KIT DI SOPRAVVIVENZA PER OMNET++ (su Windows)

Indice

INTRODUZIONE

Premetto che il contenuto di questo"kit" è tratto dal manuale ufficiale di Omnet++ versione 4.2.1 e non può assolutamente essere ritenuto sostitutivo al suddetto manuale, bensì un aiuto per coloro che si affacciano a Omnet++ e desiderano conoscerne gli strumenti base.

Omnet++ è un ambiente object-oriented per simulazioni ad eventi discreti quali possono essere:

- Reti di comunicazione wired e/o wireless;
- Modellazione di protocolli;
- Modellazione di reti di code;
- Modellazione di sistemi multiprocessore e altri sitemi distribuiti
- Validazione di architetture hardware
- Valutazione delle performance di sistemi software complessi
- In generale, modellazione e simulazione di qualunque sistema in cui sia applicabile l'approccio a eventi discreti e che può essere mappato in entità che comunicano attraverso lo scambio di messaggi.

INSTALLAZIONE

L'installazione è la base:

- Effettuare il download http://www.omnetpp.org/omnetpp/cat_view/17-downloads/1-omnet-releases (è possibile scaricare anche versioni meno recenti).
- Scompattare l'archivio su una directory a piacere (io ho usato direttamente C:\) e alla fine ottengo una cartella del tipo C:\omnetpp-xxx dove al posto delle "x" c'è la versione che avete scaricato.
- Spostarsi all'interno della cartella appena ottenuta e lanciare il programma mingwenv.cmd che trovate all'interno il quale aprirà una shell (è un emulatore di sh linux).
- Prima di procedere disattivare completamente qualsiasi antivirus perché potrebbe bloccare la corretta esecuzione del programma.
- Digitare all'interno della shell i comandi
    - ./configure
    - make
- Le due operazioni precedenti potrebbero impiegare parecchio tempo; una volta terminate sarà possibile riabilitare l'antivirus, si consiglia di aggiungere alla variabile d'ambiente PATH il percorso della cartella *bin* di omnet++ (per me ad esempio è *C:\omnetpp-4.2.1\bin*) in questo modo potrete lanciare omnet++ da prompt in qualunque percorso vi troviate. In alternativa potreste creare un collegamento al file *omnetpp.cmd* (che si trova sempre all'interno di *bin*) e metterlo sul desktop o dove preferite.
- Per lanciare, finalmente, l'ambiente dovrete semplicemente fare doppio click sul collegamento oppure eseguire il comando *omnetpp* da prompt dei comandi (questa seconda scelta è valida solo se è stata aggiunta al PATH la directory *bin* di omnet++).

Riferimento: http://omnetpp.org/doc/omnetpp/InstallGuide.pdf

Se durante la compilazione di qualunque project si verificasse il seguente errore

```
g++ CreateProcess: No such file or directory
make[1]: *** [../out/gcc-debug/src/libinet.dll] Error 1
make: *** [all] Error 2
```

(in Omnet++ 4.1) si verifica perchè il linker riesce a linkare un certo numero di files insieme ed, in questo caso, il limite è stato raggiunto.

Soluzione: escludere qualche percorso dalla compilazione attraverso l'opzione "Esclude from build" e lasciando senza spunta le voci *gcc_debug* e *gcc_release* (in omnet 4.2 non c'è questa possibilità).

ARCHITETTURA

Omnet++ fornisce l'infrastruttura e tools per realizzare simulatori. Uno degli ingredienti fondamentali di questa infrastruttura è *l'architettura a componenti* per i *modelli di simulazione*.

I modelli di simulazione vengono realizzati tramite l'impiego di componenti riusabili chiamati *moduli*. Questi moduli possono essere combinati come dei blocchi LEGO.

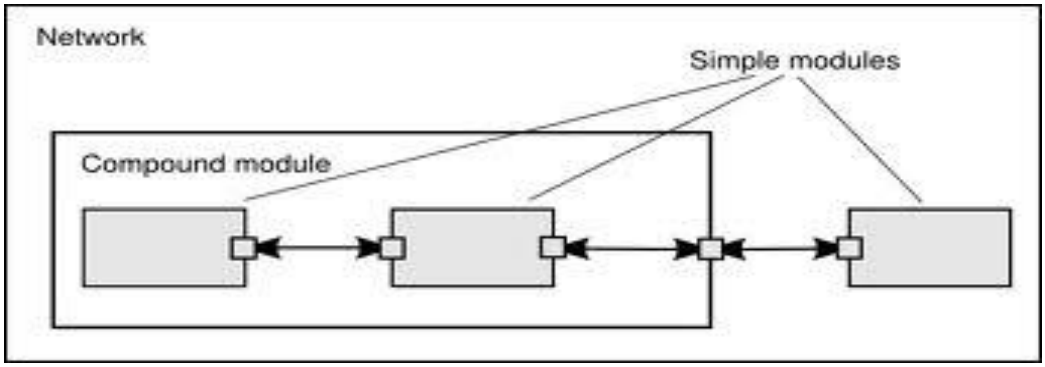
I moduli possono essere connessi tra di loro attraverso i *gates* e combinati insieme per formare dei *moduli composti (compound modules)*. Non c'è un limite sulla profondità di nesting dei moduli.

La comunicazione tra moduli normalmente avviene tramite message passing e i messaggi possono contenere delle strutture dati arbitrarie (a parte informazioni predefinite tipo i timestamp). Questi messaggi possono viaggiare attraverso percorsi predefiniti di *gates* e *connections* oppure essere inviati direttamente alla loro destinazione (quest'ultima scelta è molto utile nel caso dell comunicazioni wireless). Inoltre un modulo può essere corredato di *parameters* per personalizzarne il comportamento e per renderlo parametrico.

Omnet++ permette di eseguire simulazioni attraverso interfaccia grafica (per dimostrazioni e debug) oppure attraverso interfaccia command-line (migliore per esecuzione batch). I simple modules possono essere raggruppati in *compound modules*

ELEMENTI PRINCIPALI

Un *modello di simulazione (simulation model)* omnet++ consiste in moduli che comunicano attraverso message passing. I moduli che racchiudono la vera e propria logica si chiamano *simple modules*, essi sono scritti in C++ attraverso delle librerie di supporto messe a disposizione da omnet++.I simple modules possono essere raggruppati in *compound modules* senza alcun limite in livelli di gerarchia. L'intero modello, chiamato *network*, è esso stesso un *compound module*. Nella figura sotto, i rettangoli grigi sono i simple modules mentre quelli bianchi sono compound modules, si può anche notare che vi sono dei quadratini che rappresentano i *gates* (d'ingresso e/o d'uscita) e le frecce che sono le *connections*



Come detto in precedenza i moduli comunicano attraverso messaggi, i quali possono attraversare vari gates oppure essere inviati direttamente al modulo destinazione. I gates quindi sono le interfacce di input e di output dei moduli: i messaggi sono spediti attraverso un output gates e arrivano agli input gates. Un input gate e un output gate possono essere collegati da una *connection (o link)*; ogni connection è creata all'interno di un singolo livello della gerarchia di un modulo: all'interno di un compound module si possono connettere i gates di 2 simple module o un gate di un simple module e uno del compound module stesso.

Connection che interessano diversi livelli di gerarchia non sono possibili perché ostacolerebbero la riutilizzabilità. A causa della natura gerarchica del modello, i messaggi tipicamente viaggiano attraverso una catena di connections partendo da e arrivando a simple modules. Alle connections possono essere assegnati dei parametri come:

- Propagation delay;
- Data rate;
- Bit error rate.

È anche possibile definire delle connections con proprietà specifiche.

I moduli possono avere dei *parameters*, usati principalmente per passare dati di configurazione ai simple modules e per aiutare a definire la topologia del modello di simulazione. I parameters possono essere stringhe, valori numerici o booleani. Oltre all'essere usati come costanti, i parameters possono essere anche sorgenti di numeri random secondo una data distribuzione.

La libreria C++ di omnet++ mette a disposizione delle classi per modellare:

- Moduli, gates, connections, parameters;
- Messagges, packets (i packets sono un'estensione dei messages usati per le reti di comunicazione);
- Containers (array, code);
- Data collection classes;
- Statistiche e distribuzioni (istogrammi, etc…);

USING Omnet++

Per realizzare un simulatore omnet++ sono necessarie tre tipi di file:

- NED files (.ned) che servono per la descrizione della network da simulare (insieme di moduli e di collegamenti tra loro) e per la descrizione di ciascun simple module in termini di gates e parameters;
- Message definitions (.msg). Si possono definire dei messaggi (che ereditano da message o da packet) descrivendone gli attributi, un tool si occuperà di tradurre la descrizione nella corrispondente classe C++;
- Sorgenti dei simple module: sono file C++ (.h e .cc) che descrivono il comportamento di ciascun simple module.

Il sistema di simulazione fornisce i seguenti componenti:

- Simulation kernel: contiene il codice che gestisce la simulazione e la sua class library;
- User interfaces: usate durante l'esecuzione della simulazione per facilitare debugging, dimostrazioni ed esecuzioni batch.

Per prima cosa i file .msg vengono tradotti in classi C++ attraverso il tool `opp_msgc`; fatto ciò tutti i sorgenti C++ vengono compilati e linkati dal simulation kernel e una user interface library per costruire un eseguibile o una libreria condivisa. I file NED sono caricati dinamicamente quando il programma di simulazione viene eseguito.

Quando il programma di simulazione viene lanciato, esso legge prima i relativi NED files contenenti la topologia del nostro simulation model, poi legge un file di configurazione (*omnetpp.ini*) che contiene dei settaggi che controllano come è eseguita la simulazione, valori per i parameters, etc. Il file di configurazione può anche prevedere più run di simulazione; nel caso semplice, essi verrano eseguiti dal programma di simulazione uno dopo l'altro.

THE NED LANGUAGE

Il linguaggio NED (Network Description) è usato per realizzare una descrizione modulare di una network, di un simple module o compound module.Una file NED può contenere I seguenti componenti in quantità arbitraria:

- Direttive di import (meccanismo Java-like)
- Definizione di channels (usati per realizzare le connection)
- Definizioni di simple and compound module
- Definizione di interfacce (sono descrizioni astratte di moduli, quelli concreti "estenderanno" l'interfaccia)
- Definizioni di network

Per quanto riguarda gli identificatori (usati per dare un nome a qualsiasi componente) sono ammesse lettere (a-z,A-Z), numeri (0-9) e il carattere underscore; un identificatore non può iniziare con un numero,

inoltre è buona norma per nomi composti da più parole utilizzare la lettera maiuscola per l'inizio di ogni parola (EsempioDiIdentificatore). È convenzione anche iniziare con lettera maiuscola i nomi di modules, channels e networks, invece con lettera minuscola i nomi di parameters, gates e sottomoduli.

Il linguaggio è case sensitive. I commenti iniziano con il doppio slash '//' e terminano alla fine della riga; essi sono ignorati dal compilatore NED.

SIMPLE MODULE

Vediamo un esempio semplicissimo di simple module:

```
simple MyFirstModule
{
      parameters:
            int myparameter;
      gates:
            input in;
            output out;
            inout inputoutput;
}
```

Abbiamo descritto un modulo (MyFirstModule) il quale ha un parameter di tipo intero e poi tre interfacce verso l'esterno (ovvero i gates): un gate di input (il modulo riceve quindi i messaggi da questo gate), uno di output (usato dal modulo per comunicare verso l'esterno) e uno di input/output (è l'unione di un gate di input e uno di output). È possibile definire array di gates (es. **input** in[]), la dimensione può essere definita oppure no (verrà determinata automaticamente quando il modulo verrà collegato ad altri).

COMPOUND MODULE

Vediamo ora un esempio di compound module:

```
module Node
{
      parameters:
            int address;
      gates:
            inout port[];
      submodules:
            app: App;
            routing: Routing;
            queue[sizeof(port)]: Queue;
      connections:
            routing.localOut --> app.in;
            routing.localIn <-- app.out;
            for i=0..sizeof(port)-1
            {
                  routing.out[i] --> queue[i].in;
                  routing.in[i] <-- queue[i].out;
                  queue[i].line <--> port[i];
            }
}
```

In questo caso il modulo modella un generico nodo di una rete: possiede un address e un numero indefinito (per ora) di gates inout. Essendo un compound module però è composto da degli altri moduli (simple o a loro volta compound, quelli di questo esmpio immaginiamoli definiti in precedenza); nello specifico un Node è composto da un modulo di tipo App chiamato *app*, un modulo di tipo Routing di nome *routing* e un array di moduli (la dimensione dell'array è uguale al numero di porte, anche questo indice assumerà un valore certo al momento della definizione della *network*) di tipo Queue di nome queue. Oltre a definire i submodules, sono descritti anche i collegamenti tra i gates di questi submodules: il gate di nome *localOut* di routing (è un output gate) è collegate al gate (di input) di nome *in* di app (le frecce vanno dal gate di output a quello di input, nel caso di gates inout le freccie sono bidirezionali). Più interessante è il ciclo **for** usato per collegare gli array di gates dei vari componenti.

NETWORK

Vediamo ora come costruire una network istanziando più moduli Node:

```
network Network
{
	types:
	channel C extends ned.DatarateChannel
	{
		datarate = 100Mbps;
	}
	submodules:
		node1: Node;
		node2: Node;
		node3: Node;

	connections:
		node1.port++ <--> C <--> node2.port++;
		node2.port++ <--> C <--> node3.port++;
		node3.port++ <--> C <--> node1.port++;
}
```

Con questa descrizione possiamo vedere varie cose: all'inizio viene definito un channel (C) che estende un tipo predefinito (DatarateChannel, vedremo altri canali più avanti) assegnando un preciso valore al parameter *datarate* (posseduto da DatarateChannel); andando avanti vediamo che la network è costituita da 3 nodi (node1,node2,node3) di tipo Node (definito prima) mentre le connessioni sono fatte in modo particolare: con port++ si intende che si usa l'indice corrente del vettore port (all'inizio è 0) e che poi si incrementa di 1 questo indice:

esempio equivalente:

```
node1.port[0] <--> C <--> node2.port[0]
node2.port[1] <--> C <--> node3.port[0]
node3.port[1] <--> C <--> node2.port[1]
```

Quindi ora ciascun node si ritroverà assegnato il valore 2 come dimensione dell'array *port*.
Notiamo pure che il collegamento tra i nodi sfrutta un channel di tipo C cioè a datarate di 100 Mbps.

CHANNEL

Esistono tre tipi di channel:

- IdealChannel: non ha alcun parameter e indica un channel a trasferimento immediato, senza delay o errori; una connessione senza channel equivale a una connessione con IdealChannel;
- DelayChannel: possiede due parameters; *delay* (double) che indica il propagation delay dei messaggi. È necessario specificare una unità di tempo (es. 5s ,5ms, 5us, etc); *disabled* (booleano, valore di default *false*), quando settato a *true* il channel "droppa" tutti i messaggi che lo attraversano.
- DatarateChannel: possiede i parameters del DelayChannel e in più *datarate* per specificare il data rate del channel, utile per calcolare la durata di trasmissione dei messaggi(si specifica pure una unità di misura: bps, kbps, Mbps, Gbps). Il valore 0 è il valore di default e indica una banda infinita. Gli altri parameters sono *ber (Bit error rate)* e *per (Packet error rate)* per specificare una probabilità [0,1] che ci siano errori di trasmissione; quando il channel "decide" sulla base di un numero random che è avvenuto un errore, setta un apposito flag nel messaggio che dovrà essere controllato dal modulo destinazione; valore di default 0.

Esempi di estensione di channel:

```
channel Ethernet100 extends ned.DatarateChannel
{
      datarate = 100Mbps;
      delay = 100us;
      ber = 1e-10;
}

channel DatarateChannel2 extends ned.DatarateChannel
{
      double distance @unit(m);
      delay = this.distance / 200000km * 1s;
}
```

PARAMETERS

I parameters, come detto in precedenza, sono variabili appartenenti a un modulo o channel. È interessante definire l'utilità del modificatore *volatile* nella dichiarazione di un parameter e anche definire come si assegnano i valori a questi parameters.

Per assegnare un valore ai parameters ci sono varie possibilità.
Una è quella di farlo quando si estende un modulo (analogamente ai channel):

```
simple PingApp extends App
{
      parameters:
            protocol = "ICMP/ECHO"
            sendInterval = default(1s);
```

```
            packetLength = default(64byte);
}
```

I parameters sono già di proprietà del modulo App ma in PingApp essi assumono un valore preciso (la parola chiave default indica il valore da utilizzare se non ne viene specificato uno all'interno del file di configurazione omnetpp.ini).

Vediamo un'altra possibilità:

```
module Host
{
      submodules:
            ping : PingApp
            {
                  packetLength = 128B; // always ping with 128-byte packets
            }
      ...
}
```

In questo caso è un compound module che assegna un preciso valore al parameter *packetLenght* del submodule ping di tipo *PingApp*. Altri esempi sono presenti sul manuale omnet++.

```
network Network
{
      submodules:
            host[100]: Host
             {
                  ping.timeToLive = default(3);
                  ping.destAddress = default(0);
            }
      ...
}

network Network
{
      parameters:
            host[*].ping.timeToLive = default(3);
            host[0..49].ping.destAddress = default(50);
            host[50..].ping.destAddress = default(0);
      submodules:
            host[100]: Host;
      ...
}
```

Il carattere '*' viene usato per simboleggiare una sequenza di caratteri escluso il punto (in questo caso serve a settare il parameter *timeToLive* in tutti i moduli host evitando molte righe di codice noioso). Mentre l'espressione 0..49 indica la successione di numeri da 0 a 49 con incrementi di uno (analogamente 50.. indica da 50 fino all'ultimo indice del vettore).

Un parameter può essere assegnato anche tramite file di configurazione:

```
Network.host[*].ping.sendInterval = 500ms # for the host[100] example
Network.host*.ping.sendInterval = 500ms # for the host0,host1,... example
**.sendInterval = 500ms
```

Il doppio asterisco serve a indicare/sostituire una sequenza di caratteri compreso il punto (è un meccanismo per risparmiare digitazioni); risulta molto utile quando la gerarchia da attraversare per raggiungere un parameter è lunga, si risparmia anche se comunque il meccanismo ovviamente è molto generale (in questo caso qualsiasi modulo possegga un parameter *sendInterval*, gli verrà settato a 500ms).

```
**.sendInterval = 1s + normal(0s, 0.001s) # or just: normal(1s, 0.001s)
```

L'esempio quì sopra invece mostra come si possono assegnare valori random a un parameter, in questo caso assegnamo a sendInterva un valore secondo la distribuzione gaussiana (normal) a media 1 e varianza 0.001.

```
**.sendInterval = default
**.sendInterval = ask
```

Con la parola *default* si specifica di assegnare al parameter il valore di default (che deve essere specificato come visto prima nel file .ned); con *ask* invece fa sì che il valore venga richiesto all'utente tramite interfaccia grafica al momento del lancio del programma di simulazione.

Parliamo adesso del modificatore *volatile*.
Questo modificatore specifica che l'espressione legata al valore del parameter viene ricalcolata ogni qualvolta il parameter viene letto; questo è molto utile quando, infatti, il valore del parameter non è costante ma si desidera che cambi, vediamo un esempio:

```
simple Queue
{
      parameters:
            volatile double serviceTime;
}

**.serviceTime = simTime()<1000s ? 1s : 2s # queue that slows down after 1000s
```

Inserendo nel file di configurazione la riga scritta sopra indica che il parameter *serviceTime* assumerà il valore 1s durante i primi 1000s secondi di simulazione (simTime() ritorna il tempo di simulazione corrente), dopodichè assumerà il valore 2s!

GATES

Vediamo qualche particolarità sui gates:

```
simple GridNode
{
      gates:
            inout neighbour[4] @loose;

}
```

L'attributo @loose serve a indicare che i gates dell'array potrebbero non essere connessi ad altri.

```
simple WirelessNode
{
```

```
        gates:
                input radioIn @directIn;
}
```

L'attributo @directIn indica invece che il gate non sarà mai connesso a nessun altro perché i messaggi gli verranno spediti direttamente (si usa la funzione *sendDirect* invece della normale *send*, vedremo meglio in seguito).

CHANNELS

Vediamo qualche esempio anche per i collegamenti tra gates:

```
a.g++ <--> Ethernet100 <--> b.g++;
```

Ethernet100 è un channel specificato in precedenza (non esiste di default), questo esempio è come quello fatto in precedenza con il channel C.

```
a.g++ <--> Backbone {cost=100; length=52km; ber=1e-8;} <--> b.g++;
```

Anche Backbone è un channel specificato in precedenza e presenta dei parameter (cost,length e ber) che vengono settati contestualmente alla dichiarazione del channel usato (come si può vedere)

```
a.g++ <--> Backbone {@display("ls=green,2");} <--> b.g++;
```

Quì sopra invece si specifica attraverso l'attributo *@display* il colore del collegamento (verde).

```
a.g++ <--> {delay=10ms;} <--> b.g++;
a.g++ <--> {delay=10ms; ber=1e-8;} <--> b.g++;
```

Quando non viene specificato un preciso channel type, il linguaggio ne deduce uno (IdealChannel, DelayChannel o DatarateChannel) in base ai parameters che vengono specificati. Nel primo caso verrà scelto un DelayChannel, nel secondo un DatarateChannel.

MULTIPLE CONNECTIONS

Vediamo ora qualche esempio per le connessioni multiple:

Chain

```
module Chain
{
        parameters:
                int count;
        submodules:
                node[count] : Node
                {
                        gates:
                                port[2];
                }
        connections allowunconnected:
        for i = 0..count-2
         {
```

```
            node[i].port[1] <--> node[i+1].port[0];
      }
}
```

connections allowunconnected serve a permettere che alcuni gates possano restare non connessi a nessuno.

Binary Tree

```
simple BinaryTreeNode
{
      gates:
            inout left;
            inout right;
            inout parent;
}

module BinaryTree
{
      parameters:
            int height;
      submodules:
            node[2^height-1]: BinaryTreeNode;
      connections allowunconnected:
      for i=0..2^(height-1)-2
      {
            node[i].left <--> node[2*i+1].parent;
            node[i].right <--> node[2*i+2].parent;
      }
}

module RandomGraph
{
      parameters:
            int count;
            double connectedness; // 0.0<x<1.0
      submodules:
            node[count]: Node
            {
                  gates:
                        in[count];
                        out[count];
            }
      connections allowunconnected:
      for i=0..count-1, for j=0..count-1
      {
            node[i].out[j] --> node[j].in[i] if i!=j &&
      uniform(0,1)<connectedness;
      }
}
```

All'interno del ciclo for il collegamento verrà creato solo se la condizione del blocco if sarà verificata.

PARAMETRIC SUBMODULES TYPES

Il tipo di un submodule può essere specificato attraverso una stringa utilizzando i *moduleinterface* e la parola chiave *like*. Vediamo un esempio:

```
network Net6
{
	parameters:
		string nodeType;
	submodules:
		node[6]: <nodeType> like INode
		{
			address = index;
		}
	connections:
	...
}
```

Notiamo come il tipo dell'array *node[]* non è specificato ma lo sarà assegnando un valore alla stringa *nodeType*, questo meccanismo è identico a quello delle interfacce dei linguaggi come Java. Se per esempio avessimo un module *SensorNode* che estendesse *INode* allora potremmo assegnare a nodeType la stringa "SensorNode" in fase di configurazione. È possibile anche usare una notazione che prevede solo <> senza alcun parameter di tipo stringa, in questo caso si dovrà però settare il *typename* del module da file di configurazione (lo vedremo in seguito). Vediamo un esempio di INode e SensorNode:

```
moduleinterface INode
{
	parameters:
		int address;
	gates:
		inout port[];
}


module SensorNode like INode
{
	parameters:
		int address;
		...
	gates:
		inout port[];
		...
}
```

Lo stesso approccio (seppur meno utilizzato) è applicabile anche ai channel:

```
a.g++ <--> <channelType> like IMyChannel <--> b.g++;
a.g++ <--> <channelType> like IMyChannel {@display("ls=red");} <--> b.g++;
```

PACKAGES

Omnet++ usa un sistema di packaging analogo a Java per organizzare I moduli in varie directories.
In particolare il file *package.ned* è molto utile perché rappresenta l'intero package; un commento scritto al suo interno funge da documentazione per l'intero package. In esso inoltre è contenuta la root della gerarchia di packaging. Se ad esempio volessimo che il nostro progetto fosse nel package `org.example.myproject` ma non desideriamo le directory org, example e myproject (che resterebbero vuote!) basta dichiarare all'interno di package.ned:

```
package org.example.myproject;
```

Analogamente si usa il meccanismo di *import* (esattamente come Java) per poter usare moduli definiti in altri package all'interno del proprio progetto.

SIMPLE MODULES

SIMULATION CONCEPTS

Cerchiamo di capire meglio come funziona un sistema ad eventi discreti come Omnet++.
Un sistema ad eventi discreti è un sistema in cui dei cambiamenti di stato (eventi) avvengono in certi istanti di tempo, inoltre si assume che tra due eventi consecutivi non accada niente di "interessante" quindi non avviene nessun cambiamento di stato; questo ovviamente va in contrasto con i sistemi continui.
Alcuni esempi di evento sono:

- L'inizio della trasmissione di un pacchetto/messaggio;
- La fine della trasmissione di un pacchetto/messaggio;
- La scadenza di un timeout.

Ciò implica che tra due eventi come l'inizio e la fine della trasmissione di un pacchetto non avviene nulla di "interessante" (cioè lo stato del pacchetto resta "in trasmissione"); il concetto di interesse è ovviamente relativo in base a cosa si vuole modellare; potremmo per esempio essere interessati all'inizio e la fine di trasmissione di ogni singolo bit.

L'istante in cui un evento si verifica si chiama in Omnet++ *arrival time*.

THE EVENT LOOP

Un sistema ad eventi discreti mantiene l'insieme di eventi futuri in una struttura dati chiamata FES (Future Event Set) o FEL (Future Event List). Il funzionamento segue queste pseudocodice:

```
initialize -- this includes building the model and
              inserting initial events to FES
while (FES not empty and simulation not yet complete)
{
      retrieve first event from FES
      t:= timestamp of this event
      process event
      (processing may insert new events in FES or delete existing ones)
}
finish simulation (write statistical results, etc.)
```

La fase di inizializzazione serve per costruire il simulation model, quindi i vari moduli necessari per ognuno dei quali verrà richiamato un metodo per l'inizializzazione. Successivamente in loop ogni evento viene processato; gli eventi vengono sempre ordinati per arrivat time in modo tale che nessun evento corrente possa avere un effetto sui precedenti. La simulazione termina quando non ci sono più eventi da schedulare o quando si è raggiunto un limite di tempo di simulazione preimpostato e a questo punto tutti i risultati e i dati statistici vengono salvati in appositi files.
Cerchiamo di essere più precisi sull'ordine di schedulazione degli eventi: ogni messaggio ha un arrival time e uno *scheduling priority value*; detto questo, dati 2 messaggi:

1) Il messaggio con minore arrival time è processato prima. In caso i 2 messaggi abbiano lo stesso arrival time,
2) quello con minor scheduling priority value viene processato prima. Se le priorità sono uguali allora,
3) viene processato per primo quello che è stato schedulato o inviato per primo.

SIMULATION TIME

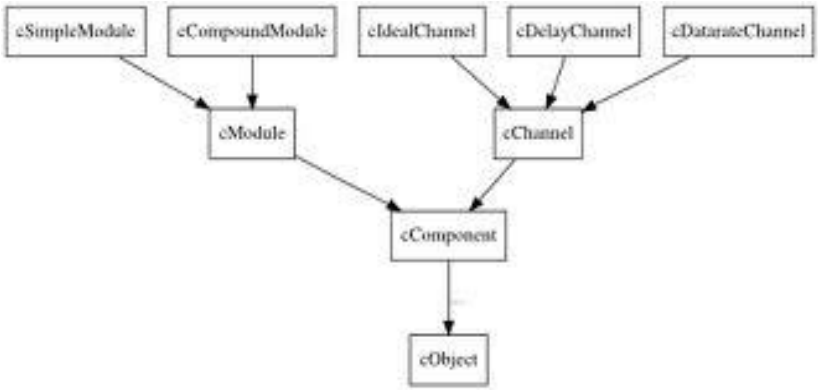
Il tempo di simulazione corrente può essere ottenuto con la funzione *simTime()*.
In Omnet++ il tempo di simulazione viene rappresentato tramite il tipo C++ *simtime_t* che è un typedef della classe *SimTime*. SimTime mantiene il tempo di simulazione attraverso un intero di 64 bit. La risoluzione è regolata tramite la variabile globale *scale exponent* (settabile solo dal file omnetpp.ini). questo esponente può assumere valori tra -18 (risoluzione dell'attosecondo) e 0 (secondo). Nella seguente tabella sono forniti alcuni valori di esponente con i range approssimativi che possono raggiungere:

Exponent	Resolution	Approx. Range
-18	1as	±9.22s
-15	1fs	±153.72 minutes
-12	1ps	±106.75 days
-9	1ns	±292.27 years
-6	1us	±292271 years
-3	1ms	±2.9227e8 years
0	1s	±2.9227e11 years

Sebbene il tempo non possa essere negativo possiamo vedere che un simtime_t può assumere valori negativi, questo potrebbe capitare dopo la valutazione di espressioni aritmetiche che coinvolgono tempi.

COMPONENTS, SIMPLE MODULES, CHANNELS

In Omnet++ i modules (simple o compound) e I channels sono chiamati *components.*
I components sono rappresentati dalla classe C++ *cComponent*. La classe astratta per i moduli *cModule* e la classe astratta per i channel *cChannel* derivano entrambe da cComponent.
cModule ha due sottoclassi: *cSimpleModule* e *cCompoundModule*. L'utente definisce i propri simple modules derivando da cSimpleModule. cChannel invece deriva in tre sottoclassi: cIdealChannel, cDelayChannel e cDatarateChannel; l'utente può creare i propri channel derivando da cChannel o da una delle sue sottoclassi. Vediamo il diagramma di ereditarietà:



Simple modules (e channels) possono essere programmati ridefinendo dei metodi delle classi padre. In particolare cComponent offre i seguenti metodi:

- `initialize()`. Questo metodo viene richiamato dopo che Omnet++ ha costruito la network (istanziando i vari moduli e connettendoli secondo le definizioni NED) e serve ad eseguire codice di inzializzazione. Ne esiste una versione con un valore intero per poter modellare una inizializzazione in più step.
- `finish()`. È richiamato quando la simulazione termina con successo, utile per il salvataggio di risultati e statistiche.

Per programmare il comportamento di un simple module è necessario ridefinire il metodo

`handleMessage(cMessage *msg)` il quale viene invocato ogni qualvolta il modulo riceve un messaggio; il simulation time non trascorre durante l'esecuzione del metodo ma solo tra le invocazioni.

Di seguito l'esempio più semplice possibile di simple module:

```
// file: HelloModule.cc
#include <omnetpp.h>
class HelloModule : public cSimpleModule
{
      protected:
            virtual void initialize();
            virtual void handleMessage(cMessage *msg);
};

// register module class with OMNeT++
Define_Module(HelloModule);
```

```
void HelloModule::initialize()
{
      ev << "Hello World!\n";
}

void HelloModule::handleMessage(cMessage *msg)
{
      delete msg; // just discard everything we receive
}
```

È molto importante la differenza tra il costruttore e il metodo initialize:

- il costruttore è invocato per l'allocazione di un componente, per cui non è possibile reperire informazioni relative ai parameters o ai gates perché il modulo è in fase di creazione;
- il metodo initialize è richiamato una volta che l'oggetto è stato allocato, qui è possibile non solo accedere ai parameters e ai gates, ma anche inizializzare timers (non sono altro che messaggi che un modulo manda a se stesso) e varibili membro.

Un discorso analogo può essere fatto tra finish e il distruttore:

- finish è usato solo per la memorizzazzione di risultati e statistiche;
- il distruttore serve a deallocare tutte quelle risorse che il modulo ha istanziato.

Vediamo in pseudocodice come si comporta il sistema:

```
perform simulation run:
      build network
        (i.e. the system module and its submodules recursively)
      insert starter messages for all submodules using activity()
      do callInitialize() on system module
            enter event loop // (described earlier)
      if (event loop terminated normally) // i.e. no errors
            do callFinish() on system module
      clean up

callInitialize()
{
      call to user-defined initialize() function
      if (module is compound)
            for (each submodule)
                  do callInitialize() on submodule
}

callFinish()
{
      if (module is compound)
            for (each submodule)
                  do callFinish() on submodule
      call to user-defined finish() function
}
```

Come preannunciato un modulo può essere inizializzato in più step, i metodi da ridefinire sono:

```
void initialize(int stage);
int numInitStages() const;
```

numInitStages conterrà un semplice return con il numero di step che si prevedono per l'inizializzazione. Omnet++ richiama per ogni modulo initialize(0), poi initialize(1) e così via a seconda di numInitStages; i moduli che ridefiniscono initialize() senza parametri equivalgono ad initialize(0).

PROGRAMMING WITH handleMessage

Le funzioni che principalmente vengono usate all'interno di handleMessage sono:

- `send()`, per inviare un messaggio ad un altro modulo (attraverso un proprio gate di out o inout);
- `scheduleAt()`, per implementare timers (il modulo manda un messaggio a se stesso!);
- `cancelEvent()`, per eliminare un evento schedulato con scheduleAt.

È importante capire che handleMessage viene richiamata ogni qualvolta un modulo riceve un messaggio, quindi è inutile dichiarare al suo interno informazioni globali (come lo stato) perché queste informazioni andrebbero perse non appena il metodo termina. Per mantenere informazioni che devono sopravvivere all'esecuzione del metodo (lo stato, delle strutture dati come le code, puntatori a messaggi riusabili come i timer,etc) quindi vanno dichiarate delle variabili attributo proprie della classe .

Vediamo un esempio classico di programmazione:

```
class FooProtocol : public cSimpleModule
{
      protected:
      // state variables
      // ...
      virtual void processMsgFromHigherLayer(cMessage *packet);
      virtual void processMsgFromLowerLayer(FooPacket *packet);
      virtual void processTimer(cMessage *timer);
      virtual void initialize();
      virtual void handleMessage(cMessage *msg);
};
// ...

void FooProtocol::handleMessage(cMessage *msg)
{
      if (msg->isSelfMessage())
            processTimer(msg);
      else if (msg->arrivedOn("fromNetw"))
            processMsgFromLowerLayer(check_and_cast<FooPacket *>(msg));
      else
            processMsgFromHigherLayer(msg);
}
```

Di seguito un altro esempio per realizzare un generatore di traffic:

```
class Generator : public cSimpleModule
{
      public:
      Generator() : cSimpleModule() {}
      protected:
      virtual void initialize();
      virtual void handleMessage(cMessage *msg);
};
```

```
Define_Module(Generator);

void Generator::initialize()
{
      // schedule first sending
      scheduleAt(simTime(), new cMessage);
}

void Generator::handleMessage(cMessage *msg)
{
      // generate & send packet
      cMessage *pkt = new cMessage;
      send(pkt, "out");
      // schedule next call
      scheduleAt(simTime()+exponential(1.0), msg);
}
```

Qui vediamo come si usi scheduleAt all'interno di initialize con lo scopo di poter cominciare autonomamente a trasmettere senza dover per forza aspettare un messaggio dall'esterno (anche perché un generatore di traffico non ha gates di input!). All'interno di send() la stringa "out" indica il nome di un gate di out definito nel NED file del modulo. Una volta inviato un messaggio si schedula un nuovo timer che scadrà in un tempo random secondo distribuzione esponenziale a media 1; ovviamente il tempo di ritardo va sommato a simTime() perchè questo evento dovrà avvenire nel futuro!

Ecco un altro esempio in cui il generatore di traffico usa un ritardo differenziato realizzando una generazione a burst:

```
class BurstyGenerator : public cSimpleModule
{
      protected:
      int burstLength;
      int burstCounter;
      virtual void initialize();
      virtual void handleMessage(cMessage *msg);
};
Define_Module(BurstyGenerator);

void BurstyGenerator::initialize()
{
      // init parameters and state variables
      burstLength = par("burstLength");
      burstCounter = burstLength;
      // schedule first packet of first burst
      scheduleAt(simTime(), new cMessage);
}

void BurstyGenerator::handleMessage(cMessage *msg)
{
      // generate & send packet
      cMessage *pkt = new cMessage;
      send(pkt, "out");
      // if this was the last packet of the burst
      if (--burstCounter == 0)
      {
            // schedule next burst
            burstCounter = burstLength;
            scheduleAt(simTime()+exponential(5.0), msg);
      }
      else
```

```
        {
                // schedule next sending within burst
                scheduleAt(simTime()+exponential(1.0), msg);
        }
}
```

In questo esempio si vede come accedere ai parameters di un modulo: usando la funzione *par()* che accetta una stringa contenente il nome del parameter desiderato, ovviamente il tipo C++ deve matchare con il tipo definito nel NED file. Per informazioni più dettagliate sui parameter si consiglia di guardare il manuale.

GATE OBJECTS

Vediamo ora qualche semplice funzione per accedere ai gates di un modulo.

```
cGate *outGate = gate("out");
```

In questo caso si ricava l'oggetto gate dato il suo nome sotto forma di stringa.
Nel caso in cui un gate sia di inout è possibile accedere alla specifica parte di input o di output (in fondo si tratta dell'unione di input e un output gate), il modo corretto di farlo è questo, supponiamo che il gate inout si chiami "g":

```
cGate *gIn = gate("g$i");
cGate *gOut = gate("g$o");
```

Oltre all'oggetto gate si può anche usare l'ID di un gate (questo è un identificativo univoco ed è molto utile nel caso di array di gates) in 2 modi diversi:

```
int gateId = gate("in")->getId(); // or:
int gateId = findGate("in");
```

Ecco alcuni esempi su come accedere ai singoli gate di un vettore:

```
for (int i=0; i<gateSize("out"); i++)
{
        cGate *gate = gate("out", i);
        //...
}
```

Oppure:

```
int baseId = gateBaseId("out");
int size = gateSize("out");
for (int i=0; i<size; i++)
{
        cGate *gate = gate(baseId + i);
        //...
}
```

E infine un esempio generale per vedere alcuni metodi della classe cGate (ev è lo stream che scrive nella finestra di debug dell'interfaccia grafica):

```
for (cModule::GateIterator i(this); !i.end(); i++)
{
        cGate *gate = i();
        ev << gate->getFullName() << ": ";
```

```
        ev << "id=" << gate->getId() << ", ";
        if (!gate->isVector())
                ev << "scalar gate, ";
        else
                ev << "gate " << gate->getIndex()
                << " in vector " << gate->getName()
                << " of size " << gate->getVectorSize() << ", ";
                ev << "type:" << cGate::getTypeName(gate->getType());
                ev << "\n";
}
```

SENDING AND RECEIVING MESSAGES

I messaggi sono un element fondamentale in Omnet++. Un modulo può creare, inviare, ricevere, immagazzinare, distruggere, schedulare e modificare messaggi.

I messaggi sono oggetti della classe *cMessage*, i pacchetti delle reti di comunicazione sono oggetti della classe *cPacket*, quest'ultima deriva da cMessage.Con l'istruzione

```
cMessage *msg = new cMessage("foo");
```

creiamo un nuovo messaggio, "foo" è solo un nome descrittivo usato per visualizzazioni e debug.

Self-Messages

Ecco un ulteriore esempio per capire meglio i self-message (timer), supponiamo di aver definito in initialize un attributo di tipo cMessage* chiamato *timeoutEvent* e un attributo chiamato *timeout* di tipo simtime_t per indicare la durata del timer:

```
void Protocol::handleMessage(cMessage *msg)
{
        if (msg == timeoutEvent)
        {
                // timeout expired, re-send packet and restart timer
                send(currentPacket->dup(), "out");
                scheduleAt(simTime() + timeout, timeoutEvent);
        }
        else if (...)
        {       // if acknowledgement received
                // cancel timeout, prepare to send next packet, etc.
                cancelEvent(timeoutEvent);
                ...
        }
        else
        {
                ...
        }
}
```

cancelEvent serve a eliminare il messaggio dal FES, cioè per fermare il timer. Usando invece *cancelAndDelete()* oltre a eliminarlo dal FES si opera proprio un delete cioè il messaggio viene deallocato.

Se un self-message è già schedulato (cioè inserito nel FES) non è possibile invocare un ulteriore scheduleAt, quello che si fa è questo:

```
if (timeoutEvent->isScheduled())
      cancelEvent(timeoutEvent);
scheduleAt(simTime() + delay, timeoutEvent);
```

Sending messages

Di seguito tutte le possibili signature per la funzione send:

```
send(cMessage *msg, const char *gateName, int index=0);
send(cMessage *msg, int gateId);
send(cMessage *msg, cGate *gate);
```

Esempi:

```
send(msg, "out");
send(msg, "outv", i); // send via a gate in a gate vector
```

Ed ecco il caso in cui si abbia un array di gates inout:

```
send(msg, "g$o");
send(msg, "g$o", i); // if "g[]" is a gate vector
```

Broadcasting e retransmissions

Quando si vuole realizzare un broadcast, viene subito in mente di invocare send più volte passando lo stesso messaggio…NO!!! Non si può inviare più volte lo stesso messaggio! Bisogna crearne dei duplicati!

Perché? Un messaggio è come un oggetto reale: non può essere in più posti contemporaneamente, una volta che sarà inviato esso non sarà più proprietà del modulo che lo invia, apparterrà al simulation kernel, il quale lo passerà al modulo destinazione. Il sender non potrà più riferirsi al suo puntatore. Una volta giunto al modulo destinazione, questi avrà piena autorità sul messaggio, potrà modificarlo, distruggerlo o reinviarlo. Lo stesso vale per i self-messages, essi apparterranno al simulation kernel fin quando non li restituirà ai rispettivi moduli. Per rafforzare questo concetto, le funzioni per l'invio di messaggi lanceranno un runtime error nel caso si tenti di inviare o schedulare un messaggio di cui non si è il proprietario.

Ecco un esempio per il broadcasting:

```
for (int i=0; i<n; i++)
{
      cMessage *copy = msg->dup();
      send(copy, "out", i);
}
delete msg;
```

Nel caso in cui non si desideri conservare il messaggio per invii futuri è possibile inviare n-1 duplicati e poi inviare l'originale:

```
int outGateBaseId = gateBaseId("out");
for (int i=0; i<n; i++)
      send(i==n-1 ? msg : msg->dup(), outGateBaseId+i);
```

Sulla base di quanto detto prima, se si ipotizza di inviare un messaggio che potrebbe necessitare di ritrasmissioni non si può assolutamente inviare l'originale, ma solo duplicati:

```
// (re)transmit packet:
cMessage *copy = packet->dup();
send(copy, "out");
```

e quando non saranno più necessarie le ritrasmissioni:

```
delete packet;
```

Direct message sending

Alle volte è conveniente essere in grado di inviare un messaggio direttamente all'input gate di un altro modulo. *sendDirect* serve a questo scopo:

```
sendDirect(cMessage *msg, cModule *mod, int gateId)
sendDirect(cMessage *msg, cModule *mod, const char *gateName, int index=-1)
sendDirect(cMessage *msg, cGate *gate)
```

Ecco un esempio (inviamo un messaggio a un modulo di pari livello al nostro chiamato "node2"):

```
cModule *targetModule = getParentModule()->getSubmodule("node2");
sendDirect(new cMessage("msg"), targetModule, "in");
```

Per il modulo destinazione non fa differenza se il messaggio arriva tramite send o sendDirect però:

- un gate di input per ricevere con sendDirect non deve avere connessioni ad altri gate (quindi un modulo deve avere dei gate dedicati per comunicazioni dirette);
- conviene assegnare la property @directIn nel NED file ai gates che riceveranno via sendDirect.

Esempio:

```
simple Radio
{
      gates:
      input radioIn @directIn; // for receiving air frames
}
```

Receiving a packet

Un pacchetto (cPacket object) quando viene ricevuto potrebbe contenere errori (ber del channel), in questo caso basta controllare un flag tramite il metodo *packet->hasBitError()* e se è true allora il pacchetto andrebbe scartato. Inoltre un pacchetto viene consegnato ad un modulo all'istante in cui corrisponde la fine della ricezione dello stesso (cioè quando il suo ultimo bit viene ricevuto). Ad ogni modo un modulo preferirebbe che l'evento venisse sollevato nell'istante in cui il pacchetto inizia ad essere ricevuto, per questo è possibile invocare il metodo *setDeliveryOnReceptionStart()* dell'input gate desiderato:

```
gate("in")->setDeliverOnReceptionStart(true);
```

Questo metodo può essere invocato solo su input gates di input modules.

Ecco un esempio per poter gestire la fine della ricezione del pacchetto:

```
simtime_t startTime, endTime;
if (pkt->isReceptionStart())
{
      // gate was reprogrammed with setDeliverOnReceptionStart(true)
      startTime = pkt->getArrivalTime(); // or: simTime();
      endTime = startTime + pkt->getDuration();
}
else
{
      // default case
      endTime = pkt->getArrivalTime(); // or: simTime();
      startTime = endTime - pkt->getDuration();
}
ev << "interval: " << startTime << ".." << endTime << "\n";
```

Aborting transmissions

Quando un sender desidera forzare l'interruzione di una trasmissione può agire così (supponendo che la trasmissione da interrompere stia avvenendo dal gate "out"):

```
gate("out")->getTransmissionChannel()->forceTransmissionFinishTime();
```

Questo permette al sender di poter inviare subito un altro messaggio senza problem di "busy channel", il receiver comunque dovrebbe essere avvisato dell'interrotta trasmissione per esempio attraverso l'invio di un apposito pacchetto.

NAVIGATING THE MODULE HIERARCHY

Può essere veramente utile ottenere un puntatore a un qualsiasi modulo:

- per poter inviare messaggi attraverso sendDirect();
- per invocare dei metodi del modulo senza la necessità di dovergli per forza inviare un messaggio (questa tecnica è molto usata in framework come inet/inetmanet).

Ogni modulo possiede un ID univoco che può essere acquisito tramite il metodo *getId()*:

```
int myModuleId = getId();
```

Se si conosce l'ID del modulo si può ottenere il suo puntatore grazie al simulation object (un oggetto globale):

```
int id = 100;
cModule *mod = simulation.getModule( id );
```

Naturalmente non bisogna per forza conoscere l'ID del modulo (chi ci potrebbe dare a runtime l'ID di un altro modulo?), quindi sono a disposizione dei metodi per muoversi su e giù per la gerarchia. Il modulo padre di un determinato submodule può essere acceduto così:

```
cModule *parent = getParentModule();
```

E ancora ad esempio un parameter del modulo padre può essere ottenuto nel seguente modo:

```
double timeout = getParentModule()->par( "timeout" );
```

Esistono altri metodi per accedere ai moduli passando una stringa contenente il nome e un indice(opzionale, serve a recuperare un preciso modulo all'interno di un array):

```
int submodID = compoundmod->findSubmodule("child",5);
cModule *submod = compoundmod->getSubmodule("child",5);
```

Nel caso sopra si cerca il modulo di indice 5 all'interno dell'array di module con nome "child", compoundmod è un puntatore al modulo padre ottenuto con getPatentModule(). *findSubmodule* restituisce l'ID del modulo trovato altrimenti -1; *getSubmodule* invece restituisce il puntatore, se il modulo non esiste ritorna NULL.

Vediamo un altro esempio per accedere a un submodule ad un livello di profondità di gerarchia più basso sfruttando un path relativo:

```
compoundmod->getModuleByRelativePath("child[5].grandchild");
```

Che equivale a:

```
compoundmod->getSubmodule("child",5)->getSubmodule("grandchild");
```

Attenzione a questa seconda scelta perchè se child[5] non esiste (la getSubmodule più a sinistra ritorna NULL) si avrà un crash perchè si invoca getSubmodule su un puntatore NULL!

Un puntatore a un modulo si può ottenere anche attraverso un gate:

```
cModule *neighbour = gate("out")->getNextGate()->getOwnerModule();
```

In questo modo si sfruttano le connessioni tra gates per "navigare". *getNextGate()* restituisce il prossimo gate a cui si è collegati (per gli input gates si usa *getPreviousGate()*).

DIRECT METHOD CALLS BETWEEN MODULES

Come anticipato prima è possibile richiamare dei metodi di altri moduli piuttosto che effettuare un message passing, questo evita la costruzione di collegamenti diretti tra vari moduli e un altro (cosa complessa e dispendiosa quando si vuole comunicare con un modulo distante più di un livello di gerarchia) e permette la modellazione di elementi come le tabelle (un esempio sono le RoutingTables e le InterfaceTables dell'inet framework).

Vediamo subito un esempio, vogliamo invocare il metodo *doSomething* di un modulo istanza della classe *Callee* (il modulo è all'interno dello stesso compound module del chiamante):

```
cModule *calleeModule = getParentModule()->getSubmodule("callee");
Callee *callee = check_and_cast<Callee *>(calleeModule);
callee->doSomething();
```

Ovviamente è necessaria la seconda riga per poter effettuare un casting da cModule altrimenti non si potrebbe invocare un metodo specifico della classe Callee. c*heck_and_cast* è messa a disposizione da Omnet++, essa opera un *dynamic_cast* C++ ma se il risultato è NULL solleva un'eccezione; questo evita di scrivere codice errato. Ad ogni modo tutto questo non basta perché il simulation kernel deve essere avvisato del fatto che si deve cambiare contesto (nel senso che non stiamo più eseguendo codice del modulo corrente ma codice del modulo *callee*!), per far ciò basta aggiungere un macro a doSomething:

```
void Callee::doSomething()
{
      Enter_Method("doSomething()");
      ...
}
```

Enter_Method notifica la GUI che a sua volta avvia un'animazione relativa alla chiamata del metodo, se non si desidera questa notifica alla GUI basta usare *Enter_Method_Silent* in modo del tutto analogo.

SIGNALS

Si riportano di seguito alcune semplici informazioni sui *signal* (inseriti a partire da Omnet++4.1), per dettagli maggiori si rimanda al manuale.

I *signal* possono servire per:

- esporre delle statistiche, senza specificare come memorizzarle;
- ricevere delle notifiche su dei cambiamenti del simulation model a runtime ed agire di conseguenza;
- implementare uno stile di comunicazione publish-subscribe tra i moduli; questo è utile quando il publisher e il subscriber delle informazioni non si conoscono potrebbero esserci relazione molti-a-uno o molti-a-molti tra di loro;
- emettere informazione per altri scopi, ad esempio input per animazioni customizzate.

I signal sono emessi dai components (modules e channels).I signal si propagano lungo la gerarchia di moduli fino alla radice. A qualunque livello, si possono registrare dei *listener* (callback objects), i quali saranno notificati (called back) qualora venga emesso un signal. Il risultato della propagazione verso l'alto è che i listeners registrati in un compound module possono ricevere tutti i segnali emessi dai componenti nel suo submodule tree. Un listener registrato nel system module può ricevere i segnali dall'intero sistema.

I signal sono identificati da un nome (stringa) ma per questioni di efficienza a runtime si usano degli identificatori numerici assegnati dinamicamente (*simsignal_t*). Il mapping tra signal names e signal IDs è globale quindi tutti i componenti che richiedono un certo signal name riceveranno lo stesso signal ID.

I listener possono sottoscriversi a signal names o IDs. Se per esempio avessimo 2 moduli diversi i cui tipi sono rispettivamente *Queue* e *Buffer*, entrambi emettono un signal di nome "lenght"; un listener che si sottoscrive a "lenght" in un compound module superiore riceverà notifiche da entrambe le istanze Queue e Buffer. Il listener dovrebbe indagare sulla sorgente del signal se vuole distinguerli (questo è possibile perché la sorgente del signal è passata come parametro del metodo di callback).
Quando si emette un signal, esso porta con se un valore, ciò è realizzato tramite vari overload del metodo *emit()* dei components e vari overload del metodo *receiveSignal()* nei listener; il valore può essere di un tipo primitivo oppure un puntatore a *cObject*, quest'ultimo permette di wrappare tutto ciò che non è un tipo primitivo e di inviarlo.

Vediamo ora come si registra un signal:

```
simsignal_t lengthSignalId = registerSignal("length");
```

Ecco ora un esempio per vedere come si emette un signal, sono necessari il signal ID (simsignal_t) e il valore:

```
emit(lengthSignalId, queue.length());
```

In questo caso abbiamo emesso la lunghezza di una coda. Adesso vedremo come emettere un oggetto; prima di tutto bisogna costruire il nostro oggetto derivando da cObject:

```
class cPreGateAddNotification : public cObject, noncopyable
{
      public:
            cModule *module;
            const char *gateName;
            cGate::Type gateType;
            bool isVector;
};
```

noncopyable serve a specificare che non occorre scrivere il costruttore di copia e la funzione *dup()* risparmiando di scrivere codice. Ora vediamo come emettere il signal *PRE_MODEL_CHANGE*(è un signal di Omnet++)

```
if (hasListeners(PRE_MODEL_CHANGE))
{
      cPreGateAddNotification tmp;
      tmp.module = this;
      tmp.gateName = gatename;
      tmp.gateType = type;
      tmp.isVector = isVector;
      emit(PRE_MODEL_CHANGE, &tmp);
}
```

hasListeners è intuibile e serve a capire se ci sono dei listener sul signal, altrimenti è inutile emetterlo!

Adesso con qualche esempio vediamo come sottoscriversi a un signal:

```
cIListener *listener = ...;
simsignal_t lengthSignalId = registerSignal("length");
subscribe(lengthSignalId, listener);
```

I listener ereditano dalla classe cListener (e ne implementano i metodi) e possono registrarsi a più signal. Ecco una versione più compatta:

```
cIListener *listener = ...;
subscribe("length", listener);
```

E' possibile sottoscriversi ad altri moduli, non per forza a quello locale. Ecco come poter ricevere signals da tutte le parti del model, ci si può sottoscrivere al livello del system module:

```
cIListener *listener = ...;
simulation.getSystemModule()->subscribe("length", listener);
```

Così facendo si riceverà il signal "length" da tutti i moduli del model (bisognerà poi discriminare in base alla sorgente del signal). Per deregistrarsi:

```
unsubscribe(lengthSignalId, listener);
```

oppure:

```
unsubscribe("length", listener);
```

Vediamo la dichiarazione della classe cListener:

```
class cIListener
{
      public:
      virtual ~cIListener() {}
      virtual void receiveSignal(cComponent *src, simsignal_t id, long l) = 0;
      virtual void receiveSignal(cComponent *src, simsignal_t id, double d) = 0;
      virtual void receiveSignal(cComponent *src, simsignal_t id, simtime_t t) =
      0;
      virtual void receiveSignal(cComponent *src, simsignal_t id, const char *s)
      = 0;
      virtual void receiveSignal(cComponent *src, simsignal_t id, cObject *obj)
      = 0;
      virtual void finish(cComponent *component, simsignal_t id) {}
      virtual void subscribedTo(cComponent *component, simsignal_t id) {}
      virtual void unsubscribedFrom(cComponent *component, simsignal_t id) {}
};
```

E' possibile ovviamente implementare solo l'overload che riguarda il tipo di dato che si desidera ricevere.

MESSAGES AND PACKETS

Qualche piccolo approfondimento sui cMessage e i cPacket.
Inutile dire che sono concetti fondamentali in Omnet++. cMessage possiede i seguenti attributi, alcuni sono usati dal simulation kernel, altri sono a disposizione del programmatore:

- *name* è una stringa che può essere stabilita dal programmatore a scopo descrittivo; essa viene visualizzata dall'interfaccia grafica;
- *message kind* è un campo intero. È utilizzabile per assegnare una categoria o un'identità al messaggio; i valori positivi sono a disposizione del programmatore, quelli negativi sono usati dalla simulation library.
- *scheduling priority* è usato raramente ma serve al simulation kernel per determinare l'ordine di consegna dei vari messaggi che hanno lo stesso *arrival time*;
- *send time, arrival time,source module, source gate, destination module, destination gate* mantengono utili informazioni riguardo l'ultimo invio o scheduling del message;
- *timestamp* (da non confondere con arrival time) è un campo di utilità sfruttabile dal programmatore per qualsiasi scopo; esso non è mai esaminato o modificato da simulation kernel.
- *parameter list, control info e context pointer* sono campi molto utili di cui si parlerà in seguito.

cPacket estende cMessage e presentano in più i seguenti attributi:

- *packet lenght* rappresenta la lunghezza del pacchetto in bit, è usato dal simulation kernel per calcolare la durata di trasmissione del pacchetto lungo un channel a un certo datarate;
- *encapsulated packet* aiuta a modellare protocol layers supportando il concetto di encapsulation e decapsulation;
- *bit error flag* indica se il pacchetto risulta compromesso a causa della trasmissione lungo un channel a ber non nulla; importantissimo per i ricevitori;
- *duration* contiene la durata di trasmissione dopo che il pacchetto è stato inviato lungo un channel con datarate;
- *is-reception-start* specifica se il pacchetto rappresenta l'inizio o la fine della ricezione dopo che è stato inviato lungo un channel con datarate. Questo flag è regolato dal flag *deliver-on-reception-start* del gate ricevente.

cMessage

Ecco alcuni esempi di creazione di cMessage (la prima riga è la signature del costruttore):

```
cMessage(const char *name=NULL, short kind=0);
```

```
cMessage *msg1 = new cMessage();
cMessage *msg2 = new cMessage("timeout");
cMessage *msg3 = new cMessage("timeout", KIND_TIMEOUT);
```

Una volta creato si possono richiamare I seguenti metodi per setting (esistono i corrispondenti getter):

```
void setName(const char *name);
void setKind(short k);
void setTimestamp(); //equivalente a setTimestamp(simTime())
void setTimestamp(simtime_t t);
void setSchedulingPriority(short p);
```

Parliamo un pò del campo *controlInfo*. Questo campo serve per corredare il messaggio/pacchetto di informazioni aggiuntive. L'esempio classico è dato dai protocolli di rete: ad esempio un modulo TCP quando invia un pacchetto al livello IP sottostante oltre ai dati dovrebbe specificare info aggiuntive tipo l'indirizzo IP del destinatario. Attraverso oggetti cObject (o derivati) è possibile assegnare queste informazioni che non sono parte integrante del pacchetto ma bensì aggiunte per il corretto funzionamento dei moduli successivi:

```
void setControlInfo(cObject *controlInfo);
cObject *getControlInfo() const;
cObject *removeControlInfo();
```

I metodi seguenti servono ad avere informazioni relative l'ultimo invio del messaggio in questione:

```
simtime_t getSendingTime() const;
simtime_t getArrivalTime() const;
```

Altri metodi sono utili per capire da dove viene il messaggio:

```
int getSenderModuleId() const;
int getSenderGateId() const;
int getArrivalModuleId() const;
int getArrivalGateId() const;
cModule *getSenderModule() const;
cGate *getSenderGate() const;
cModule *getArrivalModule() const;
cGate *getArrivalGate() const;
```

Altri metodi invece ritornano risultati booleani (se ad esempio volessimo sapere se un messaggio proviene da un determinato gate):

```
bool arrivedOn(int gateId) const;
bool arrivedOn(const char *gatename) const;
bool arrivedOn(const char *gatename, int gateindex) const;
```

Adesso spostiamoci un pò sui self-messages. Ecco intanto 2 metodi utili, il primo per capire se il messaggio è realmente un self-message; il secondo per capire se il messaggio è schedulato (cioè inserito nel FES; ricordiamo che se è schedulato non è possibile chiamare scheduleAt()):

```
bool isSelfMessage() const;
bool isScheduled() const;
```

Nell'ambito dei self-message è utile l'attributo *contextPointer* (di tipo void*); questo campo infatti può contenere qualsiasi cosa. Se ad esempio un modulo lancia molti timer potrebbe usare questo campo per

salvare un riferimento al messaggio cui si riferisce il singolo timer; oppure si potrebbe salvare un riferimento a una struttura dati che contiene importanti informazioni sul contesto. Ecco le signature:

```
void setContextPointer(void *p);
void *getContextPointer() const;
```

cPacket

Parliamo adesso di cPacket. Vediamo il costruttore:

```
cPacket(const char *name=NULL, short kind=0, int64 bitLength=0);
```

E i metodi setter :

```
void setBitLength(int64 l);
void setByteLength(int64 l);
void addBitLength(int64 delta);
void addByteLength(int64 delta);
int64 getBitLength() const;
int64 getByteLength() const;
void setBitError(bool e);
bool hasBitError() const;
```

Particolarmente utili sono i metodi per l'encapsulation/decapsulation, vediamo le signature più un esempio di utilizzo:

```
void encapsulate(cPacket *packet);
cPacket *decapsulate();
cPacket *getEncapsulatedPacket() const;
```

decapuslate() decrementa la dimensione totale del pacchetto e ritorna il messaggio incapsulato (se esiste), altrimenti genera un errore; *getEncapsulatedPacket* invece restituisce il puntatore al pacchetto incapsulato (se esiste, altrimenti NULL) ma non lo toglie dal pacchetto "contenitore".

```
cPacket *data = new cPacket("data");
data->setByteLength(1024);
UDPPacket *udp = new UDPPacket("udp"); // subclassed from cPacket
udp->setByteLength(8);
udp->encapsulate(data);
ev << udp->getByteLength() << endl; // --> 8+1024 = 1032
cPacket *payload = udp->decapsulate();
```

Precisiamo che uno stesso pacchetto non può invocare encapsulate più di una volta! Si possono fare incapsulamenti successivi ma usando diversi messaggi.

MESSAGE DEFINITION

In questa sezione vediamo come poter estendere ulteriormente cMessage e cPacket senza dover implementare classi C++ ma editando speciali files .msg che verranno automaticamente trasformati in sorgenti .h e .cc da Omnet++. Come sempre per approfondimenti si consiglia di consultare il manuale.

Immaginiamo che ci serva un pacchetto che contenga anche gli indirizzi sorgente e destinazione e un contatore di hop. È possibile farlo scrivendo un file .msg in questo modo:

```
packet MyPacket
{
      int srcAddress;
      int destAddress;
      int remainingHops = 32;
};
```

Sarà il *message compiler* (invocato automaticamente al momento del build) a trasformare questa definizione in un file .h e un .cc. Se avessimo chiamato il file MyPacket.msg avremmo ottenuto i file *MyPacket_m.h* e *MyPacket_m.cc*, ecco cosa potremmo vedere nel file header:

```
class MyPacket : public cPacket
{
      ...
      virtual int getSrcAddress() const;
      virtual void setSrcAddress(int srcAddress);
      ...
};
```

MyPacket_m.cc conterrà l'implementazione dei costruttori e dei metodi setter e getter del MyPacket. Sarà possibile usare MyPacket all'interno di altri sorgenti includendo il file generato:

```
#include "MyPacket_m.h"
...
MyPacket *pkt = new MyPacket("pkt");
pkt->setSrcAddress(localAddr);
...
```

Vediamo che tipi di dato si possono usare per i packet nel file .msg:

- logical: **bool**
- integral types: **char**, **short**, **int**, **long**; and their unsigned versions **unsigned char**, **unsigned short**, **unsigned int**, **unsigned long**
- floating-point types: **float**, **double**
- C99-style fixed-size integral types: **int8_t**, **int16_t**, **int32_t**, **int64_t**; and their unsigned versions **uint8_t**, **uint16_t**, **uint32_t**, **uint64_t**; 1

- OMNeT++ simulation time: **simtime_t**
- **string**. Getters and setters use the const char* data type; NULL is not allowed. The object will store a copy of the string, not just the pointer.
- structs and classes, defined in message files or elsewhere
- typedef'd names declared in C++ and announced to the message compiler).

È possibile assegnare valori iniziali ai campi:

```
packet RequestPacket
{
      int version = HTTP_VERSION;
      string method = "GET";
      string resource = "/";
      int maxBytes = 100*1024*1024; // 100MiB
      bool keepAlive = true;
};
```

È possibile indicare che dei valori interi sono relative a una *enum*:

```
packet FooPacket
{
      int payloadType @enum(PayloadType);
};
```

La enum stessa dovrà essere dichiarata in un file .msg così:

```
enum PayloadType
{
      NONE = 0;
      UDP = 1;
      TCP = 2;
      SCTP = 3;
};
```

Il message compiler la trasformerà in una enum C++ però nella definizione di FooPacket andrà incluso il suo header file (di seguito il file FooPacket.msg completo):

```
cplusplus {{
#include "PayloadType_m.h"
}}

enum PayloadType;

packet FooPacket
{
      int payloadType @enum(PayloadType);
};
```

Si possono specificare array:

```
packet SourceRoutedPacket
{
      int route[4];
};
```

E i metodi che verranno creati a supporto sono:

```
virtual long getRoute(unsigned k) const;
virtual void setRoute(unsigned k, long route);
```

Se viene passato un indice che supera la dimensione dell'array verrà lanciato un errore.

Nel caso in cui non si desideri un array a dimensione prefissata:

```
packet SourceRoutedPacket
{
      int route[];
};
```

Verranno creati i metodi:

```
virtual long getRoute(unsigned k) const;
virtual void setRoute(unsigned k, long route);
virtual unsigned getRouteArraySize() const;
virtual void setRouteArraySize(unsigned n);
```

setRouteArraySize deve essere invocato prima di utilizzare l'array altrimenti esso verrebbe considerate a dimensione zero e quindi genererebbe un errore!

Di seguito un esempio particolare dove si usa un puntatore (è necessario passare per un typedes):

```
cplusplus {{ typedef Foo *FooPtr; }} // C++ typedef
class noncobject FooPtr; // announcement for the message compiler
packet Bar
{
      FooPtr fooPtr; // leaky pointer field
};
```

È ammessa l'ereditarietà anche nei file .msg (si eredita già da cMessage o cPacket):

```
packet Ieee80211DataFrame extends Ieee80211Frame
{
      ...
};
```

Ecco un esempio che possiamo definire completo:

```
cplusplus {{
#include <vector>
#include "IPAddress.h"
#include "Location.h"
#include "AppPacket_m.h"
typedef std::vector<int> IntVector;
typedef cModule *ModulePtr;
}};

class noncobject IPAddress;

struct Location;

packet AppPacket;

class noncobject IntVector;

class noncobject ModulePtr;

packet AppPacketExt extends AppPacket
{
      IPAddress destAddress;
      Location senderLocation;
```

```
      IntVector data;
      ModulePtr originatingModule;
}
```

Quì si può notare come sia possibile includere strutture dati come i vector C++ (analogamente le liste), la parola chiave *noncobject* inserita nelle righe di definizione preliminari serve ad avvisare il compiler che la classe **non** deriva da *cOwnedObject* (cioè un oggetto che può essere di proprietà esclusiva di un oggetto alla volta e di nessun altro; i cMessage e cPacket sono esempi di cOwnedObject).

C'è da dire che il codice auto-generato non può sempre rispondere ai nostri desideri. Se, allo stato di conoscenza attuale, noi modificassimo dei metodi delle classi generate questi andrebbero subito perduti dopo un semplice build del progetto. Come fare allora? Aggiungiamo una *property* alla definizione del pacchetto:

```
packet FooPacket
{
      @customize(true);
      int payloadLength;
};
```

Attraverso @customize il compiler creerà una classe FooPacket_Base; a questo punto potremo derivare FooPacket da FooPacket_Base (scrivendocela a manina) modificando i metodi che ci interessano per ottenere i comportamenti desiderati. È importante in ogni caso ridefinire il metodo *dup()* e l'operatore di assegnamento *operator=()*.

CLASSI CONTAINER

In Omnet++ si può usufruire di speciali classi container realizzate apposta per immagazzinare oggetti di classe cObject (come cMessage e cPar)

cQueue

cQueue è una classe container che agisce come una coda. Essa può contenere tutti i tuoi derivati da cObject (pressoché tutte le classi della Omnet++ library). Internamente implementa una lista doppiamente linkata. Gli elementi vengono inseriti in testa e rimossi dalla coda. I metodi base sono *insert()* e *pop()*:

```
cQueue queue("my-queue");
cMessage *msg;
// insert messages
for (int i=0; i<10; i++)
{
      msg = new cMessage;
      queue.insert(msg);
}
// remove messages
while(!queue.empty())
{
      msg = (cMessage *)queue.pop();
      delete msg;
}
```

Il metodo *length()* ritorna il numero di elementi all'interno della cQueue, *empty()* ritorna true se la coda è vuota.

front() e *back()* ritornano puntatori rispettivamente al primo e all'ultimo elemento della cQueue senza modificare il contenuto della stessa.

pop() invece elimina l'elemento in coda alla cQueue e ne restituisce il puntatore. È possibile anche usare il metodo *remove()* per eliminare un elemento noto il puntatore:

```
queue.remove(msg);
```

Di base una cQueue è gestita FIFO, si può comunque cambiare questo comportamento passando in fase di costruzione una funzione di comparazione che prende 2 cObject e ritorna un intero di valore -1, 0 o 1 a seconda del risultato del confronto:

```
cQueue queue("queue", someCompareFunc);
```

Quando si invocherà il metodo insert() la cQueue userà someCompareFunc tra l'oggetto passato e ognuno di quelli presenti per stabilire la posizione corretta di inserimento.

È inoltre possibile (in ogni caso) visitare tutti gli elementi attraverso un iterator; esso necessita del puntatore alla cQueue e un intero che specifichi la posizione di partenza (0 = coda; 1= testa). Si possono usare gli operatori *++()* e *--()* per avanzare, l'operatore *()* per ottenere il puntatore all'elemento corrente e il metodo *end()* per capire se si è raggiunta la fine della cQueue:

```
for( cQueue::Iterator iter(queue,1); !iter.end(), iter++)
{
      cMessage *msg = (cMessage *) iter();
      //...
}
```

cArray

Anche cArray è un container di cObject; cArray conserva puntatori piuttosto che creare delle copie, funziona come un array ma la sua dimensione cresce automaticamente quando si riempie; è implementato da un array di puntatori, quando si riempie viene riallocato. Vediamo degli esempi:

```
cArray array("array");
cPar *p = new cMsgPar("par");
int index = array.add( p );
```

Si può anche inserire un elemento ad un determinate indice (però se è già occupato verrà lanciato un errore):

```
cPar *p = new cMsgPar("par");
int index = array.addAt(5,p);
```

Ricerca di un elemento:

```
int index = array.find(p);

cPar *p = (cPar *) array[index];
```

Si possono fare ricerche anche in base al nome del cObject:

```
int index = array.find("par");
Par *p = (cPar *) array["par"];
```

Vediamo come rimuovere un oggetto dal cArray:

```
array.remove("par");
array.remove(index);
array.remove( p );
```

remove() comunque non dealloca l'oggetto! Per farlo:

```
delete array.remove( index );
```

cArray non ha iterator ma è semplice da esplorare:

```
for (int i=0; i<array.size(); i++)
{
      if (array[i]) // is this position used?
      {
            cObject *obj = array[i];
            ev << obj->getName() << endl;
      }
}
```

ROUTING SUPPORT: cTopology

cTopology è stata progettata per il support al routing. Un oggetto di questa class mantiene una rappresentazione della network sotto forma di grafo:

- ogni *node* corrisponde a un modulo (simple o compound);
- ogni *edge* corrisponde a un collegamento o a una serie di collegamenti di connessione.

È possibile specificare quali moduli considerare nel grafo; esso includerà tutte le connessioni tra i moduli selezionati. Le connessioni esistenti tra i *node* sono rappresentate come *edge*. Gli edge hanno una direzione proprio come i gates.

Vediamo meglio. Come prima cosa bisogna "estrarre" la topologia dalla network, ecco alcuni modi per farlo:

```
cTopology topo;
topo.extractByModuleType("Router", "Host", NULL);
```

In questo modo si prendono in considerazione come *node* solo i moduli (simple o compound) della classe Router o Host. Ecco un modo equivalente:

```
cTopology topo;
const char *typeNames[3];
typeNames[0] = "Router";
typeNames[1] = "Host";
typeNames[2] = NULL;
topo.extractByModuleType(typeNames);
```

Notiamo che l'ultimo elemento deve essere NULL in entrambi i casi. Si possono anche estrarre i moduli che possiedono un certo parameter:

```
topo.extractByParameter( "ipAddress" );
```

O quelli che possiedono una certa property (ad esempio tutti quelli con property @node):

```
topo.extractByProperty( "node" );
```

Una volta estratta la topologia è possibile esplorarla:

```
for (int i=0; i<topo.getNumNodes(); i++)
{
      cTopology::Node *node = topo.getNode(i);
      ev << "Node i=" << i << " is " << node->getModule()->getFullPath() <<
      endl;
      ev << " It has " << node->getNumOutLinks() << " conns to other nodes\n";
      ev << " and " << node->getNumInLinks() << " conns from other nodes\n";
      ev << " Connections to other modules are:\n";
      for (int j=0; j<node->getNumOutLinks(); j++)
      {
            cTopology::Node *neighbour = node->getLinkOut(j)->getRemoteNode();
            cGate *gate = node->getLinkOut(j)->getLocalGate();
            ev << " " << neighbour->getModule()->getFullPath()
            << " through gate " << gate->getFullName() << endl;
      }
}
```

Le corrispondenze tra un module e un node possono essere ottenute così:

```
cTopology::Node *node = topo.getNodeFor( module );
cModule *module = node->getModule();
```

La vera potenza di cTopology sta nel fatto che può determinare il cammino minimo tra due node, per supportare un routing ottimale. Vediamo un esempio pratico:

```
cModule *targetmodulep =...;
cTopology::Node *targetnode = topo.getNodeFor( targetmodulep );
topo.calculateUnweightedSingleShortestPathsTo( targetnode );
```

Viene eseguito l'algoritmo di Dijkstra e il risultato viene immagazzinato nell'oggetto cTopology. Chiamate successive al metodo *calculateUnweightedSingleShortestPathTo* sovrascrivono di volta in volta il risultato precedente.

In questo esempio si vede come si può attraversare il path fino al node target:

```
cTopology::Node *node = topo.getNodeFor( this );
if (node == NULL)
{
      ev < "We (" << getFullPath() << ") are not included in the topology.\n";
}
else if (node->getNumPaths()==0)
{
      ev << "No path to destination.\n";
}
else
{
      while (node != topo.getTargetNode())
      {
            ev << "We are in " << node->getModule()->getFullPath() << endl;
            ev << node->getDistanceToTarget() << " hops to go\n";
            ev << "There are " << node->getNumPaths()
            << " equally good directions, taking the first one\n";
            cTopology::LinkOut *path = node->getPath(0);
            ev << "Taking gate " << path->getLocalGate()->getFullName()
            << " we arrive in " << path->getRemoteNode()->getModule()-
            >getFullPath()
            << " on its gate " << path->getRemoteGate()->getFullName() << endl;
            node = path->getRemoteNode();
}
```

Qualche ulteriore esempio e funzionalità è riportato nel manuale.

RECORDING SIMULATION RESULTS

cOutVector

Un oggetto cOutVector è responsabile della scrittura di una serie di dati in un file. Il metodo *record()* e usato per memorizzare un valore con un timestamp. Esempio:

```
cOutVector responseTimeVec("response time");
```

Normalmente i cOutVector vengono dichiarati come membri di un modulo e il loro nome viene settato nel metodo initialize().

```
class Sink : public cSimpleModule
{
      protected:
      cOutVector endToEndDelayVec;
      virtual void initialize();
      virtual void handleMessage(cMessage *msg);
};

Define_Module(Sink);

void Sink::initialize()
{
      endToEndDelayVec.setName("End-to-End Delay");
}

void Sink::handleMessage(cMessage *msg)
{
      simtime_t eed = simTime() - msg->getCreationTime();
      endToEndDelayVec.record(eed);
      delete msg;
}
```

Tutti i cObject registrano i loro dati in un file .vec.

Output Scalars

Mentre gli output vectors registrano una grande quantità di dati durante un simulation run, gli output scalar sono pensati per registrare un singolo valore per simulation run. Si possono usare:

- per memorizzare dati riassuntivi alla fine di un simulation run;
- per determinare le dipendenze di alcune misurazioni dai parametri, modificandoli ed eseguendo più simulation run.

Si usa sempre il metodo record() per registrare il dato ma questo metodo viene invocato all'interno di finish().

```
void Transmitter::finish()
{
      double avgThroughput = totalBits / simTime();
```

```
      recordScalar("Average throughput", avgThroughput);
}
```

Nel prossimo esempio vedremo come si possono registrare anche I dati di oggetti *cStatistic* come ad esempio *cStdDev*, esso colleziona una serie di valori dei quali è poi possibile calcolare la media aritmetica, la deviazione standard, etc.

```
class Sink : public cSimpleModule
{
      protected:
      cStdDev eedStats;
      virtual void initialize();
      virtual void handleMessage(cMessage *msg);
      virtual void finish();
};

Define_Module(Sink);

void Sink::initialize()
{
      eedStats.setName("End-to-End Delay");
}

void Sink::handleMessage(cMessage *msg)
{
      simtime_t eed = simTime() - msg->getCreationTime();
      eedStats.collect(eed);
      delete msg;
}

void Sink::finish()
{
      recordScalar("Simulation duration", simTime());
      eedStats.record();
}
```

WATCHES

Sfortunatamente, variabili di tipo *int,double, long* e nemmeno le classi STL (*std::string, std:vector, etc*) non vengono mostrate, di default, dall'interfaccia grafica durante le simulazioni. Questo perché il simulation kernel, essendo una libreria, non conosce nulla dei tipi e variabili nel nostro codice sorgente. Omnet++ mette a disposizione *WATCH()* e un insieme di macro per rendere la variabili ispezionabili in Tkenv (l'ambiente grafico di simulazione). Le macro WATCH() sono di solito posizionate in initailize(), dato che dovrebbero essere eseguite una volta sola:

```
long packetsSent;
double idleTime;

WATCH(packetsSent);
WATCH(idleTime);
```

Anche membri di classi e struct possono essere "watched":

```
WATCH(config.maxRetries);
```

Queste variabili possono essere controllate durante le simulazioni facendo doppio click sul simple module desiderato e poi sul suo tab *Objects/Watches*

Per le STL esistono varianti di WATCH:

```
std::vector<int> intvec;
WATCH_VECTOR(intvec);
std::map<std::string,Command*> commandMap;
WATCH_PTRMAP(commandMap);
```

CONFIGURING SIMULATIONS

Dati di configurazione e di input per la simulazione si inseriscono in un file di configurazione solitamente chiamato *omnetpp.ini*. Per cominciare vediamone un semplice esempio che può essere usato per il progetto di esempio *Fifo* (a disposizione già quando si installa Omnet++).

```
[General]
network = FifoNet
sim-time-limit = 100h
cpu-time-limit = 300s
#debug-on-errors = true
#record-eventlog = true

[Config Fifo1]
description = "low job arrival rate"
**.gen.sendIaTime = exponential(0.2s)
**.gen.msgLength = 100b
**.fifo.bitsPerSec = 1000bps

[Config Fifo2]
description = "high job arrival rate"
**.gen.sendIaTime = exponential(0.01s)
**.gen.msgLength = 10b
**.fifo.bitsPerSec = 1000bps
```

Questo file è suddiviso in sezioni chiamate *[General], [Config Fifo1]* e *[Config Fifo2]*, ognuna delle quali contiene diverse *entries*.

File Syntax

Un file di configurazione Omnet++ è un file di testo ASCII, ma sono permessi caratteri non-ASCII nei commenti e nei valori stringa. Non c'è limite sulla dimensione del file o sulla lunghezza di una linea.

I commenti possono essere aggiunti alla fine di qualsiasi linea attraverso il carattere marker '*#*' , essi si estendono fino alla fine della linea e verranno ignorati durante il processing. Sono permesse anche le linee vuote (anch'esse ignorate).

Il file è *line oriented* e consiste in :

- `Section heading lines`: contengono il nome di una sezione tra parentesi quadre;
- `Key-value lines`: rappresentano coppie *<key> = <value>*; sono ammessi spazi (non richiesti) prima e dopo il segno uguale. Se la linea contiene più di un segno uguale, quello più a sinistra viene considerato il separatore della coppia key-value;

- `Directive lines`: attualmente l'unico tipo è *include* (seguito da un nome di file); permette l'inclusione di file.

Alcune key-value lines possono non trovarsi di seguito alla prima section heading line.

Le chiavi (keys) possono essere a loro volta classificate:

- Chiavi che non presentano caratteri '.' Rappresentano *configuration options* globali e per-run;
- Se una chiave contiene dei '.' allora viene considerata la sottostringa dopo l'ultimo '.'; se questa sottostringa contiene un trattino o è uguale a *typename*, la chiave è detta *per-object configuration option*;
- Altrimenti la chiave rappresenta un *parameter assignment*. Quindi non contengono trattini dopo l'ultimo '.'.

Linee lunghe possono essere interrotte usando la *backslash notation*: se l'ultimo carattere di una linea è '\' essa verrà unita alla linea successiva.

Esempio:

```
# This is a comment line
[General]                          # section heading
network = Foo                      # configuration option
debug-on-errors = false            # another configuration option
**.vector-recording = false        # per-object configuration option
**.app*.typename = "HttpClient"    # per-object configuration option
**.app*.interval = 3s              # parameter value
**.app*.requestURL = "http://www.example.com/this-is-a-very-very-very-very\
-very-long-url?q=123456789"        # a two-line parameter value
```

File inclusion

Come descritto prima è possibile includere ini file, attraverso *include*; i files inclusi possono anche provenire da percorsi diversi:

```
# omnetpp.ini
...
include params1.ini
include params2.ini
include ../common/config.ini
...
```

L'inclusione è trattata come semplice sostituzione di testo, anche se le sezioni del file incluso non cambieranno la sezione corrente del file principale. Esempio:

```
# incl.ini
foo1 = 1          # no preceding section heading: these lines will go into
foo2 = 2          # whichever section the file is included into
[Config Bar]
bar = 3           # this will always go to into [Config Bar]

# omnetpp.ini
[General]
include incl.ini  # adds foo1/foo2 to [General], and defines [Config Bar] w/ bar
baz1 = 4          # include files don't change the current section, so these
baz2 = 4          # lines still belong to [General]
```

SECTIONS

Ogni file ini può contenere una section [General] e diverse sections [Config <configname>]. L'ordine delle sections non ha importanza.

Section [General]

Le option più comunemente usate di questa section sono:

- L'option *network* seleziona il simulation model da mandare in run;
- La durata della simulazione può essere settata con le options *sim-time-limit* e *cpu-time-limit* (si possono usare le tipiche unità di misura del tempo come: *ms, s, m, h,* etc.).

Named Configurations

Sono sezioni nella forma [Config <configname>], dove <*configname*> è una stringa che per convenzione inizia per lettera maiuscola. Ad esempio per una simulazione di *Aloha*, omnetpp.ini potrebbe avere la seguente struttura:

```
[General]
...
[Config PureAloha]
...
[Config SlottedAloha1]
...
[Config SlottedAloha2]
...
```

L'interfaccia grafica permette poi di selezionare la configurazione desiderata con un dialog.

Quando qualche option non viene trovata in una Config, allora si andrà a cercare automaticamente nella section [General]. Ad ogni modo [General] non è obbligatoria, nel caso sia assente verrà trattata come una section [General] vuota. È comunque possibile fare in modo che una Config erediti da altre, questo può far evitare la riscrittura di option identiche:

```
[General]
...
[Config SlottedAlohaBase]
...
[Config LowTrafficSettings]
...
[Config HighTrafficSettings]
...
[Config SlottedAloha1]
extends = SlottedAlohaBase, LowTrafficSettings
...
[Config SlottedAloha2]
extends = SlottedAlohaBase, HighTrafficSettings
...
[Config SlottedAloha2a]
extends = SlottedAloha2
```

```
...
[Config SlottedAloha2b]
extends = SlottedAloha2
...
```

In questo modo quindi:

- Se una Config non “estende” nessuno, le options non trovate verrano ricercate in [General];
- Se invece “estende”, allora verrà cercata in ognuna delle Config “estese” nell’ordine in cui sono state speificate.

ASSINGNING MODULE PARAMETERS

Le simuazioni ricevono input dai parameters dei moduli, che possono essere assegnati nei NED files o in omnetpp.ini (in questo ordine). Dato che i parameters assegnati nei NED files non possono essere sovrascritti in omnetpp.ini si può pensare a questi come “fissi”. È più facile e flessibile assegnarli in omnetpp.ini.

In omnetpp.ini, i parameters sono identificati dai loro *full paths* (nomi gerarchici); questi nomi consistono in liste dot-separated di nomi di moduli (dal modulo di livello più alto, giù fino al modulo contenente il parameter), più il parameter name. Vediamo un esempio dove si setta il parameter *numHosts* del modulo di livello più alto e il parameter *transactionsPerSecond* del modulo *server*:

```
[General]
Network.numHosts = 15
Network.server.transactionsPerSecond = 100
```

Sono possibili anche assegnamenti di *typename* (nel caso in cui si usi nei NED files il costrutto *like* senza specificare parameters di tipo stringa tra <>):

```
[General]
Network.host[*].app.typename = "PingApp"
```

Using wildcard patterns

Un simulation model può aver un numero molto grande di parameters da configurare; quindi può essere abbastanza stressante settarli tutti uno per volta in omnetpp.ini (senza contare che se ne potrebbe dimenticare qualcuno). Omnet++ supporta *wildcard patterns*, i quali permettono di settare più parameters in una volta.

Sintassi:

- *?* : matcha qualsiasi carattere singolo ad eccezione del punto (.);
- *** : matcha zero o più caratteri ad eccezione del punto (.);
- ***** : matcha zero o più caratteri (qualsiasi carattere);
- *{a-f}* : matcha un carattere nel range a-f;
- *{^a-f}* : matcha un carattere NON nel range a-f;
- *{38..150}* : (numeric range) qualunque numero nel range 38..150; entrambi i limiti sono opzionali;
- *[38..150]* : (index range) qualunque numero tra parentesi quadre nel range 38..150; entrambi i limiti sono opzionali;
- *backslash (\)* : indica il significato speciale del carattere seguente.

Usando wildcards l'ordine diventa importante; se il nome di un parameters matcha più patterns, viene usata la prima corrispondenza; ciò implica che bisogna specificare prima i setting particolari e poi i generali:

```
[General]
*.host[0].waitTime = 5ms # specifics come first
*.host[3].waitTime = 6ms
*.host[*].waitTime = 10ms # catch-all comes last
```

Altro esempio:

```
[General]
*.*.queue[3..5].bufSize = 10
*.*.queue[12..].bufSize = 18
*.*.queue[*].bufSize = 6 # this will only affect queues 0,1,2 and 6..11
```

In quest'ultimo possiamo notare qualcosa: se invece di *.* si fosse messo ** le cose sarebbero potute cambiare! *.* indica che si deve considerare l'array di moduli di nome queue che si trova al terzo livello della gerarchia; se ci fosse un altro array con lo stesso nome al quinto livello, questo non verrebbe interessato dalla specifica configurazione. Usando ** invece anche l'array al quinto livello sarebbe stato configurato con quei dati! È importante fare attenzione quindi e distinguere * da **.

Adesso vediamo quale algoritmo segue Omnet++ nell'assegnazione dei singoli value:

- se il parameter è assegnato nel NED file, non può essere sovrascritto. Il value viene assegnato e il processo finisce;
- se il primo match è una linea *<parameter-fullpath> = <value>*, il value viene assegnato e il processo finisce;
- se il primo match è una linea *<parameter-fullpath> =default,* viene assegnato il value di default definito nel NED file (costrutto *default()*) e il processo finisce;
- se il primo match è una linea *<parameter-fullpath> =ask* , il value viene richiesto all'utente interattivamente;
- se non c'è stato alcun match e il parameter ha un valore di default viene assegnato quello e il processo finisce;
- altrimenti il parameter è dichiarato *unassigned* e gestito di conseguenza dalla user interface, la quale potrebbe generare un errore e report o potrebbe richiederlo interattivamente all'utente.

PARAMETER STUDIES

È esigenza comune che la simulazione debba essere eseguita più volte con parametri diversi per studiare il comportamento del modello al variare dei parameters d'ingresso. Omnet++ permette di automatizzare questo processo utilizzando il file di configurazione. Esempio:

```
[Config AlohaStudy]
*.numHosts = ${1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential(${0.2, 0.4, 0.6}s)
```

In questo caso la simulazione verrà eseguita in tutti 24 volte, dove numHosts assumerà i valori 1, 2, 5, 10, 20, 30, 40, 50 e, per ognuno di questi numeri, tre esecuzioni saranno lanciate con generationInterval exponential(0,2), exponential(0,4), exponential(0,6).

La sintassi *${…}* specifica una iterazione; è una specie di macro. Ad ogni iterazione la stringa ${…} verrà sostituita con il valore corrente; i valori non devono per forza essere dei numeri, e la sotituzione può avere luogo anche all'interno di stringhe:

```
*.param = 1 + ${1e-6, 1/3, sin(0.5)}
```

```
      ==>   *.param = 1 + 1e-6
            *.param = 1 + 1/3
            *.param = 1 + sin(0.5)

*.greeting = "We will simulate ${1,2,5} host(s)."
      ==>   *.greeting = "We will simulate 1 host(s)."
            *.greeting = "We will simulate 2 host(s)."
            *.greeting = "We will simulate 5 host(s)."
```

Si può anche dare un nome a queste iteration variables; in modo da poterle riferire in posti diversi del file:

```
[Config Aloha]
*.numHosts = ${N=1, 2, 5, 10..50 step 10}
**.host[*].generationInterval = exponential( ${mean=0.2, 0.4, 0.6}s )
**.greeting = "There are ${N} hosts"
```

Il riferimento a una iteration variable si ottiene attraverso la sintassi ($var); questo permette di avere dei loop:

```
**.foo = ${i=1..10} # outer loop
**.bar = ${j=1..$i} # inner loop depends on $i
```

Ovviamente questo meccanismo può essere usato in maniera errata:

```
**.foo = ${i=0..$j}
**.bar = ${j=0..$k}
**.baz = ${k=0..$i} # --> error: circular references
```

Per rimarcare il concetto che avviene una sostituzione testuale, mostriamo un esempio:

```
**.foo = ${i=1..3, 1s+, -}001s
```

foo assumerà i valori: 1001s, 2001s, 3001s, 1s+001s, -001s, il risultato della sostuzione non viene valutato immediatamente, quindi eventuali errori verrebbero sollevato in seguito.

Il body di una iterazione potrebbe contenere un punto esclamativo seguito dal nome di un'altra iteration variable. Questa sintassi denota una *parallel iteration*. Una parallel iteration non definisce un loop vero e proprio, ma in realtà la sequenza avanza in base alla variabile dichiarata dopo il punto esclamativo; in altre parole il '!' sceglie l'n-esimo valore dall'iteration, dove n è la posizione del valore corrente dell'iteration variable seguita dal '!'. Esempio:

```
**.plan = ${plan= "A", "B", "C", "D"}
**.numHosts = ${hosts= 10, 20, 50, 100 ! plan}
**.load = ${load= 0.2, 0.3, 0.3, 0.4 ! plan}
```

Nell'esempio sopra, l'unico loop è definito dalla variable *plan*, le altre due la seguono (per il primo valore di plan verranno presi il primo di *hosts* e il primo di *load*, per il secondo di plan verranno presi il secondo di hosts e il secondo di load e così via).

Maggiori dettagli sulle configurazioni sono riportati nel manuale.