UNIVERSITA' DEGLI STUDI DI CAMERINO

FACOLTA' DI SCIENZE E TECNOLOGIE

Corso di Laurea in Informatica, Classe 26

Dipartimento di Matematica e Informatica

Simulazione di Eventi Discreti

Uno studio dei software OMNeT++ e NS-2 In ambiente Linux/Windows

Tesi di Laurea Compilativa

In

Reti di Calcolatori

Laureando:
Sergio Elia Tupini

Relatore:
Dott. Fausto Marcantoni

Anno Accademico 2007 / 2008

“ Il calcolatore è straordinariamente veloce, accurato e stupido. L'uomo è incredibilmente lento, impreciso e creativo. L'insieme dei due costituisce una forza incalcolabile.
È , inoltre , necessario considerare che il computer non è una macchina intelligente che aiuta le persone stupide; anzi è una macchina stupida che funziona solo nelle mani delle persone intelligenti.“

Indice

Introduzione

Scopo di questa tesi è quello di fornire una panoramica sulla simulazione dei sistemi ad eventi discreti, fornendo alcuni esempi di software utilizzabili a tal fine.
La generalità, intenzionalmente utilizzata nella stesura, ha come obiettivo quello di garantire al lettore la trasparenza e la comprensibilità degli argomenti trattati.
Come studente mi posso ritenere molto soddisfatto, ho intrattenuto rapporti di grande cordialità e cortesia con tutto il personale docente e non docente che colgo l'occasione per ringraziare sinceramente. Un ringraziamento speciale va al mio relatore, il professor Fausto Marcantoni, che ha avuto la tenacia e la costanza di seguirmi ed incoraggiarmi in ogni momento perché potessi raggiungere quest'ambizioso traguardo. Infine

un grazie è doveroso rivolgerlo ai miei genitori, ai parenti, agli amici, ai compagni di lavoro ed a tutte le persone che mi sono sempre state vicine contribuendo, inconsapevolmente, affinché questo giorno diventasse una realtà.

Capitolo 1

Simulazioni

Per simulazione si intende un modello della realtà che consente di valutare e prevedere lo svolgersi dinamico di una serie di eventi susseguenti all'imposizione di certe condizioni da parte dell'analista o dell'utente.
Un simulatore di volo, ad esempio, consente di prevedere il comportamento dell'aeromobile a fronte delle sue caratteristiche e dei comandi del pilota.
Le simulazioni sono uno strumento sperimentale molto potente e si avvalgono delle possibilità di calcolo offerte dall'informatica; la simulazione, infatti, non è altro che la trasposizione in termini logico-matematica-procedurali di un "modello concettuale" della realtà; tale modello concettuale

può essere definito come l'insieme di processi che hanno luogo nel sistema valutato e il cui insieme permette di comprendere le logiche di funzionamento del sistema stesso.

Le simulazioni possono anche avere carattere ludico; oggi esistono sul mercato diversi software (videogiochi di simulazione) che consentono di simulare il comportamento di persone, veicoli, civiltà.

Ovviamente il livello di approfondimento di tali simulazioni, in termini di modello concettuale sottostante, è più basso.

Nell'ambito delle simulazioni, acquisisce notevole importanza la simulazione del funzionamento dei processi produttivi e logistici.

Tali sistemi sono infatti caratterizzati da elevata complessità, numerose inter-relazioni tra i diversi processi che li attraversano, guasti dei segmenti, indisponibilità, stocasticità dei parametri del sistema.

Consideriamo, ad esempio, un impianto semplice per la produzione di un unico articolo, con solamente due macchine automatiche ed imballaggio manuale; in questo semplice sistema l'arrivo delle materie prime, la durata delle lavorazioni, il tempo necessario agli operatori per imballare sono tutte variabili stocastiche, in quanto il ritmo produttivo e di arrivo non è costante; inoltre, le macchine sono soggette a guasti e manutenzione, gli operatori possono non essere sempre disponibili etc.

Il progettista degli impianti industriali e il responsabile delle operations possono certamente avere interesse a valutare con anticipo l'effetto delle loro scelte su tali sistemi complessi, in termini, ad esempio, di capacità di produzione, tempo di attraversamento, scorte, blocchi.

Possono ,inoltre , avere dei problemi riguardo al dimensionamento di macchine, magazzini, flotta dei carrelli trasportatori e simili.

La simulazione, consentendo l'analisi della realtà ad un elevato livello di dettaglio e padroneggiando facilmente la complessità del sistema, fa si che alla fine sia possibile ottenere un gran numero di informazioni utili.

Il prezzo da pagare per tale completezza è ovviamente il tempo; le operazioni di programmazione sono infatti assai lunghe, affinché si possano ottenere dei dati sufficientemente sensati e tali da dare la possibilità di ottenere un modello della realtà ad essa aderente.

1.1 Passi e procedure

Al fine di poter procedere correttamente per avere un modello di simulazione utile e funzionante è opportuno procedere con una serie di passi:

- Definizione degli obiettivi e delle problematiche da esaminare: un' attenta analisi del problema consente di circoscriverne l'esame riducendo il successivo tempo di analisi;
- Stesura di un modello concettuale: consiste nella comprensione e modellazione del sistema produttivo che si intende simulare; questa fase è particolarmente importante in quanto definirà il comportamento dei diversi flussi di materiale e di informazioni che attraverseranno il modello.
- Validazione del modello concettuale: si tratta di un confronto con la direzione dell'impresa e con gli operatori per assicurarsi della capacità del modello di offrire un immagine consistente della realtà.
- Analisi dei dati in ingresso: la raccolta e l'analisi dei dati che diverranno la base per la definizione dei parametri di funzionamento del sistema (ad esempio: i diversi tempi di lavoro di una singola macchina).

Attraverso le tecniche del calcolo delle probabilità diviene possibile definire una distribuzione di probabilità per ogni parametro, da inserire all'interno del modello.

- Scrittura del modello in termini matematici
- Calibrazione e valutazione
- Definizione di un piano degli esperimenti: una singola iterazione ("run") di simulazione non ha alcun significato; rappresenta solo una delle possibili evoluzioni del sistema.

  È quindi opportuno effetture diversi "run" per poi analizzare i parametri in uscita.

  La lunghezza della singola iterazione e il numero delle iterazioni vengono determinate in questa fase.
- Analisi dei dati in uscita: dopo aver raccolto i dati relativi ai parametri, depurati da eventuali transitori è possibile creare degli intervalli di confidenza ovvero stimare il "range" di valori in cui i parametri che analizzano il problema proposto al primo passaggio possono oscillare.

1.2 Elementi di un modello di simulazione

- Entità - Le entità sono gli elementi "trattati" dal processo; tali "oggetti" hanno la caratteristica di essere "temporanei", e di subire passivamente le trasformazioni.
  Ad esempio, in un impresa di lavorazioni meccaniche, i semilavorati e le materie prime, che devono essere fresati, spianati etc possono essere modellati come "entità".
  Naturalmente, è possibile simulare anche processi in cui la produzione non riguardi un bene fisico, ma un servizio: in questo caso, le entità rappresenteranno informazioni, documenti, clienti, a seconda delle necessità.
  Le entità, all'interno del modello, possono essere considerate a loro volta come:
  - Anonime - Nella maggior parte dei casi, non interessa tenere traccia del singolo pezzo in lavorazione o in generale in transito nel sistema.
    Pertanto le entità non sono caratterizzate, e vengono considerate come un "flusso" indistinto.
  - Personalizzate - Caso duale del precedente, si presenta quando l'analista, spesso per il numero esiguo di pezzi in lavorazione, ha interesse a

considerare i parametri di lavorazione del singolo pezzo.

- Operazione: rappresenta una delle trasformazioni che avranno luogo sull'entità.

  Possono essere individuati due cicli di operazioni:

  - *Il ciclo macchina*: attinente agli stati ed operazioni che la macchina attraverserà, ovvero l'insieme di tutte le possibili successioni di operazioni e attese.
  - *Il ciclo pezzo*: rappresenta il percorso delle entità nel sistema, le macchine visitate e le operazioni subite

- Macchine: rappresentano gli elementi "fissi" del sistema, la cui definizione degli stati definisce univocamente la situazione generale del sistema, e delle quali sono di rilevanza per l'analista soprattutto le prestazioni.

  Le macchine possono essere fisiche, ed in questo caso ci si riferisce a macchine realmente presenti nel sistema da modellare, o "logiche", ed in questo caso compiono operazioni "fittizie" fisicamente, ma presenti logicamente nel sistema (ad esempio, il controllo di quantità in ingresso nell'impianto non ne provoca trasformazioni

"fisiche" ma lo "trasforma" da "lotto da controllare" a "lotto controllato").

- Stati: gli stati sono delle variabili (di tipo vario: possono essere numeri o valori logici) che descrivono lo stato del sistema e delle sue componenti, per ogni istante di tempo.

- Eventi: fenomeni che modificano lo stato del sistema (ad esempio, la fine di una lavorazione modifica lo stato di una macchina da "occupata" a "libera").

- Code: insiemi di entità che non possono accedere alle trasformazioni successive in quanto la macchina risulta occupata.

- Attributi: proprietà permanenti di un insieme di entità o di una macchina.

- Orologio locale: orologio che contiene, a livello di singola macchina, l'istante di tempo che identifica la fine della lavorazione in corso.

- Orologio generale: orologio che regola lo scorrere generale del tempo di simulazione.

Capitolo 2

Sistemi

Un **sistema**, nella sua accezione più generica è un insieme di entità connesse tra di loro tramite reciproche relazioni visibili o definite dal suo osservatore.
La caratteristica di un sistema può essere l'equilibrio complessivo che si crea fra le singole parti che lo costituiscono.
Ogni disciplina ha i suoi propri sistemi, sia a scopo funzionale, che a scopo strutturale/organizzativo, o con intenti di classificazione e ordinamento.
Le componenti di un sistema possono essere:

- *parti*, statiche o in movimento, riunite in un unico apparato o corpo;
- *grandezze fisiche, matematiche, numerarie, descrittive*, ecc. riunite in un unico sistema di riferimento o di misura, o di classificazione;
- *metodi e regole* che utilizzati insieme caratterizzano un'attività;
- *elementi strutturali* che costruiscono/fondano una rete con i nodi e gli archi.
- *elementi funzionali* per organizzazione e scopo, riuniti in un unico insieme che ne riassume le caratteristiche salienti e persegue obiettivi comuni.

In genere, un sistema è un concetto relativo, e può essere riconosciuto e classificato secondo della natura dei suoi componenti e le preferenze del suo osservatore/costruttore.

2.1 Sistemi ad eventi discreti

Un sistema ad eventi discreti è un sistema dinamico i cui *stati* possono assumere valori logici o simbolici, piuttosto che numerici, e il cui comportamento è caratterizzato dall'occorrenza di *eventi* istantanei che si verificano con un cadenzamento irregolare non necessariamente noto a priori. Il comportamento di tali sistemi è descritto, appunto, in termini di stati e di eventi.

Si consideri un deposito di parti in attesa di venir lavorate da una macchina. Si suppone che il numero di parti in attesa non possa superare le due unità e che la macchina possa essere in lavorazione oppure guasta.

Lo stato del sistema complessivo è dato dallo stato della macchina e dal numero di parti in attesa nel deposito. Sono dunque possibili sei stati:

- L_0,L_1,L_2: macchina in lavorazione; deposito: vuoto, con una parte o con due parti.
- G_0,G_1,G_2: macchina guasta; deposito: vuoto, con una parte o con due parti.

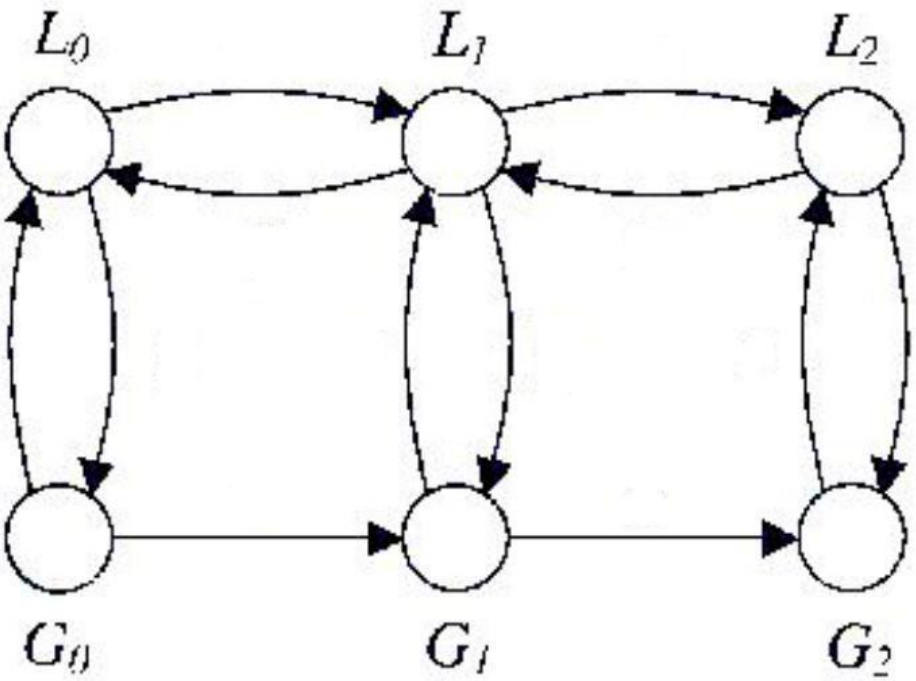
Gli eventi che determinano un cambiamento di stato sono:

- *a*: arrivo di una nuova parte nel deposito;
- *p*: prelievo da parte della macchina di una parte dal deposito;

- *g*: la macchina si guasta;
- *r*: la macchina viene riparata.

L'evento *a* può sempre verificarsi purché il deposito non contenga due parti (in tal caso non possono arrivare nuove parti); tale evento modifica lo stato da L_i (ovvero G_i) a L_{i+1} (ovvero G_{i+1}). L'evento *p* può verificarsi solo se il deposito non è vuoto e la macchina è in lavorazione; tale evento modifica lo stato da L_i a L_{i-1}. Infine gli eventi *g* e *r* determinano, rispettivamente, il passaggio da L_i a G_i e viceversa.

Tale comportamento può essere descritto formalmente mediante il modello in figura che assume la forma di un automa a stati finiti.



Esistono sistemi intrinsecamente ad eventi discreti quale il sistema descritto nell'esempio precedente. Molti sistemi di questo tipo si trovano nell'ambito della produzione, della robotica, del traffico, della logistica (trasporto e immagazzinamento di prodotti, organizzazione e consegna di servizi) e delle reti di elaboratori elettronici e di comunicazioni. Altre volte, dato un sistema la cui evoluzione potrebbe essere descritta con un modello ad avanzamento temporale, si preferisce descrivere il suo comportamento in termini di eventi, rinunciando ad una descrizione in termini di segnali al fine di mettere in evidenza i soli fenomeni di interesse.

Capitolo 3

Reti

Col termine **rete** si intende comunemente una serie di componenti, sistemi o entità interconnessi tra di loro.
Una **rete di calcolatori** è un sistema che permette la condivisione di informazioni e risorse (sia hardware che software) tra diversi calcolatori.
Il sistema fornisce un servizio di trasferimento di informazioni ad una popolazione di utenti distribuiti su un'area più o meno ampia.

Le reti di calcolatori generano traffico di tipo fortemente impulsivo, a differenza del telefono, e per questo hanno dato origine - e usano tuttora - la tecnologia della commutazione di pacchetto.

La costruzione di reti di calcolatori può essere fatta risalire alla necessità di condividere le risorse di calcolatori potenti e molto costosi (mainframe).

La tecnologia delle reti, e in seguito l'emergere dei computer personali a basso costo, ha permesso rivoluzionari sviluppi nell'organizzazione delle risorse di calcolo.

Si possono indicare almeno tre punti di forza di una rete di calcolatori rispetto al mainframe tradizionale:

- *fault tolerance* (resistenza ai guasti): il guasto di una macchina non blocca tutta la rete, ed è possibile sostituire il computer guasto facilmente (la componentistica costa poco e un'azienda può permettersi di tenere i pezzi di ricambio in magazzino);
- *economicità*: come accennato sopra, hardware e software per computer costano meno di quelli per i mainframe;
- *gradualità della crescita e flessibilità*: l'aggiunta di nuove potenzialità a una rete già esistente e la sua espansione sono semplici e poco costose.

Tuttavia una rete ha alcuni punti deboli rispetto a un mainframe:

- *scarsa sicurezza*: un malintenzionato può avere accesso più facilmente ad una rete di computer che ad un mainframe: al limite gli basta poter accedere fisicamente ai cablaggi della rete.

  Inoltre, una volta che un *virus* o, peggio, un *worm* abbiano infettato un sistema della rete, questo si propaga rapidamente a tutti gli altri e l'opera di disinfezione è molto lunga, difficile e non offre certezze di essere completa;
- *alti costi di manutenzione*: con il passare del tempo e degli aggiornamenti, e con l'aggiunta di nuove funzioni e servizi, la struttura di rete tende ad espandersi e a diventare sempre più complessa, e i computer che ne fanno parte sono sempre più eterogenei, rendendo la manutenzione sempre più costosa in termini di ore lavorative.

  Oltre un certo limite di grandezza della rete (circa 50 computer) diventa necessario eseguire gli aggiornamenti hardware e software su interi gruppi di computer invece che su singole macchine, vanificando in parte il vantaggio dei bassi costi dell'hardware.

Capitolo 4

OMNeT++

OMNeT++ è un simulatore di eventi discreti orientato agli oggetti.

Il simulatore può essere utilizzato per:

- Modellare il traffico nelle reti di comunicazione
- Modellare i protocolli
- Modellare sistemi hardware multiprocessore o distribuiti
- Validare architetture hardware

- Valutare le caratteristiche e le prestazioni di sistemi software complessi
- . . . in generale modellare ogni altro sistema dove l'approccio ad eventi discreti è adatto

Un modello OMNeT++ consiste in un insieme di moduli annidati.

La profondità di annodamento dei moduli non è limitata; questo consente all'utente di riflettere la struttura logica dell'ambiente reale nel modello di simulazione.

Più modelli possono comunicare tra di loro attraverso lo scambio di messaggi.

Un modello può contenere, al suo interno, strutture dati arbitrariamente complesse.

I moduli possono inviare i loro messaggi direttamente alla destinazione o lungo un percorso predefinito, avvalendosi di porte e connessioni.

I moduli possono avere parametri propri.

I parametri possono essere utilizzati per personalizzarne il comportamento e parametrizzarne la topologia.

I moduli al livello più basso della gerarchia incapsulano il comportamento di tutti gli altri.

Questi moduli sono definiti "Simple Modules" (Moduli Semplici) e sono programmati in C++ utilizzando le librerie di simulazione.

Le simulazioni di OMNeT++ sono caratterizzate da varie interfacce utente, ognuna con scopi differenti: debug, dimostrazione, esecuzione batch, . . .

Interfacce utente avanzate consentono all'utente di visualizzare l'intero modello , di controllarne l'esecuzione ed anche di intervenire, modificando le variabili / oggetti all'interno del modello (il che è molto utile in fase di progettazione/debug della simulazione).

Il simulatore, così come le interfacce utente ed i vari tools (strumenti), è portatile; essi infatti sono noti per funzionare sotto Windows e sotto varie distribuzioni Unix, utilizzando diversi compilatori C++.

4.1 Modellazione

Il simulatore fornisce all'utente strumenti efficaci per descrivere la struttura del sistema reale.

Alcune delle caratteristiche principali sono :

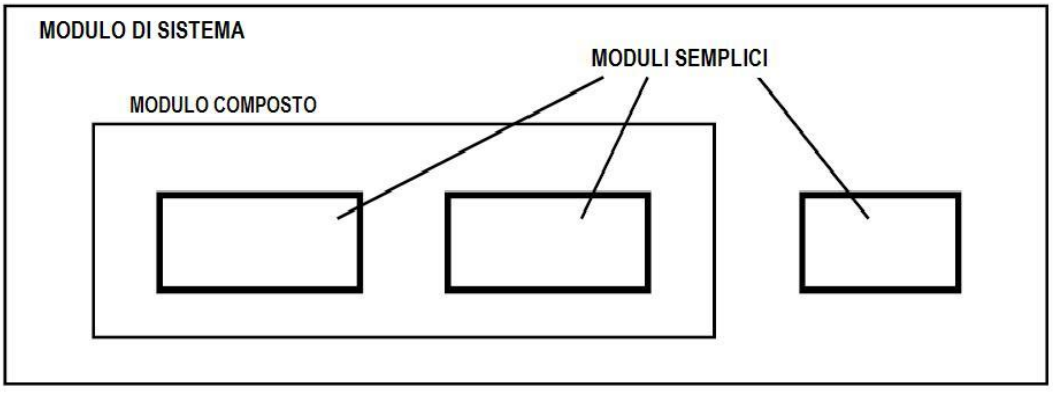
- Moduli nidificati gerarchicamente
- Moduli istanze di tipologie predefinite
- Moduli che comunicano con messaggi attraverso dei canali
- Parametri modulo flessibili
- Linguaggio di descrizione della topologia

Gerarchia dei moduli

Un modello OMNeT consiste in un insieme di moduli nidificati gerarchicamente, che comunicano tra loro attraverso lo scambio di messaggi.

Per tale motivo i modelli vengono spesso chiamati "Reti".

Il livello più alto del modulo è il modulo di sistema. Esso contiene vari sottomoduli, che a loro volta possono contenere altri sottomoduli.



La struttura del modello è definita tramite il linguaggio NED (NEtwork Description) di OMNeT++.
Esso è caratterizzato da una struttura particolarmente semplice. Contrariamente a quanto si pensi, esso è uno strumento estremamente potente, soprattutto quando si è in materia di definizioni di topologie regolari come i modelli a catena, ad anello, a maglia, ad ipercubo per poi arrivare alle strutture ad albero.
I moduli che contengono uno o più sottomoduli sono definiti Composti, in contrapposizione ai moduli semplici (i quali sono al livello più basso della gerarchia e contengono gli algoritmi del modello).
Sta all'utente, quindi, implementare i moduli semplici in C++, utilizzando la libreria delle classi di simulazione di OMNeT++.

Tipologie

In fase di editing del modello l'utente definisce le varie tipologie di moduli; istanze di questi moduli serviranno come componenti per istanze ancora più complesse.
Infine, l'utente crea il modulo di sistema come istanza di una delle tipologie precedentemente definite; tutti i moduli della

rete sono istanziati come sotto-moduli e sottosotto-moduli del modulo di sistema.
Le definizioni delle tipologie dei vari moduli possono essere memorizzate in file separati da quello del loro utilizzo.
Questo significa che l'utente può raggruppare tipologie di moduli esistenti e creare librerie di componenti.

Definizione dinamica

In alcune situazioni si può avere la necessità di creare, o magari distruggere, uno o più moduli dinamicamente.
Ad esempio, quando si simula una "mobile network" si può voler creare un nuovo modulo nel momento in cui un utente entra in una determinata area.
A tal fine si potrebbe creare un modulo manager che riceverà le richieste di connessione e definirà un nuovo modulo per ogni connessione.
Da notare che nel momento in cui si crea dinamicamente un modulo composto, tutti i suoi sottomoduli vengono creati a cascata.
Una volta definito ed avviato, un modulo dinamico non è molto differente da un modulo statico; si potrebbero, infatti, sostituire i moduli statici con i dinamici durante la connessione (ma forse prima è il caso di identificarne l'utilità . . .) .

Per comprendere il funzionamento della definizione dinamica dei moduli è necessario avere una idea di come OMNeT instanzia i suoi moduli.
Ogni tipologia di modulo (classe) ha un oggetto di definizione corrispondente. Tale oggetto viene creato sotto la copertura di alcune macro ed ha una funzione di definizione che può istanziare la classe.
Una volta memorizzato il puntatore, è possibile richiamare tale funzione di definizione e creare una istanza della classe-modulo corrispondente; tutto questo senza dover scrivere codice C++ per includere i file header sorgenti contenenti le dichiarazioni delle classi del modulo.
L'oggetto conosce anche quali porte e parametri deve avere il modulo dato (questa informazione gli viene dal codice NED compilato).
I moduli semplici possono essere creati in un unico passo.
Per un modulo composto la situazione è un po' più complicata, dato che la sua struttura interna (connessioni, sottomoduli, ecc. . . .) può dipendere da valori , contenuti in parametri, e/o da la dimensione di eventuali vettori di porte.
Per questo motivo, per i moduli composti, è richiesta per prima cosa la creazione del modulo stesso.
Secondariamente, vengono assegnati i valori ai parametri e le dimensioni di eventuali vettori di porte.

Infine viene chiamata la funzione che definisce le connessioni e la struttura interna del modulo.
Come già probabilmente si sarà intuito, i moduli che utilizzano determinate funzioni di inizializzazione necessitano di un messaggio iniziale che funga da evento scatenante.
Per i moduli creati staticamente, questo messaggio viene creato automaticamente da OMNeT.
Per quelli creati dinamicamente, invece, è necessario richiamare e creare una istanza di tale classe attraverso le funzioni specifiche.
Tale chiamata deve avvenire dopo la definizione/inserimento del messaggio di avvio, poiché il codice di inizializzazione potrebbe, a sua volta, inserire un messaggio nella coda FES.
In questo caso, nonostante si tratti della funzione di inizializzazione, tale messaggio deve comunque essere elaborato dopo il messaggio di avvio.

Messaggi, porte, link

I moduli comunicano tra di loro attraverso lo scambio di messaggi.
In un ambiente reale, i messaggi possono rappresentare frame, pacchetti o altri tipi di entità.

I messaggi possono contenere strutture dati arbitrariamente complesse.

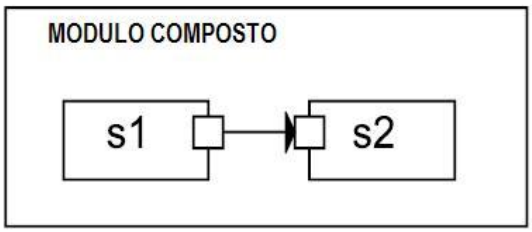
I moduli semplici possono spedire i messaggi direttamente alla loro destinazione, o tramite percorsi predefiniti, attraverso porte e connessioni.

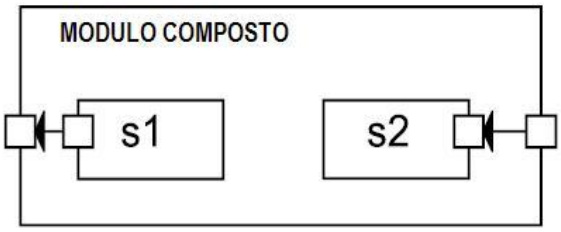
Il “Local Simulation Time” di un modulo anticipa quando esso riceve un messaggio.

Il messaggio può arrivare da un altro modulo, oppure può essere auto spedito.

Le porte sono le interfacce input ed output dei moduli, dato che i messaggi sono spediti (e ricevuti) tramite di esse.

All’interno di un modulo composto si possono collegare le porte corrispondenti di due sottomoduli, oppure una porta di un sottomodulo ed una del composto che lo contiene.





A causa della struttura gerarchica del modello, i messaggi tipicamente viaggiano attraverso una serie di connessioni, per partire ed arrivare in moduli semplici.

Tali serie di connessioni, che vanno da un modulo semplice ad un altro modulo semplice sono dette “Route”.
I moduli composti , in tale scenario, fungono da “scatole di cartone”; in tale modo essi consentono una trasmissione trasparente tra l’esterno ed i sottomoduli al loro interno.

4.2 Concetti di simulazione

Un “sistema ad eventi discreti” è un ambiente in cui i cambiamenti di stato (eventi) non solo accadono ad istanze di tempo precise e predefinite, ma impiegano un tempo pari a zero per accadere.
Si presume, infatti, che non accada nulla tra due eventi consecutivi o, per meglio dire, che non vi siano cambi di stato in tale lasso di tempo simulato (in totale contrasto con quella che è la gestione del tempo nei sistemi a flusso continuo).
Un esempio si tutti sono le reti di computer, le quali generalmente vengono considerate proprio sistemi ad eventi discreti.
Alcuni esempi di eventi all’interno di tale tipologia di sistema sono:

- L’inizio di una trasmissione di pacchetti

- La fine di una trasmissione di pacchetti
- La scadenza di un timeout, con relativa ritrasmissione.

Questo implica che tra due eventi, come ad esempio l'inizio e la fine di una trasmissione di pacchetti, non accada nulla di interessante.

Notare che la definizione di "eventi interessanti" e stati dipende sempre dagli intenti e dagli obiettivi del soggetto che definisce la simulazione.

Se il programmatore è interessato, ad esempio, alla trasmissione di singoli bit includerà qualcosa come "inizio trasmissione di bit" e "fine trasmissione di bit" tra gli eventi da gestire.

In OMNeT il tempo in cui gli eventi si verificano è chiamato evento "TimeStamp", anche se spesso tale definizione viene sostituita con quella di "arrival time" (dato che la prima coincide con una parola riservata ai settagli dei parametri degli oggetti della classe Event).

Eventi

OMNeT utilizza i messaggi per rappresentare gli eventi.
Ogni evento è rappresentato da una istanza della classe cMessage, o una sua sottoclasse.
I messaggi vengono inviati da un modulo all'altro; questo significa che il luogo ed il tempo in cui "accade l'evento" non sono altro che i modulo di destinazione del messaggio ed il tempo di arrivo di quest'ultimo.
Eventi come la "scadenza di timeout" vengono implementati tramite l'auto invio di messaggi che un particolare modulo esegue su se stesso.
Il tempo di simulazione, in OMNeT, è memorizzato all'interno di una particolare classe scritta in C++, le cui istanze hanno dei parametri in formato Double.
Gli eventi sono contenuti in quello che, con poca fantasia, viene definito "Future Event Set" (Insieme di eventi che devono accadere).
Gli eventi ivi contenuti vengono consumati in base al loro "arrival time", al fine di mantenerne la casualità.
Più precisamente, dati due messaggi, vengono applicate le seguenti regole :

- Il messaggio con arrival time minore viene eseguito per primo; se due messaggi ha tale parametro uguale

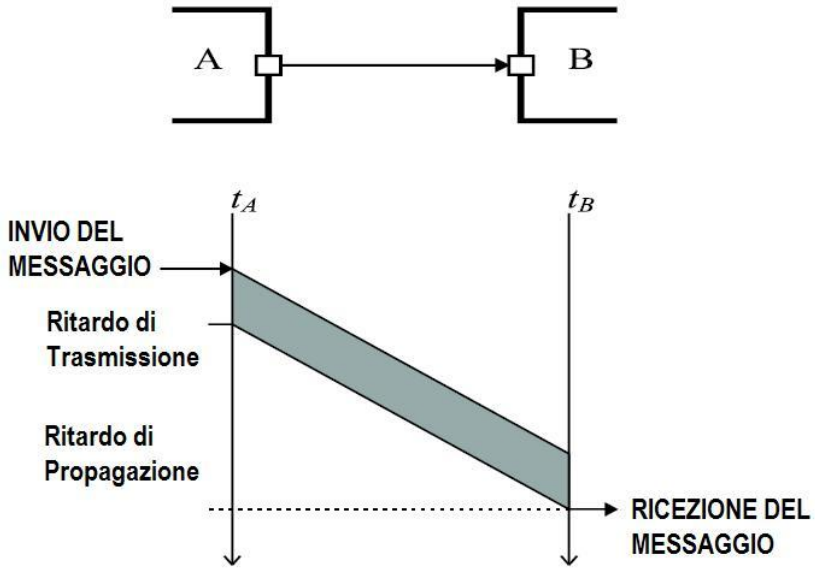
- Quello con valore di priorità minore viene eseguito per primo; se due messaggi hanno anche tale parametro uguale
- Quello schedulato o spedito precedentemente viene eseguito per primo.

La priorità non è altro che un attributo intero, della classe di definizione del messaggio, assegnato dall'utente.
Tornando al "Future Event Set", la sua implementazione è un fattore cruciale nelle prestazioni di un simulatore ad eventi discreti.
In OMNeT, il FES è implementato utilizzando una struttura dati definita "Heap Binario", che è anche quella più utilizzata per questo obiettivo.

4.3 Trasmissione e Comunicazione

Alle connessioni possono essere assegnati 3 parametri opzionali, i quali facilitano la modellazione delle reti di comunicazione e che possono essere utili anche in altri ambienti:

- Propagation delay
- Bit error rate
- Data rate



Tali parametri possono essere impostati per ogni connessione, oppure in base alla tipologia e all'uso che l'intero modello di simulazione ne fa.

Il “Propagation delay” rappresenta la quantità di tempo di cui l’arrivo del messaggio viene ritardato quando viaggia attraverso il canale.

Il “Bit error rate” specifica la probabilità con cui un bit è trasmesso in maniera non corretta, e consente di avere una semplice modellazione del rumore del canale.

Il “Data rate” è specificato in bit/secondo e viene utilizzato per il calcolo del tempo di trasmissione di un pacchetto.

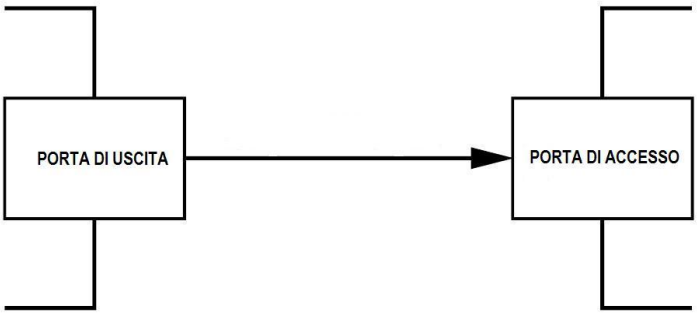
Quando si utilizzano i tassi di dati , l’invio dei messaggi nel modello corrisponde alla trasmissione del primo bit, mentre l’arrivo corrisponde alla ricezione dell’ultimo bit.

Questo modello non è sempre applicabile, ad esempio i protocolli come Token Ring o FDDI non aspettano che il frame arrivi nella sua interezza.

Se si vuole modellare questa tipologia di reti non si può utilizzare il parametro “Data rate” fornito da OMNeT++.

Trasmissioni multiple

Se il parametro “Data rate” viene specificato per una connessione, un messaggio avrà un tempo di trasmissione; tale tempo sarà maggiore di zero e dipenderà dalla lunghezza della connessione.
Questo implica che un messaggio, in fase di attraversamento di una porta di output, “riserva” la porta per un determinato periodo (nel quale esso sarà in fase di trasmissione).



Nel lasso di tempo in cui un messaggio è in fase di trasmissione, gli altri messaggi che devono utilizzare la suddetta porta devono attendere il completamento di tale fase.
È comunque ancora possibile inviare messaggi mentre la porta è occupata, anche se questo comporterà un ritardo nella fase di avvio della trasmissione.

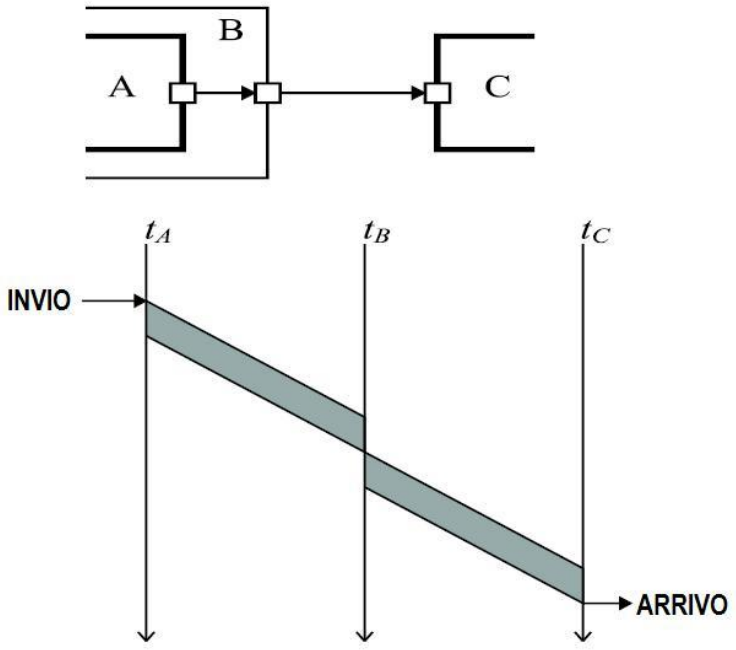
Tale gestione permette di simulare una eventuale coda interna al modulo cui appartiene la porta, nella quale dovranno attendere tutti quei messaggi che sono in attesa.

La libreria della classi di OMNeT fornisce funzioni che consentono di controllare se una determinata porta di output è in fase di trasmissione e il momento in cui tale quest'ultima termina.

Se la connessione, con impostato il parametro "data rate", non è direttamente collegata alla porta di output del modulo semplice che rappresenta la sorgente della trasmissione, si dovranno controllare tutte le porte che compongono la frazione di percorso precedente.

Invio dei messaggi

L'invio di un messaggio è implementato nel seguente modo: l' "arrival time" ed il flag "bit error" vengono elaborati immediatamente dopo l'invocazione della funzione di invio.
Ciò implica che, se il messaggio viaggerà attraversi diversi link prima di raggiungere la sua destinazione, esso non verrà schedulato individualmente per ogni link.
Ogni calcolo viene effettuato una sola volta, e precisamente nell'invocazione della funzione di invio.
I progettisti del simulatore hanno scelto questa implementazione perché essa gode di una maggiore efficienza a run-time.



Nell'implementazione attuale delle code alle porte occupate e nella modellazione del ritardo dei pacchetti, questi ultimi non vengono realmente accodati in tali porte.
Le porte, infatti, non hanno una gestione degli accodamenti.
Così come il tempo in cui una porta termina il suo stato di occupazione, dovuto ad una operazione di trasmissione in corso, è conosciuto nel momento in cui un messaggio viene spedito; allo stesso modo l'arrival-time del messaggio può essere calcolato in anticipo.
Il messaggio sarà, quindi, memorizzato nel FES (Insieme di eventi futuri) fino al momento in cui il suo tempo di arrivo sarà scattato, ed il modulo di destinazione sarà padrone del messaggio.

Tale implementazione ha le seguenti conseguenze.
Se il parametro indicante il ritardo di un link (o gli altri parametri che possono essere assegnati ad una connessione) viene modificato durante la simulazione, la modellazione dei messaggi inviati poco prima di tale modifica non ne verrà influenzata.
Vale a dire; se i parametri di un link variano nel mentre del passaggio di un messaggio nel modello, quel messaggio non ne viene influenzato (anche se dovrebbe).
Comunque, tutte le modellazioni successive di messaggi saranno aggiornate correttamente.

Vale la stessa regola per il parametro “data-rate”; se tale parametro viene modificato durante la simulazione, le modifica influenzeranno solamente le definizioni di invii successive ad essa.
Se si ha la necessità di modellare parametri che verranno modificati a run-time, si può prendere una di queste strade:

- Definire un modulo mittente in modo tale da schedulare e trasmettere gli eventi solamente nel momento in cui la coda termina una sua eventuale operazione di trasmissione in corso.
- Alternativamente, si possono implementare canali come moduli semplici (canali attivi) per poi definirne il comportamento.

Attributi

Un oggetto della classe cMessage ha un determinato numero di attributi.
Alcuni vengono utilizzati, principalmente, dal kernel di simulazione.
Altri hanno come unico scopo quello di rendere più agevole il lavoro del programmatore.

Definendo una lista dei parametri della classe cMessage, o di una sua sotto classe, non si può non considerate i seguenti:

- *Nome* : normalmente consiste in una stringa che viene utilizzata frequentemente dal programmatore della simulazione.

  Tale parametro appare in molti luoghi del visualizzatore TKENV (ad esempio nelle animazioni) ed è, generalmente, molto utile per identificare un nome descrittivo.
- *Tipologia* : è pensato come portatore di alcune informazioni che consentono l'identificazione della classe di definizione del messaggio, con la relativa decodifica di questo ultimo.

  È possibile utilizzare un qualsiasi valore positivo, o al massimo 0, per la gestione di questo parametro.
  Valori negativi sono riservati per la libreria di simulazione di OMNeT.
- *Lunghezza* : (intesa in bit) è utilizzata per elaborare il ritardo di trasmissione provocato dal passaggio del messaggio in una connessione cui è stato assegnato un parametro "data rate" maggiore di 0.
- *Bit error flag* : utilizzato per determinare la probabilità di una variazione (errore) nel passaggio di un messaggio nel suddetto canale.

- *Priorità* : è utilizzato dal kernel di simulazione per definire un ordinamento, interno alla coda FES, per quei messaggi che hanno lo stesso Arrival Time.
- *Time stamp* : non viene utilizzato dal kernel di simulazione. Se ne può avvalere il programmatore che intende rilevare il momento in cui il messaggio viene accodato, oppure rinviato.
- Altri attributi e dati membri che rendono la programmazione della simulazione più semplice.
- Un insieme di attributi di sola lettura che memorizzano informazioni riguardanti l'ultimo invio o l'ultima elaborazione del messaggio.

  Essi vengono utilizzati principalmente dal kernel di simulazione nel lasso di tempo in cui il messaggio si trova nella coda FES, anche se tali informazioni si trovano ancora nel messaggio nel momento in cui esso viene spedito e ricevuto dal modulo.

Parametri

I moduli definiti tramite le librerie forniti dall'ambiente possono avere dei parametri.
I valori di tali parametri possono essere assegnati nei file di definizione della topologia (file NED) o nel file di configurazione "omnetpp.ini".
I parametri possono essere utilizzati per personalizzare il comportamento dei moduli semplici , o per parametrizzarne la topologia del modello.
I parametri possono ricevere stringhe, valori interi o booleani oppure contenere alberi di dati XML.
Per valori si intende anche espressioni che utilizzano altri parametri o che richiamano funzioni C, variabili casuali da differenti distribuzioni o valori di input inseriti interattivamente dall'utente.
I parametri passati come valori numerici possono essere utilizzati per definire la topologia in modo flessibile.
In un modulo composto, i parametri possono indicare il numero di sottomoduli, il numero delle porte e le modalità in cui vengono definite le connessioni interne.

Definizione dei messaggi

È considerato necessario aggiungere delle varianti, o dei semplici parametri, ad un messaggio predefinito per considerare quest' ultimo realmente utile.
Ad esempio, se si sta modellando una rete di comunicazione di pacchetto è necessario prevedere un modo per memorizzare i campi che compongono l'intestazione di una pacchetto in oggetti di tipo messaggio.
Poiché la classe di definizione del messaggio è scritta in C++, il modo più naturale per estenderla è quello di "subclassarla".
Comunque, dato che per ogni campo è necessario definire almeno tre parametri (il dato che lo rappresenta e due funzioni, una di getting ed una di setting) e la classe risultante deve essere integrata con l'ambiente di simulazione, la scrittura del codice C++ necessario può risultare un processo noioso e costoso, soprattutto se si considera il fattore "Tempo".
OMNeT offre una modalità più conveniente chiamata "message definition".
Tale modalità propone una sintassi molto compatta per al definizione dei messaggi.
Il codice C++ viene automaticamente generato dalle stringhe di definizione.
Una fonte comune di denuncia su tale meccanismo è quella della perdita di flessibilità.

Infatti, non è possibile variarne e/o personalizzarne il comportamento o il funzionamento.
OMNeT, comunque, ti consente di personalizzare le classi generate in varie misure. Resta il fatto che la message definition risparmi un gran lavoro al programmatore.
La funzione di sub-classificazione, in OMNeT, è ancora un po' sperimentale e questo ci consente due lievi critiche sul suo utilizzo :

- In furto, la sintassi ed i costrutti di definizione dei messaggi possono variare (principalmente come risposta a feedback della community) e costringere i programmatori a ridefinire i propri messaggi
- Il compilatore che traduce il linguaggio di definizione dei messaggi in C++ è uno script PERL: "opp_msgc". È chiaro che questa non è altro che una soluzione temporanea, in attesa del suo gemello, o diretto discendente, scritto in C++.

4.4 Programmazione degli algoritmi

I moduli semplici di un modello contengono algoritmi definiti tramite funzioni C++.

A tal fine si può utilizzare la piena potenza e flessibilità di tale linguaggio, le quali sono supportate dalla libreria di simulazione fornita da OMNeT++.

Il programmatore della simulazione può così scegliere tra descrizioni “event-driven” e “process-style” ed utilizzare liberamente i concetti della programmazione ad oggetti (l’ereditarietà, il polimorfismo, ecc) per estendere le funzionalità del simulatore.

Gli oggetti della simulazione (i messaggi, i moduli, le code, ecc) sono, infatti, rappresentati da classi C++.

Essi sono stati definiti per cooperare efficientemente al fine di creare un potente ambiente di sviluppo per la programmazione delle simulazione.

Alcuni esempi delle classi della libreria di simulazione sono:

- Moduli, porte, connessioni, ecc
- Parametri
- Messaggi
- Classi contenitori (array, code, ecc)
- Classi di collezioni di dati
- Classi di stima statistica (grafici)

Alcune classi sono appositamente strumentalizzate per consentire all'utente di "attraversare" gli oggetti che compongono la simulazione in esecuzione al fine di visualizzare informazioni come il nome dell'oggetto, il nome della classe, le variabili di stato o il loro contenuto.

Questa caratteristica ha consentito di creare una interfaccia grafica di simulazione dove tutti i componenti interni della simulazione sono visibili.

4.5 Utilizzando OMNeT++

Un modello OMNeT++ si compone delle seguenti parti:

- File di definizione della topologia (con estensione .NED) i quali descrivono la struttura del modulo con parametri, porte, ecc. essi possono essere scritti utilizzando un semplice editor di testo oppure l'editor grafico GNED fornito dell'ambiente di sviluppo OMNeT.
- File di definizione dei messaggi (con estensione .MSG). è possibile definire varie tipologie di messaggi ed aggiungere ad esse vari campi di dati. Sarà poi OMNeT a tradurre tali definizioni in vere e proprie classi C++.
- File sorgenti dei moduli semplici. Essi sono file contenenti codice C++ (con estensione .h / .cc)

Il sistema di simulazione, invece, fornisce i seguenti componenti :

- Kernel di simulazione : esso contiene il codice di gestione e la libreria di classi della simulazione. Esso è scritto in C++, compilato e messo insieme per formare una libreria (un file con estensione .a o .lib).
- Interfacce utente : tali interfacce vengono utilizzate durante l'esecuzione, in fase di debug, in dimostrazione

o per l’esecuzione “batch” delle simulazioni. Ci sono diverse interfacce utente scritte in C++, compilate e poste insieme per formare un’unica libreria (con estensione .a o .lib)

I programmi che definiscono le simulazioni sono composti da quanto definito sopra.

Per prima cosa, i file di definizione dei messaggi (.msg) vengono tradotti in codice C++ utilizzando il programma “OPP_MSGC”.

A questo punto tutti i sorgenti C++ vengono compilati e legati con il kernel di simulazione e la libreria delle interfacce utente per definire l’eseguibile della simulazione.

I file .NED, di definizione della topologia.

Possono essere tradotti in codice C++ (utilizzando NEDTOOL) e legati insieme, oppure caricati dinamicamente, nella loro forma testuale originale quando il programma di simulazione viene eseguito.

Analisi dei risultati

L'eseguibile della simulazione è un programma "StandAlone", quindi può essere eseguito in altre macchine, prive di OMNeT++, o che i file di definizione dei moduli siano presenti.

Quando il programma viene eseguito, esso legge un file di configurazione (normalmente chiamato "omnetpp.ini"). questo file contiene i dati di setting (che indicano le modalità di esecuzione della simulazione), i valori per i parametri dei moduli, ecc.

Tale file può anche prevedere più esecuzioni di simulazioni che, nel caso più semplice, verranno eseguite sequenzialmente.

L'output della simulazione viene riportato in alcuni file di dati: file vettori, file scalari o file di output definiti dall'utente stesso.

OMNeT fornisce uno strumento, con tanto di interfaccia grafica, chiamato "Plove" che consente di visualizzare i dati contenuti nei vettori di output che, eventualmente, possono essere utilizzati per definire dei grafici.

Non ci si aspetta che i dati contenuti nei file di output prodotti da OMNeT siano utilizzati, ed analizzati, autonomamente.

Essi, infatti, non sono altro che file di testo, in un formato che può essere analizzato da strumenti matematici come

“MATLAB” o “OCTAVE” , oppure importato in fogli elettronici come “CALC” di Open Office o Excel di Microsoft Office.
Sono proprio questi programmi esterni ad offrire ricche funzionalità per le analisi statistiche e per la visualizzazione.

Definizione delle simulazione

Come si è già accennato, un modello OMNeT consiste, fisicamente, nelle seguenti parti:

- file di descrizione della topologia scritti in linguaggio NED (hanno estensione .ned)
- file di definizione dei messaggi (estensione .msg)
- implementazione dei moduli semplici o altro codice C++ (estensione .cc , [.cpp sotto Windows])

Per definire l'eseguibile del programma di simulazione, è anzi tutto necessario tradurre i file di descrizione della topologia e quelli di definizione dei messaggi in codice C++ compilabile. Tale compito spetta al compilatore Ned (nedtool) ed a quello dei messaggi (opp_msgc).

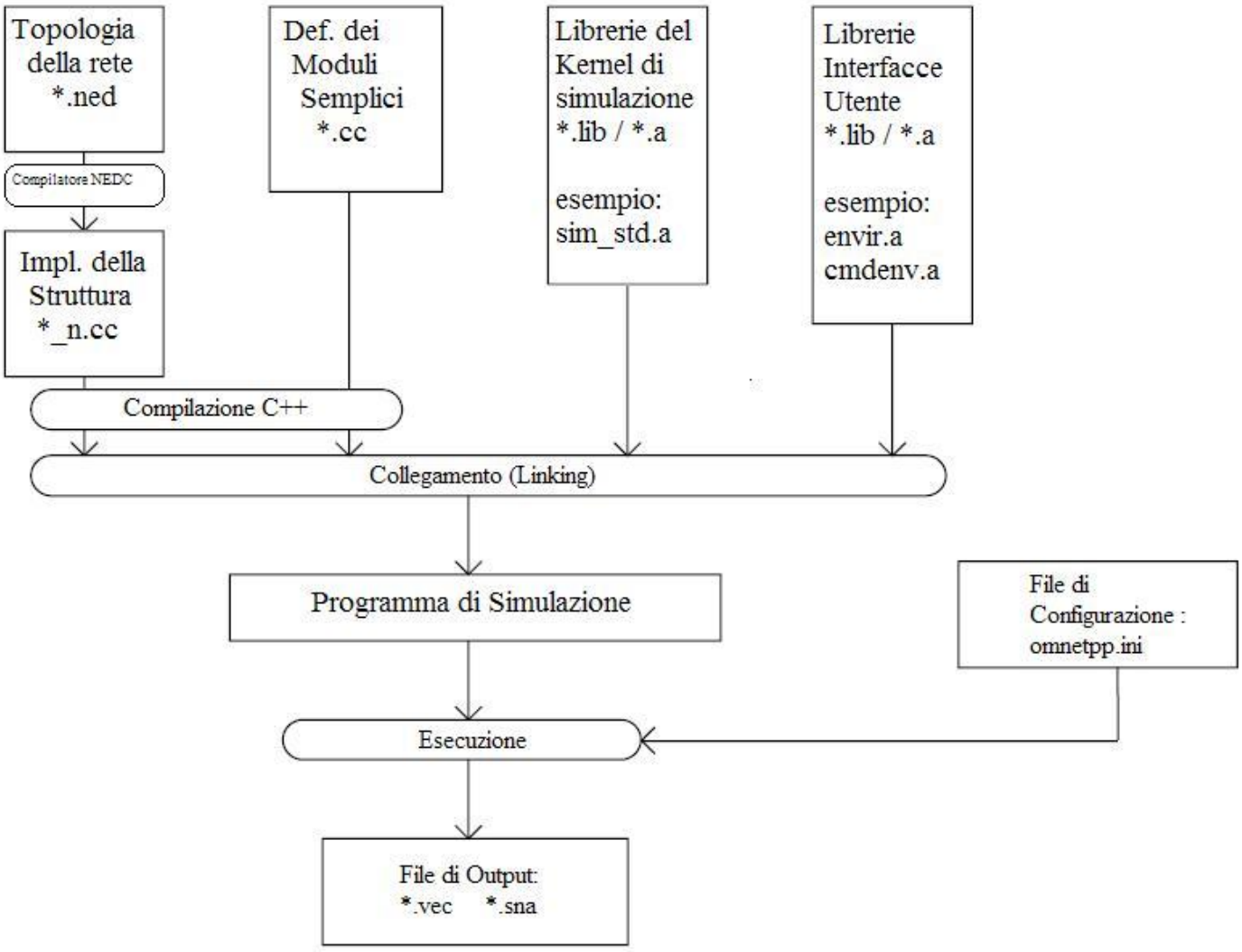
Il resto del procedimento combacia con quello di definizione di programmi che hanno come sorgenti file contenenti codice C/C++.
Tutti i sorgenti C++, infatti, devono essere compilati in file oggetto (con estensione .o in Unix/Linux, .obj in Windows) per poi essere legati (link) con le librerie di simulazione; definendo così l'eseguibile del programma.
I nomi dei file librerie differiscono tra gli ambienti Unix/Linux e quelli Windows, oltre che per le librerie statiche e quelle condivise.
Supponiamo di avere una libreria chiamata TKENV.
In un ambiente Unix/Linux, il nome della libreria sarà qualcosa come "*libtkenv.a*" (o lintkenv.a.[versione]) mentre per le librerie condivise si avrà "*libtkenv.so*" (o libtkenv.so.[versione]).
Le versioni Windows delle librerie statiche saranno, invece, "*tkenv.lib*", mentre le DLL (che rappresentano l'equivalente Windows delle librerie condivise) saranno *"tkenv.dll"*.
Tornando al processo di definizione del programma è necessario, in fase di linking, legare le seguenti librerie:

- le librerie del Kernel di simulazione e delle classi; chiamata *"sim_std"* (file libsim_std.a , sim_std.lib , ecc).

– Le interfacce utente. La parte comune di tutte le interfacce utente è contenuta nella libreria "*envir*" (file libenvir.a , etc), mentre le interfacce utente specifiche sono "*tkenv*" e "*cmdenv*" (libtkenv.a , libcmdenv.a , etc).

Fortunatamente questi non sono dettagli di cui ci dobbiamo preoccupare.
Esistono, infatti, strumenti automatici come *" opp_makemake"* che fanno il "lavoro sporco" per noi.

La seguente figura ha come obiettivo quello di illustrare il processo di definizione e di esecuzione del programma di simulazione.



UNIX e gcc

Il processo di installazione dipende soprattutto da quale distribuzione si sta utilizzando.
Inoltre, esso può variare da release a release.
La cosa migliore, quindi, è leggere il file “readme” contenuto nel pacchetto d'installazione.
Se si installa attraverso la compilazione dei sorgenti, ci si può aspettare le solite procedure GNU:
./configure seguiti immediatamente dei *make*.
Lo script “ opp_makemake” riesce a generare automaticamente i “makefile” per il programma di simulazione, basandosi sui file sorgenti contenuti nella directory corrente.
Inoltre, esso riesce a gestire correttamente modelli di grandi dimensioni, i cui sorgenti sono distribuiti in directory diverse.

Windows e Microsoft Visual C++

La procedura più semplice è quella che prevede l'utilizzo della versione di installazione basata su componenti binari, che consente di ottenere un sistema funzionante e molto veloce.

Essa contiene tutto il software necessario, fatta eccezione per lo MSVC.
Il passo successivo sarà quello di scaricare e gestire anche i sorgenti della distribuzione.
Tale passo consente di compilare le librerie con flag differenti, eseguirvi il debug, ricompilarle con l'ausilio di pacchetti addizionali (e.g. Akaroa, MPI).
La compilazione è un processo relativamente indolore (infatti necessita di un singolo comando nmake -f Makefile.vc), dopo di che si ottengono le varie directory componenti riepilogate nel file *configuser.vc* .
OMNeT dispone di un creatore di makefile MSVC automatico chiamato " *opp_nmakemake*", il quale è probabilmente lo strumento più semplice da utilizzare.
La sua funzione è molto simile a quella della sua controparte in Unix.
Se si esegue opp_nmakemake in una directory contenente alcuni sorgenti del modello, esso collega tutti i nomi di tutti i sorgenti ivi contenuti e crea un makefile chiamato *" Makefile.vc".*
Ritengo necessario spendere qualche parola su uno dei problemi più comuni e frequenti che si presentano in questa fase del processo di definizione dell'eseguibile della simulazione.

Può capitare che *nmake*, che parte è dello *MSVC*, non viene identificato perché non è nella directory in cui si sta operando. Una soluzione totalmente indolore è quella di eseguire il file *" vcvars32.bat" , il quale può essere trovato nella directory MSVC bin* (C:\Program Files\Microsoft Visual Studio\VC98\Bin).

Interfacce utente

L'obiettivo principale delle interfacce utente è quello di rendere la struttura interna del modello visibile all'operatore al fine di controllarne l'esecuzione ed, eventualmente, consentire all'utente di intervenire per modificare le variabili / oggetti all'interno del modello.

Questo è molto importante per le fasi si sviluppo / debug del progetto di simulazione.

Tali interfacce grafiche possono essere usate per definire presentazioni basate sulle operazioni compiute dal modello.

Lo stesso modello di simulazione può essere eseguito con differenti interfacce utente, senza la necessità di modificare i file di definizione del modello.

L'utente può voler utilizzare , nelle fasi di test e debug, una interfaccia grafica molto potente ed, infine, eseguire l'intera

simulazione con un’altra interfaccia, più semplice e veloce, che supporti l’esecuzione batch.

Le simulazioni OMNeT possono essere eseguite sotto interfacce utente differenti.

- *TKENV* : basata su Tcl/Tk (interfaccia utente a finestre)
- *CMDENV* : interfaccia utente a riga di comando per le esecuzioni batch

In genere le fasi di test e di debug vengono programmate sotto TKENV , per poi eseguire l'esperimento di simulazione attuale da riga di comando, utilizzando CMDENV.
Il primo è utile anche per simulazioni di presentazione e dimostrazione.
Entrambi vengono forniti sotto forma di libreria; quello scelto sarà quello che verrà legato (linking) all'eseguibile della nostra simulazione.
Sia TKENV che CMDENV sono supportati sulle piattaforme LINUX e WINDOWS.
Le funzionalità comuni alle due interfacce sono collegate e posizionate all'interno della libreria “Envir”, la quale può essere considerata come una classe base comune.
Le due interfacce sono separate dal kernel di simulazione; esse interagiscono tra di loro utilizzandone una terza.

Ciò implica anche che , se necessario, è possibile scriverne una propria, oppure incapsulare una simulazione OMNeT senza apporre nessuna modifica ai file ed alle librerie del modello.
I dati di configurazione e di input della simulazione sono descritti in un file che viene generalmente chiamato "omnetpp.ini".
In questo file, alcune voci si applicano solamente a TKENV, altre solamente a CMDENV, altre ancora hanno effetto indipendentemente dalla interfaccia attualmente in uso.

Capitolo 5

NS-2

NS è un simulatore orientato agli oggetti, scritto in C++, con un interprete OTcl come controparte.

Il simulatore supporta una gerarchia di classi C++ (che per praticità verrà indicata con il termine di “Gerarchia Compilata”), ed una simile all’interno dell’interprete OTcl (“Gerarchia Interpretata”).

Le due gerarchie sono strettamente relazionate tra di loro.

Dalla prospettiva dell’utente (programmatore), c’è una corrispondenza biunivoca (uno ad uno) tra una classe della gerarchia interpretata ed una della gerarchia compilata.

La radice di questa gerarchia è la classe “TlcObject” .

Gli utenti creano nuovi oggetti simulator attraverso gerarchia interpretata; questi oggetti vengono istanziati all'interno dell'interprete e vengono rispecchiati da altrettanti oggetti corrispondenti della gerarchia compilata.

La gerarchia di classi interpretata viene automaticamente stabilita attraverso dei metodi definiti all'interno della classe "TclClass".

Gli oggetti istanziati dagli utenti vengono rispecchiati attraverso metodi definiti all'interno della classe "TclObject".

In realtà ci sono altre gerarchie contenute all'interno del codice C++ e degli script OTcl.

Tali gerarchie però non sono rispecchiate, cioè non sono direttamente collegate tra di loro.

5.1 Panoramica concettuale

Perché due linguaggi?

NS utilizza due linguaggi differenti dato che il simulatore ha due generi di obiettivi e funzionalità differenti.

Da una parte, le simulazioni dettagliate di protocolli richiedono un linguaggio di sistema che consenta di manipolare efficientemente byte, intestazioni di pacchetti, e di implementare algoritmi da eseguire su un ampio numero di dati.

Per questi compiti la velocità di esecuzione ed elaborazione a run-time è fondamentale, mentre il tempo di turn-around (quantità di tempo impiegata per eseguire la simulazione, determinare eventuali bug, ricompilare, rieseguire, ecc…) è meno importante.

Dall'altra parte, una larga percentuale delle risorse di rete impiegano parametri o configurazioni leggermente diversi, o presentano frequentemente un gran numero di scenari differenti.

OTcl è tendenzialmente più lento in fase di esecuzione ma supporta modifiche più rapide (ed interattive), rendendolo ideale per le configurazioni delle simulazioni.

NS (tramite Tclcl) fornisce il collante che consente di utilizzare e gestire gli oggetti e le variabili in entrambi i linguaggi.

5.2 Panoramica sul codice

In questo trattato, il termine “Interprete” verrà utilizzato come sinonimo dell’interprete OTcl.
Il codice che consente di interfacciare i vari oggetti con l’interprete risiede in una directory separata, “*Tclcl*”.
Non è un caso, quindi, se tale directory contiene un buon numero di classi.
Di tutte queste, verranno trattare solamente quelle utilizzate in NS :

- Classe *Tcl*
- Classe *TclObject*
- Classe *TclCommand*
- Classe *EmbeddedTcl*
- Classe *InstVar*

Classe Tcl

Questa classe incapsula l’istanza attuale dell’interprete OTcl, e fornisce i metodi per accedere e comunicare con esso.
I metodi forniti adempiono ai seguenti compiti :

- Ottenere il riferimento ad una istanza Tcl
- Invocare le procedure OTcl attraverso l’interprete

- Ritrovare, o ritornare i risultati delle elaborazioni effettuate all'interprete
- Riportare eventuali errori delle simulazioni ed uscire in modo ottimale
- Memorizzare gli oggetti TclObject
- Acquisire l'accesso diretto all'interprete

Classe TclObject

Essa rappresenta la classe di base della maggior parte delle classi contenute nelle due gerarchie, quella interpretata e quella compilata.

Ogni oggetto appartenente a questa classe viene creato dall'utente dall'interno dell'interprete.

Un "oggetto ombra" equivalente viene creato automaticamente, come controparte di questo, nella gerarchia compilata.

I due oggetti sono strettamente relazionati l'uni con l'altro.

In sostanza, la classe TclObject contiene i meccanismi che eseguono questo automatismo, ovvero la creazione automatica di un oggetto ombra.

Nel resto di questo trattato capiterà spesso, quindi, che io faccia riferimento ad un oggetto come "oggetto TclObject".

In tali casi, mi riferirò ad un particolare oggetto che è contenuto all'interno della classe TclObject , oppure in una classe da essa derivata.
Se necessario, comunque, qualificherò esplicitamente un oggetto indicando se esso è all'interno dell'interprete oppure se è all'interno di un particolare segmento di codice compilato.
In entrambi i casi, utilizzerò la terminologia di "oggetto interpretato" e "oggetto compilato" per distinguere i due.

Classe TclClass

Si tratta di una classe compilata che ha la caratteristica di essere una classe puramente virtuale.
Le classi derivate da essa forniscono due funzioni:

- Costruiscono la gerarchia di classi interpretata per rispecchiare la gerarchia di classi compilata
- Forniscono metodi per istanziare nuovi oggetti TclObject

Ogn'una della classi derivate è associata ad una particolare classe compilata appartenente alla gerarchia delle classi compilate e può, quindi, istanziare nuovi oggetti nella classe ad essa associata.

Classe TclCommand

Questa classe ha l'unico compito di fornire i meccanismi che NS utilizza per esportare comandi semplici all'interprete.

A questo punto, essi possono essere eseguiti da quest'ultimo all'interno del contesto globale.

Classe EmbeddedTcl

NS consente lo sviluppo di nuove funzionalità, sia all'interno del codice compilato che di quello interpretato, che vengono valutate al momento dell'inizializzazione.

Esempi validi sono gli script "*.../tclcl/tcl-object.tcl*" e "*.../ns/tcl/lib*".

Sia l'operazione di caricamento che quella di valutazione degli script vengono eseguite attraverso l'utilizzo di oggetti appartenenti a questa classe.

Classe InstVar

Essa definisce i metodi ed i meccanismi che consentono di legare ad una variabile membro C++, all'interno di un oggetto compilato, una specifica variabile istanza OTcl contenuta nell'oggetto interpretato equivalente.

Questo legame è impostato in modo che il valore della variabile può essere modificato, o comunque vi si può accedere, sia dal codice compilato che da quello interpretato.

5.3 La classe Simulator

Il simulatore è descritto dalla classe "*Simulator*".
Essa fornisce un insieme di interfacce che consentono di configurare una simulazione e di scegliere la tipologia di scheduler di eventi utilizzata per guidarla.
Uno script di simulazione generalmente inizia con a creazione di una istanza di questa classe e con le chiamate a vari metodi, la cui funzione è quella di creare i nodi, le topologie e configurare altri aspetti della simulazione.
Degna di nota è la sottoclasse "*OldSim*"; essa garantisce la compatibilità con la versione precedente di NS: *NS v1*.

Inizializzazione del simulatore

Quando un nuovo oggetto *Simulator* viene creato in tcl, la procedura di inizializzazione esegue le seguenti operazioni:

- Inizializza il formato dei pacchetti
- Crea uno scheduler
- Crea un "*Null Agent*"

L'inizializzazione del formato dei pacchetti imposta gli offsets dei campi all'interno dei pacchetti, usati dall'intera simulazione.
Lo scheduler esegue la simulazione in una modalità "*event-driven*" e può essere rimpiazzato da uno scheduler alternativo il quale fornisce funzioni semanticamente differenti.
Il *null agent* è generalmente utile per i pacchetti scartati o come destinazione per quei pacchetti che non sono stati contati o memorizzati.

Scheduler ed Eventi

Come già accennato, il simulatore ha una gestione *event-driven*.
Attualmente sono 4 le versioni disponibili all'interno del simulatore, ogn'una delle quali è implementata utilizzando una struttura dati differente:

- Semplice lista concatenata
- Heap
- Coda *calendar* (la cui gestione somiglia molto a quella di un calendario annuale)
- *Real time*

L'esecuzione dello scheduler consiste nella scelta del prossimo evento da elaborare, nella sua elaborazione e nella selezione del suo vicino.
La sua unità di tempo sono i secondi.
Attualmente, il simulatore è a *flusso unico*, cioè elabora solamente un evento a ciclo.
Se capita che più eventi sono schedulati per essere eseguiti nello stesso istante, la loro elaborazione viene gestita con il metodo FIFO (il primo nella fila è il primo ad uscirne).
Gli eventi simultanei non vengono mai riordinati, da nessuna tipologia di scheduler.

Questi ultimi, infatti, devono produrre il medesimo output con il medesimo input (quindi con il medesimo ordinamento).

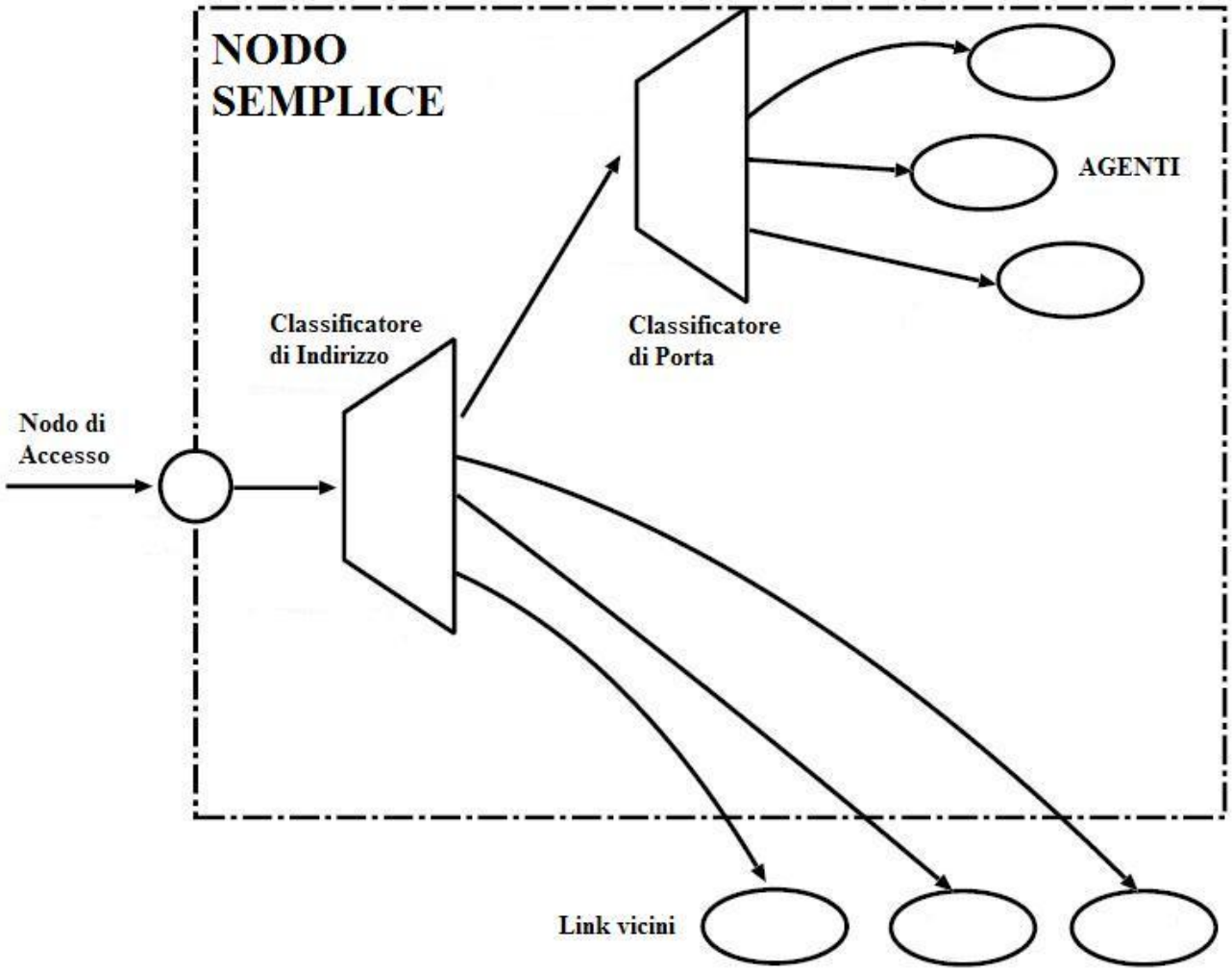
5.4 Trasmissione dei pacchetti

Ricordo che ogni simulazione richiede una singola istanza della classe *Simulator* per essere controllata ed eseguita.
La classe fornisce le procedure istanza che consentono di crearne e gestirne la topologia e memorizzarne internamente i riferimenti a tutti gli elementi in essa contenuti.

Nodi

La procedura istanza *node* permette di creare un nodo nella sua forma più semplice.
Esso stesso è una classe autonoma scritta in OTcl.
Comunque, la maggior parte dei suoi componenti sono oggetti *TclObject.*

La tipica struttura di un nodo è definita nella figura seguente .



Questa semplice struttura consta di due oggetti :

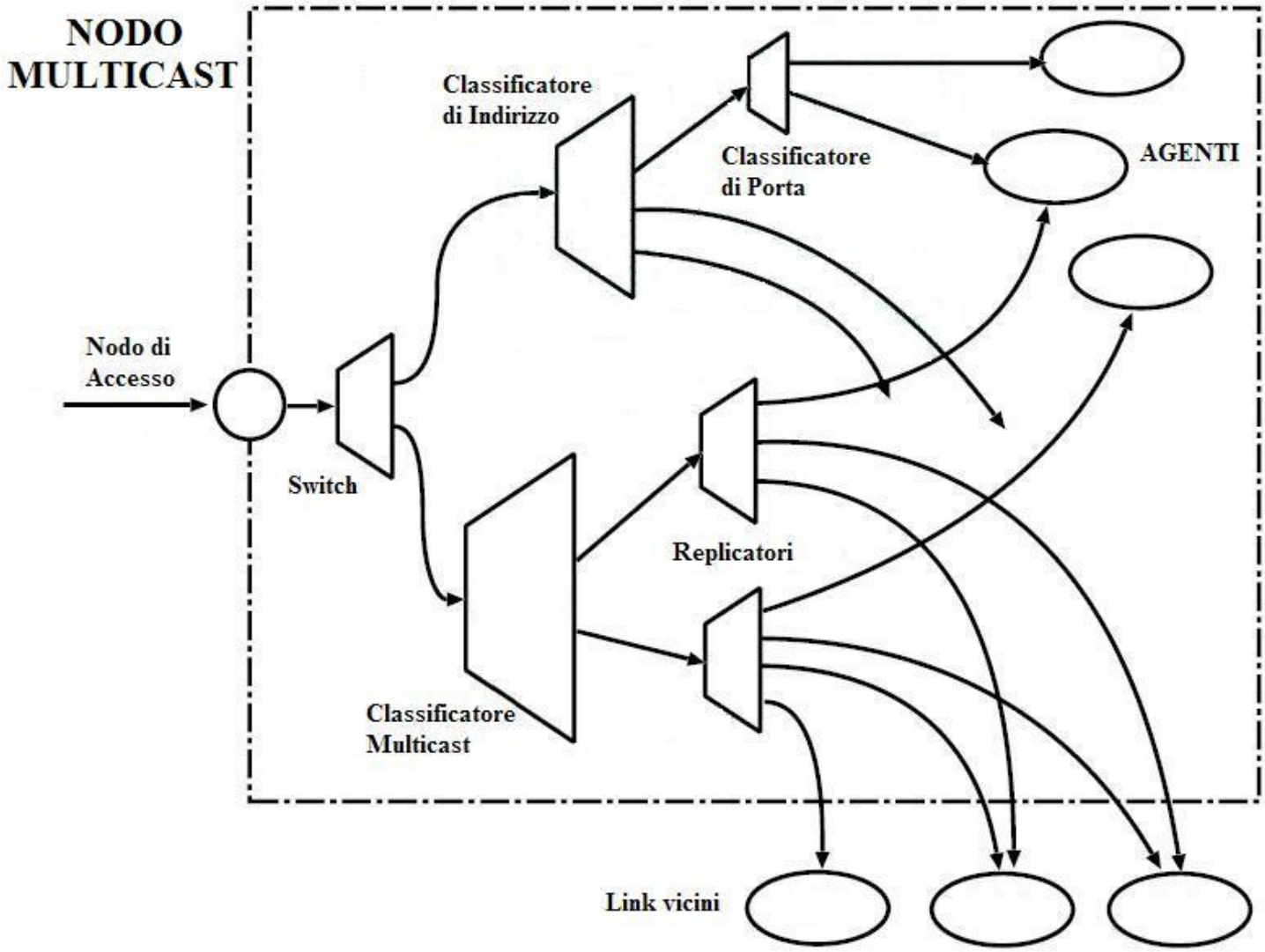
- Un classificatore di indirizzo
- Un classificatore di porta

La funzione di questi due classificatori è quella di distribuire i pacchetti in arrivo agli agenti o ai link corretti.
Tutti i nodi contengono alcuni dei seguenti componenti:

- Un indirizzo, incrementato di uno (valore iniziale 0) ad ogni creazione di un nuovo nodo
- Una lista di nodi confinanti
- Una lista di agenti
- Un identificatore di tipo
- Un modulo di istradamento (routing)

Per default, i nodi in NS sono definiti per simulazioni *uni casting* (a flusso singolo).
Per abilitare le simulazioni *multi casting*, la simulazione (e quindi lo scheduler) deve essere creata con una opzione particolare.

La struttura interna di un tipo nodo multi casting è la seguente



Quando una simulazione utilizza il routing multicast , il primo bit dell'indirizzo indica se un particolare indirizzo è un indirizzo uni casting, oppure multicast.
Se questo bit è 0, l'indirizzo è di tipo unicast, altrimenti essa rappresenta un indirizzo multicast.

Classificatori

La funzione di un nodo, quando riceve un pacchetto, è quella di esaminare i campi che compongono quest'ultimo.
Nella maggior parte dei casi i campi ad essere esaminati sono quelli che compongono l'indirizzo del pacchetto; capita , a volte, che ad essere controllati siano i campi della sorgente.
A questo punto esso dovrebbe mappare il valore ad un corretto oggetto interfaccia di uscita che è il prossimo "*contenitore*" che conterrà il pacchetto in questione.
In NS, questo compito viene eseguito da un semplice oggetto *classifier* (classificatore).
Oggetti classificatori multipli, ogn'uno dei quali esamina una particolare porzione dell'indirizzo del pacchetto che attraversa il nodo.
Un nodo , normalmente, utilizza più tipologie differenti di classificatori per ovviare a questa necessità.

Un classificatore fornisce, quindi, le funzioni di controllo e di matching tra un pacchetto ed alcuni criteri logici al fine di trovare , ed inoltrare, il pacchetto ad un altro *simulation object*.
Ogni classificatore contiene una tabella con tutti gli object della simulazione indicizzati per numero di slot.
Il lavoro di un classificatore è quello di determinare il numero di slot associato con un pacchetto ricevuto ed inoltrare quest'ultimo all'oggetto di riferimento attraverso un particolare slot.

Moduli di istradamento

Come già detto in precedenza, un nodo in NS è essenzialmente una collezione di classificatori.
Il nodo più semplice (*unicast*) contiene solamente un classificatore di indirizzo ed un classificatore di porta.
Quando le funzionalità del nodo vengono estese, un numero di classificatori viene aggiunto al nodo base.
Un esempio di nodo con maggiori funzionalità è quello mostrato nella figura precedente (*multicast*).
Nel momento in cui si aggiungono dei blocchi funzionali ad un nodo, ogn'uno con un proprio classificatore, diviene essenziale fornire una interfaccia uniforme che consente di organizzarli e

collegarli ai blocchi di elaborazione delle politiche di istradamento.

Il metodo migliore per gestire queste casistiche è attraverso l'utilizzo dell'*ereditarietà di classe*.

Se di ha la necessità di avere un nodo che supporti l'istradamento gerarchico si dovrà , semplicemente, derivare la classe base dei nodi di NS e sovrascrivere i metodi di setting del classificatore per inserirvi un classificatore gerarchico.

Questa metodologia è utile se i nuovi blocchi funzionali sono indipendenti e non possono essere mescolati arbitrariamente.

Un esempio valido di come non è sempre utile utilizzare l'ereditarietà è rappresentato dalle politiche di istradamento.

Entrambi i *routing*, gerarchico e ad hoc, utilizzano infatti il proprio insieme di classificatori.

Una struttura che supporta queste casistiche è, invece, quella della *composizione degli oggetti*.

Il nodo di base deve necessariamente definire una serie di interfacce che consentono la comunicazione e la gestione dei vati classificatori.

Queste interfacce dovrebbero :

- Supportare moduli di istradamento individuali che implementano i propri classificatori e che li inseriscono all'interno del nodo

- Consentire ai blocchi di elaborazione dell'istradamento di popolare i percorsi di tutti i moduli di routing che necessitano di queste informazioni
- Utilizzare un punto singolo di gestione dei moduli di routing esistenti.

Inoltre, si dovrebbe anche definire una interfaccia uniforme per i moduli di routing che consenta di collegare quest'ultimi alle interfacce dei nodi, al fine di fornire un approccio sistematico ad ulteriori estensioni delle funzionalità.

Link

Essi rappresentano l'altro aspetto (oltre ai nodi) della definizione della topologia di una rete.
Così come un nodo è composto da un insieme di classificatori, un link è definito da una sequenza di connettori.
La classe *Link* è un'altra classe autonoma scritta in OTcl che fornisce alcune semplici *primitive*.
Da questa deriva la classe *SimpleLink*, la quale implementa la possibilità di connettere due nodi con una connessione "*punto a punto*".

NS fornisce, inoltre, un'altra procedura istanza chiamata *simplex-link*() la cui finalità è quella di creare un link unidirezionale.

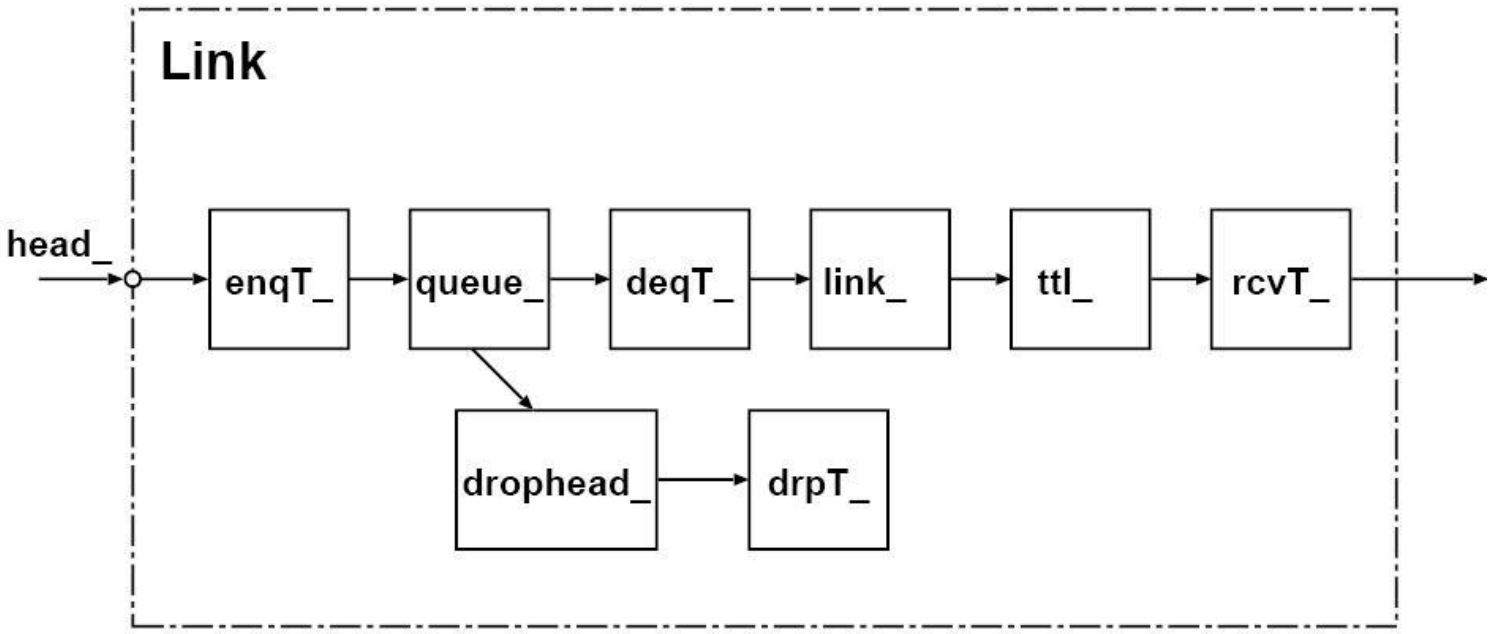
Il comando che normalmente viene utilizzato per definire un link prende come parametri non solo i due nodi che fungono da capi del collegamento, ma anche l'eventuale ampiezza di banda del canale, il ritardo di propagazione e la politica di gestione della coda.

La procedura consente di aggiungere altri elementi che descrivono le proprietà del link, come ad esempio il *TTL checker* (contatore del parametro *Time to Live* , un parametro utilizzato nelle procedure di controllo delle congestioni) oppure come quelli riportati di seguito:

- *Head_* : punto di entrata del link; è direttamente connesso con il primo oggetto che compone il link
- *Queue_* : riferimento all'elemento coda principale del link. Normalmente, i link semplici hanno una sola coda. Chiaramente più è complesso il link, più code può avere.
- *Link_* : riferimento all'elemento che attualmente modella il link e che ne detta alcune caratteristiche, come l'ampiezza di banda ed il ritardo di propagazione.
- *dropHead_* : riferimento ad un oggetto che rappresenta la testa della coda che contiene tutti gli elementi scartati dal link.

Oltre agli elementi sopra riportati è possibile , se definita una particolare variabile istanza, tracciare gli elementi che rilevano gli istanti in cui un pacchetto viene accodato oppure viene rilasciato.

Alla luce di questo, un link può essere considerato come una struttura simile a quella rappresentata nella seguente figura:



Ritardi di propagazione

I ritardi di propagazione rappresentano il tempo impiegato da un pacchetto ad attraversare un link.
Una particolare tipologia di link (link dinamici) consentono, inoltre, di gestire le varie casistiche di interruzioni casuali dei link.
L'ammontare del tempo impiegato da un pacchetto per attraversare un link è definito dalla seguente formula : $S/B + D$ dove

- S rappresenta la dimensione del pacchetto (come registrato nel suo campo *header*, cioè intestazione)
- B indica la velocità del link calcolata come bits/sec
- D è il ritardo in termini di secondi.

La classe *LinkDelay* deriva direttamente dalla classe *Connector*.
Un oggetto di tale classe è caratterizzato, tra le altre cose, dalla variabile istanza *dinamic_*.
Essa determina il comportamento del link (dinamico o non).

Agenti

Gli agenti rappresentano i punti finali del ciclo di vita dei pacchetti; ogni pacchetto viene creato da un agente e poi spedito, per terminare il suo percorso in un altro agente.
La classe *Agent*, non a caso, ha una implementazione che è parzialmente scritta in OTcl e parzialmente in C++.
Essa contiene un buon numero di stati interni, necessari per l'assegnazione di campi differenti ai pacchetti in partenza o in arrivo.
Queste variabili possono essere impostate, o modificate, da una qualsiasi classe derivata da *Agent*; sebbene sia molto difficile che un agente si interessi di un numero così grande di parametri.
Tale classe supporta la generazione e la ricezione dei pacchetti.
Gli agenti, inoltre, sono utilizzati anche come supporto all'implementazione di protocolli di rete appartenenti a livelli differenti.
Da ciò se ne deduce che, per alcuni protocolli di trasporto (come ad esempio *UDP*), la distribuzione della dimensione dei pacchetti e/o i tempi di arrivo/partenza possono essere dettati da alcuni oggetti separati che rappresentano le necessità dell'applicazione.

A tal fine, gli agenti dispongono di una interfaccia di programmazione dell'applicazione (*API*).
Per quelli di essi impiegati nell'implementazione di protocolli di basso livello (ad esempio agenti di routing), la dimensione e la temporizzazione delle partenza vengono generalmente dettate da un agente di elaborazione proprio.
Gli agenti possono essere definiti attraverso l'OTcl ed i loro stati interni modificati tramite l'utilizzo di un insieme ristretto di funzioni Tcl.
Degna di nota è la caratteristica che alcuni degli stati interni degli agenti possono esistere solamente all'interno dell'OTcl, ragion per cui non tutti sono direttamente accessibili da C++.

Intestazioni

Gli oggetti definiti nella classe *Packet* sono le unità fondamentali dello scambio di informazioni tra gli oggetti che compongono una simulazione.

Tale classe fornisce, ad esempio, le informazioni che consentono di collegare un pacchetto ad una lista, di riferirsi ad un buffer contenente le intestazioni dei pacchetti che sono definite su un punto protocollare di base o contenente dei pacchetti dati semplici.

I nuovi protocolli possono definire autonomamente le interstazioni dei propri pacchetti, oppure possono estendere quelle esistenti con campi addizionali.

Le nuove intestazioni sono, quindi, introdotte all'interno del simulatore attraverso:

- la definizione di una nuova struttura C++ contenente i campi necessari
- la definizione di una classe statica, la cui funzione sarà quella di fornire il collegamento OTcl
- la modifica di alcuni segmenti del codice di inizializzazione del simulatore, al fine di assegnare un *offset* specifico ad ogni pacchetto che utilizzerà l'intestazione in questione.

Quando il simulatore viene inizializzato attraverso il codice OTcl, l'utente può decidere di abilitare solamente un sottoinsieme dei formati di pacchetti precompilati.
Tale operazione si tradurrà in un risparmio della memoria necessaria per l'esecuzione della simulazione.
Normalmente, comunque, la maggior parte di tali formati viene lasciata abilitata per non ridurre le funzionalità ed incappare in errori non previsti.
La gestione di quali formati di pacchetti sono correntemente abilitati in una simulazione viene delegata ad uno speciale oggetto gestore delle interstazioni.
Da notare che se un oggetto, all'interno della simulazione, utilizza un campo di una intestazione che è stata malauguratamente disattivata, verrà prodotto un errore fatale che terminerà istantaneamente l'esecuzione.

Le classi *packet*

Ci sono 4 classi C++ rilevanti se si parla di gestione dei pacchetti o delle intestazioni di quest'ultimi:

- la classe *Packet* : essa definisce la tipologia per tutti i pacchetti utilizzati all'interno della simulazione. Essa è una sottoclasse delle classe *Event* ed è proprio questa caratteristica a consentire la schedulazione dei pacchetti.
- La classe *Packet_info* : contiene tutte le rappresentazioni testuali dei nomi dei pacchetti.
- La classe *PacketHeader* : fornisce una classe di base per tutte le intestazioni dei pacchetti configurate all'interno della simulazione.

  Essa, sostanzialmente, fornisce un buon numero di stati interni che consentono di localizzare ogni particolare intestazione contenuta in una collezione di intestazioni, all'interno di un particolare pacchetto.
- La classe *PacketHeaderManager* : definisce una classe utilizzata per collezionare e gestire le intestazioni attualmente configurate. Essa, infatti, viene invocata da un metodo scritto in OTcl, nella fase di configurazione della simulazione, al fine di abilitare alcuni offset delle intestazioni dei pacchetti precompilati.

5.5 Tracing e Monitoraggio

Ci sono un buon numero di modalità che consentono di raccogliere dati traccia o output sulla simulazione.
Generalmente i dati traccia vengono visualizzati direttamente durante l'esecuzione della simulazione, oppure vengono (più comunemente) memorizzati in un file che verrà analizzato ed elaborato ad esecuzione finita.
Due sono la capacità primarie, ma distinte, di monitoraggio che sono attualmente supportate dal simulatore.
La prima, chiamata *Traces*, registra gli arrivi, le partenze e gli scarti che ogni singolo pacchetto effettua all'interno di vari link e code. Gli oggetti traccia sono configurati all'interno della simulazione come nodi della topologia della rete.
Ad essi viene agganciato un oggetto *Channel* Tcl che rappresenta la destinazione dei dati raccolti.
La seconda utilizza una particolare tipologia di oggetti chiamati *Monitors*.
Essi registrano contatori di vari valori interessanti, come ad esempio gli arrivi/partenze di byte/pacchetti; inoltre possono monitorare contatori associati a tutti i pacchetti oppure solamente quelli appartenenti ad un determinato flusso.
Per supportare la prima tipologia (Traces), viene inserita in ogni pacchetto una speciale intestazione comune.
Essa, attualmente, include:

- Un identificatore univoco per ogni pacchetto
- Un campo che ne indica la tipologia (impostato dall'agente nel momento in cui il pacchetto viene generato)
- Un campo che ne indica la dimensione (in byte, utilizzato per determinare il tempo di trasmissione per i pacchetti)
- Una etichetta interfaccia (utilizzata per l'elaborazione di albero di distribuzione per flussi multipli).

I Monitor vengono supportati tramite un insieme separato di oggetti creati ed inseriti all'interno della topologia della rete; posizionati intorno alle code.
Essi rappresentano il punto in cui le statistiche sugli arrivi e le partenze vengono riunite ed elaborate tramite l'utilizzo della classe *Integrator*.

5.6 Librerie dinamiche

A partire dalla release NS-2.33, NS-2 supporta librerie caricabili dinamicamente.

Molti ricercatori, in tutto il mondo, procedono con lo sviluppo di versioni modificate e personalizzate di NS-2, al fine di implementare nuove caratteristiche e funzionalità (es. agenti, protocolli, algoritmi, ecc…).

La pratica standard adottata per tale sviluppo è la seguente

- Download di una versione ufficiale della distribuzione dei sorgenti di NS-2
- Implementazione delle modifiche necessarie al codice sorgente
- Aggiunta di nuovi file in un qualsiasi nodo/foglia dell'albero che identifica la gerarchia del codice
- Traduzione in un eseguibile NS-2 .

L'introduzione delle librerie caricabili dinamicamente fornisce una nuova modalità per l'estensione di NS-2.

Tale modalità presenta le seguenti caratteristiche:

- Gli utenti possono sviluppare componenti aggiuntivi per NS-2 (es. introducendo nuovi agenti, tipologie di pacchetti, protocolli) senza dover modificare il cuore del simulatore.

- Nuove intestazioni e tipologie di pacchetti, così come i rispettivi *Tracers*, possono essere definiti per assistere le operazioni di debuggin, raccolta di dati statistici o comunicazioni all'interno dei moduli. Questi ultimi possono anche essere caricati in base alle richieste ed alle necessità dei singoli utenti.
- Le librerie dinamiche possono essere caricate durante la fase di esecuzione della simulazione, senza avere la necessità di compilare l'intera distribuzione NS-2 o di scaricare file binari differenti.
- L'istallazione di terze parti che estendono le funzionalità di NS-2 risulta molto più semplice. Questo ne facilità implicitamente la diffusione.
- Tali librerie rendono la vita molto più semplici ai tecnici di laboratorio ed agli studenti. Infatti, una versione ufficiale NS-2 può essere istallata da un amministratore e gli studenti possono solamente programmarla ed impostarvi una priorità indipendente sulle estensioni.
- All'interno, queste modifiche accrescono la modularità e la scalabilità di NS-2. L'aggiunta di nuove caratteristiche al simulatore è più semplice e la compatibilità con versioni precedenti è garantita.

Dalla prospettiva dell'utente, l'unica operazione da compiere per utilizzare queste librerie e, quindi, beneficiare del loro contenuto è semplicemente quella di caricarle.

Il caricamento di una libreria dinamica dovrebbe essere eseguito all'inizio di uno script tcl, utilizzato per la simulazione.

Degno di nota è il fatto che il formato delle librerie varia in base alla tipologia di sistema operativo utilizzato.

Librerie " *.so* " si trovano nei sistemi UNIX, mentre per istanze di sistemi Windows sotto *Cygwin* si utilizzeranno librerie " *.dll* "; infine troveremo librerie " *.dylib* " sotto sistemi OS X.

A variare non sono sole le estensioni.

Anche il *LongName* dei file contenenti le librerie può cambiare.

5.7 NAM

Nam è uno strumento di animazione basato sulle librerie Tcl/TK che consente di visualizzare i dati traccia delle simulazioni.

Il disegno di base di questo strumento era quello di creare un "animatore" capace di comprendere ed interpretare vaste quantità e tipologie di dati e di renderlo abbastanza flessibile da consentirne l'utilizzo in varie simulazioni.

Stabiliti questi obiettivi, Nam è stato progettato per leggere semplici comandi di gestione degli eventi di una animazione riepilogati in una quantità più o meno grande di tracce.

Per gestire al meglio le simulazioni più ampie è stato poi ridotto al minimo il consumo di memoria.

I comandi di gestione degli eventi vengono , quindi, memorizzati in un file e riletti ogni qual volta se ne presenta la necessità.

Il primo passo per utilizzare Nam è , chiaramente, la definizione dei file di traccia.

Questi file possono contenere informazioni sulla topologia della rete (es. nodi, link), così come tracce di rilevazione dei pacchetti.

Normalmente, le tracce vengono generate da NS-2.

Durante una simulazione, infatti, l'utente può prevedere la traccia delle configurazioni della topologia, delle informazioni

sul layout o delle movimentazioni dei pacchetti utilizzando gli eventi traccianti, messi a disposizione da NS-2.

Quando un file di traccia viene generato, esso è pronto per essere interpretato ed animato da Nam.

Al momento dell'avvio dell'animazione Nam leggerà il file di traccia, creerà la topologia, genererà una finestra di visualizzazione, eseguirà l'interfaccia (se necessario) ed , infine, si metterà in pausa all'istante 0.

Attraverso la sua interfaccia utente, esso fornisce un buon numero di controlli che consentono di gestire la visualizzazione dell'animazione appena caricata in maniera ottimale.

Interfaccia Utente

In fase di attivazione di Nam, la prima operazione ad essere effettuata è la creazione della finestra "*console*".
È importante considerare che si possono avere esecuzioni di animazioni multiple sotto la stessa istanza Nam.
Quest'ultime possono essere salvate e convertite in "*gif*" animate , oppure convertite in filmati *mpeg*.
Per memorizzare i fotogrammi dei filmati è necessario avviare Nam con i file traccia adatti ed impostare alcuni parametri specifici (densità dei fotogrammi, dimensione, ecc . . .).
In Nam, la topologia viene definita tramite degli oggetti nodo alternativi.
Inoltre, per visualizzarla in un modo comprensibile (senza avere i nodi sparsi in modo totalmente casuale) è necessario adottare un "*meccanismo di posizionamento*".
I meccanismi attualmente forniti da Nam sono 3.
Nel primo, l'utente può specificare il la posizione tramite l'orientamento dei link, inteso come l'angolo tra il bordo del nodo ed una linea orizzontale.
Sarà poi Nam, durante la fase di posizionamento, ad interpretare le direttive fornite dall'utente attraverso i seguenti passi:

- Per prima cosa viene scelto un nodo di riferimento

– A questo punto gli altri nodi vengono posizionati utilizzando l'orientamento e la lunghezza dei link che li collegano.

Questo meccanismo è utile ed efficiente solamente se si deve generare manualmente una topologia di piccole dimensioni.
Nel secondo, quando è necessario gestire delle topologie generare casualmente, si può avere la necessità di un meccanismo automatico.
A tal fine è stato adattato ed implementato un algoritmo di posizionamento grafico particolare.
L'idea di base dell'algoritmo è quella di modellare il grafico come dei cerchi (nodi) collegati tramite delle linee (link).
I cerchi si respingeranno a vicenda, mentre le linee eserciteranno l'effetto contrario.
In pratica, dopo un limitato numero di cicli (decine o centinaia), il grafico finisce con il convergere e genera una struttura visuale comprensibile.
Per i grafici di grandi dimensioni è possibile, comunque, utilizzare una combinazione di questi due meccanismi di posizionamento per raggiungere un layout accettabile.
Nel terzo, ed ultimo, meccanismo viene utilizzato un sistema di coordinate bidimensionali.

Esso è stato studiato per essere utilizzato nella visualizzazione di topologie wireless, o comunque particolarmente dinamiche, nelle quali non esistono link permanenti.
Utilizzando questo meccanismo, ai nodi vengono assegnate due coordinate che indicano la posizione precisa all'interno del piano cartesiano.

Oggetti di animazione

Nam esegue le animazioni avvalendosi dei blocchi dei seguenti blocchi di controllo :

- Nodo : i nodi vengono creati da un numero di eventi "n" contenuti all'interno dei file di traccia.
  Essi possono rappresentare delle sorgenti, degli host, dei router, ecc.
  Nam tralascerà , chiaramente , ogni definizione duplicata dello stesso controllo.
  Un nodo può assumere tre aspetti (cerchio, quadrato o esagono); questi vengono impostati al momento della creazione del nodo e non possono essere più modificati.

Ad ogni nodo è possibile assegnare un colore identificativo; inoltre, può essere prevista una etichetta di riconoscimento.

- Link : i link vengono creati tra due nodi e sono il secondo componente della topologia di una qualsiasi rete.

  Visto dall'interno, ogni link Nam consta di due link semplici (monodirezionali incrociati).

  Gli eventi traccia "l" sono quelli che si occupano di definire questi due link semplici e di compiere tutte le altre operazioni di setting necessarie.

  Pertanto, dal punto di vista dell'utente (programmatore) tutti i link creati da Nam possono essere considerati dei link biunivoci.

  Come per i nodi, anche ai link può essere assegnato un colore particolare, oppure può essere agganciata una etichetta descrittiva.

  La differenza principale sta, però , nel fatto che queste due impostazioni possono essere liberamente modificate durante l'animazione.

- Code : esse devono necessariamente essere definite tra due nodi.

Una coda Nam è agganciata a solamente un arco (link) dei due presenti tra ogni nodo.

Esse possono essere benissimo considerate come dei "pacchetti accatastati".

Questi ultimi dislocati lungo una linea retta.

Da questo punto di vista, l'angolo venutosi a creare tra la linea retta ed una linea orizzontale può essere specificato all'interno dello evento "trace queue".

– Pacchetti : i pacchetti sono visualizzati come dei blocchi, il cui orientamento viene notificato da una freccia.

  È, infatti, la direzione della freccia ad indicare il verso del flusso di trasmissione del flusso.

  Contrariamente a quelli in movimenti, i pacchetti accodati vengono visualizzati come dei piccoli quadrati.

  Ricordando che un pacchetto può essere scartato da una coda o da un link, questa particolare casistica viene visualizzata come una serie di quadrati rotolanti che cadono verso il bordo inferiore della finestra di visualizzazione, per poi sparire nelle sue immediate vicinanze.

  Sfortunatamente, non possiamo "deliziarci" di questa particolare animazione se il senso di elaborazione dello scheduler è posto al contrario sulla linea del tempo.

Questo piccolissimo bug è più una caratteristica delle strutture utilizzate, che una loro mancanza.

– Agenti : essi hanno il preciso scopo di separare i protocolli dai nodi.

  Ogni agente è sempre associato ad un particolare nodo ed è dotato di un nome che lo identifica univocamente.

  Esso viene visualizzato come un quadrato etichettato dal proprio nome e appare nelle immediate vicinanze del nodo cui fa riferimento.

Bibliografia

OMNeT++ Community Site
http://www.omnetpp.org/

K. Entacher, B. Hechenleitner, and S. Wegenkittl.
A Simple OMNeT++ Queuing Experiment Using Parallel Streams.
Editori: R. Trobec, P. Zinterhof, M. Vajtersic and A. Uhl.

Gábor Lencse. Graphical Network Editor for OMNeT++.
Tesi Master, Technical University of Budapest.

André Maurits, George van Montfort, and Gerard van de Weerd.
OMNeT++ Extensions and Examples. Technical report, Technical University of Budapest, Dept. of Telecommunications.

Y. Ahmet ¸ Sekercio˘glu, András Varga, and Gregory K. Egan.
Parallel Simulation Made Easy with OMNeT++.
In Proceedings of the European Simulation Symposium (ESS 2003), 26-29 Oct, 2003, Delft, The Netherlands. International Society for Computer Simulation,
2003.

András Varga.
OMNeT++ - Portable Simulation Environment in C++.
In Proceedings of the Annual Students' Scientific Conference (TDK), 1992.
Technical University of Budapest.

András Varga.
Portable User Interface for the OMNeT++ Simulation System.
Tesi Master, Technical University of Budapest, 1994.

András Varga.
Using the OMNeT++ Discrete Event Simulation System in Education.
IEEE Transactions on Education, November 1999.

NS-2 sites
http://www.isi.edu/nsnam/ns/
http://nsnam.isi.edu/nsnam/index.php/User_Information

Brent Welch.
Practical Programming in Tcl and Tk.
Prentice-Hall, 1995.

S. McCanne and S. Floyd. ns—Network Simulator. http://www-mash.cs.berkeley.edu/ns/.

Xiaoliang (David)Wei.
Amini-tutorial for TCP-Linux in NS-2. http://netlab.caltech.edu/projects/ns2tcplinux/.

Xiaoliang (David)Wei and Pei Cao.
NS-2 TCP-Linux: an NS-2 TCP implementationwith congestion control algorithms from Linux.

In *WNS2 '06: Proceeding from the 2006 workshop on ns-2: the IP network simulator*, page 9, New York, NY, USA, 2006. ACM Press.