velocità, falsando la misura. Al rilevamento dell'errore tramite gli ACK duplicati, gli andamenti sperimentati nella simulazione sono paragonabili a quelli previsti. La finestra di congestione torna alla dimensione iniziale e la SS-thresh viene dimezzata; essendo ora Wcong < SS-thresh si riparte in regime di Slow-Start fino alla soglia, per poi ritornare a Congestion Avoidance una volta superata. La Wcong sale linearmente fino al massimo consentito. L'andamento del bitrate è anch'esso in linea con quanto detto in precedenza; l'andamento non molto regolare una volta a regime è imputabile al fatto che allo scadere di uno step è difficile che l'inizio e la fine dei pacchetti di frontiera coincidano perfettamente con esso, quindi vengono inclusi nel calcolo pacchetti che in pratica appartengono ad un altro intervallo.
Per concludere si può affermare che Tahoe è correttamente implementato dal simulatore perché l'analisi dei dati sperimentali ha verificato molto fedelmente quelli predetti in teoria.

TCP Reno

Passiamo ora all'analisi di una versione migliorata di Tahoe, ovvero Reno. Tahoe si comportava discretamente in presenza di un errore, ma ad ogni errore rilevato la finestra di congestione veniva sempre impostata al valore iniziale, che faceva ritornare il protocollo in Slow-Start. Nella versione Reno questo non accade in quanto è implementato il Fast Recovery. Quindi Reno implementa i seguenti algoritmi:

- Slow-Start
- Congestion Avoidance
- Fast Recovery
- Fast Retransmit

Questa versione reagisce in maniera leggermente diversa rispetto alla precedente: essa infatti al rilevamento della congestione non riporta la Wcong al valore iniziale, ma la incrementa di una unità

per ogni ACK duplicato ricevuto, fino a che non riceve la notifica che tutti i pacchetti della finestra sono stati ricevuti. A questo punto Wcong viene dimezzata in modo da non rientrare in Slow-Start e a SS-thresh viene assegnato un valore pari alla metà di quello precedente.
Lo scenario della simulazione sarà uguale al precedente: stessa topologia, stessa sorgente di traffico, stessi errori. L'unica cosa da cambiare sarà la parte relativa alla dichiarazione dell'agente che sarà Reno al posto di Tahoe. Il relativo codice sarà quindi

```
...
#Definizione della sorgente Reno
set TCP0 [new Agent/TCP/FullTcp]
$ns attach-agent $Node0 $TCP0
set sink0 [new Agent/TCP/FullTcp]
$ns attach-agent $Node2 $sink0
$sink0 listen
$ns connect $TCP0 $sink0
...
```

In questo modo è stato settato l'agente Reno. Dato che tutto il resto dello scenario della simulazione è rimasto identico, passiamo a ipotizzare il comportamento teorico di Reno. Il codice relativo a questa simulazione è inserito nella sezione 2 dell'appendice A.
All'istante zero della simulazione si avrà la Wcong settata al valore iniziale di un MSS, e la SS-thresh al valore di 40, valore impostato dal simulatore come visto in precedenza pari al valore massimo raggiungibile dalla finestra. Con il passare del tempo avremo la Wcong che incrementerà il proprio valore esponenzialmente, grazie all'algoritmo Slow-Start, sino ad arrivare a stabilizzarsi al valore massimo della finestra e senza entrare in Congestion Avoidance (SS-thresh vale quanto la dimensione massima della finestra di trasmissione). Il bitrate si comporterà allo stesso modo. Arrivati al rilevamento dell'errore Reno reagirà incrementando la Wcong di un pacchetto per ogni ACK duplicato ricevuto, ovvero di tre unità (arrivando a 43) per poi dimezzarsi (ed arrivare a 21,5). La SS-thresh verrà dimezzata ed il bitrate avrà una flessione dovuta all'errore per poi risalire linearmente fino a stabilizzarsi al valore massimo ammesso. Anche se la finestra aumenterà il bitrate non varierà in quanto la trasmissione è già continua. Non si rientrerà in fase di Slow-Start. La rappresentazione grafica degli andamenti considerati teoricamente può essere analizzata in figura 23.

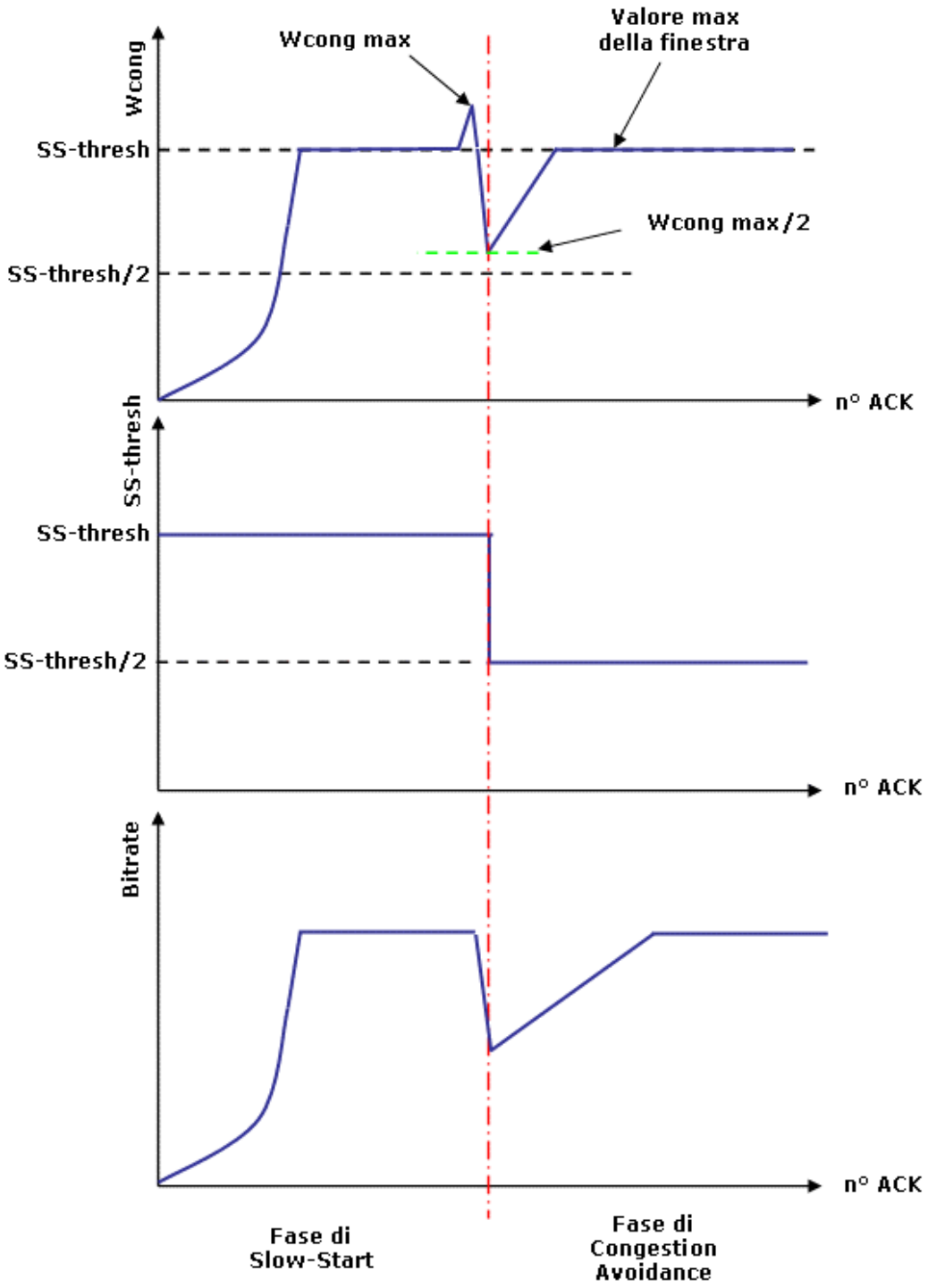
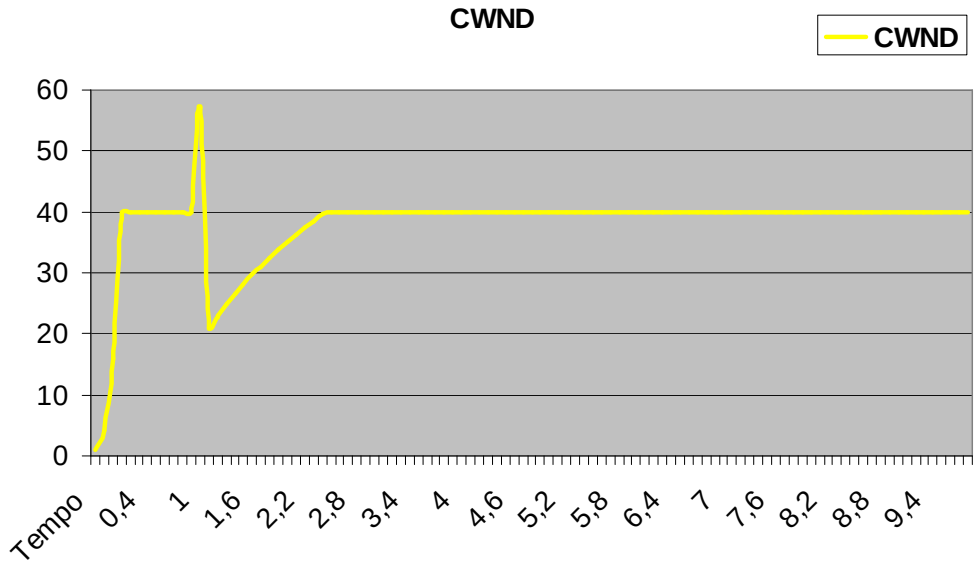


Fig. 23: Rappresentazione dell'andamento della Wcong, SS-thresh e del bitrate teorico

Analizziamo ora i dati ottenuti dopo l'esecuzione della simulazione. Come in precedenza avviamo l'animazione della rete simulata ed analizziamone il comportamento. Anche qui le fasi iniziali sono coerenti con le nostre aspettative: la finestra iniziale è posta ad uno e si nota che sale esponenzialmente fino a raggiungere una larghezza tale da permettere una trasmissione continua. Da qui è impossibile capire se si esce dallo Slow-Start o no, per cui l'analisi di questo fenomeno verrà effettuata tramite i file di traccia. Al verificarsi dell'errore si nota che, come in precedenza, vengono trasmessi svariati pacchetti prima di avere la ritrasmissione, che comunque avviene senza attendere che scada il time-out grazie al Fast Retransmit. Si nota inoltre che alla ripartenza la finestra non è ritornata al valore iniziale ma è rimasta molto estesa, la cui dimensione sarà analizzata tramite i file di traccia. Fino ad ora il funzionamento segue le nostre aspettative. Analizziamo ora le tracce attraverso i consueti grafici in figura 24:



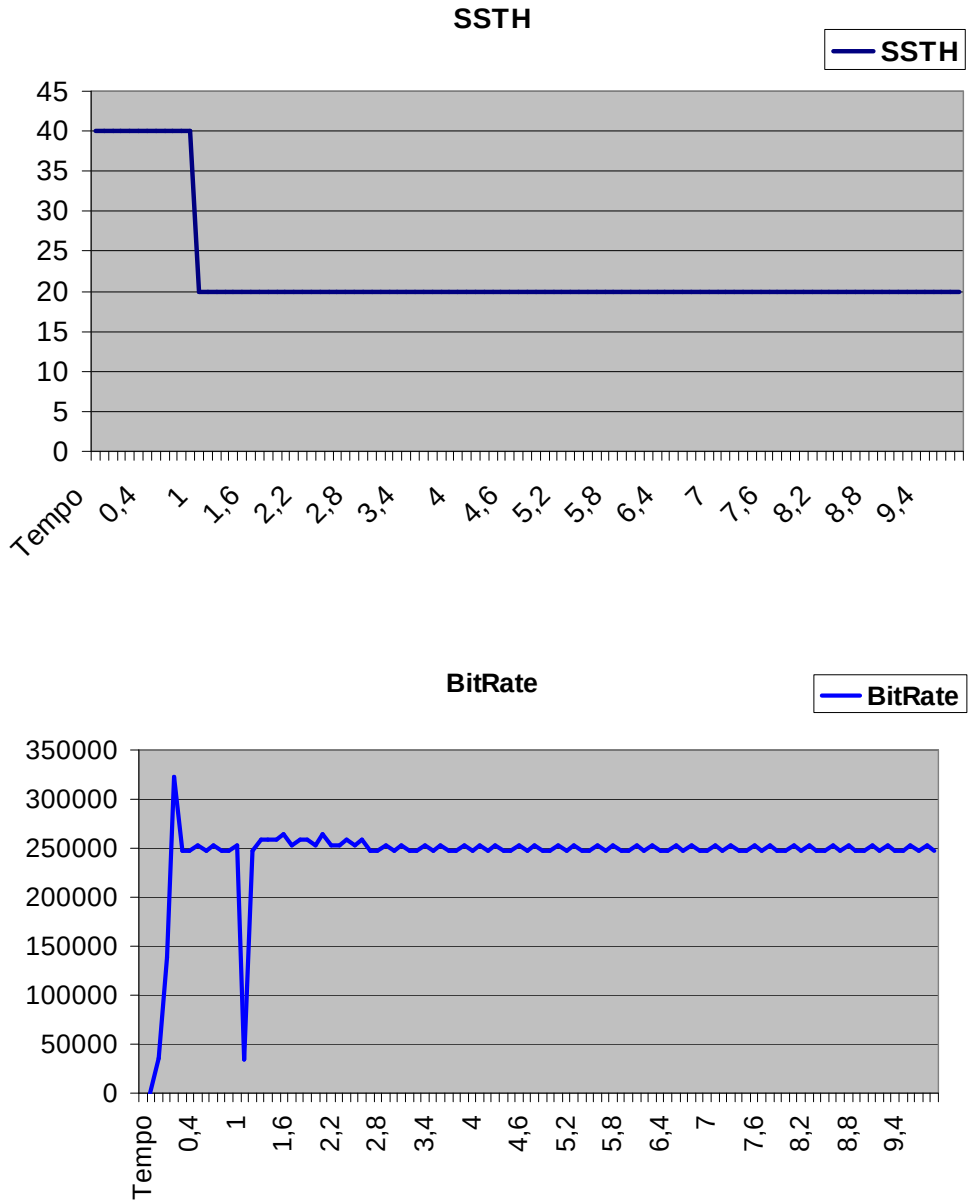


Fig. 24: Rappresentazione dell'andamento della Wcong, SS-thresh e del bitrate simulato

Si può notare come la Wcong salga esponenzialmente come previsto sino alla soglia imposta superiormente e, dopo il rilevamento dell'errore, venga ulteriormente incrementata e poi dimezzata: va detto che le previsioni davano un incremento di tre unità, mentre dalla traccia si può verificare che viene incrementata di ben 17 unità. Tale comportamento è anomalo, soprattutto perché poi la Wcong viene portata al valore di 21, che è il valore da noi previsto e non la metà di 57. Sembra che il simulatore abbia incrementato la Wcong anche dopo aver già ricevuto i tre ACK duplicati. A parte questo dettaglio l'andamento che si ha poi è come teorizzato: la Wcong si incrementa secondo Congestion Avoidance fino ad arrivare al valore massimo permesso. L'andamento della SS-thresh è perfettamente in linea con quanto previsto, infatti parte al valore massimo della finestra, 40 pacchetti, e dopo la notifica di congestione, viene dimezzata. Lo stesso vale per il bitrate, che sale

dapprima con un andamento esponenziale per poi stabilizzarsi. Anche in questo caso vi è il picco visto in precedenza. Alla comparsa dell'errore il bitrate scende e poi risale secondo Congestion Avoidance sino al suo valore massimo. La non stabilità a regime è presente qui come nel caso del TCP Tahoe, probabilmente per le stesse ragioni viste allora.
Anche la versione Reno di TCP può considerarsi implementata correttamente in quanto i risultati ottenuti dalla simulazione sono molto vicini a quelli teorici, salvo qualche imprecisione che può anche essere trascurata.

TCP con Selective Acknowledge

Sino ad ora sono stati analizzate situazioni in cui le perdite si limitavano ad una unità. Va detto però che sia Tahoe che Reno non riescono a gestire in modo efficiente le perdite multiple in uno stesso RTT: infatti entrambe le versioni reagiscono ad una perdita multipla con lo scadere del time-out:

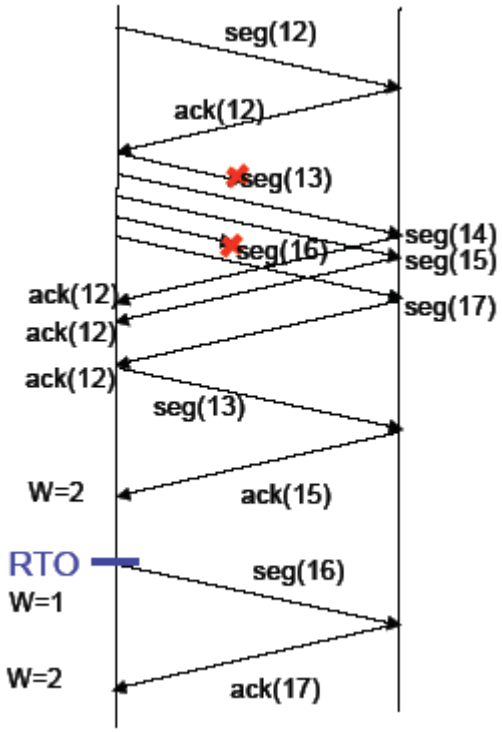


Fig. 25: Si nota come scatti il time-out in caso di perdite multiple nello stesso RTT

Supponiamo ad esempio che i primi 12 segmenti siano stati trasmessi e riconosciuti: tutte le trasmissioni sono fino ad ora andate a buon fine e la finestra è pari a 5 segmenti. I segmenti 13 e 16 vengono poi persi a causa della rete congestionata. Il ricevente notificherà al mittente che stanno arrivando pacchetti fuori sequenza e manderà gli ACK duplicati per chiedere la ritrasmissione del segmento perso. Quando il mittente riceverà 3 ACK duplicati applicherà l'algoritmo di Fast Recovery, ritrasmetterà il segmento 13 e ne uscirà; il ricevente risponderà con una richiesta del

pacchetto 16 tramite un ACK. Questo fa si che il mittente si blocchi in quanto si aspettava l'arrivo dell'ACK riguardante il segmento 17; allo scattare del time-out la trasmissione riprenderà dal segmento 16 con la finestra impostata ad un segmento. In pratica, dato che Tahoe e Reno usano un meccanismo di ACK cumulativo, il mittente può accorgersi di una sola perdita per ogni RTT. In fig. 26 si nota come il bitrate resti per un lungo periodo a zero, dovuto all'attesa del time-out.

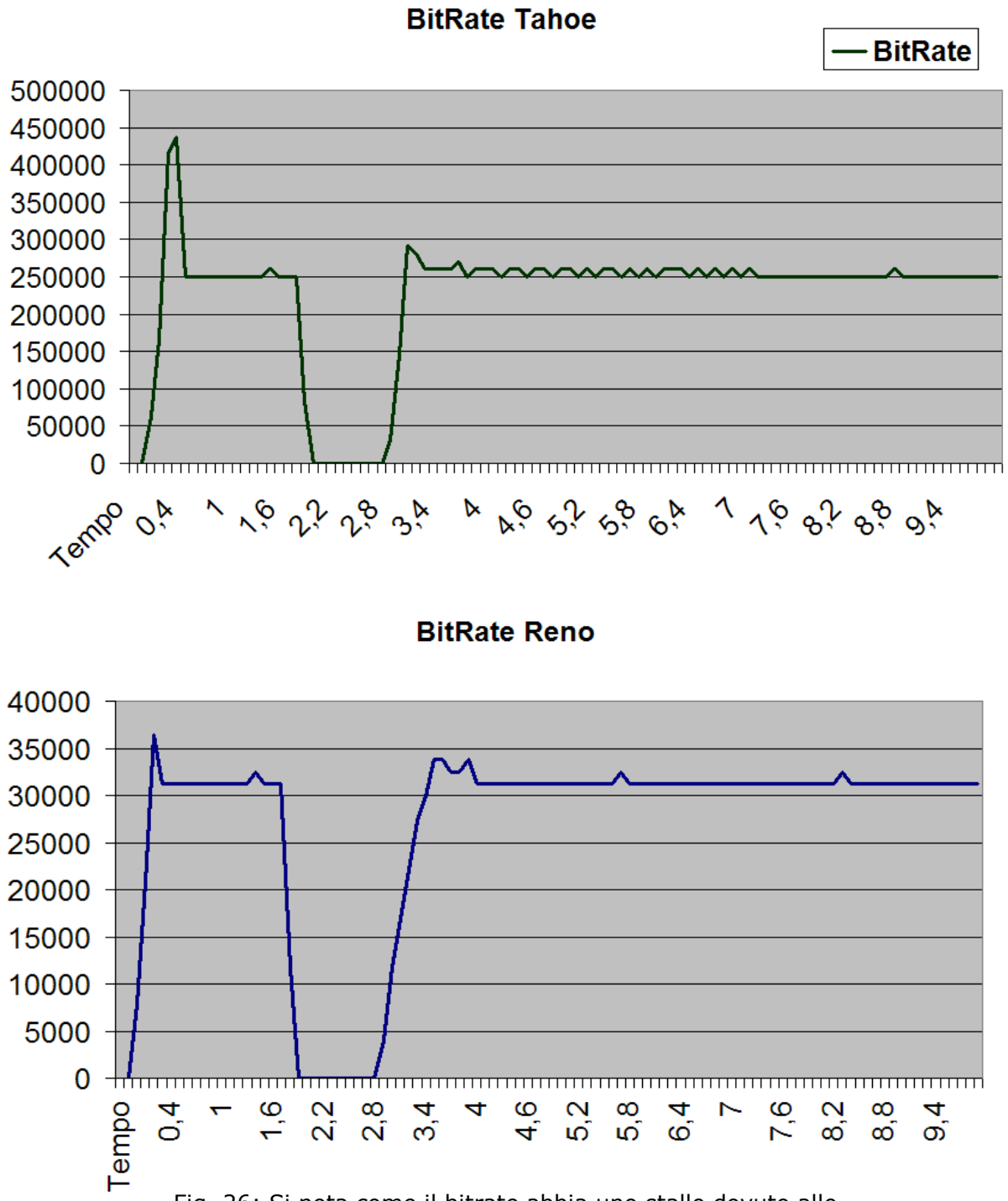


Fig. 26: Si nota come il bitrate abbia uno stallo dovuto allo scattare del time-out

Questo fenomeno ha portato all'implementazione di un nuovo algoritmo tramite il quale possono venire recuperati errori multipli senza incombere in scadenze di time-out: nasce così la versione

SACK di TCP. Esso implementa un meccanismo di ACK selettivo, che combinato con una opportuna politica di ritrasmissione, supera tale limitazione: il mittente non deve aspettare la ricezione di 3 ACK duplicati o la scadenza del time-out per ritrasmettere i pacchetti persi, ma lo fa immediatamente. Un esempio del funzionamento è fornito in figura 27:

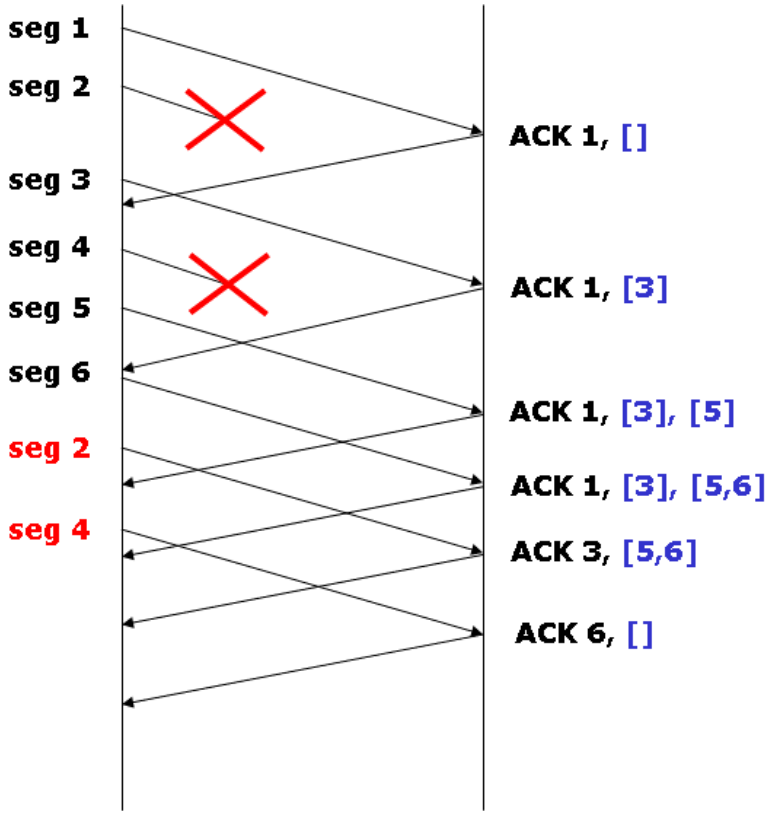


Fig. 27: schema del principio di funzionamento di SACK

Supponiamo che vengano persi i segmenti 2 e 4 e che la finestra sia pari a 6 segmenti. Il ricevente riceverà il primo e manderà la relativa conferma, poi riceverà il terzo, manderà un ACK richiedendo il segmento 2 andato perso e terrà in memoria il 3 nella pipe. Successivamente esso riceverà i segmenti 5 e 6, li salverà nella pipe, ma continuerà a notificare al mittente che manca il secondo segmento. A questo punto il mittente ritrasmetterà il segmento 2, il destinatario avviserà la mancanza del 4 e verrà ritrasmesso. Il nuovo ACK riguarderà quindi il segmento 6 e la trasmissione potrà riprendere, in quanto tutti gli errori sono stati recuperati; questo si capisce anche dal fatto che la pipe è tornata ad essere vuota. Gli algoritmi di controllo implementati da SACK sono una miglioria di Reno, da cui eredita le modalità di gestione delle finestre e delle soglie, e l'introduzione di un nuovo algoritmo per il recupero dei pacchetti persi. Come in Reno, SACK entra in Fast Recovery quando riceve tre ACK duplicati, ma successivamente si comporta in modo diverso: viene utilizzata una variabile *pipe* che contiene le sequenze di pacchetti ricevuti in ordine. Essa sarà incrementata di una unità per ogni pacchetto nuovo o ritrasmesso che il mittente manda ed è

decrementata di una unità per ogni ACK che conferma la ricezione di un pacchetto. L'introduzione della variabile pipe comporta anche al problema di sapere quando e quali pacchetti ritrasmettere, risolto mediante una struttura dati, chiamata *scoreboard,* nella quale il mittente scrive gli ACK ricevuti in precedenza: quando esso riceve l'autorizzazione a trasmettere un nuovo pacchetto, trasmette il primo fuori sequenza della lista che si riferisce al primo pacchetto perso. Il mittente trasmetterà nuovi pacchetti solamente quando il valore della pipe sarà minore di quello della Wcong. Ecco la struttura dell'opzione SACK nell'header di TCP:

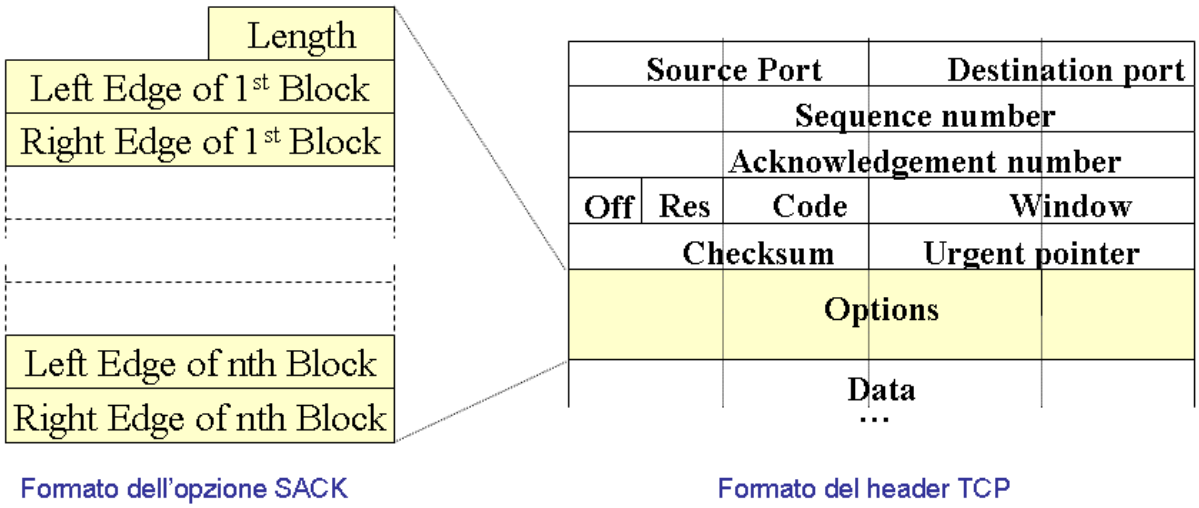


Fig. 28: Struttura dell'opzione SACK nell'header di TCP

Il significato dei vari campi è il seguente:

- Left Edge: il primo numero di sequenza del blocco
- Right Edge: il minimo numero di sequenza non appartenente al blocco
- Block: blocco contiguo di byte ricevuti fuori ordine
- Length: numero dei blocchi non contigui

Passiamo ora a descrivere lo scenario della simulazione. Manteniamo di nuovo lo stesso layout di quelle precedenti, ovvero due end-point tra i quali avviene lo scambio di pacchetti e un nodo che vuole stilizzare tutta la rete intermedia; qui avrà luogo la perdita. Saranno da cambiare i comandi per la dichiarazione del SACK, ovvero

```
...
#Definizione della sorgente Tahoe
set TCP0 [new Agent/TCP/FullTcp/Sack]
```

```
$ns attach-agent $Node0 $TCP0
set sink0 [new Agent/TCP/FullTcp/Sack]
$ns attach-agent $Node2 $sink0
$sink0 listen
...
```

Dopo aver impostato l'agente l'ultima cosa da cambiare è la droplist, alla quale dovrà essere aggiunto almeno un pacchetto per simulare una perdita multipla: aggiungiamo alla droplist il pacchetto successivo a quello di prima:

```
$loss droplist 400 401
```

Per il resto tutto il codice può essere lasciato inalterato; l'intero script è inserito nella sezione 3 dell'appendice A.

Dopo aver lanciato la simulazione analizziamo per prima cosa l'animazione: si nota che la partenza del traffico ha effettivamente un comportamento simile a quello di Reno ovvero la finestra viene aperta con andamento esponenziale. Analizzeremo in seguito il file di traccia per trovarne conferma. Alla notifica dell'errore si nota come la trasmissione non attenda nessun time-out, gli errori vengano recuperati immediatamente e come la finestra venga impostata alla metà della precedente, in accordo con le specifiche di Fast Recovery (si ricordi che SACK è una estensione conservativa di Reno, da cui eredita gli algoritmi). A questo punto è necessario passare all'analisi dei file di traccia, poiché dall'animazione non è possibile capire esattamente il comportamento della finestra di congestione né tanto meno quello di SS-thresh. Anche in questo caso per avere una visione globale degli andamenti di Wcong e SS-thresh, data la grande quantità di dati raccolti, ricorriamo all'utilizzo di grafici opportunamente tracciati.

Partiamo dall'analisi della Wcong:

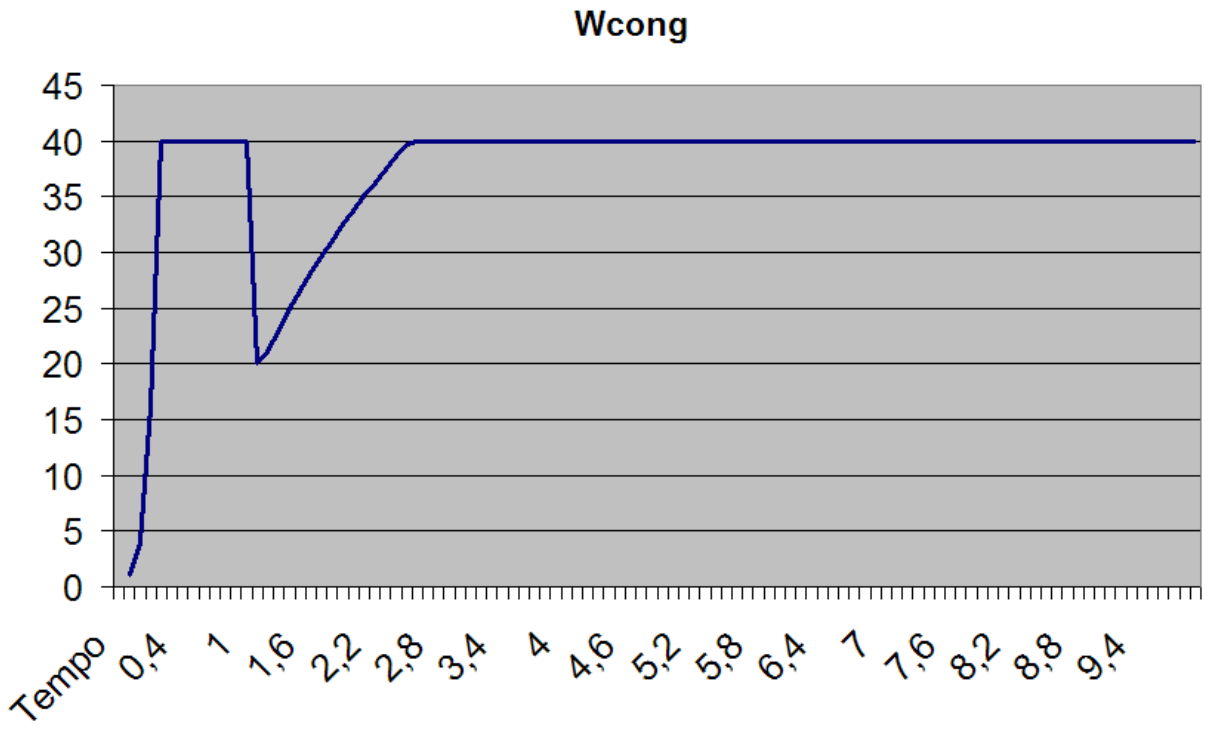


Fig. 29: Andamento della Wcong in SACK

Si noti come la finestra abbia all'inizio un andamento esponenziale, dovuto alla fase di Slow-Start, fino a che si raggiunge la massima apertura ammessa della finestra senza entrare in Congestion Avoidance; questo è causato dal fatto che all'inizio SACK impone una SS-thresh pari alla dimensione massima ammissibile della finestra per arrivare rapidamente a raggiungere un throughput elevato. Dopo che la trasmissione si è stabilizzata per alcuni istanti a velocità di regime avviene la perdita dei pacchetti: la Wcong viene dimezzata come previsto da Fast Recovery, riparte la trasmissione in regime di Congestion Avoidance in quanto la SS-thresh è stata dimezzata e vengono recuperati immediatamente gli errori grazie a Fast Recovery; la Wcong si aprirà con un andamento lineare fino a ritornare al valore massimo della finestra.

Analizziamo ora l'andamento della SS-thresh mostrato nel grafico in fig.30: si nota come all'inizio della connection essa venga settata al valore massimo ammesso della finestra, nel nostro caso 40 pacchetti, per avere una rapida accelerazione e raggiungere rapidamente la massima velocità ammessa dal canale. Essa rimane a questo valore fino a che avviene la notifica dell'errore: a questo punto, come pronosticato, essa viene dimezzata e posta a 20 pacchetti. Ecco il grafico:

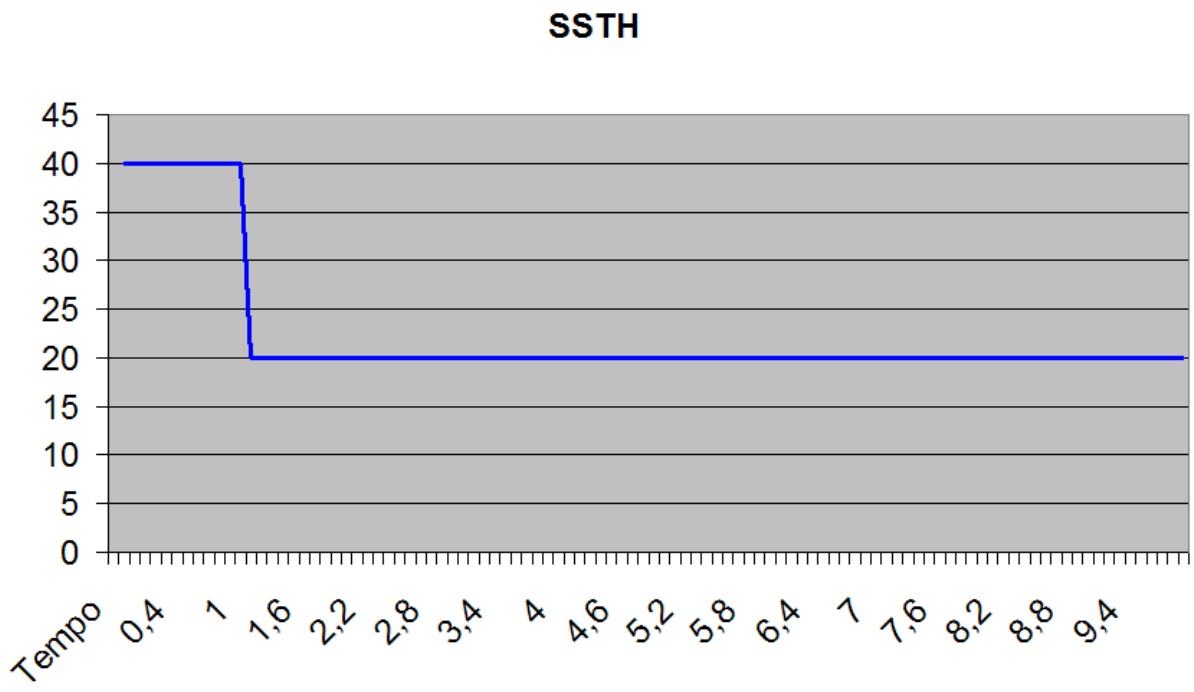


Fig. 30: Andamento della SS-thresh in SACK

L’ultimo fattore che rimane da analizzare è l’andamento della velocità sperimentata sul canale:

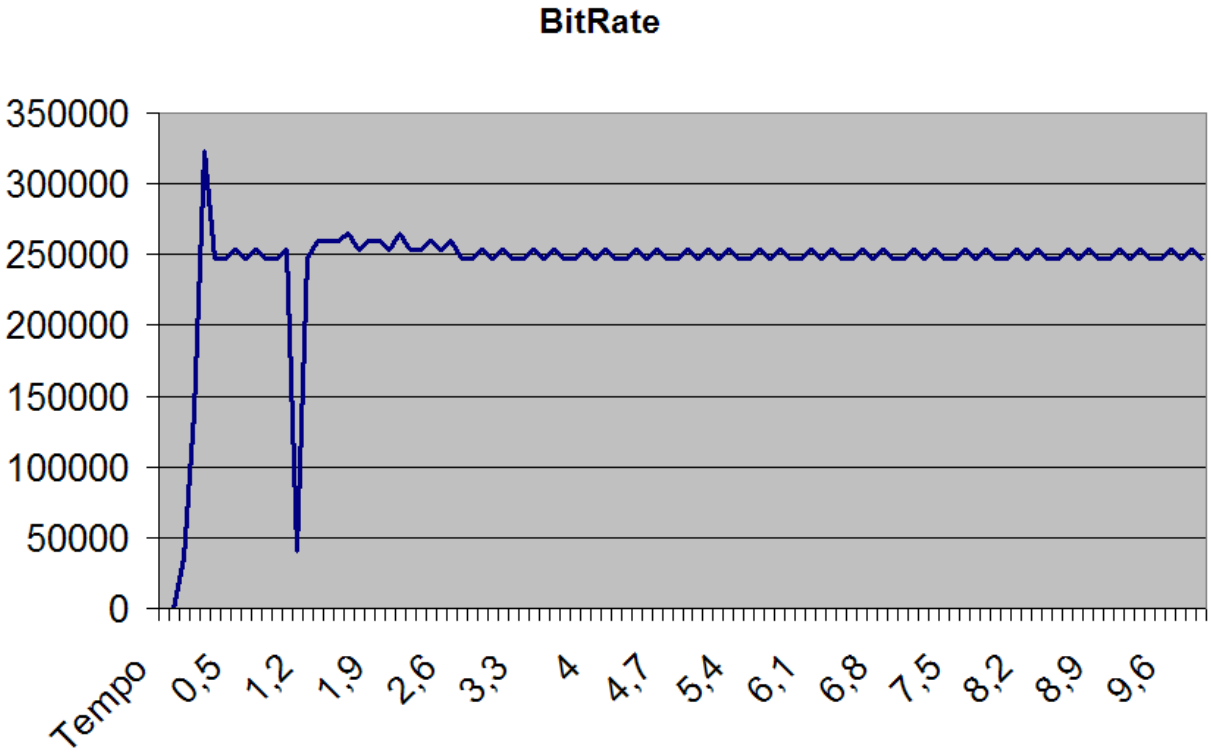


Fig. 31: Andamento della velocità di trasmissione in SACK

Si può notare la solita somiglianza con l'andamento della Wcong, infatti le due sono strettamente legate in quanto più si apre la finestra, tanto più la velocità aumenta; questo è dovuto al fatto che il mittente trasmette sempre più pacchetti prima di attendere l'autorizzazione a trasmetterne di nuovi, grazie al recapito dell'ACK relativo al primo pacchetto trasmesso della finestra. La velocità aumenterà fino a quando l'ACK relativo al primo pacchetto della finestra ritornerà prima che l'intera finestra sia stata trasmessa; si avrà quindi in questo caso una trasmissione continua. Nella prima parte del grafico si nota come la Wcong si apra in modo esponenziale e, dopo il solito picco dovuto alla granularità con la quale si calcola il bitrate, si stabilizzi al valore massimo di regime. Alla notifica degli errori il bitrate cala a causa del restringimento della Wcong, per poi risalire linearmente durante la fase di Congestion Avoidance fino a ristabilizzarsi al valore massimo come in precedenza.

In conclusione si può affermare che il TCP con SACK è efficace in presenza di più pacchetti persi per uno stesso RTT,e riesce a sopperire alle lacune di Tahoe e Reno in presenza di perdite multiple. Grazie a questa sua proprietà, esso può permettere sostanziali miglioramenti in connessioni stabilite su link con elevati tassi d'errore come nelle Wireless LAN oppure in connessioni satellitari, facilitando così lo sviluppo di queste tecnologie che oggi sempre più vanno diffondendosi.

3.4 Conclusioni sulla simulazione di TCP

Attraverso le simulazioni sopra citate si è voluto chiarire il funzionamento di TCP, quel protocollo che governa le connessioni in una rete di grandi dimensioni quale Internet, mostrandone i principali comportamenti delle sue varie implementazioni in presenza di una perdita di uno o più pacchetti dovuta alla congestione della rete. Grazie agli algoritmi propri di TCP è possibile recuperare i pacchetti persi e prevenire la congestione della rete, se non ancora intasata, o cercare di riportarla a livelli di congestione accettabili se già intasata.

Sono stati inoltre mostrati i limiti di Tahoe e Reno in presenza di errori multipli e i vantaggi che SACK ha portato in queste situazione; quest'ultima problematica è destinata inoltre a diventare sempre più oggetto di studio, grazie all'affermazione delle nuove tecnologie di comunicazione portatile come Wireless o tramite satellite. E' lecito infine affermare che l'aggiunta di SACK in TCP aprirà la strada ad ulteriori miglioramenti nella struttura di TCP, soprattutto in vista dell'imminente affermazione della versione 6 di IP, che causerà l'aumento della dimensione di host presenti in Internet e quindi la crescita del traffico sulla rete mondiale.

Capitolo 4

Simulazione di protocolli di Routing tramite NS2

4.1 Introduzione al routing

Abbiamo detto accennato in precedenza come una rete di calcolatori consista in un certo numero di host connessi tra di loro tramite un canale condiviso, dove le informazioni viaggiano tra un host mittente ed uno ricevente grazie alla connection TCP che supervisiona il corretto invio dei dati e il carico di traffico sulla rete. Esso però non si occupa di come sia possibile individuare univocamente un host tra tutti quelli della rete, locale o globale che sia ed indirizzare il traffico verso esso. Per questo TCP si appoggia sul protocollo di livello network IP, acronimo di Internet Protocol, il quale è il responsabile dell'indirizzamento dei vari host nella rete; detto protocollo è snello, non è confermato e non assicura la consegna dei messaggi in sequenza, ma riesce ad effettuare l'instradamento dei pacchetti attraverso un percorso calcolato tramite appositi algoritmi di routing e provvedo così alla consegna dei messaggi.
IP assegna ad ogni terminale connesso ad una rete un indirizzo univoco di lunghezza 32 bit; per semplicità tale indirizzo viene suddiviso in 4 byte e viene tradotto in decimale, per cui un indirizzo di un host nella rete presenta la seguente forma: **192.168.1.52**. Ogni byte può rappresentare valori tra 0 e 255. Un indirizzo IP può essere assegnato arbitrariamente in una sottorete privata solo se questa non è connessa con le altre reti, in caso contrario è necessario assegnare un indirizzo che non sia già stato usato per evitare errori di instradamento. L'indirizzo può essere suddiviso in due parti: la prima è la parte NETWORK, ovvero quella parte che descrive la rete a cui si fa riferimento, mentre la seconda è la parte HOST, che identifica univocamente un determinato calcolatore. La parte NETWORK ed HOST non sono di lunghezza fissa ma possono variare secondo delle classi di indirizzamento:

- Classe A: campo NETWORK lungo 7bit → max 128 reti, valori compresi tra 0 e 127
- Classe B: campo NETWORK lungo 14 bit→ max 16000 reti circa, valori tra 128 e 191
- Classe C: campo NETWORK lungo 21 bit→ max 2.000.000 reti circa, valori tra 192 e 223
- Classe D: indirizzamento multicast
- Classe E: riservato

Un esempio della struttura di un indirizzo di Classe C è fornito in figura 32.

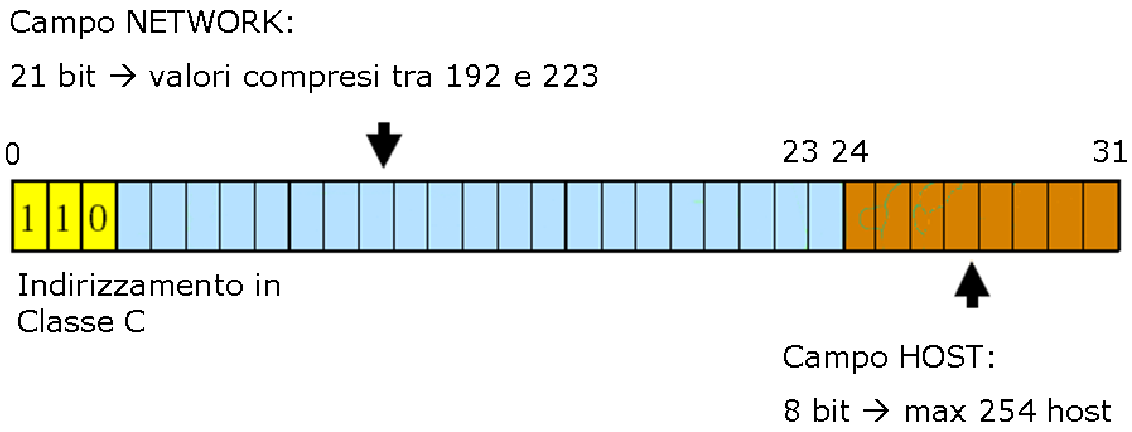


Fig. 32: schema di struttura di un indirizzo in Classe C

Gli indirizzi da assegnare agli host dovranno appartenere ad una delle classi sopra elencate; tuttavia non tutte le combinazioni di bit sono permesse poiché alcune sono riservate per scopi particolari (ad esempio 127.0.0.1, 255.255.255.255 o 0.0.0.0).

IP si occupa anche, come detto in precedenza, di trovare il percorso da far seguire ad un pacchetto dal mittente verso la destinazione in una rete, sia essa una LAN o Internet; le tecniche di routing variano molto a seconda della topologia della rete, della sua complessità e soprattutto dall'instabilità dei link che la compongono.

In una rete dove esiste un solo percorso possibile per arrivare ad un determinato host (reti *non magliate*) si utilizza un routing statico, ovvero gli instradamenti vengono decisi inizialmente dall'amministratore di rete e non più modificati; ovviamente questo tipo di routing è utilizzabile solo all'interno di reti con un numero di host limitato. In realtà più complesse, come Internet, esistono molte reti collegate tra di loro tramite appositi hardware chiamati router: essi hanno la funzione di memorizzare i percorsi che permettono ad un pacchetto di arrivare a destinazione ed instradare i pacchetti stessi. Ogni router possiede una tabella, detta *Routing Table*, nella quale sono memorizzate i percorsi possibili. Va detto che il percorso completo del pacchetto dal mittente a destinazione in questi casi non è mai completamente noto, infatti l'instradamento è del tipo distribuito, ovvero nelle routing table viene memorizzato solo l'indirizzo del nodo successivo che porta al destinatario. Una tabella di routing memorizzata in un generico router è composta sostanzialmente da due campi: nel primo viene indicata la destinazione mentre il secondo contiene l'indirizzo del prossimo nodo a cui inoltrare il pacchetto per arrivare a destinazione.

Destination	Next Hop
193.0.1.0	193.0.0.3
145.0.0.0	193.0.0.3

Fig. 33: esempio di routing table

Ogni router ed ogni host di una rete avrà una propria tabella di instradamento, di dimensioni tanto maggiori quanto maggiore sarà la complessità della rete presa in esame. In realtà vi è anche un terzo campo nelle tabelle, che rappresenta il costo di ogni Next Hop, in base al quale si sceglie il percorso ottimale: un percorso con costo basso che richiede più passaggi tra router verrà preferito ad uno con costo più alto anche se richiede meno passaggi.
In figura 34 è mostrato un esempio di una rete IPv4 con le relative tabelle di routing:

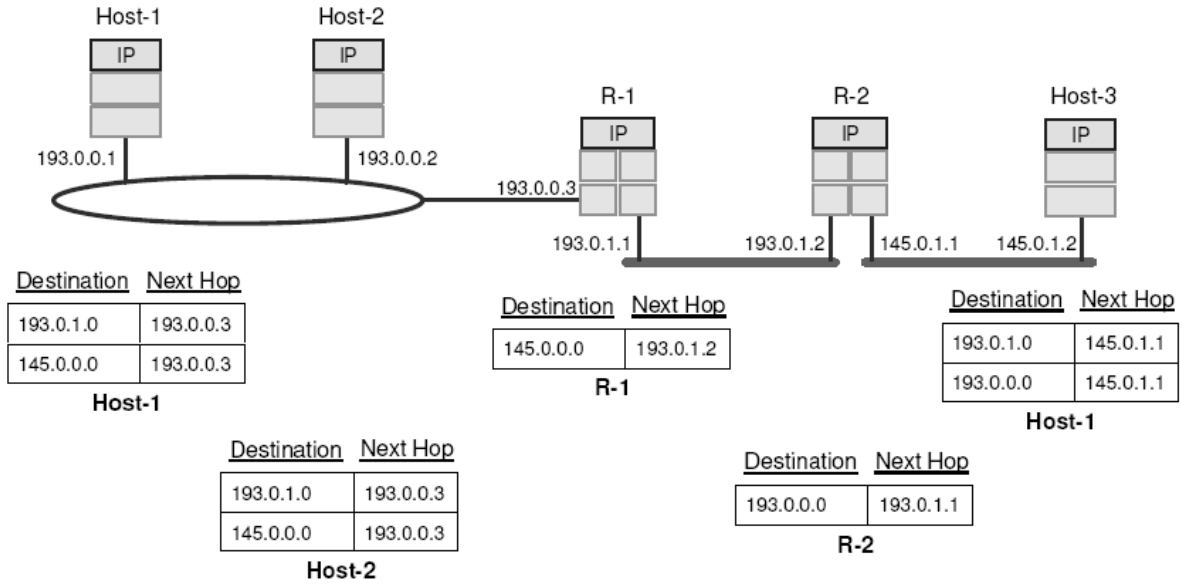


Fig. 34: esempio di routing table su un'ipotetica rete

Supponiamo ad esempio di dover spedire un messaggio da Host-1 (h1) a Host-3 (h3): h1 legge l'indirizzo di h3 e nota che non è un nodo appartenente alla stessa rete, quindi lo instrada verso l'indirizzo 193.0.0.3, ovvero verso il router R-1. Quest'ultimo legge in un apposito campo nella header del pacchetto che deve essere recapitato a h3 e lo inoltra a R-2 il quale, una volta identificato il destinatario del messaggio, capisce che è adiacente (ovvero connesso sullo stesso *link*) e glielo inoltra direttamente.

4.2 Routing statico

Il primo algoritmo di routing che verrà analizzato è il Routing Statico. Esso è l'algoritmo più semplice da implementare in quanto non necessita di particolari calcoli, ma viene bensì effettuato manualmente dall'amministratore della rete e non cambia fino a che lo stesso amministratore non decide di farlo. Questo permette che in rete non venga immesso nessun traffico di servizio, a vantaggio delle prestazioni generali. Tuttavia le tabelle compilate in questo modo non permettono

di reagire al cambiamento della topologia di rete: se un link cade non può essere trovato un percorso alternativo. Ecco il motivo per cui questo metodo è poco utilizzato nelle reti di grande dimensione come Internet, dove la stabilità dei link è tutt'altro che garantita. Nel simulatore l'operazione di compilazione delle tabelle di routing non può essere fatta dall'utente come previsto, ma viene calcolata in forma centralizzata secondo l'algoritmo di Dijkstra all'inizio della simulazione.
Lo scenario adottato per mostrare il comportamento del Routing Statico consiste in una rete formata da otto nodi dei quali uno trasmette e uno riceve, le caratteristiche dei link sono comuni sia per la velocità che per il ritardo ed i costi dei link sono stati dati secondo lo schema in figura 35:

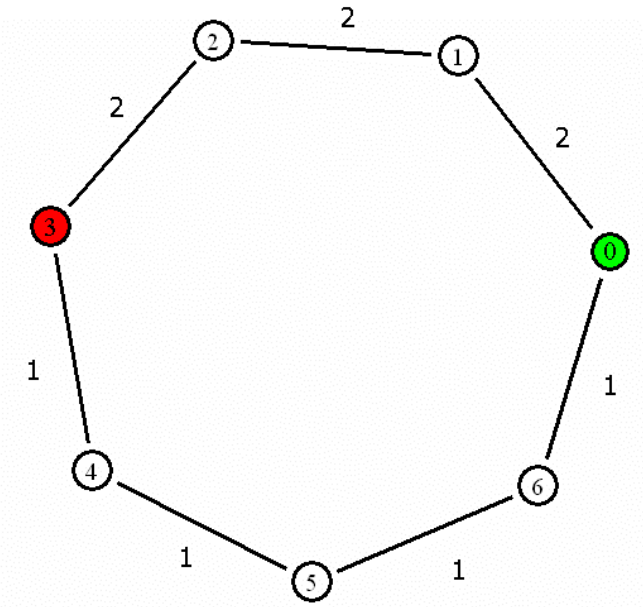


Fig. 35: costo assegnato ai vari link nello scenario di simulazione

Il traffico verrà generato dal nodo 0 e trasmesso verso il nodo 3, seguendo il percorso calcolato all'inizio della simulazione da Ns. Analizzando lo scenario si nota che il percorso più breve è quello che passa per i nodi 0 – 1 – 2 – 3 ed ha costo 6; l'altro percorso possibile è quello 0 – 6 – 5 – 4 che ha costo 4: il traffico dovrebbe quindi transitare attraverso il secondo percorso.
Ecco il codice necessario per realizzare lo scenario di rete con le caratteristiche sopra elencate:

```
...
#Creazione dei nodi tramite un vettore
for {set i 0} {$i < 7} {incr i} {
        set n($i) [$ns node]
```

```
}
#Creazione dei link
for {set i 0} {$i < 7} {incr i} {
  $ns duplex-link $n($i) $n([expr ($i+1)%7]) 1Mb 10ms DropTail
}
#Determinazione costo dei link (default 1)
$ns cost $n(0) $n(1) 2
$ns cost $n(1) $n(2) 2
$ns cost $n(2) $n(3) 2
...
```

Una volta creata la topologia della rete, creiamo l'agente e l'applicazione adatta per generare il traffico: si è deciso in questo caso di utilizzare di un agente UDP ed un'applicazione di tipo CBR in quanto non interessa effettuare un controllo di flusso tramite TCP, ma bensì mostrare come reagisca il protocollo di routing al variare delle condizioni dei link. Il codice relativo ad agente e applicazione è il seguente:

```
...
#Creazione agente UDP sul nodo n0
set udp0 [new Agent/UDP]
$ns attach-agent $n(0) $udp0
#Applicazione di una sorgente CBR ad UDP
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0
#Creazione dell'agente ricevente sul nodo n3
set null0 [new Agent/Null]
$ns attach-agent $n(3) $null0
$ns connect $udp0 $null0
```

```
...
```

Arrivati alla creazione del traffico è necessario ora far variare la topologia della rete per verificare il comportamento dell'algoritmo in esame. La variazione consiste nella caduta di un link utilizzato per la trasmissione dei messaggi, in modo da simulare ad esempio un guasto alla linea; interromperemo quindi il link tra i nodi 5 e 6 all'istante 0.6:

```
...
#Disattivazione del link
$ns rtmodel-at 0.6 down $n(5) $n(6)
#Riattivazione del link
$ns rtmodel-at 1.0 up $n(5) $n(6)
...
```

Inoltre vogliamo monitorare la tabella di routing e le informazioni sulla distanza tra i vari nodi all'istante di partenza e dopo che il link è caduto, per verificare se vengono effettuate modifiche; tali tabelle sono visualizzabili a schermo durante l'esecuzione della simulazione. E' necessario creare una procedura che venga invocata agli istanti voluti, ed essa è realizzata tramite il seguente codice:

```
...
#Creazione procedura rtable
proc rtable {} {
     set ns [Simulator instance]
     puts "Tabella di routing all'istante [$ns now]"
     $ns dump-routelogic-nh
     puts "Informazioni sulla distanza"
     $ns dump-routelogic-distance}
...
```

Dato che il link è stato fatto cadere all'istante 0.6 la procedura può essere eseguita ad esempio agli istanti 0.2 e 0.8; il codice necessario sarà:

```
...
$ns at 0.2 "rtable"
```

```
$ns at 0.8 "rtable"
...
```

Siamo ora pronti all'esecuzione della simulazione, visto che abbiamo definito ogni particolare dello scenario; il codice completo è inserito nella sezione 1 dell'appendice B.

Analizziamo il comportamento teorico che dovrebbe presentare la simulazione: all'istante di avvio della simulazione la tabella di routing sarà tale da fare passare il traffico attraverso il percorso 0 – 6 – 5 – 4 – 3 e alla caduta del link non si noteranno cambiamenti nel percorso dei pacchetti e verranno persi tutti quelli trasmessi; appena riattivato il link il traffico ritornerà ad arrivare al destinatario. Il mittente continuerà a trasmettere ignaro della caduta del link in quanto UDP non essendo confermato non effettua controlli sull'effettivo recapito dei messaggi, cosa che non succederebbe con TCP, che allo scadere del primo time-out bloccherebbe la trasmissione e trasmetterebbe un pacchetto solo dopo lo scadere di ogni time-out.

Passiamo ora ad analizzare la routing table compilata dal simulatore all'istante di partenza del traffico:

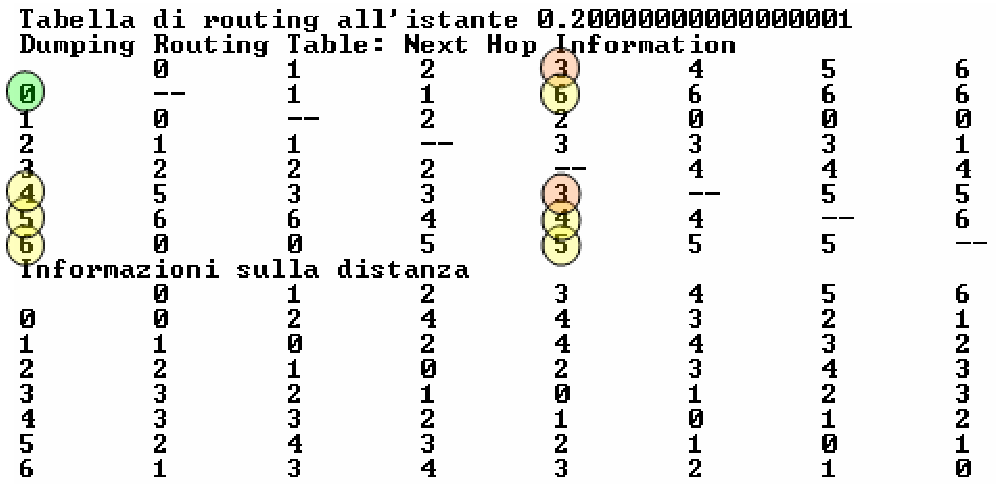


Fig. 36: tabelle di routing e di distanza tra i vari nodi all'istante 0.2

Si può notare dalla tabella di routing che per arrivare ad n3 partendo da n0 si debba passare per il nodo 6, poi per il 5, per il 4 ed infine si arriva a destinazione. Dalla seconda tabella si nota come la distanza tra n0 e n3 sia effettivamente pari a 4 e non a 6. Le tabelle sono uguali anche dopo la caduta del link: ciò significa che l'algoritmo di routing statico effettivamente non reagisce al cambiamento delle condizioni della rete.

```
Tabella di routing all'istante 0.8000000000000004
Dumping Routing Table: Next Hop Information
        0       1       2       3       4       5       6
0       --      1       1       6       6       6       6
1       0       --      2       2       0       0       0
2       1       1       --      3       3       3       1
3       2       2       2       --      4       4       4
4       5       3       3       3       --      5       5
5       6       6       4       4       4       --      6
6       0       0       5       5       5       5       --
Informazioni sulla distanza
        0       1       2       3       4       5       6
0       0       2       4       4       3       2       1
1       1       0       2       4       4       3       2
2       2       1       0       2       3       4       3
3       3       2       1       0       1       2       3
4       3       3       2       1       0       1       2
5       2       4       3       2       1       0       1
6       1       3       4       3       2       1       0
```

Fig. 37: tabelle di routing e di distanza tra i vari nodi all'istante 0.8

Passiamo ora all'analisi dell'animazione. Alla partenza del traffico si nota come i pacchetti siano instradati verso n3 passando per 6 – 5 – 4 come da noi atteso:

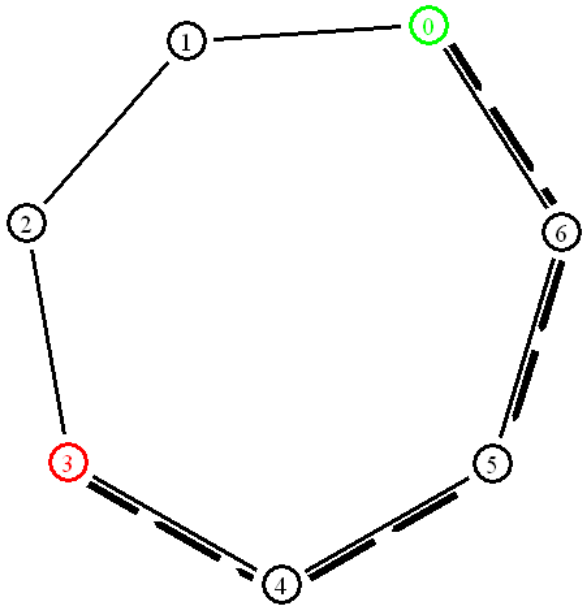


Fig. 38: Direzione del traffico all'istante $0.2 \leq t < 0.6$

All'istante 0.6 il link 5 – 4 viene a mancare, i pacchetti in transito sul link vengono persi e il mittente continua a trasmettere nella stessa direzione ignaro dell'accaduto:

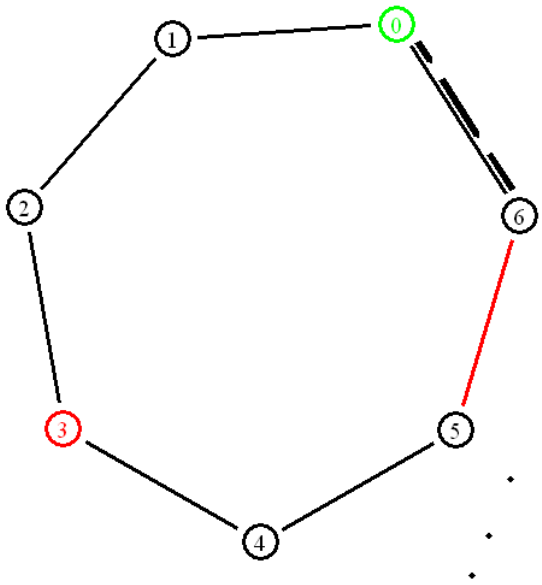


Fig. 39: Direzione del traffico all'istante $0.6 \leq t < 1.0$.

Alla riattivazione del link caduto il traffico riprenderà ad arrivare al destinatario come in precedenza, fino alla fine della simulazione.

L'algoritmo di routing statico presenta quindi una capacità nulla di adattarsi alle variazioni sulla rete; questo è il principale motivo dello scarso utilizzo che ha questo modello, limitato a reti di piccole dimensioni dove la ricompilazione delle tabelle non diventa troppo dispendiosa in termini di tempo.

4.3 Routing dinamico Distance Vector

Il principio fondamentale di un protocollo Distance Vector (DV) è che ogni router invia ai router adiacenti la propria tabella di routine; ogni router aggiorna a sua volta la propria tabella in base ai messaggi ricevuti e poi la propaga iterativamente agli adiacenti. L'idea è semplice ed efficace, in quanto le tabelle calcolano i percorsi minimi di tutta la rete, ma ha bisogno di alcuni accorgenti:

- in caso di presenza di più valori nella tabella di routing con la stessa destinazione finale ma diverse destinazioni successive (percorsi alternativi), viene mantenuto quello con il costo inferiore;
- non vengono propagate all'indietro informazioni ricevute in precedenza dal router stesso a cui le si propaga (regola dello *split horizon*)

Le informazioni di aggiornamento vengono inviate da ciascun router quando viene notata una variazione della propria tabella di routing o a intervalli regolari, o al cambiamento della struttura della rete; perciò DV riesce a reagire alla caduta di un link trasmettendo informazioni di

aggiornamento che portano alla risoluzione di un nuovo percorso possibile. I protocolli che implementano questo algoritmo si preoccupano di sfasare casualmente i messaggi nel tempo per impedire intasamenti di rete. Inoltre un router appena avviato in una rete con altri router presenti, molto presto riceve informazioni sufficienti ad avere una tabella di routing aggiornata che gli permette di conoscere la composizione della rete. Ogni stazione non solo invia aggiornamenti ai vicini ma si aspetta anche di riceverli; se non vengono ricevuti aggiornamenti consecutivamente per un certo numero di volte (tipicamente sei), il router che non invia viene dichiarato irraggiungibile. Va detto che questo tempo è utile poiché impedisce i falsi allarmi dovuti ad esempio a un momento di stallo di un router, ma dall'altro introduce un ritardo considerevole nella risposta del sistema ai cambiamenti della struttura di rete. Una volta che un router stabilisce che un altro dispositivo è irraggiungibile invia subito una notifica di aggiornamento ai vicini (*triggered update*) e tutti i router coinvolti si scambiano messaggi di update, fino a convergenza avvenuta.

Per la simulazione dell'algoritmo DV utilizzeremo lo scenario precedente, salvo apportare alcune modifiche necessarie per l'analisi. La prima modifica riguarda la topologia di rete: infatti viene inserito un ulteriore link tra i nodi n0 ed n4, viene aggiunto un ulteriore mittente sul nodo n1 ed infine vengono cambiati i costi dei link secondo lo schema riportato in figura 40:

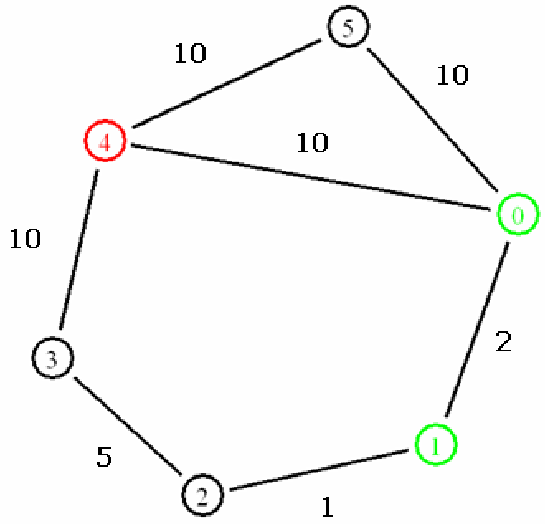


Fig. 40: Rappresentazione del costo dei link

Le sorgenti di traffico sono rimaste del tipo CBR con un agente UDP, così come è rimasta uguale la procedura per la visualizzazione della tabella di routing e della tabella delle distanze. Il codice completo può essere trovato nella sezione 2 dell'appendice B.

Passiamo ora all'analisi della simulazione osservando per prima cosa la tabella di routing all'inizio della simulazione, ovvero all'istante 0.5:

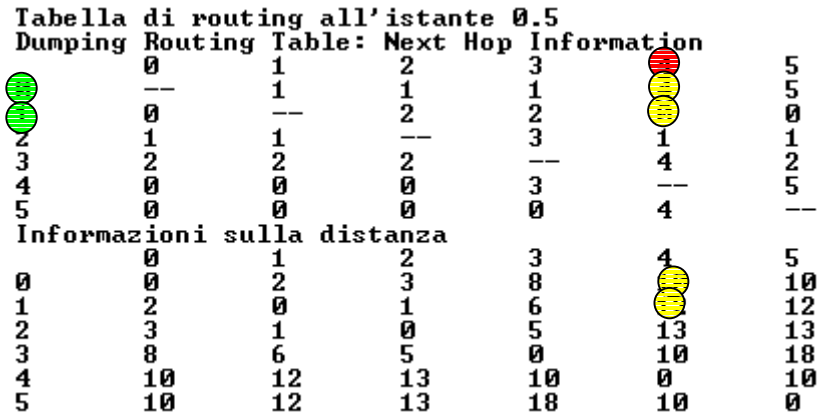


Fig. 41: Routing Table ed informazioni sulla distanza all'istante 0.5

Si può notare come giustamente i messaggi inviati da n0 raggiungano direttamente n4 mentre quelli di n1 passino prima per n0; le distanze sono rispettivamente 10 e 12.

Il traffico segue questa via fino a che il link cade; dopo questo istante le routing table subiscono una variazione:

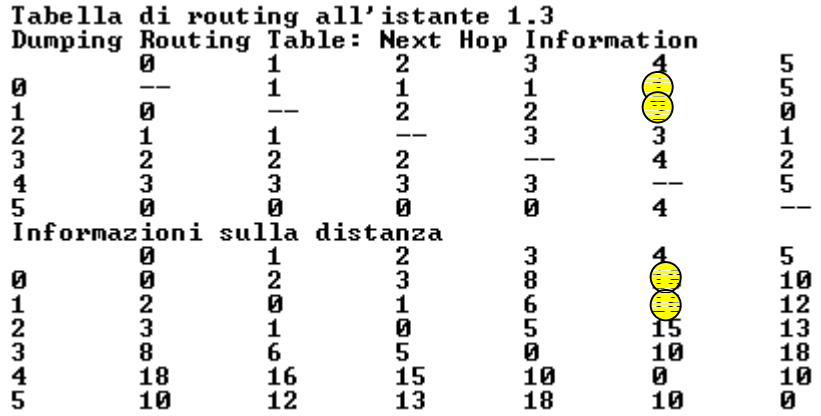


Fig. 42: Routing Table ed informazioni sulla distanza all'istante 1.3

Il traffico cambia direzione e passa per il percorso 1 – 2 – 3, anziché passare per il nodo 5, poiché il costo è inferiore anche se si ha un numero maggiori di hop da compiere. Al ripristino del link il traffico riprenderà infine a transitare lungo il percorso originale dato che verrà ricostruita la routing table originale.

Facendo partire l'animazione si nota che immediatamente vengono scambiati pacchetti DV tra tutti i nodi; tali dati consentono ai nodi di calcolare i percorsi minimi per i traffici. All'istante 0.5 parte la prima sorgente CBR dal nodo n0 e trasmette direttamente a n4 come previsto, così avviene anche

per n1 che trasmette ad n4 passando per n0. Si nota che ogni tanto vengono scambiati informazioni DV come aggiornamento.

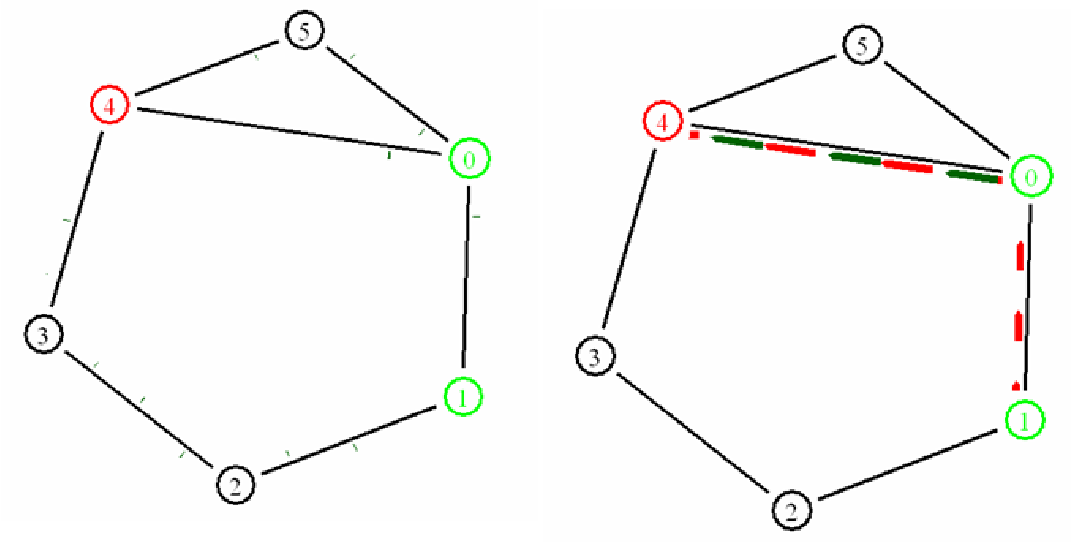


Fig. 43: a sinistra si nota lo scambio di informazioni DV, mentre a destra si nota il percorso dei pacchetti

Alla caduta del link si nota che il traffico da n1 ad n4 e da n0 a n4 non passa per n2 ed n3, percorso che ha il costo minore, ma il traffico viene instradando verso n5 anche se il costo è 20 e non 18 come ottenibile nell'altra direzione; questo accade in quanto la caduta del link provoca un reciproco scambio di informazioni DV che aggiorna tutti i nodi dell'accaduto. Si nota che l'informazione parte dai nodi n0 e n4, ovvero quelli direttamente connessi al link in questione, e viene propagata agli altri: il primo nodo utilizzabile per l'instradamento diventa n5 in quanto è il primo che riceve i DV sufficienti a creare una routing table in grado di assicurare un percorso al traffico. La routing table transitoria sarà quella mostrata in figura 44; in figura 45 viene raffigurato un istante nel transitorio in cui i traffici vengono instradati verso n5.

La routing table transitoria però viene presto aggiornata dall'arrivo dei pacchetti DV, che nel frattempo si sono propagati anche sul ramo n2 – n3, e all'istante 1.14125 le routing table vengono aggiornate: il traffico viene così deviato sul percorso meno costoso come mostrato in figura 47; in figura 46 si può osservare la nuova tabella di routing.

```
Dumping Routing Table: Next Hop Information
        0       1       2       3       4       5
0       --      1       1       1       1       5
1       0       --      2       2       2       0
2       1       1       --      3       1       1
3       2       2       2       --      4       2
4       3       3       3       3       --      5
5       0       0       0       0       4       --
Informazioni sulla distanza
        0       1       2       3       4       5
0       0       2       3       8       8       10
1       2       0       1       6       6       12
2       3       1       0       5       23      13
3       8       6       5       0       10      18
4       18      16      15      10      0       10
5       10      12      13      18      10      0
```

Fig. 44: tabella di routing ed informazioni sulla distanza nel transitorio

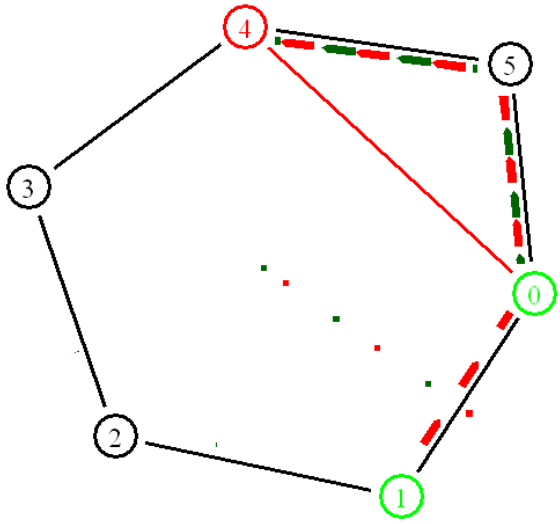


Fig. 45: si nota come n0 trasmetta passando per n5 e non per n1

```
Tabella di routing all'istante 1.5
Dumping Routing Table: Next Hop Information
        0       1       2       3       4       5
0       --      1       1       1       1       5
1       0       --      2       2       2       0
2       1       1       --      3       3       1
3       2       2       2       --      4       2
4       3       3       3       3       --      5
5       0       0       0       0       4       --
Informazioni sulla distanza
        0       1       2       3       4       5
0       0       2       3       8       8       10
1       2       0       1       6       6       12
2       3       1       0       5       15      13
3       8       6       5       0       10      18
4       18      16      15      10      0       10
5       10      12      13      18      10      0
```

Fig. 46: routing table dopo il transitorio

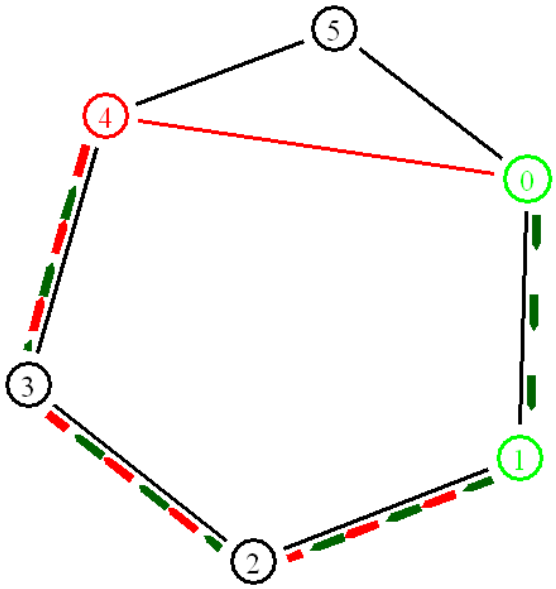


Fig. 47: direzione del traffico al termine del transitorio.

All'istante 2.0 verrà ripristinato il link inattivo e dopo lo scambio dei DV gli instradamenti ritorneranno ad essere quelli che erano stati calcolati all'inizio della simulazione ed il traffico passerà da n0 direttamente a n4 ed n1 inoltrerà il proprio traffico a n0.

Conclusioni sulla simulazione dei protocolli di routing

Le simulazioni sopra riportate vogliono essere un esempio di come funzionino nella realtà i meccanismi che governano il livello Network delle reti mondiali e, in particolare, spiegare il perché si preferisca di norma usare il protocollo Distance Vector anziché quello statico. Questa scelta, come già visto in precedenza, è dovuto al fatto che il routing statico è insensibile alle variazioni di topologia, come la caduta di un link; l'impossibilità di compilare manualmente le tabelle di routing di tutti i nodi che compongono Internet e l'elevato grado di instabilità dei link stessi gioca nettamente a favore di un protocollo dinamico quale Distance Vector, che riesce a costruirsi e ad aggiornare le tabelle in maniera del tutto automatica. In questo caso la simulazione è stata d'aiuto per verificare l'esatezza delle tabelle calcolate dall'algoritmo e per verificare la rapidità con la quale DV riesca a dirottare il traffico su un percorso alternativo.

Conclusioni

In questo elaborato è stato analizzato il simulatore di reti telematiche Network Simulator 2 con il quale sono state eseguite alcune simulazioni riguardanti il protocollo di trasporto TCP in varie implementazioni, e di alcuni algoritmi di routing. Va detto che NS2 è un simulatore ad eventi discreti molto completo, abbastanza fedele alla realtà in quanto solo in pochi casi ha presentato comportamenti anomali, e relativamente semplice da utilizzare grazie all'utilizzo del linguaggio Otcl che rende meccanica la configurazione dei vari componenti da simulare.

Durante la simulazione del protocollo TCP non sono state incontrate grosse difficoltà, se non quelle dovute alla configurazione delle varie caratteristiche proprie di TCP come ad esempio il settaggio della dimensione iniziale della finestra, o alla scelta del loss model che meglio si adattava ai nostri esempi, in modo da ottenere risultati facilmente analizzabili.

Nella sezione riguardante il routing il simulatore è stato molto utile per chiarire il comportamento di Distance Vector, apparentemente anomalo, in risposta a variazioni della topologia di rete; grazie all'analisi dei dati forniti dal simulatore si è infatti potuto osservare l'esistenza di un breve transitorio, durante il quale il funzionamento della rete differisce da quello a regime teoricamente previsto.

In conclusione si può affermare che in generale le simulazioni sono un valido strumento di supporto allo studio e allo sviluppo di nuove tecnologie o per il miglioramento di quelle esistenti, in quanto riescono a diminuire i costi e i tempi dovuti alla ricerca permettendo nel contempo una analisi più approfondita del caso preso in esame. Non bisogna però dimenticare che i simulatori modellano sempre una realtà semplificata, nella quale alcuni fattori non vengono volutamente tenuti in considerazione; è bene quindi verificare che la presenza di questi aspetti nel mondo reale sia ai fini pratici effettivamente ininfluente sul fenomeno osservato prima di considerare validi i risultati ottenuti con il calcolatore.

Ringrazio tantissimo la mia famiglia che mi ha sostenuto
in tutto questo tempo e mi ha permesso di arrivare a questo traguardo.
Un sentito ringraziamento va ad Emanuele Goldoni, che mi ha seguito in
questa tesi con grande pazienza.
Un altro ringraziamento di dovere va a Matteo Lanati, il cui aiuto
è stato fondamentale per la riuscita delle simulazioni in NS2.
Ringrazio inoltre il prof. Giuseppe Federico Rossi per l'attenzione
e la disponibilità che ha dimostrato durante la realizzazione dell'elaborato.
Vorrei ringraziare infine Ago, Ama, il Bado, il Becca, Enri, il Gabo, la Marty,
il Mala, il Siro, il Trento e la Vale, Alex, il Fuci, Mattia, Michele e Stefano,
tutti i compagni assieme ai quali ho trascorso questi meravigliosi anni di università
e tutte le persone che ho la fortuna di avere come amiche
per essermi state sempre vicine.

APPENDICE A

Sezione 1: TCP Tahoe

```
# Creazione di un nuovo Simulator object
set ns [new Simulator]
# Apertura del file per NAM
set namFile [open tcptahoe.nam w]
$ns namtrace-all $namFile
# Variabili globali
set NumBytePrec 0
# File di trace
set datafile [open tf.tr w]
set cwnfile [open cwnR.dat w]
set f1  [open rate w]
$ns trace-all $datafile
#Definizione topologia
set Node0 [$ns node]
set Node1 [$ns node]
set Node2 [$ns node]
$ns duplex-link $Node0 $Node1 2Mb 10ms DropTail
$ns duplex-link $Node1 $Node2 2Mb 10ms DropTail
$ns queue-limit $Node1 $Node2 160000
#Inserimento di errori deterministici sul link
set loss [new ErrorModel/List]
$loss unit pkt
$loss droplist 400
#Definizione della sorgente Tahoe
set TCP0 [new Agent/TCP/FullTcp/Tahoe]
$ns attach-agent $Node0 $TCP0
set sink0 [new Agent/TCP/FullTcp/Tahoe]
$ns attach-agent $Node2 $sink0
$sink0 listen
$ns connect $TCP0 $sink0
set ftp01 [new Application/FTP]
$ftp01 attach-agent $TCP0
#Configurazione parametri del TCP
$TCP0 set window_ 40
#Procedure per la registrazione dei dati di interesse
# sonde per monitorare i rate delle sorgenti TCP
```

```
set mon [$ns  monitor-queue $Node0 $Node1 [$ns get-ns-traceall]]
#Applicazione del lossmodel su un link
$ns link-lossmodel $loss $Node1 $Node2
# procedura per calcolare il bitrate del canale
proc rate {step} {
      global f1 mon ns
      global NumBytePrec
        set now [$ns now]
        set Byte [$mon set barrivals_]
        set DeltaByte [expr $Byte-$NumBytePrec]
        puts $f1 "$now [expr $DeltaByte/($step)]"
        set NumBytePrec $Byte
      $ns at [expr $now+$step] "rate $step"
}
#Procedura per monitorare la Wcong e la SS-thresh
proc cwnd { step } {
      global ns TCP0 cwnfile
      set now [$ns now]
      set cwn [$TCP0 set cwnd_]
      set ssth [$TCP0 set ssthresh_]
      puts $cwnfile "$now $cwn $ssth"
      $ns at [expr $now+$step] "cwnd $step"
}
# Definizione della procedura finish
proc finish {} {
      global ns namFile f1
      global cwnfile datafile
      $ns flush-trace
      close $namFile
      close $datafile
      close $cwnfile
      exit 0
}
#Eventi
$ns at 0.0   "rate 0.1"
$ns at 0.0 "$ftp01 start"
$ns at 0.0 "cwnd 0.1"
$ns at 10.0 "$ftp01 stop"
$ns at 10.0 "finish"
$ns run
```

Sezione 2: TCP Reno

```
# Creazione di un nuovo Simulator object
set ns [new Simulator]
# Apertura del file per NAM
set namFile [open tcpreno.nam w]
$ns namtrace-all $namFile
# Variabili globali
set NumBytePrec 0
# File di trace
set datafile [open tf.tr w]
set cwnfile [open cwnR.dat w]
set f1  [open rate w]
$ns trace-all $datafile
#Definizione topologia
set Node0 [$ns node]
set Node1 [$ns node]
set Node2 [$ns node]
$ns duplex-link $Node0 $Node1 2Mb 10ms DropTail
$ns duplex-link $Node1 $Node2 2Mb 10ms DropTail
$ns queue-limit $Node1 $Node2 160000
#Inserimento di errori deterministici sul link
set loss [new ErrorModel/List]
$loss unit pkt
$loss droplist 400
#Definizione della sorgente Reno
set TCP0 [new Agent/TCP/FullTcp]
$ns attach-agent $Node0 $TCP0
set sink0 [new Agent/TCP/FullTcp]
$ns attach-agent $Node2 $sink0
$sink0 listen
$ns connect $TCP0 $sink0
set ftp01 [new Application/FTP]
$ftp01 attach-agent $TCP0
#Configurazione parametri del TCP
$TCP0 set window_ 40
#Procedure per la registrazione dei dati di interesse
# sonde per monitorare i rate delle sorgenti TCP
set mon [$ns  monitor-queue $Node0 $Node1 [$ns get-ns-traceall]]
#Applicazione del lossmodel su un link
```

```
$ns link-lossmodel $loss $Node1 $Node2
# procedura per calcolare il bitrate del canale
proc rate {step} {
      global f1 mon ns
      global NumBytePrec
        set now [$ns now]
        set Byte [$mon set barrivals_]
        set DeltaByte [expr $Byte-$NumBytePrec]
        puts $f1 "$now [expr $DeltaByte/($step)]"
        set NumBytePrec $Byte
      $ns at [expr $now+$step] "rate $step"
}
#Procedura per monitorare la Wcong
proc cwnd { step } {
      global ns TCP0 cwnfile
      set now [$ns now]
      set cwn [$TCP0 set cwnd_]
      set ssth [$TCP0 set ssthresh_]
      puts $cwnfile "$now $cwn $ssth"
      $ns at [expr $now+$step] "cwnd $step"
}
# Definizione della procedura finish
proc finish {} {
      global ns namFile f1
      global cwnfile datafile
      $ns flush-trace
      close $namFile
      close $datafile
      close $cwnfile
      exit 0
}
#Eventi
$ns at 0.0   "rate 0.1"
$ns at 0.0 "$ftp01 start"
$ns at 0.0 "cwnd 0.1"
$ns at 10.0 "$ftp01 stop"
$ns at 10.0 "finish"

$ns run
```

Sezione 3: TCP Selective Acknowledge (SACK)

```
# Creazione di un nuovo Simulator object
set ns [new Simulator]
# Apertura del file per NAM
set namFile [open tcptahoe.nam w]
$ns namtrace-all $namFile
# Variabili globali
set NumBytePrec 0
# File di trace
set datafile [open tf.tr w]
set cwnfile [open cwnR.dat w]
set f1  [open rate w]
$ns trace-all $datafile
#Definizione topologia
set Node0 [$ns node]
set Node1 [$ns node]
set Node2 [$ns node]
$ns duplex-link $Node0 $Node1 2Mb 10ms DropTail
$ns duplex-link $Node1 $Node2 2Mb 10ms DropTail
$ns queue-limit $Node1 $Node2 160000
#Inserimento di errori deterministici sul link
set loss [new ErrorModel/List]
$loss unit pkt
$loss droplist 400 401
#Definizione della sorgente Sack
set TCP0 [new Agent/TCP/FullTcp/Sack]
$ns attach-agent $Node0 $TCP0
set sink0 [new Agent/TCP/FullTcp/Sack]
$ns attach-agent $Node2 $sink0
$sink0 listen
$ns connect $TCP0 $sink0
set ftp01 [new Application/FTP]
$ftp01 attach-agent $TCP0
#Configurazione parametri del TCP
$TCP0 set window_ 40
#Procedure per la registrazione dei dati di interesse
# sonde per monitorare i rate delle sorgenti TCP
set mon [$ns  monitor-queue $Node0 $Node1 [$ns get-ns-traceall]]
#Applicazione del lossmodel su un link
```

```
$ns link-lossmodel $loss $Node1 $Node2
# procedura per calcolare il bitrate del canale
proc rate {step} {
      global f1 mon ns
      global NumBytePrec
      set now [$ns now]
      set Byte [$mon set barrivals_]
      set DeltaByte [expr $Byte-$NumBytePrec]
      puts $f1 "$now [expr $DeltaByte/($step)]"
        set NumBytePrec $Byte
      $ns at [expr $now+$step] "rate $step"
}
#Procedura per monitorare la Wcong
proc cwnd { step } {
      global ns TCP0 cwnfile
      set now [$ns now]
      set cwn [$TCP0 set cwnd_]
      set ssth [$TCP0 set ssthresh_]
      puts $cwnfile "$now $cwn $ssth"
      $ns at [expr $now+$step] "cwnd $step"
}
# Definizione della procedura finish
proc finish {} {
      global ns namFile f1
      global cwnfile datafile
      $ns flush-trace
      close $namFile
      close $datafile
      close $cwnfile
      exit 0
}

#Eventi
$ns at 0.0   "rate 0.1"
$ns at 0.0 "$ftp01 start"
$ns at 0.0 "cwnd 0.1"
$ns at 10.0 "$ftp01 stop"
$ns at 10.0 "finish"

$ns run
```

APPENDICE B

Sezione 1: Routing Statico

```
# Creazione di un nuovo Simulator object
set ns [new Simulator]
# Apertura del file per il NAM
set tf [open RoutingSt.tr w]
set nf [open RoutingSt.nam w]
$ns trace-all $tf
$ns namtrace-all $nf
# Definizione della procedura finish
proc finish {} {
        global ns nf tf
        $ns flush-trace
      close $nf
      close $tf
        exit 0
}
#Definizione topologia
for {set i 0} {$i < 7} {incr i} {
        set n($i) [$ns node]
}
$n(0) color "green"
$n(3) color "red"
for {set i 0} {$i < 7} {incr i} {
        $ns duplex-link $n($i) $n([expr ($i+1)%7]) 1Mb 10ms DropTail
}

#Costo Link (default 1)

$ns cost $n(0) $n(1) 2
$ns cost $n(1) $n(2) 2
$ns cost $n(2) $n(3) 2
#Creazione agente e sorgente di traffico
set udp0 [new Agent/UDP]
$ns attach-agent $n(0) $udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
```

```
$cbr0 attach-agent $udp0
set null0 [new Agent/Null]
$ns attach-agent $n(3) $null0
$ns connect $udp0 $null0
#Procedura per la stampa a video della routing table
proc rtable {} {
      set ns [Simulator instance]
      puts "Tabella di routing all'istante [$ns now]"
      $ns dump-routelogic-nh
      puts "Informazioni sulla distanza"
      $ns dump-routelogic-distance
}
$ns at 0.2 "$cbr0 start"
$ns at 0.2 "rtable"
$ns rtmodel-at 0.6 down $n(5) $n(6)
$ns at 0.8 "rtable"
$ns rtmodel-at 1.0 up $n(5) $n(6)
$ns at 1.5 "$cbr0 stop"
$ns at 1.55 "finish"

$ns run
```

Sezione 2: Routing Distance Vector

```
set ns [new Simulator]
# Apertura del file per NAM
set nf [open routingDV.nam w]
$ns namtrace-all $nf
set tf [open dv3.tr w]
$ns trace-all $tf
#Dichiarazione dell'algoritmo DV
$ns rtproto DV
# Definizione della procedura finish
proc finish {} {
        global ns nf tf
        $ns flush-trace
        close $nf
        close $tf
        exit 0
}
#Definizione topologia
for  {set i 0} {$i < 6} {incr i} {
set n($i) [$ns node]
}
for  {set i 0 } {$i < 6} {incr i} {
    $ns duplex-link $n($i) $n([expr ($i+1)%6]) 1Mb 10ms DropTail
}
$n(0) color "green"
$n(1) color "green"
$n(4) color "red"
$ns duplex-link $n(0) $n(4) 1Mb 10ms DropTail
#La metrica di DV e' basata sui costi e, a parita' di costo, sul numero di hop
$ns cost $n(0) $n(1) 2
$ns cost $n(1) $n(0) 2
$ns cost $n(1) $n(2) 1
$ns cost $n(2) $n(1) 1
$ns cost $n(2) $n(3) 5
$ns cost $n(3) $n(2) 5
$ns cost $n(3) $n(4) 10
$ns cost $n(4) $n(3) 10
$ns cost $n(4) $n(5) 10
$ns cost $n(5) $n(4) 10
```

```
$ns cost $n(5) $n(0) 10
$ns cost $n(0) $n(5) 10
$ns cost $n(4) $n(0) 10
$ns cost $n(0) $n(4) 10
#Creazione agenti e sorgenti di traffico
set udp0 [new Agent/UDP]
$ns attach-agent $n(0) $udp0
$udp0 set fid_ 0
$ns color 0 darkgreen
set udp1 [new Agent/CBR]
$ns attach-agent $n(1) $udp1
$udp1 set fid_ 1
$ns color 1 red
set null4 [new Agent/Null]
$ns attach-agent $n(4) $null4
$ns connect $udp0 $null4
$ns connect $udp1 $null4
set cbr0 [new Application/Traffic/CBR]
$cbr0 attach-agent $udp0
set cbr1 [new Application/Traffic/CBR]
$cbr1 attach-agent $udp1
#Procedura per la stampa a video della routing table
proc rtable {} {
      set ns [Simulator instance]
      puts "Tabella di routing all'istante [$ns now]"
      $ns dump-routelogic-nh
      puts "Informazioni sulla distanza"
      $ns dump-routelogic-distance}
$ns at 0.5 "rtable"
$ns at 0.5 "$cbr0 start"
$ns at 0.7 "$cbr1 start"
$ns at 0.8 "rtable"
$ns rtmodel-at 1.1 down $n(0) $n(4)
$ns at 1.1 "rtable"
$ns at 1.11 "rtable"
$ns at 1.5 "rtable"
$ns rtmodel-at 2 up  $n(0) $n(4)
$ns at 2.1 "rtable"
$ns at 3 "finish"
$ns run
```

BIBLIOGRAFIA

A.A.V.V., “The *ns* Manual”, < http://www.isi.edu/nsnam/ns/doc/>

Baldi M., Nicoletti P., “Internetworking”, McGraw-Hill, 1999

Banks J et al., “Discrete-Event System Simulation”, Prentice Hall, 1996

Bhebhe L., ”NS2 Agents”, Helsinky University of Technology, Laboratory for Theoretical Computer Science, 2005

Bramo L., Peterson L., “Experiences with Network Simulators”, *ACM Sigmetrics*, 1996.

Branden R, Postel J., “Requirements for Internet Gateways”, RFC 1009, Internet Engineering Task Force, Giugno 1987

Campegiani P., “Il simulatore NS-2 (lucidi)”, <http://www.ce.uniroma2.it>

Cardwell N. et al., “Modeling the Performance of Short TCP Connections”, Technical Report, Computer Science Department, Washington University, Novembre 1998.

Clark D., “Window and Acknowledgement Strategy in TCP”, RFC 813, Internet Engineering Task Force, Luglio 1982

Damiani E, “Internet: Guida pratica alla rete internazionale”, Tecniche Nuove, 1994

De Judicibus D., “TCP/IP in pillole - seconda edizione”, Tecniche Nuove, Ottobre 2002.

Douglas E. Comer, “Internetworking with TCP/IP - Volume 1”, Prentice Hall, 1995

Fall K., Floyd S., “Simulation–Based Comparisons of Tahoe, Reno and Sack TCP”, Lawrence Berkeley National Laboratory, Computer Communications Review, Luglio 1996.

Garroppo R. G., “Appunti di Progetto e Simulazione di Reti di Telecomunicazioni: Network Simulator vers. 2 e sue Applicazioni”, SEU Edizioni, 2004

Henderson T. R. et al, “On Improving the Fairness of TCP Congestion Avoidance”, *Proc. IEEE Globecom*, 1998.

Horning C., “The standard for the transmission of IP datagrams accross an Ethernet”, RFC 894, Internet Engineering Task Force, April 1984

Jianping Wang, “ns-2 Tutorial”, Multimedia Networking Group, The Department of Computer Science, UVA, 2004

Mark A., Aaron F., “On the Effective Evaluation of TCP”, *ACM Computer Communication Review*, Ottobre 1999.

Nagle J., “Congestion control in IP/TCP internetworks”, RFC 896, Internet Engineering Task Force, Gennaio 1984

Postel J., “Transmission Control Protocol (TCP)”, Internet Engineering Task Force, RFC 793, Internet Engineering Task Force, Settembre 1981

Postel J., “The standard for the Internet Protocol”, RFC 791, Internet Engineering Task Force, Settembre 1981

Raynolds J, Postel J, ”The Internet Addressing Scheme”, RFC 1700, Internet Engineering Task Force, Ottobre 1984

Rossi G. F., Lucidi del corso di Reti di Calcolatori, Università degli Studi di Pavia, 2005

Rossi G. F., Lucidi del corso di Reti Telematiche, Università degli Studi di Pavia, 2005

Sven Ubik, Jan Klaban, “Experience with using Simulations for Congestion Control Research”, CESNET Technical Report 26/2003

Tanenbaum A. S., “Reti di Computer (seconda edizione)”, Jackson, 1994

Vern Paxson, Sally Floyd, “Difficulties in Simulatine the Internet”, IEEE/ACM Transaction on Networking, Febbraio 2001.

Visweswaraiah V., Heidemann J., “Improving Restart of Idle TCP Connections”, Technical Report 97-661, Università del Sud California, Novembre 1997.