SAPIENZA Università di Roma
Facoltà di Ingegneria

Tesina per il corso di Metodi formali nell'ingegneria del software

Anno accademico 2006/2007

Modellazione del sistema di comunicazione della rete ethernet: il protocollo CSMA/CD e verifica delle sue proprietà temporali attraverso SPIN e LTL

Vanda Piacentini, Germano Rocco

Relatore: Prof. Toni Mancini

Indice

Capitolo 1

Introduzione

Lo scopo della tesina è quello di effettuare il reverse engineering del sistema di comunicazione di una rete Ethernet: il protocollo CSMA/CD(Carrier Sense Multiple Access with Collision Detection), attraverso diagrammi degli stati e delle transizioni UML.

In seguito passeremo alla codifica dei diagrammi realizzati in moduli Spin e alla verifica di tutte le possibili proprietà temporali del sistema attraverso l'uso della logica temporale lineare proposizionale.

Ethernet è un protocollo dello strato di collegamento della pila ISO, la variante basata su supporto BNC usa un solo cavo coassiale per collegare decine di stazioni di lavoro, ciascuna delle quali riceve contemporaneamente tutti i dati che attraversano la rete, al contrario solo una stazione alla volta ha la facoltà di trasmettere.

Ogni stazione infatti è indipendente e non esiste un coordinatore centrale. Le informazioni sono trasmesse sotto forma d'impulsi che si propagano a partire dalla stazione emittente verso i due estremi della rete fino a raggiungere il punto in cui il cavo termina.

Ogni messaggio in transito sulla rete (detto anche trama o frame, poichè è composto da una sequenza di bit tra loro combinati) incapsula l'indirizzo di origine e quello di destinazione: di conseguenza ogni macchina lo copia in una piccola porzione di memoria (buffer) di cui dispone la scheda di rete, confronta l'indirizzo di destinazione con il proprio, e, se diverso lo scarta.

Tuttavia con questo approccio, non è possibile evitare il fenomeno delle collisioni sia a causa dei tempi di propagazione non nulli, sia per la lunghezza delle trame trasmesse.

Per la gestione delle collisioni Ethernet usa il protocollo CSMA/CD. Esso è un protocollo distribuito privo di master (quindi operante in modo paritario su tutte le macchine della LAN) che permette alle stazioni di condividere l'utilizzo del mezzo trasmissivo. Il protocollo, essendo di tipo ad accesso

casuale al mezzo, non esclude il verificarsi di collisioni; prevede quindi un meccanismo di riconoscimento delle collisioni da parte delle stazioni coinvolte, in modo che esse possano ritentare la trasmissione in un tempo successivo.

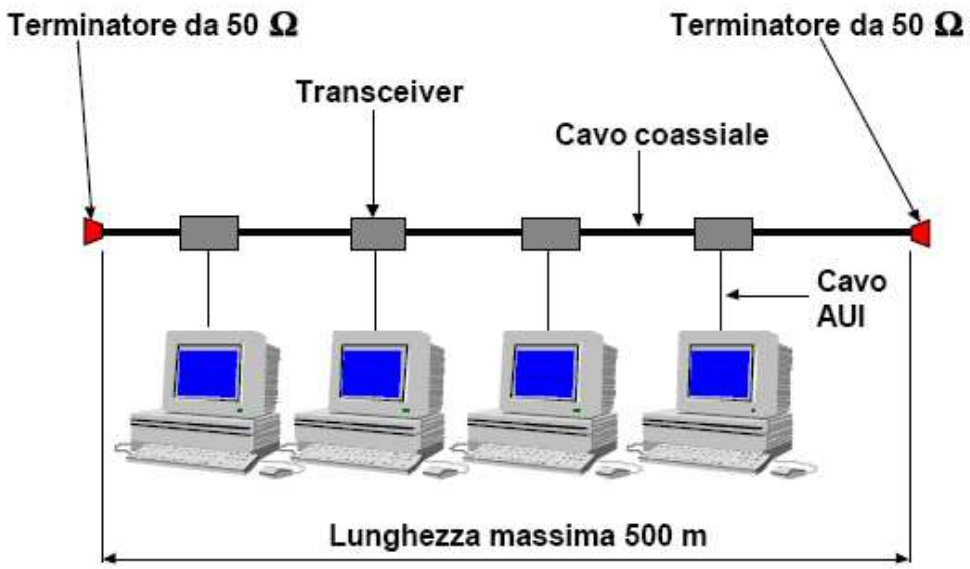
Capitolo 2

Il sistema fisico

La specifica standard di livello fisico di questo tipo di connessione fisica è denominata 10Base-T, dove il valore T indica le centinaia di metri che può essere lungo il cavo coassiale.

Gli apparati di rete ed i computer (genericamente denominati stazioni) si collegano al cavo coassiale tramite un Transceiver che trasmette e riceve i dati tra il cavo e l'interfaccia di rete.

Il segmento Ethernet 10Base-5 rappresentato nella Figura 1 è costituito da un cavo coassiale Thick-Ethernet terminato alle due estremità tramite degli appositi terminatori da 50 Ω. La comunicazione sul cavo coassiale è naturalmente broadcast in quanto, quando una stazione invia dei dati, essi (tramite il transceiver) vengono trasmessi sul cavo e tutte le altre stazioni connesse su questo ricevono i dati.



2.1 Come funziona il protocollo CSMA/CD

Supponiamo che la stazione A abbia dei dati da spedire:

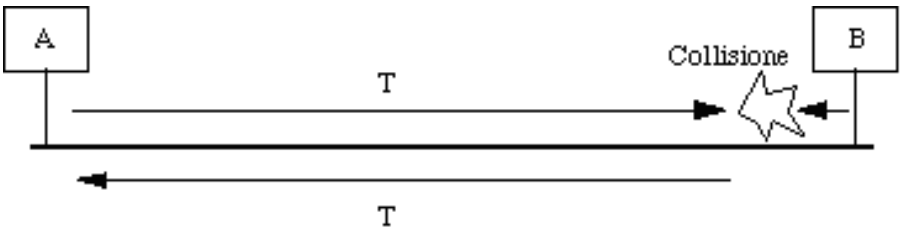
1. la stazione A ascolta il canale
   - se esso è libero (cioè se non ci sono altre stazioni che stanno trasmettendo) si inizia a spedire il messaggio;
   - se il canale è occupato, si attende un tempo casuale prima di riprovare la trasmissione
2. mentre si trasmette la stazione A monitora la rete (è questo il vero e proprio Collision Detection)
   - se non si rileva un cambiamento di energia sul canale in un tempo l + T (ove T è il tempo che impiega il messaggio per attraversare la lunghezza del canale di comunicazione e l tempo di trasmissione) si considera il frame spedito;
   - se l'energia presente sul canale è cambiata, è avvenuta una collisione. La stazione A interrompe la trasmissione e avvisa tutte le stazioni in ascolto (le quali non sanno che la trasmissione ha subito una collisione) con un segnale di disturbo (JAM);
3. Dopo aver abortito la trasmissione, si esegue l'algoritmo di back-off esponenziale, prima di riprovare la trasmissione.

2.2 Problematiche spazio-temporali

Il tempo di propagazione del canale svolge un ruolo importante nella performance del protocollo CSMA/CD. In particolare se una stazione A al tempo t_0 inizia a trasmettere ci sarà un tempo di propagazione pari a T prima che le altre stazioni percepiscano la trasmissione di A; così nell'intervallo $[t_0 ; t_0 + T]$ può accadere che il canale sia percepito libero dalle altre stazioni, che possono, quindi, iniziare una trasmissione generando una collisione.

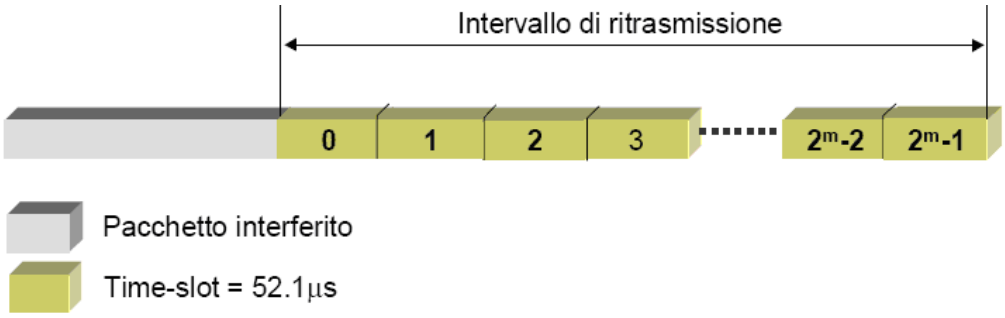
Infatti, se una stazione A posta ad una estremità della rete inizia a trasmettere al tempo t_0, il suo segnale arriva a B (posta all'altra estremità della rete) dopo, al tempo t_0 + T; se un attimo prima di tale istante anche B inizia a trasmettere, la collisione conseguente viene rilevata da B quasi immediatamente, ma impiega una ulteriore quantità T di tempo per giungere ad A, che, di conseguenza, la può rilevare solo un attimo prima dell'istante t_0 + 2T.

Quindi, dopo una collisione, una stazione attende almeno un tempo 2T prima di riprovare. Se non si verificano collisioni e, indicando con l il tempo di trasmissione di un messaggio, un messaggio sarà completamente consegnato in un tempo pari a l + T.



2.3 Caratteristiche dell'algoritmo di *Backoff-esponenziale*

Nel caso di collisione tra due o più stazioni, il protocollo CSMA/CD sceglie in modo casuale l'istante di ritrasmissione per ciascuna stazione utilizzando l'algoritmo di Back-Off. Dopo una collisione il tempo è diviso in slot discreti di durata uguale al round-trip delay 2T e quindi uguale a 52,1 μS; in questo modo in un time slot possono essere trasmessi 512 bit (e quindi un pacchetto di lunghezza minima).



Ogni stazione sceglie in modo casuale la slot in cui iniziare la ritrasmissione tra lo slot 0 (posto alla fine del proprio messaggio interferito) e lo slot $2^m - 1$, con m intero. La scelta di m è importante in quanto può influenzare prestazioni del sistema in modo significativo: se m è troppo piccolo, la probabilità di nuove collisioni può essere elevata, mentre se m è grande risulta elevato il ritardo introdotto dalle ritrasmissioni. L'algoritmo di Back-Off esponenziale sceglie m in modo adattivo, a seconda del numero n di collisioni che un pacchetto ha subito; in particolare:

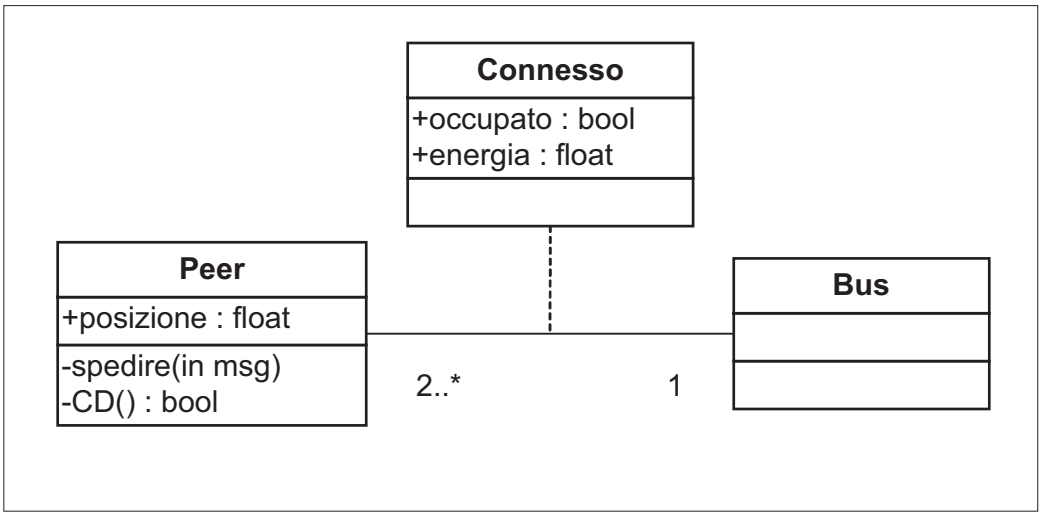
- se $n \leq 10$ si pone $m = n$;
- se $10 < n < 16$ si pone $m = 10$;
- dopo 16 tentativi senza successo la trasmissione viene abortita.

Capitolo 3

Modellazione del sistema

3.1 Diagramma UML

I componenti fondamentali del sistema sono le classi *Peer*, *Bus*, e l'associazione *Connesso*.



E' necessaria l'aggiunta di un vincolo esterno per eliminare la possibilità dell'esistenza di due Bus:

$$\forall x \forall y \text{ Bus(x)} \bigwedge \text{Bus(y)} \Longrightarrow \text{x} = \text{y}$$

Il precedente vincolo, espresso in logica del primo ordine (FOL) asserisce che per ogni coppia di Bus, essi si riferiscono allo stesso oggetto.

Il Peer possiede i seguenti metodi:

- **spedire(msg)**: permette la trasmissione del messaggio *msg* sul bus.
- **CD()**: permette il monitoraggio della rete per la rilevazione di eventuali collisioni. Restituisce il valore TRUE nel caso sia stata rilevata una collisione, FALSE altrimenti.

e la seguente proprietà:

- **posizione**: indica a che altezza del bus è connesso il *Peer*, assumiamo che il cavo sia lungo 300 metri, di conseguenza il valore posizione appartiene all'intervallo {0 ; 300}.

La proprietà *posizione* è necessaria per modellare il ritardo di propagazione introdotto dal bus, esso è pari a:

$$\frac{|A.posizione - B.posizione|}{V}$$

Dove *A.posizione* e *B.posizione* rappresentano rispettivamente la posizione di A e di B sul bus (assumiamo che A intenda trasmettere e che e B sia in attesa di ricevere dati). V è la velocità di propagazione del segnale elettrico nel cavo, ottenibile mediante la formula:

$$\mathrm{V} = \frac{c}{\sqrt{\varepsilon\mu}}$$

dove c è la velocità della luce nel vuoto pari a 299,8 $\times$ 10^8 m/s, ε costante dielettrica e μ permeabilità magnetica del cavo coassiale. Dalle precedenti assunzioni deriva il significato del valore di 52,1μS di ogni slot temporale, esso rappresenta il doppio del tempo che impiega il segnale per percorrere interamente il cavo (Round Trip Time) della lunghezza di 300 metri.

L'associazione *Connesso* connette ogni sender al bus. Le cardinalità indicano che devono esserci almeno due stazioni (2..*) mentre il bus è unico (1..1). Essa possiede le seguenti proprietà:

- **occupato**: valore booleano il quale indica se è presente una trasmissione sul bus iniziata da un'altra stazione.
- **energia**: valore corrispondente all'energia istantanea presente sul bus, necessario per la rilevazioni delle collisioni.

L'attributo *occupato* è usato per il *CARRIER SENSE*, cioè, l'ascolto del bus prima di iniziare la trasmissione per accertarsi che non sia già in corso una trasmissione.
L'attributo *energia* è invece usato per il *COLLISION DETECTION*: se durante la trasmissione tale valore risulta incrementato rispetto alla precedente

misurazione, è segno che si è verificata una collisione, poichè un'altro sender ha iniziato una trasmissione immettendo la propria energia sul bus.

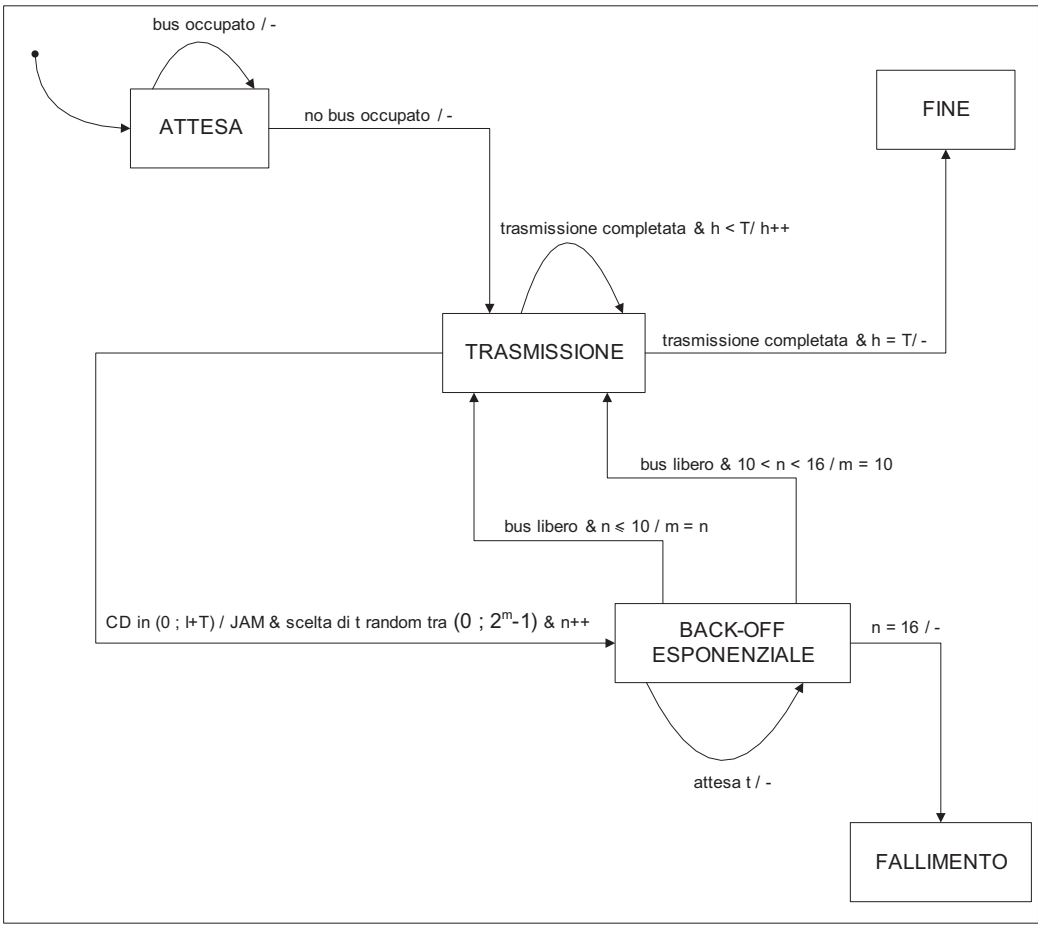
La scelta di attribuire all'associazione *Connesso* i due attributi *occupato* ed *energia* piuttosto che alla classe Bus, deriva dall'assuzione che il ritardo di propagazione fa in modo che lo stato del Bus può apparire diverso tra i vari Peer.

Se *occupato* ed *energia* fossero stati attribuiti al Bus, non sarebbe stato necessario il protocollo del CSMA/CD, poichè tutti i Peer avrebbero avuto la corretta concezione dello stato del Bus.

3.2 Il peer

Il Peer è il componente principale del sistema, esso implementa il protocollo del CSMA/CD. Come detto in precedenza, esso non caratterizza tutta l'interfaccia di rete, ma solo la parte che si occupa della trasmissione e della gestione di eventuali collisioni (viene quindi omesso per semplicità il modulo per la ricezione di messaggi da altre stazioni).

3.2.1 Diagramma degli stati e transizioni



Gli stati

Dallo studio del sistema è emerso che i possibili stati del peer sono: *Attesa*, *Trasmissione*, *Back-off esponenziale*, *Fine* e *Fallimento*. Di seguito si suppone che $t_0 = 0$ (istante di inizio di una trasmissione). Nel momento in cui si

vuole iniziare una trasmissione il Peer passa nello stato di *Attesa*, e vi rimane finchè non percepisce libero il bus.

Nel momento in cui il bus viene percepito libero (variabile occupato = FALSE) il Peer si porta nello stato di *Trasmissione*: qui inizia la trasmissione, durante la quale monitora il bus per captare eventuali collisioni (funzione CD()).

Il Peer rimane in questo stato per un tempo pari a l (durata della trasmissione), più un'ulteriore tempo T, (tempo di propagazione del segnale). Se non si verificano collisioni nell'intervallo (0 ; l + T), il Peer si porta nello stato di *Fine*, notificando al layer superiore (strato di Rete) della pila ISO OSI che la trasmissione è stata completata con successo.
Se invece, nell'intervallo (0; l + T) viene percepita una collisione, si passa nello stato di *Back-off esponenziale*, scegliendo lo slot (tempo di attesa) tra 1 e 2^m e inviando il segnale di JAM.
La scelta di usare l'intervallo (1 ; 2^m) invece di quello proposto in precedenza (0 ; $2^m - 1$) deriva da problematiche imposte dal modello che, come vedremo tra poco, l'attesa viene effettuata ciclando un tempo pari a 2T×t (se avessimo usato l'intervallo standard, t poteva anche essumere il valore nullo, e quindi il vincolo imposto dall'attesa minima di 2T non sarebbe stato rispettato).
Successivamente il Peer si porta nello stato di *Back-off esponenziale*, e si attende un tempo pari a 2T×t. Dopodichè si verifica se lo stato del bus è libero o occupato: nel primo caso ci si porta nuovamente nello stato di *Trasmissione*, se invece il bus risulta occupato, si attende che esso si liberi. In entrambe i casi si setta opportunamente la variabile m in base al numero di volte (round) che si è verificata una collisione durante la trasmissione. Se invece n > 16, cioè più di 16 tentativi senza successo, la trasmissione viene abortita e il Peer si porta nello stato di *Fallimento*.

Le transizioni

Seguono le transizioni del Peer, composte dalla coppia *Evento/Azione*.

bus occupato / -

- il bus è occupato: è cioè in corso un'altra trasmissione, di conseguenza il Peer aspetta (ciclando a tempo indeterminato nello stato di attesa) che esso diventi libero;
- *Evento* bus occupato;
- *Azione* azione nulla;

no bus occupato / -

- il bus viene percepito libero dal Peer, esso si porta quindi nello stato di *Trasmissione*;
- *Evento* bus libero;
- *Azione* azione nulla;

trasmissione completata $\wedge$ h $<$ T/ -

- nel momento in cui la trasmissione viene completata (assumiamo in un tempo l), il Peer deve aspettare un ulteriore tempo T per permettere la propagazione sul bus: questo viene ottenuto mediante un cappio sullo stato di *Trasmissione*
- *Evento* la trasmissione completata è stata completata e la variabile h $<$ T;
- *Azione* azione nulla;

trasmissione completata $\wedge$ h = T/ -

- terminata l'attesa di propagazione sul bus, e se non sono state rilevate collisioni, la trasmissione viene ritenuta corretta, di conseguenza il Peer si porta nello stato di *Fine*;
- *Evento* la trasmissione è stata completata e il tempo di attesa di propagazione è scaduto;
- *Azione* si notifica il layer superiore (strato di Rete) che la trasmissione è stata completata con successo;

CD in (0 ; l + T) / JAM $\bigwedge$ scelta di t random tra (1; 2^m) $\bigwedge$ n++

- se viene rilevata una collisione durante la trasmissione o durante l'attesa di propagazione, il Peer interrompe la trasmissione, segue l'invio del segnale di disturbo JAM, la scelta in in modo casuale (random) dello slot da cui ritentare la ritrasmissione in un'intervallo che in base al round (numero di volte che si rileva una collisione) viene aumentato in modo esponenziale base 2; viene incrementato il valore del numero di round e successivamentei l Peer si porta nello stato di *Back-off esponenziale*;
- *Evento* collisione in (1 ; l + T);
- *Azione* invio segnale di disturbo, scelta randomica di t dell'intervallo (1; 2^m) e incremento del valore del round corrente;

attesa t * 2T / -

- fase di attesa esponenziale: come detto in precendeza, ogni slot rappresenta il doppio del tempo di propagazione e cioè 2T. Il cappio presente sullo stato di *Riprocessamento* fa in modo di attendere un tempo pari a 2T moltiplicato per il numero t (slot) scelto nella fase di *Back-off esponenziale*
- *Evento* attesa;
- *Azione* azione nulla;

bus libero $\wedge$ n$\leq$ 10 / m = n

- terminata l'attesa il meccanismo del backoff-esponenziale prevede la ritrasmissione, il Peer si accerta quindi che il bus sia libero (se non lo è si attende che esso lo diventi), e se il numero di round è n$\leq$10 si ritenta la trasmissione, portandosi quindi nello stato di *Trasmissione*;
- *Evento* bus libero e numero di round n minore o uguale a 10;
- *Azione* si pone m = n;

bus libero $\wedge$ (10 < n < 16) / m = 10

- come sopra;
- *Evento* bus libero e numero di round n compreso tra 10 e 16;
- *Azione* si pone m = 10;

n = 16 / -

- se il numero di volte che si ritenta la trasmissione diviene pari a 16, essa viene annullata e il Peer si porta nello stato di *Fallimento*;
- *Evento* n = 16;
- *Azione* si notifica al layer superiore (strato di Rete) che la trasmissione è stata annullata;

3.3 Il bus

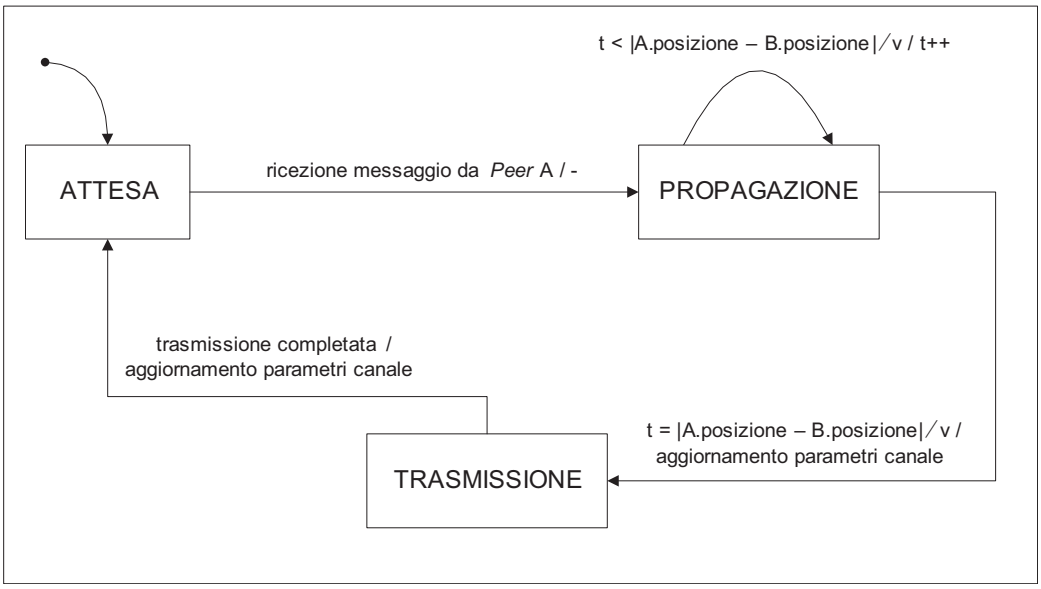
La scelta di modellare il Bus come un'entità attiva è da attribuirsi al suo comportamento, in quanto introduce un ritardo di propagazione non trascurabile.

Come già detto, la natura del modello impone che i Peer evolvono in maniera autonoma; inoltre, permodellare il ritardo di propagazione, si fa in modo che i Peer possano vedere il sistema in modo differente (alcuni lo rileveranno come occupato, mentre altri libero).

Essendo il Bus unico nel modello, esso è da vedere come un sistema *multi-thread*, dove per ogni Peer connesso al Bus e per ogni nuova trasmissione, viene istanziato un nuovo thread.

E' da notare che, nel caso in cui due Peer posti alle estremità del bus inizino, nello stesso momento, la trasmissione, i Peer posti al centro del bus possono essere interessati da due trasmissioni contemporaneamente (corrisponderanno quindi due thread). Assumendo quindi n processi e k trasmissioni, il numero di thread è pari a k(n-1).

3.3.1 Diagramma degli stati e transizioni



Gli stati

Gli stati del bus sono: *Attesa*, *Propagazione* e *Trasmissione*. Nel momento in cui un Peer inizia una nuova trasmissione, viene creato un thread per ogni Peer (tranne che per il sender) relativo a quella trasmissione. Tale thread passa quindi dallo stato di *Attesa* a quello di *Propagazione*. In questo stato si gestisce il ritardo di propagazione, ciclando un determinato numero di volte in base alla distanza tra Peer sender (A.distanza) e Peer receiver (B.distanza) e usando la formula vista precedentemente.

Una volta che il segnale si è propagato al Peer in esame, esso rileva il bus come occupato (C.occupato = TRUE) e viene aggiornato il valore dell'energia presente sul bus (C.energia). A questo punto si passa nello stato di *Trasmissione*: qui i dati vengono inviati al Peer in esame. Ultimata la trasmissione, vengono opportunamente aggiornate le variabili *occupato* ed

energia (il segnale elettrico si è attenuato), dopodichè si passa nello stato di *Attesa*, in cui tale thread viene eliminato.

Le transizioni

Seguono le transizioni del Bus, composte dalla coppia *Evento/Azione*. Assumiamo che A sia il Peer che inizia la trasmissione, e B il processo per cui viene creato il thread in esame.

ricezione messaggio da Peer A / -

- il Peer A inizia una nuova trasmissione,deve essere quindi creato un thread per gestire la propagazione verso il processo B
- *Evento* ricezione trasmissione da A;
- *Azione* azione nulla;

t < |A.posizione - B.posizione| ÷ v / t++

- questa transizione è un cappio sullo stato di Propagazione, e rappresenta l'attesa di propagazione del segnale, calcolata in base alla distanza tra terminale sender e terminare receiver, usando la formula vista precedentemente;
- *Evento* cicla finchè il valore del contatore t è < della formula. Nel momento in cui tale valore diventa lo stesso della formula, esce dallo stato di *Propagazione*, portandosi in trasmissione
- *Azione* incremento il contatore t;

t = |A.posizione - B.posizione| ÷ v / aggiornamento parametri bus

- il segnale propagandosi ha raggiunto il Peer B, di conseguenza può essere avviata la trasmissione. Inoltre il terminale B giudica correttamente lo stato del bus: devono, quindi, essere aggiornate le variabili relative allo stato: occupato ed energia (e); in particolare, quest'ultima si assumerà essere nota a priori, e uguale per ogni Peer (nella realtà l'energia di trasmissione è la stessa per ogni interfaccia ethernet).
- *Evento* il contatore t ha raggiunto il valore della formula di propagazione;
- *Azione* vengono aggiornate le variabili: C.occupato = TRUE, C.energia += e;

trasmissione completata / aggiornamento parametri bus

- completata la trasmissione tale thread viene eliminato;
- *Evento* trasmissione completata;
- *Azione* i parametri del bus vengono aggiornati.
  Si pone C.energia -= e, e se essa vale 0, il bus viene considerato esente da trasmissioni: C.occupato = FALSE;

Capitolo 4

Modellazione in SPIN

Per analizzare dal punto di vista formale il sistema visto nei capitoli precedenti si fa uso di SPIN. Il suo acronimo sta per *Simple Promela INterpreter*, ed è sostanzialmente un verificatore di teoremi basato sul linguaggio *PROMELA* (il cui acronimo è *PROcess MEta LAnguage*.

L'uso che se ne fa è focalizzato sulla prova di correttezza di sistemi composti da processi asincroni che interagiscono tra loro. Tale interazione può essere specificata in SPIN mediante primitive di tipo *rendezvous*.

I processi comunicano tra loro attraverso canali di tipo FIFO, variabili condivise o tramite una loro combinazione.

Il primo passo nella verifica del sistema è quello di descriverne la sua *specifica formale*. Nel nostro modello, essa viene descritta utilizzando il linguaggio PROMELA. Viene costruito un'automa a stati finiti (ASF) che caratterizza completamente il modello; su di esso verranno effettuate tutte le verifiche di correttezza.

La correttezza del programma è espressa mediante una formula in *logica temporale lineare proposizionale* (LTL). Oltre alla funzionalità di *model checker*, SPIN offre la possibilità di effettuare la simulazione del sistema per analizzarne la sua evoluzione.

4.1 Studio del modello

La scelta di usare SPIN per studiare il protocollo CSMA/CD è stata suggerita da alcuni fattori. Innanzitutto la possibilità di esprimere il protocollo CSMA/CD in un sistema a stati finiti, e la focalizzazione che ha SPIN verso lo studio di sistemi concorrenti. Tuttavia durante l'analisi sono sorti vari problemi: il piu importante è stato quello della sostanziale differenza che vi è tra sistemi concorrenti che operano a livello applicativo (algoritmo di

Ricart-Agrawala per la mutua esclusione in un sistema distribuito) e quelli (come appunto CSMA/CD) che operano a livello di collegamento.

SPIN offre varie primitive per lo scambio di singoli messaggi. Mediante studi approfonditi del sistema è emerso che singoli messaggi non astraggono perfettamente il concetto di trasmissione di un segnale elettrico. Il motivo sostanziale è che il segnale è un valore continuo che ha una certa durata (anche se il dato viene digitalizzato, sul bus è comunque presente un segnale continuo), mentre il messaggio di SPIN è un'oggetto discreto senza una durata definita.

A priori, non è, quindi, possibile implementare il protocollo del CSMA/CD, proprio perchè vengono a mancare i requisiti base per la sua realizzazione. La soluzione a questo problema è quella di rappresentare una trasmissione non come un singolo messaggio, ma come una raffica di messaggi. In questo modo una trasmissione può avere una lunghezza variabile.

Un'ulteriore problema è la mancanza in SPIN di costrutti fondamentali come la funzione per generare numeri casualmente (randomize) e il timer regolabile. Per risolvere la mancanza del randomize è stato creato un'apposito stato per la scelta di un numero in modo non-deterministico:

```
RANDOM:
    do
        :: num = 1; goto NEXT;
        :: num = 2; goto NEXT;
        :: num = 3; goto NEXT;
        :: num = 4; goto NEXT:
    od;
```

nel momento in cui il sistema si porta nello stato RANDOM, grazie al non determinismo, la probabilità di scegliere un numero rispetto ad un'altro è pari ad 1/4. E' stata quindi realizzata la funzione random(n) con n intero (nell'esempio n = 4). Con questo approccio è possibile realizzare funzioni random di lunghezza limitata unicamente dalla dimensione del file input di SPIN. Per ovviare alla mancanza in SPIN di un timer regolabile, si è usato il costrutto *timeout*. Una volta invocato timeout, il processo attende un tempo fisso (non impostabile), dopodichè riprende l'esecuzione dallo stato in cui era stato precedentemente messo in attesa. L'idea è stata quella di iterare un certo numero di volte il timeout, in modo da creare una sorta di timer regolabile in base al numero di iterazioni. L'implementazione è la seguente:

```
WAIT:
    do
```

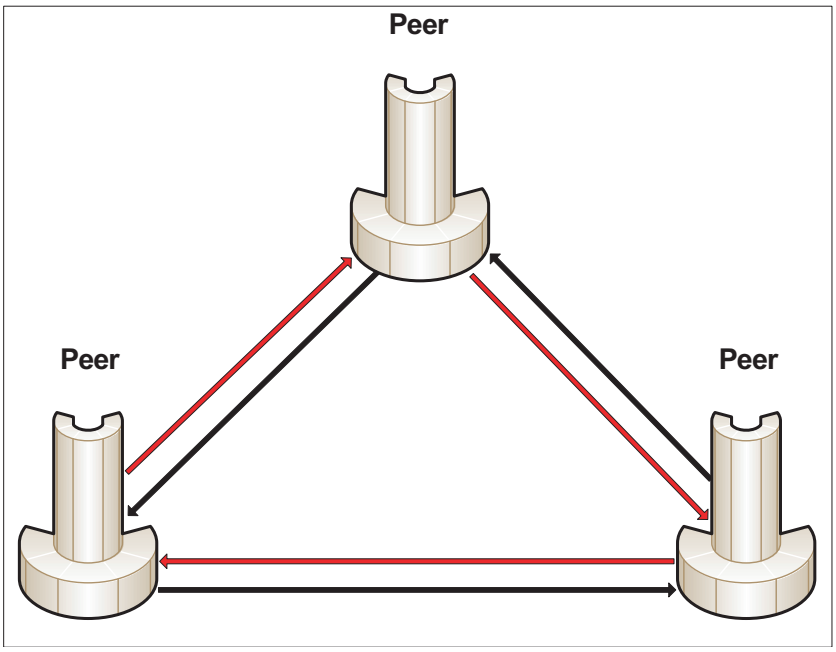
```
        ::  nr < slot -> timeout; nr = nr + 1;
        ::  nr == slot -> nr = 0; goto NEXT;
    od;
```

finchè la variabile nr è minore di slot, si attende mediante l'uso di timeout, altrimenti il processo si porta nel prossimo stato. Con questo approccio si è realizzata la funzione timer(slot). Utilizzando la sintassi del linguaggio C la funzione assume la forma:

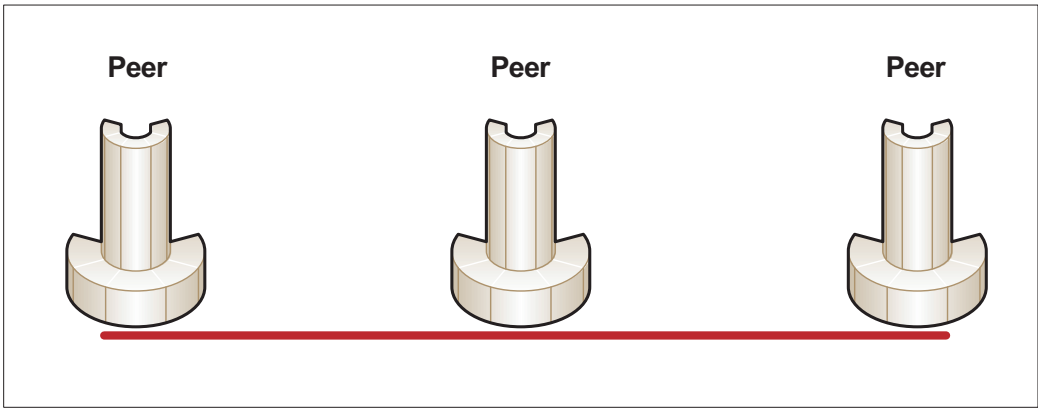
```
for(nr = 0 ; nr < slot ; nr++)
    timeout();
```

4.2 Descrizione del modello

Analizzando i requisiti, sono stati valutati vari approcci per modellare il sistema. Tutti questi contemplavano l'idea di caratterizzare i vari *Peer* come processi distinti, i quali comunicano mediante lo scambio di messaggi. La differenza tra i vari approcci è la metodologia di comunicazione, che può essere diretta o indiretta. La comunicazione diretta si suddivide a sua volta in due sottocategorie, la prima assume che i processi appartengano ad una rete completamente interconnessa:



tuttavia essa non astrae quella che è la struttura del modello, e cioè tutti i terminali connessi tramite un bus comune. La seconda sottocategoria prevede l'uso di un solo canale di comunicazione comune. Nel momento in cui un processo intende trasmettere, invia i messaggi direttamente in questo canale. La struttura è la seguente:

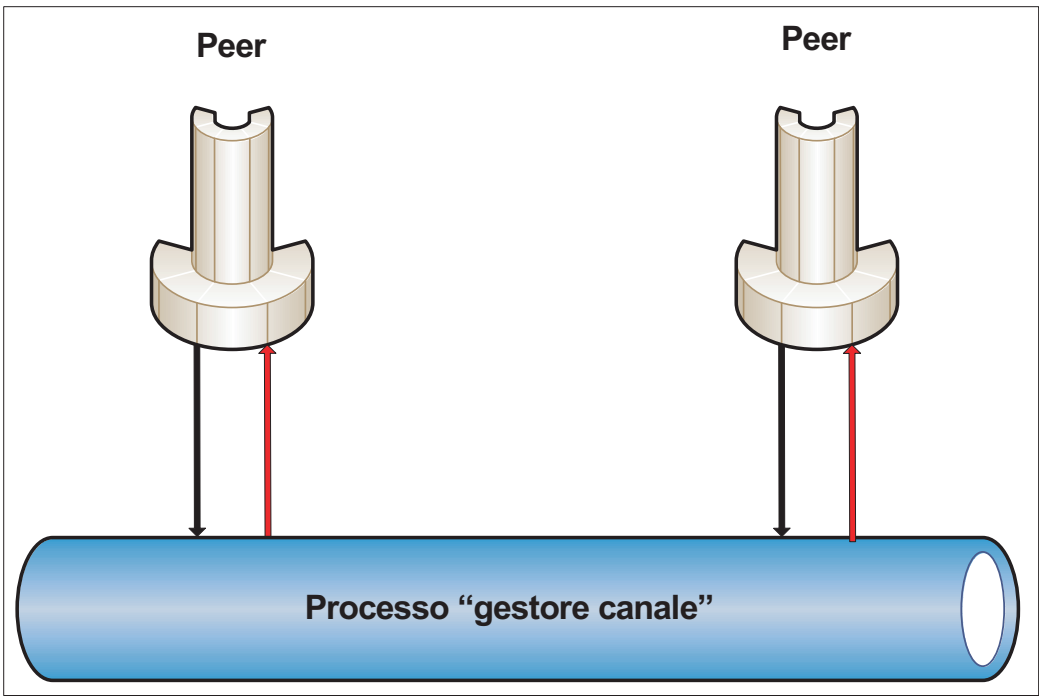


Tale soluzione sembra rispecchiare il sistema, ma sorgono una serie di problemi. Il primo tra tutti è da attriburisi alla natura del Bus: esso è un'entità attiva per il fatto che introduce un ritardo di propagazione non trascurabile. Un canale di comunicazione statico, infatti, non può garantire tale comportamento. Altri problemi sono di natura implementativa. La soluzione ottimale si ottiene modellando sia il Peer che il Bus come processi. Di seguito chiameremo semplicemente Bus ogni processo gestore del canale di comunicazione, e Peer il processo che intende trasmettere su di esso.

La soluzione prevede che tutti i processi dispongano di due canali SPIN (da non confondere con il processo gestore del canale) per effettuare la connessione verso il bus: *input* e *output*. Ogni processo invia i dati nel canale di output e riceve dati dagli altri processi dal canale di input. E' quindi il processo gestore del canale che si farà carico della ricezione e successiva consegna dei messaggi ai processi ad esso connessi.

Un'altro compito fondamentale è l'introduzione del ritardo di propagazione nella consegna, facendo in modo di creare la condizione che un processo può rilevare libero il canale, anche se in realtà non lo è.

Come verrà spiegato nella sezione apposita, ogni coppia di Peer è connessa ad un solo Bus, in questo modo può essere implementato un modello che supporti un numero qualsiasi di Peer.



4.3 Realizzazione del CSMA/CD

Una volta realizzato il modello, il passo successivo è stato quello di decidere in che modo implementare il CSMA/CD. Inizialmente il *CARRIER SENSE* veniva realizzato usando la variabile booleana `occupato`. Essa era sostanzialmente un flag, posta a TRUE indicava che il canale è occupato (è cioè in corso una trasmissione); posta a FALSE indicava che il canale è, al contrario, libero. Nella revisione il modello è stato rielaborato, aggiungendo appunto la possibilità di supportare un numero arbitrario di Peer. Si era pensato creare `m` variabili *occupato*, ogni una associata ad un certo Bus, ma, a causa dell'assenza in SPIN dei puntatori, la definizione di processo era diversa tra vari Peer, si consideri a titolo di esempio la seguente situazione: 3 Peer, 3 Bus, il Bus 12 connette il Peer 1 con il Peer 2, il Bus 13 connette il Peer 1 con il Peer 3 e il Bus 23 connette il Peer 2 con il Peer 3. In questo modello ci sono 3 variabili di stato: occupato12, occupato13 e occupato23, poste a TRUE indicano che il Bus xy è occupato. L'implementazione in codice PROMELA è la seguente:

```
bool occupato12 = FALSE;
```

```
bool occupato13 = FALSE;
bool occupato23 = FALSE;

init {

    run peer1(...);
    run peer2(...);
    run peer3(...);

    run Bus12(...);
    run Bus12(...);
    run Bus12(...);
}

proctype peer1(...) {

    if (occupato12 || occupato 13)...
}

proctype peer2(...) {

    if (occupato12 || occupato 23)...
}

proctype peer3(...) {

    if (occupato13 || occupato 23)...
}

proctype Bus12(...) {

    if (nuova trasmissione) occupato12 = TRUE;
}

proctype Bus13(...) {

    if (nuova trasmissione) occupato 13 = TRUE
}

proctype Bus23(...) {
```

```
    if (nuova trasmissione) occupato 23 = TRUE
}
```

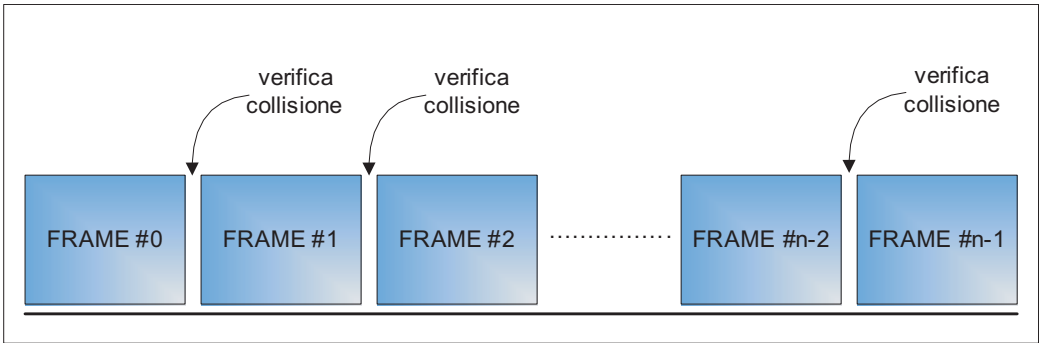
L'obiettivo è invece quello di usare una sola definizione di processo per il Peer, una per il Bus, e istanziarlo di volta in volta. Per ovviare al problema precedente, vengono creati altri 2 canali per ogni Peer. Tali canali hanno lo scopo di notificare lo stato del Bus (canale state) e l'avvenuta collisione (canale collision). In particolare se la dimensione del canale state è 0, il Peer considera il Bus libero, e può quindi effettuare la trasmissione, viceversa, se la dimensione del canale è diversa da 0 il Peer si pone in attesa.

Dalla parte opposta, è il Bus che invia un messaggio a entrambi i canali state (dei due Peer ad esso connessi), questo non viene eseguito istantaneamente alla ricezione di una nuove trasmissione, ma dopo un certo periodo dipendente dalla distanza tra i Peer (in modo da simulare il ritardo di propagazione, affrontato precedentemente).

Di conseguenza un processo può trovare il canale libero, e quindi iniziare la trasmissione (anche se è già in corso un'altra trasmissione).

Se il Bus, portandosi nello stato di attesa, non riceve trasmissioni dai Peer, rimuove il messaggio precedentemente inviato nel canale state, notificando ai Peer che il Bus è libero.

Come già detto, una trasmissione è intesa come una raffica di messaggi. Per realizzare il *COLLISION DETECTION* si fa in modo che il Peer controlli se tra un messaggio e il successivo si è verificata una collisione. La verifica di collisione viene realizzata controllando la dimensione del canale collision, se essa è 0 non vi è collisione, viceversa è avvenuta una collisione.



In questo caso, non è il Bus che provvede alla rimozione del messaggio da entrambi i canali collision, ma il Peer stesso, una volta individuata la collisione. Se così non fosse stato, poteva esserci il caso in cui il Bus rimuoveva il messaggio prima che il Peer avesse avuto l'opportunità di rilevare tale collisione.

4.4 Realizzazione del Backoff esponenziale

Il backoff esponenziale viene ottenuto mediante la funzione random vista precedentemente. In questa fase il Peer sceglie in modo casuale un numero (slot), esso equivale all'attesa che il Peer deve effettuare prima di ritentare la trasmissione. Nel modello reale, ogni slot è il *Round trip time* (RTT) del valore di 52,1μS, esso equivale al doppio del tempo che impiega il segnale per arrivare da un capo all'altro del cavo (della lunghezza di 300 metri).

In SPIN la distanza tra due processi viene modellata in base al numero di loop. Se ad esempio, due Peer distano 12, sono necessari 12 loop (effettuati dal processo Bus come vedremo in seguito) affinchè il Peer destinatario riceva la trasmissione, ne segue che RTT nel modello SPIN vale 600. Il valore del *Round Trip Time* deriva dalla considerazione che la distanza tra due processi non è nota a priori, si conosce solamente la massima distanza ammissibile, e cioè 300 meri.

Dato che nella modellazione SPIN la distanza tra i vari Peer è nota a priori, possiamo senza perdere di generalità, ridurre RTT/2 alla massima distanza tra due Peer nel sistema. Tale operazione garantisce una maggiore efficienza durante la simulazione e durante la verifica.

SPIN non offre alcun costrutto per la generazione dinamica di uno stato, è stato quindi necessario creare 10 stati. Ogni stato corrisponde al valore assunto dalla variabile `round` indicante il numero di volte che il Peer è entrato in fase di Backoff esponenziale.

Lo stato *Round_x* fa in modo che il Peer scelga in modo non deterministico il valore della variabile slot tra x elementi, il cui valore è crescente secondo la funzione $2^{indice} \times RTT$, dove indice $\in$ [0 ; x] e come detto in precedenza x $\in$ [1 ; 10]. A titolo di esempio si riporta il codice per *Round_7* e per `RTT/2` = 5 (massima distanza tra due Peer nel modello):

```
ROUND_7:
        do
            ::  slot = 10 -> goto BO_WAIT;
            ::  slot = 20 -> goto BO_WAIT;
            ::  slot = 40 -> goto BO_WAIT;
            ::  slot = 80 -> goto BO_WAIT;
            ::  slot = 160 -> goto BO_WAIT;
            ::  slot = 320 -> goto BO_WAIT;
            ::  slot = 640 -> goto BO_WAIT;
            ::  slot = 1280 -> goto BO_WAIT;
        od;
```

4.5 Implementazione

L'implementazione del sistema è stata eseguita in due fasi: sviluppo del processo Peer e sviluppo del processo Bus. Nella seguente discussione si assume che il numero dei Peer sia 2, di conseguenza il Bus è unico.

Assumiamo inoltre che entrambi i Peer vogliano iniziare una nuova trasmissione (si omettono quindi i processi passivi che si trovano solamente in ascolto). Per l'implementazione del modello vengono generati i seguenti canali:

due canali tra i Peer e il Bus : peer1_to_channel, peer2_to_channel;

due canali tra il Bus e i Peer : channel_to_peer1, channel_to_peer2

due canali per lo stato del Bus : state_p1, state_p2

due canali per la verifica di collisione : collision_p1, collision_p2

un canale per il segnale di disturbo JAM : jam_channel

inoltre i messaggi scambiati tra processi possono essere di quattro tipi: `FRAME`, `JAM`, `STATE` e `COLL`. `FRAME` ha il compito di simulare il messaggio scambiato tra Peer nel modello reale. Stessa osservazione per il segnale di `JAM`. `STATE` è il messaggio che il Bus invia ai Peer per notificarne lo stato (libero - occupato), mentre `COLL` notifica al Peer l'avvenuta collisione.

4.5.1 Processo Peer

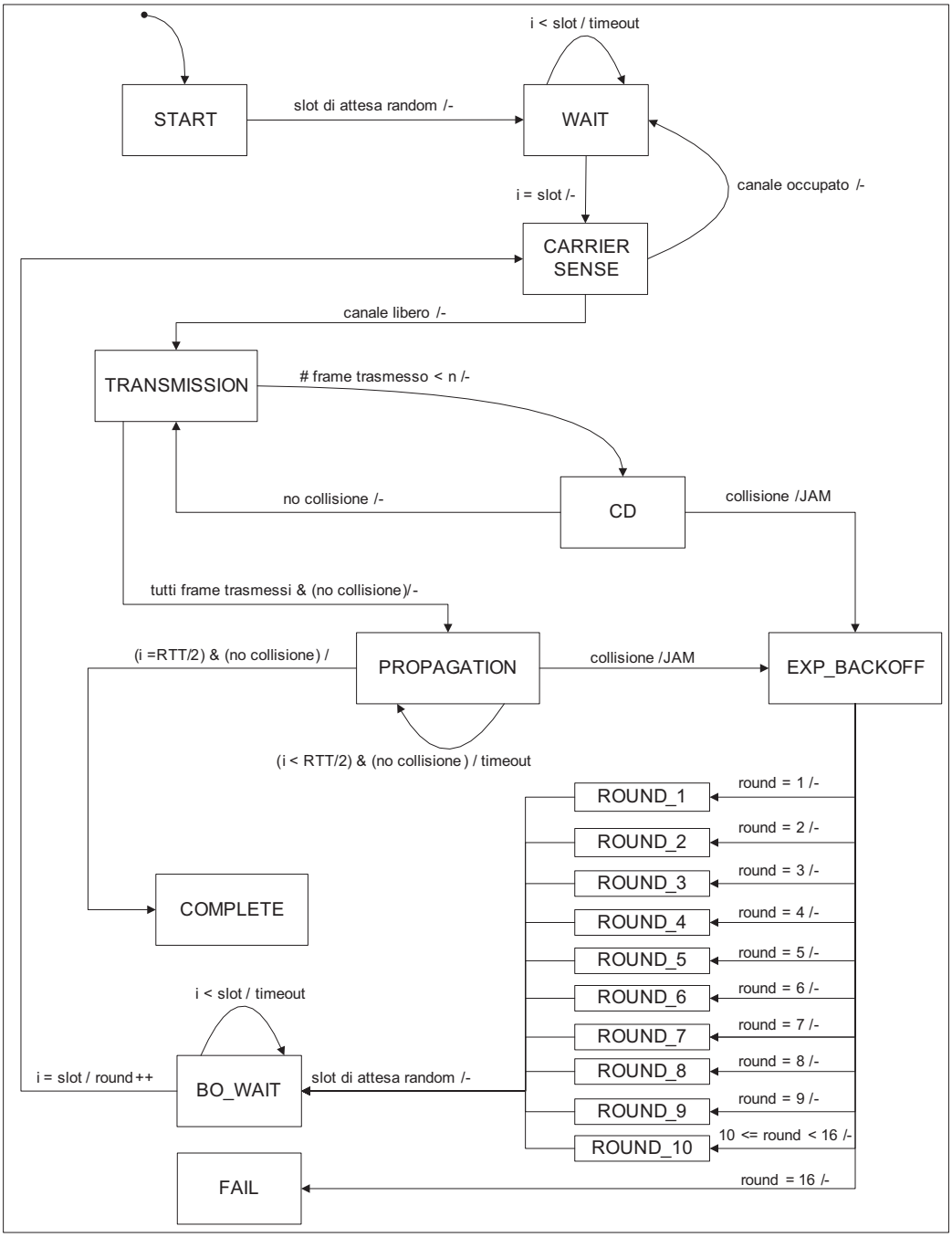
Il Peer è il componente che si occupa di simulare il comportamento del terminale visto nei capitoli precedenti. Realizza al suo interno il procollo CSMA/CD.

E' importante sottolineare che entrambi i Peer non iniziano subito la trasmissione, ma dopo un tempo random. Questo per evitare che entrambi inizino a trasmettere nello stesso istante, rendendo quindi più chiara la simulazione.

Gli stati del Peer sono: *Start*, *Wait*, *Carrier sense*, *Transmission*, *CD*, *Propagation*, *Exp_backoff*, *Round_x* (con $1 \leq x \leq 10$), *Bo_wait*, *Complete* e *Fail*. La definizione del processo Peer è la seguente:

```
proctype peer(chan out, in, jc, state, collision) {
 ...
 }
```

vengono passati come parametro i canali di input e output (`in` e `out`) per lo scambio di messaggi, il canale per l'invio del segnale di JAM (`jc`), il canale per analizzare lo stato del Bus (`state`) e il canale per la verifica della collisione (`collision`).



Il processo inizia nello stato *Start*, viene scelto randomicamente il numero di attesa (slot) mediante la seguente procedura:

```
START:
        do
            :: slot = 1 -> goto WAIT;
            :: slot = 2 -> goto WAIT;
            :: slot = 3 -> goto WAIT;
            :: slot = 4 -> goto WAIT;
        od;
```

con slot $\in \mathbb{N}$ e $1 \leq$ slot ≤ 4. Da sottolineare che, anche se viene usata la stessa variabile, questo valore di slot non ha nulla a che vedere con quello scelto nella fase di Backoff esponenziale.

Una volta effettuata la scelta di slot, il Peer si porta nello stato di *Wait*, dove, mediante un cappio su questo stato, si itera un certo numero di volte in base alla variabile slot. Il codice è il seguente:

```
WAIT:
        do
            ::  nr < slot -> timeout; nr = nr + 1;
            ::  nr == slot -> nr = 0; goto CARRIER_SENSE;
        od;
```

il modello prevede di fare uso del costrutto *timeout* offerto da SPIN. Richiamando tale funzione il Peer attende un tempo definito e eventualmente lo scheduler passa il controllo ad un'altro processo in attesa. Finchè la variabile nr è minore di slot il Peer rimane nello stato *Wait*, altrimenti si porta nello stato di *Carrier sense*. L'implementazione di questo stato è la seguente:

```
CARRIER_SENSE:
        do
            ::  len(state) == 0 -> goto TRANSMISSION;
            ::  len(state) != 0 -> break;
        od;
```

In tale stato si verifica se è già in corso una trasmissione iniziata da un'altro processo; in caso positivo (dimensione del canale state diversa da 0) il Peer attende che il Bus si liberi (lo scheduler passa il controllo ad un'altro processi).

Se invece il canale appare libero (dimensione del canale state = 0), il sistema inizia la trasmissione, portandosi nello stato di *Transmission*.

```
TRANSMISSION:
        do
            :: ind < NUM_FRAME -> ind = ind + 1;
                 out!FRAME(mypid, ind); goto CD;
            :: ind == NUM_FRAME -> ind = 0; goto PROPAGATION;
        od;
```

Il funzionamento di questo stato consiste nell'inviare un certo numero di messaggi (o frame) verso il Bus. NUM_FRAME è la variabile che indica il numero di frame da inviare.

Come descritto precedentemente, tra un frame e il successivo si controlla se si è verificata una collisione, portandosi nello stato di *CD*.

```
    CD:
        do
            ::  if
                    :: len(collision) == 0 -> goto TRANSMISSION;
                    :: else -> collision?_,_; jc!JAM; ind = 0;
                         goto EXP_BACKOFF;
                fi;
        od;
```

Nello stato di *CD* si analizza la dimensione del canale collision. Se essa è diversa da 0, si è verificata una collisione. Come spiegato nel paragrafo precedente, tale messaggio viene rimosso dal canale, viene inviato un segnale di JAM (disturbo) nell'apposito canale e, successivamente, il Peer si porta nello stato di *Exp_backoff*. Se la dimensione del canale collision è nulla, non si è verificata alcuna collisione; di conseguenza si può procedere con la trasmissione del successivo frame.

Se non si verificano collisioni durante la trasmissione del messaggio, il Peer si porta nello stato di *Propagation*.

In base a quanto visto nei capitoli precedenti, una volta trasmesso il messaggio, il Peer deve aspettare un'ulteriore tempo pari a T (per accertarsi che non si sia verificata collisione nell'intervallo $l + T$), prima di portarsi nello stato di *Complete*. Questo è reso possibile grazie allo stato di *Propagation*:

```
PROPAGATION:
        do
            ::  nr < RTT/2 ->   if
```

```
                                        ::  len(collision) == 0 ->
                                              timeout; nr = nr + 1;
                                        ::  else -> collision?_,_;
                                              jc!JAM; nr = 0;
                                              goto EXP_BACKOFF;
                                  fi;
            ::  nr >= RTT/2 -> nr = 0; goto COMPLETE;
        od;
```

Tale stato fa in modo che il Peer cicli un numero di volte pari alla costante RTT/2. Tale valore è pari al tempo che impiegherebbe il segnale per arrivare da un capo all'altro del cavo, nel nostro caso, tale valore è pari alla distanza maggiore tra due processi nel sistema. Nel caso reale, non conoscendo a priori tale distanza, tale valore è pari alla metà del *Round Trip Time*.

Se non si sono verificate collisioni nel periodo (l ; l + T), il Peer si porta nello stato di *Complete*, dando per scontato che il messaggio è stato inviato con successo. Se invece si verifica una collisione, il Peer si porta nello stato di *Exp_backoff*. Tale stato implementa il protocollo del Backoff esponenziale nel seguente modo:

```
EXP_BACKOFF:
        do
            ::  round = round + 1;
                if
                    ::  round == 1 -> goto ROUND_1;
                    ::  round == 2 -> goto ROUND_2;
                    ::  round == 3 -> goto ROUND_3;
                    ::  round == 4 -> goto ROUND_4;
                    ::  round == 5 -> goto ROUND_5;
                    ::  round == 6 -> goto ROUND_6;
                    ::  round == 7 -> goto ROUND_7;
                    ::  round == 8 -> goto ROUND_8;
                    ::  round == 9 -> goto ROUND_9;
                    ::  round == 10 -> goto ROUND_10;
                    ::  (round > 10 && round < 16) -> goto ROUND_10;
                    ::  round == 16 -> goto FAIL;
                fi;
        od;
```

Ad ogni esecuzione del Backoff esponenziale, la variabile round viene incrementata di 1. Successivamente, in base al valore di round viene scelto lo stato successivo.

L'assenza in SPIN di una funzione per la costruzione di uno stato in modo dinamico, ha imposto la presenza di uno stato per ogni valore di round, in base alla seguente descrizione:

- se round ≤ 10 lo stato successivo è *Round_x*, con x valore corrente di round;
- se $10 <$ round < 16 lo stato successivo è *Round_10*;
- se round $= 16$ la trasmissione viene abortita e lo stato successivo è *Fail*.

Il significato degli stati *Round_x* è stato ampiamente discusso nell'apposito paragrafo 4.4.

Il valore di slot rappresenta l'intervallo di attesa esponenziale dopo il quale riprovare la trasmissione. Una volta effettuata la scelta del valore di slot, in modo analogo allo stato di Wait, si attende un tempo pari al valore di slot. Lo stato che permette questo comportamento è *Bo_wait*:

```
BO_WAIT:
        do
            ::  nr < slot -> timeout; nr = nr + 1;
            ::  nr >= slot -> slot = 0; nr = 0; goto CARRIER_SENSE;
        od;
```

Finchè nr è minore di slot il Peer attende (eseguendo timeout) e successivamente nr viene incrementata di 1; se, alla successiva iterazione, nr è uguale a slot, il Peer si porta nuovamente nello stato *CARRIER SENSE*, ritentando la trasmissione se il canale è libero.

Affinchè il Peer raggiunga uno stato stabile, indifferentemente dal fatto che la trasmissione abbia avuto successo oppure no, sia lo stato *Complete* che *Fail* fanno in modo di bloccare l'esecuzione, l'unica differenza tra i due è la stringa stampata a schermo,

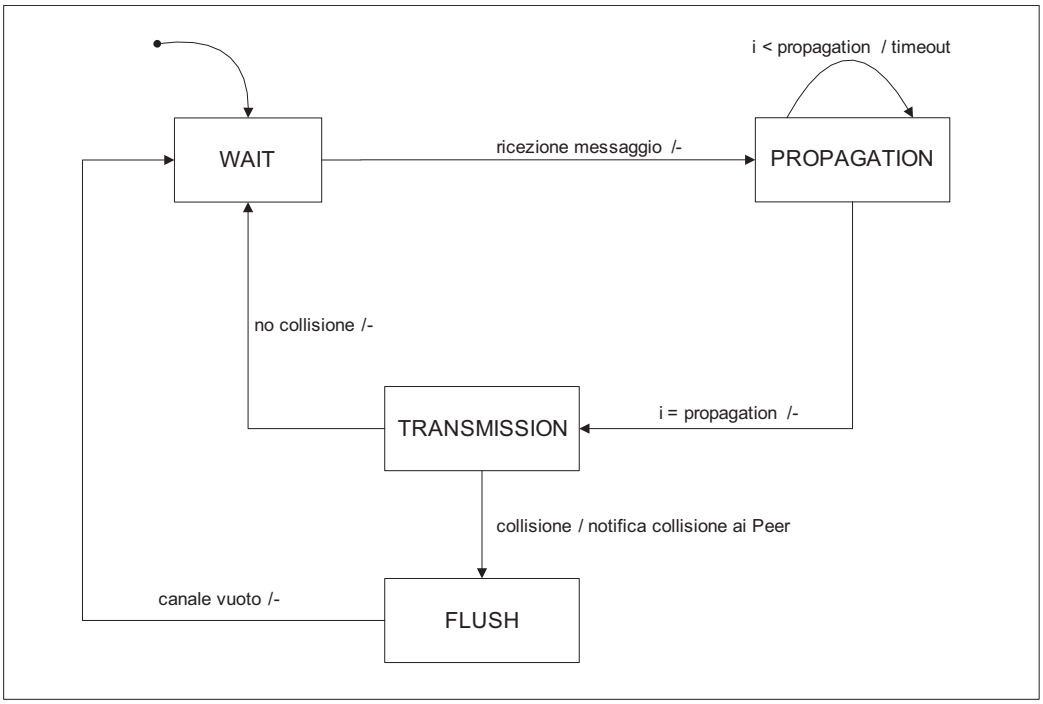
```
FAIL:   printf("Transmission aborted");


COMPLETE:   printf("Transmission complete");
```

4.5.2 Processo Bus

Il Bus è il processo che si occupa di ricevere/inviare i messaggi da/verso i Peer. Per ragioni esposte precedentemente, esso è unico nel sistema.

Gli stati che caratterizzano il Bus sono: *Wait*, *Propagation*, *Transmission* e *Flush*.



La definizione del processo Bus è la seguente:

```
proctype bus(chan in1, out1, in2, out2, state1, collision1, state2,
collision2; int propagation) {
...
...
}
```

vengono passati come parametri i canali di input e output per i due Peer (`in1` `out1`, `in2` e `out2`), i canali di state e collision per i due peer (`state1`, `collision1`, `state2` e `collision2`) e un valore intero (`propagation`) indicante la distanza tra i due Peer connessi al Bus .

Inizialmente il Bus si trova nello stato di *Wait*, e vi rimane finchè non riceve frame dai Peer ad esso connessi. Per verificare la presenza di nuove trasmissioni, il Bus controlla la dimensione dei canali di input (output per ogni singolo Peer): se essa è diversa da 0 implica che è un Peer ha iniziato una trasmissione.

Oltre a questo controlla se si è verificata una collisione: se la dimensione dei canali di input è diversa da 0, i Peer hanno iniziato a trasmettere nello

stesso istante. Di conseguenza si procede inviando un messaggio su entrambi i canali collision, notificando ai Peer l'avvenuta collisione. Se invece non vi è una nuova trasmissione da uno dei Peer, si comunica che il Bus è libero, eliminando il messaggio che precedentemente si trovava nei canali di state. L'implementazione dello stato *Wait* è la seguente:

```
WAIT:
        do
            ::  len(state1) != 0 -> state1?_,_;
            ::  len(state2) != 0 -> state2?_,_;
            ::  len(in1) == 0 && len(in2) != 0 ->
                    in2?FRAME(rcv1, rcv2);
                    if
                        :: collision == 1 -> collision = 0;
                            goto PROPAGATION;
                        :: else -> goto PROPAGATION;
                    fi;
            ::  len(in1) != 0 && len(in2) == 0 ->
                    in1?FRAME(rcv1, rcv2);
                    if
                        :: collision == 1 -> collision = 0;
                            goto PROPAGATION;
                        :: else -> goto PROPAGATION;
                    fi;
            ::  len(in1) != 0 && len(in2) != 0 ->
                    collision = 1;
                    if
                        :: len(collision1) == 0 ->
                            collision1!COLL(t);
                        :: else -> ;
                    fi;
                    if
                        :: len(collision2) == 0 ->
                            collision2!COLL(t);
                        :: else -> ;
                    fi;
                    goto FLUSH;
        od;
```

Come già detto, lo stato *Propagation* è necessario per simumale il ritardo di propagazione del segnale elettrico. In questo modo i Peer non riceveranno

istantaneamente il messaggio, ma solo dopo un certo tempo dipendente dalla distanza tra di essi.

Come visto in precedenza, per ritardare l'evento di trasmissione si usa ciclare su tale stato, richiamando la funzione timeout ad ogni iterazione:

```
PROPAGATION:
        do
            ::  if
                    ::  ind < propagation -> timeout; ind = ind + 1;
                    ::  else -> ind = 0; goto WAIT;
                fi;
        od;
```

Quando la variabile ind assume il valore di `propagation`, il Bus si porta nello stato di *Transmission* per permettere la trasmissione del messaggio a tutti i Peer ad esso connessi. Viene inoltre comunicato ad entrambi i Peer che lo stato del Bus è occupato. Restando fedele al modello fisico, tale fase viene eseguita dopo lo stato di *Propagation*, simulando quindi il tempo di propagazione.

```
TRANSMISSION:
        do
            ::  atomic {
                            ind = 0;
                            if
                                ::  len(state1) == 0 -> state1!STATE(t);
                                ::  else -> ;
                            fi;
                            if
                                ::  len(state2) == 0 -> state2!STATE(t);
                                ::  else -> ;
                            fi;
                            out1!FRAME(rcv1, rcv2);
                            out2!FRAME(rcv1, rcv2);
                        } goto WAIT;
        od;
```

A causa del comportamento FIFO dei canali di SPIN, viene introdotto lo stato *Flush* per rimuovere manualmente i messaggi dai canali non ancora scaricati dai processi (poichè questi si sono portati in fase di backoff a causa della collisione). Nel caso in cui non ci sia alcuna collisione, si procede a

trasmettere il frame a tutti i Peer connessi al Bus. In entrambi i casi, il bus si porta nello stato di *Wait*.

```
FLUSH:
        do
            ::  len(in1) == 0 && len(in2) == 0 ->
                    goto WAIT;
            ::  len(in1) != 0 && len(in2) == 0 ->
                    in1?_,_ ;
            ::  len(in1) == 0 && len(in2) != 0 ->
                    in2?_,_ ;
            ::  len(in1) != 0 && len(in2) != 0 ->
                    in1?_,_ ; in2?_,_ ;
        od;
```

4.6 Modello per un numero arbitrario di Peer

Il modello visto precedentemente prevede due processi Peer ed un solo Bus. Si vole ora estendere la possibilità di supportare un numero arbitrario di Peer.

Il modello è basato sull' astrazione che, ogni Peer è all'oscuro di quanti siano i Bus ad esso connessi. Tale modello prevede inoltre un Bus per ogni coppia di Peer; ogni Peer è connesso con n-1 Bus che a sua volta sono connessi con i rimanenti n-1 Peer (nel caso precedente trattandosi di due soli Peer ogni processo inviava i dati sull'unico Bus esistente). Nel momento in cui un Peer si porta in trasmissione, i dati vengono inviati agli n-1 Bus connessi, e di conseguenza agli n-1 Peer. Per ogni Bus è definita la distanza tra i Peer che lo caratterizzano.

Il numero di messaggi che ogni Peer invierà è pari a (n-1 * `frameNum`), esso deriva dalla considerazione che, il messaggio una volta letto, viene eliminato dal canale (proprietà dei canali SPIN), di conseguenza se ogni Peer si limitasse ad inviare un solo messaggio, ci sarebbero n-2 Bus che non ne riceverebbero alcuno (poichè il primo Bus che ha la capacità di leggere, eliminerà il messaggio dalla coda); `frameNum` è il numero di frame che ogni Peer invia durante la trasmissione.

Tale messaggio si propagherà in maniera autonoma su ogni Bus, raggiungendo nell'istante opportuno (in base alla distanza) tutti gli altri processi del modello. Il numero m di Bus necessari nel caso di n Peer viene ricavato dalla seguente formula:

$$m = \frac{(n(n-1))}{2}$$

Per esplicitare meglio il funzionamento, ipotizziamo la seguente situazione: 4 Peer, 6 Bus (numero ricavato dalla formula precedente). Il Peer 1 è connesso ai rimanenti Peer mediante tre Bus: Bus 1-2, Bus 1-3 e Bus 1-4. Ipotiziamo che la posizione dei Peer sul cavo sia la seguente:

- Peer 1 - posizione 0
- Peer 2 - posizione 7
- Peer 3 - posizione 11
- Peer 4 - posizione 15

ogni Bus opera in base alla distanza tra i Peer che lo compongono, in particolare, essa sarà:

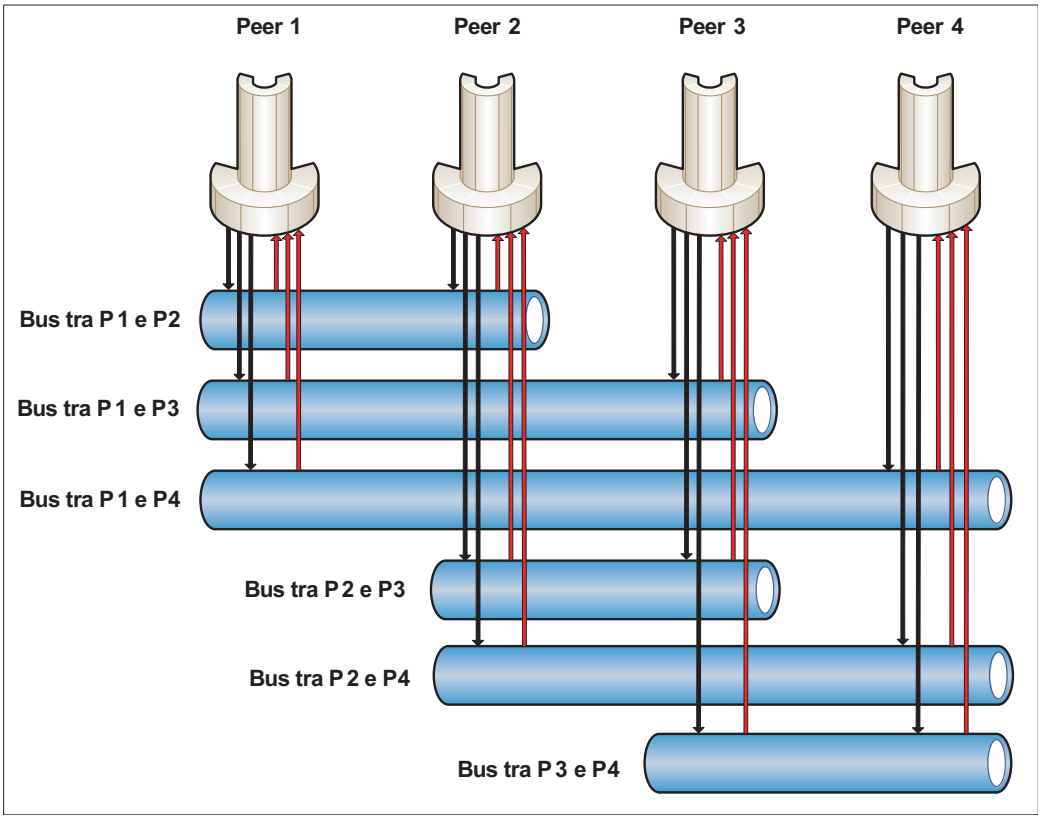
- Bus 1-2: propagation12 = 7
- Bus 1-3: propagation13 = 11
- Bus 1-4: propagation14 = 15
- Bus 2-3: propagation23 = 4
- Bus 2-4: propagation24 = 8
- Bus 3-4: propagation34 = 4

Se il Peer 1 invia un messaggio, il Bus 1-3 ciclerà 11 volte prima che il Peer 4 lo individui come occupato. In questo esempio, il *Round trip time* è pari a 30.

Ogni Peer possiede un solo canale di Output ed uno di Input, essi sono a loro volta connessi su tutti i Bus, questo vuol dire che quando il Peer 1 invia un messaggio, esso viene ricevuto contemporaneamente da tutti i Bus ad esso connessi (nel nostro caso Bus 1-2, Bus 1-3 e Bus 1-4).

Come già detto la verifica dello stato del Bus viene eseguita controllando la dimensione del canale state, se essa è diversa da 0 il Peer non sà quale sia il Bus occupato, sa solo che uno dei n-1 Bus connessi è occupato.

Discorso analogo nel caso della verifica di collisione, si analizza la dimensione del canale collision, ma non si sà quale è il Bus che ha originato tale collisione. Per questi motivi ogni Peer si considera connesso ad un solo Bus, indipendentemente dal numero di Peer.



4.7 La verifica delle proprietà

Per come è stato strutturato il modello, le proprietà del protocollo CSMA/CD da verificare tramite SPIN potrebbero essere scontante, esse derivano in modo diretto dal funzionamento del protocollo stesso. Tali proprietà vengono espresse in *Logica Temporale Lineare* (LTL). Le proprietà verificate tramite SPIN sono:

Terminazione La terminazione assicura che, prima o poi l'esecuzione termini in uno stato stabile. Esso è quindi un requisito di Liveness. La formula LTL è la seguente:

$$(\text{Peer\_1.state} = \text{Start} \longrightarrow$$
$$\text{F}(\text{Peer\_1.state} = \text{Complete} \bigvee \text{Peer\_1.state} = \text{Fail}))$$
$$\bigwedge$$
$$(\text{Peer\_2.state} = \text{Start} \longrightarrow$$

F(Peer_2.state = Complete $\bigvee$ Peer_2.state = Fail))

La formula asserisce che entrambi i Peer, una volta eseguito lo stato *Start* (il processo ha quindi avviato la computazione) devono portarsi su COMPLETE oppure su FAIL. Con tale proprietà si assicura che il Peer trasmetta correttamente il messaggio, o dopo 16 tentativi annulli la trasmissione.
Implementazione:

```
#define st_1 (peer[1]@START)
#define st_2 (peer[1]@START)
#define complete_1 (peer[1]@COMPLETE)
#define complete_2 (peer[2]@COMPLETE)
#define fail_1 (peer[1]@FAIL)
#define fail_2 (peer[2]@FAIL)

!((st_1 -> <> (complete_1 || fail_1)) &&
  (st_2 -> <> (complete_2 || fail_2)))
```

La verifica di tale formula fornisce esito positivo (file *Terminate.ltl*).

Carrier sense Ci si vuole assicurare che, prima di trasmettere si esegua il carrier sense (ascolto del canale) e solo se esso appaia libero, si inizi la trasmissione. La formula in logica temporale lineare è la seguente:

((peer_1.state = Carrier sense $\bigwedge$ len(state_p1) = 0) $\longrightarrow$ X peer_1.state = Transmission) $\bigwedge$
((peer_1.state = Carrier sense $\bigwedge$ len(state_p1) = 1) $\longrightarrow$ X peer_1.state = Carrier sense) $\bigwedge$
((peer_2.state = Carrier sense $\bigwedge$ len(state_p2) = 0) $\longrightarrow$ X peer_2.state = Transmission) $\bigwedge$
((peer_2.state = Carrier sense $\bigwedge$ len(state_p2) = 1) $\longrightarrow$ X peer_2.state = Carrier sense)

Il significato della precedente formula è il seguente: se uno dei due Peer si trova nello stato di *Carrier sense* e il canale è libero (dimensione di state = 0), allora il successivo stato deve essere *Transmission* (si inizia la trasmissione). Altrimenti, se il canale è occupato (dimensione di state = 1), si attende che esso si liberi, rimanendo nello stato di *Carrier sense*. Senza perdere di generalità, è stato chiesto a SPIN di verificare

la metà della formula, e cioè quella che si riferisce al Peer 1, essendo duale, il risultato sarà valido anche per il Peer 2 e di conseguenza per la formula completa. L'implementazione è la seguente:

```
#define cs1 peer[1]@CARRIER_SENSE
#define cs2 peer[2]@CARRIER_SENSE
#define t1 peer[1]@TRANSMISSION
#define t2 peer[2]@TRANSMISSION
#define s1_0 (len(state_p1) == 0)
#define s1_1 (len(state_p1) == 1)

!(<> (((cs1 && s0) -> <> t1) && ((cs1 && s1) -> <> cs1)))
```

La verifica ti tale formula restituisce esito positivo (file *CS.ltl*).

Collision Detection Vogliamo verificare che il meccanismo di verifica delle collisioni è sempre efficace, in particolare si vuole verificare che: se il Peer 1 è in trasmissione, e il Peer 2 inizia a trasmettere (si ha quindi una collisione) entrambi se ne accorgeranno e si comporteranno in modo opportuno. Per motivi esposti precedentemente, verifichiamo solo una parte della formula, quella riferita al caso in cui il Peer 1 si trovi già in trasmissione, e il Peer 2 inizia a trasmettere (l'altra parte avrebbe riguardato il caso in cui il Peer 2 si trovava in trasmissione, e in seguito subentrava il Peer 1). La formula in logica temporale lineare è la seguente:

$$((\text{Peer\_1.state} = \text{TRANSMISSION} \bigwedge \\ \text{F(Peer\_2.state} = \text{TRANSMISSION})) \longrightarrow \\ \text{F(Peer\_1.state} = \text{Exp\_backoff} \bigwedge \text{Peer\_2.state} = \text{Exp\_backoff})$$

La formula precedente asserisce che, se il Peer 1 si trova nello stato di *Transmission* (è cioè in trasmissione), e il Peer 2 inizia a trasmettere, allora entrambi si portano nello stato di *Exp_backoff*.
L'implementazione è la seguente:

```
#define t1 peer[1]@TRANSMISSION
#define t2 peer[2]@TRANSMISSION
#define eb1 peer[1]@EXP_BACKOFF
#define eb2 peer[2]@EXP_BACKOFF
```

```
!((t1 && <>  (t2)) -> <>  (eb1 && eb2))
```

La verifica ti tale formula restituisce esito positivo (file *CD.ltl*).

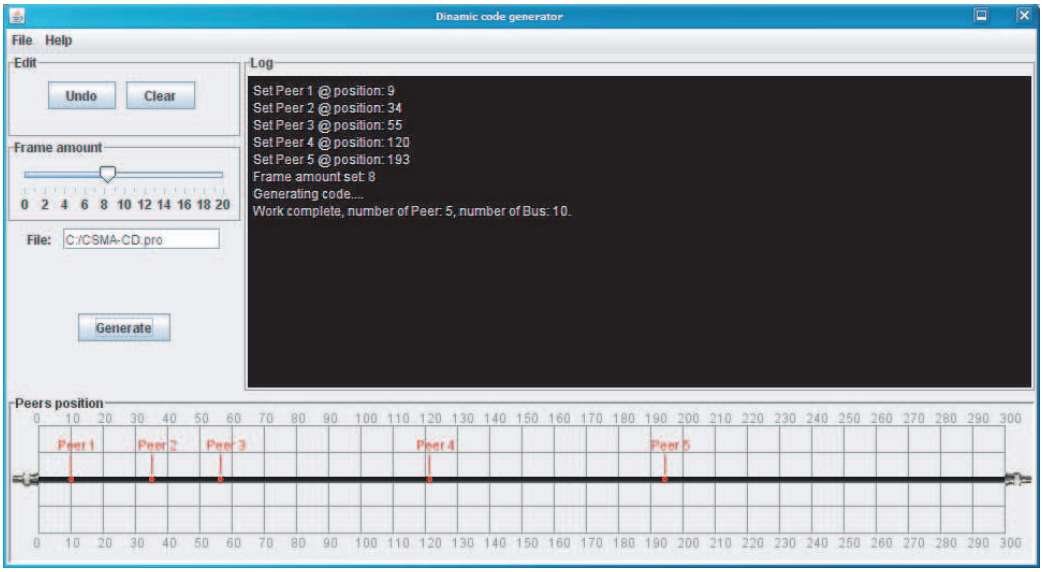
Collisione in l ; l + T Come detto in precedenza, il Peer, ultimata la trasmissione al tempo l, deve poter rilevare eventuali collisioni in un'ulteriore tempo T (tempo di propagazione) . Se non si verificano collisioni nell'intervallo l ; l + T allora la trasmissione è stata completata con successo, altrimenti viene avviato il backoff esponenziale. La formula in logica temporale lineare corrispondente a tele proprietà è la seguente:

((Peer1_state = Propagation $\bigwedge$ len(collision) = 0 $\bigwedge$ i < RTT/2) $\longrightarrow$
X Peer_1.state = Propagation) $\bigwedge$
((Peer1_state = Propagation $\bigwedge$ len(collision) = 0 $\bigwedge$ i = RTT/2) $\longrightarrow$
X Peer_1.state = Complete) $\bigwedge$
((Peer1_state = Propagation $\bigwedge$ len(collision = 1)) $\longrightarrow$
X Peer_1.state = Exp_backoff) $\bigwedge$
((Peer2_state = Propagation $\bigwedge$ len(collision) = 0 $\bigwedge$ i < RTT/2) $\longrightarrow$
X Peer_2.state = Propagation) $\bigwedge$
((Peer2_state = Propagation $\bigwedge$ len(collision) = 0 $\bigwedge$ i = RTT/2) $\longrightarrow$
X Peer_2.state = Complete) $\bigwedge$
((Peer2_state = Propagation $\bigwedge$ len(collision) = 1) $\longrightarrow$
X Peer_2.state = Exp_backoff)

La formula precedente asserisce che se uno dei due *Peer* si trovano nello stato di *Propagation* (quindi nell'intervallo l ; l + T) e se si verifica una collisione (dimensione di collision = 1), essi si portano in fase di Backoff esponenziale. Altrimenti, dopo aver atteso il tempo di propagazione (RTT/2), si portano su *Complete*. Purtroppo non è possibile verificare tale formula poichè in SPIN mancano gli operatori matematici $'' <''$ e $'' >''$.

4.8 Generazione dinamica del codice PROMELA

E' stato richiesto un programma per la generazione dinamica del codice in base al numero di Peer e alla distanza tra essi. Il programma *Code generator* scritto utilizzando il linguaggio JAVA realizza tale funzionalità. Offre la possibilità di aggiungere un massimo di 300 Peers, per mezzo di una semplice interfaccia grafica è possibile indicare il numero di frame della trasmissione, la posizione del file di output e la posizione di ogni Peer.



L'output è il file contenente il linguaggio PROMELA che caratterizza il modello del protocollo CSMA/CD e supporta il numero di Peer immessi.

Bibliografia

[1] M. Cadoli, T.Mancini. *Metodi Formali nell'ingegneria del software*. Univ. La Sapienza, Roma. 7 Giugno 2007.

[2] James F. Kurose, Keith W. Ross. *Internet e reti di calcolatori*. McGraw-Hill, 2005.

[3] *The Model Checker Spin*. IEEE Trans. on Software Engineering, pp. 279-295, Vol. 23, No. 5, May 1997.

[4] Moshe Y. Vardi, Pierre Wolper. *An automata-theoretic approach to automatic program verification*. First IEEE Symp. on Logic in Computer Science, pp. 322-331, 1986.

[5] G.J. Holzmann, D. Peled. *An Improvement in Formal Verification*. Proc. FORTE 1994 Conference, Bern, Switzerland

[6] G.J. Holzmann. *Logic Verification of ANSI-C Code with Spin*. pp. 131-147, Proc. SPIN2000, LNCS 1885, Springer Verlag.

[7] G.J. Holzmann, R. Joshi. *Model-Driven Software Verification*. pp. 77-92, Proc. SPIN2004, LNCS 2989, Springer Verlag.

[8] *Basic Spin Manual: http://spinroot.com/spin/Man/Manual.html*

[9] Theo Ruys. *A tutorial introduction to Spin*

[10] Theo Ruys. *Spin advanced tutorial*

[11] Marc Baudoin. *Apprends* LaTeX. 1994-1998