UNIVERSITA' DEGLI STUDI DI CATANIA
FACOLTA' DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN INFORMATICA

GUASTELLA MARCO

PROGETTAZIONE E IMPLEMENTAZIONE DI UN SIMULATORE DI RETI PARALLELO

Tesi di Laurea

RELATORI:
Prof. Ing. S. Riccobene
Dott. G. Costantino

Anno Accademico 2008/2009

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Sistemi di calcolo

Nel 1966 Michael J. Flynn classifica i sistemi di calcolo a seconda della molteplicità del flusso di istruzioni e del flusso dei dati che possono gestire; in seguito questa classificazione è stata estesa con una sottoclassificazione per considerare anche il tipo di architettura della memoria. In base a questa classificazione ogni sistema di calcolo rientra in una delle categorie rappresentate in *figura 1.1*.

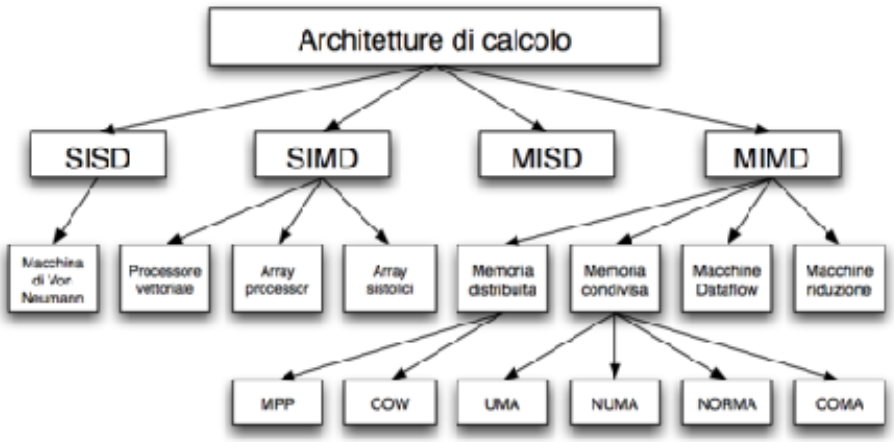

Figura 1.1: Architetture di calcolo

SISD

Single Istruction Single Data è un'architettura dove non c'è nessun parallelismo e le operazioni vengono eseguite sequenzialmente su un dato alla volta. E' la classica architettura di **Von Neumann**.

SIMD

Single Instruction Multiple Data è un'architettura in cui più unità elaborano dati diversi in parallelo. Questa architettura viene utilizzata da processori vettoriali o da processori che funzionano in parallelo. La SIMD è spesso usata dai supercomputer e con alcune varianti anche nei moderni microprocessori. Le istruzioni SIMD sono progettate per manipolare elevate quantità di dati in parallelo e per le usuali operazioni si appoggiano ad un altro insieme di istruzioni usualmente gestito dal microprocessore.

MISD

Multiple Instruction Single Data è un'architettura parallela in cui diverse unità effettuano diverse elaborazioni sugli stessi dati. Attualmente non esistono macchine MISD. Sono stati sviluppati alcuni progetti di ricerca ma non esistono processori commerciali che ricadono in questa categoria.

MIMD

Multiple Instruction Multiple Data è un'architettura parallela in cui diverse unità effettuano diverse elaborazioni su dati diversi.
Nello schema principale della classificazone di Flynn questa architettura ha una sotto classificazione:

1. *Sistemi a memoria distribuita*: ogni nodo ha una propria memoria riservata e se deve accedere ai dati memorizzati in un altro nodo deve farne richiesta attraverso uno scambio di messaggi tra i nodi;

2. *Sistemi a memoria condivisa*: più unità di calcolo pur eseguendo programmi differenti accedono alla stessa memoria;

3. *Macchine dataflow* : utilizzano un approccio data-driven dove le computazioni vengono eseguite solamente quando i dati delle elaborazioni sono disponibili;

4. *Macchine a riduzione*: utilizzano un approcio demand-driven dove le computazioni vengono eseguite solamente se vi è una richiesta dei dati da elaborare.

1.2 Grid Computing

Il termine **Grid computing** (*letteralmente "calcolo a griglia"*) sta ad indicare un paradigma del calcolo distribuito costituito da un'infrastruttura altamente decentralizzata e di natura variegata in grado di consentire ad un vasto numero di utenti l'utilizzo di risorse (prevalentemente CPU e storage) provenienti da un numero indistinto di calcolatori (anche e soprattutto di potenza non particolarmente elevata) interconnessi da una rete (solitamente, ma non necessariamente, Internet). Il termine griglia deriva dalla similitudine fatta dai primi ideatori del Grid Computing secondo i quali in un prossimo futuro si sarebbe arrivati a poter reperire risorse di calcolo con la stessa facilita' con la quale oggi si puo' usufruire dell'energia elettrica, ovvero semplicemente attaccandosi ad una delle tante prese presenti nel nostro appartamento (Power grid). Le prime definizioni di Grid computing, di cui si sente spesso parlare come della prossima rivoluzione dell'informatica (come a suo tempo fu il World Wide Web), risalgono di fatto a circa metà degli anni Novanta. Le 'griglie di calcolo' vengono prevalentemente utilizzate per risolvere problemi computazionali di larga scala in ambito scientifico e ingegneristico (la cosiddetta e-Science). Sviluppatasi originariamente in seno alla fisica delle alte energie (in inglese HEP = High Energy Physics), il loro impiego è già da oggi esteso alla biologia, all'astronomia e in maniera minore anche ad altri settori. Una grid è in grado di fornire agli utenti di un gruppo scalabile senza una particolare caratterizzazione geografica ne tantomeno istituzionale (gergalmente detto Virtual Organization, VO) la potenzialità di accedere alla capacità di calcolo e di memoria di un sistema distribuito, garantendo un accesso coordinato e controllato alle risorse condivise e offrendo all'utente la visibilità di un unico sistema di calcolo logico cui sottomettere i propri job. L'idea del Grid computing è scaturita dalla constatazione che in media l'utilizzo delle risorse informatiche di una organizzazione è pari al 5% della sua reale potenzialità. Le risorse necessarie sarebbero messe a disposizione da varie entità in modo da creare un'organizzazione virtuale con a disposizione un'infrastruttura migliore di quella che la singola entità potrebbe sostenere. Un altro importante fenomeno da evidenziare è la nascita accanto alle grandi GRID nazionali ed internazionali, di molteplici implementazioni su scala locale o metropolitana di sistemi distribuiti che mantengono le caratteristiche di una GRID. Tali sistemi vengono indicati con i termini **Local Area Grid** (*LAG*) e **Metropolitan Area Grid** (*MAG*) o Metropolitan Grid con chiaro riferimento alla classificazione introdotta nell'ambito del network. Come la coordinazione di Grid nazionali prevede la futura costituzione di un world wide Grid, le implementazioni di Grid locali o Metropolitane si avvicinano al mondo delle Intranet. Esse infatti forniscono un tipo di infrastuttura che

più semplicemente può essere integrata per l'introduzione del computing distribuito in ambito aziendale.
Un esempio di Grid è dato dalla *Figura 1.2*. In questo esempio si nota che l'user si connette ad un broker che gestisce le risorse. Il broker assegna le risorse all'user ma quest'ultimo non sà dove sono localizzate. Quindi l'user vede la griglia come un sistema unico.

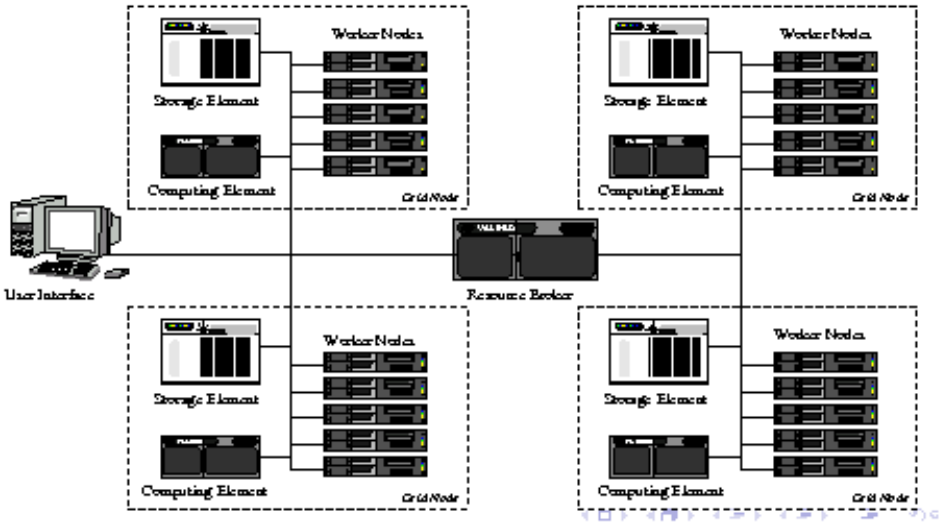

Figura 1.2: Grid

1.3 MPI

MPI (*Message Passing Interface*) è una specifica e portatile interfaccia per scrivere programmi *message-passing*, ed ha lo scopo di essere pratica, efficiente e flessibile per molto tempo.
Lo standard include:

- comunicazioni point-to-point;
- operazioni collettive;
- gruppi di processi;
- ambiente di comunicazione;

- processi topologici;
- bindings per Fortran77 e C++;
- gestione e richiesta informazioni;
- interfaccia del profilo.

Questo paragrafo parla delle librerie MPI in C++ dove per utilizzarle bisogna includere nel listato il file "mpi.h".
MPI usa oggetti chiamati comunicatori e gruppi per definire la collezione di processi che possono comunicare tra loro. Per un uso semplice possiamo utilizzare il comunicatore MPI_COMM_WORLD che comprende tutti i processi attivi per l'applicazione. Tramite un comunicatore ad ogni processo viene assegnato un unico identificatore chiamato **Rank**. I Ranks sono numeri interi che vanno da 0 a $n-1$, dove n è il numero di processi attivi per l'applicazione, che vengono utilizzati dal programmatore per specificare la sorgente e la destinazione dei messaggi.
Gli **MPI environment management routines** sono utilizzati per inzial-

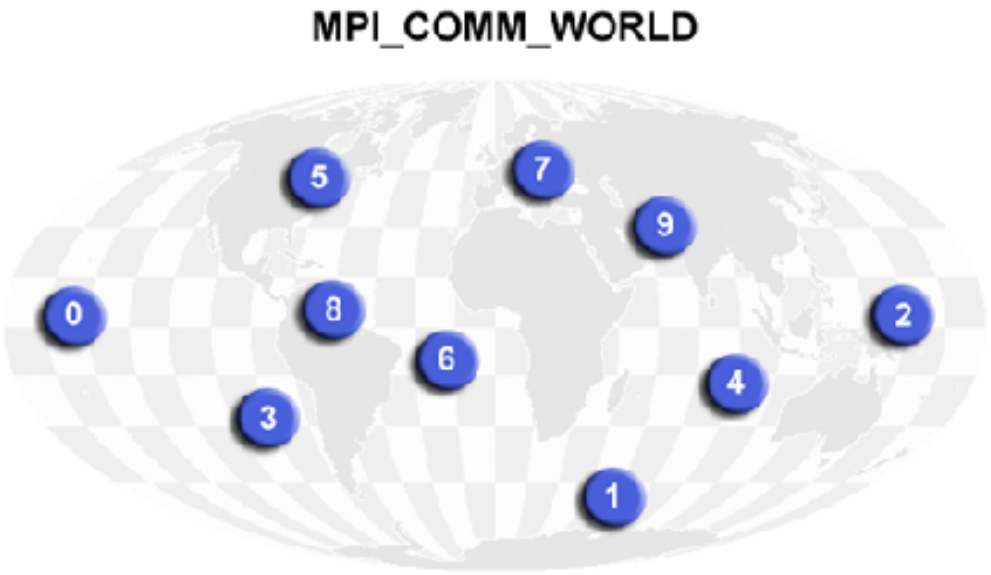

Figura 1.3: MPI_COMM_WORLD

izzare , terminare, indagare e identificare l'ambiente MPI. Alcune di queste routine sono le seguenti:

- *MPI_ Init(int *argc,char **argv)* inizializza l'esecuzione dell'ambiente MPI;

- *MPI_ Comm_ size(MPI_ Comm com,int *size)* memorizza in *size* il numero di processi assegnati al comunicatore *com*;
- *MPI_ Comm_ rank(MPI_ Comm com,int *rank)* memorizza in *rank* il range del processo che la esegue rispetto al comunicatore *com*;
- *MPI_ Abort(MPI_ Comm com,int errorcode)* forza tutti i processi assegnati al comunicatore *com* a terminare con codice di errore *errorcode*;
- *MPI_ Finalize()* termina l'esecuzione MPI.

Lo scambio di dati tra i vari processi si ha tramite lo scambio di messaggi. Per identificare i tipi di dato da scambiare si utilizzano gli *MPI_ Datatype*. Si possono utilizzare i datatype già esistenti come *MPI_ INT*, *MPI_ DOUBLE* oppure definirne dei nuovi. Un datatype particolare è *MPI_ PACKED* che identifica un tipo di dato eterogeneo. Per utilizzare *MPI_ PACKED* bisogna impacchettare i dati in un vettore e spacchettarli dopo lo scambio tramite le funzioni *MPI_ Pack()* e *MPI_ Unpack()*.
Alcune comunicazioni possono utilizzare la struttura *MPI_ Status* che identifica il suo stato:

```
typedef struct {
    int count;
    int MPI_SOURCE;
    int MPI_TAG;
    int MPI_ERROR;
#if (MPI_STATUS_SIZE > 4)
    int extra[MPI_STATUS_SIZE - 4];
#endif
} MPI_Status;
```

Ogni routine MPI ritorna *MPI_ SUCCESS* se va a buon fine altrimenti ritorna un codice di errore (definizione nel file mpi_errno.h).

1.3.1 Le comunicazioni point-to-point

Le comunicazioni point-to-point servono a far comunicare tramite un messaggio due processi. Le routine di comunicazione possono essere *bloccanti* e *non bloccanti*.
Le routine bloccanti hanno le seguenti caratteristiche:

- una send bloccante "return" dopo che ha modificato il buffer dell'applicazione per il riuso;

- una send bloccante può essere sincronizzata. In questo caso il processo ricevente deve ricevere il messaggio prima che la send "return";
- una send bloccante può essere asincrona se un buffer di sistema è usato per contenere i dati per eventuali consegne alle recv;
- una recv bloccante "return" quando riceve il messaggio dal processo remoto.

Le routine non bloccanti hanno le seguenti caratteristiche:

- send e recv non bloccanti "return" immediatamente dopo l'esecuzione;
- la sincronizzazione delle operazioni non bloccanti è più difficoltosa soprattutto nella gestione dei buffer che rischiano di essere sovrascritti;
- esistono routine MPI che servono a capire se le operazioni non bloccanti sono state completate. Ad esempio se mandiamo in esecuzione una send non bloccante il suo buffer di comunicazione non può essere modificato finchè non ha completato l'operazione.

Mpi garantisce che se un processo manda due messaggi verranno ricevuti nell'ordine in cui sono stati mandati. Se invece due processi diversi mandano un messaggio ad un altro processo l'ordine di arrivo di questi messaggi non è garantito .
Ecco un elenco di alcune routine bloccanti:

- *MPI_Send(void *buf,int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm com)* manda il messaggio in *buf* di lunghezza *count* e tipo di dato *datatype* al processo con rank *dest* in *com* con identificativo *tag*;
- *MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm com, MPI_Status)* riceve un messaggio di lunghezza *count* e tipo *datatype* in *buf* dal processo con rank *source* del comunicatore *com* identificato da *tag*. In *status* viene memorizzato lo stato del messaggio;
- *MPI_Probe(int source, int tag, MPI_Comm com, MPI_Status *status)* si blocca finchè non riceve un messaggio con rank *source* nel comunicatore *com* identificato da *tag*. Memorizza lo stato in *status*;
- *MPI_Get_count(MPI_Status *status, MPI_Datatype datatype, int *count)* memorizza in *count* il numero di elementi di tipo *datatype* in un messaggio identificato tramite *status*.

Dopo aver visto alcune routine bloccanti ecco un elenco di alcune non bloccanti:

- *MPI_ Isend(void *buf,int count,MPI_ Datatype datatype,int dest,int tag, MPI_ Comm com, MPI_ request *request)* simile a *MPI_ Send()* solo che è non bloccante. Il parametro *request* servirà alle funzioni che testano il completamento dell'operazione;
- *MPI_ Irecv(void *buf,int count,MPI_ Datatype datatype,int source,int tag,MPI_ Comm com, MPI_ Request *req)* simile a *MPI_ Recv()* solo che è non bloccante;
- *MPI_ Test(MPI_ Request *request,int *flag,MPI_ Status *status)* testa se una non blocking *request* è completa e memorizza il risultato in *flag* e lo stato in *status*;
- *MPI_ Wait(MPI_ Request *request,MPI_ Status *status)* si blocca finchè una non blocking *request* è completa.

1.3.2 Le comunicazioni collettive

Le comunicazioni collettive servono a far comunicare e sincronizzare più processi tra loro.
Alcune routine per le comunicazioni collettive sono le seguenti:

- *MPI_ Barrier (MPI_ Comm com)*: blocca i processi riferiti al comunicatore *com* finchè non viene chiamata da tutti i processi
- *MPI_ Bcast (void *buffer, int count, MPI_ Datatype datatype, int root, MPI_ Comm com)* manda un messaggio dal processo con rank *root* a tutti gli altri processi;
- *MPI_ Reduce (void *sendbuf, void *recvbuf, int count, MPI_ Datatype datatype, MPI_ Op op, int root, MPI_ Comm comm)* riduce tutti i valori in unico valore tramite un'operazione *op*. Il tipo MPI_Op assume un valore operazione come ad esempio *MPI_ SUM* che porta la routine a sommare tutti i valori nei *sendbuf* dei processi che la chiamano e a memorizzare il risultato in *recvbuf* del processo principale;
- *MPI_ Allreduce (void *sendbuf, void *recvbuf, int count, MPI_ Datatype datatype, MPI_ Op op, MPI_ Comm comm)* fa un'operazione di riduzione e memorizza il risultato nel *recvbuf* di ogni processo che la chiama;

- *MPI_Scatter (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcnt, MPI_Datatype recvtype, int root, MPI_Comm comm)* manda un dato dal processo *root* a tutti gli altri del gruppo;
- *MPI_Gather (void *sendbuf, int sendcnt, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)* serve ad un processo *root* a ricevere tutti i dati dagli altri processi del gruppo;

1.4 Introduzione alla simulazione

Per simulazione si intende l'imitazione delle operazioni eseguite nel tempo da un processo reale, al fine di studiarne le caratteristiche ed il comportamento. La simulazione ha il seguente scopo:

1. generazione di una storia artificiale del sistema;
2. studio e valutazione delle caratteristiche del sistema (analisi delle prestazioni);
3. risposta alla domanda "cosa accade se...";
4. analisi di sistemi "ipotetici";
5. confronto tra due o più sistemi;
6. determinazione di punti critici (bottlenecks);
7. predizione delle prestazioni del sistema;
8. capacity planning (capacità di pianificazione).

La risoluzione del problema tramite una simulazione non è sempre la soluzione migliore. Ci sono dei casi in cui i problemi sono risolvibili con metodi più semplici, p.es con soluzioni analitiche di modelli matematici, oppure con una semplice sperimentazione diretta del sistema esistente. In altri casi la simulazione può avere un costo proibitivo oppure non è in grado di convalidare il modello di simulazione.
I simulatori sono utilizzati anche nel campo delle reti multiprotocollo che costituiscono uno strumento molto utile per l'esecuzione di efficienti sperimentazioni, che possono avere come fine:

1. il progetto di nuovi protocolli, applicazioni o sistemi;
2. una migliore convalida del comportamento di protocolli già esistenti;

3. l'oppurtunità di studiare l'interazione tra i protocolli su larga scala in un ambiente controllato;

4. il confronto di sistemi già esistenti;

5. la determinazione del valore ottimale di un certo tipo di parametri;

6. la determinazione dei punti critici di un dato sistema;

7. la predizione delle prestazioni del sistema nel futuro;

8. la stima della capacità di un sistema.

1.5 Motivazioni della ricerca

Con la nascita dei moderni sistemi di calcolo multiprocessore e delle Grid un software deve essere progettato in maniera apposita per sfruttarne le potenzialità. Le tecniche di parallelizzazione sono nate a questo scopo. Questa tesi studia l'applicazione delle tecniche di parallelizzazione ai simulatori di rete multiprotocollo.
Il simulatore studiato in questa tesi è ns che serve come riferimento per capire le strategie utilizzate per la simulazione delle network.
La struttura di ns viene presa come riferimento per la progettazione di Pdnet, il simulatore nato per consentire l'implementazione di scheduler paralleli semplicemente. Lo scopo finale della tesi è quello di progettare uno scheduler parallelo per Pdnet e calcolarne le performance.

1.6 Organizzazione tesi

La tesi è organizzata nel seguente modo:

- il capitolo 2 introduce i concetti di modellazione di un sistema e parla in particolare dei modelli DES utilizzati nei simulatori seriali;

- il capitolo 3 parla di applicazioni parallele ed in particolare delle tecniche PDES utilizzate nei simulatori paralleli;

- il capitolo 4 descrive il funzionamento di ns focalizzandosi sulle strutture che utilizza per la simulazione delle network;

- il capitolo 5 descrive Pdnet focalizzandosi sulle differenze strutturali rispetto ad ns e fornendo i dettagli tecnici per l'evoluzione di questo simulatore;

- il capitolo 6 descrive lo scheduler parallelo implementato in Pdnet e fornisce i risultati sul calcolo delle performance di questo scheduler.

Capitolo 2

Modelli di simulazione

2.1 Sistemi e modelli

Un sistema è un insieme di elementi o fenomeni interdipendenti. L'evoluzione di un sistema può essere rappresentata da leggi descriventi le interconnessioni tra le varie parti. Le grandezze di uscita sono le grandezze che caratterizzano lo stato del sistema, mentre quelle di ingresso agiscono sul sistema dall'esterno influenzando le uscite *(Figura 2.1)*. Solitamente la scelta delle grandezze d'ingresso e uscita non è univoca ma dipende dal tipo di richieste per le quali il sistema è studiato. Un modello è la descrizione matematica del sistema e

Figura 2.1: Rappresentazione di un Sistema

delle relazioni intercorrenti tra le sue parti. Il modello deve includere tutte le caratteristiche del sistema reale che si vuole studiare. Se esiste una teoria o una previsione matematica che descrive il comportamento del sistema può essere utilizzata per la costruzione del modello. La modellazione matematica di un sistema si traduce nella determinazione della funzione di trasferimento $f(I, \alpha)$ che descrive il comportamento del sistema (in dipendenza delle grandezze d'ingresso). Spesso la funzione f contiene un set di parametri α_k che non possono essere determinati a priori ma vengono scelti (tramite confronto del modello con la realtà) per la riproduzione del comportamento

del sistema reale. La funzione f può essere determinata unicamente sulla base delle informazioni sperimentali, senza cercare di spiegare la natura delle relazioni tra le variabili (modelli "empirici") oppure tentando di spiegare le relazioni tra gli elementi del sistema (modelli "meccanicistici"):

- ogni modello meccanicistico è in qualche maniera empirico;
- un modello empirico può essere predittivo ma non aggiunge elementi utili al sistema.

L'insieme delle possibili variabili d'ingresso scelte per la costruzione di un modello non è in genere univoco:

- non è detto che tutte le variabili d'ingresso siano ugualmente importanti;
- e' bene non inserire un grande numero di variabili d'ingresso per non complicare il modello;
- non si devono tralasciare eventuali grandezze d'ingresso che abbiano un ruolo fondamentale nella descrizione del sistema.

Le variabili utilizzate nella costruzione di un modello possono schematicamente essere suddivise in:

1. **variabili di stato:** definiscono lo stato del sistema;
2. **tassi:** determinano la variazione delle variabili di stato;
3. **condizioni:** rappresentano elementi esterni al sistema che hanno influenza sul sistema;
4. **costanti:** dati fissi del problema / modello;
5. **parametri:** costanti il cui valore viene determinato tramite il confronto con la realtà (taratura).

La modellazione di un sistema ha i seguenti vantaggi:

1. organizzazione delle conoscenze e osservazioni empiriche;
2. comprensione del sistema;
3. rilevanza di componenti e/o interazioni;
4. facilita l'analisi del sistema.

In fase di progettazione del modello dobbiamo tenere conto dei seguenti rischi:

1. livello di astrazione non appropriato;
2. tendenza ad estrapolare i risultati del modello oltre il suo campo di applicabilità.

2.2 Descrizione dei modelli di simulazione

I modelli di simulazione servono a riprodurre virtualmente un sistema reale per studiarne il comportamento e analizzarne i risultati. Ognuno di questi modelli è caratterizzato dalle seguenti componenti:

- variabili di stato;
- eventi;
- entità;
- attributi;
- risorse;
- liste e code;
- attività;
- ritardi;
- stato;
- clock.

Le **variabili di stato** descrivono lo stato del sistema al tempo (simulato) t e permettono di interrompere e riprendere la simulazione. Sono definite come strutture dati del modello.
Gli **eventi** sono fenomeni che modificano lo stato del sistema. Un **evento interno** *(endogeno)* riguarda le variabili interne al modello, mentre un **evento esterno** *(esogeno)* riguarda le variabili esterne al modello.
Le **entità** sono oggetti esplicitamente definiti nel modello. Possono essere **dinamici** o **statici** , competere per ottenere le risorse ed essere accodati nelle rispettive code di attesa. Le **entità** hanno dei valori locali chiamati **attributi**.
Le **liste** e le **code** sono strutture dati che caratterizzano un insieme di entità

che non possono accedere alle trasformazioni successive in quanto la macchina risulta occupata.
Le **attività** sono una collezione di operazioni che trasformano lo stato di una componente. Nella simulazione a eventi discreti le attività fanno avanzare il tempo simulato t.
I **ritardi** sono la durata indefinita di un'attività, legata alle condizioni e all'evoluzione del sistema (attesa).
Lo **stato** descrive ad ogni istante di tempo la condizione del sistema.
Il **clock** regola lo scorrere del tempo di simulazione.

2.3 Progettazione di un modello di simulazione

Un modello di simulazione deve essere progettato per riprodurre alla perfezione il comportamento di un sistema reale. Per progettare un modello di simulazione si deve seguire il seguente schema:

1. **definizione degli obiettivi e delle problematiche da esaminare:** un' attenta analisi del problema consente di circoscriverne l'esame riducendo il successivo tempo di analisi;

2. **stesura di un modello concettuale:** consiste nella comprensione e modellazione del sistema 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;

3. **validazione del modello concettuale:** si tratta di un confronto con il sistema reale e della capacità del modello di offrire un immagine consistente della realtà;

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

5. **scrittura del modello in termini matematici**;

6. **calibrazione e valutazione:** si tratta di controllare che i parametri determinati non assumano valori al di fuori dell'intervallo dei valori possibili;

7. **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;

8. **analisi dei dati in uscita:** dopo aver raccolto i dati relativi ai parametri 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.

2.4 Discrete Event Simulation

Discrete event simulation (**DES**) è una potente tecnica per capire il comportamento del sistema. Le variabili di stato cambiano solo in corrispondenza di eventi discreti, determinati a loro volta da attività e ritardi.
Il **tempo simulato** è definito tramite una variabile (clock), da non confondere con il **tempo reale** del sistema da simulare e con il **tempo di esecuzione** che ci indica il tempo di elaborazione del programma di simulazione.
Il **tempo simulato** può essere avanzato in due modi:

1. per intervalli fissi (**unit-time**);
2. per eventi (**event-driven**);

Quando si utilizzano intervalli fissi si incrementa il clock di una quantità fissa Δ, si esamina il sistema per determinare gli eventi che devono aver luogo e si effetuano le necessarie trasformazioni. Si trattano tutti gli eventi con tempo di occorrenza $t_i \epsilon (t, t + \Delta)$. La scelta dell'intervallo Δ è importante perchè con questo tipo di gestione del tempo di simulazione si devono fare le seguenti considerazioni:

- eventi con tempi di occorrenza diversi possono essere trattati come simultanei;
- ci possono essere degli intervalli vuoti.

Nei meccanismi di avanzamento event-driven non si verificano i problemi descritti sopra. In questo caso si avanza il clock fino al tempo di occorrenza del prossimo evento. L'avanzamento del tempo in questo modo comporta le seguenti considerazioni:

- gli incrementi sono irregolari;

- gli eventi sono simultanei solo se hanno lo stesso tempo di occorrenza;
- si evitano tempi di inattività.

Per strutturare questi modelli di simulazione esistono quattro metodologie:

1. **Interazione tra processi:** il flusso di esecuzione di un processo emula il flusso di un oggetto (entità) attraverso il sistema. L'esecuzione procede finchè il flusso non viene bloccato, ritardato, terminato o inizia una nuova attività. Quando il flusso di un'entità viene bloccato il tempo di simulazione avanza al tempo di inizio previsto dalla successiva entità in esecuzione.

2. **Scansione di attività:** esiste un insieme di moduli in attesa di esecuzione (uno per attività). Avanza il tempo a intervalli fissi e periodicamente esegue un test sulle condizioni che determinano l'esecuzione degli eventi. Se le condizioni sono verificate esegue l'evento aggiornando le variabili di stato.

3. **"Tre fasi":** avanza il tempo simulato a intervalli fissi, rilascia le risorse mantenute dalle attività che risultano terminate dopo l'avanzamento ed esegue attività per le quali siano disponibili le risorse.

4. **Scheduling di eventi:** il tempo simulato avanza con un meccanismo event-driven. Lo **scheduler** di eventi mantiene una lista ordinata per tempo simulato di eventi futuri. La routine di evento aggiorna le variabili di stato e la lista di eventi (inserice, cancella o rinvia eventi). La routine di inizializzazione definisce lo stato iniziale del sistema.

2.5 Modello di esempio DES con scheduling di eventi

Questo paragrafo illustra la costruzione di un modello di simulazione **DES con scheduling di eventi** del sistema rappresentato in *figura 2.2.*
Il sistema in esame è una rete con quattro nodi (n1,n2,n3,n4) in cui il nodo n1 è connesso a n3 tramite un link l1, n2 a n3 tramite un link l2, n3 a n4 tramite un link l3 . Dai nodi n1 e n2 vengono spediti dei pacchetti in continuazione verso n4. Dai link può passare un solo pacchetto per volta. Nel nodo n3 esiste una coda di attesa (q) per i pacchetti che arrivano in n3 per essere spediti in n4 mentre l3 risulta occupato. I pacchetti in coda vengono mandati verso n4 nell'ordine in cui sono arrivati.
Lo scopo del modello è quello di calcolare il numero di pacchetti che arrivano

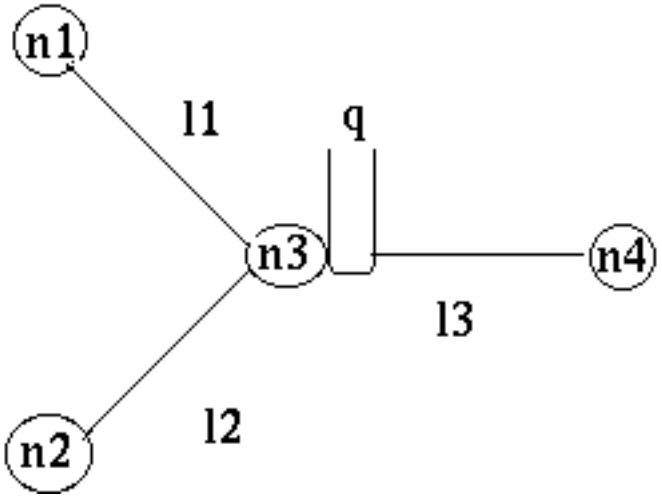

Figura 2.2: Sistema simulato

in n4 rispettivamente partiti da n1 e n2 e il numero medio di pacchetti in coda per un tempo di simulazione T.
Un set di stati è specificato per il sistema, e la sua evoluzione è vista come una sequenza nella forma $< s_0, (e_0, t_0), s_1, (e_1, t_1), s2.... >$ dove gli s_i sono gli stati, gli e_i gli eventi e t_i i tempi di occorrenza degli eventi (reali positivi con $t_i <= t_{i+1}$).
Lo stato s_0 è quello iniziale, l'evento e_0 occorre al tempo t_0 e porta il sistema allo stato s_1, l'evento e_1 occorre al tempo t_1 e porta il sistema allo stato s_2 etc.....
Fatta l'evoluzione del sistema si determinano i risultati prefissati.
In generale un evento dipende da un insieme di variabili di input (subset delle variabili di stato) e la sua esecuzione genera un nuovo stato modificando un subset delle variabili di stato del sistema. L'insieme delle variabili di stato del sistema descritto sono:

1. i tempi che impiega un pacchetto ad attraversare i link.
   - t_1**:** tempo che ad un pacchetto serve per attraversare il link l1;
   - t_2**:** tempo che ad un pacchetto serve per attraversare il link l2;
   - t_3**:** tempo che ad un pacchetto serve per attraversare il link l3.
2. la coda dei pacchetti (<n1,n2..>);
3. la variabile booleana **qblocked** che indica se l3 è bloccato (true) oppure no (false);

4. il tempo **clock** di simulazione;

5. **npack1:** indica il numero di pacchetti partiti da n1 che arrivano in n4;

6. **npack2:** indica il numero di pacchetti partiti da n2 che arrivano in n4;

Dopo l'esecuzione del modello si determinano i risultati raggiunti dalla simulazione in base allo stato finale del sistema. L'obiettivo del modello è quello di definire i seguenti risultati:

- **npack1**;
- **npack2**;
- **N:** il numero medio di pacchetti in coda. Utilizzando una funzione $N(t)$, rappresentata tramite una lista $< (t_i, n_i), (t_{i+1}, n_{i+1}) ..... >$ dove n_i è il numero di pacchetti in coda al tempo t_i, si calcola N:

$$N = \frac{1}{T} * \int_0^T N(t)dt$$

.

Lo stato del sistema è rappresentato dalla tupla {q, qblocked, npack1, npack2, $N(t)$}, dal tempo di simulazione clock, dai tempi di percorrenza dei link e dalla *event list*. I pacchetti sono identificati da un'etichetta che indica il nodo da cui sono partiti. Ogni pacchetto che parte da n1 ha un'etichetta n1, mentre se parte da n2 ha un'etichetta n2. La coda q se è vuota è rappresentata con <>, se ci sono elementi è definita come <n1, n2, n1...> con gli elementi messi in ordine di arrivo. Si utilizza un meccanismo FIFO (First In First Out) cioè il primo che arriva sarà il primo ad uscire dalla coda. Indicando con N_{coda} (inizialmente 0) il numero di elementi in coda si hanno le seguenti procedure di inserimento e cancellazione in coda:

enque(etichetta)
    q=q+<etichetta>//inserisce pacchetto alla fine della coda
    $N_{coda} = N_{coda} + 1$*//incrementa il numero di elementi in coda*
    $N(clock) = N_{coda}$*//assegna il valore alla funzione al tempo clock*
end procedure

deque()
if q = <> then
qblocked=false;
else
n=first(q) // ottiene e rimuove primo pacchetto in coda
$N_{coda} = N_{coda} - 1$*//decrementa il numero di elementi in coda*
$N(clock) = N_{coda}$
link3(n) //manda un pacchetto al link3 (descritta dopo)
end procedure

Lo **scheduler di eventi** utilizza una **event list**, caratterizzata da una seguenza di tuple (e,t) ordinate in base al tempo. Le procedure utilizzate dallo scheduler per interagire con l'event list ed eseguire gli eventi sono le seguenti:

- *schedule(e,t):* inserisce l'evento *e* nella event list nella giusta posizione in base al tempo;
- *getEvent(e,t):* rimuove il primo elemento dalla event list (quello con tempo minore) copiandolo in *e*. Il suo tempo di esecuzione lo copia in *t*;
- exec(e): esegue l'evento *e*.

I **link** devono simulare il tempo (**delay**) che un pacchetto perde per passare in esso. Questo si simula schedulando l'arrivo del pacchetto dopo un tempo t. Quando il pacchetto è passato ne può passare subito un'altro. Questo si simula schedulando la partenza del prossimo pacchetto dopo un tempo t.
Le seguenti procedure rappresentano i tre link e la procedura arrival3 che deve essere chiamata quando il pacchetto arriva in n3. Quest'ultima procedura serve a mettere il pacchetto in coda se l3 è occupato oppure se è libero fa passare il pacchetto direttamente:

link1()
schedule(arrival3(n1),clock+t1)//schedula l'arrivo del pacchetto
//schedula il passaggio del prossimo pacchetto
schedule(link1(),clock+t1)
end procedure

```
link2()
    schedule(arrival3(n2),clock+t2)
    schedule(link2(),clock+t2)
end procedure

link3(etichetta)
    //schedula la consegna al tempo di arrivo
    schedule(consegna(etichetta),clock+t3)
    schedule(deque(),clock+t3)
end procedure

arrival3(etichetta)
    if qblocked = false then
        qblocked=true //blocca il link
        link3(etichetta) // passa il pacchetto nel link3
    else
        enque(etichetta) // mette il pacchetto in coda
    endif
end procedure
```

Nelle procedure link1, link2, link3 si hanno le tre variabili di input t1, t2, t3 che sono i tempi di passaggio del pacchetto. Se i pacchetti sono tutti uguali si assegnano dei valori fissi all'inizio della simulazione (modello deterministico). Se invece i pacchetti hanno diverse dimensioni si assegnano, ad ogni passaggio, dei valori random entro un range x, y (modello stocastico) tramite una funzione *random(x,y)*. Se si suppone che per il link1 $x = 6$ e $y = 12$, per il link2 $x = 8$ e $y = 14$ e per il link3 $x = 6$ e $y = 10$, si modificano le procedure nel seguente modo:

```
link1()
    t1=random(6,12)
    //schedula l'arrivo del pacchetto
    schedule(arrival3(n1),clock+t1)
    //schedula il passaggio del prossimo pacchetto
    schedule(link1(),clock+t1)
end procedure
```

```
link2()
    t2=random(8,14)
    schedule(arrival3(n2),clock+t2)
    schedule(link2(),clock+t2)
end procedure

link3(etichetta)
    t3=random(6,10)
    //schedula la consegna al tempo di arrivo
    schedule(consegna(etichetta),clock+t3)
    schedule(deque(),clock+t3)
end procedure
```

Nella procedura link3 si schedula l'evento *consegna(etichetta)* che identifica la procedura di consegna del pacchetto a n4. Questa procedura non fa altro che incrementare **npack1** se l'etichetta del pacchetto è n1 altrimenti incrementa **npack2**.

```
consegna(etichetta)
    if etichetta=n1 then
        npack1=npack1+1
    else
        npack2=npack2+1
    endif
end procedure
```

La funzione **main** (descritta sotto) inizializza il sistema schedulando al tempo 0 due eventi: link1 e link2 che iniziano a mandare pacchetti a n4. Poi c'è il ciclo di routine che dura finchè la simulazione termina. Il ciclo di routine prende il prossimo evento dall' event list, lo esegue e aggiorna il clock al tempo dell'evento eseguito.

```
main()
    schedule(link1,0.0)
    schedule(link2,0.0)
    clock = nextEventclock()
    while clock <= T do
        getEvent(e,t)//prende il prossimo evento
        exec(e) //lo esegue
        clock = nextEventclock() //setta il timer al prossimo evento
    end while
end procedure
```

La tecnica di simulazione appena descritta è simile alla tecnica di simulazione usata in **ns** (Network Simulator) , un simulatore di reti multiprotocollo. Naturalmente quello descritto sopra è un modello molto semplificato per capire il funzionamento del DES.

Esecuzione del modello deterministico

Siano $t_1 = 8.0$, $t_2 = 10.0$, $t_3 = 6.0$ e $T = 26.0$ i valori assegnati alle variabili. Lo stato iniziale è dato dalla tupla {<>,false,0,0,<>} e dall'event list <(link1(),0.0),(link2(),0.0)>.
Si esegue il ciclo:

1.
   - clock = 0.0 exec(link1())
   - stato raggiunto {<>,false,0,0,<>}
   - event list <(link2(),0.0), (arrival3(n1),8.0), (link1(),8.0)>
2.
   - clock = 0.0 exec(link2())
   - stato raggiunto {<>,false,0,0,<>}
   - event list <(arrival3(n1),8.0),(link1(),8.0), (arrival3(n2),10.0), (link2(),10.0)>
3.
   - clock = 8.0 exec(arrival3(n1))
   - stato raggiunto {<>,true,0,0,<>}
   - event list <(link1(),8.0), (arrival3(n2),10.0), (link2(),10.0), (consegna(n1),14.0), (deque(),14.0)>
4.
   - clock = 8.0 exec(link1())
   - stato raggiunto {<>,true,0,0,<>}
   - event list <(arrival3(n2),10.0), (link2(),10.0), (consegna(n1),14.0), (deque(),14.0), (arrival3(n1),16.0), (link1(),16.0)>

5.
- clock = 10.0 exec(arrival3(n2))
- stato raggiunto {<n2>,true,0,0,<(10.0,1)>}
- event list <(link2(),10.0), (consegna(n1),14.0), (deque(),14.0), (arrival3(n1),16.0), (link1(),16.0)>

6.
- clock = 10.0 exec(link2())
- stato raggiunto {<n2>,true,0,0,<(10.0,1)>}
- event list <(consegna(n1),14.0), (deque(),14.0), (arrival3(n1),16.0), (link1(),16.0), (arrival3(n2),20.0), (link2(),20.0)>

7.
- clock = 14.0 exec(consegna(n1))
- stato raggiunto {<n2>,true,1,0,<(10.0,1)>}
- event list <(deque(),14.0), (arrival3(n1),16.0), (link1(),16.0), (arrival3(n2),20.0), (link2(),20.0)>

8.
- clock = 14.0 exec(deque())
- stato raggiunto {<>,true,1,0,<(10.0,1),(14.0,0)>}
- event list <(arrival3(n1),16.0), (link1(),16.0), (arrival3(n2),20.0), (link2(),20.0) ,(consegna(n2),20.0), (deque(),20.0)>

9.
- clock = 16.0 exec(arrival3(n1))
- stato raggiunto {<n1>,true,1,0,<(10.0,1),(14.0,0),(16.0,1)>}
- event list <(link1(),16.0), (arrival3(n2),20.0), (link2(),20.0) ,(consegna(n2),20.0), (deque(),20.0)>

10.
- clock = 16.0 exec(link1())
- stato raggiunto {<n1>,true,1,0,<(10.0,1),(14.0,0),(16.0,1)>}
- event list <(arrival3(n2),20.0), (link2(),20.0) ,(consegna(n2),20.0), (deque(),20.0), (arrival3(n1),24.0), (link1(),24.0)>

11.
- clock = 20.0 exec(arrival3(n2))
- stato raggiunto {<n1,n2>,true,1,0, <(10.0,1), (14.0,0), (16.0,1), (20.0,2)>}
- event list <(link2(),20.0) ,(consegna(n2),20.0), (deque(),20.0), (arrival3(n1),24.0), (link1(),24.0)>

12.
- clock = 20.0 exec(link2())
- stato raggiunto {<n1,n2>,true,1,0, <(10.0,1), (14.0,0), (16.0,1), (20.0,2)>}

- event list <(consegna(n2),20.0), (deque(),20.0), (arrival3(n1),24.0), (link1(),24.0), (arrival3(n2),30.0), (link2(), 30.0)>

13.
- clock = 20.0 exec(consegna(n2))
- stato raggiunto {<n1,n2>,true,1,1, <(10.0,1), (14.0,0), (16.0,1), (20.0,2)>}
- event list <(deque(),20.0), (arrival3(n1),24.0), (link1(),24.0), (arrival3(n2),30.0), (link2(), 30.0)>

14.
- clock = 20.0 exec(deque())
- stato raggiunto {<n2>,true,1,1, <(10.0,1), (14.0,0), (16.0,1), (20.0,1)>}
- event list <(arrival3(n1),24.0), (link1(),24.0), (consegna(n1),26.0), (deque(),26.0), (arrival3(n2),30.0), (link2(), 30.0) >

15.
- clock = 24.0 exec(arrival3(n1))
- stato raggiunto {<n2,n1>,true,1,1, <(10.0,1), (14.0,0), (16.0,1), (20.0,1), (24.0,2)>}
- event list < (link1(),24.0), (consegna(n1),26.0), (deque(),26.0), (arrival3(n2),30.0), (link2(), 30.0) >

16.
- clock = 24.0 exec(link1())
- stato raggiunto {<n2,n1>,true,1,1, <(10.0,1), (14.0,0), (16.0,1), (20.0,1), (24.0,2)>}
- event list < (consegna(n1),26.0), (deque(),26.0), (arrival3,30.0), (link2(), 30.0), (arrival3(n1),32.0), (link1(),32.0) >

17.
- clock = 26.0 exec(consegna(n1))
- stato raggiunto {<n2,n1>,true,2,1, <(10.0,1), (14.0,0), (16.0,1), (20.0,1), (24.0,2)>}
- event list < (deque(),26.0), (arrival3(n2),30.0), (link2(), 30.0), (arrival3(n1),32.0), (link1(),32.0) >

18.
- clock = 26.0 exec(deque())
- stato raggiunto {<n1>,true,2,1, <(10.0,1), (14.0,0), (16.0,1), (20.0,1), (24.0,2), (26.0,1)>}
- event list < (arrival3(n2),30.0), (link2(), 30.0), (arrival3(n1),32.0), (link1(),32.0), (consegna(n2),32.0), (deque(),32.0)>

19.
- clock = 30.0 esce dal ciclo perchè termina la simulazione

- stato finale {<n1,n2>,true,2,1, <(10.0,1), (14.0,0), (16.0,1), (20.0,1), (24.0,2), (26.0,1), (30.0,2)>}

I risultati della simulazione sono:

- npack1=2 (numero pacchetti partiti da n1 arrivati in n4);
- npack2=1 (numero pacchetti partiti da n2 arrivati in n4);
- la funzione $N(t)$.
  Da questa funzione si calcola il numero medio di elementi in coda N:

$$N = \frac{1}{26} * \int_0^{26} N(t)dt = 0.615$$

Esecuzione del modello stocastico

In questo tipo di simulazione si inserisce come input solo il tempo $T = 26.0$ perchè i valori t_1, t_2, t_3 vengono calcolati nelle funzioni link1, link2, link3.
Lo stato iniziale è dato dalla tupla {<>, false, 0, 0, <>} e dall'event list <(link1(), 0.0),(link2(), 0.0)>.
Si esegue il ciclo:

1. - clock = 0.0 exec(link1()) t1=10
   - stato raggiunto {<>,false,0,0,<>}
   - event list <(link2(),0.0), (arrival3(n1),10.0), (link1(),10.0)>
2. - clock = 0.0 exec(link2()) t2=9.0
   - stato raggiunto {<>,false,0,0,<>}
   - event list <(arrival3(n2),9.0), (link2(),9.0), (arrival3(n1),10.0), (link1(),10.0)>
3. - clock = 9.0 exec(arrival3(n2)) t3=6.0
   - stato raggiunto {<>,true,0,0,<>}
   - event list <(link2(),9.0), (arrival3(n1),10.0), (link1(),10.0), (consegna(n2),15.0), (deque(),15)>
4. - clock = 9.0 exec(link2()) t2=14.0
   - stato raggiunto {<>,true,0,0,<>}
   - event list <(arrival3(n1),10.0), (link1(),10.0), (consegna(n2),15.0), (deque(),15), (arrival3(n2),23.0), (link2(),23.0) >
5. - clock = 10.0 exec(arrival3(n1))

- stato raggiunto {<n1>,true,0,0,<(10.0,1)>}
- event list <(link1(),10.0), (consegna(n2),15.0), (deque(),15), (arrival3(n2),23.0), (link2(),23.0) >

6.
- clock = 10.0 exec(link1()) t1=7.0
- stato raggiunto {<n1>,true,0,0,<(10.0,1)>}
- event list <(consegna(n2),15.0), (deque(),15),(arrival3(n1),17.0), (link1(),17.0), (arrival3(n2),23.0), (link2(),23.0) >

7.
- clock = 15.0 exec(consegna(n2))
- stato raggiunto {<n1>,true,0,1,<(10.0,1)>}
- event list <(deque(),15),(arrival3(n1),17.0), (link1(),17.0), (arrival3(n2),23.0), (link2(),23.0) >

8.
- clock = 15.0 exec(deque()) t3=10.0
- stato raggiunto {<>,true,0,1,<(10.0,1), (15.0,0)>}
- event list <(arrival3(n1),17.0), (link1(),17.0), (arrival3(n2),23.0), (link2(),23.0), (consegna(n1),25.0), (deque(n1), 25.0) >

9.
- clock = 17.0 exec(arrival3(n1))
- stato raggiunto {<n1>,true,0,1,<(10.0,1), (15.0,0), (17.0,1)>}
- event list <(link1(),17.0), (arrival3(n2),23.0), (link2(),23.0), (consegna(n1),25.0), (deque(n1), 25.0) >

10.
- clock = 17.0 exec(link1()) t1=8.0
- stato raggiunto {<n1>,true,0,1,<(10.0,1), (15.0,0), (17.0,1)>}
- event list <(arrival3(n2),23.0), (link2(),23.0), (consegna(n1),25.0), (deque(n1), 25.0), (arrival3(n1),25.0), (link1(),25.0)>

11.
- clock = 23.0 exec(arrival3(n2))
- stato raggiunto {<n1,n2>,true,0,1,<(10.0,1), (15.0,0), (17.0,1),(23.0,2)>}
- event list <(link2(),23.0), (consegna(n1),25.0), (deque(n1), 25.0), (arrival3(n1),25.0), (link1(),25.0)>

12.
- clock = 23.0 exec(link2()) t2=8.0
- stato raggiunto {<n1,n2>,true,0,1,<(10.0,1), (15.0,0), (17.0,1),(23.0,2)>}
- event list <(consegna(n1),25.0), (deque(n1), 25.0), (arrival3(n1),25.0), (link1(),25.0), (arrival3(n2),31.0), (link2(),31.0)>

13.
- clock = 25.0 exec(consegna(n1))
- stato raggiunto {<n1,n2>,true,1,1,<(10.0,1), (15.0,0), (17.0,1),(23.0,2)>}
- event list <(deque(n1), 25.0), (arrival3(n1),25.0), (link1(),25.0), (arrival3(n2),31.0), (link2(),31.0)>

14.
- clock = 25.0 exec(deque()) t3=6
- stato raggiunto {<n2>,true,1,1,<(10.0,1), (15.0,0), (17.0,1), (23.0,2), (25.0,1)>}
- event list <(arrival3(n1),25.0), (link1(),25.0), (arrival3(n2),31.0), (link2(),31.0), (consegna(n1),31.0), (deque(),31.0)>

15.
- clock = 25.0 exec(arrival3(n1))
- stato raggiunto {<n2,n1>,true,1,1,<(10.0,1), (15.0,0), (17.0,1), (23.0,2), (25.0,2)>}
- event list <(link1(),25.0), (arrival3(n2),31.0), (link2(),31.0), (consegna(n1),31.0), (deque(),31.0)>

16.
- clock = 25.0 exec(link1()) t1=12
- stato raggiunto {<n2,n1>,true,1,1,<(10.0,1), (15.0,0), (17.0,1), (23.0,2), (25.0,2)>}
- event list <(arrival3(n2),31.0), (link2(),31.0), (consegna(n1),31.0), (deque(),31.0), (arrival3(n1),37.0), (link1(),37.0)>

17.
- clock = 31.0 termina la simulazione
- stato finale {<n2,n1,n2>,true,1,1,<(10.0,1), (15.0,0), (17.0,1), (23.0,2), (25.0,2), (31.0,3)>}
- event list <(link2(),31.0), (consegna(n1),31.0), (deque(),31.0), (arrival3(n1),37.0), (link1(),37.0)>

I risultati della simulazione sono:

- npack1=1 (numero pacchetti partiti da n1 arrivati in n4);
- npack2=1 (numero pacchetti partiti da n2 arrivati in n4);
- la funzione $N(t)$.
  Da questa funzione ci calcoliamo il numero medio di elementi in coda N:

$$N = \frac{1}{26} * \int_0^{26} N(t)dt = 0.615$$

Tempo di esecuzione

Il tempo di esecuzione di un modello dipende da fattori fisici. Si Suppone che per eseguire ogni ciclo di simulazione dei modelli sopra ci vuole un tempo medio t_{medio}. Il modello deterministico sarà eseguito mediamente in un tempo $19 * t_{medio}$, mentre quello stocastico in un tempo $17 * t_{medio}$.

Capitolo 3

Programmazione parallela

3.1 Definizione di programmazione parallela

Il calcolo parallelo si riferisce alla possibilità di velocizzare l'esecuzione di un codice per mezzo di elaboratori che sfruttano l'utilizzo simultaneo di un numero (anche) elevato di processori. Tutto ciò è ottenuto attraverso tecniche di programmazione che permettono di suddividere un codice sequenziale in sottoprogrammi o sottoprocessi che possono essere eseguiti simultaneamente su più processori.
Questo tipo di programmazione può avere i seguenti vantaggi:

- in principio se il numero di processori è p, il tempo t_p impiegato per risolvere il problema potrà essere $t_p = t_1/p$ (*essendo* t_1 *il tempo impiegato da un singolo processore*);
- e' possibile che parallelizzando il problema, il singolo processore dell'elaboratore parallelo richieda meno memoria;
- problemi computazionalmente onerosi possono essere affrontati in tempi ragionevoli senza che si debba aspettare una generazione di nuovi processori più prestanti.

Ci sono anche i seguenti svantaggi:

- la programmazione parallela è molto complessa;
- l'efficienza dipende dalla natura degli algoritmi, ma anche dalla natura dell'elaboratore parallelo.

I passi per la parallelizzazione di un codice sono i seguenti:

- capire il problema che si vuole parallelizzare;

- capire il codice seriale (se esiste);
- capire se il problema può essere parallelizzato;
- identificare le parti maggiormente pesanti e quelle su cui focalizzare l'attenzione;
- identificare i possibili "colli di bottiglia" e provare a eliminarli o ridurli;
- identificazione inibitori al parallelismo (*dipendenza dai dati*);
- studiare differenti algoritmi.

3.2 Legge di Amdahl

La legge di **Amdahl(1967)** ci aiuta a capire se un codice può essere parallelizzato. Si immagini di avere un codice composto da una frazione f perfettamente parallelizzabile e da una frazione $1-f$ che invece non può essere parallelizzata (*Figura 3.1*) e di avere p processori. Se con t_1 si identifica il

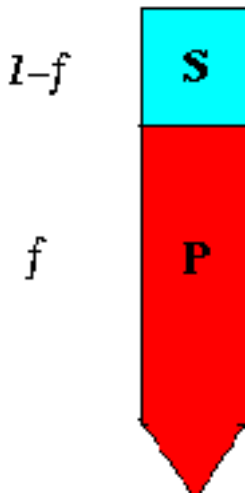

Figura 3.1: Frazione di codice parallelizzabile

tempo necessario per elaborare in modo completamente seriale il codice in questione si ha:

- $f * t_1$**:** tempo impiegato per elaborare in modo seriale la parte di codice P completamente parallelizzabile;

- $(1-f)*t_1$**:** tempo impiegato per elaborare in modo seriale la parte di codice S che non può essere parallelizzata;
- $f*t_1/p$**:** tempo impiegato per elaborare in modo parallelo su p processori la parte di codice P.

A questo punto si può ricavare t_p (*tempo necessario per elaborare l'intero codice su di un calcolatore parallelo con p processori*):

$$t_p = (1-f)*t_1 + f*t_1/p = t_1[f + (1-f)*p]/p.$$

E' possibile ora ricavare lo **speed-up:**

$$s_p = t_1/t_p = p/[f + (1-f)*p] = 1/[f/p + (1-f)].$$

Utilizzando un numero sempre più elevato di processori è possibile far tendere a zero il rapportof/p. Si ottiene perciò che s_p è minore o al massimo uguale ad $1/(1-f)$ (*Figura 3.2*). Questa è la legge di *Amdahl*, la quale

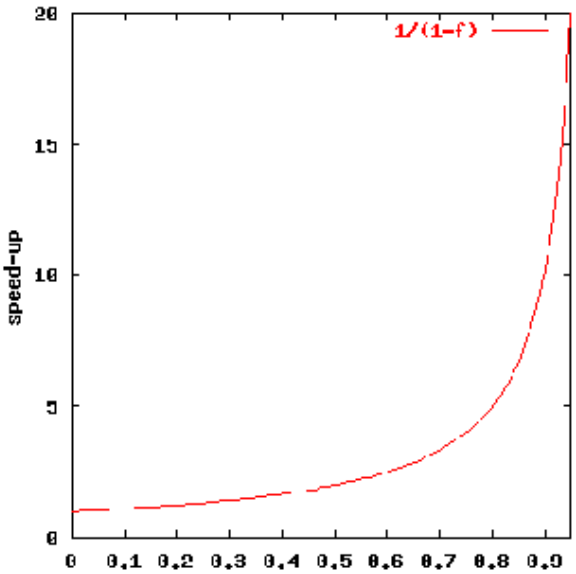

Figura 3.2: Legge di Amdahl

impone un limite superiore per il possibile speed-up in un codice in cui esiste una parte irrimediabilmente seriale. Spesso la frazione non parallelizzabile $(1-f)$ tende a zero quando le dimensioni computazionali del problema da parallelizzare aumentano e possiamo raggiungere speed-up elevati.
E' ovvio che questo è un discorso molto astratto ed in realtà l'analisi della

situazione è più complessa. Ad esempio non si è mai preso in considerazione il fatto che quando i processori cooperano in modo parallelo per elaborare una certa quantità di calcoli, esiste un tempo speso nel comunicare i risultati parziali necessari spesso ai vari processori per poter continuare l'elaborazione. Le prestazioni di una elaborazione parallela vengono quindi a dipendere da quante informazioni si devono scambiare i vari processori durante l'elaborazione parallela e dalla velocità con cui la rete di interconnessione scambia i dati tra i vari processori.
Per quanto riguarda il degrado delle prestazioni che nascono dalla necessità di comunicare dati durante l'elaborazione parallela, si deve notare che questo spesso può comportare per alcuni processori dei punti morti in cui questi rimangono inattivi perchè in attesa di dati da altri processori sottoposti ad un carico di lavoro maggiore, sono problemi legati alla sincronizzazione tra i processori.
Per quanto riguarda invece la velocità con cui vengono scambiati i dati tra i processori, due sono le proprietà della rete di interconnessione che risultano essere cruciali per le prestazioni dell'elaborazione parallela:

1. **la larghezza di banda** L della rete di interconnessione, cioè la quantità di bit al secondo che possono essere trasferiti;

2. **la latenza** l della rete, cioè il tempo necessario all'attivazione della comunicazione.

In presenza di latenza la larghezza di banda effettiva L_{eff} diviene:

$$L_{eff} = D_m/t_{tot} = D_m/(l + D_m/L) = L/(1 + L * l/D_m)$$

dove t_{tot} è il tempo totale di comunicazione, mentre D_m è la dimensione del messaggio; quindi L_{eff} tende ad L solo se le dimensioni del messaggio D_m tendono all'infinito.

3.3 Regole per parallelizzare

Per avere una buona parallelizzazione bisogna tenere conto delle seguenti regole:

- aumentare la frazione di codice parallelizzabile;

- riuscire a bilanciare in modo ottimale il lavoro tra i processori;

- minimizzare la comunicazione come quantità di dati e quantità di volte;

- ripensare radicalmente l'algoritmo per adeguarlo al calcolo parallelo.

Quando si parallelizza un codice i processi devono avere dei punti di sincronizzazione (barriere):

- un task si ferma quando arriva ad una barriera;
- quando l'ultimo task arriva alla barriera, i task sono sincronizzati;
- l'elaborazione può ripartire.

Le barriere servono per scambiare dati e informazioni e per mantenere coerente tutta la simulazione. Si possono verificare dei problemi con molti processi. Se ogni processo che arriva alla barriera manda un messaggio a tutti gli altri processi si può saturare la rete. Le implementazioni reali si progettano cercando di limitare lo scambio di messaggi.
La distribuzione del carico è importante per le performance dei programmi paralleli:

- distribuire il carico di lavoro in modo che tutti i processi siano occupati per tutto il tempo;
- minimizzare i tempi morti (*tempi di attesa*) dei processi;
- partizionare ugualmente il lavoro di ogni processo.

L'assegnazione del carico può essere fatta in modo statico:

- in genere semplice e proporzionale al volumi;
- soffre di possibili sbilanciamenti;

o in modo dinamico:

- può curare problemi di sbilanciamenti;
- introduce un overhead dovuto alla gestione bilanciamento.

La granuralità è la misura qualitativa del rapporto tra calcoli e comunicazioni. Il parallelismo può essere a grana fine:

- pochi calcoli tra le comunicazioni (*rapporto piccolo*);
- facile da bilanciare;
- overhead di comunicazioni;

o a grana grossa:

- molti calcoli tra le comunicazioni (*rapporto grande*);
- può essere difficile bilanciare il carico;
- probabili aumenti nelle performance.

Le operazioni I/O sono generalmente seriali e possono creare dei "colli di bottiglia". La gestione dell'I/O porta all'utilizzo di costrutti specifici del linguaggio e del modello di programmazione usato.
Per associare il carico ad ogni processore bisogna decomporre il problema originale in sottoproblemi oppure il dominio dei dati in sottodomini. Assegnare ad ogni processo una parte del lavoro (*sottoproblema o sottodominio*).

3.4 Tecniche di parallelizzazione

Se si vuole parallelizzare un codice bisogna capire quale parte deve essere eseguita in modo seriale da tutti i processi e quale parallelizzare, definire un meccanismo per le operazioni di Input ed Output e stabilire un metodo per la sincronizzazione e la comunicazione tra i processi coinvolti nella simulazione. In questo paragrafo sono trattate alcune tecniche basilari per la risoluzione di questi problemi.

3.4.1 Parallelizzazione cicli

In alcuni programmi, la maggior parte del tempo di CPU è consumato da una piccola parte di codice. Supponiamo di avere il seguente codice seriale:

```
for i=1 to n {
    vect a(n)
    /*Codice A*/
        ........
    /*Codice B*/
    j=0
    do
        a(j)=....
        j=j+1
    while j < n;
    /*Codice C*/
    operazioni in vect a
}
```

In questo caso la parte B del codice consuma molto tempo CPU mentre le parti A e C contengono tante linee di codice ma non consumano molto tempo CPU. In questo caso conviene parallelizzare solo la parte B del codice. Si deve fare attenzione all'array a() perchè è aggiornato in B ed è referenziato in C. Il codice parallelizzato sarà il seguente:

```
for i=1 to n {
    vect a(n)
    /*Codice A*/
        ......
    /*Codice B*/
    j=ista
    do
        a(j)=....
        j=j+1
    while j < iend;
    /*codice B'*/
    syncdata(a)
    /*Codice C*/
    operazioni in vect a
    /*fine Codice C*/
}
```

Il carico dell'iterazione è distribuito in tutti i processi. L'array a() referenziato in C è aggiornato in B' tramite *syncdata*. Da notare i valori *ista* e *iend* che indicano la porzione di array elaborata da ogni processo.
Un altro esempio è quello di programmi che usano il metodo delle differenze finite, dove ci sono alcuni do loops che contribuiscono quasi ugualmente al tempo totale di comunicazione. Il codice sotto descrive un algoritmo con le caratteristiche appena descritte:

```
do t=t1,tn
   do i=1,6
      b(i)=b(i)+a(i)
   end do
   do i=1,6
      a(i)=b(i-1)+b(i+1)
   end do
   do i=1,6
      a(i) = a(i)+1.0
   end do
end do
```

Se si sincronizzano i dati dei processi dopo ogni do loops, l'overhead di comunicazione può negare il beneficio della parallelizzazione. In questo caso si cerca di minimizzare il numero dei messaggi cercando di scambiare solo alcuni dati.

```
do t=t1,tn
   do i=ista,iend
      b(i)=b(i)+a(i)
   end do
   //scambia solo i valori adiacenti con
   //gli altri processi
   shift(b)
   do i=ista,iend
      a(i)=b(i-1)+b(i+1)
   end do
   do i=ista,iend
      a(i) = a(i)+1.0
   end do
end do
```

Le iterazioni dei do loops sono distribuite in tutti i processi. In ogni do loops un processo esegue solamente le operazioni sui vettori per valori di i compresi tra ista e iend. Ogni processo non ha bisogno di conoscere tutti i valori dei vettori a() e b(), eccetto per il secondo loop che ha bisogno dei valori adiacenti ai dati che elabora di b() per calcolare a(). Si scambiano questi dati tramite la funzione *shift*.

3.4.2 Grana grossa contro grana fine

Un programma qualche volta può avere la possibilità di essere parallelizzato in più livelli di scoping. Supponiamo di avere il seguente codice seriale:

```
proc main
     .....
     do i=1,k
        .....
        generate(a,k)
        solve(a)
        .....
     end do
end proc
```

```
proc solve(a)
     .....
     do while(NOT CONVERGED)
          .......
          sub(a,x)
          .......
     end do
endo proc

proc sub(a,x)
     .....
     do i=1,n
        y(i)=0.5*(x(i-1)+x(i+1))+a(i)
     end do
     do i=1,n
        x(i)=y(i)
     end do
     .....
end proc
```

Se si parallelizza la sub ci deve essere una comunicazione per aggiornare il valore di x in tutti i processi (parallelizzazione grana fine).

```
proc sub(a,x)
     .....
     do i=ista,iend
        y(i)=0.5*(x(i-1)+x(i+1))+a(i)
     end do
     do i=ista,iend
        x(i)=y(i)
     end do
     /*si scambia i valori di x*/
     Comunicazione(x)
     .....
end proc
```

Un'altra soluzione è quella di parallelizzare la funzione principale (grana grossa).

```
proc main
    .....
    do i=ista,iend
        .....
        generate(a,k)
        solve(a)
        .....
    end do
    /*si scambiano i valori di a*/
    comunicazione(a)
end proc
```

In generale se è possibile conviene parallelizzare con un meccanismo a grana grossa.

3.4.3 Operazioni di input

Questo paragrafo descrive delle tecniche per parallelizzare la lettura di un file (operazione di input). Se esiste un modo per avere un file condiviso allora ogni processo può leggere direttamente dal file.
Se il file è di sola lettura si può creare una copia locale per ogni processo. Si ha un miglioramento delle performance.
Un'altra tecnica è quella che un processo legge il file e lo distribuisce tramite messaggi BROADCAST agli altri processi. Un miglioramento delle performance si può avere mandando ai vari processi solo i dati che devono elaborare.

3.4.4 Operazioni di output

Le operazioni di output devono essere eseguite solo dal processo principale e i dati per l'output devono essere mandati al processo principale tramite dei messaggi.
Un altro modo è quello di avere un file condiviso dove tutti i processi possono accedere in scrittura. Il processo principale leggerà questi dati e li scriverà nello STANDARD OUTPUT.

3.4.5 Metodi di comunicazione

Nelle simulazioni parallele i processi coinvolti si scambiano dei messaggi per sincronizzarsi. Le comunicazioni possono essere collettive o point-to-point. Nella scelta delle comunicazioni da utilizzare si deve tenere conto che limitare

il numero dei messaggi diminuisce i tempi di latenza.
Il metodo di comunicazione più semplice è quello che ogni processo manda a tutti gli altri un messaggio dopo un determinato intervallo di esecuzione. Questo tipo di comunicazione ha un tempo di latenza molto alto.
Per avere un numero ridotto di messaggi si può utilizzare una delle seguenti tecniche:

- ogni processo manda i messaggi solo ai processi a cui interessano i dati del mandante;
- si identifica un processo master che ha il compito di sincronizzare il lavoro suo e degli altri processi (slave). Le comunicazioni sono solo tra il master e gli slave;
- per eseguire un aggiornamento globale si può utilizzare il meccanismo a "catena". Ogni processo manda un messaggio al processo successivo. L'ultimo processo manda un messaggio al primo (*Figura 3.3*).

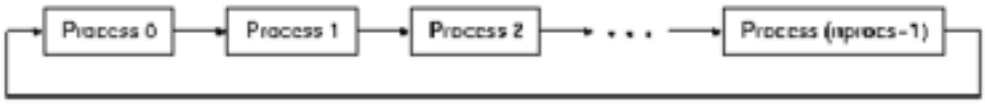

Figura 3.3: Metodo di comunicazione a catena

3.5 PDES

Parallel discrete event simulation (**PDES**) è una tecnica di simulazione distribuita derivata dal discrete event simulation (**DES**). Il **DES** è implementato per essere eseguito come unico processo. Il **PDES** attiva più processi (di solito ne assegna uno per ogni processore) che vengono eseguiti in parallelo e cooperano tra di loro tramite lo scambio di messaggi. Un fattore fondamentale per avere un buon funzionamento del PDES è quello di distribuzione. Tale distribuzione deve avere i seguenti obbiettivi:

- distribuire il carico di lavoro in maniera uniforme tra i processi coinvolti nella simulazione parallela;
- limitare il numero di messaggi di scambio tra i processi in modo che il tempo di *latenza* sia il minimo possibile.

Un altro problema del PDES è quello della sincronizzazione dei processi che serve a garantire gli stessi risultati delle sincronizzazioni seriali. In un'applicazione parallela ogni processo esegue una lista di eventi che possono essere generati localmente oppure generati da processi remoti. Siccome rispetto al tempo di simulazione i processi non sono sincronizzati tra loro, può accadere che quando un processo esegue un evento *event* un evento remoto che deve essere eseguito prima di esso non sia ancora arrivato. Se l'esecuzione in seguito dell'evento remoto non modifica il subset delle variabili di input di *event* allora l'integrità della simulazione non è stata violata altrimenti si. Per ovviare a questo problema si possono utilizzare due modelli di sincronizzazione:

1. conservativo;
2. ottimistico.

3.5.1 Distribuzione del carico

La distribuzione del carico è fondamentale per il funzionamento ottimale del **PDES**. Una distribuzione di carico ottimale comporta una migliore velocità di esecuzione.
I simulatori hanno delle strutture dati che sono elaborate dagli eventi durante la simulazione. La struttura può essere divisa in atomi che hanno i punti di confine dove un evento eseguito in quella porzione di struttura genera una schedulazione con ritardo maggiore di zero. Il carico di lavoro associato ad ogni atomo è proporzionale al tempo che gli eventi perdono per elaborare il suo stato durante la simulazione. Un altro dato da considerare è il carico di scambio tra gli atomi che è proporzionale al numero di schedulazioni che si verificano tra gli atomi durante la simulazione oppure ai ritardi di schedulazione nei punti di confine degli atomi.
Lo scopo della distribuzione è quello di dividere la struttura principale in sottostrutture, da posizionare nei processi attivi per la simulazione, ed avere un carico equilibrato tra le sottostrutture. Il carico di ogni sottostruttura è uguale alla somma dei carichi degli atomi che la compongono. Un altro fattore fondamentale è quello di minimizzare il carico di scambio tra le sottostrutture. In alcuni casi si deve trovare un compromesso tra l'equilibrio del carico e la minimizzazione delle comunicazioni.
La distribuzione del carico può essere fatta manualmente oppure utilizzando le tecniche di **clustering**.

Clustering

Il **clustering** o **analisi dei cluster** (dal termine inglese *cluster analysis* introdotto da Robert Tryon nel 1939), o analisi di raggruppamento, è un insieme di tecniche di analisi multivariata dei dati volte alla selezione e raggruppamento di elementi omogenei in un insieme di dati. Tutte le tecniche di clustering si basano sul concetto di distanza tra due elementi. Infatti la bontà delle analisi ottenute dagli algoritmi di clustering dipende essenzialmente da quanto è significativa la metrica, e quindi da come è stata definita la distanza. La distanza è un concetto fondamentale, dato che gli algoritmi di clustering raggruppano gli elementi a seconda della distanza, e quindi l'appartenenza o meno ad un insieme dipende da quanto l'elemento preso in esame è distante dall'insieme.
Le tecniche di clustering si possono basare principalmente su due "filosofie":

- dal basso verso l'alto (Bottom-Up):
  questa filosofia prevede che inizialmente tutti gli elementi siano considerati cluster a sè, e poi l'algoritmo provvede ad unire i cluster più vicini. L'algoritmo continua ad unire elementi al cluster fino ad ottenere un numero prefissato di cluster, oppure fino a che la distanza minima tra i cluster non supera un certo valore.

- dall'alto verso il basso (Top-Down):
  all'inizio tutti gli elementi sono un unico cluster, e poi l'algoritmo inizia a dividere il cluster in tanti cluster di dimensioni inferiori. Il criterio che guida la divisione è sempre quello di cercare di ottenere elementi omogenei. L'algoritmo procede fino a che non ha raggiunto un numero prefissato di cluster. Questo approccio è anche detto "gerarchico".

Le tecniche di clustering vengono utilizzate generalmente quando si hanno tanti dati eterogenei, e si è alla ricerca di elementi anomali.
Queste tecniche possono essere utilizzate per dividere il carico tra vari processi per l'esecuzione di algoritmi paralleli.
Esistono varie classificazioni delle tecniche di clustering comunemente utilizzate. Una prima categorizzazione dipende dalla possibilità che ogni elemento possa o meno essere assegnato a più clusters:

- clustering esclusivo, in cui ogni elemento può essere assegnato ad esattamente un solo gruppo. I clusters risultanti, quindi, non possono avere elementi in comune. Questo approccio è detto anche **Hard Clustering**;

- clustering non-esclusivo, in cui un elemento può appartenere a più cluster con gradi di appartenenza diversi. Questo approccio è noto anche con il nome di **Soft Clustering**.

Un'altra suddivisione delle tecniche di clustering tiene conto della tipologia dell'algoritmo utilizzato per dividere lo spazio:

- Clustering Partitivo (detto anche k-clustering), in cui per definire l'appartenenza ad un gruppo viene utilizzata una distanza ed un punto rappresentativo del cluster (centroide, medioide ecc...).

- Clustering Gerarchico, in cui viene creata una visione gerarchica dei cluster, visualizzando in un trellis i passi di accorpamento/divisione dei gruppi. Le tecniche di clustering gerarchico non producono un partizionamento flat dei punti, ma una rappresentazione gerarchica ad albero(*Figura 3.4*).
  Questi algoritmi sono a loro volta suddivisi in due classi:

  - *Agglomerativo*: Questi algoritmi assumono che inizialmente ogni cluster (foglia) contenga un singolo punto; ad ogni passo, poi, vengono fusi i cluster più "vicini" fino ad ottenere un singolo grande cluster. Questi algoritmi necessitano di misure per valutare la similarità tra clusters, per scegliere la coppia di cluster da fondere ad ogni passo;
  - *Divisivo*: Questi algoritmi, invece, partono considerando lo spazio organizzato in un singolo grande cluster contenente tutti i punti, e via via lo dividono in due. Ad ogni passo viene selezionato un cluster in base ad una misura, ed esso viene suddiviso in due cluster più piccoli. Normalmente viene fissato un numero minimo di punti sotto il quale il cluster non viene ulteriormente suddiviso (nel caso estremo questo valore è 1). Questi tipi di algoritmi necessitano di definire una funzione per scegliere il cluster da suddividere;

- Clustering density-based, in cui il raggruppamento avviene analizzando l'intorno di ogni punto dello spazio. In particolare, viene considerata la densità di punti in un intorno di raggio fissato.

Un esempio di algoritmo di clustering è il **QT** (*Quality Threshold*) Clustering (Heyer et al., 1999). Questo metodo è stato inventato per il clustering dei geni e restituisce sempre lo stesso risultato quando si ripete diverse volte.
L'algoritmo è il seguente:

- l'utente sceglie un diametro massimo per i clusters;

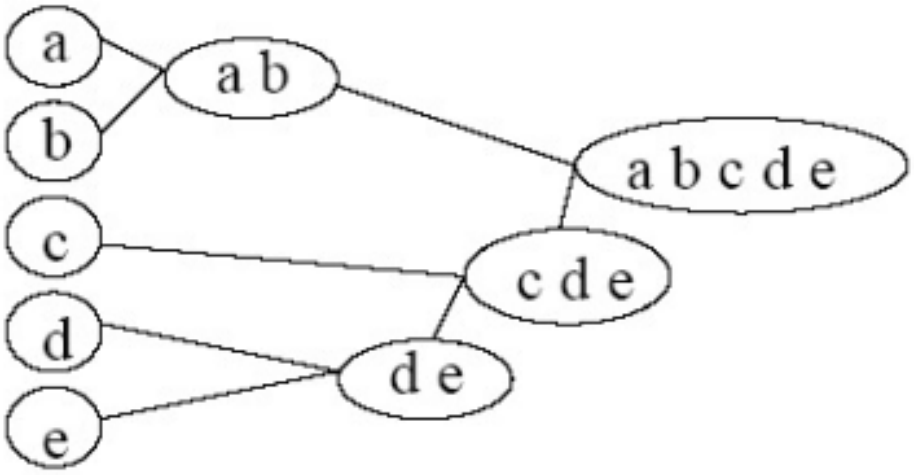

Figura 3.4: Clustering gerarchico

- costruzione di un cluster candidato per ogni punto, includendo il punto più vicino, il prossimo più vicino, e cosi via, fino a che il diametro del cluster non supera la soglia;
- salvataggio del cluster candidato con la maggior parte dei punti come primo vero cluster, e rimozione di tutti i punti nel cluster da ulteriori considerazioni;
- ricorsione col ridotto insieme di cluster.

Applicazione di QT al partizionamento dei grafi

Il **QT** può essere una soluzione al partizionamento dei grafi su più cluster. Se si vogliono utilizzare le tecniche **PDES** in simulatori di rete come **Ns** e **Pdnet** (descritti nei capitoli successivi) ci troviamo di fronte a strutture dati di tipo grafo. L'atomo strutturale del grafo è il nodo. Gli archi sono i punti di confine tra gli atomi. La divisione del grafo in sottografi è l'obiettivo di questa distribuzione.
Per prima cosa si deve calcolare il carico di lavoro medio associato ai singoli nodi e il carico di scambio tra i nodi (distanza). Per la creazione di sottografi si parte dal primo cluster a cui si assegna un nodo. In base ad una strategia di scelta si assegnano altri nodi al primo cluster , uno per volta, fin quando non si supera una soglia che indica il carico massimo da assegnare ad un sottografo. Questa operazione viene fatta ricorsivamente per tutti i cluster.

3.5.2 Modello di sincronizzazione conservativo

La sincronizzazione conservativa è conservativa nel senso che ogni LP (Local Process) esegue un evento solo se è sicuro che non vi siano eventi esterni (eventi eseguiti dagli altri LP) che influenzano il suo stato di input.
Tra gli LP si hanno i canali di comunicazione, che non variano durante la simulazione, in cui i messaggi sono ricevuti nell'ordine in cui sono stati mandati. In ogni simulazione conservativa ogni LP esegue ciclicamente questi due passi:

- LP riceve un messaggio da un canale di input;
- esegue gli eventi fino a che è sicuro che un messaggio relativo ad un evento esterno non modifichi l'input dell'evento.

Un problema che affligge questo modello di simulazione è il **deadlock**. Supponiamo ci siano tre LP A, B, C che comunicano tra di loro in questo modo: A->B->C->A. Nel corso della simulazione può capitare che in un intervallo di sincronizzazione gli LP non hanno messaggi remoti da mandare. Nel nostro caso ci troviamo in una situazione in cui B attende un messaggio da A, C attende un messaggio da B e A attende un messaggio da C. Tutti gli LP rimangono bloccati in attesa di un messaggio (stallo). Per risolvere questo problema gli LP mandano dei **NULL MESSAGE** che servono ad avanzare il tempo di "libertà di esecuzione" degli altri LP.
Per capire il funzionamento di questi metodi viene illustrato il modello di Chandy-Misra che utilizza un meccanismo di sincronizzazione con **NULLMESSAGE** mandati al limite del bloccaggio. Le seguenti definizioni definiscono le componenti di questo modello:

- Un **LP** (Logical Process) o **federato** è definito come un set che contiene componenti base mappato in modo che è conforme all'architettura parallela e distribuita.
- **LVT** (Local Virtual Time): tempo associato con l'LP. Ogni LP non conosce gli LVT degli altri LP a meno che non comunichino tra di loro con dei messaggi;
- **FEL** (Future Event List): la lista degli eventi interni di un LP;
- **Message**: un meccanismo per mandare e ricevere messaggi e associarli ai relativi eventi remoti;
- **Eventi** : gli eventi possono generare cambiamenti di stato del modello, generare altri eventi locali o mandare messaggi agli altri federati per la schedulazione di un evento remoto;

- **Lookahead**: il tempo più piccolo che deve passare da una schedulazione remota all'esecuzione di questo evento nell'LP remoto. Se ad esempio il primo evento eseguibile in un LP si ha al tempo T, questo evento non può schedulare un evento in un LP remoto con un tempo di esecuzione $<$ T+lookahead;

- **EIT(r)**(Earliest Input Time): vettore che indica i tempi entro cui un LP deve ricevere un messaggio dagli LP remoti; il minimo valore di questo vettore ci indica il tempo entro cui un LP deve ricevere un messaggio (**eit**);

- **EOT(r)** (Earliest Output Time) : vettore che indica i tempi entro cui un LP deve mandare un messaggio in remoto;

- **ECOT(r)** (Earliest Conditional Output Time): vettore che indica i tempi entro cui un LP deve mandare un messaggio in remoto assumendo che non riceva nessun messaggio nell'intervallo.

Prima di iniziare l'esecuzione della simulazione il sistema effettua il clustering e crea i canali di comunicazione tra gli LP (max uno per ogni coppia di LP sorgente destinazione).
Dopo l'inizializzazione generale del sistema in ogni LP è definito un sottosistema che ha bisogno di comunicare con gli altri sottosistemi assegnati agli altri LP. Per le comunicazioni ogni LP si crea le seguenti strutture:

- il vettore *look* che rappresenta i valori di lookahead verso gli altri LP. Se L'LP locale è i il valore $look(j)$ rappresenta il lookahead $i- > j$. Il calcolo dei lookahead dipende dal minimo ritardo delle schedulazioni tra gli LP;

- il vettore *eot* formato da elementi di questo tipo: $eot(j) = TIME\_START+ look(j)$ (rappresenta il tempo entro cui l'LP locale deve mandare necessariamente un messaggio all'LP j);

- il vettore *Eit* che rappresenta il vettore dei tempi di ricezione entro cui L'LP locale deve ricevere un messaggio dagli altri LP;

- la variabile *eit*, calcolata come il minimo degli elementi del vettore *Eit*, che rappresenta il punto di bloccaggio dell'LP;

L'esecuzione del modello esegue i seguenti passi:

1. ogni LP esegue gli eventi finchè hanno un tempo minore (minore o uguale) di *eit*. Ogni evento può generare dei messaggi remoti da mandare agli altri LP che possono essere gestiti in due modi:

- li manda al momento in cui vengono generati;
- li memorizza nella coda del canale a cui appartengono e li manda tutti insieme al tempo *eot* assegnato al canale;

Nel primo caso si genera un numero di messaggi maggiore e l'aggiornamento del tempo di bloccaggio è più frequente mentre nel secondo caso si avranno messaggi più lunghi. Il secondo di solito è più veloce. Per avere il pieno controllo dei messaggi da mandare ogni LP schedula un evento particolare (evento "NULL Message") per ogni LP remoto al tempo *eot* corrispondente. Un evento del genere quando viene eseguito si comporta nel seguente modo:

- calcola il nuovo $eot(j)$ in questo modo:

$$eot(j) = min(first\_event\_time, eit) + look(j)$$

  dove *first_ event_ time* rappresenta il tempo del primo evento "non NULL Message" e $look(j)$ il lookahead riferito all'LP a cui mandiamo il messaggio;
- manda il valore di eot più eventuali messaggi all'LP assegnato con l'evento;
- si rischedula al tempo $eot(j)$.

2. se $eit > first\_event\_time$ l'LP non può eseguire un evento locale e si mette in ricezione di un messaggio. Quando lo riceve setta il campo del vettore *Eit* corrispondente al processo mandante, si ricalcola il valore di *eit* e ritorna al passo precedente.

Una variante di questo algoritmo non utilizza i valori di *eot* per mandare i NULL MESSAGE bensì i valori di Ecot ($Ecot = first\_event\_time + look(j)$). Questo algoritmo suppone che non ci siano messaggi ricevuti nell'intervallo. Questo tipo di simulazione è però un modello ottimistico e devono essere utilizzate delle tecniche particolari per ripristinare il sistema in caso di errore.
In *Figura 3.5* è rappresentato il calcolo di *Eot*, *Ecot* in un modello di simulazione supponendo che il lookahead è uguale a 1.

3.5.3 Modello di sincronizzazione ottimistico

Nel modello di sincronizzazione ottimistico ogni LP esegue gli eventi in maniera aggressiva senza considerare se ci sono dei vincoli rispetto ad eventi eseguiti negli altri LP. Il *Time Warp* di M.Fujimoto è un modello ottimistico

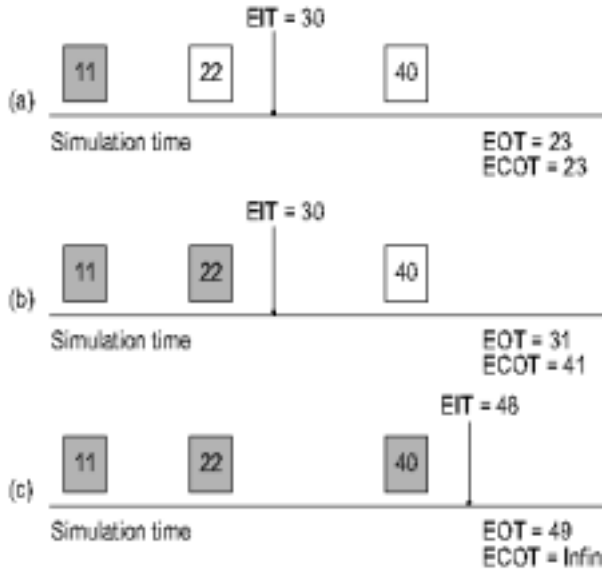

Figura 3.5: Calcolo Eot ed Ecot in un modello di simulazione parallelo

in cui la ricezione di un messaggio da parte di un LP è gestita in questo modo:

- se l'evento remoto ha un tempo di esecuzione maggiore dell' LVT (Local Virtual Time) lo inserisce semplicemente nella FEL (Future Event List);
- se il tempo di esecuzione dell'evento remoto ricevuto è minore dell' LVT utilizza dei metodi di **RollBack** per ripristinare lo stato del sistema al tempo dell'evento ricevuto.

Per implementare un meccanismo di RollBack si deve tenere traccia dei vari stati del sistema ai tempi minori dell'LVT e utilizzare un meccanismo di anti-message. Gli anti-message servono ad annullare eventuali messaggi mandati nell'intervallo di tempo tra il processo con tempo di esecuzione ricevuto da remoto e il tempo LVT ed hanno le seguenti proprietà:

- servono a cancellare i messaggi;
- ogni messaggio spedito da un LP ha un potenziale anti-message;
- l'anti-message ha un contenuto identico ad un messaggio ad eccezione di un bit di segno;

- quando un anti-message e il rispettivo messaggio si incontrano nella stessa coda avviene un annichilimento di entrambi;
- per "rimediare gli effetti" dei messaggi precedentemente spediti un LP manda un anti-message;
- la spedizione di un messaggio comporta anche una copia del messaggio con segno negativo mantenuta nella coda di output dell'LP (coda degli anti-message che potrebbero servire durante un eventuale rollback).

Durante l'esecuzione un LP che riceve un anti-message si può trovare nelle seguenti situazioni:

- l'evento corrispondente non è ancora stato elaborato. In questo caso effettua un annichilimento della coppia messaggio anti-message;
- il messaggio corrispondente è già stato elaborato. In questo caso effettua il rollback al tempo precedente all'elaborazione del messaggio e l'annichilimento messaggio anti-message;
- il messaggio corrispondente non è ancora stato ricevuto. In questo caso l'anti-message viene mantenuto in coda e l'annichilimento avverrà in futuro quando si sarà formata la coppia messaggio anti-message.

Per effettuare il rollback si propongono tre stategie:

- **cancellazione aggressiva** : assume che tutte le computazioni effettuate siano sbagliate con conseguente spedizione degli anti-message relativi e ripristino allo stato di ritorno;
- **cancellazione lazy**: gli anti-message non vengono mandati subito ma si attende la riesecuzione degli eventi ripristinati. Se un messaggio uguale è stato ricreato dalla nuova esecuzione non si manda un anti-message;
- **Lazy revalutation**: simile a **lazy cancellation** ma in più viene collezionato un vettore di stati. Si consideri il caso in cui arriva un messaggio che viene processato. Subito dopo arriva un anti-message e si deve rieseguire il rollback. Però tramite il vettore di stati ci si accorge che la riesecuzione degli eventi ci porta di nuovo nello stesso stato. In questo caso si può evitare il rollback. Il problema sta nel fatto di poter capire tramite il vettore degli stati se la riesecuzione ci porta di nuovo allo stesso stato.

Per supportare il rollback è necessario salvare la storia del processo logico. Per ridurre l'utilizzo della memoria si può utilizzare il **memory management** che è basato sul concetto di GVT (Global Virtual Time) che fornisce un lower bound ai rollback che possono avvenire in futuro. Il GVT ha le seguenti caratteristiche:

- il time stamp minimo di ogni messaggio o anti-message, parzialmente o totalmente processato, all'interno del sistema in un dato momento. Il GVT fornisce un lower bound al time stamp dei rollback futuri;
- lo spazio di memoria per eventi e salvataggi di stato, con tempo precedente al GVT, possono essere deallocati;
- le operazioni di I/O con time stamp minori di GVT possono essere eseguite.

Il GVT ci fornisce uno stato del sistema dal quale si può evolvere.
In una applicazione distribuita il calcolo di GVT può essere difficoltoso perchè è difficile reperire tutti i messaggi in transito.
In Git/BellCore tutti i processi sono congelati prima che le GVT computino. Utilizzando un meccanismo di basso livello di comunicazione, tutti i messaggi transienti garantiscono di arrivare a destinazione prima che la computazione GVT inizia. Questo tipo di computazione lavora come segue:

- un coordinatore inizia la procedura congelando l'esecuzione di ogni processo logico;
- dopo che tutti i messaggi transienti arrivano a destinazione, ogni processo logico riporta un minimo valore locale al coordinatore;
- il coordinatore calcola il GVT come il valore minimo dei valori locali ricevuti;
- il coordinatore manda in broadcast il valore GVT agli altri processi logici.

3.5.4 Considerazioni sui modelli conservativi e ottimistici

Per implementare le tecniche Pdes si devono fare delle considerazioni per capire quale modello utilizzare e come utilizzarlo.
Alcuni software, come possono essere i simulatori di rete, hanno dei moduli che permettono di fare delle varie configurazioni della struttura. Il programmatore tramite degli script o interfacce grafiche interagisce con questi moduli

e costruisce le strutture. In seguito indica quali sono gli eventi iniziali e fa partire la simulazione.
Prima di iniziare una schedulazione parallela si deve dividere la struttura del modello nei vari processi. Questa divisione può essere fatta manualmente. In questo caso il programmatore deve fare del lavoro in più rispetto ad una simulazione seriale e la responsabilità di eventuali squilibri tra i vari cluster dipende dalla sua analisi.
Un altro modo per dividere il carico è tramite le tecniche di clustering. In questo caso spetta allo schedulatore capire dove andare a posizionare le varie componenti. Per calcolare i carichi delle componenti si possono utilizzare dei metodi deterministici, utilizzati in sistemi in cui il carico può essere calcolato in maniera "certa" oppure metodi probabilistici che fanno delle stime sul carico. Se si è certi che il carico durante la simulazione rimane costante è vivamente consigliato utilizzare queste tecniche. Se esiste una variazione del carico delle varie componenti si deve essere certi che non sia così eclatante da squilibare in modo consistente i carichi dei vari cluster. Per ovviare a questo problema si può calcolare il valore dei carichi durante la simulazione e implementare l'algoritmo in modo che capisca quando lo squilibrio di carico è eccessivo e in tal caso rifare la clusterizzazione. Per implementare una situazione del genere bisogna stabilire una soglia di squilibrio dei carichi tra i cluster che quando viene superata riclusterizza tutto. In alcuni sistemi questi cambi di carico possono essere repentini e quindi si possono avere riclusterizzazioni frequenti aumentando il tempo reale di simulazione in maniera significativa e non rendendo più conveniente l'utilizzo di tecniche parallele.
Per la sincronizzazione si possono utilizzare le tecniche conservative o ottimistiche. Nei modelli conservativi ci sono dei punti di bloccaggio e lo scopo e quello di far "bloccare" il meno possibile la simulazione. Questo può avvenire creando dei punti di confine tra i cluster con lookahead alto in modo che ogni processo ha più "spazio" di esecuzione e cercando di equilibrare il più possibile il carico. Con uno squilibrio elevato dei carichi c'è il rischio che alcuni processi restino in attesa per molto tempo. Esistono vari esperimenti su simulatori di rete che utilizzano questo metodo come **Pdns** (modello applicato ad ns) e **Omninet**. Il clustering in questi due casi è manuale.
Per evitare i punti di bloccaggio possiamo utilizzare i modelli ottimistici. Per utilizzare questi modelli dobbiamo porci le seguenti domande:

- quanta memoria ci vuole per salvare uno stato del sistema?
- quanto tempo ci vuole per ripristinarlo?

Alcuni modelli hanno un numero di variabili di stato elevato, salvare i vari stati occuperebbe molta memoria e il ripristino troppo tempo. In modelli in cui lo stato ha poche variabili e il ripristino è banale si consiglia questo metodo.

3.5.5 Modelli Ibridi

Gli approci ibridi possono essere sviluppati aggiungendo delle tecniche ottimistiche ai meccanismi di sincronizzazione conservativa.
Lubachecky, Swartz e Maiss proposero un estenzione dell'algoritmo *bounded lag* chiamato *filtered rollback*. L'algoritmo *bounded lag* usa il concetto di distanza minima tra i processi logici come base per decidere quali eventi possono essere eseguiti in maniera certa. In *filtered rollback* il simulatore può violare questi lower bound. Se si creano degli errori viene utilizzato un meccanismo di rollback come Time-Warp. Tramite l'adattamento dei valori delle distanze si possono utilizzare questi approci ottimistici all'interno di intervalli conservativi.
Dickens e Reynols hanno proposto un'estenzione dell'algoritmo conservativo SRADS dove viene permesso un "local rollback". Questo approcio è comunque applicabile ad ogni schema conservativo. In questo approcio il protocollo conservativo utilizzato da ogni processo ci definisce gli eventi sicuri che possono essere eseguiti. Quando un processo non ha più eventi sicuri da eseguire processa ottimisticamente altri eventi pendenti. Il risultato dell'esecuzione degli eventi pendenti non può essere mandato agli altri processi. Questo ci permette di non avere bisogno del meccanismo degli anti-message. Il rollback utilizzato è "aggressivo". Utilizzando questo approcio il tempo di attesa dei processi viene utilizzato per eseguire eventi in maniera ottimistica.

Capitolo 4

NS

4.1 Le basi di Ns

Ns è un simulatore di reti ad eventi scritto interamente in C++, con l'ausilio di un interprete OTcl(*Figura 4.1*). Ns usa due linguaggi di programmazione

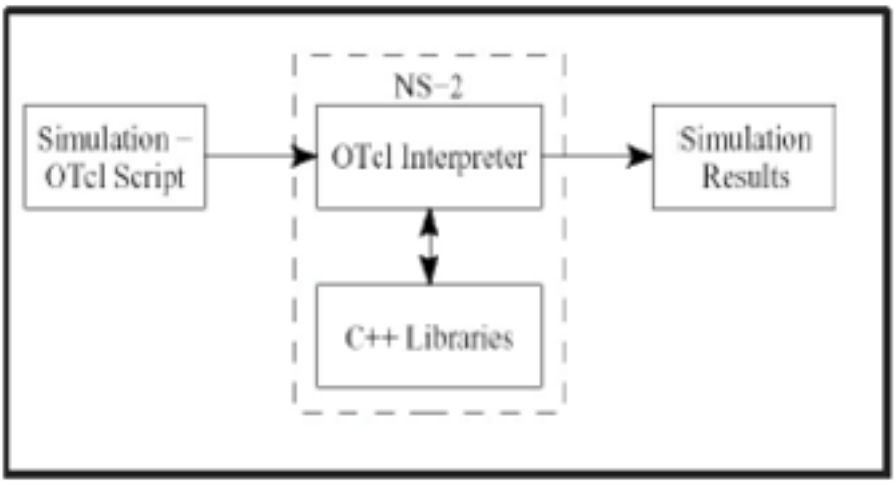

Figura 4.1: Architettura di ns

poichè il simulatore ha due necessità principali; infatti, da una parte dettagliate simulazioni di protocolli richiedono un linguaggio di programmazione che possa efficientemente manipolare bytes, headers dei pacchetti, ed implementare algoritmi che utilizzino un gran numero di dati. Per queste attività la velocità a tempo di esecuzione(*run-time*) è importante, mentre il tempo impiegato per altre attività come l'esecuzione della simulazione, trovare degli

errori, correggerli, ricompilare e rieseguire di nuovo il tutto, è di gran lunga meno importante. Dall'altro lato invece gran parte dello studio delle reti si evolve variando parametri o cambiando alcune configurazioni, o esplorando rapidamente un certo numero di scenari. In questi casi il tempo di iterazione (*cambiare il modello e rieseguire il tutto*) è più importante.
Ns utilizza il C++ basato sulla gerarchia di classi cosidetta **compilata** e Otcl basato sulla gerarchia cosidetta **interpretata**. Il C++ è rapido da eseguire ma più lento da modificare rendendolo adatto per dettagliate implementazioni di protocolli. OTcl invece è lento da eseguire ma può essere modificato molto rapidamente.
Le due gerarchie, compilata ed interpretata, sono strettamente correlate l'una all'altra; infatti dal punto di vista dell'utente, c'è una corrispondenza 1:1 tra una classe nella gerarchia interpretata e una classe in quella compilata. La radice di questa gerarchia è la classe **TclObject**. L'utente può creare nuovi oggetti del simulatore attraverso l'interprete, all'interno del quale vengono istanziati e dal quale vengono associati immediamente ad un oggetto corrispondente nella gerarchia compilata. La gerarchia di classi interpretata è automaticamnte stabilita attraverso i metodi definiti nella classe **TclClass**, mentre gli oggetti istanziati dall'utente, vengono associati all'oggetto della gerarchia compilata attraverso i metodi definiti nella classe **TclObject**.

4.2 La classe Simulator

Il simulatore, nel suo complesso, è descritto dalla classe Tcl denominata *Simulator*: tale classe fornisce un insieme di interfacce sia per configurare una simulazione sia per scegliere il tipo di **schedulatore di eventi** da usare per guidare la simulazione.
Uno script di simulazione generalmente inizia con una istruzione di creazione di una istanza della classe *Simulator* del tipo *set ns [new Simulator]*, cui fanno seguito una serie di istruzioni di chiamata a vari metodi per la creazione di topologie di rete più o meno complicate e per la configurazione di tutta una serie di aspetti della simulazione.
Quando viene creato, in OTcl, un nuovo oggetto di simulazione, la procedura di inizializzazione esegue le seguenti operazioni:

1. inizializza il formato dei pacchetti da usare per la simulazione;

2. crea uno schedulatore: è possibile scegliere diversi tipi di schedulatore, oppure accettare direttamente quello di default (*calendar scheduler*);

3. crea opzionalmente un cosiddetto **"null agent"**, ossia una sorta di "cestino" per i pacchetti simulati.

4.3 La gestione dei pacchetti

I pacchetti sono le entità che gli oggetti di Ns si scambiamo durante la simulazione. La gestione di questi oggetti comprende la definizione degli header che contengono le informazioni dei protocolli da implementare e un meccanismo per una loro veloce allocazione e deallocazione. In questo paragrafo sono descritte le singole componenti che servono per la gestione dei pacchetti in ns.

Header dei pacchetti

Gli header dei pacchetti sono le strutture dati che contengono le informazioni di un protocollo. Per la creazione di un nuovo header, chiamato ad esempio *"newhdr"*, si devono seguire i seguenti passi:

- creare una nuova struttura (chiamata *"hdr_ newhdr"*); questa definizione di struttura viene usata dal compilatore per calcolare l'offset (espresso in byte) dei vari campi; nessun oggetto di questo tipo di struttura viene mai direttamente allocato.
- creare una classe statica per effettuare il linkage tra C++ e OTcl, in modo che in OTcl sia disponibile la classe denominata *PacketHeader/Newhdr*;
- editare il file *ns/tcl/lib/ns-packet.tcl* al fine di abilitare il nuovo formato di header.

La struttura *"hdr_ newhdr"* definisce i campi contenuti nell'header e la loro dimensione. Tale struttura deve inoltre fornire per ogni campo una funzione membro che rappresenta una sorta di strato di "copertura di dati" (*data hiding*) per quegli oggetti che desiderano leggere o modificare i campi dell'header. Si tratta cioè di *"funzioni membro pubbliche di accesso ai dati"* della struttura, che consentono operazioni sui dati nascondendo, agli oggetti che richiedono tali operazioni, la struttura interna. Deve essere inoltre definita una variabile membro statica, chiamata *offset_* , che viene usata per individuare l'offset (sempre in byte) con il quale l'header è localizzato in un generico pacchetto di ns. Per consentire l'accesso all'header del pacchetto devono essere implementate due funzioni membro statiche:

- la funzione *access()* è quella che la maggior parte degli utenti dovrebbero scegliere per accedere a questo particolare header in ciascun pacchetto;

- la funzione *offset()*, invece, viene usata dal PacketHeaderManager e dovrebbe essere usata più raramente.

Per accedere all'header di un pacchetto puntato dalla variabile p, è sufficiente usare l'istruzione *hdr_ newhdr::access(p)*.

La classe Packet

La classe **Packet** definisce la struttura di un pacchetto e fornisce le funzioni membro per manipolare una lista libera di oggetti di questo tipo. La classe **Packet**, definita nel file packet.h, è derivata dalla classe base Event. Essa contiene in primo luogo tre variabili private e una funzione di tipo friend. Tali tre variabili private sono due puntatori (*bits_* e *data_*) ed un intero *(datalen_)*. Segue una variabile statica protetta (il puntatore *free_* ad un altro oggetto di classe *Packet*) e poi un certo numero di variabili e funzioni membro pubbliche. Questa classe fornisce un puntatore (*bits_*) ad un generico vettore di caratteri non segnati (unsigned characters), comunemente chiamato **BOB** (che sta per *bag of bits*): in questo vettore sono memorizzati i campi degli header di pacchetto. La variabile *bits_* contiene appunto l'indirizzo del primo byte del vettore BOB. Tale vettore è concretamente implementato come una concatenazione di tutte le strutture definite per ciascun header di pacchetto. Per convenzione, tali strutture hanno nomi che cominciano con *hdr_* <*qualcosa*>. Generalmente, il vettore BOB rimane di dimensione fissa durante una simulazione e la sua dimensione è registrata nella variabile membro statica *Packet::hdrlen_*. La dimensione del vettore BOB viene impostata, direttamente tramite OTcl, quando viene configurata la simulazione e, come detto, rimane generalmente invariata durante la simulazione, anche se è previsto che la si possa variare dinamicamente. Gli altri metodi della classe *Packet* servono per creare nuovi pacchetti e per memorizzare quelli non più utilizzati in una lista libera privata. Le funzioni di Packet sono le seguenti:

- *alloc()*: si tratta di una funzione di supporto comunemente usata per la creazione di nuovi pacchetti. Questa funzione viene a sua volta tipicamente chiamata, in favore di un dato Agent, dal metodo *Agent::allocpkt()*, mentre invece generalmente non viene invocata direttamente da altri oggetti. Essa tenta per prima cosa di allocare un vecchio pacchetto

della lista libera e, se tale operazione non riesce (il che accade quando l'indirizzo contenuto nella variabile *free_* è nullo), alloca un nuovo pacchetto usando l'operatore C++ denominato *new*.
E' importante sottolineare che gli oggetti della classe *Packet* e il vettore BOB vengono allocati separatamente.

- *free()*: serve invece a "liberare" un pacchetto, restituendolo alla lista libera. I pacchetti non vengono mai restituiti all'allocatore della memoria di sistema: al contrario, essi sono memorizzati in una lista libera quando la funzione *free()* viene invocata.

- *copy()*: crea una nuova e identica copia di un dato pacchetto, con l'eccezione del campo denominato *uid_*, che invece deve essere unico per ciascun pacchetto.

La Classe p_info

La classe **p_info** è usata come "colla" per collegare i valori numerici corrispondenti ai vari tipi di pacchetti con i loro relativi nomi simbolici. Quando un nuovo tipo di pacchetto viene definito, il suo codice numerico deve essere aggiunto alla variabile "packet_t" ed il suo nome simbolico deve essere aggiunto al costruttore della classe p_Info:

```
enum packet_t
{
        PT_TCP,
        ...
        PT_NTYPE // This MUST be the LAST one
};

class p_info
{
    public:
        p_info()
        {
                name_[PT_TCP]= "tcp";
                ...
        }

}
```

Come si vede, la variabile *packet_t* è un enumerato, il cui ultimo elemento deve sempre essere costituito dalla variabile PT_NTYPE.
La classe "p_info", invece, presenta un omonimo costruttore che esegue il predetto collegamento tra codici numerici e nomi simbolici dei pacchetti.

La classe PacketHeader

La classe *PacketHeader* costituisce la classe base con cui configurare un qualsiasi header di pacchetto da usare nelle simulazioni. Essa essenzialmente fornisce uno stato interno sufficiente per allocare ogni particolare header di pacchetto nell'insieme di tutti gli header presenti in un dato pacchetto.

La classe PacketHeaderManager

La classe **PacketHeaderMenager** è usata per gestire l'insieme dei tipi di pacchetti correntemente utilizzati dalla simulazione e per assegnare a ciascuno di essi un offset univoco all'interno del vettore BOB.
Quando il simulatore si inizializza fa il mapping tra i nomi delle classi e i *PacketHeader* corrispondenti e chiama una funzione Tcl della classe simulator (*create::packetformat*) che crea un singolo oggetto della classe *PacketHeaderManager*. Il suo costruttore collega la variabile OTcl denominata *hdrlen_* (indica la lunghezza del BOB) alla variabile membro C++ *hdrlen_* della classe *Packet*. Questa operazione ha l'effetto di settare la suddetta variabile *Packet::hdrlen_* al valore zero.
Dopo aver creato il gestore dei pacchetti (packet manager) il simulatore abilita ciascuno degli header di pacchetto di interesse attraverso un ciclo che itera una lista di predefiniti header di pacchetto, nella forma (h_i,o_i), dove h_i è il nome dell'i-simo header, mentre o_i è il nome della variabile contenente la posizione dell'header h_i nel vettore BOB.
Il posizionamento degli header è effettuato dalla procedura *allochdr()* della classe *PacketHeaderManager*. Questa procedura mantiene una variabile dinamica *hdrlen_* al valore dell'attuale lunghezza del vettore BOB, in modo che sempre nuovi header di pacchetto possano essere abilitati.

4.4 Lo Schedulatore di eventi

Ns è un simulatore guidato ad eventi (*event-driven*). Attualmente, sono previsti quattro diversi schedulatori ognuno dei quali è implementato usando una differente struttura dati come coda di priorità:

- *linked-list scheduler* utilizza una lista linkata;

- *heap scheduler* utilizza un heap;
- *calendar queue scheduler (lo schedulatore usato per default)* utilizza una struttura dati tipo un calendario da tavolo;
- *real time scheduler* tenta di sincronizzare gli eventi in tempo reale.

Lo schedulatore, quale sia la sua topologia, funziona nel modo seguente:

- sceglie l'evento successivo da eseguire: se la simulazione non è ancora partita, si tratterà dell'evento di "*start*" della simulazione, invece a simulazione già in corso, si tratterà dell'evento successivo all'ultimo che è stato eseguito ed in fase di complementamento;
- esegue l'evento successivo fino al suo completamento;
- ritorna ad individuare l'evento successivo e così via, fino all'evento finale della simulazione (evento di "*stop*").

L'unità di tempo utilizzata dallo schedulatore sono i secondi.
Il simulatore è progettato in modo che un solo evento possa essere eseguito in un dato istante. Ciò significa che, se fossero previsti due eventi per lo stesso istante, il simulatore eseguirà comunque prima uno e poi l'altro: in particolare, viene eseguito per primo l'evento che è stato schedulato per primo. A tal proposito, si tenga conto che gli eventuali *eventi simultanei* non vengono riordinati in alcun modo dagli schedulatori, proprio in modo tale che gli schedulatori rispettino, nell'esecuzione degli eventi, lo stesso ordine con cui tali eventi sono stati schedulati dall'utente. Inoltre si deve segnalare che non sono supportate le *esecuzioni parziali* degli eventi.
Un generico evento comprende generalmente un firing time e una *funzione manipolatrice* (detta "*handler*") ed è definito dalla classe Event.

```
class Event {
public:
        Event* next_;
        Event* prev_;
        Handler* handler_;
        double time_;
        scheduler_uid_t uid_;
        Event() : time_(0), uid_(0) {}
};
```

```
class Handler {
 public:

        virtual void handle(Event* event) = 0;
};
```

Nella definizione della classe Event sono incluse cinque variabili pubbliche ed un costruttore. Le cinque variabili hanno il seguente significato:

- "*next_*" è un puntatore al successivo evento nelle liste usate dallo schedulatore;
- "*prev_*" è un puntatore al precedente evento nelle liste usate dallo schedulatore;
- "*handler_*" è un puntatore ad un oggetto di classe **Handler**, ossia ad un manipolatore di eventi, da chiamare quanto l'evento in questione deve essere eseguito;
- "*time_*" è l'istante di tempo in cui l'evento deve essere eseguito;
- "*uid_*" è un identificatore univoco dell'evento.

La classe denominata **Handler** contiene semplicemente una *funzione virtuale* che dovrà essere specializzata tramite le classi da essa derivate.
Tornando invece alla classe **Event**, da essa vengono derivati due tipi di oggetti fondamentali per ns:

- i pacchetti (classe *Packet*);
- le procedure **at-event**: si tratta di procedure Tcl che devono essere eseguite nel momento in cui arriva un determinato istante (specificato dall'utente). Questo strumento viene usato molto spesso negli script di simulazione.

4.5 Gli Ns-Object

La classe **NsObject** è una classe derivata dalla classe **Handler** e dalla classe **TclObject**. Questa classe è la classe base per gli oggetti del simulatore che vengono utilizzati dalle librerie Tcl di ns per formare degli oggetti più complessi (Nodi, SimplexLink etc) che servono per costruire le topologie di rete. Gli oggetti derivati dalla classe **NsObject** possono essere anche utilizzati come *Handler_* della classe **Event** per la schedulazione di eventi.

Questi oggetti implementano delle funzioni che gestiscono i pacchetti e possono anche bypassarli ad altri **NsObject**. In poche parole gli oggetti più complessi creati dalle librerie Tcl sono formate dalla concatenazione di questi oggetti.
Le classi che derivano da **NsObject** devono implementare la classe virtule *void recv(Packet*, Handler* callback = 0)*, definita come classe virtuale in NsObject, che serve per la gestione di un pacchetto quando viene ricevuto da un'oggetto di questo tipo. Nella classe NsObject viene implementata la funzione *handle* (funzione virtuale di Handler) che viene utilizzata dagli schedulatori per eseguire un evento. Questa funzione esegue la funzione *recv* con *Handler* 0. Alcuni NsObject possono voler fare qualche operazione diversa all'atto dell'esecuzione. In questo caso basta rimplementare la funzione *handle()* nella classe derivata da NsObject.

4.5.1 Connector

La classe **Connector** viene utilizzata come classe base di **NsObiect** unidirezionali. Questa classe contiene due istanze ad altri **NsObject**: *target_* e *drop_*. L'istanza *target_* è utilizzata nella funzione *send()* per passare il pacchetto all'oggetto puntato tramite la chiamata alla funzione *recv()* di quest'ultimo. Comunque le classi che derivano da **Connector** possono implementare la *recv()* a propria discrezione.
L'istanza ad un oggetto *drop_* è utilizzata per la simulazione della perdita dei pacchetti. Le funzioni *drop()* vengono utilizzate per il passaggio dei pacchetti all'NsObject che si occuperà della perdita dei pacchetti.

4.5.2 BiConnector

La classe **BiConnector** viene utilizzata come classe base di **NsObject** bidirezionali. Questa classe contiene tre istanze ad altri **NsObject**: *uptarget_*, *downtarget_* e *drop_*. La variabile *drop_* ha la stessa funzione che ha nella classe **Connector**, cioè punta ad un **NsObject** che simula la perdita dei pacchetti. Le variabili *uptarget_* e *downtarget_* puntano a due **NsObject** simulando un passaggio di pacchetti bidirezionale per questo oggetto. Le funzioni virtuali *SendDown()* e *SendUp()* (implementate nelle classi derivate) utilizzano rispettivamente le variabili *downtarget_* e *uptarget_* per passare i pacchetti ai rispettivi **NsObject** puntati.

4.5.3 Agent

Gli Agent rappresentanno i punti terminali (*end-points*) in cui i pacchetti di livello network vengono costruiti e/o letti. La classe **Agent** deriva dalla classe **Connector** ed è implementata da una parte in C++ ed una parte in OTcl.
La classe **Agent** contiene uno stato interno, ossia un insieme di variabili il cui contenuto viene usato per assegnare valori a vari campi di un pacchetto simulato, prima che questo venga inviato. Questo stato può essere modificato da qualsiasi classe derivata dalla classe **Agent**.
La classe Agent supporta meccanismi di generazione e di ricezione di pacchetti. Le funzioni per allocare un nuovo pacchetto sono le seguenti:

- *allocpkt()* alloca un nuovo pacchetto (assegnando opportuni valori alla maggior parte dei campi dei corrispondenti header), restituendo come risultato un puntatore di esso;

- *allocpkt(int)* è del tutto analoga alla precedente, con la differenza che, oltre ai campi dell'header del pacchetto, viene anche allocato un *payload* di *n byte*, dove *n* viene fornito come argomento.

Nella classe Agent sono anche definite le funzioni membro *timeout()* e *recv()* che si prevede vengano sovrapposte dalle classi derivate. Il metodo *Agent::recv()* è sostanzialmente il principale "punto di entrata" per un Agent che riceve pacchetti. La classe Agent ha un puntatore *app_* che punta all'applicazione che si appoggia all'Agent considerato. Se tale puntatore risulta non nullo, quando un Agent riceve un pacchetto invoca il metodo *recv()* dell'applicazione, in modo che quest'ultima, prendendo in consegna il pacchetto dall'Agent di trasporto ricevente, lo tratti nel modo previsto, dopodichè il pacchetto viene "liberato" tramite la procedura *Packet::free()*.
Nella creazione di un Agent si possono osservare due cose:

- in primo luogo, non tutti gli Agent attualmente implementati prevedono un proprio metodo *recv()* : solo gli Agent che possono essere usati come ricevitori di pacchetti prevedono questa funzione. Ovviamente, questo significa che il metodo *recv()* viene specializzato negli Agent utilizzabili come ricevitori, mentre invece non viene specializzato per gli Agent non destinati a ricevere pacchetti, i quali quindi ereditano il metodo *Agent::recv()* della loro classe base, anche se tale metodo non viene mai invocato.

- in secondo luogo, il metodo *recv()*, qualora previsto, risulta diverso da Agent ad Agent, come è ovvio che sia.

Per creare un nuovo Agent, bisogna fondamentalmente compiere le seguenti operazioni:

- decidere la sua *struttura ereditaria* e creare le opportune definizioni di classe;
- definire il metodo *recv()* ed eventualmente il metodo *timeout()*;
- definire ogni necessaria classe **timer**;
- definire le funzioni per l' OTcl linkage;
- scrivere il codice OTcl necessario per accedere all'agente;
- ricompilare ns per rendere "operative" le modifiche effettuate.

Application

La classe **Application** rappresenta la classe base per le possibili applicazioni associate ad un Agent. Questa classe ha un'istanza all'Agent associato ad essa. L'assegnamento avviene tramite il comando *attach-agent* definito nella sua funzione *command()*. Nella funzione *command* vengono chiamate delle funzioni virtuali dichiarate nella classe. Queste funzioni possono essere implementate nella classe derivata che rappresenta ogni specifica applicazione:

- la funzione *start()* serve per far partire l'applicazione;
- la funzione *stop()* per fermare l'applicazione;
- la funzione *send()* per mandare *nbytes* ad un suo pari;
- la funzione *recv()* per la ricezione di *nbytes*.
- la funzione *resume()* indica all'applicazione che l'agent ha mandato tutti i dati fino a quel punto nel tempo.

Alcune applicazioni possono essere implementate in OTcl, sempre derivando la classe **Application**, oppure in C++ e OTcl oppure solo in C++.

Timer

I Timer sono degli oggetti scritti in C++ che derivano solamente dalla classe **Handler** e quindi possono essere usati solamente in classi C++, ma hanno la particolarità che possono essere utilizzati nel campo *handler_* della classe **Event** per la schedulazione (**attenzione a non confondere questi tipi di timer con quelli derivati dalla classe Timer scritta in OTcl, che utilizza un'altro meccanismo ed in cui è possibile chiamare un'istanza negli script Tcl con i rispettivi comandi**). La maggior parte di questi Timer derivano dalla classe *TimerHandler* che è caratterizzata dalle seguenti funzioni:

- *sched(double delay)* schedula il timer tra *delay* secondi;
- *resched(double delay)* rischedula un timer tra *delay* secondi (il timer deve essere pending);
- *cancel()* cancella un timer pending;
- *status()* ritorna lo stato del timer (*TIMER_IDLE* che non deve essere rischedulato, *TIMER_HANDLING* che può essere rischedulato, *TIMER_PENDING* che può essere cancellato).
- *virtual void expire(Event *)* deve essere implementata nella classe che deriva il timer. E' chiamata dalla funzione *handle()* del timer e può essere utilizzata per chiamare *resched()* qualora si vuole rischedulare un evento timer;
- *virtual void handle(Event *)* può essere implementata nella classe derivata è identifica la funzione che verrà chiamata dalla routine di schedulazione per l'esecuzione.

Nella classe *TimerHandler* sono definite anche due variabili:

- *status_* lo stato corrente del timer;
- *event_* l'evento consumato dal timer.

4.5.4 Queue

Le queues rappresentano la locazione dove i pacchetti possono essere contenuti o scartati. La coda può essere bloccata finchè non si libera il canale dove i pacchetti devono passare. La classe C++ **Queue** è una classe derivata da **Connector** è viene utilizzata per implementare vari tipi di code (CBQ,

DropTail, FQ, SFQ etc). Le funzioni principali di questa classe sono *enque()* e *deque()*, due funzioni virtuali implementate dalle classi derivate che servono per inserire e togliere elementi dalla coda e una funzione *recv()* che serve per ricevere i pacchetti da un altro **NsObject**. Questa classe è caratterizzata dalle seguenti variabili: *qlim_* indica il numero di pacchetti che la coda può contenere, *blocked_* indica se il link in cui è inserita è bloccato (0 no 1 si) e la variabile *qh_* che è un'istanza ad un oggetto di tipo **QueueHandler** che viene istanziata alla creazione della coda. L'oggetto **QueueHandler** quando viene inizializzato contiene la referenza all'oggetto *Queue* a cui appartiene. Questo lo fà nel costruttore di **Queue** usando l'espressione *qh_ (*this)*. Quando la coda riceve un pacchetto se il link è libero fa transitare il pacchetto direttamente e setta il canale come blocked altrimenti lo mette in coda. Quando viene mandato il pacchetto viene mandato anche il puntatore al **QueueHandler** che verrà schedulato da un **Ns-Object** in seguito. La funzione *handle()* di questo oggetto quando è invocata per l'esecuzione esegue la funzione *resume()* di **Queue** che serve a prendere un pacchetto in coda e mandarlo nel link. Se non trova nessun pacchetto in coda setta il link come libero.
Nella classe **Queue** c'è una variabile *pq_* . Questa variabile è un puntatore ad un oggetto **PacketQueue** che implementa una struttura per i pacchetti in coda se non viene implementata nelle sottoclassi.

4.5.5 errorModel

Un modulo di errore simula errori di livello fisico (link-level errors) oppure perdite, tramite due distinti meccanismi: la presenza di errori su un pacchetto è indicata settando l'apposito *error flag nel common header* del pacchetto stesso, mentre invece la perdita completa del pacchetto viene simulata consegnando il pacchetto, invece che al suo destinatario, ad un apposito *drop target*.
In una simulazione, gli errori possono essere generati tramite un modello molto semplice, specificando un tasso di errore sui pacchetti, oppure tramite modelli statistici ed empirici più complessi. Al fine di supportare un'ampia varietà di tali modelli, l'unità di errore può essere specificata in termini di pacchetti o di bit oppure essere basata sul tempo (timer-based). La classe **ErrorModel** è derivata dalla classe base **Connector**. Di conseguenza, essa eredita alcuni metodi per "agganciare" oggetti (come i pacchetti), come ad esempio i metodi *target()* e *drop-target()*. Se esiste un *drop target*, esso riceve tutti i pacchetti che sono stati "corrotti" dal modello di errore. Altrimenti, il modello di errore semplicemente setta il *flag error del common header* del

pacchetto, in modo da demandare agli agenti di trasporto la responsabilità di gestire gli errori.
La classe **ErrorModel** contiene sia i meccanismi sia la "politica" per la perdita dei pacchetti. In particolare, il meccanismo per la perdita dei pacchetti è gestito dal metodo *recv()*, mentre invece la "politica" per la gestione dei pacchetti corrotti è eseguita dal metodo *corrupt()*. La classe **ErrorModel** implementa solo una semplice politica basata su un singolo tasso di errore, espresso in termini di pacchetti o di bit. Politiche più sofisticate possono invece essere implementate derivando apposite classi da **ErrorModel** e ridefinendo il metodo *corrupt()*.

4.5.6 LinkDelay

La classe **LinkDelay** è una classe derivata da **Connector** e serve per simulare il tempo che un pacchetto impiega per attraversare un link. Questo oggetto, tramite la funzione *recv()*, riceve un un pacchetto e un *Handler* (di solito un **QueueHandler**), calcola il tempo che il pacchetto ci mette ad attraversare il link tramite la seguente funzione:
$txttime(Packet\ *p)\{\ 8. * hdr\_cmn :: access(p) - > size()/bandwidth\_\ \}$
e schedula due eventi. La schedulazione di questi eventi dipende dalla variabile *dinamic_* che stabilisce se l'oggetto è dinamico oppure no.
Un oggetto non dinamico schedula un evento riferito all'oggetto *target_* con ritardo *txttime(p)+delay_* e un evento riferito all'**Handler** con ritardo *txttime(p)*.
Per un **LinkDelay** dinamico invece di schedulare l'evento riferito al *target_* inserisce il *Packet* in una coda interna e schedula se stesso con ritardo *txttime(p)*. L'esecuzione dell'*handle* relativo a **LinkDelay** toglie un elemento dalla coda e manda in esecuzione una funzione *send()* che a sua volta esegue *target_ ->recv(Packet,0)* per mandare il pacchetto all'NsObject successivo.

4.6 Node

Un nodo è un TclObject composto da NsObject. Possiamo vedere il nodo come un contenitore di NsObject collegati tra loro. Osservando la struttura di un nodo unicast (*Figura 4.2*) possiamo notare le componenti address classifer (*classifer_*) e port classifer (*dmux_*) che servono a distribuire i pacchetti entranti nel nodo negli agenti o negli outlink corretti.
Tutti i nodi hanno le seguenti componenti:

- un indirizzo (*id_*) univoco;

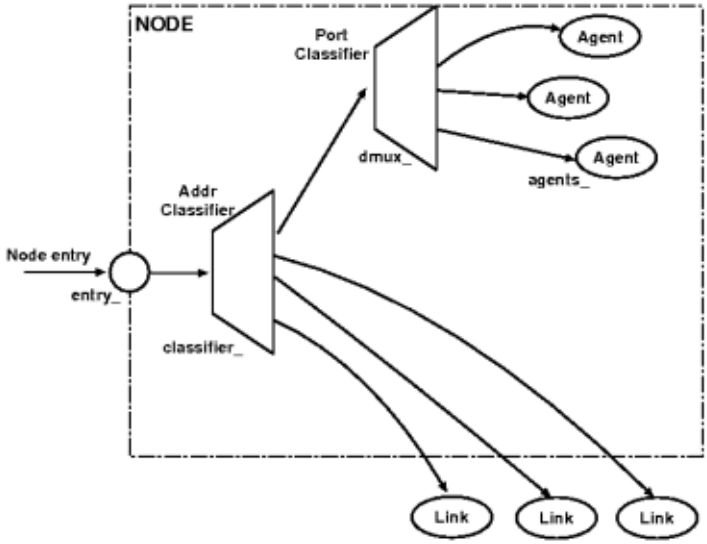

Figura 4.2: Nodo unicast

- una lista di neighbors (*neighbor_*);
- una lista di agenti (*agent_*);
- un routing module.

4.6.1 Classifier

Ns simula la ricezione di un pacchetto da parte di un nodo e la relativa analisi per trovare la destinazione tramite gli oggetti di tipo **Classifier**. La classe Classifier deriva dalla classe NsObject che a sua volta viene derivata dalle classi che implementano i vari classificatori. Ogni classificatore contiene una tabella degli oggetti simulati indicizzata tramite uno *slot number*. Lo scopo del classificatore è quello di determinare lo slot associato con l'oggetto destinazione del pacchetto in arrivo ed indirizzare il pacchetto verso il percorso esatto. Quando un classificatore riceve un pacchetto si trova lo slot associato con la destinazione tramite una funzione *classify()* che è definita diversamente in ogni classificatore.
Ns supporta questi tipi di classificatori:

- **Address Classifier**: supporta l'unicast routing e applica un bitwise shift e una mask operation all'indirizzo di destinazione del pacchetto per produrre uno slot number;

- **Multicast Classifier**: classifica i pacchetti in base all'indirizzo sorgente e destinazione (gruppo). Esso mantiene una tabella hash che assegna ogni source/group ad uno slot;
- **Multipath Classifier**: quando si hanno più destinazioni con lo stesso costo di routing li usa tutte simultaneamte;
- **Hash Classifier**: è usato per classificare un pacchetto che ha un particolare flusso;
- **Replicator**: non usa la funzione *classify()* ma mantiene una tabella di n slot. Per ogni pacchetto in arrivo produce n copie dell'oggetto e li manda a tutti gli n slot.

4.6.2 Routing

L'implementazione di un routing in ns consiste in un blocco di tre funzioni:

- **routing agent**: serve a scambiare i pacchetti di routing con i vicini;
- **route logic**: usa le informazioni dei routing agent, oppure quelle globali in caso di routing statico, per calcolare le computazioni di routing;
- **classifier**: usa la tabella di routing per indirizzare i pacchetti.

L'oggetto **routing module** guida i tre blocchi di funzioni descritte in precedenza e si interfaccia con i nodi organizzando i classificatori.

4.7 Simplex Link

Il link è un contenitore di NsObject che serve a collegare due nodi. La classe **Link** è una classe standard OTcl, utilizzata come classe base per la creazione di link. La classe **SimpleLink** (*Figura 4.3*) è una classe derivata da **Link** che simula una connessione point to point tra due nodi. Ci sono cinque istanze a variabili che identificano il simplex-link:

- *head_* : il punto di entrata del link;
- *queue_* : referenza alla coda del link;
- *link_* : referenza all'oggetto che simula i tempi di transizione dei pacchetti nel link (**LinkDelay**);

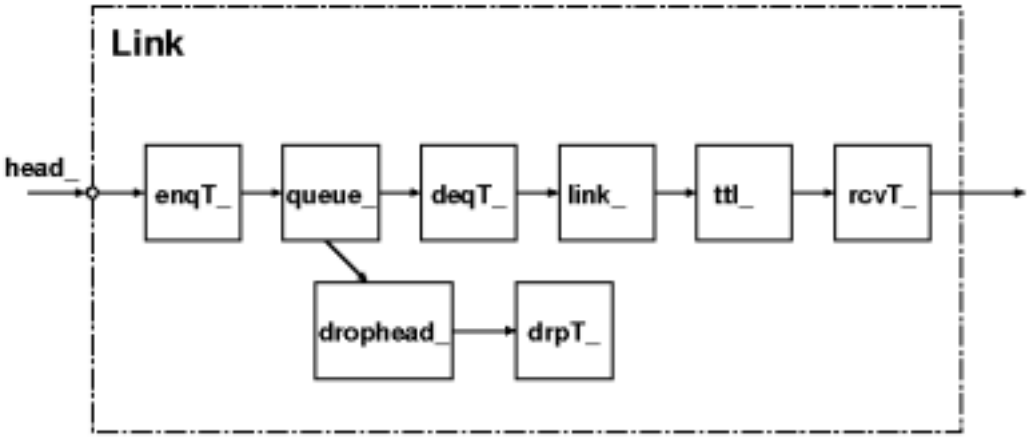

Figura 4.3: Simplex Link

- *ttl_* : referenza all'oggetto che manipola il campo ttl in un pacchetto;
- *drophead_*: referenza che punta all'oggetto che gestisce i pacchetti che vengono "droppati" dalla coda.

In aggiunta ci sono le referenze ad oggetti per il trace:

- *enqT_*: referenza all'oggetto che traccia i pacchetti in entrata nella coda;
- *deqT_*: referenza all'oggetto che traccia i pacchetti in uscita dalla coda;
- *drpT_*: referenza all'oggetto che traccia i pacchetti "droppati" dalla coda;
- *rcvT_*: referenza all'oggetto che traccia i pacchetti che raggiungono il nodo destinazione.

Per il collegamento con i nodi, ns da la referenza *head_* ad un oggetto del nodo sorgente e all'ultimo elemento del link da la referenza *head_* del nodo destinazione. Per creare un link bi-direzionale ns utilizza due simplex-link.

Capitolo 5

Pdnet

5.1 Motivazioni

Pdnet è un simulatore di reti scritto interamente in C++ strutturato in modo che l'implementazione di **Scheduler Paralleli** sia fatta semplicemente. In questa versione sperimentale sono implementate solo le basi del funzionamento di una rete perchè lo scopo dell'esperimento è quello di testare uno scheduler parallelo in esso.
Pdnet prende spunto dal funzionamento di ns con l'aggiunta di varianti strutturali che consentono una facile implementazione di scheduler paralleli. Le caratteristiche principali di Pdnet sono le seguenti:

- struttura centralizzata: le componenti del simulatore si scambiano le entità tramite la classe principale. Questa implementazione serve a semplificare il calcolo statistico sul passaggio dei pacchetti tra le componenti del sistema, che viene fatto nella classe centrale senza coinvolgere le altre strutture del simulatore. Queste statistiche sono utili per l'applicazione di alcune tecniche di simulazione parallela in cui si deve distribuire il carico negli LP (Local Process) coinvolti nella simulazione;

- struttura degli header dei pacchetti a livelli: serve ad avere una dimensione ridotta dei pacchetti utilizzati nella simulazione rispetto al meccanismo utilizzato in ns. Nelle simulazioni parallele i messaggi possono essere formati dalle entità della simulazione, che nel simulatore sono i pacchetti. Diminuendo la dimensione dei pacchetti diminuisce la lunghezza dei messaggi remoti;

- gli oggetti contenitore derivano da un'unica classe base: alcune tecniche di parallelizzazione hanno bisogno che la struttura sia divisa in atomi.

Una classe base unica che ha delle regole strutturali ben definite è l'ideale per la definizione di atomi strutturali;

- riferimento delle schedulazioni ad alcuni oggetti particolari: per implementare semplicemente scheduler paralleli che spostano i contenitori frequentemente tra gli LP. Gli eventi, siccome sono associati a delle componenti di questi contenitori, possono essere facilmente trovati per essere spostati;

- lo stato delle varie componenti è composto da semplici variabili: in alcune tecniche di parallelizzazione lo stato del sistema, prima dell'esecuzione di un evento, deve essere memorizzato per eventuali ripristini. Se questo stato è composto da semplici variabili questa operazione si può fare semplicemente;

5.2 Struttura

La *Figura 5.1* rappresenta la struttura principale di Pdnet. La struttura del

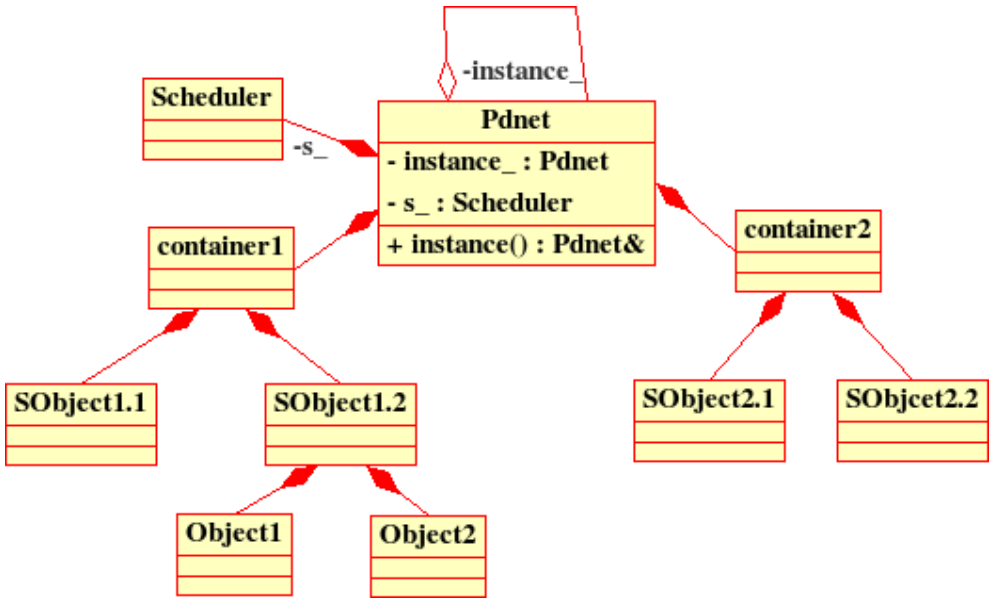

Figura 5.1: Struttura Pdnet

simulatore è una struttura centralizzata in cui la classe principale è **Pdnet**. Questa struttura consente ad ogni componente del simulatore di comunicare con un'altra componente tramite la classe Pdnet. Le componenti del simulatore possono essere raggiunte gerarchicamente partendo dalla classe

principale. Le caratteristiche dell'oggetto Pdnet, che rappresenta il cuore del simulatore, sono le seguenti:

- viene creato quando inizia la simulazione;
- ha un'istanza statica che permette alle altre componenti del simulatore di comunicare con esso. Questa istanza si reperisce tramite la funzione *instance()*;
- inizializza, configura, esegue e termina la simulazione;
- ha un'istanza dello scheduler (*s_*);
- ha un'array di istanze (*table_*) agli oggetti container;
- utilizza un meccanismo di comunicazione con le componenti scheduler e container tramite alcune funzioni, che utilizzano le istanze memorizzate di queste componenti;

5.2.1 Scheduler

L'oggetto Scheduler identifica lo schedulatore di eventi in Pdnet. Per implementare uno scheduler in Pdnet si eseguono i seguenti passi:

- si crea una classe che deriva dalla classe virtuale **Scheduler**. Nella classe creata si implementano le seguenti funzioni:
  - *void run()*: funzione di routine;
  - *void insert(Event *)*: inserisce un evento in coda;
  - *Event *deque()*: toglie il prossimo evento dalla coda restituendolo;
  - *Event *head()*: restituisce il primo elemento della coda;
  - *void cancel(double)*: cancella un elemento con un determinato uid (identificatore evento) dalla coda.
- si associa il nome dello scheduler alla classe creata nella funzione *Pdnet::SchedType(char *t)*:

```
void Pdnet::schedType(char *t)
{
    if(s_!=0)
        delete s_;
    if(strcmp("heap",t) == 0)
```

```
    {
        s_ = new HeapScheduler();
    }
    else if(strcmp("Parheap",t) == 0)
    {
        s_ = new ParScheduler();
    }
}
```

La funzione *SchedType()* viene utilizzata per cambiare lo scheduler nelle simulazioni di Pdnet. Attualmente gli scheduler disponibili sono due:

1. l'heap scheduler: utilizza una struttura heap come coda di priorità e un meccanismo DES di tipo event-driven per la gestione degli eventi;
2. il ParScheduler : scheduler parallelo che utilizza un heap come coda di priorità e MPI per la comunicazione tra gli LP.

La classe Pdnet ha una collezione di funzioni che servono a comunicare con gli scheduler:

- *schedule()*: schedula un evento di routine. Utilizza la funzione *insert()* dello Scheduler per inserire l'evento in coda;
- *atSchedule()*: schedula gli eventi iniziali della simulazione. Utilizza la funzione *insert()* dello Scheduler per inserire l'evento in coda;
- *deleteEvent(double uid)*: cancella l'evento in coda identificato da uid utilizzando la funzione *cancel()* dello Scheduler.

5.2.2 Container

I Container sono gli oggetti che rappresentano la struttura atomica delle componenti di una rete da simulare. Ogni Container è identificato tramite un indice univoco (*Cid Container id*) che rappresenta lo slot di *Pdnet::table_* in cui è contenuta la sua istanza. Quando un oggetto di questo tipo viene creato si inserisce nella *Pdnet::table_* tramite la funzione *Pdnet::addContainer()*.
La classe **container** è una classe virtuale che viene utilizzata come classe base per la creazione di svariati tipi di contenitori. Questa classe ha un puntatore *stable_* che punta ad un array di istanze di *SObject* che sono contenuti nel container. Un SObject può essere inserito in un container tramite la funzione *container::addSobject()*.

Nella classe container sono definite due funzioni virtuali *type()* e *handle()* che sono implementate dalle classi derivate. La prima funzione ritorna il tipo di container mentre la seconda viene invocata quando la routine di esecuzione del simulatore esegue un evento associato ad un oggetto contenuto nel container.

5.2.3 SObject

Gli SObject sono gli oggetti che compongono i container e rappresentano i punti di riferimento per le schedulazioni. Ogni SObject è identificato tramite un indice univoco (*Sid SObject id*), all'interno del container in cui è contenuto, che rappresenta lo slot di *container::stable_* in cui è contenuta la sua istanza. Un oggetto di questo tipo, ha una chiave univoca (Cid, Sid) all'interno di Pdnet. Gli SObject possono essere composti da Object e il puntatore *SObject::table_* punta ad un array di istanze ad oggetti Object che sono contenuti in SObject. Questi Object possono essere inseriti all'interno degli SObject tramite la funzione *SObject::addObject()*. Una cosa importante da sottolineare è che gli SObject non devono per forza contenere Object.
La classe SObject è una classe virtuale che definisce le seguenti funzioni virtuali implementate dalle classi derivate:

- *handle()*: viene eseguita all'esecuzione di un evento associato con l'SObject;
- *recv()*: viene eseguita quando un SObject riceve un pacchetto direttamente da un altro SObject senza effettuare schedulazioni (scambio di entità all'interno dello stesso evento);
- *type()*: identifica il tipo dell'SObject.

5.2.4 Object

Gli Object sono oggetti che possono essere inseriti all'interno degli SObject. Questi oggetti non sono punti di schedulazione ma servono a definire alcune componenti di un SObject.
La classe Object ha due funzioni virtuali che vengono implementate dalle classi derivate per ricevere (*recv()*) e mandare (*send()*) eventi ad altre componenti del simulatore.
Gli Object possono essere utilizzati per rappresentare singole componenti, come un'applicazione all'interno di un Agent (che sarà un SObject), o per costruire delle catene di oggetti all'interno di un SObject. In queste catene non ci possono essere schedulazioni tra un Object ed un altro.

5.2.5 Regole per la creazione di strutture

Pdnet è stato creato per rendere semplice l'applicazione di tecniche di parallelizzazione. In alcune di queste tecniche si deve suddividere la struttura da simulare in atomi che sono posizionati nei vari LP. Lo scambio di messaggi tra un atomo ed un altro avviene tramite delle schedulazioni con ritardo maggiore di zero. Le strutture atomiche di Pdnet sono i container che hanno come confine i punti di schedulazione. In base a questa osservazione possiamo dedurre che i container sono i nodi e i canali di una network.
All'interno del container ci sono degli SObject che vengono identificati tramite la coppia di valori (*Cid, Sid*). Ogni oggetto che può essere un punto di riferimento per una schedulazione deve essere di questo tipo. Oggetti di questo tipo sono gli Agent, i Simplex Link (prima parte di un link), le NetCard (seconda parte di link) etc. Un esempio può essere il collegamento tra due nodi da parte di un link. Supponiamo che il nodo n_1 comunica con il nodo n_2 tramite un link . I nodi sono creati come container e il link è diviso in due parti nel punto in cui si effettua la schedulazione che simula il tempo che un pacchetto impiega per attraversarlo. Le due parti del link sono create come due SObject separati. Il primo SObject è inserito in n_1 mentre il secondo in n_2. In questo modo il passaggio di un pacchetto da n_1 a n_2 avviene tramite una schedulazione con ritardo maggiore di zero.

5.3 Packet

I *Packet* sono gli oggetti che simulano i pacchetti reali di una rete e servono a rappresentare le strutture per la simulazione di protocolli.
La classe **Packet** è simile a quella di ns con relativo PacketHeaderMenager (*PacketMenager* in Pdnet) che inizializza la lunghezza degli header da utilizzare. Le funzioni statiche *Packet::alloc()* e *Packet::free()* servono ad allocare e deallocare un pacchetto. Il meccanismo utilizzato per l'allocazione e la deallocazione degli oggetti Packet utilizza il metodo della lista di elementi liberi, che è utile per avere un procedimento di allocazione e deallocazione più veloce.
La differenza tra ns e Pdnet è dovuta alla struttura dell'header dei pacchetti. L'accesso alle variabili della struttura che rappresenta un header è diretto, a differenza di ns che utilizza delle funzioni membro per accedere ai dati. In ns ogni header ha una posizione in un array chiamato BOB e la dimensione di questo vettore è data dalla somma delle dimensioni di tutti gli header definiti. In Pdnet il BOB è diviso in livelli. Ad ogni livello sono associati un gruppo di header. Ogni pacchetto può utilizzare un solo header per ogni livello. La

lunghezza del BOB è data dalla somma delle dimensioni massime delle strutture degli header assegnate ad ogni livello. Quindi la posizione di un header nel BOB è data dalla posizione assegnata al suo livello di appartenenza. I livelli degli header sono i seguenti:

- il livello di comunicazione: caratterizzato dall'header hdr_cmn che contiene alcuni valori che servono alla comunicazione, come la dimensione reale del pacchetto e gli offset degli altri livelli. E' allocato all'inizio del BOB;
- il livello data link (LEVELDL): utilizzato per i protocolli dello strato MAC della classificazione OSI (in questa versione di Pdnet non è stato ancora implementato nessun protocollo di questo tipo);
- il livello IP (LEVELIP): utilizzato per i protocolli dello strato di rete della classificazione OSI;
- il livello Trasporto (LEVELTR): utilizzato per i protocolli dello strato trasporto della classificazione OSI (TCP,UDP etc);
- il livello Applicativo (LEVELAP): utilizzato per i protocolli degli strati superiori a quello di trasporto della classificazione OSI.

Questa scelta architetturale della rappresentazione dei pacchetti è stata fatta per diminuire la dimensione reale del pacchetto rispetto ad ns, in modo che lo scambio di messaggi tra processi in scheduler paralleli produca uno scambio di dati ridotto.
Per la creazione di nuovi header di pacchetto si eseguono i seguenti passi:

- si crea una nuova struttura per l'header e si definisce una costante identificativa in hdr.h:

```
#define HDRUDP 3

struct hdr_udp
{
    int sport_;
    int dport_;
};
```

- si inserisce nell'apposito livello della funzione *init()* del PacketMenager il codice per valutare la dimensione dell'header. La dimensione di ogni livello viene calcolata in base alla dimensione massima degli header associati al livello.

```
void PacketMenager::init() {
    int dlevel = 0;
    int maxSize = sizeof(struct hdr_cmn);
    unsigned int max;
    Packet::hpos_ = new int[NLEVEL];
    while(dlevel < NLEVEL)
    {
        switch(dlevel)
        {
            case LEVELDL:
                max = 0;
                Packet::hpos_[LEVELDL] = maxSize;
                maxSize += max;
                break;
            case LEVELIP:
                max = 0;
                Packet::hpos_[LEVELIP] = maxSize;
                if(max < sizeof(struct hdr_ip))
                    max = sizeof(struct hdr_ip);
                maxSize += max;
                break;
             case LEVELTR:
                max = 0;
                Packet::hpos_[LEVELTR] = maxSize;
                if(max < sizeof(struct hdr_tcp))
                    max = sizeof(struct hdr_tcp);
                if(max < sizeof(struct hdr_udp))
                    max = sizeof(struct hdr_udp);
                maxSize += max;
                break;
             case LEVELAP:
                max = 0;
                Packet::hpos_[LEVELAP] = maxSize;
                maxSize += max;
                break;
             default : return;
        }
        dlevel++;
    }
    Packet::hdrlen_ = maxSize;
}
```

La funzione *PacketMenager::init()* viene chiamata quando il simulatore viene inizializzato per configurare la struttura del BOB, con il relativo offset dei livelli. Questa configurazione è valida per tutta la simulazione.

- si inserisce nell'apposito livello della funzione *setHeader()* di Packet il codice per settare il tipo di pacchetto assegnato ad un livello.

```
void Packet::setHeader(int leveldl,int levelip,
                       int leveltr,int levelap)
{
      struct hdr_cmn *cmn = (hdr_cmn *) hdr_;
      /*header livello data link-mac*/
      switch(leveldl)
      {
          default:
                  cmn->htype_[LEVELDL] = HDRNO;
                  break;
      }
      /*header livello ip*/
      switch(levelip)
      {
          case HDRIP:
                  cmn->htype_[LEVELIP] = HDRIP;
                  break;
          default:
                  cmn->htype_[LEVELIP] = HDRNO;
                  break;
       }
      /*header livello di trasporto*/
      switch(leveltr)
      {
          case HDRTCP:
                  cmn->htype_[LEVELTR] = HDRTCP;
                  break;
          case HDRUDP:
                  cmn->htype_[LEVELTR] = HDRUDP;
                  break;
      }
      /*header livello applicativo*/
      switch(levelap)
      {
```

```
            default:
                    cmn->htype_[LEVELAP] = HDRNO;
                    break;
        }
    }
```

Questa funzione viene chiamata dagli oggetti che creano un nuovo pacchetto per inizializzare i livelli associati al pacchetto creato;

Un'altra cosa da notare è la struttura di *hdr_cmn*:

```
struct hdr_cmn
{
    /*comunicazione*/
    int size_;
    int errcount_;

    /*informazioni header pacchetto*/
    int htype_[NLEVEL];
};
```

Il campo *size_* contiene la dimensione reale del pacchetto, il campo *errcount_* ci dice se il pacchetto è corrotto, l'array *htype_* ci indica il tipo di header associato a ciascun livello nel pacchetto.
Per capire come gli oggetti accedono ai campi di un pacchetto guardiamo il seguente esempio:

```
/*recupera l'header del pachetto*/
unsigned char *h = (e->pack_) -> getHdr();
/*recupera hdr_cmn che si
* trova all'inizio del BOB*/
hdr_cmn *cmn = (hdr_cmn *) h;
/*controlla se a livello ip c'è
* l'header desiderato*/
if(cmn->htype_[LEVELIP] == HDRIP)
{
    /*recupera l'header del livello ip*/
    hdr_ip *ip = (hdr_ip *)& h[Packet::hpos_[LEVELIP]];
    /*modifica il campo ttl dell'header*/
    ip -> ttl_++;
}
```

Gli offset dei livelli sono contenuti nell'istanza statica dell'array *Packet::hpos* che viene inizializzata dalla funzione *PacketMenager::init()* e i valori assegnati risultano invariati durante la simulazione.

5.4 Event e schedulazioni

Gli Event sono le entità che gli oggetti di Pdnet si scambiano durante la simulazione. La struttura di un Event ha i seguenti campi:

1. *time_* : tempo di esecuzione evento;
2. *cid_* : indice container di riferimento;
3. *sid_* : indice SObject di riferimento;
4. *type_* : tipo di Event (EVENTSTART, EVENTSTOP, EVENTHANDLE);
5. *uid_* : identificativo univoco Event;
6. *pack_* : pacchetto assegnato all'Event;
7. *input_* ,*output_* : campi che servono per le schedulazioni parallele.

Gli oggetti si scambiano gli Event in maniera diretta o tramite schedulazioni. Due oggetti che appartengono allo stesso container si scambiano un Event in maniera diretta. In *figura 5.2* è rappresentato il meccanismo che utilizza un SObject per mandare un Event ad un altro SObject dello stesso container utilizzando una comunicazione diretta. Il primo SObject comunica con

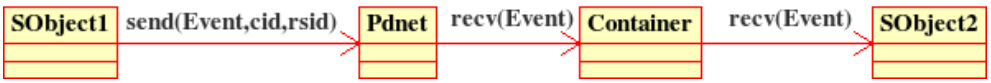

Figura 5.2: Comunicazione diretta

l'oggetto Pdnet chiamando la sua funzione *send()* con gli argomenti Event, cid e rsid. L'argomento Event rappresenta l'entità da mandare all'SObject identificato dalla coppia di valori cid e rsid. Siccome il container a cui manda l'Event è quello in cui lui è contenuto, il valore del cid è uguale al suo. La funzione *send()* di Pdnet aggiorna i campi cid e sid di Event, ricerca il container associato con quel cid per passargli l'Event tramite la funzione *recv()* del container. Questa funzione non fa altro che bypassare l'Event all'SObject con quel sid nel container tramite la funzione *recv()* dell'oggetto destinazione. Un'implementazione del genere si utilizza per rendere semplice il calcolo del

carico di un singolo Event coinvolgendo solo le funzioni della classe Pdnet per effettuare questi calcoli e calcolandolo in base al numero di SObject che coinvolge durante la sua esecuzione.
Quando un Event deve passare da un SObject in un container ad un altro SObject contenuto in altro container utilizza un meccanismo di schedulazione con ritardo maggiore di zero. Il primo SObject chiama la funzione *schedule(Event *,cid,sid,delay)* di Pdnet in cui i parametri di input rappresentano rispettivamente l'Event da schedulare, l'oggetto a cui fa riferimento (coppia cid, sid), il ritardo si schedulazione. Questa funzione riempie i campi dell'Event in cui assegna il cid, il sid, il tempo di esecuzione (*clock()+delay*), il tipo (EVENTHANDLE evento di routine) e l'uid. Infine inserisce l'Event nella coda degli eventi dello scheduler.
Quando lo scheduler esegue il ciclo di routine esegue gli Event chiamando la funzione *exec(Event *)* di Pdnet che cerca il container associato con il cid ed esegue la sua funzione *handle(Event *)*. Questa funzione a sua volta cerca l'SObject associato al sid di Event ed esegue la sua funzione *handle(Event*)*. Gli Event iniziali della simulazione sono creati tramite la funzione *atSchedule(cid,sid,type,delay)*. Questa funzione crea un Event e gli associa cid, sid, type, uid e tempo di schedulazione. Il tipo di Event può essere un EVENTSTART o EVENTSTOP. Questo serve per far partire o stoppare un Agent. La funzione handle di un container di tipo Node è la seguente:

```
void Node::handle(Event *e)
{
    if(e->type_ == EVENTHANDLE)
    {
       stable_[e->sid_] -> handle(e);
    }
    else if(e->type_ == EVENTSTART)
    {
          Agent *a = (Agent *) stable_[e->sid_];
          a->start();
          delete e;
    }
    else if(e->type_ == EVENTSTOP)
    {
          Agent *a = (Agent *) stable_[e->sid_];
          a->stop();
          delete e;
    }
    else
```

```
    {
        delete e;
    }
}
```

Dal codice si nota che un Event di routine esegue la funzione *handle()* dell'-SObject associato, un Event di tipo EVENTSTART esegue la sua funzione *start()* e un Event EVENTSTOP esegue la sua funzione *stop()*.

5.5 Node

L'oggetto *Node* è un container che serve a rappresentare un nodo della rete da simulare. A differenza di ns in Pdnet i confini di un oggetto di questo tipo sono diversi. La differenza sta nel fatto che in Pdnet i link semplici non sono un oggetto indipendente dal nodo come in ns. Qui ogni link viene diviso in due parti nel punto in cui in ns (rappresentato dalla classe LinkDelay) viene effettuata una schedulazione. La prima parte del link viene inserita nel Node sorgente e la seconda parte viene inserita nel Node destinazione. Questo concetto dovrà essere preso in considerazione anche nelle possibili future implementazioni di vari tipi di rete(wireless, satellitari etc..).

```
class Node : public container
{
    private:
        int address_;
        int cid_;
        int head_;
        int demux_;
    public:
        Node();
        ~Node();
        int type();
        void handle(Event *);
        void setCid(int);
        void setAddress(int);
        int getAddress();
        int addRoute(int,int,double);
        int addSimplexLink(double,double,int,int,int,int,int);
        int addNetCard();
```

```
void addDemux();
void print();
int addAgent(Agent *);
double getTime(int address); };
```

La classe Node è caratterizzata dalle seguenti variabili:

- *address_* : indirizzo del nodo;
- *cid_* : cid associato al nodo;
- *head_* : identifica il sid dell'SObject che viene utilizzato per il routing;
- *demux_* : identifica il sid dell'SObject che simula le porte associate agli Agent;

Oltre alle funzioni implementate perchè virtuali nella classe base, in Node sono implementate le funzioni che servono per settare i valori delle variabili descritte sopra come *setAddress()*, le funzioni per inserire le componenti del nodo come SimplexLink (prima parte del link), netCard(seconda parte del link), demux etc. e le funzioni per settare il routing.

5.6 routing e demux

Gli SObject routing e demux servono a simulare l'instradamento dei pacchetti in un nodo. La *Figura 5.3* rappresenta un esempio di una struttura di un nodo con un routing a tre livelli. Un pacchetto quando transita in un nodo passa per l'oggetto routing che in base a delle tabelle di instradamento lo manda in un link o in un'altro oggetto di tipo routing. Quando il pacchetto transita nel link viene mandato in un altro nodo mentre se deve raggiungere un Agent situato nel nodo ad un certo punto si troverà a transitare in un SObject demux che contiene gli indirizzi delle porte associate a ciascun Agent.

In questa versione sperimentale di Pdnet si utilizza un routing statico ad un livello. In ogni nodo si ha una variabile *head_* che identifica il *sid_* del primo oggetto routing in cui si deve instradare il pacchetto. Ogni oggetto di tipo routing contiene due tabelle:

- *cn_*: ogni record di questa tabella contiene i campi *type*, *daddr*, *cid* e *sid* che rispettivamente indicano il tipo di SObject a cui fa riferimento il record (ROUTING, SIMPLEXLINK, DEMUX), l'indirizzo di destinazione associato all'oggetto a cui si riferisce , il cid dell'oggetto di riferimento e il suo sid. La classe routing implementa una funzione *addConn()* che serve ad inserire un record in questa tabella;

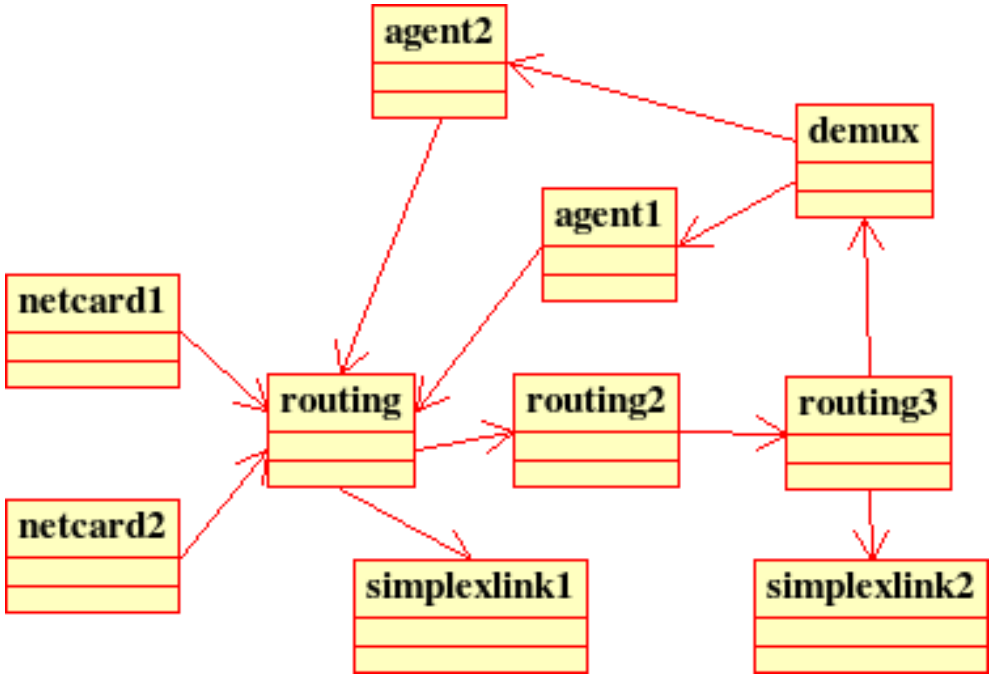

Figura 5.3: Esempio struttura nodo

- *rt_* : ogni record di questa tabella contiene i campi *address*, *time* e *dindex* che rispettivamente indicano l'indirizzo di routing, il tempo stimato per raggiungere la destinazione e l'indice di *cn_* associato all'indirizzo. La funzione *Node::addRoute()* serve a modificare un record o a inserirne uno nuovo per un determinato indirizzo.

Quando un pacchetto raggiunge l'oggetto routing questo oggetto legge il suo indirizzo di destinazione e lo cerca in *rt_* . Se lo trova manda il pacchetto all'oggetto in *cn_* identificato con l'indice che ha trovato in *rt_* .
Quando inizia la simulazione si inizializzano tutte le tabelle di routing della rete tramite la funzione *init_ routing()* della classe **Pdnet**. In questa versione di Pdnet non sono toccate per tutta la simulazione perchè si utilizza un routing statico.
La funzione di inizializzazione utilizza i delay e la banda dei link per stimare un ritardo tra i nodi e tramite l'algoritmo di **Floyd-Warshall** calcola i cammini minimi tra i nodi e i percorsi di questi cammini. In base a questi valori la funzione *init_ routing()* inserisce i record di routing nei nodi.
L'oggetto *demux* contiene la lista di porte assegnate agli Agent del nodo. Quando un Agent è inserito nel nodo, tramite la funzione *Node::addAgent()*, il numero di porta riferito all'Agent è inserito nel *demux*. Quando un pac-

chetto raggiunge il demux, tramite il numero della porta di destinazione, è instradato verso l'Agent corrispondente.

Algoritmo di Floyd-Warshall

L'algoritmo di **Floyd-Warshall** risolve il problema dei cammini minimi tra tutte le coppie di nodi di un grafo orientato G=(V,E) dove V sono i nodi del grafo e E i suoi archi. Questo algoritmo ha un tempo di esecuzione $O(V^3)$. L'algoritmo considera i vertici "intermedi" di un cammino minimo, dove un vertice *intermedio* di un cammino semplice $p = (V_1, V_2, ..., V_n)$ è un qualunque vertice di p diverso da V_1 e da V_n cioè un qualunque vertice nell'insieme $\{V_2, V_3, ..., V_{n-1}\}$. L'algoritmo di **Floyd-Warshall** è basato sulla seguente osservazione. Sia $V = (1, 2, ..n)$ l'insieme dei vertici di G, e si consideri un sottoinsieme $(1, 2.., k)$ di vertici per un qualunque k. Per ogni coppia di vertici $i, j \epsilon V$ si considerino tutti i cammini da i a j in cui tutti i vertici intermedi sono presi in $\{1, 2, ..., k\}$, e sia p un cammino di peso minimo tra di essi. L'algoritmo di Floyd-Warshall sfrutta una relazione tra il cammino p ed i cammini minimi da i a j aventi tutti tutti i vertici intermedi nell'insieme $\{1, 2, .., k-1\}$. Questa relazione cambia se k è un vertice intermedio del cammino p oppure no.

- Se k non è un vertice intermedio del cammino p, allora tutti i vertici intermedi sul cammino p sono nell'insieme $\{1, 2, .., k-1\}$. Quindi un cammino minimo dal vertice i al vertice j con tutti i vertici intermedi nell'insieme $\{1, 2, ..., k-1\}$ è anche un cammino minimo da i a j con tutti i vertici intermedi nell'insieme $\{1, 2, ..., k\}$.

- Se invece k è un vertice intermedio sul cammino p, allora si spezza p in $i \rightarrow k \rightarrow j$. Si ha un cammino minimo da i a k con tutti i vertici intermedi nell'insieme $\{1, 2, .., k-1\}$; analogamente il cammino da k a j con tutti i vertici intermedi nell'insieme $\{1, 2, .., k-1\}$ è un cammino minimo.

Sulla base dell'osservazione precedente si propone una nuova formulazione ricorsiva delle stime di cammino minimo. Sia $d^k_{i,j}$ il peso di un cammino minimo dal vertice i al vertice j con tutti i vertici intermedi nell'insieme $\{1, 2, ..., k\}$. Quando $k = 0$, un cammino dal vertice i al vertice j senza vertici intermedi aventi un numero maggiore di zero è un cammino che non ha alcun vertice intermedio: di conseguenza esso ha al massimo un arco, e

quindi $d_{ij}^0 = pesoarco(i,j)$.

$$d_{ij}^k = \begin{cases} pesoarco(i,j) & se\ k = 0 \\ min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) & se\ k \geq 1 \end{cases}$$

La matrice $D^n = (d_{ij}^n)$ ci dà i cammini minimi. Per costruire i cammini minimi si crea la matrice dei predecessori T con le seguenti formule di ricorrenza. Per $k = 0$

$$T_{ij}^0 = \begin{cases} NIL & se\ i = j\ oppure\ pesoarco(i,j) = \infty \\ i\ se\ i \neq j\ e\ pesoarco(i,j) < \infty \end{cases}$$

Per $k \geq 1$

$$T_{ij}^k = \begin{cases} T_{ij}^{k-1} & se\ d_{ij}^{k-1} \leq d_{ik}^{k-1} + d_{kj}^{k-1} \\ T_{kj}^{k-1} & altrimenti \end{cases}$$

In base alle formule di ricorrenza si ha il seguente algoritmo:

```
Floyd-Warshall(pesoarco)
      n = rows(pesoarco)
      D = pesoarco
      T = inizializzazione predecessori ricorrenza k=0
      for k=1 to n
          for j=1 to n
              for i=1 to n
                  if(D[i][j] > D[i][k]+D[k][j]
                          T[i][j]=T[k][j]
                          D[i][j]=D[i][k]+D[k][j]
```

Per aggiungere il routing nelle tabelle dei nodi del simulatore per ogni percorso i j la funzione *init_ routing()* va a ritroso nella tabella T partendo dal predecessore di j fino ad arrivare al nodo k successivo ad i in questo percorso. Aggiunge nel record della tabella di routing di i con destinazione j l'indirizzo di k.

5.7 SimplexLink e Netcard

SimplexLink e NetCard sono gli SObject che simulano un link reale .Quando si crea un link tramite la funzione *addSimplexLink()* della classe **Pdnet** il simulatore esegue i seguenti passi:

- crea un oggetto netCard nel nodo destinazione tramite la funzione *addNetCard()*;

- crea un oggetto SimplexLink nel nodo sorgente tramite la funzione *addSimplexLink()*. A questo oggetto dà il cid e il sid del netCard in modo che un pacchetto che passa nel SimplexLink è bypassato tramite una schedulazione al netCard corrispondente;

- ritorna il cid e il sid del SObject SimplexLink.

La classe **SimplexLink** simula la prima parte del link dove è simulata una coda di eventi tramite le variabili *qmaxSize_* (dimensione massima della coda), *qsize_* (dimensione della coda in un dato istante) e *stime_* (tempo in cui il link si libera). Oltre alla coda dei pacchetti contiene le variabili *bw_* (bandwitdh del link) e *delay_* (ritardo del link). La funzione *recv()* del SimplexLink è chiamata quando un pacchetto raggiunge il link e si comporta nel seguente modo: controlla se la coda è piena e se lo è scarta il pacchetto. Se la coda non è piena calcola il tempo che il pacchetto ci mette a transitare nel link, controlla se ci sono altri pacchetti in coda ed in base a questo si comporta nel seguente modo:

- se non ci sono pacchetti in coda: controlla se ci sono altri pacchetti che stanno transitando nel link in quel preciso momento. Se non ci sono pacchetti in transito, setta *stime_* con il valore del tempo corrente più il tempo di passaggio di questo pacchetto nel link, schedula l'evento (autoschedulazione) che serve a liberare il link al tempo *stime_* e l'evento che simula l'arrivo al netcard al tempo *stime_* + *delay_* . Se invece c'è un pacchetto già in transito simula l'inserimento del pacchetto in coda in questo modo:

  - calcola *stime_* come il tempo di partenza di questo pacchetto più il tempo di passaggio nel link;
  - incrementa la variabile *qsize_* che indica il numero dei pacchetti in coda;
  - schedula l'evento che serve a decrementare la dimensione della coda del link oppure a liberare il link se non ci sono altri eventi in coda. Questo evento serve a simulare la partenza dell'evento successivo in coda nel link. Questa schedulazione sarà riferita a se stesso;
  - schedula l'evento che simula l'arrivo del pacchetto nel netCard al tempo *stime_* +*delay_* .

- se ci sono pacchetti in coda: inserisce un pacchetto in coda con il meccanismo spiegato sopra.

La classe **NetCard** simula la seconda parte del link. Quando un pacchetto arriva nell'oggetto NetCard, questo oggetto verifica se il link è interrotto oppure no (funzionalità non implementata in questa versione di Pdnet), incrementa il campo *ttl_* del livello IP del pacchetto e inoltra il pacchetto verso l'oggetto routing del nodo a cui appartiene.

5.8 Agent

Nella simulazione di protocolli e applicazioni oltre alle strutture per gli header si devono definire gli Agent che sono gli **end points** delle applicazioni. Le applicazioni possono essere implementate direttamente nell' Agent oppure utilizzando, insieme all'Agent, un oggetto Application che simula il livello superiore a quello di trasporto. L'utilizzo degli oggetti Application dipende dai protocolli e dal tipo di applicazione che si vuole implementare. In alcune applicazioni gli end points sono Agent dello stesso tipo mentre in altre applicazioni sono Agent diversi. Gli oggetti di tipo Agent devono essere accoppiati in base ai criteri di implementazione.
Gli Agent in Pdnet sono implementati da classi derivate dalla classe virtuale Agent che definisce uno schema generale per questi tipi di oggetti. La classe **Agent** è caratterizzata dalle seguenti variabili:

- *app_*: istanza ad un Object Application. Questo tipo di Object può essere utilizzato per simulare i livelli superiori a quello di trasporto.
- *address_*: indirizzo del nodo a cui appartiene l'Agent;
- *raddress_*: indirizzo del nodo dell'Agent connesso con questo Agent;
- *port_*: la porta che identifica l'Agent;
- *dport_*: la porta che identifica l'Agent connesso con questo Agent;
- *psize_*: dimensione dei pacchetti che gestisce l'Agent;
- *cid_*: cid del nodo a cui appartiene l'Agent;
- *sid_*: sid dell'Agent;
- *rsid_*: il sid dell'oggetto a cui manda i pacchetti questo Agent. Di solito è l'oggetto routing del nodo.

La classe Agent implementa delle funzioni che servono a settare e reperire i valori delle variabili descritte sopra, connettere due Agent e ad aggiungere un'applicazione all'Agent:

- *addApplication(Application *)*: aggiunge un'applicazione all'Agent;
- *getAddress()*: ritorna l'indirizzo dell'Agent;
- *setAddress(int a)*: setta l'indirizzo dell'Agent;
- *getRAddress()*: ritorna l'indirizzo dell'Agent remoto;
- *setRAddress(int a)*: setta l'indirizzo dell'Agent remoto;
- *getPort()*: ritorna la porta associata all'Agent;
- *setPort(int p)*: setta la porta dell'Agent;
- *getRPort()*: ritorna la porta associata all'Agent remoto;
- *setRPort(int p)*: setta la porta dell'Agent Remoto;
- *getPackSize()*: ritorna la dimensione del pacchetto che questo Agent gestisce;
- *setPackSize(int s)*: setta la dimensione del pacchetto che questo Agent manda;
- *setConnection(int cid,int sid,int rsid)*: setta i parametri di connessione dell'Agent.

La classe Agent definisce delle funzioni virtuali che devono essere implementate dalle classi che la derivano:

- *virtual void start()*: funzione che serve a far partire l'Agent;
- *virtual void stop()* : funzione che serve a fermare l'Agent;
- *virtual void handle(Event *)*: esecuzione di un evento rispetto a questo SObject;
- *virtual void recv(Event *)*: riceve un pacchetto da un'altro SObject (di solito il demux);
- *virtual void setHeader()*: setta i valori dell'header del pacchetto durante la simulazione;
- *virtual int type()*: il tipo di SObject;
- *virtual void printResult()*: stampa i risultati della simulazione di questa applicazione;

5.8.1 UdpAgent

La simulazione del protocollo Udp in Pdnet si realizza tramite due oggetti **UdpAgent** di cui uno è il mandante e l'altro il ricevente. Questi Agent comunicano con oggetti Application (unica implementazione CountApplication) che simulano l'applicazione associata ad essi.
La simulazione di questa applicazione parte tramite un EVENTSTART riferito all'UdpAgent mandante che esegue la funzione *start()* che non fa altro che eseguire la sua funzione *handle()*. La funzione *handle()* come prima cosa controlla se continuare a mandare pacchetti oppure no. Ci sono due casi in cui la funzione stoppa:

- se viene settato un numero totale di pacchetti da mandare e sono già stati mandati;
- se viene schedulato un evento EVENTSTOP che all'esecuzione chiama la funzione *stop()* che setta la variabile *state_* uguale a zero (condizione di uscita).

Se invece l'invio dei pacchetti deve continuare procede nel seguente modo:

- richiede un pacchetto dall'applicazione (identificato tramite l'evento);
- setta i campi degli header del pacchetto Udp e IP ;
- setta la dimensione del pacchetto;
- manda il pacchetto all'oggetto del nodo con cui è collegato (di solito oggetto routing) e schedula se stesso tra *interval_* time.

Quando un pacchetto raggiunge l'oggetto UdpAgent ricevente, tramite la sua funzione *recv()*, questo oggetto manda il pacchetto all'Application che è stata associata ad esso tramite la funzione *recv()* di Application.
Due funzioni importanti della classe UdpAgent che vengono utilizzate per settare i parametri dell'UdpAgent mandante sono:

- *void setInterval(double i)*: setta l'intervallo tra la spedizione di un pacchetto e il successivo;
- *void setTotPack(int tp)* : setta il numero totale di pacchetti da mandare.

5.8.2 TcpAgent e TcpAgentListener

TcpAgent e **TcpAgentListener** sono gli oggetti utilizzati per la simulazione di applicazioni **TCP**. Il protocollo TCP non è trattato in questo elaborato perchè va al di là dello scopo di questo progetto. L'unica cosa che si deve introdurre sono i timer di ritrasmissione perchè hanno un peso rilevante nella schedulazione e cancellazione di eventi.
Quando TcpAgent manda un pacchetto fa partire un timer:

- se il timer supera un certo tempo rimanda il pacchetto;
- se arriva la risposta prima della scadenza del timer, viene mandato un nuovo pacchetto e resettato il timer.

Il timer utilizzato da TcpAgent utilizza un algoritmo dovuto a Jacobson(1988) e funziona come segue. Per ogni, connessione TCP mantiene una variabile, RTT, che rappresenta la migliore stima attuale del tempo di round-trip per la destinazione in questione. Quando viene inviato un segmento si avvia un Timer che serve a due scopi: per sapere quanto tempo richiede l'acknowledgement e per innescare un'eventuale ritrasmissione. Se l'acknowledgement torna indietro prima della scadenza del timer, TCP misura il tempo richiesto, che chiameremo M. RTT viene così aggiornato secondo la formula

$$RTT = \alpha RTT + (1 - \alpha)M$$

dove α è un fattore di perequazione che determina il peso dato al vecchio valore. Generalmente corrisponde a 7/8.
Anche con valore di RTT valido, la scelta di un timeout di ritrasmissione adatto è una questione complessa. Di norma, TCP utilizza βRTT, ma la parte complessa sta nello scegliere β. Nelle implementazioni iniziali β era sempre 2, ma l'esperienza ha insegnato che un valore costante era inflessibile perchè non rispondeva al cambiamento della varianza. Nel 1988 Jacobson ha proposto di rendere β proporzionale alla deviazione standard della funzione di densità di probabilità relativa al tempo di arrivo dell'acknowledgement, in modo che una varianza grande producesse un valore β elevato e viceversa. In particolare, ha suggerito l'utilizzo della *deviazione media* come stima approssimata della *deviazione standard*. Il suo algoritmo richiede di tenere traccia di un'altra variabile perequata, vale a dire la deviazione D. Al ricevimento di un acknowledgement è elaborata la differenza tra i valori previsto e osservato, $|RTT - M|$. Un valore perequato di questo risultato è inserito in D con la formula

$$D = \alpha D + (1 - \alpha)|RTT - M|$$

dove α può essere essere o meno lo stesso valore utilizzato per perequare RTT. Anche se D non equivale esattamente alla deviazione standard, è un'approssimazione sufficiente; Jacobson ha mostrato come può essere elaborato utilizzando solo addizioni intere, sottrazioni e scorrimenti. La maggior parte delle implementazioni TCP ora utilizza questo algoritmo e imposta l'intervallo di timeout a

$$Timeout = RTT + 4 * D$$

La scelta del fattore 4 è arbitraria, ma presenta due vantaggi. Innanzi tutto, la moltiplicazione per 4 può essere eseguita con un solo scorrimento. In secondo luogo, riduce i timeout e le ritrasmissioni inutili perchè meno dell'1% dei pacchetti giunge in un tempo superiore a quattro volte la deviazione standard. In realtà Jacobson inizialmente proponeva di utilizzare 2, ma studi successivi hanno dimostrato che 4 genera prestazioni migliori.
Per la realizzazione di un Timer non vengono utilizzati oggetti particolari come in ns. In Pdnet il timer è definito all'interno della classe Agent e gli eventi che simulano la scadenza del timeout sono schedulati con riferimento all'Agent stesso. La classe TcpAgent è caratterizzata da uno stato interno che comprende oltre ai campi utili per le connessioni Tcp anche i valori che caratterizzano lo stato del timer:

- *seq_* : numero di sequenza da inviare;
- *rtt_* : round-trip timer;
- *mtime_* : tempo in cui sono settati i valori di timeout;
- *crono_* : tempo che passa dall'invio del pacchetto alla risposta;
- *sdev_* : deviazione standard;
- *timeout_* : valore di timeout;
- *spack_* : numero di pacchetti mandati;
- *state_* : stato della connessione TCP;
- *rack_* : ack ricevuti;
- *rduplex_* : ack duplicati ricevuti.

TcpAgent utilizza le seguenti funzioni per settare i timeout e mandare i pacchetti:

- *setTimeout()*: esegue l'algoritmo di Jacobson per il calcolo del timeout;
- *timeout()*: fa partire un nuovo timeout e cancella un eventuale timeout pendente;
- *sendPack()*: manda un pacchetto a TcpAgentListener.

TcpAgentListener riceve i pacchetti mandati da TcpAgent tramite la funzione *recv()* che cambia il suo stato interno e spedisce un ack al mittente. Lo stato interno di questi oggetti è definito dalle seguenti variabili:

- *ack_* : sequenza del pacchetto ricevuta;
- *sack_* : numero di ack mandati;
- *rpack_* : numero pacchetti ricevuti;
- *duplex_* : numero duplicati ricevuti;
- *state_* : stato della connessione TCP.

Quando viene invocata la funzione *start()* (tramite un evento EVENTSTART) di TcpAgent, viene inizializzata la connessione tcp, vengono mandati i pacchetti ad TcpAgentListener che risponde con gli ack di risposta come descritto nel protocollo TCP standard. Quando un TcpAgent deve mandare un pacchetto fà le seguenti operazioni:

- setta i campi relativi agli header IP e TCP;
- setta la dimensione del pacchetto;
- manda il pacchetto e schedula se stesso ad un tempo pari al timeout calcolato.

Se viene eseguito l'evento associato all'Agent vuol dire che la risposta non è ancora arrivata entro il tempo di timeout: spedisce un pacchetto duplicato e rischedula un nuovo evento di timeout (rischedula se stesso). Invece se arriva la risposta cerca l'evento associato con il timeout e lo toglie dalla coda di priorità (coda degli eventi), manda un nuovo pacchetto e rischedula un nuovo evento di timeout. Infine quando viene invocata la funzione *stop()* di TcpAgent (tramite un evento EVENTSTOP) simula la chiusura del TCP e termina.

5.9 Application

La classe *Application* è un Object che serve come classe base per l'implementazione di applicazioni. Questo Object è inserito in alcuni tipi di Agent (SObject) e simula la gestione dei protocolli superiori a quelli di trasporto. La classe Application è una classe virtuale utilizzata come classe base per le applicazioni e definisce le seguenti funzioni:

- *void addHeader(int ip,int tr)*: serve a settare i parametri che identificano gli header da utilizzare nel pacchetto nei livelli ip e di trasporto. Questo settaggio serve perchè l'Agent richiederà un Packet (tramite un evento) ed in ogni Packet quando è creato verrà definito l'header tramite questi valori (questa operazione può essere fatta anche nell'Agent).
- *virtual void recv(Event *)*: funzione implementata dalla classe derivata che serve per ricevere un Evento;
- *void send(Event *)*: funzione implementata dalla classe derivata che serve per mandare un Evento;
- *virtual int getNextPack(Event *)*: funzione implementata dalla classe derivata chiamata dagli Agent per ricevere un pacchetto dall'Application;
- *getSpack()* e *getRpack()* ritornano i valori delle variabili associate ai pacchetti mandati e ricevuti;
- *setSpack(int s)* e *setRpack(int s)* settano i valori delle variabili associate ai pacchetti mandati e ricevuti;

In questa versione del simulatore è implementata solo un'Application, la **countApplication** che non fa altro che contare il numero di pacchetti mandati e ricevuti.

5.10 Creare una simulazione in Pdnet

L'esecuzione di **Pdnet** esegue i seguenti passi:

- inizializza il simulatore;
- inizializza la struttura di simulazione (*topologia, tempo di simulazione, scheduler utilizzato, routing*);
- esegue la simulazione;

- restituisce i risultati.

Per creare una simulazione in **Pdnet** si procede creando una struttura nella funzione *topology* della classe **Pdnet** nel seguente modo:

1. si definisce lo scheduler da utilizzare tramite la funzione *Pdnet::schedType(char *schedType)*;

2. si creano i nodi con la funzione *Pdnet::addNode()* che ritorna il cid del nodo creato;

3. si creano i link tra i nodi tramite la funzione *Pdnet::addSimplexLink(int src,int dst,double bw,double delay,int qtype,int qsize)* dove *src* è il cid del nodo sorgente, *dst* il cid del nodo destinazione, *bw* la banda del link, *delay* il ritardo del link, *qtype* la coda da utilizzare (unica implementazione DROPTAIL), *qsize* la dimensione della coda e ritorna la coppia di valori *cid*, *sid* corrispondente al link creato;

4. si creano gli Agent, si settano eventuali parametri, si creano eventuali applicazioni che si devono utilizzare e si attaccano agli Agent, si attaccano gli Agent ai nodi, si connettono tra loro gli Agent e si schedulano gli eventi di partenza dell'invio di pacchetti ed eventuali eventi di stop;

```
/*creazione applicazione udp*/

UdpAgent *agent1 = new UdpAgent();
UdpAgent *agent2 = new UdpAgent();
agent1 -> setPackSize(400);
agent1 -> setInterval(0.007);
agent1 -> setTotPack(100);
countApplication *c1 = new countApplication("agent1");
agent1 -> addApplication(c1);
SOBJ a1 = attackAgent(n0,agent1);
SOBJ a2 = attackAgent(n5,agent2);
connection(agent1,agent2);
atSchedule(a1.cid,a1.sid,EVENTSTART,0.0);
atSchedule(a1.cid,a1.sid,EVENTSTOP,100000.0);

/*creazione applicazione tcp*/

TcpAgent *tcp1 = new TcpAgent();
TcpAgentListener *tcpl1 = new TcpAgentListener();
```

```
SOBJ tagent1 = attackAgent(n0,tcp1);
attackAgent(n5,tcpl1);
connection(tcp1,tcpl1);
atSchedule(tagent1.cid,tagent1.sid,EVENTSTART,0.0);
atSchedule(tagent1.cid,tagent1.sid,EVENTSTOP,100000.0);
```

5. si inizializza il tempo virtuale della durata della simulazione tramite la funzione *simTime(double time)*.

Capitolo 6

Scheduler parallelo in Pdnet

6.1 Impacchettamento e spacchettamento dei messaggi remoti

Nella progettazione di scheduler paralleli si devono ideare dei meccanismi di sincronizzazione. Per utilizzare le tecniche di sincronizzazione gli LP devono comunicare tra loro. I meccanismi di comunicazione in Pdnet utilizzano le librerie MPI che forniscono un ambiente ideato per le parallelizzazioni.
I messaggi che gli LP si scambiano durante una simulazione sono composti da variabili con tipi diversi. Per rendere semplice e trasparente l'impacchettamento e lo spacchettamento dei messaggi di vario tipo si è creata un'interfaccia tra la libreria MPI e gli oggetti di Pdnet .
La classe **CommBuffer** è stata implementata a questo scopo ed caratterizzata dalle seguenti variabili:

- *char *mBuffer*: il buffer allocato;
- *int mBufferSize*: la size del buffer;
- *int mMsgSize*: la posizione del buffer per la prossima allocazione;
- *int mPosition*: la posizione del buffer per la deallocazione.

Le funzioni *pack()* e *unpack()* servono a impacchettare e spacchettare un dato. In questa versione di Pdnet sono implementati i tipi di dato int, unsigned int, double, unsigned double e string.

Il seguente codice descrive le funzioni per impacchettare e spacchettare una stringa:

```
// pack una stringa
int CommBuffer::pack(const unsigned char *d,int len)
{
    if(len == 0)
        return pack(len);
    int ret = pack(len);
    if (ret != MPI_SUCCESS)
        return ret;
    extendBuffer(len*sizeof(char));
    return MPI_Pack((void *)d, len,
           MPI_UNSIGNED_CHAR, mBuffer,
           mBufferSize, &mMsgSize, MPI_COMM_WORLD);
}

// unpack a string
int CommBuffer::unpack(const unsigned char *d,int len)
{
    int ret;
    if((ret = MPI_Unpack(mBuffer, mMsgSize,
        &mPosition,(void *) d,len,
        MPI_UNSIGNED_CHAR, MPI_COMM_WORLD) != MPI_SUCCESS))
        return ret;
    return MPI_SUCCESS;
}
```

6.2 Una tecnica di parallelizzazione centralizzata

Questo paragrafo parla della tecnica di parallelizzazione sperimentata in Pdnet.
Un sistema ha uno stato identificato dall'insieme delle variabili di stato e da una coda di eventi. Questo stato può essere modificato con l'esecuzione di un evento. L'esecuzione di un evento dipende da un insieme di variabili di input (subset delle variabili di stato), genera un output che modifica un subset di variabili di stato e modifica la coda degli eventi. Sia $S_i = (v_0, v_1, v_2, v_3)$ l'insieme delle variabili di stato di un sistema ad un determinato tempo di

simulazione. Si suppone che con l'esecuzione dell'evento *event* lo stato delle variabili del sistema cambi in $S_{i+1} = (v_0, v_1^1, v_2, v_3)$. Come si può notare l'evento *event* ha modificato solo una variabile del sistema. Se si ha un evento successivo che come variabili di input non dipende da v_1 può essere eseguito simultaneamente ad *event*. Questa idea parte dal presupposto che in un sistema con uno stato contenente un gran numero di variabili, ogni evento modificherà solo un sottoinsieme di variabili che rispetto al numero totale è minimo. Quindi si può avere una buona probabilità che eseguendo un evento l'output generato non andrà a toccare l'insieme di variabili di input dell'evento successivo. Se questi due eventi non modificano le variabili di input di un terzo evento allora sono tre gli eventi che possono essere eseguiti in contemporanea etc. In poche parole se un evento ha il subset delle varibili di stato di input che non viene modificato da eventi precedenti la sua esecuzione può essere anticipata ed essere eseguito in parallelo con gli eventi precedenti. Partendo da questo concetto si realizza l'algoritmo parallelo tramite un meccanismo "centralizzato". Si suppone ci siano n LP identificati come LP 0, LP 1 ..... LP n-1, dove l'LP 0 è il "master" e gli altri LP sono gli "slave". Il master ha una coda di eventi da eseguire (*Event Code EC*) e una coda di eventi eseguiti dagli slave ma non ancora verificati dal master per stabilire se sono validi (*Conditional Event Code CEC*). Gli LP slave hanno una coda di eventi inizialmente vuota. Quando inizia la simulazione il master prende i primi n eventi da EC (oppure tutti gli eventi che ci sono se il numero di eventi in coda è minore di n) e li divide nel seguente modo:

- il primo evento lo tiene per se;
- gli altri n-1 li manda uno per ogni slave. Ad ogni slave oltre all'evento viene mandato lo stato attuale del sistema. Il master tiene traccia di questi n-1 eventi inserendoli in CEC con il relativo stato di input.

Dopo questa suddivisione il master esegue il primo evento che è sicuramente corretto perchè essendo il primo non ha nessuna dipendenza da altri eventi e gli slave eseguono gli eventi assegnati che invece non è detto che siano corretti.
Dopo l'esecuzione gli slave mandano i risultati all'LP master. I risultati comprendono un nuovo stato del sistema ed eventuali eventi futuri da eseguire. Il master accoppia gli output ricevuti dagli slave con gli eventi relativi in CEC. Dopo il primo passo il sistema si trova in una situazione in cui la CEC può essere non vuota e il master ad ogni passo successivo al primo farà la seguente operazione: controlla se il prossimo evento da eseguire si trova in EC oppure in CEC:

- se si trova in EC ripete l'operazione, descritta sopra, di divisione negli LP degli eventi in EC;
- se si trova in CEC controlla se l'evento che è stato eseguito da uno slave è valido. Per la verifica della validità controlla se il sottoinsieme di variabili di input da cui dipende l'evento è stato modificato da qualche evento precedente.
  - se l'evento è valido ripristina lo stato del sistema in base all'output corrispondente e schedula eventuali eventi generati;
  - se l'evento non è valido "butta via" il lavoro svolto dallo slave e lo rischedula nella coda EC degli eventi da eseguire.

Ricapitolando abbiamo un master che esegue ciclicamente le operazioni descritte sopra e degli slave che si mettono in attesa di ricevere un "lavoro" da svolgere dal master. Quando gli slave ricevono il messaggio aggiornano il proprio stato, svolgono il lavoro assegnato e restituiscono il risultato al master a cui toccherà il compito di capire se è un "lavoro" valido oppure no. In poche parole il "cervello" della simulazione è il master che coordina il lavoro, ne esegue una parte e stabilisce se il lavoro eseguito dagli slave è valido oppure no, mentre gli slave non "pensano" ma eseguono il lavoro a loro assegnato senza capirne l'utilità (*Figura 6.1*).

Per implementare un algoritmo del genere bisogna capire bene il ruolo delle

Figura 6.1: Strategia centralizzata

variabili di stato del sistema e cercare per ogni evento di estrarre il sottoinsieme di variabili che lo riguarda. In alcuni modelli di simulazione trovare le

variabili "certe" che riguardano l'evento non è semplice. Per risolvere questo problema si adotta la seguente strategia:

- si prende in considerazione un sottoinsieme di variabili che "potrebbero interessare" l'esecuzione dell'evento (sottoinsieme condizionato) e si conservano come variabili di input;
- dopo l'esecuzione dell'evento si ha la certezza di quali siano le variabili del sottoinsieme scelto che servono per il controllo. Nell'output si indicano quali sono in modo che il controllo è fatto rispetto a queste variabili che è di numero uguale o inferiore al sottoinsieme ipotizzato.

La strategia descritta sopra è utilizzata per aumentare la probabilità del numero di eventi validi.
Un'altra ottimizzazione dell'algoritmo può essere fatta per l'aggiornamento degli slave. Infatti è inutile aggiornare lo stato globale dello slave quando possiamo aggiornare un ristretto numero di variabili. Se si riesce a definire un sottoinsieme di possibili variabili che "potrebbero interessare" l'esecuzione dell'evento si aggiornano solo quelle. Quest'ultima ottimizzazione oltre a diminuire il tempo computazionale per le operazioni di aggiornamento dello stato degli slave serve a diminuire la lunghezza dei messaggi per la comunicazione.

6.3 Implementazione dello Scheduler parallelo centralizzato in Pdnet

Lo scheduler parallelo è implementato in Pdnet tramite la classe **ParScheduler** che eredita la classe **HeapScheduler** definita per l'implementazione dell'heap scheduler.

```
class HeapScheduler: public Scheduler
{
    protected :
        Heap *h_;
        FILE *fdebug;
    public:
        HeapScheduler() { h_= new Heap();}
        ~HeapScheduler() {delete h_;}
        void run();
        void insert(Event *);
        void cancel(double);
```

```
        Event *deque();
        Event *head();

};


class ParScheduler: public HeapScheduler
{
    private:
        Heap *hpar_;
        int idcluster_;//Lp corrente
        int ncluster_;//numero LP totali
    public:
        ParScheduler() {hpar_ = new Heap();}
        ~ParScheduler(){delete hpar_;}
        void run();
        void cancel(double);
        Event *head();
        int Recv(CommBuffer *,int *);
        void Send(int cl,CommBuffer *);
        void stop(int cl);
};
```

Le strutture dati utilizzate per l'implementazione delle code di priorità degli eventi sono l'heap *h_*, definito nella classe HeapScheduler, e l'heap *hpar_* definito nella classe ParScheduler. L'heap *h_* è utilizzato per simulare la EC (Event Code) mentre *hpar_* è utilizzato per simulare la CEC (Conditional Event Code). La EC è la coda di priorità degli eventi che devono essere eseguiti. La CEC è la coda di priorità degli eventi eseguiti dagli slave ma non ancora verificati dal master. Le funzioni implementate in Parscheduler, oltre a quella di routine (*run()*) e di cancellazione degli eventi (*cancel()*), comprendono le funzioni *Send()* e *Recv()* che servono per mandare e ricevere messaggi dal master agli slave e viceversa e la funzione *stop()* che serve al master per comunicare agli slave che la simulazione è terminata. L'esecuzione della funzione di routine (*run()*) inizia con l'assegnazione del ruolo degli LP nel modello. L'LP con id 0 è il master mentre gli altri LP sono gli slave. In fase di inizializzazione i processi slave svuotano la coda di priorità *h_* e si mettono in attesa di un messaggio dal master. I messaggi che possono ricevere sono di due tipi:

- *routine*: un messaggio del genere è composto da un evento da eseguire e dall'insieme delle variabili di stato del nodo a cui si riferisce l'even-

to. Lo slave quando riceve questo tipo di messaggio aggiorna lo stato del nodo, esegue l'evento e crea un messaggio di risposta. Il messaggio di risposta è composto dai riferimenti agli SObject del nodo che compongono lo stato di input dell'evento, dal nuovo stato del nodo dopo l'esecuzione dell'evento e dai nuovi eventi che sono stati schedulati dall'evento. Infine lo slave manda il messaggio di risposta al master;

- *stop*: lo slave esce dalla simulazione.

Il ciclo di routine è eseguito dal master e ad ogni iterazione controlla se il prossimo evento da eseguire si trova nella coda $h\_$ oppure in $hpar\_$. Nel primo caso fà le seguenti operazioni:

- controlla se il tempo dell'evento da eseguire è minore o uguale al tempo di simulazione previsto (altrimenti esce dal ciclo);
- estrae il primo evento dalla coda $h\_$ e lo memorizza come evento e;
- per ogni slave estrae un evento dalla coda $h\_$ (se il numero degli eventi in $h\_$ è minore del numero degli slave li estrae tutti) ed esegue i seguenti passi:
  - crea un oggetto CommBuffer;
  - inserisce i dati che caratterizzano l'evento nel CommBuffer;
  - inserisce le variabili di stato relative al container a cui l'evento si riferisce nel CommBuffer (sottoinsieme condizionato);
  - manda il CommBuffer allo slave corrispondente tramite la funzione *Send()*;
  - memorizza il CommBuffer come input dell'evento;
  - tiene traccia dell'evento memorizzando l'uid in un vettore che tiene traccia degli eventi mandati agli slave;
  - inserisce l'evento in $hpar\_$.
- esegue l'evento e;
- riceve la risposta dagli slave;
- ricerca l'evento corrispondente in $hpar\_$ tramite l'uid memorizzato nel vettore che tiene traccia degli eventi e memorizza l'output ricevuto in corrispondenza dell'evento.

Nel secondo caso invece fà le seguenti operazioni:

- estrae l'evento da *hpar_* e controlla se le variabili di input dell'evento sono state modificate da un evento precedente. Se sono state cambiate inserisce l'evento in *h_*. Se non sono state cambiate cambia lo stato del sistema in base ai valori di output ricevuti e schedula gli eventi generati dall'evento eseguito dallo slave.

La metodologia utilizzata per interagire con lo stato del sistema è implementata tramite delle funzioni della classe **Container**, raggiunte tramite l'istanza **Pdnet** indicando il numero del Container. La classe Container interagisce con gli SObject contenuti in essa tramite delle funzioni definite virtualmente nella classe **SObject** ed implementate in ogni oggetto SObject. Questo meccanismo implica che ogni SObject creato deve essere predisposto per supportare l'aggiornamento, il reperimento e il controllo del suo stato. Un altro meccanismo poteva essere quello di memorizzare le variabili di stato in unico vettore ma essendo i tipi delle variabili variegate si è preferito non utilizzare questo ultimo metodo.
Le funzioni virtuali per la gestione dello stato delle componenti di Pdnet da implementare per ogni SObject sono le seguenti:

- *void supdate(CommBuffer *)*: memorizza le variabili di stato nel Commbuffer;

- *void supdate2(CommBuffer *c)*: in linea di principio uguale a *supdate()* ma differisce nel caso che oltre alla memorizzazione nel buffer deve essere fatta qualche altra operazione. Viene chiamata quando si deve ottenere un aggiornamento negli slave.

- *void rupdate(CommBuffer *)* : ripristina le variabili di stato tramite il CommBuffer;

- *void rupdate2(CommBuffer *c)*: come *rupdate()* ma differisce nel caso di operazioni particolari. Viene chiamata per l'aggiornamento dello stato dell'SObject nel master;

- *int isUpdate()*: controlla se lo stato viene aggiornanto dopo l'esecuzione dell'evento. Utilizzata dagli slave per capire quali sono gli SObject interessati dall'evento eseguito;

- *int control(CommBuffer *input)*: controlla se lo stato dell'SObject è valido confrontando i dati contenuti nel CommBuffer di input e le sue variabili;

- *void removeInput(CommBuffer *input)*: rimuove le variabili di input dal CommBuffer. Questa funzione serve quando avviene il controllo

e si capisce che le variabili di un SObject memorizzato in input non servono per il controllo.

6.4 Performance e valutazioni dell'esperimento

Questo paragrafo presenta le valutazioni sullo scheduler parallelo centralizzato in Pdnet con l'ausilio di un modello di esempio. I risultati sono calcolati con l'utilizzo di due, tre e quattro LP.
La prima valutazione riguarda la percentuale di eventi parallelizzabili, la dimensione media delle code locale e remota, la percentuale di eventi eseguiti da ogni LP. La seconda valutazione riguarda l'impatto temporale che si ha nell'esecuzione del modello di esempio con lo scheduler parallelo rispetto all'esecuzione con l'heap scheduler seriale.
In base ai risultati delle simulazioni si traggono delle conclusioni che indicano quando conviene utilizzare questa tecnica di parallelizzazione e le possibili ottimizzazioni che possono essere utili per dei casi di studio futuri.

6.4.1 Modello di esempio per l'esperimento

Il modello di esempio implementato per verificare il funzionamento e le performance dello scheduler parallelo centralizzato in Pdnet è descritto in *figura 6.2*. Questo modello è composto da tredici nodi indicizzati come $n_0, n_1 ..... n_{12}$ connessi tramite dei duplex-link. La *tabella 6.1* riporta le caratteristiche dei duplex-link presenti nel modello. I duplex-link sono rappresentati come due simplex-link con direzioni opposte che hanno le stesse caratteristiche. I campi della tabella dei link rappresentano rispettivamente i nodi coinvolti nel collegamento, la banda dei link, i delay dei link e la dimensione massima delle code associate ai link.
Le applicazioni eseguite in questa topologia di rete sono di tipo udp e sono descritte nella *tabella 6.2*. I campi di questa tabella rappresentano rispettivamente il nodo sorgente e il nodo destinazione dove sono allocati gli Agent dell'applicazione, la dimensione dei pacchetti gestiti dall'applicazione, l'intervallo di tempo che passa tra l'invio di un pacchetto da parte dell'Agent mandante e il successivo, il tempo virtuale di start dell'applicazione e il suo tempo virtuale di stop.
La simulazione è eseguita per un tempo virtuale di 100 secondi.

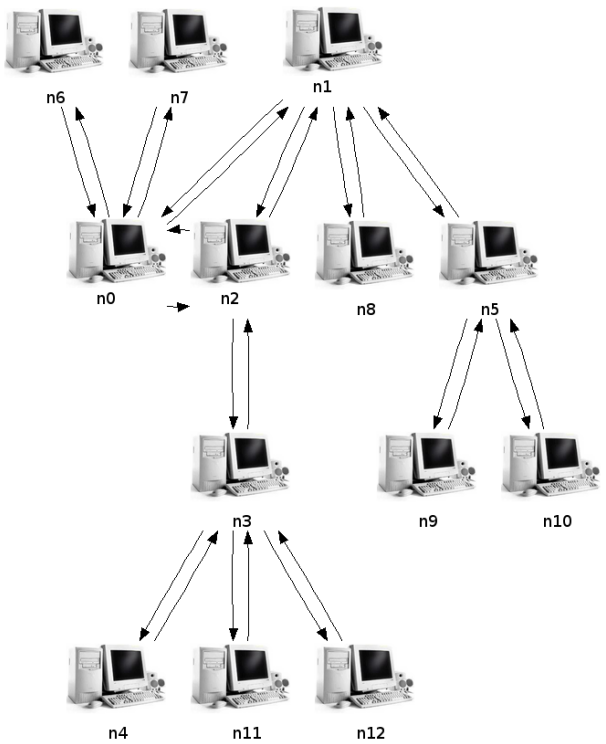

Figura 6.2: Modello di esempio

nodo1	nodo2	bw	delay	code size
n_0	n_1	1 Mb	5 ms	100
n_1	n_2	1 Mb	5 ms	100
n_0	n_2	2 Mb	5 ms	100
n_1	n_5	1 Mb	5 ms	100
n_0	n_6	0.5 Mb	5 ms	100
n_0	n_7	0.5 Mb	5 ms	100
n_1	n_8	0.7 Mb	4 ms	100
n_5	n_9	1 Mb	6 ms	100
n_5	n_{10}	1 Mb	6 ms	100
n_2	n_3	2 Mb	5 ms	100
n_3	n_{12}	1.5 Mb	4 ms	100
n_3	n_{11}	0.8 Mb	3 ms	100
n_3	n_4	0.8 Mb	6 ms	100

Tabella 6.1: Link modello di esempio

src	dst	pack size	interval	start	stop
n_6	n_8	400 B	7 ms	0 s	100 s
n_7	n_{11}	200 B	8 ms	0 s	100 s
n_9	n_{12}	100 B	1 ms	0 s	100 s
n_4	n_{10}	120 B	5 ms	0 s	100 s
n_{11}	n_{12}	200 B	1 ms	0 s	100 s
n_{10}	n_7	200 B	4 ms	0 s	100 s

Tabella 6.2: Applicazioni modello di esempio

6.4.2 Bilancio sugli eventi parallelizzabili

Per analizzare gli eventi parallelizzabili per prima cosa si analizza il carico di lavoro di ogni LP nelle simulazioni effettuate. Il carico di lavoro è rappresentato dall'*istogramma della figura 6.3*. Le ascisse di questo istogramma

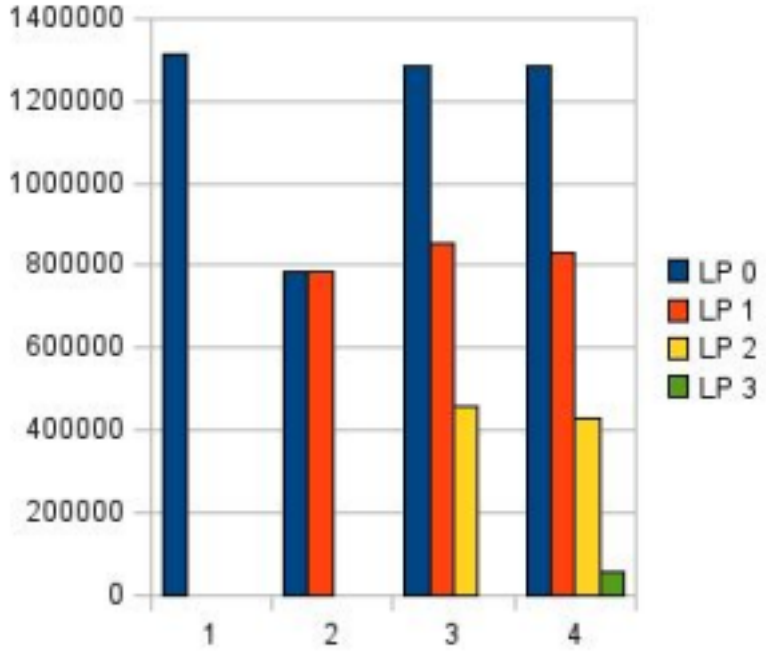

Figura 6.3: Eventi eseguiti dagli LP nel modello di esempio

indicano il numero di LP utilizzati dalla simulazione. Le ordinate indicano il numero di eventi eseguiti da ogni LP. Un secondo dato da analizzare è il numero di eventi totale eseguito dalle simulazioni (*tabella 6.3*). Da questi risultati la prima cosa che si nota è che l'esecuzione con quattro LP non ha

numero LP	eventi totali eseguiti
1	1314623
2	1567505
3	2685804
4	2595805

Tabella 6.3: Eventi totali eseguiti dalle simulazioni nel modello di esempio

senso perchè l'ultimo LP esegue un numero minimo di eventi. Gli LP dal 3 in poi sono quasi inutilizzati. Nell'esecuzione con tre LP si nota che l'LP 0 esegue un numero di eventi leggermente inferiore a quello seriale, l'LP 1 un numero di eventi considerevole mentre l'LP 2 un numero di eventi basso. La somma totale del numero di eventi eseguiti è di molto superiore a quella seriale. Questo fa presupporre che il numero di eventi che vengono rieseguiti è molto alto. Nell'esecuzione con due LP si vede che il numero di eventi eseguiti dai due LP è pressocchè uguale. La somma totale del numero di eventi eseguiti è di poco superiore a quella seriale. Questo fa presupporre ad un numero basso di eventi rieseguiti.
Le presuppozioni fatte sopra vengono confermate dall'analisi degli eventi eseguiti negli slave che risultano validi (remote positive). Le percentuali descritte in *tabella 6.4* ci danno la conferma che l'esecuzione con due LP ha un fattore di parallelizzazione soddisfacente mentre con più di due LP il fattore di parallelizzazione non è accettabile. Un'altra osservazione importante è

numero LP	% remote positive
2	67.7%
3	2.5%
4	2.5%

Tabella 6.4: Percentuale remote positive nel modello di esempio

quella che indica la lunghezza media delle code (*tabella 6.5*) in cui EC è la coda degli eventi da eseguire e CEC la coda degli eventi eseguiti dagli slave ma non ancora verificati dal master.

Il campo della tabella "size EC" rappresenta il numero di eventi medio in EC durante la simulazione e il campo "size CEC" il numero di eventi medio in CEC. I valori ci indicano che con due LP mediamente c'è un solo evento in CEC mentre con più di due LP CEC è molto grande. Questo implica che con più di due LP se gli eventi sono quasi tutti in CEC la EC ha pochi eventi e molto spesso nei cicli di routine viene eseguito un solo evento, e in molti

numero LP	size EC	size CEC
2	457	1
3	7	451
4	7	451

Tabella 6.5: Lunghezza media code nel modello di esempio

altri casi solo due. Il risultato del numero di eventi eseguiti da ogni LP nelle simulazioni ci dà la conferma. Un'altra implicazione è che con un numero di LP maggiore di due la maggior parte degli eventi elaborati dagli slave vengono eseguiti molto in anticipo e la condizione di validità non è quasi mai verificata.
In conclusione questo algoritmo risulta efficiente solo se viene eseguito con due LP.

6.4.3 Impatto temporale sulla simulazione

Dall'analisi fatta sul numero di LP da utilizzare per una simulazione è risultato che con più di due LP l'algoritmo non risulta efficiente. Di conseguenza la simulazione per verificare l'impatto temporale della parallelizzazione è stata fatta solamente con due LP è il tempo di esecuzione viene confrontato con il tempo di simulazione dell'heap scheduler seriale.
La macchina utilizzata per la simulazione ha un processore Intel(R) Core(TM)2 CPU T5300 1.73GHz e un 1 Gbyte di ram.
Il tempo di esecuzione per eseguire il modello di esempio con l'utilizzo dello scheduler parallelo centralizzato e di circa 65 volte più alto dal tempo richiesto per l'esecuzione del modello di esempio con l'heap scheduler seriale.

6.4.4 Considerazioni sui risultati

L'analisi dei risultati ci porta alla conclusione che la tecnica di parallelizzazione implementata in Pdnet produce una percentuale di eventi parallelizzabili soddisfacente solo con due LP. I tempi di esecuzione non sono soddisfacenti ma si deve tenere conto che ogni evento in Pdnet ha un tempo di esecuzione basso. Questo implica che i tempi di latenza per lo scambio dei messaggi sono superiori ai tempi di esecuzione di un singolo evento. Con l'implementazione di nuove strutture in Pdnet se gli eventi avranno un costo computazionale più alto, in modo che i tempi di latenza per lo scambio dei messaggi risultino trascurabili rispetto ai tempi di esecuzione dell'evento, questa tecnica potrà essere utilizzata.

Un'ottimizzazione di questa tecnica è quella di far eseguire ad ogni LP un gruppo di eventi per ogni ciclo. Per la realizzazione di questa ottimizzazione il master deve trovare le dipendenze che gli eventi hanno tra loro prima dell'esecuzione in modo che gli eventi assegnati ad LP diversi siano slegati (non hanno dipendenze tra loro). Questa ottimizzazione serve a ridurre il numero di messaggi tra il master e gli slave.
In conclusione l'utilizzo della tecnica di simulazione parallela sperimentata dipenderà molto dall'evoluzione futura di Pdnet. Questa tecnica può essere applicata ad altri tipi di simulatori che hanno tempi di esecuzione di un singolo evento elevati oppure in cui sia semplice trovare le dipendenze degli eventi in anticipo rispetto alla loro esecuzione in modo da applicare l'ottimizzazione descritta prima.

Capitolo 7

Conclusioni

Nel campo della ricerca le sperimentazioni possono avere dei costi elevati per la realizzazione degli ambienti che servono per gli esperimenti. I simulatori sono nati con lo scopo di creare virtualmente gli ambienti necessari per gli esperimenti e diminuirne i costi. La ricerca nel campo delle network utilizza i simulatori di rete multiprotocollo per analizzare il comportamento di nuovi protocolli applicazioni o sistemi, per convalidare il comportamento di sistemi già esistenti e stimarne le performance. L'utilizzo delle simulazioni è molto utile per avere una panoramica chiara sul funzionamento dei sistemi prima della loro reale implementazione.
Le simulazioni effettuate nei laboratori di ricerca molte volte hanno un costo computazionale elevato che comporta un lento svolgimento dell'analisi dei dati. L'idea è quella di diminuire drasticamente i tempi di simulazione applicando ai simulatori delle tecniche di schedulazione parallela che sfruttano la potenza di calcolo dei moderni sistemi multiprocessore.
Per studiare il funzionamento dei simulatori si è preso come riferimento ns che utilizza degli scheduler seriali di tipo DES con meccanismo event-driven. La creazione di scheduler paralleli in ns è risultata complicata (in alcuni casi addirittura impossibile) per via della sua struttura. A tal proposito è nato Pdnet, un simulatore di reti simile ad Ns ma strutturato in modo che l'implementazione di scheduler paralleli risulti semplice. In questa versione del simulatore sono state implementate solo le strutture basilari di una rete perchè lo scopo dell'esperimento è quello di testare uno scheduler parallelo in esso.
Lo scheduler parallelo implementato in Pdnet utilizza una tecnica di parallelizzazione centralizzata in cui un LP (Local Process) master gestisce i meccanismi per la parallelizzazione degli eventi. Il master ha una lista di eventi da eseguire ordinata in base al tempo virtuale di esecuzione e il suo compito è quello di dividere gli eventi negli LP coinvolti nella simulazione

parallela in modo che un gruppo di eventi venga eseguito nello stesso momento. Ogni LP slave dopo l'esecuzione dell'evento restituisce il risultato al master che ne verifica la validità. Più eventi validi si verificano migliore è il fattore di parallelizzazione. L'analisi delle performance ci porta alla conclusione che la tecnica di parallelizzazione utilizzata in Pdnet presenta un buon fattore di parallelizzazione quando viene eseguita con due LP. I tempi di esecuzione non sono soddisfacenti ma si deve tenere conto che ogni evento in Pdnet ha un tempo di esecuzione basso. Questo implica che i tempi di latenza per lo scambio dei messaggi sono superiori ai tempi di esecuzione di un singolo evento. Con l'implementazione di nuove strutture in Pdnet se gli eventi avranno un costo computazionale più alto, in modo che i tempi di latenza per lo scambio dei messaggi risultino trascurabili rispetto ai tempi di esecuzione dell'evento, questa tecnica potrà essere utilizzata.
Un'ottimizzazione di questa tecnica è quella di far eseguire ad ogni LP un gruppo di eventi per ogni ciclo. Per la realizzazione di questa ottimizzazione il master deve trovare le dipendenze che gli eventi hanno tra loro prima dell'esecuzione in modo che gli eventi assegnati ad LP diversi siano slegati (non hanno dipendenze tra loro). Questa ottimizzazione serve a ridurre il numero di messaggi tra il master e gli slave. Questo può essere un caso di studio futuro.
Altri casi di studio possono essere le tecniche descritte nel capitolo 3 visto che la struttura di Pdnet ne permette una facile implementazione.
Per utilizzare concretamente Pdnet nell'ambito della ricerca si devono trovare tecniche di parallelizzazione che hanno una velocità di esecuzione più bassa rispetto a quella seriale, si devono creare le strutture per svariati tipi di network (seguendo le rigide regole di implementazione) e si deve creare un'interfaccia per l'iterazione con l'utente (grafica o tramite linguaggi di script).

Bibliografia

[1] Wikipedia enciclopedia libera *http://www.wikipedia.org*

[2] IBM *rs/6000 SP:Pratical MPI Programming* Agosto 1999

[3] A.Udaya Shankar *Discrete-event-simulation* Departement of computer scienze University of Marylan. Gennaio 1991

[4] *Ns Manual.* UC Berkelay,LBL,USC/ISI,and Xeseros PARK. Kevin Fall, Kannan Varhadan. 28 febbraio 2008.

[5] *Performance Evalutation of Conservative Algorithms in Parallel Simulation Language* Rajive L. Bagrodia - Mineo Takai - Vikas Jha

[6] *Asynchronous Parallel Discrete Event Simulation.* Yi-Bing Lin - Paul A. Fishwick.

[7] Tenembaum *Reti di Calcolatori 4 edizione* Vrije Universiteit Amsterdam, Olanda.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest *Introduzione agli Algoritmi*

[9] Fujimoto R.M *Time Warp on a Shared Memory Multiprocessor.*Transaction of the society for computer simulation,1989.

[10] Fujimoto R.M *Parallel Discrete Event Simulation.*