Capitolo 3

Implementazione del NCS e del modello di rumore in NS

3.1 Introduzione al Network Simulator

Il sistema di controllo network-based da progettare è stato implementato con l'ausilio del calcolatore. Le simulazioni ci permettono di verificare la validità dei risultati ottenuti teoricamente e di valutare le prestazioni del sistema in un ambiente virtuale il più aderente alle condizioni reali [11].

Il software utilizzato per simulare il sistema di controllo network-based è il *Network Simulator* (NS) [12,13]. La scelta di tale simulatore è dettata da due motivi:

- Il software scelto è *"open source"*. In questo modo possiamo modificare il codice sorgente in modo tale da apportare i cambiamenti per i nostri scopi.
- NS è riconosciuto internazionalmente come uno dei migliori software della sua categoria.

Il Network Simulator, sviluppato presso i laboratori di ricerca dell'università californiana di Berkeley, è un simulatore di reti di telecomunicazioni, scritto in C++ [14] orientato ad oggetti (*object-oriented*) e pilotato ad eventi [15]: esso permette di simulare diversi tipi di reti basate sul protocollo IP, implementando protocolli di trasporto come il TCP e l'UDP, protocolli per la gestione della congestione, meccanismi per la gestione delle code, protocolli di routing, protocolli MAC (Medium Access Control) per reti LAN e WLAN (Wireless Area Network), collegamenti mobili e satellitari ed altro ancora [16].

Lo scenario di una simulazione viene definito da uno script OTcl, una versione object-oriented di Tcl (*Tool Command Language*). Infatti, utilizzando il linguaggio OTcl si gestiscono delle classi di oggetti, implementate in C++ e organizzate in maniera gerarchica, struttura che è alla base del simulatore [14]. Come si può vedere in figura 3.1, NS poggia su due fondamentali linguaggi

di programmazione: OTcl e C++.

La caratteristica di utilizzare due diversi linguaggi è dovuta al fatto che il simulatore deve soddisfare due diverse esigenze. NS, da un lato, deve assicurare una manipolazione dei byte e degli header dei pacchetti veloce, necessaria per la simulazione dettagliata dei protocolli, e un'implementazione degli algoritmi che lavori in maniera ottimale su grandi collezioni di dati. In secondo luogo il simulatore deve garantire, per ambiti quali la ricerca, la possibilità di variare con facilità i parametri e la configurazione delle simulazioni e deve permettere di esaminare rapidamente il grande numero di scenari che si deve poter studiare.

Il primo aspetto richiede che i singoli processi in memoria relativi alle simulazioni siano elaborati molto velocemente, meno importante è la durata totale della simulazione, il tempo perso per il debugging o per la ricompilazione.

La seconda esigenza comporta invece una discreta semplicità e rapidità nel cambiare il modello della simulazione e nel far ripartire l'elaborazione dello script modificato.

Network Simulator concilia entrambe queste due esigenze con l'utilizzo di due linguaggi di programmazione: il C++ è veloce

nell'elaborazione ma lento e macchinoso da cambiare, adatto quindi per l'implementazione dettagliate dei protocolli; l'OTcl è più lento nell'esecuzione ma più rapido e intuitivo da modificare, perfetto per configurare e impostare le simulazioni [15].

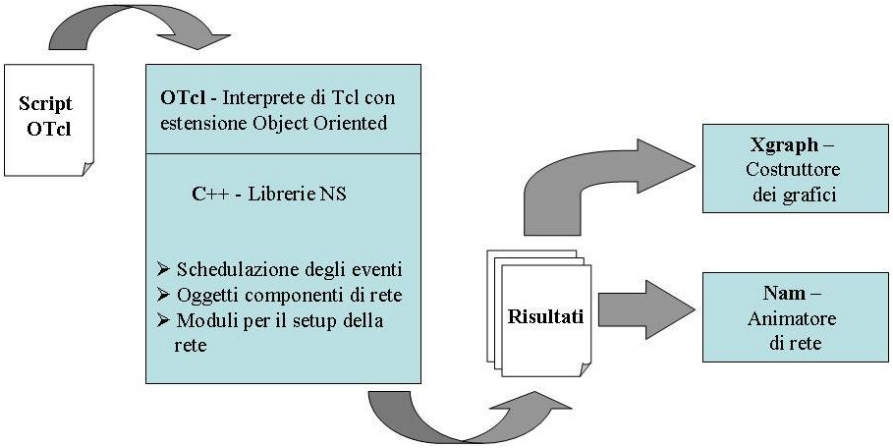

Fig. 3.1 – Struttura generale del Network Simulator

Come si nota in fig. 3.1, NS, a seconda di quanto viene specificato nello script OTcl di input, può produrre, in output, diversi tipi di *trace files* (i risultati, in figura) , che possono a loro volta essere forniti in input ai programmi *Xgraph* e *Nam*. I trace files contengono informazioni che riguardano la topologia della rete (nodi e collegamenti), gli eventi di simulazione, i pacchetti che hanno viaggiato nella rete [14].

Il Nam è un *Network Animator,* che legge un nam trace file prodotto da NS e apre una finestra nella quale dà vita ad una animazione della simulazione: mostra la topologia della rete, i flussi dei pacchetti, i pacchetti in coda nei buffer e quelli dropped, e così via [14]. Un esempio di animazione Nam è presente in figura 3.2.

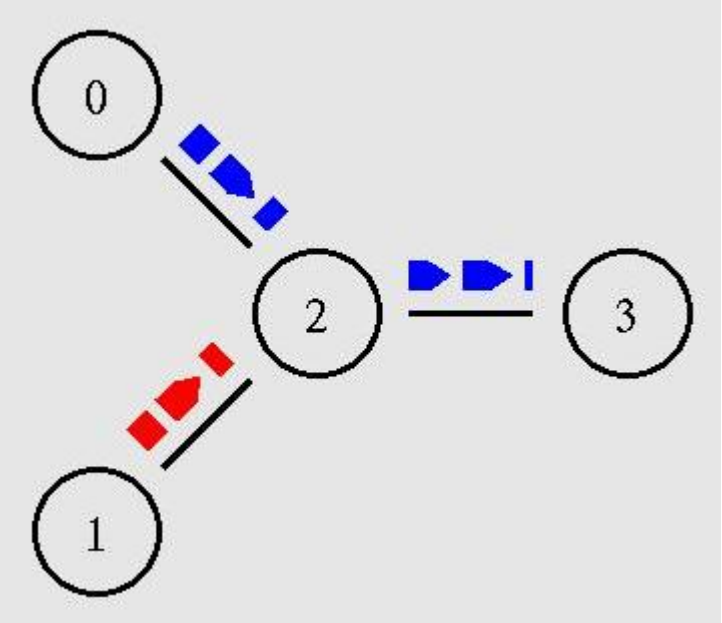

Fig. 3.2 – Screenshot di animazione Nam

Xgraph è un programma che disegna grafici bidimensionali: legge i trace files prodotti da NS ed apre una finestra nella quale visualizza alcuni risultati della simulazione [14]. Un esempio di grafico Xgraph è presente in figura 3.3.

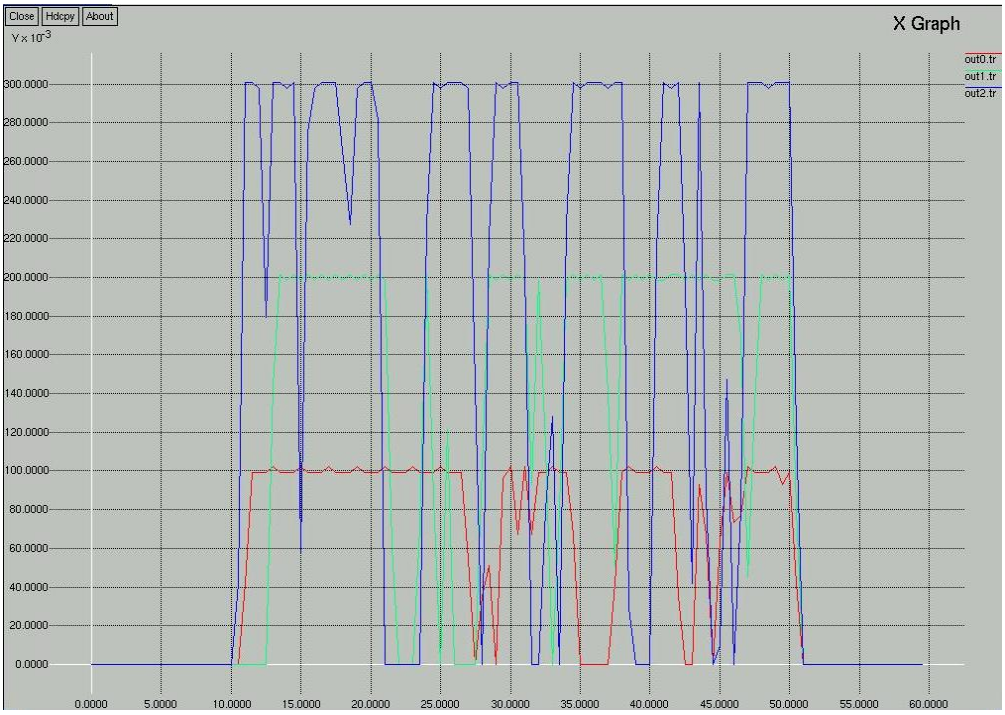

Fig. 3.3 – Screenshot di un grafico Xgraph

3.2 Implementazione del NCS

Per poter realizzare un sistema di controllo network-based è necessario installare sul Network Simulator un modulo specifico: il modulo NCS.

L'estensione NCS[8] rende possibile la simulazione di sistemi fisici collegati ad una rete. Nel nostro caso, possiamo simulare il motore elettrico connesso alla rete di controllo.

L'agente base del NCS è il "*plant*", nome storico utilizzato nella comunità di automazione industriale. L'agente plant rappresenta l'interfaccia tra le dinamiche fisiche e le dinamiche di rete del sistema. Esso può assumere il ruolo di interfaccia di un controllore, di un sensore o di un attuatore in relazione all'utilizzo previsto dallo script di simulazione.

L'estensione NCS può essere utilizzata per implementare lo scenario mostrato in figura 3.4.

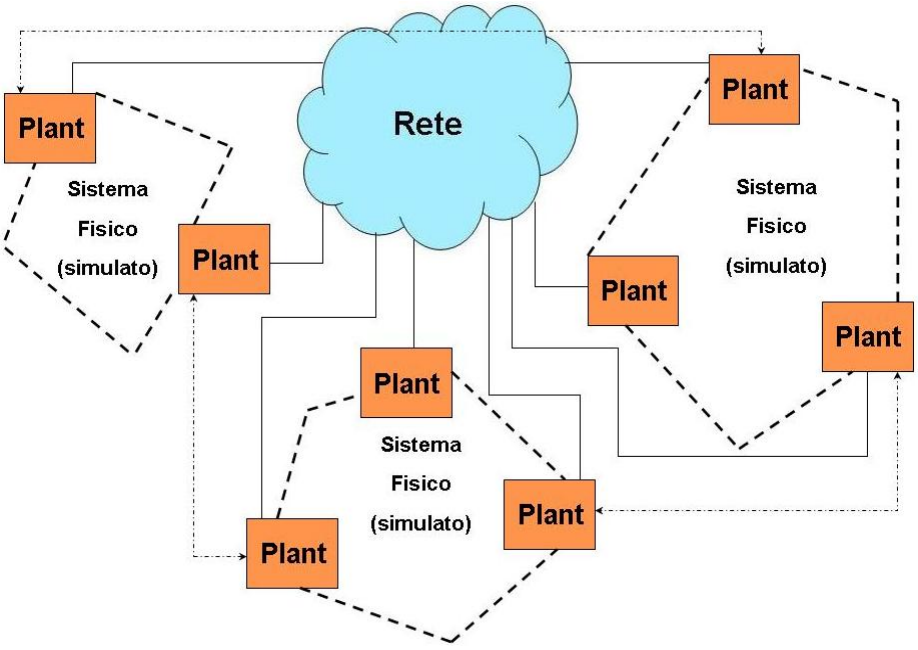

Fig. 3.4 – Uso dell'estensione NCS per la co-simulazione di reti e sistemi fisici

Una coppia di plant connessi tra loro si scambia dati campionati, ottenuti dalla simulazione del sistema fisico, e segnali di controllo, usati come input dello stesso sistema nella simulazione.

Nel codice Tcl l'agente plant richiede che siano definite le due funzioni principali dell'estensione NCS: `sysphy` e `smplschd`.

Funzione sysphy

La funzione sysphy simula le dinamiche fisiche del sistema a cui è connesso il plant. Questa funzione deve tener traccia le variabili globali t_0, x_0, x_1 e u_0 che rappresentano rispettivamente l'istante iniziale, lo stato iniziale del sistema (x_0 e x_1) e il segnale di input che è applicato al sistema all'istante t_0.

Per quanto riguarda il motore elettrico, la variabile di stato x_0 indica la corrente i_a, x_1 rappresenta la velocità angolare ω di uscita del sistema ed u_0 indica la tensione di armatura in ingresso al motore. Quando la funzione sysphy viene invocata all'istante di simulazione $t_1 \geq t_0$, sysphy deve:

- calcolare il nuovo stato del sistema all'istante t_1 come funzione di x_0, x_1, u_0 e $t_1 - t_0$;
- aggiornare i valori di x_0, x_1 e u_0;
- settare il nuovo valore di t_0 al valore di t_1;
- restituire l'uscita del sistema all'istante t_1.

La frequenza di invocazione di tale funzione dipende dal traffico di pacchetti NCS nella rete e dall'istante di campionamento. Infatti, la funzione sysphy viene invocata quando il plant riceve un pacchetto dal suo pari o quando campiona l'uscita. L'agente plant che può invocare la funzione è sia quello associato al motore DC che quello associato al controllore. La funzione selezionerà la dinamica da simulare a seconda del valore del parametro `description` che è un parametro alfanumerico della classe `PlantAgent`. In questo lavoro di tesi, tale parametro può assumere due soli valori: `plant` se l'agente è associato al motore elettrico, la cui dinamica sarà tradotta dalle funzioni `ode_motore` e `ode1_motore`, oppure `controller` se l'agente è associato al controllore, la cui dinamica sarà tradotta dalle funzioni

ode_controllore e ode1_controllore.

La funzione sysphy richiede in ingresso il parametro description dell'agente che l'ha richiamata, lo start_time ed il system_input. Il campo start_time ha un duplice significato: se sysphy è stata invocata è stata invocata dopo che un agente ha ricevuto un pacchetto, il parametro start_time rappresenta l'istante di tempo in cui tale pacchetto è stato inviato dall'agente che ha richiamato la funzione sysphy; se sysphy, invece, è stata richiamata dopo il campionamento del segnale di uscita, start_time rappresenta l'istante di tempo corrente. Il parametro system_input è l'ingresso che deve essere applicato al sistema (il vettore system_input è vuoto se l'ingresso del sistema non varia).

Funzione smplschd

La funzione smplschd permette di schedulare le invocazioni future del campionamento dell'uscita.

La funzione risulta particolarmente utile nel caso si implementino sensori in cui il campionamento avviene a frequenza variabile o fissa. Essa è anche utilizzata dal controllore come trigger dei segnali di controllo.

In ingresso la funzione `smplschd` richiede il campo `description` ed un valore numerico, `origin`, il quale può assumere due distinti valori: `origin` pari a 1 se viene invocata da un agente plant dopo la ricezione di un pacchetto, oppure pari a 0 se viene invocata dopo il campionamento dell'uscita. La funzione `smplschd` non restituisce alcun valore o variabile.

La classe C++ `PlantAgent`

I campioni dell'uscita e i segnali di controllo sono inviati in pacchetti di tipo NCS. La struttura del pacchetto NCS è la seguente:

```
struct hdr_ncs {
int seqno_;
double send_time; //istante di invio
int input_dim;
double *system_input; //input per il ricevente
static int hdr_ncs::offset_;
inline static hdr_ncs* access(const Packet* p) {
   return (hdr_ncs*) p->access(offset_);
}
int& seqno() { return (seqno_); }
```

```
};
```

La classe `PlantAgent` è derivata dalla classe `Agent`:

```
class PlantAgent : public Agent {
public:
PlantAgent();
int command(int argc, const char*const* argv);
void recv(Packet*, Handler*);
protected:
int off_plant_;
int sent_seqno_;
int recv_seqno_;
void sample();
char plant_description[36];
};
```

I due metodi fondamentali della classe sono `recv` e `sample`. La funzione `recv` prima richiama la funzione `sysphy` per calcolare lo stato del sistema e per settare il nuovo segnale di input del sistema, in seguito invoca la funzione `smplschd` per schedulare l'istante futuro in cui l'agente del plant dovrà campionare l'uscita del sistema. Questa funzione elabora i pacchetti nell'ordine

d'arrivo, senza tener conto dell'ordine reale.

La funzione `sample` richiama la funzione `sysphy` per calcolare l'uscita corrente del sistema, invia il pacchetto con questa informazione ad un altro agente plant ed infine invoca la funzione `smplschd` per schedulare l'istante futuro in cui verrà richiamato il comando `sample`.

3.3 Il caso di studio

Il sistema di controllo basato su rete della velocità angolare ω del motore DC è stato implementato, inizialmente, sia in Network Simulator che in MATLAB® utilizzando un sistema NCS a struttura diretta. La scelta di simulare il motore elettrico con il MATLAB® è dettata dalla volontà di verificare la correttezza del modello costruito con il Network Simulator. Partendo dalla correttezza del modello sifatto , si analizzeranno le prestazioni del sistema di controllo supponendo che sulla rete siano presenti errori di trasmissione.

Nel nostro caso, simuleremo un sistema NCS che controlla un solo sistema fisico rappresentato dal motore DC. In generale, con opportune modifiche agli script di simulazione, si potrà simulare un

sistema di controllo basato su rete che controlla un numero N di sistemi fisici, così come mostrato in figura 3.5.

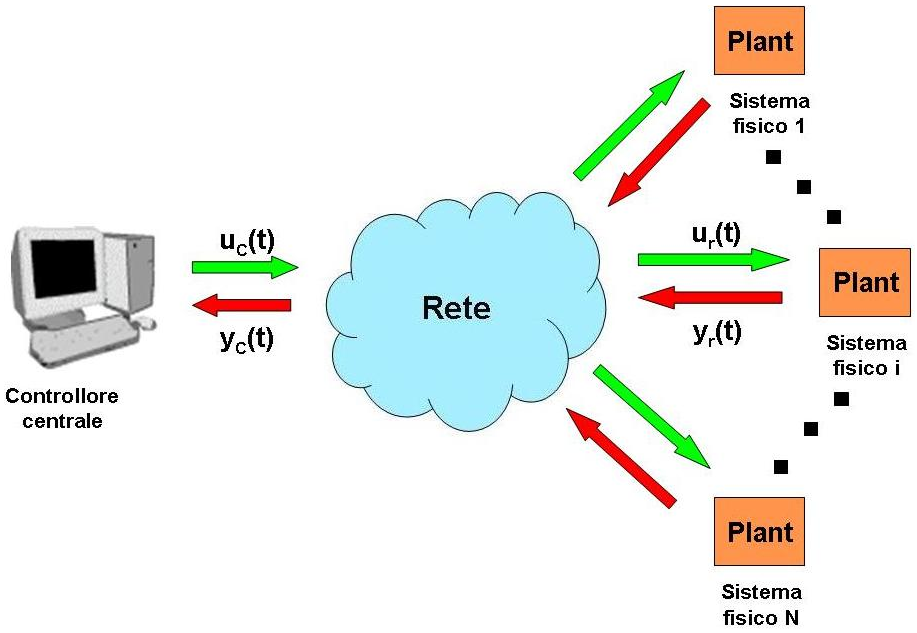

Fig. 3.5 – Sistema di controllo basato su rete di telecomunicazioni con N sistemi fisici da controllare

Lo scenario simulato prevede l'implementazione di due nodi wired, il plant e il controllore, e del mezzo trasmissivo che li connette. In figura 3.6 è mostrata una schermata ottenuta con l'animatore di rete (NAM) che mostra la topologia dello scenario simulato. Come è stato detto in precedenza, lo scenario è stato implementato anche con il MATLAB® .

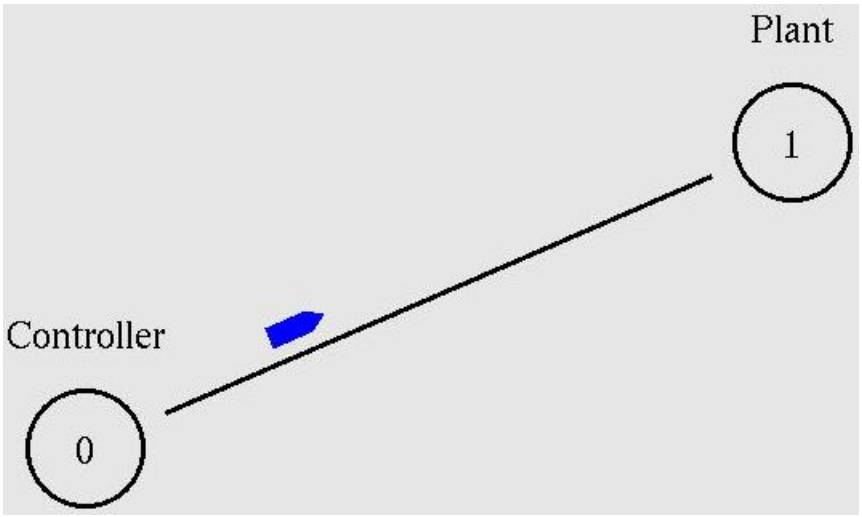

Fig. 3.6 – Screenshot della topologia del primo scenario simulato

Successivamente, utilizzando la medesima topologia della rete, si sono analizzate le prestazioni della rete in presenza di errori sul link e di variazioni di ritardi di propagazione della linea.

Come risulta evidente dalla figura 3.5, è possibile definire due nuove variabili $u_c(t)$ e $y_c(t)$, rispettivamente uscita e ingresso del controllore, legate all'ingresso e all'uscita del motore elettrico secondo le relazioni:

$$\begin{aligned} u_c(t) &= u_r(t-\tau_{CP}) \\ y_c(t) &= y_r(t-\tau_{PC}) \end{aligned} \qquad (3.1)$$

dove τ_{CP} indica il ritardo controllore-plant e τ_{PC} si riferisce al ritardo plant-controllore. Entrambi i valori dipendono dalle

caratteristiche e dalle variabili del sistema quali, ad esempio, il ritardo della linea. Nelle simulazioni considereremo i due ritardi di pari valore.

In questo lavoro di tesi si è implementato un controllore PI (proporzionale-integrale) che presenta l'uscita

$$u_{PI}(t) = K_P \cdot e(t) + K_I \cdot \int_0^t e(\tau) d\tau \qquad (3.2)$$

dove K_P è la costante di guadagno proporzionale, K_I è la costante di guadagno integrale, $e(t) = r(t) - y(t)$ è la funzione di errore calcolata istante per istante come differenza del segnale di riferimento $r(t)$ e l'uscita del sistema fisico $y(t)$.

Nel nostro caso, quindi, $y(t) = \omega(t)$ è la velocità angolare del motore e $r(t)$ è il valore di riferimento costante pari a ω_{rif} (in questo lavoro di tesi pari a 200 rad/sec).

3.3.1 Implementazione in NS

Per simulare in Network Simulator gli scenari presentati precedentemente dobbiamo implementare il comportamento del motore elettrico (il nostro plant) e del controllore usato.

Quindi bisogna, in primo luogo, calcolare la dinamica del

motore elettrico da implementare negli script di simulazione e per fare questo bisogna partire dal sistema (2.9) che riporto per comodità:

$$\begin{cases} \dot{x}_1 = -\frac{R_a}{L_a} x_1 - \frac{K_b}{L_a} x_2 + \frac{1}{L_a} u \\ \dot{x}_2 = \frac{K}{J} x_1 - \frac{B}{J} x_2 + \frac{1}{J} T_L \end{cases}$$

Essendo il sistema lineare, possiamo trasformare secondo Laplace e, ricordando che stiamo considerando $T_L = 0$, ottenere:

$$sX_1(s) - X_1(0) = -\frac{R_a}{L_a} X_1(s) - \frac{K_b}{L_a} X_2(s) + \frac{1}{L_a} U(s), \quad (3.3)$$

$$sX_2(s) - X_2(0) = \frac{K}{J} X_1(s) - \frac{B}{J} X_2(s) \quad (3.4)$$

Lavorando sulla (3.3) e sulla (3.4) si ottengono le espressioni di $X_1(s)$ e $X_2(s)$ in funzione di $X_1(0)$, $X_2(0)$ e $U(s)$ dove $X_1(s)$ e $X_2(s)$ rappresentano rispettivamente la trasformata di Laplace della corrente elettrica di armatura $i_a(t)$ e della velocità di rotazione $\omega(t)$ del motore, $X_1(0)$ e $X_2(0)$ indicano i valori iniziali delle trasformate e $U(s)$ si riferisce alla trasformata dell'ingresso del motore rappresentato dalla tensione di armatura $v_a(t)$.

Le equazioni alle quali si giunge sono:

$$X_1(s) = \frac{s + \frac{B}{J}}{D(s)} X_1(0) - \frac{K_b}{L_a} \frac{1}{D(s)} X_2(0) + \frac{1}{L_a} \frac{s + \frac{B}{J}}{D(s)} U(s) \qquad (3.5)$$

$$X_2(s) = \frac{\frac{K}{J}}{D(s)} X_1(0) + \frac{s + \frac{R_a}{L_a}}{D(s)} X_2(0) + \frac{\frac{K}{JL_a}}{D(s)} U(s), \qquad (3.6)$$

dove $D(s) = s^2 + \frac{BL_a + JR_a}{JL_a} s + \frac{BR_a + KK_b}{JL_a} = (s - p_1)(s - p_2)$ e

p_1 e p_2 sono i poli calcolati in precedenza nella (2.13).

Ora per trovare la dinamica del motore nel dominio del tempo bisogna antitrasformare la (3.5) e la (3.6) e, per questa operazione, è utile esprimere le due equazioni in forma modale o in rappresentazione di poli e zeri. Dopo l'operazione di antitrasformata otteniamo:

$$\begin{aligned} x_1(t) = {} & (A_{11} e^{p1t} + A_{12} e^{p2t}) \cdot x_1(0) + \\ & - (B_{11} e^{p1t} + B_{12} e^{p2t}) \cdot x_2(0) + . \\ & + (C_{11} e^{p1t} + C_{12} e^{p2t}) * u(t) \end{aligned} \qquad (3.7)$$

e

$$\begin{aligned} x_2(t) = &(A_{21}e^{p1t} + A_{22}e^{p2t}) \cdot x_1(0) + \\ &+ (B_{21}e^{p1t} + B_{22}e^{p2t}) \cdot x_2(0) + \\ &+ (C_{11}e^{p1t} + C_{12}e^{p2t}) * u(t), \end{aligned} \qquad (3.8)$$

dove si è posto per semplicità di notazione

$$A_{11} = \frac{p_1 + \frac{B}{J}}{(p_1 - p_2)}, \qquad A_{12} = \frac{p_2 + \frac{B}{J}}{(p_2 - p_1)},$$

$$B_{11} = \frac{\frac{K_b}{L_a}}{(p_1 - p_2)}, \qquad B_{12} = \frac{\frac{K_b}{L_a}}{(p_2 - p_1)},$$

$$C_{11} = \frac{A_{11}}{L_a}, \qquad C_{12} = \frac{A_{12}}{L_a},$$

$$A_{21} = \frac{\frac{K}{J}}{(p_1 - p_2)}, \qquad A_{22} = \frac{\frac{K}{J}}{(p_2 - p_1)},$$

$$B_{21} = \frac{p_1 + \frac{R_a}{L_a}}{(p_1 - p_2)}, \qquad B_{22} = \frac{p_2 + \frac{R_a}{L_a}}{(p_2 - p_1)},$$

$$C_{21} = \frac{A_{21}}{L_a}, \quad \text{e} \quad C_{22} = \frac{A_{22}}{L_a}.$$

Adesso è necessario sviluppare, per la (3.7) e la (3.8), la convoluzione dei termini in cui è presente $u(t)$.

$$u_1(t) = (C_{11}e^{p1t} + C_{12}e^{p2t}) * u(t) = \int_0^t (C_{11}e^{p1(t-\tau)} + C_{12}e^{p2(t-\tau)}) \cdot u(\tau) d\tau .$$

Se poniamo $u(\tau) = u(0)$ e $0 \le \tau \le t$, ricaviamo che

$$u_1(t) = \left[\frac{C_{11}}{p_1}(e^{p1t} - 1) + \frac{C_{12}}{p_2}(e^{p2t} - 1) \right] \cdot u(0). \qquad (3.9)$$

Utilizzando lo stesso procedimento per la seconda relazione,

$$u_2(t) = (C_{21}e^{p1t} + C_{22}e^{p2t}) * u(t) = \int_0^t (C_{21}e^{p1(t-\tau)} + C_{22}e^{p2(t-\tau)}) \cdot u(\tau) d\tau ,$$

e ponendo la stessa ipotesi $u(\tau) = u(0)$, otteniamo che

$$u_2(t) = \left[\frac{C_{21}}{p_1}(e^{p1t} - 1) + \frac{C_{22}}{p_2}(e^{p2t} - 1) \right] \cdot u(0). \qquad (3.10)$$

Ora, sostituendo la (3.9) nella (3.7) e la (3.10) nella (3.8) e raggruppando i termini mettendo in evidenza i termini esponenziali, otteniamo le dinamiche nel dominio del tempo di $x_1(t)$ e $x_2(t)$:

$$\begin{aligned} x_1(t) = {} & \left[A_{11}x_1(0) - B_{11}x_2(0) + U_{11}u(0) \right] \cdot e^{p1t} + \\ & + \left[A_{12}x_1(0) - B_{12}x_2(0) + U_{12}u(0) \right] \cdot e^{p2t} + \\ & - \left[U_{13} \right] \cdot u(0) \end{aligned} \qquad (3.11)$$

e

$$\begin{aligned} x_2(t) = &\left[A_{21}x_1(0) + B_{21}x_2(0) + U_{21}u(0)\right]\cdot e^{p1t} + \\ &+ \left[A_{22}x_1(0) + B_{22}x_2(0) + U_{22}u(0)\right]\cdot e^{p2t} + \\ &- \left[U_{23}\right]\cdot u(0), \end{aligned} \qquad (3.12)$$

dove si è posto

$$U_{11} = \frac{C_{11}}{p_1}, \qquad U_{12} = \frac{C_{12}}{p_2},$$

$$U_{21} = \frac{C_{21}}{p_1}, \qquad U_{22} = \frac{C_{22}}{p_2},$$

$$U_{13} = U_{11} + U_{12}, \qquad U_{23} = U_{21} + U_{22}.$$

Le relazioni (3.11) e (3.12) sono quelle effettivamente tradotte implementate nello script OTcl rispettivamente nelle funzioni `ode_motore` e `ode1_motore`.

Adesso bisogna tradurre in Network Simulator la dinamica del controllore nel dominio del tempo. La relazione che esprime tale dinamica è la (3.2), che riporto in seguito per comodità:

$$u(t) = K_P \cdot e(t) + K_I \cdot \int_0^t e(\tau)d\tau$$

Il comportamento del controllore PI (proporzionale-integrale) è stato effettivamente implementato nelle funzioni `ode_controllore` e `ode1_controllore`, in cui, rispettivamente, la prima implementa la dinamica del controllore P (proporzionale) [8]

e la seconda implementa la dinamica del controllore I (integrale).

3.3.2 Implementazione in MATLAB®

L'implementazione in MATLAB® dello scenario simulato è rappresentato dallo schema a blocchi mostrato in figura 3.7.

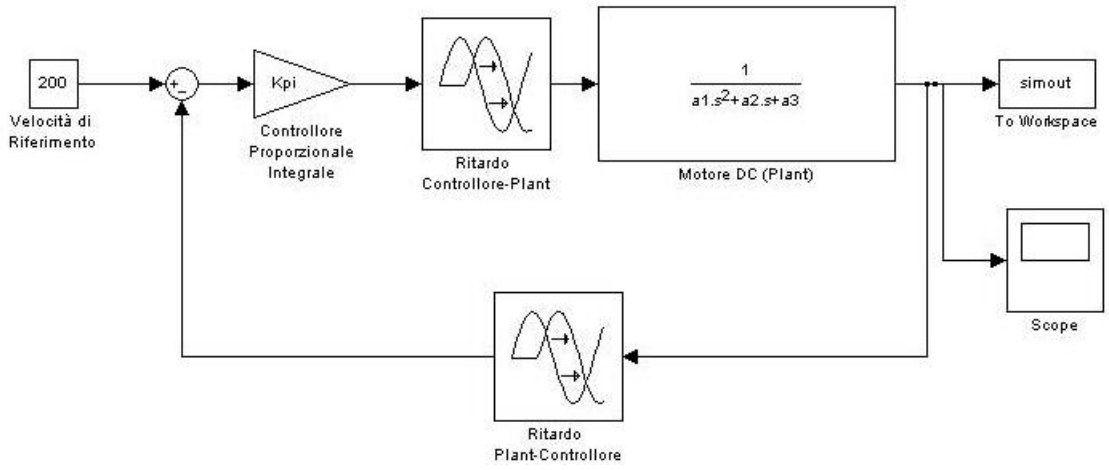

Fig. 3.7 – Schema a blocchi del sistema di controllo NCS di un motore DC

Come si evidenzia dalla figura 3.7, la dinamica del motore DC (plant) e del controllore proporzionale-intergrale sono rappresentate, rispettivamente, dalle funzioni di trasferimento $G_p(s)$ e $G_c(s)$ nel dominio di Laplace:

$$G_p(s) = \frac{1}{a_1 s^2 + a_2 s + a_3}, \qquad (3.13)$$

$$G_c(s) = K_P + \frac{K_I}{s}. \qquad (3.14)$$

La relazione (3.13) è stata ottenuta con il seguente procedimento. Inizialmente si è ricavata la trasformata di Laplace dell'equazione differenziale che descrive la dinamica del motore elettrico data dalla (2.10) del capitolo 2. Infatti, considerando nulle le condizioni iniziali si ha che

$$U(s) = a_1 s^2 \cdot Y(s) + a_2 s \cdot Y(s) + a_3 \cdot Y(s) = (a_1 s^2 + a_2 s + a_3) \cdot Y(s)$$

dove $U(s)$ indica la trasformata di Laplace dell'ingresso $u(t)$ del motore, mentre $Y(s)$ indica la trasformata di Laplace dell'uscita $y(t)$ del motore. Infine, si è calcolata la funzione di trasferimento $G_P(s)$ come:

$$G_P(s) = \frac{Y(s)}{U(s)}.$$

La relazione (3.14) è stata calcolata in maniera analoga. Trasformando nel dominio di Laplace la relazione (3.2)

$$U_{PI}(s) = K_P \cdot E(s) + \frac{K_I}{s} \cdot E(s) = \left(K_P + \frac{K_I}{s} \right) \cdot E(s)$$

e rapportando la trasformata dell'uscita $u_{PI}(t)$ del controllore con la trasformata dell'ingresso del controllore rappresentato dalla

funzione di errore $e(t)$:

$$G_c(s) = \frac{U_{PI}(s)}{E(s)}$$

Le relazioni (3.13) e (3.14) sono presenti nello schema a blocchi in figura 3.7.

Il mezzo trasmissivo viene tradotto in MATLAB® con il blocco ritardatore. Tale blocco tiene conto dei ritardi di propagazione introdotti dalla linea di trasmissione e dei tempi di trasferimento del pacchetto NCS. Per tale motivo esso è presente sia nel ramo diretto che nel ramo di retroazione e assume un valore pari al ritardo di propagazione della linea di trasmissione a cui è stato sommato il tempo di trasferimento del pacchetto NCS.

3.4 Modello di rumore implementato

Uno dei problemi che affliggono una qualsiasi rete di telecomunicazioni è rappresentato dal rumore elettromagnetico presente sul canale qualora si trasmettano dei dati. Inoltre, se i dati trasmessi siano usati per il controllo di un sistema fisico, il

rumore gioca un ruolo fondamentale nella stabilità del sistema fisico.

I parametri che quantificano l'effetto del rumore sul canale sono il *BER* (*Bit Error Ratio*) e il *FER* (*Frame Error Rate*). Il primo parametro è definito come il rapporto tra i bit corrotti durante la trasmissione e i bit correttamente ricevuti. Il secondo, invece, è definito come il tasso di frame corrotti, a livello di collegamento, rispetto al numero dei frame trasmessi.

Come vedremo nel capitolo successivo, i rumori implementati saranno di 2 tipologie: *Uniform* e *Bursty*. Il rumore Uniform è caratterizzato da un Frame Error Rate costante nel tempo. Infatti, detti N i pacchetti trasmessi, supporremo che su N pacchetti inviati, in maniera deterministica ne vengano corrotti K durante la trasmissione, con K costante. Il rumore Bursty, invece, è caratterizzato da un Frame Error Rate variabile nel tempo. Un valido modello di rumore Bursty, che verrà implementato in questo lavoro di tesi, è il modello di Gilbert [2].

Il modello in esame si basa una catena di Markov a due stati. La catena di Markov è definita come un processo markoviano a stati o valori discreti, dove per processo markoviano si intende processo stocastico in cui, scelto un certo istante t di osservazione,

l'evoluzione del processo, a partire da t, dipende solo da t mentre non dipende in alcun modo dagli istanti precedenti.

Gli stati del modello di Gilbert sono lo stato Good (G) e lo stato Bad (B). Nello stato Good, la trasmissione avviene senza errori. Nello stato Bad, il canale presenta una probabilità h di trasmettere senza errori. Il diagramma delle transizioni è riportato in figura 3.8.

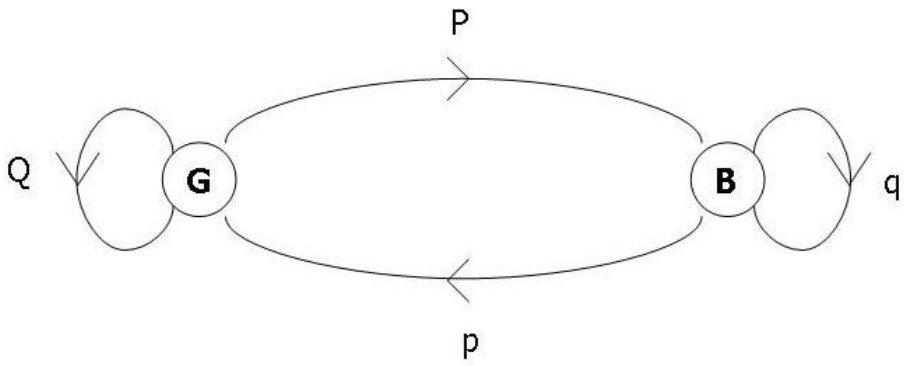

Fig. 3.8 – Diagramma di transizione del modello di Gilbert

Come è mostrato in figura 3.8, il canale presenta le seguenti probabilità:

- P : probabilità di passare dallo stato Good allo stato Bad;
- Q : probabilità di rimanere nello stato Good;
- p : probabilità di passare dallo stato Bad allo stato Good,

- q: probabilità di rimanere nello stato Bad;

Essendo il modello di Gilbert una catena di Markov a due stati, il processo in un generico istante di tempo si trova in uno dei due possibili stati. Se indichiamo con:

- T_B il Time Slot nello stato Bad,
- T_G il Time Slot nello stato Good,
- N_B il numero di Time Slot consecutivi nello stato Bad,
- N_G il numero di Time Slot consecutivi nello stato Good,

possiamo calcolare il tempo medio di permanenza nello stato Good e nello stato Bad.

Infatti, se ci troviamo nello stato Bad, la probabilità che nel Time Slot successivo rimaniamo nello stato Bad è $1-q$, la probabilità che nei due Time Slot successivi rimaniamo nello stato Bad è $(1-q)\cdot q$, la probabilità che nei tre Time Slot successivi rimaniamo nello stato Bad è $(1-q)\cdot q^2$, e così via. La probabilità che nel generico k Time Slot successivo rimaniamo nello stato Bad è $(1-q)\cdot q^{-1}$.

Il valore medio del numero di Time Slot nello stato Bad è

$$E[N_B] = \sum_{k=1}^{\infty} kq^{k-1}(1-q) = (1-q)\cdot\sum_{k=1}^{\infty} kq^{k-1} = (1-q)\cdot\frac{1}{(1-q)^2} = \frac{1}{(1-q)}$$

da cui si ricava il tempo medio di permanenza nello stato Bad, ricordando che $p = 1-q$:

$$E[T_B] = T_B \cdot E[N_B] = \frac{T_B}{1-q} = \frac{T_B}{p} \qquad (3.15)$$

Analogamente si ricava che:

$$E[T_G] = T_G \cdot E[N_G] = \frac{T_G}{1-q} = \frac{T_G}{p} \qquad (3.16)$$

Dalla (3.15) e dalla (3.16) si ricavano i tassi d'uscita dallo stato Bad e dallo stato Good, definiti rispettivamente come il reciproco del tempo medio di permanenza nello stato Bad e nello stato Good:

$$\mu = \frac{1}{E[T_B]} = \frac{p}{T_B} \qquad (3.17)$$

$$\lambda = \frac{1}{E[T_G]} = \frac{P}{T_G} \qquad (3.18)$$

I tassi μ e λ rappresentano, rispettivamente, la frequenza di media di transizione dallo stato Bad allo stato Good e la

frequenza media di transizione dallo stato Good allo stato Bad, così come mostrato in figura 3.9.

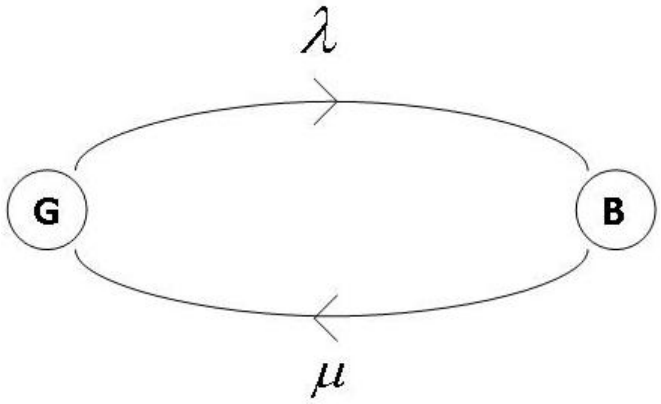

Fig. 3.9 – Diagramma di transizione del modello di Gilbert con riferimento ai tassi di uscita dagli stati

Ricordando che le variabili indipendenti del sistema sono p, P e h possiamo calcolare la probabilità P_B di trovarci nello stato Bad e la probabilità P_G di trovarci nello stato Good. Infatti, mediante una semplice procedura di calcolo basata sull'equilibrio dei flussi entranti e uscenti intorno a una superficie chiusa ($\lambda \cdot P_G = \mu \cdot P_B$) e utilizzando la congruenza che lega le probabilità di stato ($P_G + P_B = 1$), si ricava facilmente P_B e P_G:

$$\begin{cases} \lambda \cdot P_G = \mu \cdot P_B \\ P_G + P_B = 1 \end{cases} \Rightarrow \begin{cases} P_B = \dfrac{\lambda}{\lambda + \mu} \\ P_G = \dfrac{\mu}{\lambda + \mu} \end{cases} \Rightarrow \begin{cases} P_B = \dfrac{\dfrac{1}{E[T_G]}}{\dfrac{1}{E[T_G]} + \dfrac{1}{E[T_B]}} \\ P_G = \dfrac{\dfrac{1}{E[T_B]}}{\dfrac{1}{E[T_G]} + \dfrac{1}{E[T_B]}} \end{cases}$$

da cui svolgendo i calcoli si ottiene che

$$\begin{aligned} P_B &= \frac{E[T_B]}{E[T_G] + E[T_B]} \\ P_G &= \frac{E[T_G]}{E[T_G] + E[T_B]} \end{aligned} \qquad (3.19)$$

Calcoliamo il Frame Error Rate medio. Se T_F il tempo di trasmissione di un frame, supposto costante, in un intervallo di tempo pari a $E[T_B]$ vengono trasmessi un numero di frame pari a

$$N_F \cong \frac{E[T_B]}{T_F}. \qquad (3.20)$$

Dato che nello stato Bad la probabilità che il frame sia corrotto è $(1-h)$, allora il numero di frame errati è pari a

$$N_{FE} \cong (1-h) \cdot N_F \cong (1-h) \cdot \frac{E[T_B]}{T_F} \qquad (3.21)$$

Per il calcolo dei frame trasmessi correttamente, dobbiamo tener conto sia di quelli trasmessi nello stato Good, sia quelli trasmessi nello stato Bad, nel cui caso si ha una probabilità h di trasmettere un frame con successo. Da quanto appena detto, si ricava che

$$N_{FC} = N_{FC\_Good} + N_{FC\_Bad} \tag{3.22}$$

Il numero di frame correttamente inviati nello stato Good corrisponde al numero di frame trasmessi nell'intervallo di tempo medio di permanenza nello stato Good:

$$N_{FC} = \frac{E[T_G]}{T_F} \tag{3.23}$$

Al contrario, il numero di frame correttamente inviati nello stato Bad è pari al numero di frame trasmessi nel tempo medio di permanenza nello stato Bad (dato dalla (3.20)) per la probabilità h di trasmissione corretta:

$$N_{FC\_Bad} = N_F \cdot h = \frac{E[T_B]}{T_F} \cdot h \tag{3.24}$$

Quindi, sostituendo la (3.23) e (3.24) nella (3.22), otteniamo che

$$N_{FC} = \frac{E[T_G]}{T_F} + \frac{E[T_B]}{T_F} h \qquad (3.25)$$

Applicando la definizione di Frame Error Rate medio e lavorando con le relazioni (3.21), (3.25) e (3.19), si ottiene che

$$FER = \frac{N_{FE}}{N_{FE} + N_{FC}} = \frac{(1-h) \cdot \frac{E[T_G]}{N_F}}{(1-h) \cdot \frac{E[T_G]}{N_F} + \frac{E[T_G]}{T_F} + \frac{E[T_B]}{T_F} h} =$$

$$= \frac{E[T_G]}{E[T_G] + E[T_B]} \cdot (1-h) = P_B \cdot (1-h)$$

Quindi, se fissiamo il FER medio e conoscendo la probabilità h di trasmettere correttamente, possiamo calcolare il valore di P_B e successivamente il valore di p e P.

Infatti, noti T_B, T_G e T_F e scegliendo un valore di $E[T_B]$, si può ricavare il valore $E[T_G]$ e successivamente il valore di p e P utilizzando le seguenti relazioni

$$\begin{cases} E[T_B] = \frac{P_B}{1-P_B} \cdot E[T_G] \\ E[T_G] = \frac{1-P_B}{P_B} \cdot E[T_B] \end{cases} \qquad \begin{cases} p = \frac{T_B}{E[T_B]} \\ P = \frac{T_G}{E[T_G]} \end{cases},$$

le quali sono state ricavate dalla (3.17), (3.18) e (3.19). In questo possiamo dimensionare il modello di rumore Bursty che andremo ad implementare.

3.4.1 Moduli di errore nel Network Simulator

Nel Network Simulator è presente un modulo di errore che simula sia gli errori di livello fisico (link-level errors), sia le perdite dei pacchetti, tramite due distinti meccanismi. La corruzione del pacchetto è definita sfruttando un campo, “error flag”, presente nel common header del pacchetto stesso. Se settato a 1 da informazione al nodo ricevente dell’avvenuta corruzione del pacchetto. Al contrario, la perdita completa del pacchetto viene simulata consegnando il pacchetto, invece che al suo destinatario, ad un apposito “drop target”. In una generica simulazione, gli errori possono essere generati sia tramite un modello molto semplice, specificando un tasso di errore sui pacchetti, sia tramite modelli statistici ed empirici più complessi. Al fine di supportare una ampia varietà di tali modelli, l’unità di errore può essere specificata in termini di pacchetti o di bit oppure essere basata sul tempo (*time based*).

La classe `ErrorModel` è derivata dalla classe base `Connector`. Di conseguenza, essa eredita alcuna 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` definisce anche un metodo addizionale Tcl denominato `unit`: esso serve a specificare l'unità di errore desiderata dall'utente. E' previsto anche il metodo denominato `ranvar`, per specificare una variabile casuale per la generazione degli errori. A meno di specifiche contrarie, l'unità di errore è sempre espressa in pacchetti, mentre invece la variabile aleatoria è uniformemente distribuita nell'intervallo [0,1].

Il modello d'errore utilizzato, per simulare la catena di Markov implementata, è il "multi-state error". In questo sono implementate le transizioni di stato basate sul tempo (*timer based error state transition*) mediante l'uso di una matrice di stato. In pratica, le transizioni da uno stato al successivo, ad esempio dallo stato Good allo stato Bad, avvengono al termine di un intervallo di

tempo opportunamente calcolato. La scelta dello stato successivo viene effettuata facendo riferimento all'apposita matrice.

Per creare un modello di errore di questo tipo, bisogna impostare i seguenti parametri (definiti nel file *ns/tcl/lib/ns-errormodel.tcl*):

- `states`: vettore di stati (ogni stato corrisponde ad un diverso modello di errore);
- `periods`: vettore delle durate medie degli stati;
- `trans`: matrice delle transizioni di stato;
- `transunit`: da scegliere tra pacchetti, byte o secondi;
- `sttype`: tipo di transizioni di stato da usare, da scegliere tra secondi o pacchetti;
- `nstates`: numero di stati;
- `start`: stato di partenza.

3.4.2 Implementazione del Rumore Bursty

L'implementazione del modello d'errore Bursty viene effettuato utilizzando un apposito set di comandi, che riporto di seguito:

```
set good [new ErrorModel]
$good set rate_ 0.0
$good unit pkt

set bad [new ErrorModel]
$bad set rate_ $err
$bad unit pkt

set states [list $good $bad]
set periods [list 0.01 0.001]
set trans {{0.9 0.1} {0.1 0.9}}
set transunit pkt
set sttype pkt
set nstates 2
set start $good

set errmod [new ErrorModel/MultiState $states $periods $trans \
            transunit $sttype $nstates $start]
```

La variabile err è inserita da riga di comando, all'atto dell'avvio della simulazione. I valori assunti da tale variabile definiscono il fattore di perdita all'interno dello stato Bad e, di conseguenza, $1-h$ definito nel paragrafo 3.4. In particolare, tale variabile, in questo lavoro di tesi, assumerà i valori di 0.0001, 0.001, 0.01 e 0.1.

I valori T_G e di T_B sono settati rispettivamente a 0.01 s e 0.001 s. Questa scelta è stata presa in base a studi effettuati su metodi di progettazione e dimensionamento di modelli di rumore

per sistemi wired (cablati). La matrice di transazione degli stati assume i seguenti valori:

$$\begin{Bmatrix} Q & P \\ p & q \end{Bmatrix} = \begin{Bmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{Bmatrix}$$

Gli stati sono ovviamente due e lo stato di partenza è lo stato Good.