TANGO

Algoritmi e modelli dell'ingegneria del traffico per l'ottimizzazione di reti IP di nuova concezione

Project Number:	**RBNE01BNL5**
Project Acronym:	TANGO

Document Number:	WP2-UnivCT-002-
Document Title:	- Panoramica su NS - The Network Simulator
WP Contributing to Document:	WP2
Nature of Document:	Report
Status:	Draft
Authors:	Mario Barbera, Alfio Lombardo, Giovanni Schembra, Giovanni Tusa

SOMMARIO

1. Introduzione alla struttura del simulatore

NS (Network Simulator) è un simulatore di rete pilotato ad eventi realizzato alla Berkeley, che consente di testare il comportamento di una grande varietà di reti IP, implementando protocolli di rete come TCP e UDP, protocolli per la gestione della congestione come RAP e TFRC, meccanismi per la gestione delle code nei routers come DropTail, RED e CBQ, algoritmi di routing, oltre al multicasting, all'ambiente wireless ed alcuni dei protocolli di MAC per la simulazione di reti LAN. NS contiene anche moduli per testare architetture di tipo DiffServ. Attualmente è disponibile la versione due di NS, in particolare si è arrivati alla 2.26.
La figura 1 mostra una vista d'utente del Network Simulator:

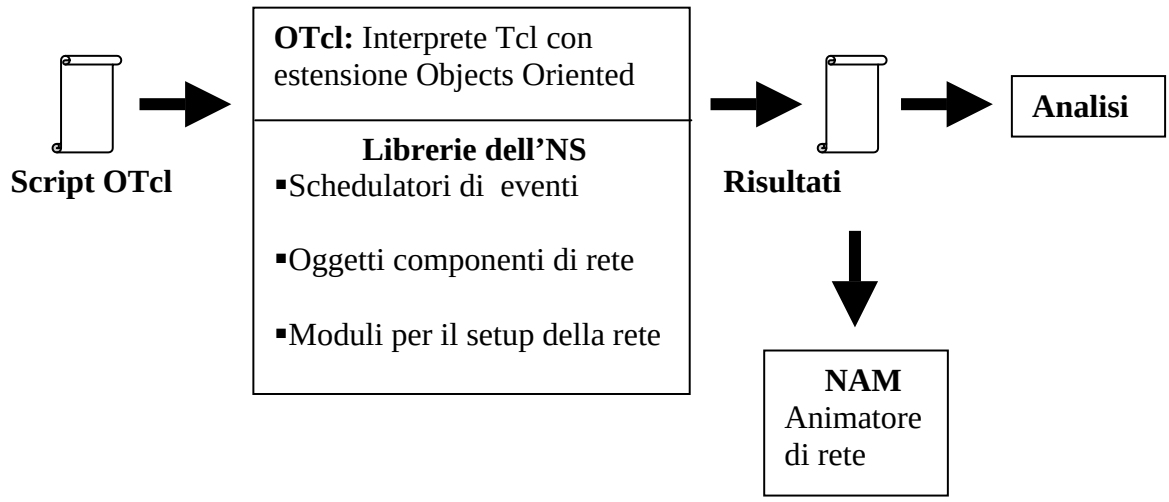

Figura 1. Vista d'utente dell'NS.

NS è un interprete Tcl con estensione Object Oriented, e per utilizzarlo bisogna dapprima scrivere un programma in linguaggio OTcl che deve contenere una schedulazione degli eventi (quando una sorgente deve iniziare o smettere di trasmettere, quando acquisire i parametri da analizzare, ecc.), la creazione della topologia della rete tramite gli oggetti OTcl presenti (classi e sottoclassi), e utilizzare le funzioni di libreria per la connessione degli oggetti (setup della rete).

Durante la simulazione, se specificato nel programma OTcl, NS produce in uscita dei file che permettono di effettuare l'analisi prestazionale, oppure di ripercorrere su display l'andamento della simulazione, tramite il Network AniMator (NAM).
In NS, la realtà è modellata come una collezione di eventi: uno schedulatore di eventi tiene memoria del tempo corrente di simulazione e della lista di eventi associati alla stessa, e fa in modo che gli oggetti legati ad un dato evento realizzino l'azione appropriata. Ovviamente gli eventi vengono schedulati in istanti di tempo "virtuali", mentre il tempo reale consumato è arbitrario. Gli elementi di rete comunicano tra loro attraverso lo scambio di pacchetti, e ciò non produce impiego di tempo di simulazione. Gli oggetti di rete che devono simulare l'impiego di un certo tempo nel manipolare i pacchetti, simulano questo ritardo schedulando l'evento ed attendendo che scada il tempo voluto prima di compiere qualunque azione. E' il caso del tempo di propagazione dei link, in cui il componente di rete che processa il pacchetto utilizza lo schedulatore per la consegna del pacchetto al link di uscita allo scadere del ritardo settato dall'utente.
NS è scritto sia in OTcl sia in C++, in altre parole supporta sia una gerarchia di classi C++, detta gerarchia compilata, sia una gerarchia OTcl, detta gerarchia interpretata. Esiste una corrispondenza uno ad uno tra gli oggetti delle due gerarchie, ed il motivo dell'esistenza di entrambe risiede essenzialmente in questioni di efficienza: viene separata l'implementazione delle funzionalità che gestiscono il percorso dati durante la simulazione, dall'implementazione delle funzionalità di controllo e di setup della rete.
Così, per ridurre il tempo di processamento dei pacchetti e degli eventi legati alle funzioni dei protocolli, gli oggetti componenti di rete e gli schedulatori di eventi relativi al percorso dati (vedi timer di ritrasmissione del TCP) sono implementati in C++, mentre per il setup della rete ed il controllo dei parametri durante la simulazione si utilizza il codice OTcl, più semplice e rapido da scrivere.
Per modificare o aggiungere nuovi oggetti da poter utilizzare nelle simulazioni bisogna scrivere e compilare in C++. Gli oggetti della gerarchia compilata in C++, sono poi resi disponibili all'interprete OTcl tramite un collegamento che crea un corrispondente OTcl per ognuno degli oggetti C++, e traduce le funzioni membro e le variabili specificate dagli oggetti C++ come funzioni membro e variabili membro del

corrispondente oggetto OTcl. Così, il controllo degli oggetti C++ è demandato all'OTcl. Quando l'utente definisce l'istanza di una classe nello script OTcl, viene creata un istanza dell'oggetto corrispondente della classe compilata in C++ attraverso i metodi della classe TclObject. Ovviamente ci sono alcuni oggetti C++ che, non dovendo essere controllati durante la simulazione, non necessitano di quest'accoppiamento.

La figura 2 mostra una vista dell'architettura di NS. L'utente interessato al suo utilizzo a scopo puramente simulativo, ma che non sia interessato al suo sviluppo (aggiunta e compilazione di nuovi oggetti), può fermarsi alla stesura di script in linguaggio Tcl, utilizzando gli oggetti della libreria OTcl per la configurazione della topologia di rete desiderata.

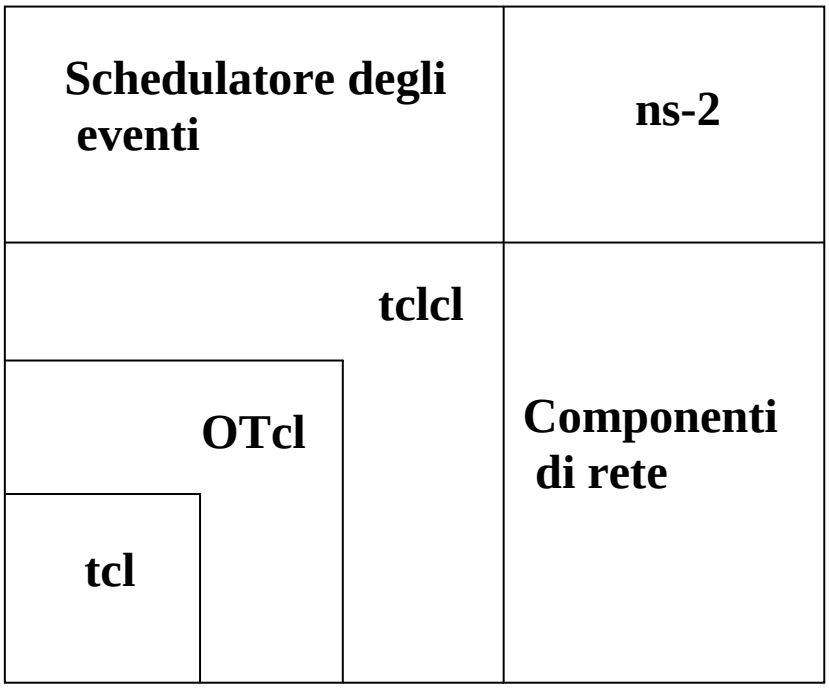

Figura 2. Architettura dell'NS.

Lo sviluppatore utilizzerà invece la gerarchia C++, con cui sono implementati gli schedulatori di eventi e la maggior parte dei componenti di rete relativi al percorso

dati. L'interfaccia OTcl/C++ con cui tali oggetti sono mappati nei corrispondenti OTcl è realizzata dal tclcl.

2. La gerarchia di classe

La figura 3 illustra uno schema, solamente parziale per ovvi motivi, della gerarchia di classe OTcl, facendo riferimento agli oggetti più importati e comunemente utilizzati.

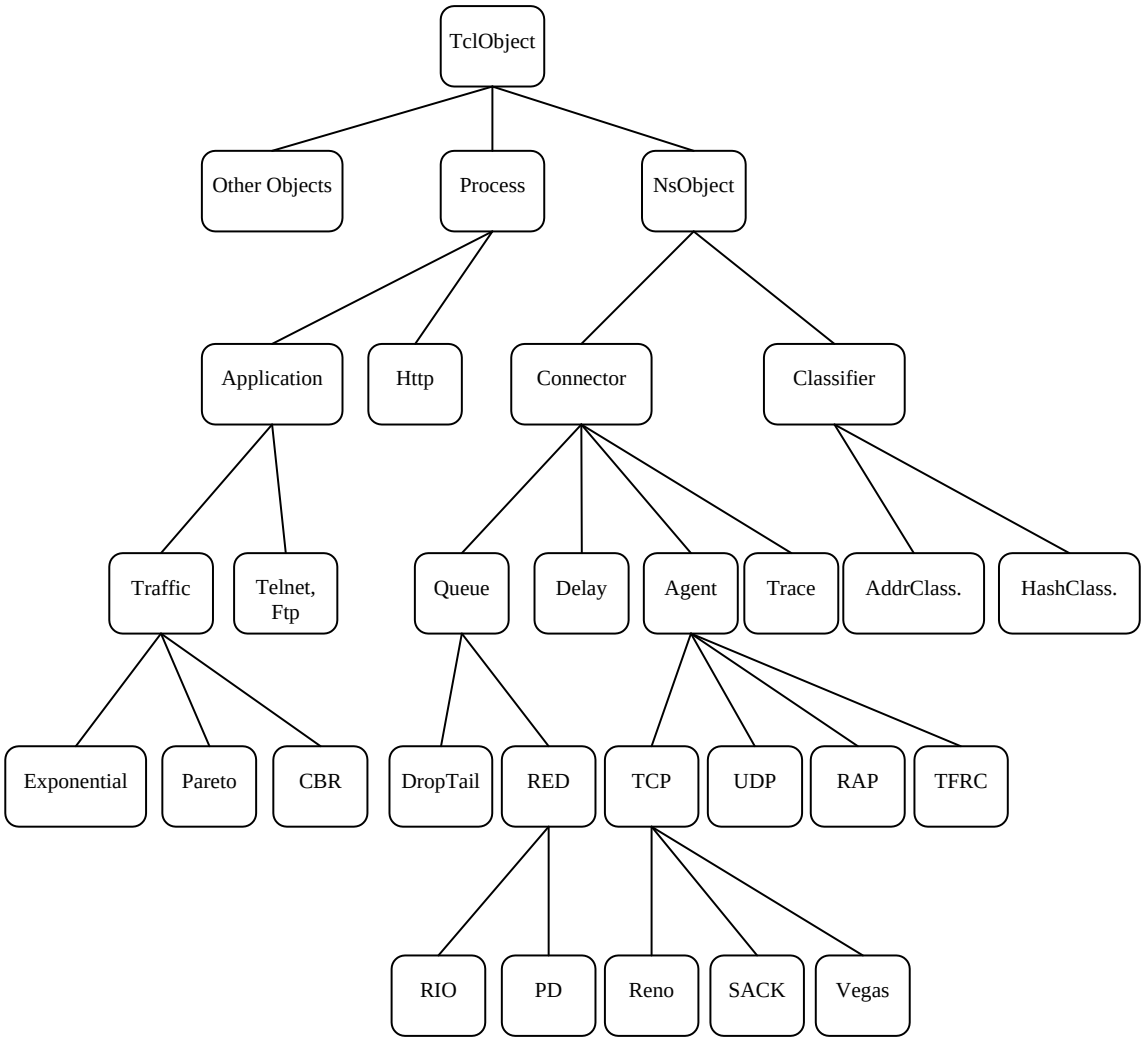

Figura 3. Gerarchia di classe.

La radice dell'albero gerarchico è la classe *TclObject*, da cui derivano tutti gli oggetti OTcl all'interno di NS. Tra le classi derivate, una grande importanza rivestono la classe *Process*, da cui derivano tutte le entità capaci di processare ADU, consegnare o richiedere dati da altre entità (tipicamente il livello applicativo), e la classe *NsObject*, che rappresenta la classe base di tutti i componenti fondamentali di rete, ossia le entità che processano i pacchetti. Tali componenti sono a loro volta suddivisi in due sottoclassi fondamentali, la *Connector* e la classe *Classifier*. Dalla prima discendono tutti i componenti di rete che hanno a disposizione un solo percorso d'uscita nel cammino dei pacchetti, mentre della seconda fanno parte i componenti che hanno più percorsi di uscita possibili. Quando un nodo riceve un pacchetto, esamina le informazioni relative alla sua destinazione, prima di instradare il pacchetto sull'interfaccia verso il prossimo componente che dovrà processarlo. Questa funzionalità è normalmente svolta dagli oggetti *Classifier*.

Tra gli oggetti che discendono dalla classe *Connector*, le classi *Queue* e *Delay* sono entrambe parte della struttura dei link. Infatti, la gestione delle code è attualmente implementata in NS all'ingresso del link corrispondente e, nel momento in cui un pacchetto lascia la coda, viene passato all'oggetto *Delay* che simula il ritardo di propagazione, secondo le modalità cui si è già accennato. La classe *Agent* è un' altra classe fondamentale perché da essa discendono, tra gli altri, i ben noti protocolli di trasporto, i protocolli per il controllo di congestione, oltre ad alcuni protocolli di routing. Dalla classe *Trace* derivano gli oggetti che consentono, se specificato, di tenere una traccia delle attività di rete in corrispondenza di ciascun link (ad esempio il passaggio di pacchetti dati o ack).

Le più comuni sorgenti di traffico discendono invece dalla classe *Application*, a sua volta sottoclasse della *Process*.

2.1 Il livello di rete

La classe base degli oggetti coda è la classe (astratta) *Queue*, da cui derivano un certo numero di classi che, ereditandone metodi ed attributi, ne definiscono di nuovi per caratterizzare la specifica tecnica di gestione delle code implementata.

Così la classe *DropTail*, mappata in OTcl col nome *Queue/DropTail*, definisce la classica gestione delle code di tipo FIFO con lo scarto dei pacchetti che sopraggiungono in condizioni in cui non si hanno più risorse disponibili nel buffer, *Queue/RED* implementa il Random Early Detection, *Queue/CBQ* la gestione delle code basata sulle classi di traffico differenziate per livelli di priorità, e così via dicendo.

A sua volta, è possibile derivare ancora dalle sottoclassi per definire delle ulteriori varianti o specializzazioni di una tecnica particolare, come nel caso della *Queue/RED/RIO,* che implementa il RED con differenziazione del trattamento dei pacchetti a seconda che rientrino all'interno delle specifiche dichiarate dalla sorgente (in) o meno (out).

2.2 Il livello di trasporto

I differenti protocolli di trasporto vengono derivati dalla classe (astratta) *Agent*. Tra questi troviamo ovviamente il protocollo TCP nelle sue varie versioni: così *Agent/TCP* si riferisce alla vecchia versione Tahoe, mentre le classi derivate *Agent/TCP/Reno*, *Agent/TCP/NewReno,* ecc., costituiscono una specializzazione della stessa per l'implementazione delle versioni successive. Da notare poi che dalla classe *Agent* derivano, oltre che il protocollo di trasporto *Agent/UDP*, anche i protocolli per il controllo di congestione *Agent/TFRC* e *Agent/RAP*, comunemente utilizzati in presenza di applicazioni multimediali in concomitanza all'UDP per risolvere i problemi di congestione ed unfairness che l'utilizzo di sorgenti totalmente unresponsive può determinare, ma che in NS integrano le funzioni di trasporto dell'UDP e vengono perciò utilizzati singolarmente.

2.3 Il livello applicativo

Il livello applicativo è rappresentato essenzialmente dalla classe astratta *Application*, da cui derivano le applicazioni simulate ed i generatori di traffico. Delle applicazioni simulate fanno parte la classe *Application/Telnet* ed *Application/FTP*, utilizzate per usufruire del servizio di trasporto del TCP. I generatori di traffico sono derivati dalla classe astratta *TrafficGenerator*, a sua volta derivata dalla *Application* con il nome OTcl *Application/Traffic*. Da queste discendono sorgenti come la CBR, la

distribuzione Esponenziale ON/OFF e la Paretiana ON/OFF, mappate in OTcl rispettivamente come *Application/Traffic/CBR, Application/Traffic/Exponential,* ed *Application/Traffic/Pareto.*

3. Elementi base di uno script in NS

La stesura di un semplice script NS non può prescindere dai seguenti blocchi fondamentali:

- istanza del simulatore;
- apertura di eventuali oggetti di traccia o file per il monitoraggio dei parametri d'interesse dell'utente (entrambi opzionali), e loro chiusura;
- topologia della rete;
- definizione e connessione delle entità di rete;
- schedulazione degli eventi.

3.1 La classe Simulator

La classe *Simulator* è la classe fondamentale in NS poiché per effettuare una simulazione è necessario creare un oggetto Simulator, ossia un'istanza della simulazione stessa tramite l'istruzione OTcl:

set ns [*new Simulator*] .

La variabile ns sarà così l'istanza desiderata a cui si potrà far riferimento nel resto dello script.

3.2 Definizione della struttura topologica della rete

Per la definizione della topologia di rete voluta NS mette a disposizione dell'utente un certo numero di entità e comandi. E' possibile così definire i nodi che andranno a costituire la nostra rete, collegandoli tra loro opportunamente tramite link, caratterizzati dalla capacità, espressa in bit/s, dal ritardo di propagazione, espresso in secondi e dal tipo di tecnica per la gestione della coda. La definizione dei nodi è realizzata col semplice comando:

set <nomenodo1> [$ns *node*]

set <nomenodo2> [$ns *node*]

mentre, supponendo di aver dichiarato due nodi n1 e n2 la loro connessione si realizza con l'istruzione:

$ns *duplex-link* $n1 $n2 <bw> <delay> <queue_type>

nel caso di un link bidirezionale tra i due nodi, con capacità pari a bw bit/s, ritardo delay secondi, e gestione della coda del tipo che si vuole (*DropTail, RED, CBQ,* ecc.). Per imporre l'unidirezionalità del nodo si sostituirà la parola chiave *duplex-link* con *simplex-link.*

3.3 Definizione dei protocolli di rete

Definita l'architettura di rete, il passo successivo consiste nel fornire alla stessa l'intelligenza voluta, in termini di protocolli di vario livello che si vuole "girino" sui nodi.

3.3.1 Strategie di routing

NS dà la possibilità di utilizzare il protocollo di routing desiderato da un certo set messo a disposizione. Il comando per scegliere la strategia (routing statico, dinamico) ed il protocollo (algoritmo) di routing specifico in NS è unico, anche perché alcune strategie sono attualmente associate ad un determinato protocollo e vengono considerate esse stesse come protocolli di routing:

$ns *rtproto* <routing_proto>

dove al posto di <routing_proto> andrà una parola chiave che specificherà la scelta fatta (*Static, Session, LS, DV, Manual*). Nel caso unicast, per default viene utilizzato il routing di tipo *Static* in cui il percorso dei pacchetti per ogni coppia sorgente/destinazione è fissato una volta per tutte sulla base dei costi dei vari link, anch'essi settati per default ma eventualmente modificabili dall'utente:

$ns *cost* $n1 $n2 <value>

essendo <value> per default pari ad 1.

3.3.2 Generazione e controllo del traffico

In un tipico scenario applicativo, alcuni nodi saranno gli host sorgente e ricevitore, caratterizzati da un generatore/ricevitore di traffico ed un protocollo di trasporto. Supponendo di aver definito un oggetto udps della classe che implementa il protocollo

UDP dal lato sorgente, operazione che in linguaggio OTcl si realizza con la semplice creazione dell'istanza della classe:

set udps [*new Agent/UDP*]

e di volere che questo "giri" sul nodo udp1, definito precedentemente con l'istruzione appropriata, il comando da utilizzare sarà:

$ns *attach-agent* $udps $udp1.

con il quale viene imposto che un agent UDP agisca su quel nodo.

Un'operazione simile va fatta dal lato ricevente per creare un agent Null, ossia un ricevitore che elimina i pacchetti senza compiere nessuna azione di retroazione (nel caso UDP):

set udpr [*new Agent/NULL*]

$ns *attach-agent* $udpr $udp2.

Ovviamente sorgente e ricevitore vanno connessi per specificare la destinazione verso cui dovranno essere diretti i pacchetti, e questo si realizza attraverso il comando OTcl *connect*:

$ns *connect* $udps $udpr.

I nodi udp1 ed udp2 vanno collegati agli estremi del link tra i nodi n1-n2 attraverso ulteriori link:

$ns *duplex-link* $udp1 $n1 <bw> <delay> <queue_type>

$ns *duplex-link* $n2 $udp2 <bw> <delay> <queue_type>.

Supponendo di essere interessati alle prestazioni della rete in relazione al link n1-n2 definito precedentemente, è buona pratica settare per gli altri un valore di banda molto più elevato, ed una gestione della coda non attiva (tipicamente DropTail) e con un buffer di capacità molto elevata, per renderli ininfluenti ai fini dell'analisi prestazionale, facendo in modo che n1-n2 sia il collo di bottiglia (bottleneck) del sistema. Nel caso, più realistico, in cui si preveda che più sorgenti debbano caricare il bottleneck, il parametro <delay> sui link d'ingresso può essere utilizzato per differenziare l'aggressività con cui le sorgenti andranno a caricare il nodo interno.

3.4 La classe Scheduler

Essendo NS un simulatore di rete pilotato ad eventi, all'utente viene offerta la possibilità di schedulare gli eventi della simulazione in ordine di tempo, sempre a livello di script OTcl. La classe *Scheduler* si occupa di eseguire gli eventi della simulazione seguendo l'ordine temporale fornito dall'utente, e quelli schedulati nello stesso istante secondo l'ordine di schedulazione. Tra i differenti tipi di schedulatori ricordiamo la sottoclasse *List*, mappata in OTcl col nome *Scheduler/List*, implementata con una semplice struttura a lista i cui elementi sono gli eventi della simulazione sistemati in ordine crescente di tempo e dunque estratti secondo questo stesso ordine, e la classe *Scheduler/Calendar*, il default in NS, che utilizza una struttura simile ad un calendario.

Supponendo che il generatore di traffico che consegna i dati al protocollo di trasporto UDP sia una sorgente CBR, per creare un'istanza di questo oggetto ed attaccarla all'agent UDP si utilizzano istruzioni identiche a quelle già viste:

set cbrs [*new Application/Traffic/CBR*]

$cbrs *attach-agent* $udps.

Se si vuole che questa sorgente inizi a spedire pacchetti all'istante 5 dall'inizio della simulazione, e si disattivi allo scadere del tempo che indica la durata della simulazione:

$ns *at* 5 "$cbrs *start*"

$ns *at* $durata "$cbrs *stop*".

La durata della simulazione deve essere definita prima con l'istruzione:

set durata <value> .

Normalmente uno script si chiude con l'istruzione che inizializza lo schedulatore:

$ns *run*.

3.5 Monitoraggio dei link e dei parametri

NS offre la possibilità di tenere traccia dei pacchetti che transitano sui link, del loro tipo, della loro avvenuta ricezione, ecc. L'utente può settare quest'opzione attraverso l'apertura del tracefile:

set nf [*open* out.tr w]

e decidere di monitorare l'attività su tutti i link:

$ns *trace-all* $nf.

Questa è un'operazione non sempre consigliata, viste le dimensioni che un tracefile può assumere, per cui esiste la possibilità di monitorare soltanto il link cui si è maggiormente interessati:

$ns *trace-queue* $n1 $n2 $nf.

In ogni caso alla fine della simulazione bisogna procedere alla chiusura del suddetto file :

$ns *flush-trace*

close $nf

Analogamente si procede per l'apertura di un file di testo, il salvataggio dei parametri all'interno dello stesso e la sua chiusura:

set outfile [*open* file.txt w]

puts $outfile "$mioparametro"

close $outfile.

4. Realizzazione di nuovi moduli in NS

Per una qualunque versione di NS sono disponibili i codici sorgente in C++, il che consente, con una buona conoscenza del C++, della struttura gerarchica di NS e del linguaggio OTcl, di inserire nuovi protocolli o semplicemente estendere quelli già presenti (ad esempio un particolare algoritmo all'interno di un protocollo già esistente) che, una volta compilati, diventano parte integrante del simulatore. Data una certa collezione di classi base sono, infatti, derivate le sottoclassi che permettono da una parte la specializzazione e dunque la differenziazione di tecniche afferenti allo stesso tipo di problematica, dall'altra una facile espansibilità di NS stesso, cui chiunque può contribuire a partire dai propri interessi di ricerca.

A titolo d'esempio, nel seguito del documento sarà illustrata la procedura seguita per la realizzazione di un nuovo particolare scenario applicativo.

Per il momento vengono elencate le linee guida generali da seguire per l'inserimento di un nuovo oggetto all'interno del simulatore:

- dichiarazione ed implementazione dei metodi della classe C++;
- creazione del collegamento OTcl corrispondente;
- collegamento delle variabili membro della classe C++ con le corrispondenti variabili dell'oggetto OTcl;
- implementazione, all'interno della nuova classe, dell'interprete OTcl dei comandi;
- dichiarazione della struttura di eventuali nuovi pacchetti;
- dichiarazione della classe OTcl corrispondente al nuovo tipo di pacchetto definito.

4.1 Creazione delle nuove classi

Nell'implementare la nuova classe, bisogna dapprima decidere il suo posizionamento all'interno della gerarchia di classe, al fine di derivarla da uno degli oggetti già definiti ereditandone i metodi fondamentali. E' importante poi osservare che quando dallo script si crea una nuova istanza della classe OTcl secondo le modalità già viste, ad esempio nel caso di una sorgente TCP:

set tcps [*new Agent/TCP*]

dovrà essere disponibile anche un'istanza del corrispondente oggetto C++, e perché ciò avvenga ogni oggetto C++ deve essere mappato nel corrispondente oggetto OTcl. Questo viene realizzato attraverso la derivazione di una sottoclasse della classe TclClass:

```
static class TcpClass : public TclClass {
public:
      TcpClass() : TclClass("Agent/TCP") {}
      TclObject* create(int, const char*const*)
      {return (new TcpAgent());
                  }
} class_tcp;
```

L'invocazione del costruttore della classe base definisce il nome OTcl assegnato alla sottoclasse, mentre il metodo *create()* ritorna l'oggetto C++ corrispondente.

4.2 Collegamento delle variabili

NS offre la possibilità di manipolare una serie di parametri all'interno dello script OTcl, sia per la fase di inizializzazione delle variabili stesse, sia per un loro controllo durante l'andamento di una simulazione: si può ad esempio avere la necessità di monitorare l'andamento di un parametro salvandolo su un file da analizzare successivamente.

Il settaggio dei valori è realizzato in NS con la semplice istruzione:

$tcps *set* **cwnd_** 400

mentre un parametro da monitorare può essere acquisito e salvato in una variabile locale con il comando:

set window [$tcps *set* **cwnd_**]

Ovviamente, le variabili di un oggetto OTcl, devono ancora una volta essere collegate con le corrispondenti variabili membro utilizzate all'interno del codice C++.

Tale collegamento si realizza utilizzando il metodo 'public' *bind()* della classe TclObject, da cui discendono i principali componenti di rete (si faccia riferimento alla figura 3), all'interno del costruttore della classe C++:

```
TcpAgent::TcpAgent() {
        ……
        bind("ack_",&highest_ack_);
        bind("cwnd_",&cwnd_);
        ……
}
```

4.3 Interpretazione dei comandi

Un altro metodo della classe TclObject, la funzione *command()*, funge da interprete OTcl dei comandi in modo tale che quando viene utilizzato un metodo OTcl all'interno dello script, venga invocata anche la corrispondente procedura dell'oggetto C++. Così nel caso del metodo **advance** delle sorgenti TCP l'istruzione:

$tcps **advance** 10

produce una chiamata alla funzione *command()* implementata nel codice C++, la quale esegue un parsing per l'esecuzione della procedura C++ associata, in questo caso la *advanceby()*:

```
int TcpAgent::command(int argc, const char*const* argv) {
     if(argc==3) {
            if(strcmp(argv[1],"advance" )==0) {
                 int newseq = atoi(argv[2]);
                 if(newseq > maxseq_)
                     advanceby(newseq – curseq_);
                 else
                     advanceby(maxseq_ – curseq_);
                 return(TCL_OK);
             }
      }
      ……
      return Agent::command(argc,argv);
}
```

4.4 Formato dei pacchetti

Lo sviluppatore può avere la necessità di utilizzare pacchetti creati ad hoc per le proprie applicazioni: NS offre anche questa possibilità.

Un pacchetto in NS è costituito da uno stack di headers ed uno spazio dati opzionale. Al momento della creazione di un nuovo oggetto *Simulator* nello script OTcl, è inizializzato un formato di header contenente lo stack completo di tutti gli headers esistenti (figura 4), dal *common header*, utilizzabile da ogni oggetto a seconda delle necessità, all'header IP, TCP, RAP, ecc., ed è registrato l'offset di ogni header dello stack, tramite cui un oggetto della rete può accedere all'header di un pacchetto che stà processando.

Gli eventuali nuovi header di pacchetto definiti vanno realizzati come semplici strutture dati e per essi và creata una classe C++ derivata dalla *PacketHeaderClass,* a cui và assegnato un nome OTcl.

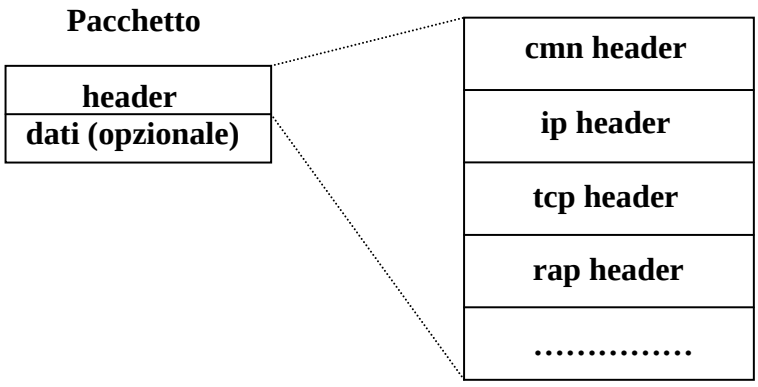

Figura 4: Formato dei pacchetti in NS

5. Un esempio: applicazione Video su protocollo RAP o TFRC

Come esempio della espansibilità di NS, viene di seguito riportata l'esperienza relativa all'aggiunta, all'interno del simulatore, di un'applicazione video MPEG le cui fasi di emissione e codifica sono modellate da un processo statistico SBBP (Switched Batch Bernoulli Process), e che utilizza il modello strutturato TM5 per il controllo del rate di emissione. Al fine di realizzare il feedback necessario a rendere la sorgente responsiva alle variazioni di banda disponibili in rete, si utilizza un protocollo per il controllo di congestione, quale RAP o TFRC. Nella realtà si dovrebbe prevedere anche il protocollo di trasporto UDP, ma in NS i protocolli RAP e TFRC integrano tale funzione, e sono anch'essi implementati come classi derivate della classe *Agent*.

5.1 Collocazione degli oggetti nella gerarchia delle classi

L'applicazione utilizzata è denominata "*video_app*" nella gerarchia compilata in C++, ed è derivata in maniera public dalla classe *TrafficGenerator*, a sua volta derivata dalla classe *Application*. Il nome corrispondente nella gerarchia OTcl è *Application/Traffic/video_app*.
Al fine di mappare la classe C++ nel suo corrispondente oggetto OTcl, si rende necessario aggiungere le seguenti linee di codice nel file "video_app.cc":

```
static class video_appClass : public TclClass {
public:
      video_appClass() : TclClass("Application/Traffic/video_app") {}
      TclObject* create(int, const char*const*) {
      {return (new video_app());}
        }
} class_video_app;
```

E' stata definita anche una nuova classe per il RAP nella gerarchia compilata in C++, denominata "*RapMmAgent*" e derivata in maniera 'public' dalla classe astratta *Agent*. Si è preferito non derivarla dalla classe *Agent/RAP* per mantenere una certa indipendenza nel caso di modifiche sostanziali dettate da esigenze di ricerca. Il nome corrispondente nella gerarchia OTcl è *Agent/RAPMm*. Al fine di creare tale

corrispondenza, è necessario inserire nel file “rap-mm.cc” da compilare in C++ le seguenti linee di codice:

```
static class RapMmClass : public TclClass
{
public:
      RapMmClass():TclClass("Agent/RAPMm") {}
       TclObject* create(int, const char*const*)
            { return (new RapMmAgent()); }
} class_rapmm;
```

E’ superfluo sottolineare che passi del tutto analoghi possono essere seguiti per la definizione del nuovo agent TFRC, e che ora e nel resto della trattazione ometteremo per semplicità.

5.2 Definizione dei nuovi pacchetti

Nel caso in esame è stata definita una nuova struttura di header dal lato applicazione, in maniera tale che quando si hanno informazioni da trasmettere all’agent, o viceversa l’agent abbia necessità di comunicare informazioni di feedback relative al cambiamento delle risorse di rete stimate, queste siano trasmesse nel formato della nuova struttura. Un nuovo header di pacchetto in NS si definisce semplicemente tramite una struttura dati contenente i campi d’interesse, oltre ad una funzione che ritorna l’offset necessario al riconoscimento del pacchetto:

```
struct hdr_mapp {
        int rc;              /* used to indicate if it's a rate
                                control message or a data packet */
        int seq;             /* sequence number */
        int nbytes;          /* total length of the information passed to
                                the underlying agent, in bytes */

        double rate;         /* current estimated rate
        double time;         /* current time
        int packno;          /* total length of the information passed to
                                 the underlying agent, in packets */

        //Packet header access functions
        static int offset_;
        inline static int& offset() { return offset_;}
        inline static hdr_mapp* access(const Packet *p) {
        return (hdr_mapp*) p->access(offset_);
        }
  };
```

Fa parte della struttura anche il metodo tramite cui ogni oggetto della rete può andare a leggere o manipolare l'header in questione (es. leggere o settare campi specifici):

```
hdr_mapp::access(pkt)
```

Definita la struttura dell'header del pacchetto, bisogna definire la corrispondente classe, in questo caso chiamata *TM5HeaderClass* e derivata dalla *PacketHeaderClass*, nel file "video_app.cc".

```
static class TM5HeaderClass : public PacketHeaderClass
{
public:
 TM5HeaderClass() : PacketHeaderClass("PacketHeader/TM5",
                                      sizeof(hdr_mapp)) {
            bind_offset(&hdr_mapp::offset_);
      }
} class_mhdr;
```

Si noti come nella definizione di questa classe è necessario solo implementare il costruttore di copia, e come nell'invocazione del costruttore della classe base vengano definiti il nome OTcl della struttura e la sua dimensione.
Analogamente si definisce l'header per i pacchetti *RAPMm*:

```
struct hdr_rapm
{
      int seqno_;         /* Seq num of packet being
                             sent/acked */
      int size_;          /* Size of user data inside this
                             packet */
      int rap_flags_;     /* Flags to separate
                             data/ack packets */
      int packem;        /* Number of packet in a burst to be sent */

      int lastRecv;      /* Last hole at the RAPMm
                                sink... */
      int lastMiss;
      int prevRecv;

      static int offset_;     /* offset for this header */
      inline static int& offset() { return offset_; }
      inline static hdr_rapm* access(const Packet* p) {
            return (hdr_rapm*) p->access(offset_);
      }

      int& seqno() { return seqno_; }
      int& size() { return size_; }
      int& flags() { return rap_flags_; }
}
```

```
static class RapMmHeaderClass : public PacketHeaderClass
{
public:
      RapMmHeaderClass() : PacketHeaderClass("PacketHeader/RAPMm",
                                   sizeof(hdr_rapm)) {
            bind_offset(&hdr_rapm::offset_);
      }
} class_rapmhdr;
```

La registrazione di nuovi headers nello stack è completata aggiungendo le nuove informazioni nel file "ns-packet.tcl" (per la gerarchia OTcl):

```
foreach prot {
        AODV
        ...
        ...
        TM5
       RAPMm
} {
    add-packet-header $prot
}
```

e "packet.h" (per la compilazione in C++):

```
enum packet_t {
        PT_TCP,
         ...
   //Multimedia Packet
        PT_TM5,
   //RAPMm packets
      PT_RAPMm_DATA,
      PT_RAPMm_ACK,
      PT_NTYPE      //This MUST be the LAST one
}

class p_info {
public:
      p_info() {
           name_[PT_TCP]= "tcp";
                .......
          //Multimedia Packets
           name_[PT_TM5] = "TM5";
         //RAPMm packets
           name_[PT_RAPMm_DATA] = "rapmm_data";
           name_[PT_RAPMm_ACK] = "rapmm_ack";
           name_[PT_NTYPE]= "undefined";
      }
      ....
 }
```

5.3 L’ interprete OTcl dei comandi

Per interpretare i comandi OTcl passati attraverso lo script, è necessario implementare la funzione *command()* appropriata.

Nel caso della nuova sorgente video è stato semplicemente implementato un interprete che rilevata la stringa **attach-agent**, consente di riconoscere l’oggetto agent sottostante e fa in modo che il puntatore agent_, definito come membro protected della classe base *Application*, punti all’oggetto corretto:

```
int video_app::command(int argc,const char*const* argv)
{
  Tcl& tcl=Tcl::instance();
  if (argc==3) {
      if (strcmp(argv[1],"attach-agent")==0) {
             agent_=(Agent*) TclObject::lookup(argv[2]);
             if (agent_==0) {
               tcl.resultf("no such agent %s",argv[2]);
               return(TCL_ERROR);
             }
             agent_->attachApp(this);
             return(TCL_OK);
      }
  }
  return(Application::command(argc,argv));
}
```

In tal modo, nel momento in cui vengono definite le istanze e collegati gli oggetti tra loro:

set app [*new Application/Traffic/video_app*]

set raps [*new Agent/RapMm*]

$app **attach-agent** $raps

ogni qualvolta servirà all’interno del codice C++ si potrà far riferimento all’agent sottostante, indipendentemente dal fatto che questo sia RAP o TFRC, tramite il puntatore agent_.

5.4 System calls ed upcalls

La comunicazione tra una qualsiasi applicazione e l’agent sottostante è resa possibile in NS attraverso l’utilizzo di particolari metodi, chiamati **system calls** ed **upcalls.**

Tramite le prime l'applicazione può utilizzare il servizio fornito dall'entità del livello sottostante. Tra le chiamate di sistema attualmente implementate in NS ne riportiamo una:

```
void sendmsg(int nbytes,const char*flags = 0){}
```

Il secondo parametro passato alla funzione può essere utilizzato per vari scopi, come ad esempio per permettere all'applicazione di indicare l'ultimo burst di pacchetti da spedire.
Le **upcalls** permettono invece all'agent di notificare all'applicazione particolari eventi, come quello della ricezione di un certo numero di bytes, e di consegnarglieli. Tra queste riportiamo la seguente:

```
void recv(int nbytes) {};
```

Si è già detto che le entità del caso in esame utilizzano per lo scambio delle informazioni la struttura del nuovo header *hdr_mapp*. Questo è possibile passando tale struttura come parametro di una system call o di una upcall.
Nel momento in cui l'applicazione video ha un certo numero di pacchetti da spedire in rete, risultanti dal suo processo di codifica, provvederà a riempire i campi informativi necessari nella struttura *hdr_mapp* e passerà tali informazioni all'agent tramite una chiamata alla funzione *sendmsg()*:

```
hdr_mapp mapp;
mapp.rc = 0;
mapp.packno = npacket_;
……
agent_->sendmsg(npacket_*size_,(char*)
                               &mapp);
```

Analogamente nel caso in cui sia l'agent a dover trasmettere delle informazioni alla sorgente video, ad esempio l'ultima stima del rate gestibile dalla rete. A tale scopo è stata definita, nella classe *Application*, una nuova **upcall**, la *recvmsg()*:

```
void recvmsg(int nbytes,const char *msg=0) {}
```

che verrà invocata dall'agent attraverso il puntatore alla classe *Application*, membro della classe *Agent*:

```
hdr_mapp mapp;
mapp.rc = 1;
mapp.rate = bit_rate_;
……
app_->recvmsg(mapp.nbytes,(char*)
                                &mapp);
```

Vale la pena sottolineare come i metodi appena visti vanno dichiarati come **virtual** nella definizione delle classi *Agent* ed *Application*, per poter essere implementati in maniera specializzata nelle sottoclassi secondo le necessità, permettendo così di attuare il polimorfismo necessario in una struttura articolata come quella di NS.
A titolo illustrativo, di seguito vengono riportati spezzoni della dichiarazione delle classi *Agent* ed *Application*, nei file rispettivamente "agent.h" ed "app.h":

```
class Agent : public Connector {
 public:
      Agent(packet_t pktType);
      virtual ~Agent();
      ……
      virtual void sendmsg(int nbytes, const char *flags = 0);
      virtual void send(int nbytes) { sendmsg(nbytes); }
      virtual void sendto(int nbytes, const char* flags, nsaddr_t dst);
      ……
 protected:
      ……
      Application *app_;
      ……
}

class Application : public Process {
public:
      Application();
      virtual void send(int nbytes);
      virtual void recv(int nbytes);
      virtual void recvmsg(int nbytes,const char *msg=0) {};
      virtual void resume();
      ……
protected:
      ……
      Agent *agent_;
      ……
```

```
}
```

Si osservi come molti metodi siano funzioni virtuali pure, che vengono semplicemente dichiarate senza fornire un'implementazione, a testimonianza del fatto che le classi *Agent* ed *Application* sono classi astratte, delle quali non possono esistere oggetti, ma la cui importanza risiede nel fatto di fornire un'interfaccia comune per tutte le sottoclassi da loro derivate.

5.5 Parametri configurabili e loro inizializzazione

Per completare l'implementazione dei nuovi moduli, bisogna ricordarsi di creare il collegamento tra le variabili OTcl e le corrispondenti C++ tramite l'utilizzo del metodo *bind()* nel costruttore delle nuove classi, come illustrato di seguito nel caso della sorgente video (file "video_app.cc"):

```
video_app::video_app():running_(0), sndtimer_(this), btimer_(this),
nextPkttime_(-1), wait_(-1), seqno_(0), packet_(0), packcount_(0)
{
      bind_bw("rate_", &rate_);
      bind ("random_", &random_);
      bind("packetSize_", &size_);
      bind("M_", &M_);
      bind("Nmaxprob_", &Nmaxprob_ );
      bind("frame_rate_", &frame_rate);
      bind("isteresi_", &isteresi_);
      bind("durata_isteresi_", &durata_isteresi_);
      bind("PSNR_", &PSNR_);
      bind("qs_", &qs_);
      bind("livello_qual_", &livello_qual_);
      bind("dji_", &dji_);
      bind("djb_", &djb_);
      bind("djp_", &djp_);
      bind("Xi_", &Xi_);
      bind("Xb_", &Xb_);
      bind("Xp_", &Xp_);
      bind("Ti_", &Ti_);
      bind("Tb_", &Tb_);
      bind("Tp_", &Tp_);
      bind("Res_", &Res_);
      bind("GOPSize_",&GOPSize_);
      bind_bool("q_sensitive_",&q_sensitive_); //default false
      bind("bit_rate_",&bit_rate);
      bind("network_rate_",&network_rate);
      bind("th_srtt_",&th_srtt_);
      bind("th_spsnr_",&th_spsnr_);
      bind("FRmin_",&FRmin_);
      bind("FRmax_",&FRmax_);
      bind("Stato_sorgente_", &stato_sorgente_);
      bind("v_attivity_", &v_attivity);
      bind("stato_attivita_", &stato_attivita_);
```

```
        bind("range_attivita_", &range_attivita_);
        bind("n_act_", &N_act);
        bind("Si_", &Si);
        bind("Sb_", &Sb);
        bind("Sp_", &Sp);
}
```

Senza entrare nel merito del significato delle variabili relative al caso in questione, si ricorda semplicemente che questo andrà fatto per tutti quei parametri che si può avere la necessità di settare o controllare durante il corso di una simulazione.
L'ultimo passo consiste nell'inizializzare gli stessi parametri ai loro valori di default, nel file "ns-default.tcl", come di seguito riportato nel caso dell'oggetto *video_app*:

```
Application/Traffic/video_app set rate_ 576;
Application/Traffic/video_app set random_ 0
Application/Traffic/video_app set packetSize_ 576;
Application/Traffic/video_app set M_ 24;
Application/Traffic/video_app set Nmaxprob_ 229;
Application/Traffic/video_app set frame_rate_ 4;
Application/Traffic/video_app set isteresi_ 0;
Application/Traffic/video_app set durata_isteresi_ 3;
Application/Traffic/video_app set PSNR_ 45;
Application/Traffic/video_app set qs_ 15;
Application/Traffic/video_app set livello_qual_ 3;
Application/Traffic/video_app set dji_ 0.0;
Application/Traffic/video_app set djb_ 0.0;
Application/Traffic/video_app set djp_ 0.0;
Application/Traffic/video_app set Xi_ 0;
Application/Traffic/video_app set Xb_ 0;
Application/Traffic/video_app set Xp_ 0;
Application/Traffic/video_app set Ti_ 0;
Application/Traffic/video_app set Tb_ 0;
Application/Traffic/video_app set Tp_ 0;
Application/Traffic/video_app set Res_ 0;
Application/Traffic/video_app set Nframe_ 5;
Application/Traffic/video_app set GOPSize_ 6;
Application/Traffic/video_app set q_sensitive_ false;
Application/Traffic/video_app set bit_rate_ 500;
Application/Traffic/video_app set network_rate_ 500;
Application/Traffic/video_app set FRmin_ 1;
Application/Traffic/video_app set FRmax_ 25;
Application/Traffic/video_app set th_srtt_ 0.04;
Application/Traffic/video_app set th_spsnr_ 45;
Application/Traffic/video_app set Stato_sorgente_ 0;
Application/Traffic/video_app set v_attivity_ 0;
Application/Traffic/video_app set stato_attivita_ 0;
Application/Traffic/video_app set range_attivita_ 0;
Application/Traffic/video_app set n_act_ 0;
Application/Traffic/video_app set Si_ 0.0;
Application/Traffic/video_app set Sb_ 0.0;
Application/Traffic/video_app set Sp_ 0.0;
```

5.6 Compilazione

Affinché i nuovi oggetti implementati possano essere riconosciuti e così utilizzabili dal simulatore, bisogna inserire i file oggetto corrispondenti nella lista che si trova all'interno del "Makefile", nella directory principale del nostro NS-2. Questo si realizza semplicemente aggiungendo le voci "nomefile.o" per ogni nuovo oggetto definito, ad esempio "video_app.o". Nel caso si siano voluti inserire i propri file all'interno di una nuova cartella per avere una struttura più ordinata, ovviamente andrà specificato il percorso completo: "nomecartella/nomefile.o".

A questo punto si potrà passare alla ricompilazione, con il comando "make", ricordando in ogni caso che è bene lanciare un "make depend" o "make clean" prima della ricompilazione, nel caso in cui siano state effettuate modifiche al "Makefile".