Università di Modena e Reggio Emilia
Facoltà di Ingegneria Enzo Ferrari

Analisi e integrazione di tecniche di Packet Scheduling e Active Queue Management

Progetto per Laboratorio di Ingegneria Informatica

Referente: Prof. Maurizio Casoni

Project Leader: Guido Borghi
Andrea Palazzi

Anno Accademico 2014/2015

Sintesi

La presente relazione descrive il lavoro di studio e implementazione di tecniche di *Packet Scheduling* e *Active Queue Management* sul simulatore di reti ns-3, svolto nell'ambito dell'esame di Laboratorio di Ingegneria Informatica.

Il documento può essere suddiviso in alcune parti principali:

- I primi due capitoli (1, 2) sono di carattere teorico/descrittivo e hanno l'obiettivo di introdurre il concetto di *Quality of Service* (QoS) come requisito ormai indispensabile di un servizio informatico moderno. Vengono quindi da un lato descritte le cause che possono compromettere la qualità del servizio, dall'altro le soluzioni che possono essere adottate onde evitare che ciò avvenga.

- I due capitoli centrali (3, 4) sono ancora di carattere teorico, e presentano i due essenziali meccanismi per ottenere un'alta QoS, cioè il *Packet Scheduling* e l'*Active Queue Management*. Si spiega in cosa differiscano i due metodi, molto spesso confusi in letteratura. Sia per l'uno che per l'altro sono presentate più varianti e diverse modalità di implementazione.

- Il capitolo 5 è il più corposo, e al suo interno viene esposto il modo in cui gli algoritmi di *Packet Scheduling* e *Active Queue Management* sono stati implementati e testati sul simulatore ns-3. Viene dato molto spazio ai grafici per visualizzare in modo chiaro i risultati delle varie simulazioni. Vengono proposti quindi anche alcuni casi di studio per verificare l'efficacia delle soluzioni presentate in differenti scenari.

- Nell'ultimo capitolo vengono presentate le conclusioni e sintetizzati i risultati del lavoro svolto.

Indice

Capitolo 1

Introduzione

La rete Internet nasce per offrire una consegna di tipo *Best Effort* di pacchetti tra gli host. Questi possono quindi essere persi, duplicati, corrotti o subire ritardo indefinito: la rete non garantisce all'utente alcun livello di qualità o priorità nella trasmissione, ed anzi non può garantire nemmeno che i dati giungano sicuramente e nel giusto ordine a destinazione.
In una rete best-effort tutti gli utenti ottengono lo stesso servizio best-effort, il che significa che ogni connessione avrà latenza e bit-rate variabile a seconda del carico della rete.

Il problema della mancanza di garanzie riguardo alla qualità del servizio offerto non venne avvertito nella fase iniziale della vita di Internet, perchè le applicazioni di rete storiche (si pensi ad esempio alla condivisione di file mediante FTP o alla posta elettronica basata su protocollo SMTP) non avevano forti requisiti in termini di banda nè erano particolarmente sensibili al ritardo, ragion per cui erano in grado di funzionare anche su reti con prestazioni degradate.
Col passare degli anni tuttavia l'estensione, la capillarità e il numero di utenti della rete Internet sono cresciuti esponenzialmente, e i servizi fruibili tramite questa si sono moltiplicati. Ai servizi tradizionali *loss sensitive* si sono aggiunti nuovi servizi *delay sensitive*, che fino a poco prima erano offerti solo su reti a commutazione di circuito: l'esempio principe è probabilmente la tecnologia VoIP (Voice over IP) che permette l'instaurazione di connessioni telefoniche sulla rete Internet. E' proprio su reti a commutazione di circuito che tali servizi possono disporre di risorse in maniera *esclusiva*, paradigma non sempre possibile in contesti di reti a commutazione di pacchetto.

Emerge dunque con forza il problema di garantire una certa qualità del servizio al singolo utente che ne faccia richiesta: si apre quindi il tema della *Quality of Service (QoS)*.

Capitolo 2

Quality of Service

2.1 Definire la QoS

Poichè il sintagma Qualità di Servizio è estremamente generico, nonostante l'attualità e la diffusione di questo termine è difficile darne una definizione univoca.
A grandi linee si può affermare che la QoS è riconducibile al garantire un determinato valore di throughput ad una parte selezionata del traffico di rete. Tale definizione non è tuttavia soddisfacente, perché la QoS vorrebbe indicare la qualità di servizio vista agli estremi della comunicazione, quindi in ultima analisi dagli utenti della rete. La visione della QoS come qualcosa che di percepito agli end-point della comunicazione, dato dalla somma dei contributi di componenti intermedi è chiarita dalla seguente immagine, tratta dalla raccomandazione E.800 di ITU-T che cerca appunto di definire i termini relativi alla QoS.

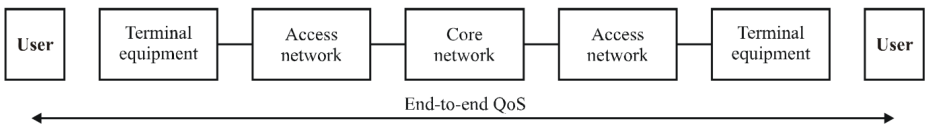

Fig. 2.1: Diversi contributi alla QoS percepita agli endpoints

Inoltre, lo stesso significato del termine QoS acquista luci diverse a seconda della prospettiva da cui lo si guarda.
Abbiamo infatti un primo tipo di QoS, che coincide con i requisiti e le aspettative dell'utente della rete, che possono essere diverse dalla QoS dei servizi che il provider propone. Approcciando il problema da una seconda angolazione, possiamo definire la QoS come il livello di qualità effettivamente raggiunto dal fornitore del servizio, che può essere diverso rispetto a quanto pensava di poter offrire sulla carta. Infine, la QoS effettivamente erogata

dal provider sarà quella che formerà la percezione di QoS dell'utente, e quest'ultimo sarà più o meno soddisfatto a seconda che la QoS percepita sia maggiore o minore delle sue aspettative iniziali (quest'ultima sfumatura è indicata talvolta in letteratura come QoE, ovvero *Quality of Experience*). Tale visione circolare è riassunta nell'immagine seguente, sempre tratta dalla raccomandazione ITU-T E.800 sopra citata:

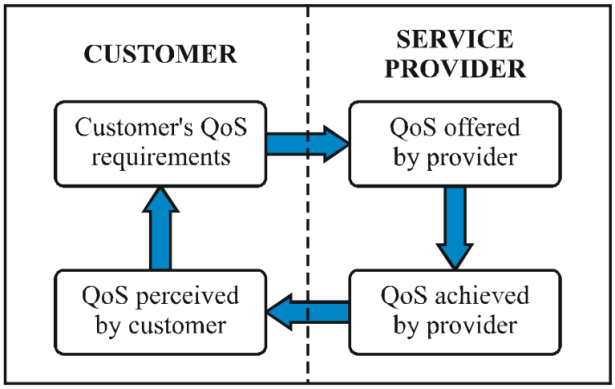

Fig. 2.2: Quattro punti di vista distinti di QoS

Finora si sono date definizioni generali, astratte e quindi non facilmente quantificabili. L'azienda leader mondiale di apparati di networking Cisco Systems propone invece, come si può leggere nella documentazione presente sul sito dell'azienda, una definizione puntuale, che pone l'accento su alcuni parametri fondamentali nella teoria delle reti, come larghezza di banda, latenza, jitter e probabilità di perdita.

> Quality of Service (QoS) refers to the capability of a network to provide better service to selected network traffic over various technologies[...]. The primary goal of QoS is to provide priority including dedicated bandwidth, controlled jitter and latency (required by some real-time and interactive traffic), and improved loss characteristics.

2.2 Elementi che degradano la QoS

Basandoci sulle precedenti definizioni, possiamo quindi introdurre alcuni elementi che possono degradare la qualità del servizio in una rete a commutazione di pacchetto: una buona politica di QoS cercherà di limitare al minimo questi fattori.

- **Basso throughput**: se la bit-rate di un certo flusso di traffico è troppo bassa, molto probabilmente sarà impossibile la fruizione corretta di

servizi che prevedono un scambio di dati significativo, come i servizi multimediali.

- **Perdita, Errori**: se la probabilità di perdita dei pacchetti trasmessi è significativa, ad esempio nel caso di una rete congestionata, la comunicazione subirà forti ritardi (dovuti alle ritrasmissioni dei pacchetti persi) o sarà comunque compromessa. Un analogo disagio può crearsi quando i pacchetti sono corrotti a livello di bit a causa del rumore o interferenze (più probabili in determinati contesti, come le comunicazioni wireless).

- **Latenza, Jitter**: quando un pacchetto impiega molto tempo per raggiungere la destinazione, ad esempio perchè rimane imbottigliato nei buffer dei router o perchè prende strade alternative per evitare i tratti di rete congestionati, si dice che c'è un'alta latenza. Per ovvie ragioni, una latenza eccessiva rende inutilizzabile qualsiasi servizio che abbia esigenze di real-time. Quando la latenza è molto variabile nel corso della trasmissione di parla di jitter.
  Il jitter rende difficile prevedere quando arriverà il prossimo pacchetto, causando problemi al corretto dimensionamento dei buffer e compromettendo quindi i servizi di streaming.

- **Consegna fuori ordine**: in assenza di politiche di qualità di servizio, quando una sequenza di pacchetti è trasmessa nella rete ogni pacchetto può prendere una strada diversa dagli altri e quindi impiegare un tempo diverso per giungere a destinazione. Per questa ragione i pacchetti possono arrivare al destinatario in un ordine completamente diverso da quello in cui sono stati trasmessi. L'arrivo fuori ordine crea molti problemi ai servizi in cui la corretta sequenza di ricezione dei pacchetti è fondamentale per non alterare la semantica (es: VoIP).

Come si è anticipato, servizi di rete diversi avranno esigenze diverse in termini di larghezza di banda, latenza, jitter, probabilità di perdita. Ne deriva che ogni applicazione ha una diversa tolleranza, più o meno lasca, ad ognuno dei fattori sopra citati.

2.3 Meccanismi per la QoS

In una rete priva di meccanismi di QoS, funzionante cioè in modalità Best-Effort, ogni flusso riceve le risorse disponibili in quel dato momento, in base alla situazione di carico della rete: l'implementazione è immediata, perché tutti i flussi di traffico sono trattati nello stesso modo. Tuttavia, se nasce l'esigenza di offrire una determinata QoS, devono essere introdotti meccanismi e algoritmi che massimizzino l'utilizzazione delle risorse di rete e cerchino di

Application	loss	bandwidth	time-sensitive
File transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web browsing	no loss	elastic (few kbps)	no
VoIP	loss-tolerant	[few kbps, 1 Mbps]	100s msec
VIP	loss-tolerant	[10 kbps, 5 Mbps]	100s msec
Stored audio/video	loss-tolerant	like VoIP and VIP	few seconds
Gaming	loss-tolerant	[few kbps, 10 kbps]	100s msec
Chat	no loss	elastic	depends

Fig. 2.3: Ogni applicazione ha i propri peculiari requisiti

garantire ad ogni flusso di dati i requisiti minimi di cui la comunicazione ha bisogno. Tali meccanismi possono essere implementati nei modi più diversi, anche se schematicamente possiamo sempre individuare questi elementi:

- **Admission Control**: la rete valuta di poter accettare l'ingresso di un nuovo utente in base alle risorse a sua disposizione, in quanto la nuova accettazione non deve andare a penalizzare la qualità del servizio agli utenti già attivi.
- **Traffic Control**: si articola sostanzialmente in due parti. La prima prevede di andare a smussare eventuali picchi di traffico generati dall'utente con il fine di preservare i requisiti di QoS per gli altri utenti connessi alla rete. La seconda prevede di imporre all'utente il rispetto del profilo di traffico contrattato nel momento di accesso alla rete mediante opportune *policing* di gestione del flusso dell'utente.
- **Packet Scheduling**: selezionare quali pacchetti di quale utente devono essere trattati in modo particolare rispetto agli altri, a seconda dei requisiti di QoS richiesti. Determinate tipologie di traffico necessitano di essere trasmesse con priorità maggiore rispetto ad altre.
- **Buffer Management**: è un aspetto particolarmente delicato in quanto le memorie sono condivise tra un certo numero di flussi di dati; è quindi necessario definire le politiche di scarto dei pacchetti nel momento in cui risultano essere sovraccariche. E' intuitivo capire come tali politiche possano influire significativamente sulle prestazioni della rete.
- **Controllo di flusso e congestione**: è l'insieme di tutte quelle tecniche di controllo di trasmissione che lavorano basandosi sullo stato della rete e dell'utente mittente e ricevente. Nello specifico il controllo di flusso si preoccupa di limitare i flussi di traffico degli utenti in modo tale da non mandare la rete in uno stato di overflow, saturando il

buffer di ricezione, evitando così la perdita e il conseguente rinvio dei pacchetti che abbasserebbe l'efficienza della trasmissione; il controllo di congestione serve per prevenire e limitare i fenomeni di congestione dei dispositivi all'interno della rete.

- **QoS Routing**: l'instradamento dei pacchetti deve tenere conto dei requisiti di QoS dell'utente per determinare la politica di instradamento migliore che possa soddisfare, tramite le risorse residue di rete, il livello di qualità del servizio richiesto.

2.4 QoS tramite Over-provisioning

Un discorso a parte merita un altro metodo che può essere utilizzato per aumentare la Qualità del Servizio, cioè il cosiddetto *over-provisioning*, o sovradimensionamento della rete. Questo approccio prevede che durante la fase di progettazione della rete, le risorse siano allocate generosamente, in modo da poter gestire con un buon margine di sicurezza anche il traffico dei momenti di picco.
Nonostante questa idea abbia una sua efficacia non può sicuramente essere considerata la soluzione definitiva al problema del raggiungimento della QoS in una rete, anzi presenta alcuni grossi problemi. Prima di tutto, dal momento che la rete è dimensionata a partire dai picchi di traffico, in condizioni normali le risorse sono estremamente sotto-utilizzate. Inoltre, non è detto che i picchi di carico siano facilmente prevedibili, né che si possa prevedere con precisione la loro intensità.
Infine si nota come tale approccio sia estremamente poco flessibile: in mancanza di altri meccanismi per garantire la qualità del servizio, è sufficiente che nasca un nuovo servizio con un consumo di banda maggiore di quanto previsto, o che attiri più utenti di quelli per cui la rete è dimensionata, per causare inevitabili situazioni di congestione.
In sostanza, un sovradimensionamento della rete in fase di progetto può sicuramente essere d'aiuto per poter fornire una buona QoS ai propri utenti, ma non può in nessun caso essere l'unico meccanismo utilizzato.

Capitolo 3

Packet Scheduling

Con il termine di *Packet Scheduling* ci si riferisce solitamente al processo decisionale che avviene nei nodi di una rete con l'obiettivo di scegliere il successivo pacchetto da trasmettere sul canale. Per lungo tempo la soluzione più diffusa è stata di servire e inoltrare i pacchetti nell'ordine in cui questi giungevano al nodo: tale tecnica di scheduling banale, detta FIFO (First In First Out) o FCFS (First Come First Served) corrisponde di fatto all'assenza di scheduling. Nonostante in molti contesti una politica FIFO possa tuttora dare risultati soddisfacenti, la nascente esigenza di QoS ha determinato lo sviluppo di numerosi altri algoritmi di scheduling anche molto complessi, che tengano conto della diversa priorità con cui occorre servire pacchetti.

3.1 Proprietà degli algoritmi di Packet Scheduling

La varietà attuale di algoritmi di Packet Scheduling esistenti è enorme. Tuttavia, nonostante si differenzino più o meno fortemente nell'implementazione, ogni packet scheduler tende a soddisfare una serie di requisiti, i principali dei quali sono riportati di seguito. In ogni caso poichè non di rado tali esigenze sono in contrasto tra loro (es: velocità ed equità) ed ogni algoritmo risolve in modo diverso questo *trade-off*, non esiste nè può esistere uno scheduler in assoluto migliore o peggiore in ogni contesto.

- **Velocità**: grazie alla crescente diffusione della fibra ottica come mezzo trasmissivo, il collo di bottiglia delle reti si sta spostando sempre più dai canali verso i nodi. Poiché un moderno router di fascia alta può dover smistare traffico a velocità maggiori di 1 Tbps, emerge la necessità di minimizzare la complessità computazionale dell'algoritmo di scheduling utilizzato. Questo requisito purtroppo è spesso in conflitto con altri, in quanto solitamente gli scheduler più veloci rischiano di essere meno equi e flessibili di quelli che prendono decisioni più ponderate.

- **Scalabilità**: un algoritmo di scheduling è definito scalabile se è in grado di operare indipendentemente dal numero attuale di connessioni. Si tenga infatti presente che i router di backbone operano abitualmente su migliaia di flussi contemporaneamente.

- **Equità**: uno scheduler è tanto più equo quanto più riesce a suddividere in modo equilibrato il canale tra i flussi che lo contendono. Si noti che una suddivisione giusta della capacità del canale non significa che tutti i flussi in contesa devono ricevere la stessa percentuale di banda: anzi, uno scheduler *fair* deve tenere conto delle diverse esigenze di QoS per garantire ad ogni flusso una percentuale un tempo di servizio proporzionale alla sua priorità.

- **Protezione**: la presenza di un certo flusso di pacchetti di cui garantire una determinato servizio non dovrebbe andare compromettere, per quanto possibile, la qualità del servizio di altri flussi di dati. Questa caratteristica viene definita anche come *flow isolation*. A livello teorico quindi ogni flusso dovrebbe essere indipendente e autonomo rispetto agli altri.

- **Flessibilità**: essendo la richiesta di determinati livelli di qualità del servizio molto varia, un algoritmo di scheduling deve saper gestire differenti caratteristiche di traffico e requisiti di performance degli utenti. E' una proprietà che assume notevole importanza soprattutto nel contesto attuale, orientato al Pervasive Web, dominato quindi da una molteplicità di device con differenti tipi di connessioni (LTE, 3G, Wi-Fi e altri).

3.2 Packet Scheduling nella Sicurezza Informatica

Il tema del Packet Scheduling e dell'Active Queue Management, che verrà trattato successivamente, sconfinano presto anche nell'ambito della Sicurezza Informatica. Questo tema viene qui appena citato in quanto una più lunga trattazione, vista la vastità della materia, non tra gli scopi di questa relazione.
Si accenna tuttavia al fatto che la selezione dei pacchetti da inviare è alla base dei meccanismi utilizzati per contrastare eventuali flussi di traffico dal comportamento malizioso: si sta parlando di attacchi di tipo *DoS (Denial of Service)*, *DDoS (Distributed DOS)* e *LDoS (Low-Rate DOS)*, che hanno l'obiettivo di interrompere l'erogazione del servizio della vittima saturandone le risorse disposte lungo la rete tramite l'invio di flussi abnormi di pacchetti (ad eccezione del LDoS che proprio per mascherarsi verso i sistemi di difesa basa i suoi attacchi su piccole quantità di traffico periodiche). L'implementazione nel *border router* di un Packet Scheduler o altri meccanismi

di gestione della memoria, con buone capacità di protezione, che forniscano cioè un buon isolamento tra i flussi, possono evitare che i flussi di traffico più consistenti sommergano gli altri.
Anche un algoritmo di scheduling estremamente semplice come il Round Robin (che sarà presentato in dettaglio a breve) può essere sufficiente allo scopo, dal momento che fornisce la stessa quantità di servizio a tutti i flussi indipendentemente dal numero di pacchetti in coda.
Ovviamente tali algoritmi devono essere associati a ulteriori politiche di sicurezza informatica per ottenere risultati concreti.

3.3 Generalized Processor Sharing (GPS)

Il Generalized Processor Sharing è un modello matematico che descrive il modo in cui dovrebbe operare uno scheduler perfettamente *fair*, che cioè distribuisce la risorsa contesa dai flussi di traffico nel modo più equo possibile. Tale risultato viene raggiunto visitando a turno ogni coda non vuota e servendo un infinitesimo di ciascuna: il tempo di servizio dedicato ad ogni coda è proporzionale alla priorità di tale flusso.
Siano w_i e w_j i pesi assegnati all'i-esimo e al j-esimo flusso che hanno dati in attesa di essere trasmessi. Chiamando $S(i, \Delta t)$ la quantità di dati del flusso i-esimo serviti nell'intervallo di tempo Δt, GPS garantisce che in questo intervallo di tempo ogni flusso riceva un servizio proporzionale al suo peso, più formalmente:

$$\frac{S(i, \Delta t)}{S(j, \Delta t)} \geq \frac{w_i}{w_j}$$

La ragione per cui il GPS riesce ad essere perfettamente equo è che agisce nel dominio continuo, ipotizzando che sia possibile suddividere il traffico in infinitesimi: poiché tuttavia in un contesto reale i dati sono trasmessi in unità estremamente discrete (i pacchetti) e non ulteriormente suddivisibili, il modello GPS può essere implementato solo in forma approssimata. La maggiore importanza di tale modello è quindi quella di fornire il risultato ottimo a cui occorre tendere nell'implementazione di un algoritmo di scheduling in un contesto reale.

3.4 Weighted Fair Queuing (WFQ)

Weighted Fair Queueing, solitamente tradotto come *accodamento equo pesato*, deve la sua popolarità al fatto di essere l'approssimazione "a pacchetti" del GPS. In questo caso, i pacchetti sono smistati in diverse classi a seconda della loro priorità: di nuovo, ad ogni classe i è associato un peso $w_i \in R$. Servendo ciclicamente le classi in coda, WFQ garantisce che, data una linea

di capacità L, alla classe i-esima sia garantito un throughput pari a:

$$L\frac{w_i}{\sum_j w_j}$$

Si tenga tuttavia presente che tale valore sarà raramente raggiunto in modo esatto, a causa della dimensione discreta dei pacchetti e di eventuali modalità di lavoro pre-emptive, che forzano l'interruzione della trasmissione dei pacchetti a bassa priorità all'arrivo di pacchetti a priorità maggiore.

3.5 Algoritmi di Packet Scheduling

3.5.1 First In First Out (FIFO)

E' l'algoritmo di scheduling più semplice e intuitivo; consiste nel servire i pacchetti nel loro ordine di arrivo. E' molto semplice e primitivo in quanto non tiene in considerazione dei parametri di QoS di ogni pacchetto e non effettua nessuna operazione di selezione se non, appunto, il mantenimento dell'ordine di arrivo. Occorre definire la politica di scarto quando la coda risulta essere piena; un pacchetto servito viene rimosso dalla coda. Suo punto di forza risulta essere l'estrema bassa complessità computazionale che lo rende adatto ad essere utilizzato su qualsiasi tipo di nodo, anche con hardware limitato, per servizi di tipo *best effort*.
Questa estrema semplicità però risulta essere non sufficiente a garantire un determinato livello di qualità del servizio quando richiesto: ad esempio un flusso di dati particolarmente grande che impegna interamente le risorse disponibili non rende possibile l'invio degli altri pacchetti posti in coda, che così attendono un periodo di tempo potenzialmente anche molto elevato prima di essere serviti.
Quindi, riassumendo, la maggiore caratteristica positiva è senza dubbio la bassa complessità a fronte però del trattare tutti i pacchetti senza distinzione e protezione.

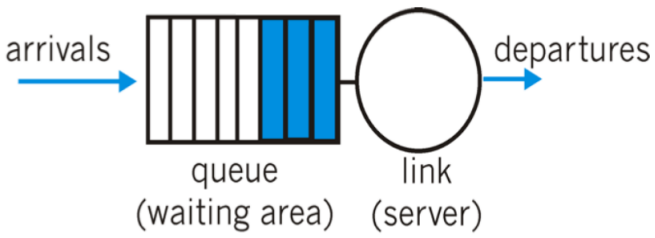

Fig. 3.1: Schema grafico del FIFO

3.5.2 Priority Scheduling (PRIO)

Rispetto alla gestione FIFO questo algoritmo introduce il concetto di classi di pacchetti: all'arrivo di un nuovo pacchetto, questo viene classificato e inserito nel rispettivo buffer di attesa; ogni classe è associata a una certa priorità e ciascuna viene servita in base a questa. L'algoritmo cerca sempre di servire la coda con priorità più alta.
Risulta quindi essere un algoritmo semplice ma che permette la distinzione dei vari flussi in arrivo, non implementa la proprietà di protezione.
Inoltre si possono avere fenomeni di *starvation*, ovvero di impossibilità perpetua di invio dei pacchetti di una classe, come nel caso della compresenza di un flusso elevato con alta priorità, sempre *backlogged*, e di un secondo flusso a più bassa priorità.

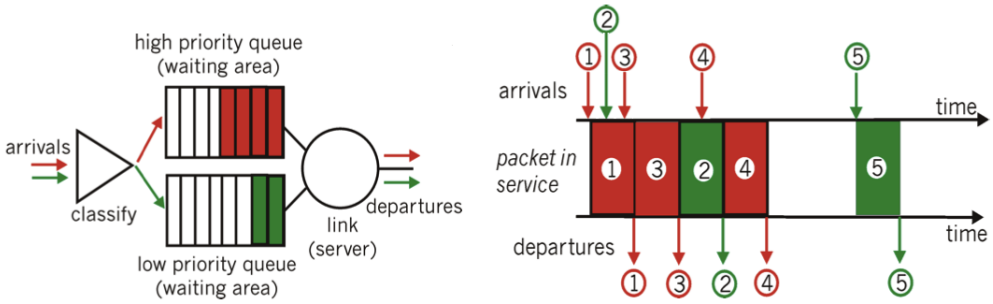

Fig. 3.2: Schema grafico del PRIO

3.5.3 Round Robin (RR)

Anche tale algoritmo risulta essere piuttosto semplice, è stato infatti uno dei primi ad essere implementato nelle reti a pacchetto. Prevede di servire in maniera ciclica tutte le code di pacchetti che risiedono all'interno il buffer di memorizzazione.
E' un algoritmo *fair* in quanto non effettua distinzioni tra i vari flussi di dati ma assegna le risorse in modo equivalente a tutti gli utenti: inutile dire che l'algoritmo si comporta in modo davvero equo solo facendo l'ipotesi di flussi che trasmettono pacchetti di uguale dimensione.
Se da un lato questa modalità circolare di servizio dei flussi in contesa risolve il problema di utilizzo esclusivo del canale da parte di burst di pacchetti particolarmente grandi (visto nel caso precedente) dall'altra non permette di effettuare nessuna politica particolare di selezione dei pacchetti, non rispettando quindi la caratteristica di flessibilità.

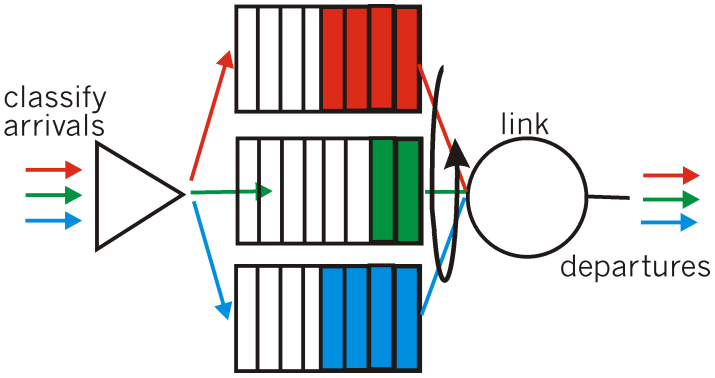

Fig. 3.3: Schema grafico del RR

3.5.4 Weighted Round Robin (WRR)

L'algoritmo Round Robin è molto diffuso anche nella sua variante pesata detta Weighted Round Robin (WRR). In questo caso i flussi che contendono il canale non sono considerati uguali, ma ad ognuno è assegnato un proprio peso secondo qualche criterio: dopodiché il buffer è ancora servito in modo circolare, ma in questo caso si cerca di erogare ad ogni flusso una quantità di servizio proporzionale al proprio peso.
WRR generalmente serve un numero di pacchetti per ogni coda non vuota pari a

$$number = normalized(weight/PacketSize)$$

E' interessante notare come oltre al peso venga tenuta in considerazione pure la dimensione del pacchetto, al fine di garantirne il funzionamento *fair* anche in presenza di flussi di pacchetti di dimensioni molto differenti tra di loro.
Per fare un esempio, si ipotizzi di avere due flussi di pacchetti che contendono il canale, al primo dei quali (chiamiamolo A) si vorrebbe assegnare il 60% della banda disponibile, mentre il secondo (chiamiamolo B) si deve accontentare del 40% residuo. Round Robin non effettua nessuna distinzione tra i due, quindi i due flussi saranno serviti in modo circolare (ABABABAB...) e ognuno otterrà circa metà della banda disponibile (ipotizzando che i pacchetti abbiano uguale dimensione). Utilizzando Weighted RR invece la sequenza di servizio sarà del tipo ABAABABAAB, approssimando quindi i requisiti richiesti.
Ciò è chiarito anche dalla figura 3.4.

3.5.5 Deficit Round Robin (DRR)

Il Deficit Round Robin è sostanzialmente un WRR con alcune modifiche al suo funzionamento. Il meccanismo di scheduling verte sul valore di una particolare variabile, il *deficit counter*: infatti, a differenza del WRR che serve

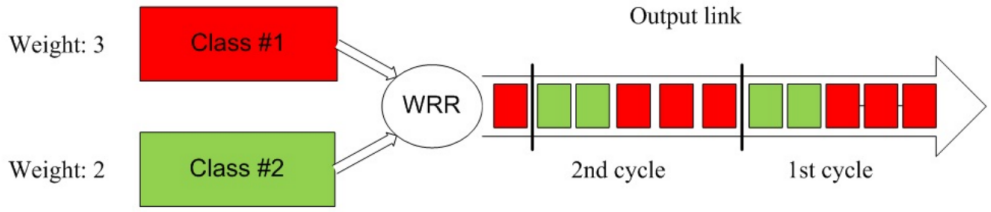

Fig. 3.4: Schema grafico del WRR

una coda non vuota, il DRR serve i pacchetti di ciascuna coda non vuota per la quale il *deficit counter* è maggiore della dimensione del pacchetto in testa alla coda; se risulta essere di dimensione inferiore, la coda non viene servita (*skip*) mentre il valore del deficit counter viene incrementato di una piccola quantità chiamata *quantum*; al contrario, se la coda viene servita il *counter* viene diminuito di una quantità pari alla dimensione del pacchetto servito. Nell'immagine 3.5 la dimensione del quantum è pari a 500 byte, si può notare inoltre il deficit counter assegnato a ogni classe di servizio.
Il maggiore beneficio è rappresentato dal non dover recuperare la dimensione dei pacchetti che arrivano allo scheduler, rendendone più semplice l'implementazione e più basso il peso computazionale.

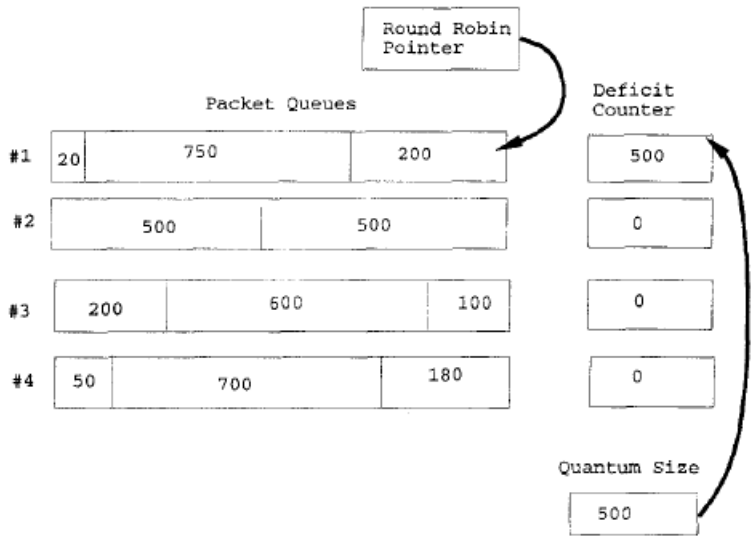

Fig. 3.5: Schema funzionamento di DRR

Capitolo 4

AQM (Active Queue Management)

La tecnica tradizionale utilizzata dai router per gestire le code è sempre stata quella di fissare una lunghezza massima (in termini di pacchetti) per ogni coda, accettando pacchetti nella coda fintantoché questa non è piena, rifiutando poi i pacchetti in arrivo (*drop*) fino a che la coda non si accorcia nuovamente in seguito alla trasmissione di alcuni dei pacchetti in attesa. Questa tecnica è nota con il nome di *tail drop*, in quanto se la coda è piena il drop coinvolge i pacchetti arrivati più di recente, cioè quelli che si trovano in fondo alla coda.
Nonostante questo metodo funzioni, ed in effetti è stato utilizzato in Internet per molti anni, presenta due fondamentali problemi:

- **Lock-Out**: In determinate situazioni la politica di tail-drop permette che una singola o poche connessioni monopolizzino lo spazio della coda, impedendo l'accodamento di pacchetti provenienti da connessioni diverse, che risultano bloccate fuori (*locked out*) dalla coda.

- **Code piene**: Gestire la coda in modalità *tail-drop* fa sì che all'aumentare del traffico per lunghi periodi di tempo la coda sia completamente piena, in quanto la congestione è segnalata attraverso il drop di pacchetti solo quando questa è ormai satura. Una coda piena per gran parte del tempo sarà quindi incapace di gestire l'arrivo a burst di pacchetti, perché ci sarà sempre molto poco spazio libero. Al contrario, una coda più libera può assorbire un burst di dati e trasmetterlo (auspicabilmente) al primo burst di silenzio, dando luogo ad un maggiore throughput e ad un minore delay di comunicazione tra gli end-point.

Una possibile soluzione a questi problemi è quella di far sì che i *router* comincino a scartare pacchetti prima che la coda si riempia completamente, in modo che gli end-point possano rilevare la situazione di congestione prima

che avvenga effettivamente il buffer overflow; questo tipo di approccio proattivo è chiamato Active Queue Management (AQM). In questo modo, poichè i *router* cominciano un moderato *drop* di pacchetti ben prima che la coda si riempia, hanno il tempo di selezionare quali e quanti pacchetti scartare: si vede quindi il vantaggio rispetto alla politica *tail drop*, nella quale il *router* agisce quando ormai la coda è piena e deve prendere la misura drastica di scartare tutti i pacchetti in arrivo senza poter effettuare distinzioni di sorta.

4.1 Algoritmi di AQM

4.1.1 Random Early Detection (RED)

Random Early Detection (RED) è un algoritmo di AQM che risolve i problemi di *tail drop* citati nella sezione precedente.
Al contrario di politiche di gestione code più datate, nelle quali si prevede di scartare pacchetti solo al riempimento del buffer, RED scarta i pacchetti in arrivo in modo probabilistico, con una probabilità di *drop* che cresce al crescere della lunghezza stimata della coda.
Si noti che RED opera sulla base della lunghezza media della coda calcolata in un certo intervallo di tempo, non della lunghezza istantanea. In altre parole, se la coda è stata quasi vuota nel passato recente, RED tenderà a non scartare pacchetti, mentre se la coda è stata tendenzialmente piena, indicando quindi una situazione di congestione persistente, anche i nuovi pacchetti in arrivo avranno maggiori probabilità di essere scartati.

Guardando più da vicino, si nota che la modalità operativa di RED può essere scomposta principalmente in due parti.
Una prima parte dell'algoritmo è dedicata alla stima della lunghezza media della coda: questo solitamente è fatto utilizzando semplici meccanismi di finestra scorrevole, pesando maggiormente le informazioni che riguardano la storia più recente della coda.
La seconda porzione dell'algoritmo invece si occupa di decidere se effettuare o no il *drop* di un pacchetto in arrivo: in questa fase entrano in gioco i due parametri fondamentali di RED, che sono la soglia minima di lunghezza della coda ($MinTh$), sotto la quale nessun pacchetto sarà scartato, e specularmente una soglia massima ($MaxTh$) raggiunta la quale nessun pacchetto sarà accettato. Al variare della lunghezza media della coda tra $MinTh$ e $MaxTh$, la probabilità che un pacchetto in arrivo venga scartato varia linearmente tra 0 e la probabilità massima di *drop* ($MaxP$).

La regolazione dei precedenti parametri (fase di *tuning*) viene poi eseguita sulla base del particolare scenario di applicazione dell'algoritmo nel tentativo di massimizzarne l'efficacia di funzionamento.

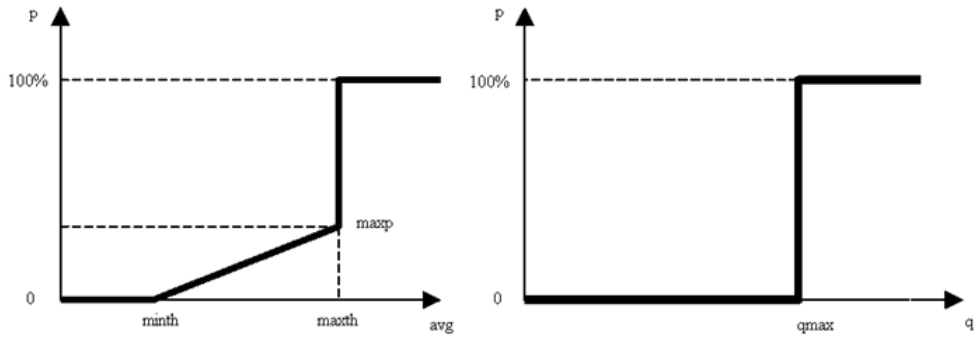

Fig. 4.1: Probabilità di drop in funzione della lunghezza della coda: RED a confronto con tail drop

4.1.2 Weighted random early detection (WRED)

Weighted Random Early Detection (WRED) è un algoritmo che estende le funzionalità di RED. Ciò avviene introducendo il concetto di peso (*weight*) associato al pacchetto, permette di definire in maniera specifica i parametri relativi a RED, analizzati nel paragrafo precedente, in funzione della tipologia di servizio che si vuole instaurare per le varie applicazioni.
In concreto, la probabilità di drop ed i valori di soglia minima e massima di lunghezza della coda (*MinTh* e *MaxTh*), vengono calcolati in base alla tipologia del pacchetto; gli altri meccanismi di funzionamento, (come, ad esempio, la stima della lunghezza media della coda) vengono conservati invariati rispetto all'algoritmo di riferimento.

Risulta quindi possibile avere differenti *classi di servizio*, andando a soddisfare per quanto possibile le esigenze di applicazioni differenti, scartando in maniera pesata i pacchetti in arrivo.

Nell'immagine 4.2 che illustra il funzionamento di WRED è stata fatta l'ipotesi di avere tre classi di servizio, marcate con i colori giallo, verde e rosso, di cui si possono notare i diversi parametri di limite di lunghezza delle code e quindi della relativa probabilità di drop.

4.1.3 Robust Random Early Detection (RRED)

Il Robust Random Early Detection (RRED) è una variante dell'algoritmo RED concepita essenzialmente per incrementare il throughput del protocollo TCP, in particolare in caso di attacchi di tipo DoS (cfr. paragrafo *Packet Scheduling nella Sicurezza Informatica*): l'intento è quindi quello di andare a migliorare una delle debolezze insite degli algoritmi *RED-like* che li rendono poco adatti a contenere attacchi informatici legati alla saturazione delle risorse della rete e/o degli host connessi.

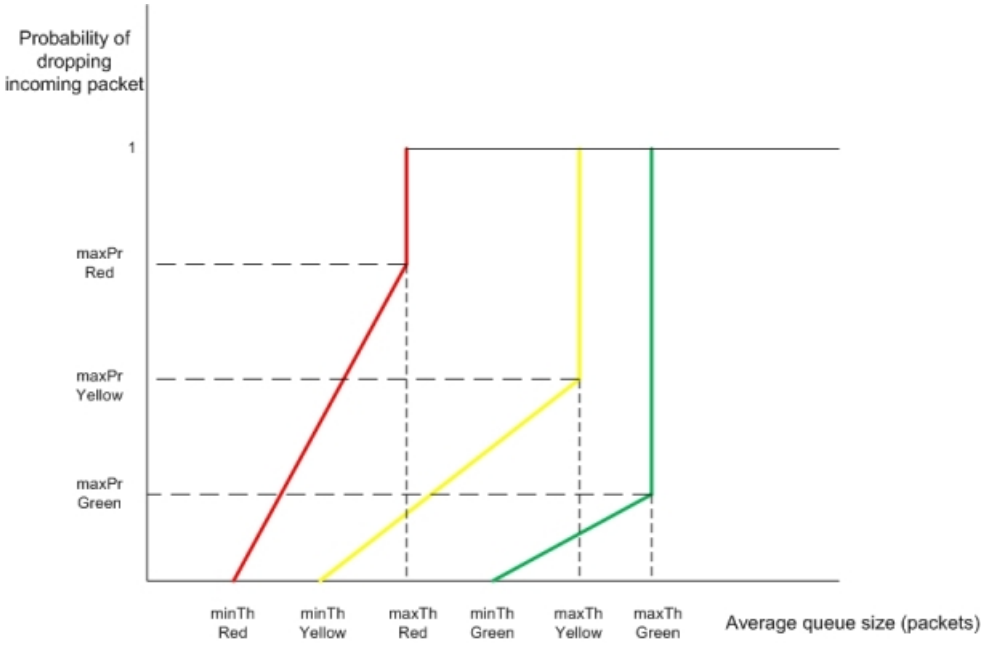

Fig. 4.2: Probabilità di drop in funzione della lunghezza della coda in WRED: tre ipotetiche classi di servizio messe a confronto

Solitamente viene utilizzato a fianco dell'algoritmo RED: l'idea di base infatti è quella di utilizzare il meccanismo RRED per filtrare i pacchetti che giungeranno poi al RED, in maniera tale da cercare di prevenire l'esplosione dei pacchetti posti in coda, in attesa di essere serviti.
Risulta comunque di difficile implementazione un algoritmo perfettamente efficiente in grado di determinare se un pacchetto TCP proviene da un attacco DoS oppure no: per fare questo RRED utilizza come parametro il tempo di arrivo dei pacchetti, a differenza delle altre versioni dell'algoritmo RED.

Il funzionamento nei dettagli può essere chiarito mediante il seguente pseudocodice:

```
RRED-ENQUE(pkt)
      f<-RRED-FLOWHASH(pkt)
      Tmax<-MAX(Flow[f].T1, T2)
      if pkt.arrivaltime is within [Tmax, Tmax+T*]
      then
         reduce local indicator by 1 for each bin
         corresponding to f
      else
              increase local indicator by 1 for each bin
              of f
```

```
Flow[f].I maximum of local indicators from bins of f
 if Flow[f].I >=0 then
         RED-ENQUE(pkt)  //pass pkt to the RED block
         if RED drops pkt then
                 T2<-pkt.arrivaltime
 else
         Flow[f].T1<-pkt.arrivaltime
         drop(pkt)
 return
```

Capitolo 5

Analisi e integrazione in ns-3

In questo capitolo si prendono in considerazione e si cercano di implementare le tecniche di Packet Scheduling e AQM sopra trattate. Nello specifico si studia, tramite il simulatore ns-3 e altri strumenti elencati nel paragrafo successivo, il problema dell'erogazione di servizi in reti congestionate. I protocolli utilizzati sono il File Transfer Protocol (FTP) e il VoIP.

5.1 Elementi utilizzati

5.1.1 Strumenti software

Sono stati utilizzati diversi software per completare il lavoro.

- **ns-3**: simulatore di reti a eventi discreti; da sottolineare la sua natura *open-source*, che lo rende adatto per studi e ricerche accademiche. E' stato eseguito su architetture *linux based*.
- **Wireshark**: software per analizzare i protocolli di rete; nello specifico viene utilizzato per studiare, tramite i file di output di ns-3 (*.pcap*) il comportamento delle reti simulate.
- **Github**: software di sincronizzazione utilizzato per condividere in maniera sempre aggiornata i file di programmazione fra i due membri del gruppo.

5.1.2 Protocolli

- **File Transfer Protocol (FTP)**: protocollo per il trasferimento di dati basato su TCP. E' uno dei primi protocolli ad esser stato implementato. Utilizza due connessioni differenti per gestire i dati e i comandi: il server FTP rimane in ascolto sulla porta 21 a cui si connette un client, la cui connessione determina l'apertura di un canale di dati sulla porta 20 per la trasmissione.

Per esigenze di simulazione viene preso in considerazione solamente il flusso dati essendo la connessione di controllo di dimensioni molto limitate e non costante nel tempo.

- **Voice over IP (VoIP)**: in realtà non è un protocollo, ma una famiglia di tecnologie che permettono di effettuare chiamate telefoniche su reti a commutazione di pacchetto; queste si basano su vari protocolli, come ad esempio l'UDP per la trasmissione veloce dei dati vocali, il TCP per la trasmissione dei dati di controllo, oltre a tutta una serie di algoritmi proprietari per applicazioni specifiche, come *Skype*.

5.1.3 Dati prodotti

Essenzialmente i risultati di output sono stati raccolti in due modi differenti:

- **Goodput**: è la quantità di dati utili trasferiti nell'unità di tempo, su un canale fisico, tra due nodi, generalmente misurata in b/s (o multipli: kb/s, kB/s ecc.).
  Nello specifico lo si ottiene dal throughput totale, da cui viene sottratto il traffico riguardante le informazioni di servizio e di segnalazione (*overhead*) associate ai vari protocolli dello stack TCP/IP (ad esempio i pacchetti di ACK del protocollo TCP). In questo modo è possibile calcolare il *payload*, da cui è possibile ricavare le prestazioni effettive di una rete.
- **Grafici**: generati grazie a Wireshark dai file .pcap prodotti da ns-3.

5.2 Integrazione algoritmi di Packet Scheduling

5.2.1 Topologia di test della rete

Il problema su cui ci si concentra e sui cui si applicano tecniche di Packet Scheduling è l'invio, su un link di larghezza di banda limitata, dei dati di due applicazioni di cui una fortemente aggressiva, poiché si basa sul protocollo UDP, che tenderà quindi a trasmettere alla maggiore velocità possibile a scapito degli altri flussi.

Viene presa in esame una rete la cui topologia è schematizzata nella figura 5.1.

Si è cercato da un lato di creare una rete semplice e funzionale ai fini della simulazione, dall'altro di caratterizzare realisticamente la rete, assegnando per esempio al server FTP una banda molto maggiore di quella che può essere la banda di un utente dotato di tradizionale ADSL.
Il nodo 0 è un server su cui è installata una applicazione FTP per il trasferimento dei file, collegato a una rete con una banda di 100 Mbps; il nodo 1 è

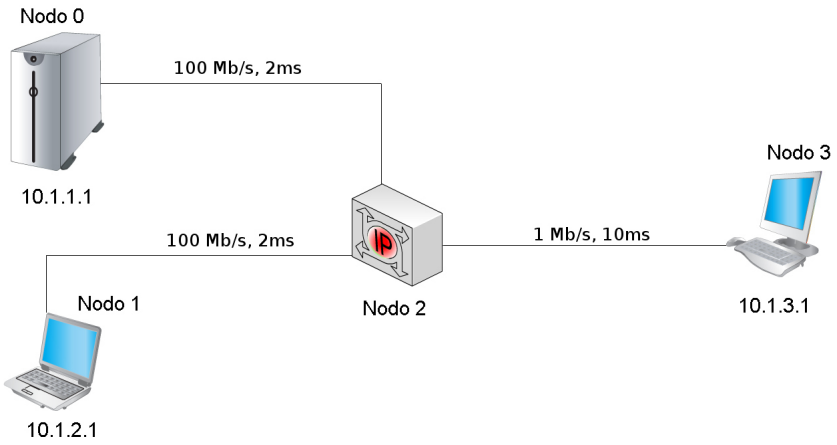

Fig. 5.1: Topologia della rete per analisi di Packet Scheduling

un utente che, collegato a una rete da 100 Mbps, effettua una chiamata VoIP verso il nodo 3, a sua volta collegato tramite una rete da 1 Mbps. Il nodo 2, un router IP, svolge funzioni di commutazione mettendo in comunicazione le tre reti.

La criticità che si presenta è legata al fatto che la larghezza di banda molto limitata in ricezione del nodo 3 deve ospitare due connessioni il cui rate di trasmissione (10 Mbps per la VoIP e 10 Mbps per FTP) supera abbondantemente la banda disponibile di 1 Mbps. Inoltre i dati della trasmissione VoIP vengono trasportati tramite protocollo UDP e quindi, a differenza delle connessioni TCP dotate di controllo di flusso e di congestione, sono trasmessi alla massima velocità possibile a scapito delle altre connessioni, correndo il rischio di interromperle.
Le due reti caratterizzate da banda di 100 Mbps sono state pensate soprattutto per la simulazione, in modo tale da poter chiaramente osservare l'azione del packet scheduler, senza che questa sia influenzata da altri fattori; inoltre l'elevata banda disponibile, unito al *sendrate* da 10 Mbps delle due applicazioni, permettono al packet scheduler di lavorare in condizioni *backlogged*.

5.2.2 Implementazione della topologia

Creazione dei nodi della rete:

```
NodeContainer c;
c.Create (4);
NodeContainer n0n2 = NodeContainer (c.Get (0),
      c.Get (2));
NodeContainer n1n2 = NodeContainer (c.Get (1),
      c.Get (2));
```

```
  NodeContainer n3n2 = NodeContainer (c.Get (3),
        c.Get (2));

  InternetStackHelper internet;
  internet.Install (c);
```

Creazione dei canali; da notare i diversi valori di *datarate* e *delay* attribuiti ai canali. In base ai parametri passati da terminale sul terzo canale viene attivato uno dei possibili meccanismi di scheduling:

```
  NS_LOG_INFO ("Create channels.");
  PointToPointHelper p2p;

  // Canale server<-->router
  p2p.SetDeviceAttribute ("DataRate",
        StringValue ("100Mbps"));
  p2p.SetChannelAttribute ("Delay",
        StringValue ("2ms"));
  NetDeviceContainer d0d2 = p2p.Install (n0n2);

  // Canale utente1<-->router
  p2p.SetDeviceAttribute ("DataRate",
        StringValue ("100Mbps"));
  p2p.SetChannelAttribute ("Delay",
        StringValue ("2ms"));
  NetDeviceContainer d1d2 = p2p.Install (n1n2);

   // Canale utente2<-->router
  if (queueType == "PRIO")
         p2p.SetQueue ("ns3::PRIO");
  else if (queueType == "RR")
         p2p.SetQueue("ns3::RR");
  else if (queueType == "WRR")
         p2p.SetQueue("ns3::WRR");

  p2p.SetDeviceAttribute ("DataRate",
        StringValue ("1Mbps"));
  p2p.SetChannelAttribute ("Delay",
        StringValue ("10ms"));
  NetDeviceContainer d3d2 = p2p.Install (n3n2);
```

Creazione delle applicazioni, viene modellato il mittente e il ricevente per ogni trasmissione di pacchetti. Come anticipato, per semplicità non viene considerata la connessione TCP di controllo per il protocollo FTP e per la chiamata VoIP.

```
// Definizione porte
  uint16_t port_ftp = 20;          // FTP data
  uint16_t port_voip = 6000;       // VoIP data

  // Mittente FTP (server su nodo 0)
```

```
OnOffHelper onoff_ftp("ns3::TcpSocketFactory",
      Address(InetSocketAddress (i3i2.GetAddress (0),
         port_ftp)));
onoff_ftp.SetConstantRate (DataRate ("10Mbps"));
ApplicationContainer ftp_server =
      onoff_ftp.Install (c.Get (0));
ftp_server.Start(Seconds(0.0));
ftp_server.Stop(Seconds(10.0));

// Destinatario FTP (client su nodo 3)
PacketSinkHelper sink_ftp ("ns3::TcpSocketFactory",
      Address (InetSocketAddress (Ipv4Address::GetAny
         (), port_ftp)));
ApplicationContainer ftp_client =
      sink_ftp.Install (c.Get (3));
ftp_client.Start (Seconds (0.0));
ftp_client.Stop (Seconds (10.0));

// Chiamante VoIP (client su nodo 1)
OnOffHelper onoff ("ns3::UdpSocketFactory",
      Address(InetSocketAddress (i3i2.GetAddress (0),
         port_voip)));
onoff.SetConstantRate (DataRate ("10Mbps"));
onoff.SetAttribute ("Remote",
      AddressValue (InetSocketAddress (i3i2.GetAddress
         (0), port_voip)));
ApplicationContainer app_voip = onoff.Install (c.Get
   (1));
app_voip.Start (Seconds (0.0));
app_voip.Stop (Seconds (10.0));

 // Ricevente VoIP (client su nodo 3)
PacketSinkHelper sink_voip ("ns3::UdpSocketFactory",
      Address (InetSocketAddress (Ipv4Address::GetAny
         (), port_voip)));
app_voip = sink_voip.Install (c.Get (3));
app_voip.Start (Seconds (0.0));
app_voip.Stop (Seconds (10.0));
```

5.2.3 Algoritmi di Packet Scheduling

Tutti gli algoritmi di Packet Scheduling condividono una parte comune di codice. Risulta infatti necessario, ad eccezione dell'algoritmo di droptail, classificare il pacchetto che giunge allo scheduler recuperando dall'header il protocollo (nel nostro caso o TCP o UDP).
Il recupero del protocollo è necessario per inserire il pacchetto nella giusta coda.

```
PacketMetadata::ItemIterator metadataIterator = q->
   BeginItem();
PacketMetadata::Item item;

while (metadataIterator.HasNext()){
        item = metadataIterator.Next();

        //Per ottenere l'header IP
        if(item.tid.GetName() == "ns3::Ipv4Header"){
                Callback<ObjectBase *> constr = item.tid.
                   GetConstructor();
                NS_ASSERT(!constr.IsNull());

                ObjectBase *instance = constr();
                NS_ASSERT(instance != 0);

                Ipv4Header* ipv4Header = dynamic_cast<
                   Ipv4Header*> (instance);
                NS_ASSERT(ipv4Header != 0);

                ipv4Header->Deserialize(item.current);

                // Dall'header estraiamo il protocollo
                protocol = ipv4Header->GetProtocol();

                // Fine. Pulizia dell'header
                delete ipv4Header;
                break;
        }
}
```

Le code vengono modellate attraverso delle *list*.

```
std::list<Ptr<Packet> > udp_queue;
std::list<Ptr<Packet> > tcp_queue;
```

I pacchetti vengono inseriti nelle code con una semplice operazione di push:

```
if(protocol==TCP)
        tcp_queue.push_back (p);
else
        udp_queue.push_back(p);
```

In alcuni protocolli si ha un'ulteriore distinzione dei pacchetti in quanto vengono presi in considerazione anche gli ACK del protocollo TCP: questi vengono individuati con un semplice controllo sulla lunghezza:

```
if(protocol==TCP) {
        if(p->GetSize() < 60) {
                tcp_ack_queue.push_back(p);
        }
}
```

DROPTAIL

L'implementazione di default della gestione dei pacchetti, che in ns-3 è appunto droptail, mette in evidenza le problematiche sopra esposte.
Come si può vedere nel grafico 5.2, la connessione VoIP (colore verde) prende possesso dell'intero canale andando a sopprimere di fatto la connessione FTP (colore rosso). Nel grafico vengono mostrati anche i pacchetti di ack del protocollo TCP (colore blu).

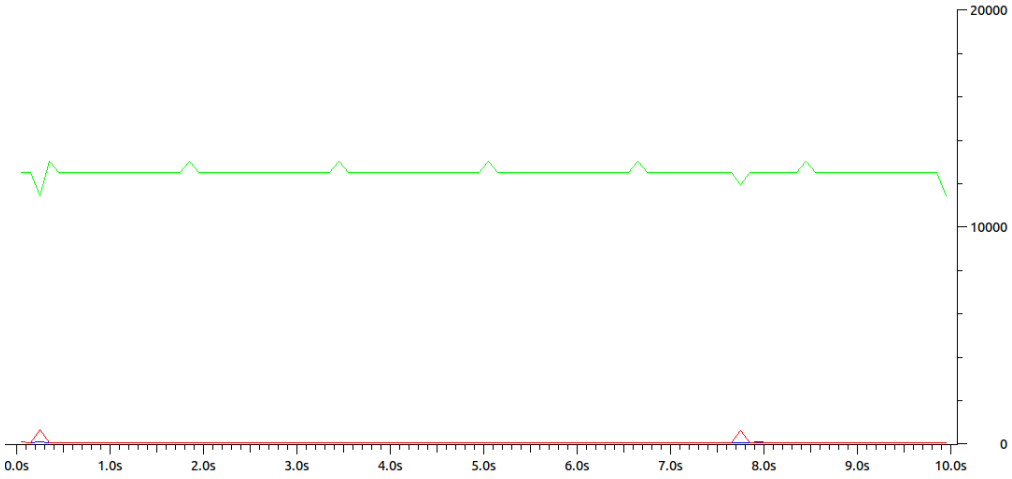

Fig. 5.2: Packet Scheduling: Droptail

I valori del goodput (fig. 5.3) confermano quanto visto nel grafico: la connessione FTP ha un goodput pressochè nullo, dal momento che il protocollo UDP (utilizzato dall'applicazione VoIP) utilizza l'intera banda senza preoccuparsi dell'equa suddivisione del canale.

```
Simulazione PS: DROPTAIL

Network topology:
  n0
    \ 100 Mb/s, 2ms
     \           1Mb/s, 10ms
      n2 ------------------------ n3
     /
    / 100 Mb/s, 10ms
  n1

Goodput FTP: 0.1072 kB/s
Goodput VOIP: 117.811 kB/s
```

Fig. 5.3: Packet Scheduling: Droptail

PRIO

Se si vuole proteggere il flusso dati TCP da quello UDP e garantirgli almeno un funzionamento minimo, occorre quindi implementare una qualche politica di scheduling dei pacchetti. Una prima soluzione può essere quella di adottare un algoritmo di tipo prioritario, che vada a dare la precedenza ai pacchetti TCP (su cui è basata l'applicazione FTP) rispetto ai pacchetti UDP.
Il codice che implementa questa soluzione risulta essere:

```
if (tcp_queue.empty ()){

        if (udp_queue.empty ()){
                return 0;
        }
        else {
                p = udp_queue.front ();
                udp_queue.pop_front ();
        }

}
else {
        p = tcp_queue.front ();
        tcp_queue.pop_front ();
}

return p;
```

Si può notare come il primo controllo per l'invio del pacchetto venga eseguito proprio sulla coda TCP e solo nel caso in cui questa sia vuota si tenta di inviare un pacchetto UDP.
Tuttavia, come si può notare in figura 5.4, l'uso dell'algoritmo PRIO rende sì possibile la connessione FTP (colore rosso), ma va ad uccidere la connessione VoIP (colore verde): si cade nel fenomeno di *starvation* descritto precedentemente, ovvero il possibile blocco di invio dei pacchetti di una determinata classe con priorità bassa.
In blu i pacchetti ack del protocollo TCP.

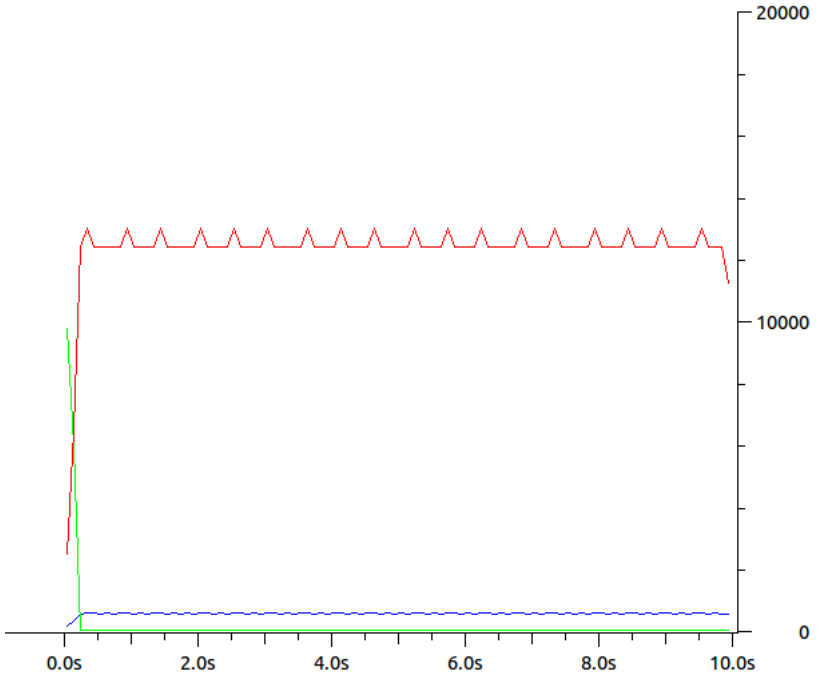

Fig. 5.4: Packet Scheduling: PRIO

I valori del goodput confermano quanto visto nel grafico: in assenza di politiche di packet scheduling *fair* l'unica ad inviare dati risulta essere la connessione FTP, a scapito della chiamata VoIP.

```
Simulazione PS: PRIO

Network topology:
  n0
    \ 100 Mb/s, 2ms
     \           1Mb/s, 10ms
      n2 ------------------------ n3
     /
    / 100 Mb/s, 10ms
  n1

Goodput FTP: 111.97 kB/s
Goodput VOIP: 1.4848 kB/s
```

Fig. 5.5: Packet Scheduling: PRIO

RR

Una soluzione che appare subito essere valida è quella del Round Robin: l'idea è quella di andare infatti a garantire entrambe le connessioni suddividendo in maniera *fair* le risorse limitate della rete.
In questo primo approccio non vengono tenute in considerazione le specifiche esigenze di ciascuna connessione, ovvero, ad esempio, il valore minimo di goodput che la chiamata VoIP deve avere per essere di buona qualità.
L'algoritmo viene implementato come segue:

```
if(turn > 0) {
        if (tcp_queue.empty ()){
                if (udp_queue.empty ()){
                        return 0;
                }
                else {
                        p = udp_queue.front ();
                        udp_queue.pop_front ();
                }
        }
        else {
                p = tcp_queue.front ();
                tcp_queue.pop_front ();
        }
}
else {
        if (udp_queue.empty ()){
                if (tcp_queue.empty ()){
                        return 0;
                }
                else {
                        p = tcp_queue.front ();
                        tcp_queue.pop_front ();
                }
        }
        else {
                p = udp_queue.front ();
                udp_queue.pop_front ();
        }
}

turn = turn *(-1);
return p;
}
```

Da notare la presenza di una variabile (*turn*) che assegna periodicamente il turno di invio ad una delle due code.

Come si nota dalla successiva figura 5.6, in questo caso i due flussi si spartiscono equamente il canale. Ciò era facilmente prevedibile, in quanto entrambe le applicazioni utilizzano la stessa *packet size* e l'algoritmo di Round Robin trasmette alternativamente dalle due code in modo circolare.

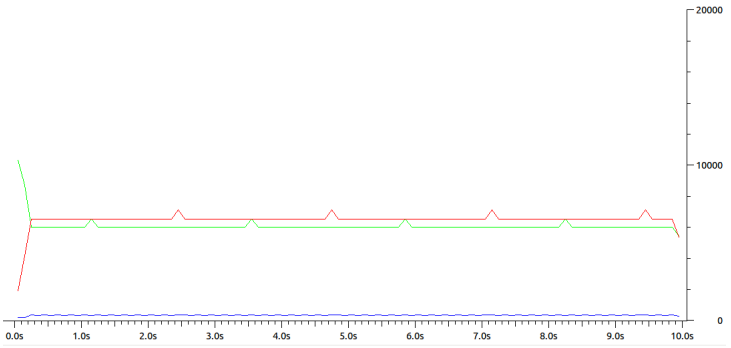

Fig. 5.6: Packet Scheduling: Round Robin

Le stesse conclusioni si possono trarre a partire dall'output della simulazione, riportato in figura 5.7: si nota infatti che il goodput FTP e quello VoIP sono molto vicini.

```
Simulazione PS: RR

Network topology:
  n0
    \ 100 Mb/s, 2ms
     \           1Mb/s, 10ms
      n2 ------------------------- n3
     /
    / 100 Mb/s, 10ms
  n1

Goodput FTP: 58.424 kB/s
Goodput VOIP: 57.1392 kB/s
```

Fig. 5.7: Packet Scheduling: Round Robin

Nonostante tale risultato sia sicuramente positivo, dal momento che rappresenta una condivisione del canale più equa rispetto ai casi precedenti, si nota che con questo algoritmo è impossibile ogni tipo di personalizzazione della condivisione del canale da parte delle due applicazioni.
In altre parole, poiché due flussi in contesa trasmettono una volta per uno pacchetti di uguale dimensione, sicuramente convergeranno verso una situazione di utilizzo paritario della banda.
Ciò non è sempre desiderabile, ma per ovviare a questo problema occorre perfezionare l'algoritmo di scheduling verso una versione pesata, che sarà presentata immediatamente a seguire.

WRR

Come anticipato nella sezione precedente, l'obiettivo che si vorrebbe raggiungere è quello di assegnare porzioni diverse di banda alla connessione TCP e a quella UDP, senza che nessuna delle due precluda il corretto funzionamento dell'altra: a questo scopo si è quindi implementato Weighted Round Robin.
Per prima cosa, sono assegnati pesi differenti ad ogni protocollo (quindi nel nostro caso a ogni connessione):

```
w_tcp_ack = 0.05;
w_udp = 0.7;
w_tcp =  0.25;
```

Notiamo che al traffico UDP è assegnato un peso triplo rispetto a quello assegnato al traffico TCP, perciò è ragionevole aspettarsi che il goodput della prima connessione sia circa il triplo di quello della seconda.
Lo scheduler è stato così costruito:

```
while(ok==0){

        x = r->GetValue();

        if(x < w_tcp_ack) {
                if(!tcp_ack_queue.empty()){
                        p = tcp_ack_queue.front ();
                        tcp_ack_queue.pop_front ();
                        ok = 1;
                }
        }
        else if ( x >= w_tcp_ack && x < (w_tcp_ack+w_udp)
             ) {
                if (!udp_queue.empty ()){
                        p = udp_queue.front ();
                        udp_queue.pop_front ();
                        ok = 1;
                }
        }
        else {
                if (!tcp_queue.empty ()){
                        p = tcp_queue.front ();
                        tcp_queue.pop_front ();
                        ok = 1;
                        }
        }

        if(tcp_queue.empty () && udp_queue.empty () &&
           tcp_ack_queue.empty ())
                                break;
}
```

L'implementazione prevede che ad ogni invocazione della funzione *DeQueue*, che ha il compito di mandare in output i pacchetti, si abbia un ciclo all'interno del quale si effettua un'estrazione per decidere quale coda servire; successivamente si tenta l'invio di un pacchetto della coda estratta, in caso di coda vuota si procede a una nuova estrazione.
Nello specifico l'estrazione consiste nel settare una variabile aleatoria il cui valore ricade all'interno di uno dei tre range associati alle code; ciascun range di valore sarà più o meno grande a seconda del peso iniziale assegnato.

Grazie al grafico in figura 5.8 si può notare come l'occupazione della banda del canale da parte delle due applicazioni sia proporzionale al peso associato (colore rosso per il protocollo TCP, verde per l'UDP e blu per gli ack del TCP).

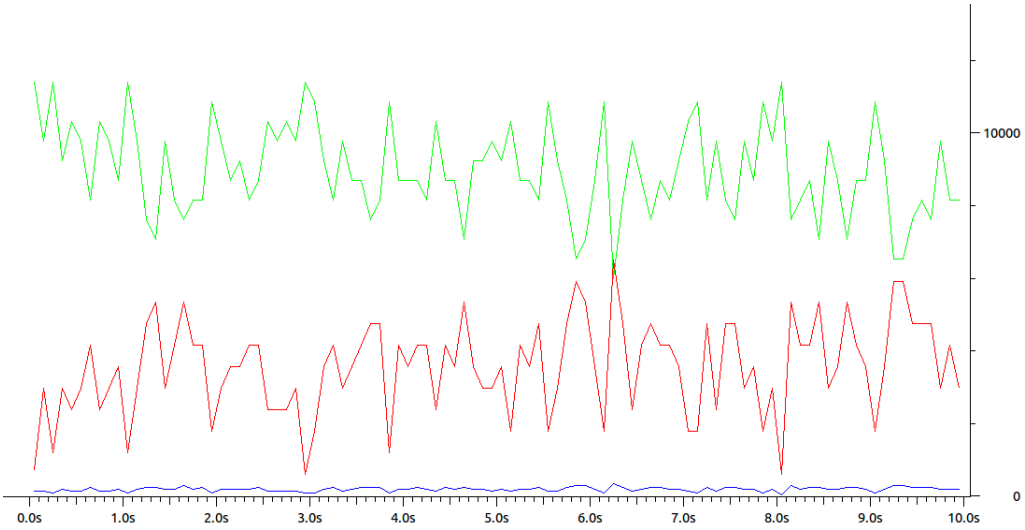

Fig. 5.8: Packet Scheduling: Weighted Round Robin

I valori del goodput ben confermano quanto visto a livello grafico: il valore associato al TCP è poco più di 1/3 rispetto a quello dell'applicazione VoIP (figura 5.9).

Il caso ora trattato dimostra che è possibile assegnare in maniera specifica la banda di un canale di comunicazione, senza precludere la corretta trasmissione dei dati delle applicazioni.
Si è cercato di assegnare i valori in modo tale da suddividere la banda in base alle necessità reali delle applicazioni: nel nostro caso il vincolo è rappresentato dalla chiamata VoIP che deve avere un valore di goodput di almeno 64 kB/s per avere una chiamata di buona qualità; per la connessione TCP si è cercato solamente di rallentarla senza bloccarla.

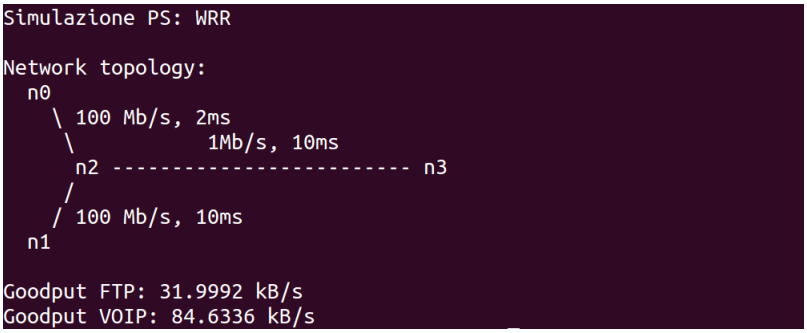

Fig. 5.9: Packet Scheduling: Weighted Round Robin

5.2.4 Casi di studio

Qui di seguito vengono trattati alcuni casi di studio, ovvero analisi di comportamento degli scheduler e della rete in presenza di parametri e settaggi differenti dai casi visti in precedenza.
Obiettivo di questo paragrafo lo studio di nuovi scenari e il testing del funzionamento degli scheduler implementati.

PRIO con priorità al protocollo UDP

Nel primo caso di studio si è preso in esame il protocollo PRIO: in particolare, a differenza del caso sopra esposto, si è data la priorità al protocollo UDP. Come prevedibile il canale è stato completamente occupato dalla connessione UDP, mentre la connessione TCP, complice la pressochè totale mancanza di arrivo dei pacchetti di ack, non è riuscita a trasmettere nessun dato.
In un contesto reale molto probabilmente non si sarebbe completato neppure il *three-way handshaking* del protocollo TCP.

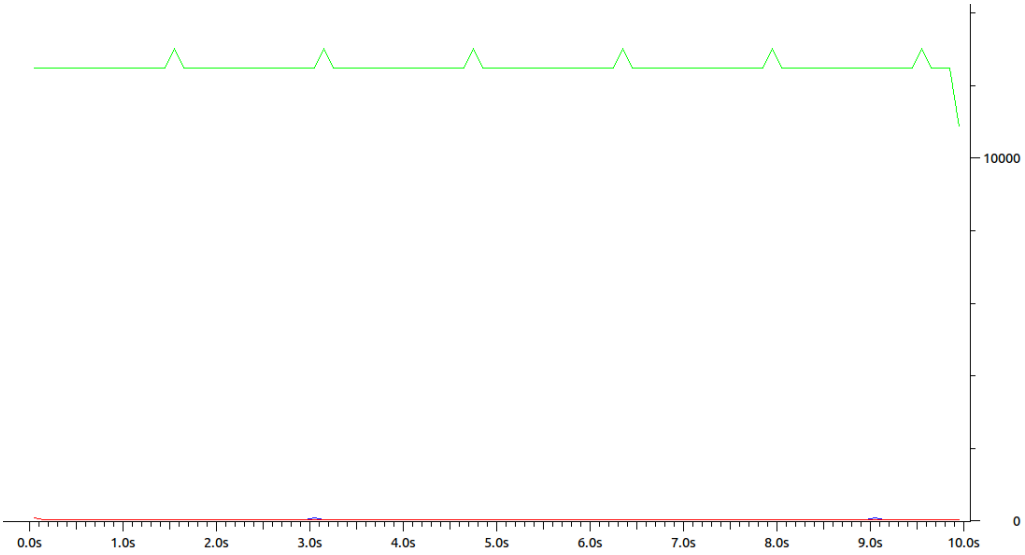

Fig. 5.10: PRIO: priorità al protocollo UDP

I valori dei due goodput confermano quanto visto a livello di grafico. Da notare che, a differenza del precedente caso di applicazione del PRIO, la connessione a bassa priorità muore per *starvation* e non riesce a trasmettere con successo nemmeno un singolo pacchetto, facendo registrare un valore di goodput di 0 kB/s.

```
Simulazione PS: PRIO

Network topology:
  n0
    \ 100 Mb/s, 2ms
     \           1Mb/s, 10ms
      n2 ------------------------- n3
     /
    / 100 Mb/s, 10ms
  n1

Goodput FTP: 0 kB/s
Goodput VOIP: 117.914 kB/s
```

Fig. 5.11: Goodput PRIO: priorità al protocollo UDP

WRR con pesi differenti

Un ulteriore interessante caso di studio consiste nel poter dare pesi differenti alle varie classi di pacchetti all'interno dell'algoritmo Weighted Round Robin.
Nel caso precedente i valori dei pesi erano assegnati in modo tale da garan-

tire una corretta trasmissione dei dati dell'applicazione VoIP.
In questo caso assumiamo come pesi:

```
w_tcp_ack = 0.6;
w_udp = 0.1;
w_tcp =  0.3;
```

Si vuole così dare maggiore peso al protocollo TCP, il triplo rispetto al protocollo UDP; il fatto che agli ACK della connessione TCP sia assegnato un peso considerevole significa che non appena accodati saranno smaltiti molto rapidamente.
Essendo il *packetsize* e il *rate* di trasmissione delle due applicazioni identici, ci si aspetterebbe che il traffico UDP sia circa un terzo del traffico TCP.
Il grafico in figura 5.12 conferma le nostre ipotesi.

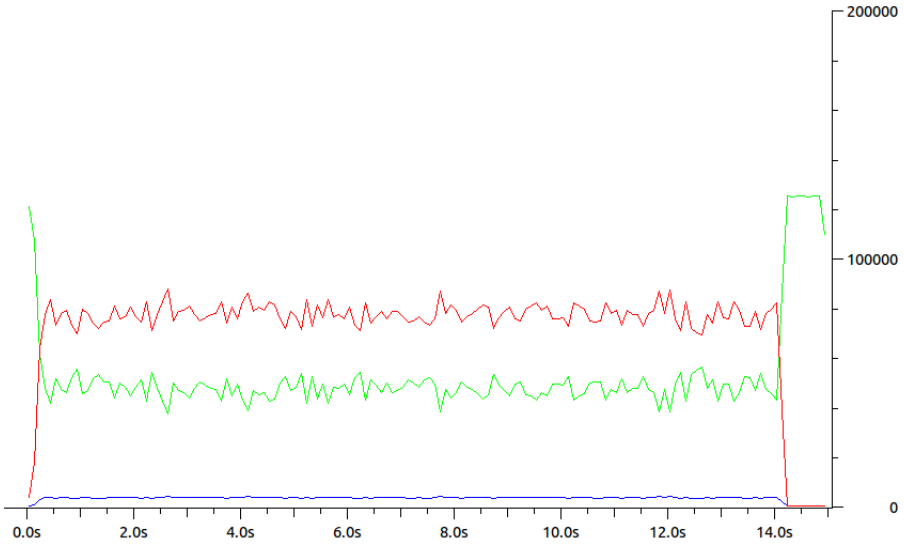

Fig. 5.12: WRR: peso maggiore al traffico TCP

A livello di dati numerici appare ancora più evidente il rapporto dei dati inviati dalle applicazioni VoIP e FTP.

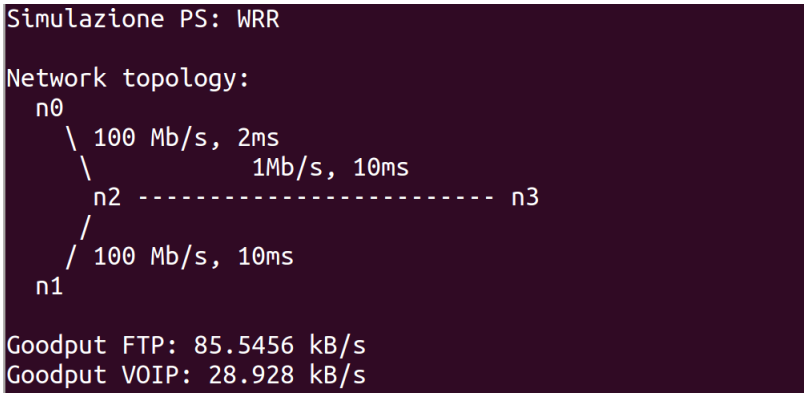

Fig. 5.13: Goodput WRR: peso maggiore al traffico TCP

5.3 Integrazione algoritmi di AQM

5.3.1 Topologia di test della rete

Come nel precedente caso del Packet Scheduling, anche per verificare l'efficacia di una politica di AQM si è deciso di semplificare al massimo la topologia di test: ciò ha avuto il duplice vantaggio di minimizzare le interferenze di elementi superflui e dare invece importanza a quelli utili a comprovare o meno il funzionamento dell'Active Queue Management.

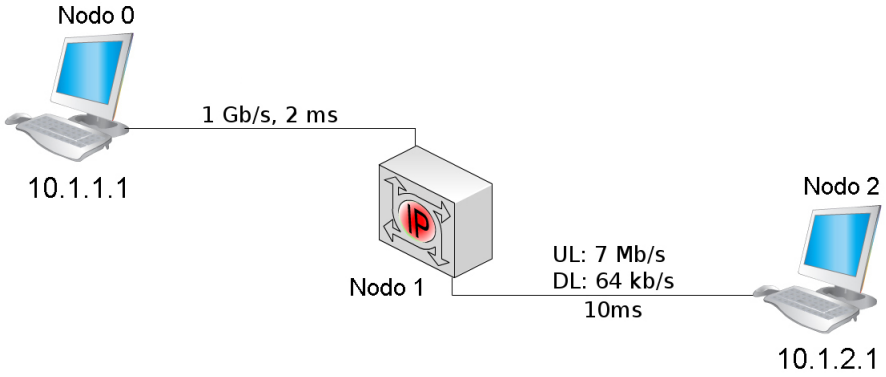

Fig. 5.14: Topologia della rete per analisi dell'Active Queue Management

Abbiamo quindi creato una rete minimale formata tra tre nodi, nella quale un nodo centrale con funzionalità di router mette in collegamento i due endpoint. Entrambi i nodi agli estremi della topologia hanno installato sia un server che un client FTP: il nodo 0 (vedi figura 5.2: di seguito i nodi saranno chiamati con la nomenclatura presente in figura) cerca di scaricare un file dal server presente sul nodo 2, e viceversa.
Per una maggiore chiarezza di esposizione, si introducono anche le seguenti definizioni che saranno utilizzate nei successivi paragrafi della trattazione:

- Connessione FTP1: si indica il trasferimento di dati tra il server FTP presente sul nodo 0 e il client FTP presente sul nodo 2.
- Connessione FTP2: si indica il trasferimento di dati tra il server FTP presente sul nodo 2 e il client FTP presente sul nodo 0.

Il problema in questo caso nasce dal fatto che nonostante le due applicazioni server cerchino di trasmettere ad identica velocità, pari a 5Mbps, la capacità dei canali di comunicazione tra i due nodi e il router è molto diversa: il nodo 0 è infatti collegato al router da un canale ad alta velocità (1Gbps, 2ms di ritardo), mentre tra il router e il nodo 2 è modellato un semplice canale di tipo ADSL, con velocità di upload estremamente limitata (7Mbps DL, 64kbps UL, 10ms di ritardo), il che causa un vero e proprio collo di bottiglia

per entrambe le comunicazioni.
Il fatto che la connessione FTP2 abbia un throughput molto basso non sorprende, a causa dell'esigua capacità di upload del canale con cui il nodo 2 è collegato al router. Tuttavia i problemi non si fermano qui: accade infatti anche che gli ACK della connessione FTP1 rimangono intasati nel collo di bottiglia e non riescono a raggiungere il nodo 0, causando continue cadute della finestra di trasmissione TCP e conseguente crollo del throughput di FTP1 nonostante il link di capacità pressoché infinita a disposizione del server sul nodo 0.

L'obiettivo quindi in questo caso è implementare una politica di AQM sulla coda di pacchetti in uscita dal nodo 2. Una gestione "intelligente" del buffer di upload del nodo 2 dovrebbe quindi privilegiare i pacchetti di ACK di FTP1 rispetto ai dati FTP2, in modo da aumentare almeno il throughput della connessione FTP1, anzichè farle crollare entrambe come nel caso di assenza di AQM.

5.3.2 Implementazione della topologia

Creazione dei tre nodi necessari:

```
NS_LOG_INFO ("Create nodes.");
NodeContainer c;
c.Create (3);
NodeContainer n0n1 = NodeContainer (c.Get (0), c.Get
    (1));
NodeContainer n2n1 = NodeContainer (c.Get (2), c.Get
    (1));

InternetStackHelper internet;
internet.Install (c);
```

Creazione dei canali, da notare i diversi valori di *datarate* e *delay* che li caratterizzano:

```
PointToPointHelper p2p1;

// Canale nodo1<-->router
p2p1.SetDeviceAttribute("DataRate",
      StringValue ("1Gbps"));
p2p1.SetChannelAttribute("Delay",
      StringValue ("2ms"));
NetDeviceContainer d0d1 = p2p1.Install (n0n1);

PointToPointHelper p2p2;
// Canale router<-->nodo2
p2p2.SetDeviceAttribute ("DataRate",
      StringValue ("7Mbps"));
p2p2.SetChannelAttribute ("Delay",
```

```
        StringValue ("10ms"));
```

Creazione della coda gestita con AQM (RED); in termini stretti di programmazione in ns-3, l'implementazione di algoritmi di AQM consiste nel settare la coda nella tipologia voluta, in questo caso ovviamente RED, fornendo anche i parametri fondamentali MinTh, MaxTh (per il loro significato si rimanda al paragrafo *Algoritmi di AQM - Random Early Detection (RED)*).

```
if (queueType == "RED") {
          double      minTh = 20;
          double      maxTh = 80;
          p2p2.SetQueue ("ns3::RedQueue",
                "MinTh", DoubleValue (minTh),
                "MaxTh", DoubleValue (maxTh),
                "LinkBandwidth", StringValue ("7Mbps"),
                "LinkDelay", StringValue ("10ms"));
 }
```

Modellazione delle applicazioni: entrambe iniziano a trasmettere dati all'istante 0s, mentre i client sono in ascolto entrambi fino a 12s.

```
  //Mittente FTP (server su nodo 0)
  OnOffHelper onoff_ftp1("ns3::TcpSocketFactory",
         Address(InetSocketAddress (i2i1.GetAddress (0),
            port_ftp)));
  onoff_ftp1.SetConstantRate (DataRate ("5Mbps"));
  [...]
  ApplicationContainer ftp_server1 = onoff_ftp1.Install (
     c.Get (0));
  ftp_server1.Start(Seconds(0.0));
  ftp_server1.Stop(Seconds(10.0));

  // Destinatario FTP (client su nodo 2)
  PacketSinkHelper sink_ftp1 ("ns3::TcpSocketFactory",
        Address (InetSocketAddress (Ipv4Address::GetAny
           (), port_ftp)));
  ApplicationContainer ftp_client1 = sink_ftp1.Install (c
     .Get (2));
  ftp_client1.Start (Seconds (0.0));
  ftp_client1.Stop (Seconds (12.0));

  // Mittente FTP (server su nodo 2)
  OnOffHelper onoff_ftp2("ns3::TcpSocketFactory",
        Address(InetSocketAddress (i0i1.GetAddress (0),
           port_ftp2)));
  onoff_ftp2.SetConstantRate (DataRate ("5Mbps"));
  [...]
  ApplicationContainer ftp_server2 = onoff_ftp2.Install (
     c.Get (2));
  ftp_server2.Start(Seconds(0.0));
  ftp_server2.Stop(Seconds(10.0));
```

```
// Destinatario FTP (client su nodo 0)
PacketSinkHelper sink_ftp2 ("ns3::TcpSocketFactory",
      Address (InetSocketAddress (Ipv4Address::GetAny
         (), port_ftp2)));
ApplicationContainer ftp_client2 = sink_ftp2.Install (c
   .Get (0));
ftp_client2.Start (Seconds (0.0));
ftp_client2.Stop (Seconds (12.0));
```

5.3.3 Implementazione dell'algoritmo RED

A differenza del caso del Packet Scheduling, l'ambiente di ns-3 ospitava già una classe di implementazione dell'algoritmo RED. Le nostre attenzioni quindi si sono concentrate sulla sua localizzazione e sul suo corretto impiego, in quanto, purtroppo, la documentazione ufficiale di ns-3 spesso è risultata carente.

Le due classi prese in esame quindi sono state *red-queue.cc* e *red-queue.h*, entrambe localizzate in *src/network/utilis*. Nello specifico sono due sottoclassi che specializzano la classe generale *queue.cc*. Per il loro utilizzo si rimanda al codice sopra riportato, in cui è mostrata la loro chiamata.

5.3.4 Risultati

I dati sono risultati essere concordi con le nostre aspettative.
Si può facilmente notare nell'immagine 5.15 come la trasmissione FTP1 (colore rosso) non riesca sostanzialmente ad avere una velocità adeguata con una trasmissione lineare dei pacchetti a causa della concomitanza con la trasmissione FTP2 (colore verde); si possono notare numerosi picchi di invio dati in corrispondenza dell'arrivo di qualche pacchetto di ACK (colore blu). Il risultato che si ottiene è un forte rallentamento per entrambe le connessioni, in maniera particolare per FTP1 che dovrebbe avere valori di throughput elevati.

I valori di goodput che si ottengono confermano quanto descritto: il collo di bottiglia determina il crollo di entrambe le trasmissioni delle applicazioni; in un contesto reale questo si traduce molto probabilmente nella chiusura della connessione FTP2 e in un'estrema lentezza della connessione FTP1, uno scenario quindi da evitare.

Con l'utilizzo invece di RED per la gestione delle code si hanno generali miglioramenti. Come si può osservare in figura 5.22 la trasmissione FTP1 (colore rosso) risulta si ancora ostacolata dalla presenza di un sovraccarico della rete, ma si può notare come dopo un periodo di assestamento dell'algoritmo RED la connessione FTP1 mostra picchi di traffico molto più elevati

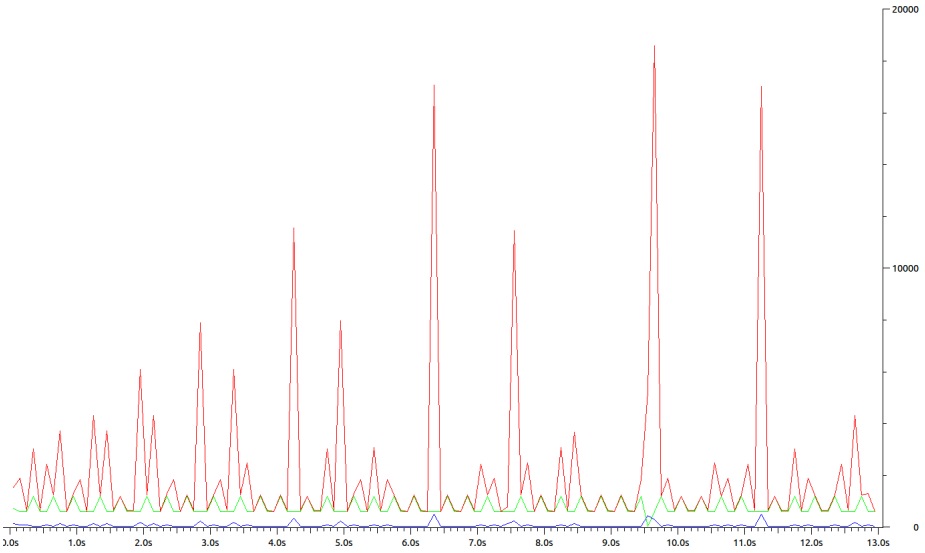

Fig. 5.15: Output dell'esecuzione dello script - Droptail

```
Simulazione AQM: DROPTAIL

Network topology:
n0 -------------------------- n1 -------------------------- n2
            1Gbps, 2ms              7Mbps DL, 64kbps UL, 10ms

ID bottleneck: 2
Goodput FTP1 (n0 --> n2): 8.32862 kB/s
Goodput FTP2 (n2 --> n0): 6.84431 kB/s
```

Fig. 5.16: Output dell'esecuzione dello script - Droptail

e compatti: in parole povere il suo trasferimento dati risulta essere notevolmente migliorato sotto il profilo della velocità.
La connessione FTP2 (colore verde) continua ad avere un throughput relativamente basso, però in questo caso, come precedentemente detto, non è importante la sua velocità di trasmissione ma il fatto che non influenzi in modo negativo la più veloce connessione FTP1.
I valori di goodput che si ottengono confermano quanto appena descritto.

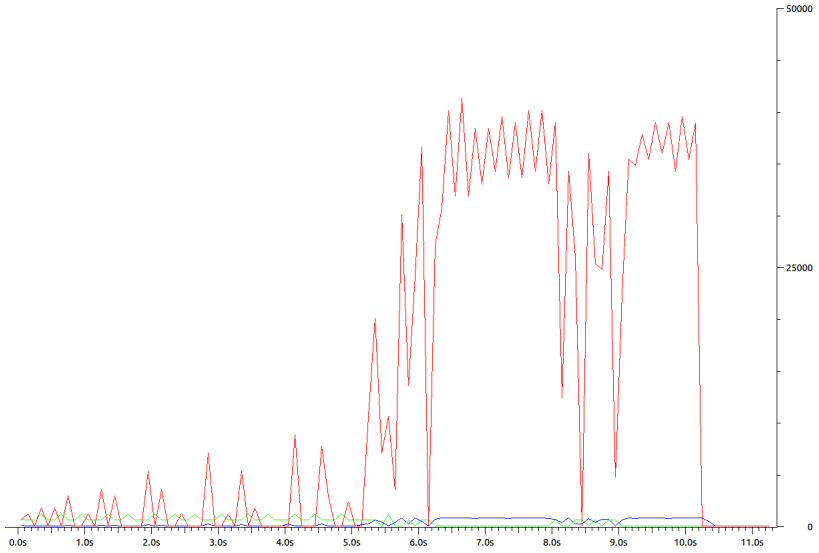

Fig. 5.17: Output dell'esecuzione dello script - AQM RED

```
Simulazione AQM: RED

Network topology:
n0 -------------------------- n1 -------------------------- n2
            1Gbps, 2ms              7Mbps DL, 64kbps UL, 10ms

ID bottleneck: 2
Goodput FTP1 (n0 --> n2): 106.496 kB/s
Goodput FTP2 (n2 --> n0): 2.68 kB/s
```

Fig. 5.18: Output dell'esecuzione dello script - AQM RED

5.3.5 Caso di studio: connessioni che iniziano in momenti diversi

Per affrontare la questione da un altro punto di vista, si è modificato il comportamento delle applicazioni in modo che comincino a trasmettere in due momenti diversi: in questo caso FTP1 invia dati fin dall'istante 0s, mentre FTP2 si aggiunge all'istante 4s.
Vogliamo dunque verificare che, nonostante il piccolo cambiamento di condizioni iniziali, i risultati della simulazione siano in linea con quelli precedentemente ottenuti confermando quindi che l'utilizzo di AQM porta effettivi benefici alla rete.

Vediamo cosa succede in assenza di AQM. Dal momento che l'applicazione FTP1 comincia immediatamente a trasmettere, nella fase iniziale ha un throughput piuttosto elevato. Tuttavia non appena parte FTP2, entrambe le connessioni crollano per i motivi esposti nella sezione precedente.

Segue il grafico di Wireshark in cui si evidenzia questo comportamento:

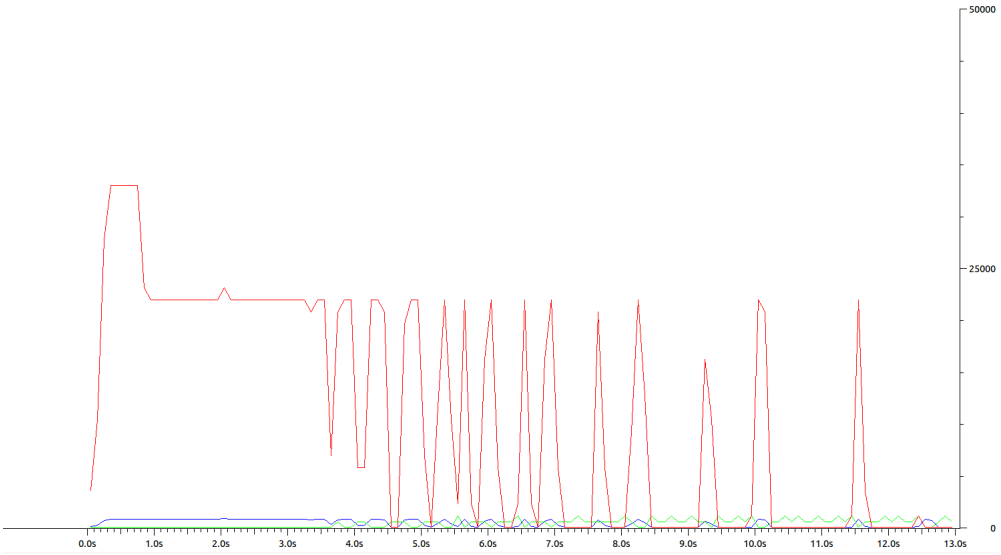

Fig. 5.19: Output dell'esecuzione dello script - Droptail

Analizzando invece l'output della simulazione, notiamo che nonostante il crollo avvenuto all'istante 4s il goodput di FTP1 è significativo, in quanto per i primi secondi di simulazione è riuscita a trasmettere indisturbata.
Al contrario, il goodput di FTP2 come previsto rasenta lo zero.

```
Simulazione AQM: DROPTAIL

Network topology:
n0 -------------------------- n1 -------------------------- n2
            1Gbps, 2ms              7Mbps DL, 64kbps UL, 10ms

ID bottleneck: 2
Goodput FTP1 (n0 --> n2): 88.4185 kB/s
Goodput FTP2 (n2 --> n0): 3.216 kB/s
```

Fig. 5.20: Output dell'esecuzione dello script - Droptail

Utilizzando una politica di AQM, ci si può aspettare che la situazione rimanga invariata nel lasso di tempo in cui è soltanto FTP1 a trasmettere, mentre migliori quando entra in gioco la seconda connessione FTP2.

Ciò è in effetti verificato e come emerge dal seguente grafico, nonostante l'inizio della trasmissione di FTP2 disturbi fortemente FTP1, i crolli della finestra di trasmissione di quest'ultima sono meno frequenti che in assenza di AQM.

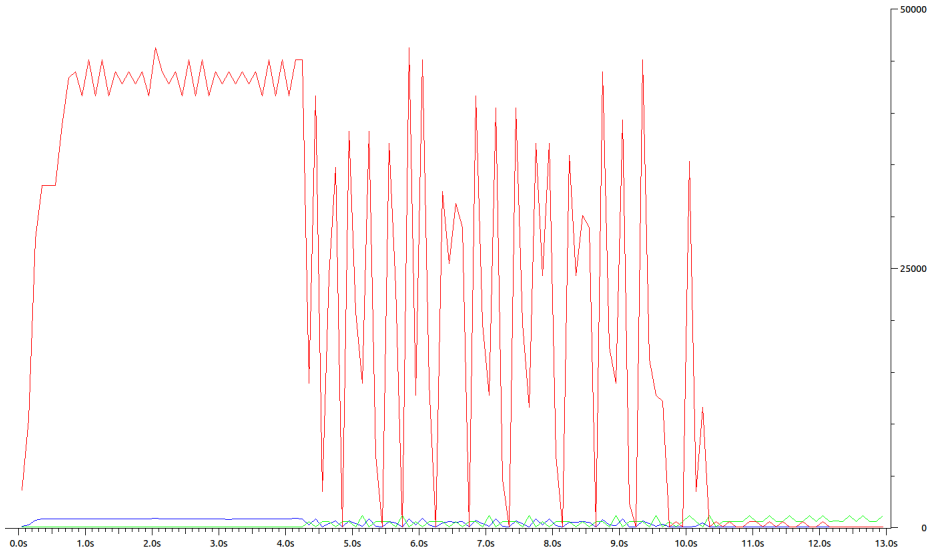

Fig. 5.21: Output dell'esecuzione dello script - AQM RED

Anche analizzando dall'output della simulazione si giunge ad analoghe conclusioni: il goodput di FTP1 è quasi doppio che nel caso di assenza di AQM.

```
Simulazione AQM: RED

Network topology:
n0 -------------------------- n1 -------------------------- n2
            1Gbps, 2ms              7Mbps DL, 64kbps UL, 10ms

ID bottleneck: 2
Goodput FTP1 (n0 --> n2): 147.441 kB/s
Goodput FTP2 (n2 --> n0): 3.62831 kB/s
```

Fig. 5.22: Output dell'esecuzione dello script - AQM RED

Capitolo 6

Conclusioni

Il *workflow* definito per il progetto era il seguente:

1. Studio delle tecniche di **AQM**
2. Studio delle tecniche di **Packet Scheduling**
3. Individuazione dei moduli di **ns-3** coinvolti da queste soluzioni
4. Eventualmente, implementazione e integrazione di semplici soluzioni AQM e/o Packet Scheduling in ns-3

Innanzitutto lo studio delle tematiche di AQM e Packet Scheduling è stato preceduto da una fase di raccolta di informazioni e inquadramento dell'argomento dal punto di vista generale, che ci ha portati ad analizzare la materia del Quality of Service (QoS), in particolare la sua definizione, gli elementi e i meccanismi che la strutturano.
Di seguito viene analizzato il risultato conseguito per punti.

1) Lo studio delle tecniche di AQM è stato approfondito grazie al materiale trovato sulla rete e alla lettura degli *RFC (Request for Comments)* ufficiali. Si è potuto così apprendere appieno il reale significato delle tecniche di gestione delle code, argomento che spesso viene confuso e associato alla diversa tematica del Packet Scheduling.
La nostra analisi si è in particolare soffermata sull'algoritmo RED (Random Early Detection); in seguito sono state studiate le sue evoluzioni quali il Weighted Random Early Detection (WRED) e il Robust Random Early Detection (RRED).

2) In maniera analoga al punto precedente, lo studio degli argomenti riguardanti il Packet Scheduling è stato svolto tramite materiale trovato in

rete e gli RFC ufficiali.
E' stato inoltre utilizzato il materiale didattico fornito dal corso di *Tecnologie e Infrastrutture di Rete* dell'Anno Accademico 2013/2014 (docente Prof. Maurizio Casoni) e altro materiale derivante da seminari tenuti durante l'anno, organizzati all'interno del corso (in particolare quello tenuto da Carlo Augusto Grazia).
Sono stati analizzati e studiati varie soluzioni quali, dal punto di vista teorico, il Generalized Processor Sharing (GPS) e la sua approssimazione Weighted Fairs Queueing (WFQ) e dal punto di vista pratico gli algoritmi Priority Scheduling (PRIO), Round Robin (RR), Weighted Round Robin (WRR) e Deficit Round Robin (DRR).

3) L'individuazione dei moduli di ns-3 coinvolti nell'implementazione delle tecniche di AQM e Packet Scheduling ha richiesto un periodo di studio abbastanza distribuito nel tempo, complice la non sempre completa ed efficace documentazione ufficiale del simulatore.
Le informazioni raccolte ci hanno permesso di comprendere come fra i moduli già inseriti nel programma fosse presente solo quello per l'implementazione di RED.
L'implementazione dei moduli per il Packet Scheduling ha richiesto di inoltrarsi in un campo non ancora trattato all'interno di ns-3, anche dal punto di vista internazionale: questo ha significato dover creare da zero gli algoritmi ed eseguire vari test per ogni loro componente.

4) Per l'implementazione delle soluzioni si è proceduto dapprima alla costruzione delle topologie delle reti di test. I principi seguiti sono stati quelli della massima semplicità, che ha avuto il duplice vantaggio di minimizzare le interferenze di elementi superflui nella simulazione e dare importanza a quelli utili a comprovare il corretto funzionamento degli algoritmi; al tempo stesso si è cercato di avere topologie realistiche assegnando, ad esempio, canali con bande proporzionate alla tipologia dell'host connesso.
Per la fase di testing di RED è stato utilizzato il modulo già presente all'interno di ns-3. Per il Packet Scheduling si è deciso di implementare tutti gli algoritmi trattati prima dal punto di vista teorico.
Non sono mancati i casi di studio in cui si sono andate a modificare le caratteristiche o i parametri degli scheduler, con l'obiettivo di studiare nuovi scenari ed eseguire ulteriori test sul corretto funzionamento degli algoritmi implementati.
I risultati sono stati del tutto soddisfacenti, in quanto le soluzioni implementate funzionano correttamente e i dati prodotti sono risultati in linea con le nostre aspettative e la teoria.