UNIVERSITÀ DEGLI STUDI DI PISA

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Scienze dell'Informazione

Anno Accademico 1997/1998

Tesi di laurea

Un sistema di controllo remoto di uno switching network per una applicazione di Fisica delle Alte Energie con visualizzazione grafica in Java.

Candidato

BOZZI Luciano

Relatori

Prof. G. Attardi

Prof. L. Zanello

Dott. S. Falciano

Controrelatore

Prof. G. Ferrari

Indice dei Capitoli

Introduzione

Il lavoro descritto in questa tesi riguarda la progettazione e realizzazione di un sistema di controllo remoto di un insieme di dispositivi hardware che possono costituire gli elementi base di un "*event builder*" per il sistema di acquisizione dati di un esperimento di Fisica delle Alte Energie, ATLAS, a LHC (Large Hadron Collider), il nuovo collider per protoni in corso di realizzazione al Cern (Ginevra).

L'obbiettivo principale ad ATLAS è la scoperta di una importante particella: il bosone di Higgs. Data l'elevata frequenza degli eventi di interesse prodotti dalle collisioni protone-protone (~10^8 eventi/s), essi vengono selezionati per mezzo di un sistema di trigger, suddiviso in tre livelli gerarchici, allo scopo di ridurre il rate di collisioni ad un valore in cui è possibile memorizzare i dati su supporto permanente per l'analisi successiva (10-100Hz).

Le latenze fissate per ogni livello impongono, oltre ad un elevato parallelismo, l'impiego di strutture d'interconnessione ad alte prestazioni tra i nodi di un livello e quelli del livello successivo. La scelta degli elementi base che costituiranno tali strutture è tuttora aperta; si dovrà tenere conto infatti delle loro caratteristiche in termini di prestazioni, ma anche della possibilità che essi offrono di effettuare il controllo ed il *monitoring* dell'intera struttura durante la fase di acquisizione dei dati.

Il presente lavoro di tesi, in linea con la seconda problematica, riguarda la realizzazione di un sistema di controllo e *monitoring* da remoto, basato sul

modello *client-server*, dello switch STC104 della SGS-Thomson, uno dei possibili candidati per la realizzazione delle strutture d'interconnessione tra nodi del sistema di trigger di ATLAS.

Nel primo capitolo viene riportata una descrizione generale di LHC, dell'esperimento ATLAS e del suo sistema di trigger ed acquisizione dati.

Nel secondo capitolo si analizzeranno le caratteristiche dell'elemento base con il quale si realizzerà un prototipo di struttura d'interconnessione, lo switch STC104 della SGS-Thomson.

Nel terzo capitolo si darà una descrizione delle problematiche inerenti il trasferimento di dati tra due piattaforme, connesse tra loro da un mezzo fisico di comunicazione, introducendo quindi l'intero sistema realizzato: un'agente di controllo (*server*) della rete di switch comandata via rete, attraverso l'invio di opportuni comandi, da un pannello grafico di visualizzazione e controllo.

Il quarto capitolo è dedicato alla descrizione dell'ambiente di programmazione Java, con il quale è stato implementato il processo remoto (*client*): una interfaccia grafica per il controllo della switching-network il cui programma, scritto con il linguaggio Java, viene eseguito da un dispositivo di calcolo astratto, la *Java Virtual Machine* (JVM), all'interno di un browser Web.

Nel quinto capitolo viene descritto il software che è stato sviluppato per la realizzazione dell'agente di controllo dei dispositivi e del pannello di visualizzazione grafica.

Capitolo 1

Descrizione dell’ esperimento ATLAS

La fisica delle Alte Energie si occupa dello studio delle componenti fondamentali della materia e della natura delle interazioni che intercorrono tra essi [Per87]. La continua ricerca di nuove particelle e la necessità di comprendere come le quattro forze fondamentali (forte, debole, elettromagnetica, gravitazionale) creano i legami tra esse, ha condotto, nel corso degli anni, alla costruzione di acceleratori di energia sempre maggiore.

In questo capitolo si vuole fornire una breve descrizione del nuovo acceleratore di particelle Large Hadron Collider (LHC) in costruzione presso il CERN, a Ginevra, e di uno dei due maggiori esperimenti, ATLAS, tramite il quale verranno rivelate le particelle prodotte nelle collisioni dei fasci di protoni e antiprotoni di LHC e ricostruiti gli eventi di interesse.

1.1 L’acceleratore di particelle LHC

Attualmente è in funzione al CERN (Ginevra) l’acceleratore di particelle LEP, la cui attività è previsto termini nell’anno 2000.

All’interno della sua cavità verrà costruito LHC, il nuovo collider circolare protone-protone costituito da due anelli di circonferenza pari a ~26 Km; all’interno di essi le particelle, raggruppate in pacchetti cilindrici lunghi ~7.5 cm, verranno accelerate nelle due direzioni opposte fino al raggiungimento dell’energia finale pari a 14 TeV. I pacchetti circolanti contemporaneamente negli anelli saranno 3600, e si avrà una collisione ogni 25 ns.

1.2 Il rivelatore ATLAS

Il rivelatore ATLAS è un apparato sperimentale, tuttora in fase di sviluppo, approvato per la fisica all’acceleratore LHC che coinvolge 144 istituti di ricerca in tutto il mondo[ATL94].

Scopo principale degli esperimenti a LHC è la scoperta del bosone di Higgs che, dopo la conferma dell’esistenza del quark top, è l’ultimo elemento mancante al quadro completo definito dal Modello Standard. Poiché però la disponibilità di una nuova regione di energia potrebbe condurre a risultati e scoperte imprevisti, il rivelatore è progettato in modo da poter sfruttare tali potenzialità.

La figura 1.1 mostra il rivelatore nel suo insieme:

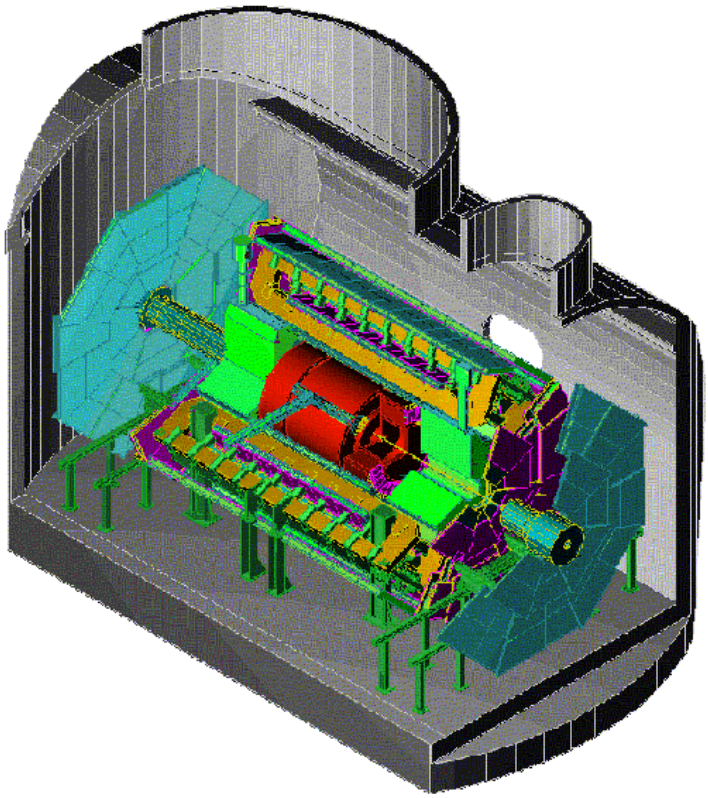

Fig. 1.1: Vista tridimensionale del rivelatore ATLAS

La sua forma è cilindrica e coassiale con l'anello di LHC. È composto da 3 rivelatori principali, ognuno avente particolari caratteristiche in relazione ai diversi tipi di particelle da rivelare:

- un rivelatore interno (**Inner Detector**), in grado di identificare e ricostruire le traiettorie di una particella carica;
- due **calorimetri**, il cui scopo è quello di fornire delle misure dell'energia delle particelle, sia cariche che neutre, che li attraversano;

· uno **spettrometro per muoni**, al fine di ottenere un'alta precisione delle misure di specifiche particelle, i muoni appunto.

1.3 Il sistema di trigger e acquisizione dati

Il sistema di trigger dell'esperimento ATLAS seleziona in tempo reale gli eventi prodotti dalle collisioni protone-protone nel rivelatore interno, nei calorimetri e nello spettrometro per muoni.

Considerando che in LHC, come detto nel paragrafo 1.1, vi è una collisione ogni 25 ns (40 MHz) e che per ciascuna di esse il volume dei dati prodotti è pari a ~1 Mbyte, il flusso totale dei dati provenienti dall'acceleratore, se si volessero registrare tutti gli eventi di collisione, sarebbe pari a ~40 Tbyte/s. In realtà, solo una piccolissima frazione degli eventi prodotti (~$1/10^7$) è di interesse per la successiva analisi fisica. Occorre pertanto progettare e realizzare una serie di dispositivi per la selezione in linea (*trigger*) delle collisioni da memorizzare.

Il sistema di trigger è suddiviso in tre livelli gerarchici, come illustrato in figura 1.2:

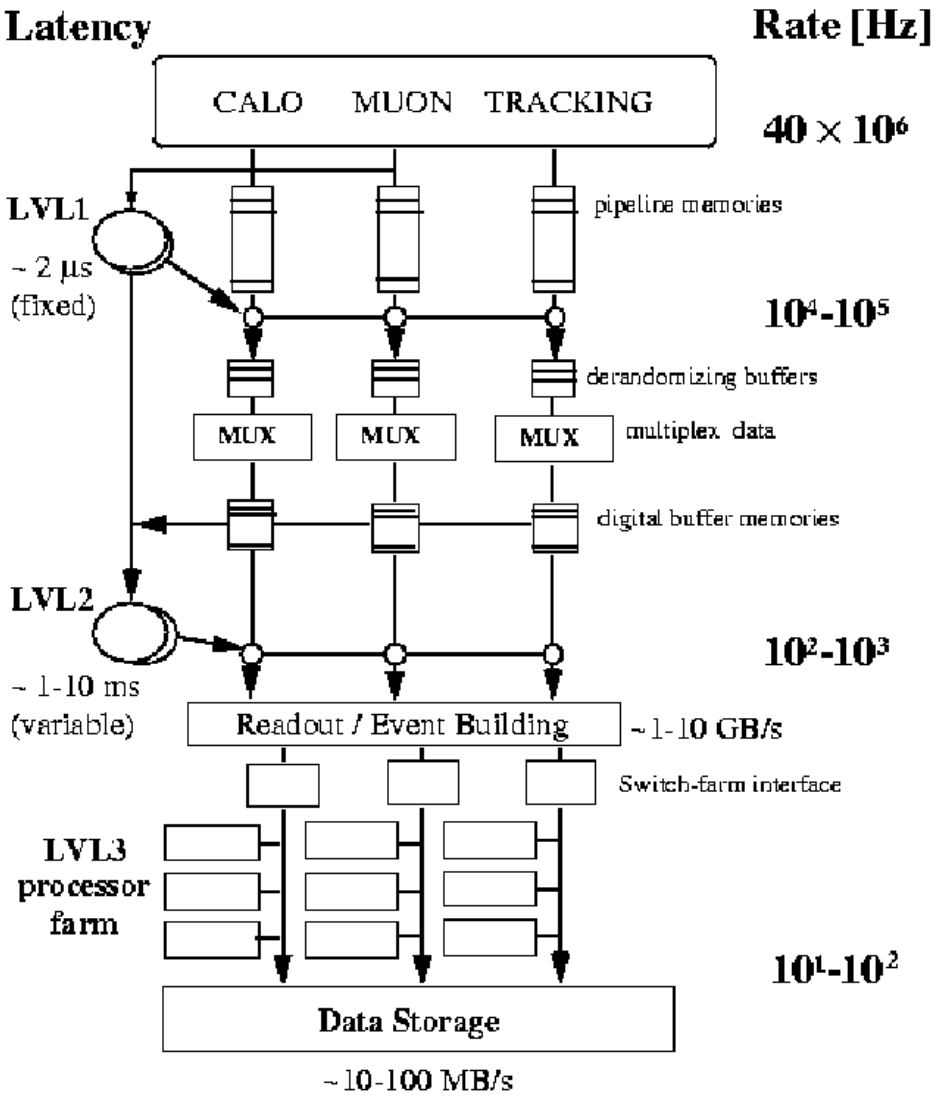

Fig. 1.2: Schema del sistema di trigger di ATLAS

Il limite posto da capacità di elaborazione dei computer *offline* e dallo spazio di storage dei dati, ha portato ad un valore di 10-100 Hz di frequenza

di scrittura su supporto magnetico (figura 1.2) che, con una quantità di dati dell'ordine di 1 Mbyte/evento, comporta un flusso di 10-100 Mbyte/s.

All'interno di ciascun livello si cerca quindi un compromesso tra la frequenza degli eventi da gestire e la complessità degli algoritmi che devono essere applicati ai dati provenienti dal livello precedente.

Il primo livello deve essere in grado di accettare dati alla frequenza di **40 MHz** (frequenza di collisione di LHC) con una latenza (tempo necessario per calcolare e trasmettere la decisione sull'evento) pari a **~2μs**: durante questo intervallo di tempo i segnali provenienti dai rivelatori vengono memorizzati in memorie *pipeline*. La massima frequenza di uscita dei dati è limitata a **100KHz**. Grazie a questa prima selezione dei dati, vengono identificate le cosiddette Regioni di Interesse (**R**egion **o**f **I**nterest, RoI) le quali contengono le informazioni da trasmettere al trigger di secondo livello.

Il secondo livello, la cui latenza massima è pari a **10 ms**, deve ridurre la frequenza dei dati in uscita di un fattore 100 (**1 KHz**). Grazie alla maggiore latenza, questo livello può applicare algoritmi più complessi sui dati relativi alle RoI, decidendo così se un evento è da rigettare oppure deve essere trasmesso al trigger di terzo livello. Nella figura 1.3 che segue viene riportato lo schema generale di questo sottosistema:

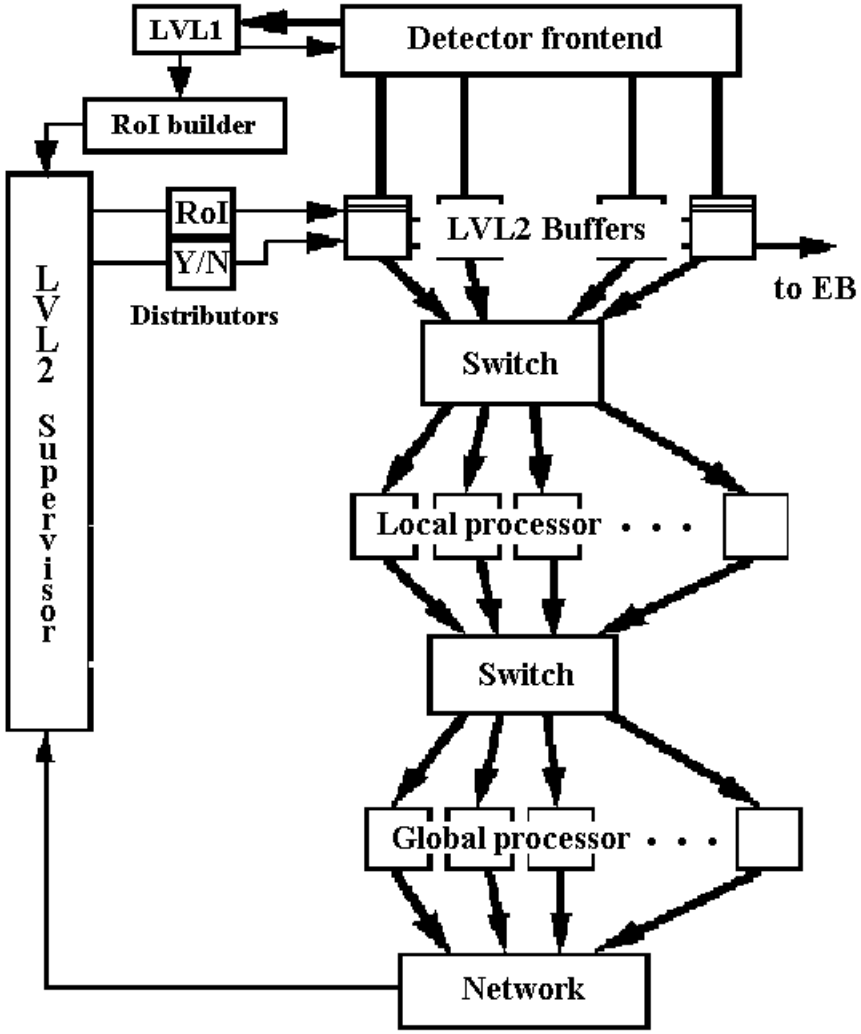

Fig. 1.3: Schema del trigger di 2° livello

Una prima elaborazione viene effettuata dai processori **locali** (figura 1.3), i quali ricevono i dati, sotto il controllo del **LVL2 Supervisor**, da circa 2000 **ROB** (Read Out Buffer); una prima switching network (**Local Switching Network**) si rende necessaria per connettere i ROB con i processori, denominati locali in quanto analizzano le RoI di ciascuno dei tre rivelatori.

Di fatto, la Local Switching Network è effettivamente costituita da tre strutture d'interconnessione, ciascuna relativa ai ROB di uno specifico rivelatore, come mostrato in figura 1.4, dove le sigle **TRT** ed **e.m.+adronic** si riferiscono rispettivamente al rivelatore interno ed ai due calorimetri:

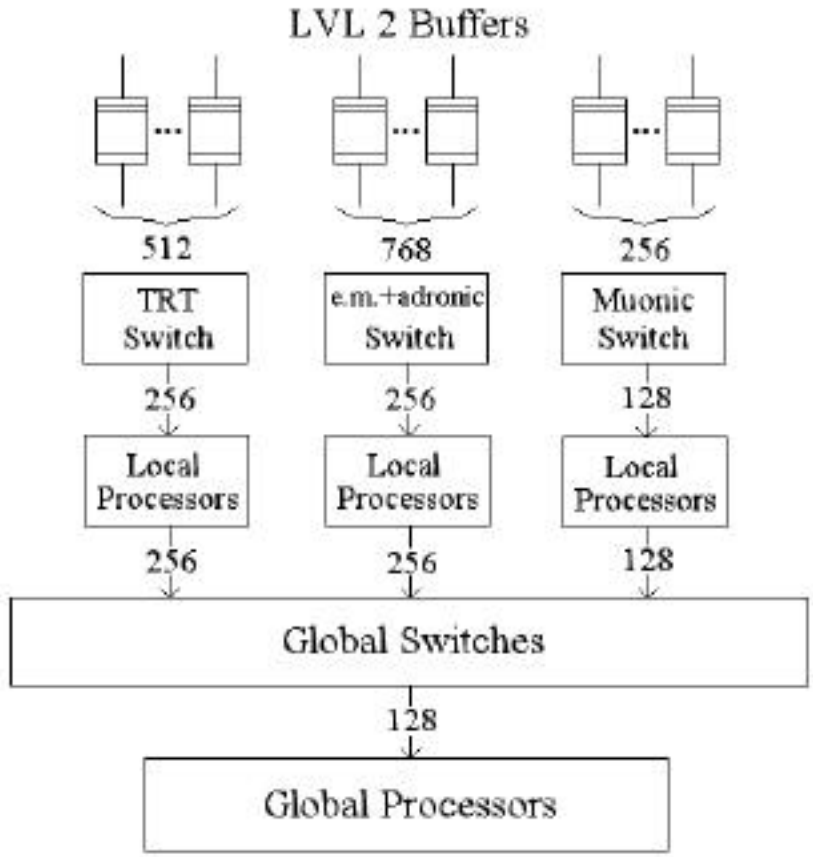

Fig. 1.4: Struttura delle reti locali e globale del trigger di 2° livello

Un' altra struttura d'interconnessione, la **Global Switching Network**, è richiesta al fine di trasferire i dati ai processori **globali**, che devono

elaborare e correlare le informazioni provenienti da più rivelatori per decidere se l'evento è da accettare, e quindi trasmetterlo al trigger di terzo livello, oppure da scartare.

Il terzo livello elabora i dati dell'evento e memorizza, con una frequenza massima di 100 eventi/s (la dimensione del singolo evento è, come detto, dell'ordine di 1 Mbyte), le informazioni finali per l'analisi *offline*. Poiché presenta una complessità maggiore dei precedenti livelli, esso impone delle latenze di qualche secondo.

Gli elementi critici della struttura hardware del LVL2 sono i ROB e gli switch con i quali realizzare le strutture d'interconnessione descritte.

Infatti, per quanto riguarda i processori, sia locali che globali, è previsto che vengano utilizzati elementi commerciali standard, sfruttando al massimo il miglioramento delle prestazioni hardware e software che senza dubbio si avrà nei prossimi anni, a parità di costi con i processori attualmente disponibili sul mercato.

Per quanto riguarda i ROB, invece, è difficile pensare all'utilizzo di sistemi commerciali in quanto non si tratta di semplici moduli di memoria che bufferizzano i dati in attesa di essere elaborati dal LVL2; in primo luogo, le porte di comunicazione devono permettere l'input dal LVL1, l'output a LVL2, I/O di segnali di controllo per le richieste dei dati provenienti dai processori locali e l'output a LVL3 se l'evento è accettato. Inoltre sarà necessario dotarli di una CPU per il preprocessamento e la formattazione dei dati per l'elaborazione e la decisione del LVL2. Per tali

ragioni, è probabile che la soluzione da adottare richiederà lo sviluppo di hardware specializzato insieme all'uso di sistemi commerciali.

Anche per quanto riguarda gli switch, la scelta tra semplice acquisto di prodotti standard e sviluppo di sistemi dedicati è un problema ancora aperto:

- i prodotti disponibili sul mercato, oltre ad essere sviluppati principalmente per le telecomunicazioni, hanno alti costi e non sono ottimizzati per il tipo di flusso dati previsto nell'esperimento[1]. Inoltre, è difficile apportare ad essi modifiche sostanziali all'hardware ed al software di controllo;
- lo sviluppo di sistemi per uso speciale ha lo svantaggio di richiedere lunghi tempi di progettazione, realizzazione e messa a punto, con il rischio di ottenere un prodotto finale di tecnologia superata. D'altro canto questa soluzione è certamente meno costosa ed ha il vantaggio di permettere il completo controllo ed adattamento del dispositivo al problema.

Una soluzione interessante potrebbe essere quella che prevede l'uso di elementi base standard inseriti in una struttura modulare che permetta l'espansione della configurazione iniziale del sistema e la sua ottimizzazione secondo lo sviluppo dell'esperimento. Con tale tipo di approccio, è fondamentale strutturare il progetto in livelli logici e funzionali, in modo da

[1] A titolo d'esempio, gli switch ATM (***A****synchronous* ***T****ransfer* ***M****ode*) [MTW93] sono attualmente dotati di 16 canali di comunicazione, la dimensione dei messaggi che gestiscono è fissa e pari a 53 byte (5 byte per le informazioni di *routing* e 48 byte di carico utile), ed il loro costo è dell'ordine delle migliaia di dollari.

permettere l’eventuale sostituzione degli elementi base con altri di prestazioni migliori senza peraltro compromettere l’intera struttura.

In questo lavoro si analizzano le caratteristiche di uno di tali elementi base, precisamente lo switch STC104 della SGS Thomson, come di un possibile candidato per un prototipo di switching-network per il trigger di livello 2 di ATLAS, realizzando inoltre un sistema di controllo remoto dei dispositivi, comandati via rete tramite un pannello di visualizzazione e controllo realizzato con l’ambiente di sviluppo Java.

Capitolo 2

Il protocollo IEEE 1355 e la sua realizzazione hardware

I componenti hardware utilizzati rispondono alle specifiche del protocollo di comunicazione IEEE 1355 [Str97]. Il presente capitolo vuole fornire una descrizione di questo protocollo e dei vari componenti elettronici, utilizzati durante lo svolgimento di questa tesi, mediante i quali esso viene realizzato.

Lo IEEE 1355 nasce come protocollo di interconnessione seriale tra dispositivi elettronici. Esso riguarda innumerevoli campi di applicazione nei quali è richiesta una connettività globale a bassa latenza, basso costo, di semplice implementazione, scalabile. Applicazioni tipiche sono la realizzazione di LAN, calcolo parallelo, server multimediali, *networking* e sistemi di acquisizione dati, quali appunto quello di ATLAS.

La possibilità di applicazione dello IEEE 1355 in campo audio, video, e la compatibilità (ad alto livello) che esso ha nei confronti di altri protocolli (TCP/IP, ATM, ecc.), nascono fondamentalmente dalla libertà di poter scegliere la lunghezza dei pacchetti da trasmettere. Un’altra caratteristica fondamentale dello IEEE 1355, che permette comunicazioni asincrone, è la

implementazione di memorie FIFO (First In First Out) tra i vari host interconnessi. In questo modo le differenti velocità dei dispositivi non limitano le comunicazioni tra essi. La realizzazione hardware dello IEEE 1355 viene effettuata da una serie di chip che saranno descritti nel seguito del capitolo e che rispondono al protocollo IEEE 1355 DS-DE (denominato anche DSLink). Come vedremo quest'ultimo completa lo IEEE 1355, mediante l'aggiunta di una serie di specifiche per il controllo del flusso di dati e per la gestione delle informazioni da trasferire.

2.1 Il Protocollo IEEE 1355 DS-DE

Lo IEEE 1355 DS-DE (DSLink) è un protocollo che integra lo IEEE 1355 con alcune specifiche più dettagliate sui segnali utilizzati nelle comunicazioni. Viene utilizzato per collegamenti punto-punto e permette di effettuare trasmissioni bidirezionali asincrone, con controllo di flusso a vari livelli gerarchici. Il canale DSLink è costituito da 8 linee sulle quali viaggiano i segnali (differenziali) di Data e di Strobe, 4 per ogni direzione; è attualmente in grado di raggiunge velocità di trasmissione di 100 Mbit/s. Il protocollo DSLink prevede 5 livelli gerarchici:

1 elettrico;
2 a livello di bit;
3 a livello di token;
4 a livello di pacchetto;
5 a livello di messaggio.

I primi tre livelli vengono realizzati in modo automatico dai dispositivi hardware che vedremo nel seguito del capitolo (STC101 ed STC104). La pacchettizzazione ed il *messaging* non vengono invece eseguiti automaticamente, ma possono essere realizzati dall'utente mediante alcuni strumenti forniti dallo stesso hardware. Sostanzialmente i tre livelli inferiori permettono di realizzare sia un controllo sul flusso dei dati (rivelazione di errori sulla parità), sia uno handshake tra i dispositivi connessi, il tutto senza alcun intervento esterno. La capacità di gestire pacchetti e messaggi permette invece di aggiungere ai dati delle informazioni addizionali, dette di *routing*, quali l'indirizzo della destinazione ed un segnale indicante la loro terminazione. Mediante queste ultime è così possibile trasferire le informazioni tra più di due host, e attraversare delle *switching network* comunque complesse (come ad esempio quelle presenti nel trigger di 2° livello di ATLAS). Analizziamo nel dettaglio i 3 livelli superiori del protocollo, rimandando il lettore a [MTW93] per la descrizione dei primi due.

Protocollo a livello di token.

I "token" sono delle stringhe di bit costituite da segnali di dato o di controllo. Questi vengono gestiti, in modo invisibile ai livelli superiori, dai singoli link, i quali sono provvisti di un'apposita elettronica. I token possono essere di due tipi: **data token** e **control token**. I primi sono costituiti da 10 bit di cui 8 di dato, uno di parità dispari (P) ed uno di controllo del token stesso (C). I control token hanno invece 4 bit: i primi due indicano la codifica dell'informazione trasportata (secondo la legenda indicata nella tabella 2.1), mentre gli altri due sono gli stessi del caso precedente (C e P).

In entrambi i token la funzione del bit C è quella di indicarne il tipo: se C=0 allora il token in esame è del tipo data, altrimenti (C=1) è un control token. Pertanto, data token e control token si contraddistinguono grazie al valore del bit di controllo e al diverso numero di bit in essi contenuti.

Dal momento che la lunghezza dei due tipi di token è differente, il bit di parità viene impostato considerando lo stesso bit P, il bit di controllo C del token in esame e tutti i bit costituenti il "corpo" del token precedente, i quali sono 8 per i data token e 2 per i control token. In questo modo si riescono ad identificare errori singoli su tutti i bit della stringa, incluso quello di controllo.

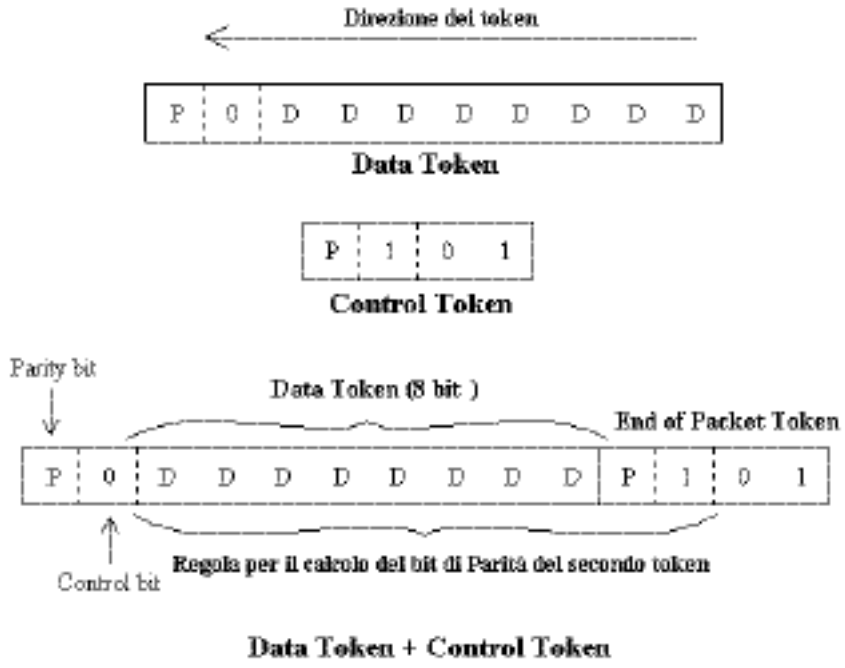

Figura 2.1: Protocollo a livello di Token.

La parità P assicura che il numero di “1” presenti nei bit considerati sia sempre dispari.

Funzione del Token	**Codice**	**Valore dei bit del token di controllo (P + C + xx)**
Flow Control Token	FCT	P100
End Of Packet	EOP	P101
End Of Message	EOM	P110
ESCape token	ESC	P111
NULL token	NUL	ESC + P100[1]

Tabella 2.1: Codifica dei token di controllo.

Il controllo di flusso a livello di token viene implementato dal protocollo DSLink per effettuare uno handshake tra la sorgente ed il buffer di destinazione: quando quest'ultimo ha spazio a sufficienza per almeno 8 data token, invia un FCT alla sorgente la quale spedisce i dati, ed attende un altro FCT. I token di controllo non devono necessariamente essere inviati dopo ogni data token ma soltanto in caso di necessità. Ad esempio, se un dato è costituito da 32 bit e viene organizzato in un unico pacchetto avente un byte di testa, per trasmetterlo si devono inviare 5 data token successivi (uno per la testa e quattro per il dato), seguiti da un control token del tipo EOP.

[1] Il NULL token è costituito da due comandi successivi.

Protocollo a livello di pacchetto e di messaggio.

A livello più elevato il protocollo DSLink prevede la pacchettizzazione dei dati ed il *messaging* degli stessi. Ciò consiste nel raggruppare le informazioni da spedire in pacchetti di lunghezza variabile (espressa in byte) aggiungendo ad essi una "testa" ed una "coda", al fine di rendere possibile l'instradamento attraverso gli switch STC104. La testa, che è costituita da uno o più data token, può avere una lunghezza variabile e definisce la destinazione del pacchetto. La coda indica invece la sua terminazione. Quest'ultima, costituita da un unico control token, può essere un segnale del tipo **EOP** (End Of Packet) oppure **EOM** (End Of Message) ed indica se il pacchetto in esame è l'ultimo di un messaggio oppure no. Inoltre, un pacchetto costituito da una testa ed un "end of packet" (senza byte di dato) costituisce un **acknowledge packet**. La figura 2.2 riassume quanto appena espresso:

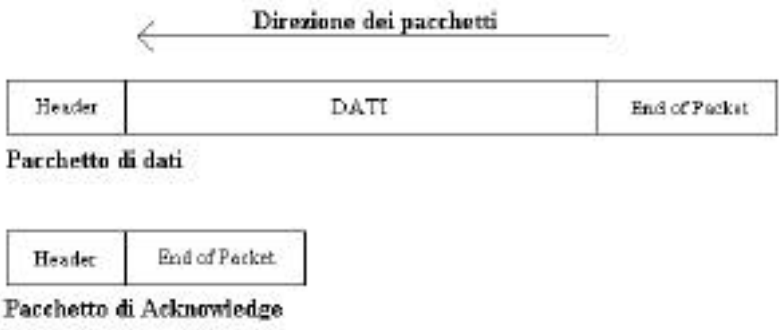

Figura 2.2: Protocollo DSLink a livello di pacchetto.

I chip SGS Thomson STC101 ed STC104, utilizzati nello sviluppo di questo lavoro e descritti nel seguito, implementano in modo automatico il protocollo DSLink fino al 3° livello incluso.

2.2 Lo switch STC104

Lo STC104 è uno switch elettronico asincrono in tecnologia **VLSI** costituito da 32 DSLink (**Data Link 0-31**) in grado di interconnettere tra loro 32 dispositivi in modo indipendente. Nella figura 2.3 viene riportato il suo diagramma a blocchi:

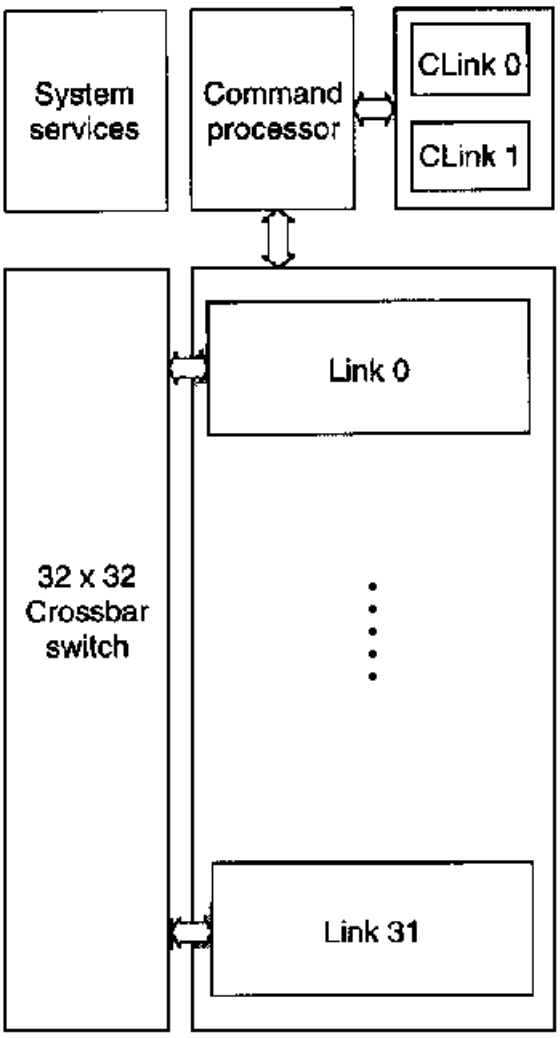

Figura 2.3: Schema a blocchi del C104.

Il C104 è l'elemento base delle *switching network* locali e globali che si vogliono realizzare per i prototipi di trigger di 2° livello di ATLAS. Come si può notare ogni link ha la sua elettronica di processamento dei pacchetti ed i collegamenti interni sono assicurati da un *crossbar switch* 32x32. Ciò rende

i link indipendenti tra loro e 32 dispositivi (host) possono essere connessi tramite un unico STC104, raggiungendo una larghezza di banda totale di 300 Mbyte/s [SGS95a].

La programmazione di questo dispositivo avviene a livello firmware tramite due link separati denominati **Clink0** (ingresso) e **Clink1** (uscita) i quali utilizzano lo stesso protocollo dei link di dato. Il Clink0 permette l'accesso ad una serie di registri interni mediante i quali è possibile configurare delle reti di C104 in termini di parametri di comunicazione quali velocità di trasmissione e lunghezza della testa dei pacchetti: tramite questi parametri si implementa il protocollo DSLink a livello di pacchetto. Il Clink1 propaga il segnale precedente al dispositivo successivo. Avendo i link lo stesso protocollo, si possono realizzare delle connessioni tra data link e control link.

L’accesso ai registri del C104 avviene mediante l’invio di token, secondo un protocollo di handshake; i vari comandi sono codificati in questi pacchetti i quali, in generale, contengono il codice del comando, l’indirizzo del link di destinazione, il dato (nel caso di operazioni di scrittura) ed un terminatore (figura 2.4).

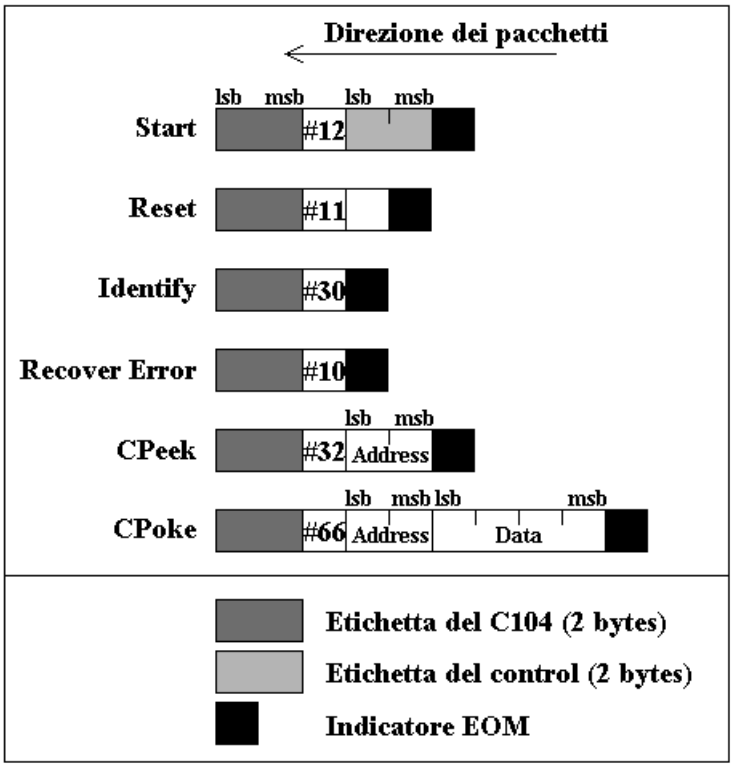

Figura 2.4: Pacchetti di comando inviabili al C104.

La presenza dell'indirizzo di destinazione rende possibile il raggiungimento, da parte dei comandi, di uno qualsiasi dei link della rete.

Il primo comando che il controller invia ad uno dei link (o anche a tutti insieme) è uno *start token*, contenente tra le altre informazioni, il suo indirizzo di ritorno. Il C104 esegue il comando e risponde con un pacchetto di handshake che dipende sia dal comando che ha ricevuto, sia dal suo stato attuale.

Dal momento che queste operazioni vengono svolte dai programmi sviluppati nel corso del lavoro di tesi, la loro descrizione viene fatta nel

capitolo 5, quello dedicato alla descrizione del software sviluppato. Inoltre, essendo i pacchetti di handshake del C104 codificati in modo analogo, non vengono riportati in questa sede, ma sono comunque disponibili sullo *Engineering Data* del C104 [SGS95a].

Si vuole ora entrare nel merito della programmazione a basso livello del dispositivo, descrivendo alcune delle funzioni svolte dai suoi registri più significativi.

I **Link0-31Command** sono 32 registri di sola scrittura, aventi ciascuno 4 bit programmabili singolarmente. I primi due bit di ognuno di essi permettono di effettuare il reset (**ResetLink**) e lo start (**StartLink**) del link corrispondente. Il terzo (**ResetOutput**), se posto ad "1", effettua il reset dei due segnali Data e Strobe, mentre il quarto (**WrongParity**) permette di forzare una parità scorretta.

Altri registri di lettura e scrittura molto importanti sono i **Link0-31Mode**. I primi 3 bit di questi ultimi permettono di impostare la velocità di trasmissione, mentre il quarto, se posto ad "1", in caso di errore scarica il pacchetto in transito sul link corrispondente (ciò fa sì che il link non rimanga bloccato).

I 32 registri di lettura e scrittura **PacketMode0-31** hanno 5 bit accessibili. Tra le impostazioni che questi permettono di effettuare si ricorda la possibilità di variare la lunghezza della testa dei pacchetti attesi in ingresso (da 1 a 2 byte) e la capacità di cancellare il primo byte del pacchetto in uscita (**Header Deletion**).

La cancellazione della testa è molto importante quando, all'interno delle reti, si vogliono raggiungere delle destinazioni intermedie (come nel caso

dell'algoritmo *two-phase routing*, descritto nel paragrafo 2.2.3). Inoltre, se un nodo di arrivo ha più destinazioni (ad esempio diversi processi in esecuzione su un singolo processore) la prima testa del pacchetto viene utilizzata per raggiungere il nodo, la seconda per far giungere i dati alla destinazione all'interno del nodo.

Per ottenere delle informazioni sullo stato dei 32 link si possono interrogare i registri **Link0-31Status** (di sola lettura). Per ogni link, i 6 bit accessibili indicano rispettivamente se:

- è avvenuto un errore;
- il link è partito correttamente;
- il reset output è stato completato;
- è avvenuto un errore di parità;
- è avvenuto un errore di disconnessione;
- è stato ricevuto un token (dopo l'ultimo reset link).

Quest'ultimo gruppo di registri è stato scelto, come sarà descritto nei prossimi capitoli, per monitorare lo stato della rete e per rappresentare graficamente i link degli STC104 sul pannello di controllo remoto.

Per la descrizione completa di tutti i registri degli STC104 si rimanda il lettore alla bibliografia [SGS95a].

2.2.1 Il *wormhole routing*

Nel paragrafo seguente ed in quello successivo vengono descritti l'importante algoritmo di routing dei pacchetti del protocollo IEEE 1355 e la sua realizzazione a livello hardware.

Come già detto, la testa dei pacchetti viene utilizzata per instradarli all'interno dello switch, mentre la coda permette loro di avere una lunghezza qualsiasi, in quanto è proprio essa che indica che il pacchetto è terminato. La coda ha comunque un'altra funzione molto importante: essa consente infatti di realizzare il *wormhole routing*. Quando un pacchetto viene inviato nello switch, al suo interno si crea un percorso fisico che scompare quando è transitata la sua coda. La figura 2.5 esemplifica questo comportamento:

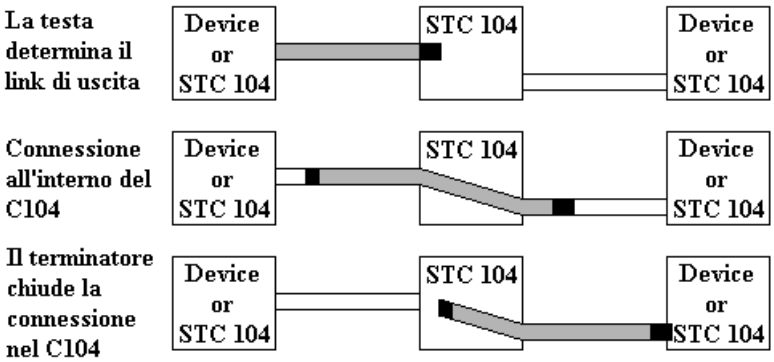

Figura 2.5: Wormhole routing all'interno di un STC104.

In questo modo i link dello switch non vengono impegnati per tutto il tempo di transito dei pacchetti, e dati successivi possono essere inviati attraverso lo stesso percorso prima ancora che il dato precedente sia giunto a

destinazione. Così vengono ridotti i tempi di latenza della rete, soprattutto nel caso in cui quest'ultima sia formata da diversi STC104 connessi in serie.

La caratteristica fondamentale del *wormhole routing* è quella di limitare la latenza dei pacchetti.

2.2.2 *Interval labelling*

Vedremo ora nel dettaglio come il routing dei pacchetti viene realizzato a livello hardware, ovvero come le loro teste vengono interpretate ed instradate all'interno del C104.

Ogni link dello switch possiede 36 registri a 32 bit denominati **Interval1-36**, programmabili indipendentemente gli uni dagli altri. I primi 16 bit di questi registri sono detti **Separator** mentre i 5 bit dal 18° al 22° vengono denominati **SelectLink**. Nei campi Separator vengono memorizzati dei valori contigui crescenti (con intersezione nulla), mentre i SelectLink contengono i numeri dei link associati ai corrispondenti intervalli; lo Interval0 non è programmabile e contiene il valore 0. Quando un pacchetto arriva su un link, viene effettuato un confronto tra la sua testa ed i valori memorizzati nei Separator; a partire dal Separator1, il primo di questi registri che contiene un valore maggiore della testa in esame indica, tramite il registro SelectLink ad esso associato, la strada che il pacchetto deve seguire, ovvero il link di destinazione. La figura 2.6 mostra la tecnica utilizzata per l'instradamento all'interno degli STC104.

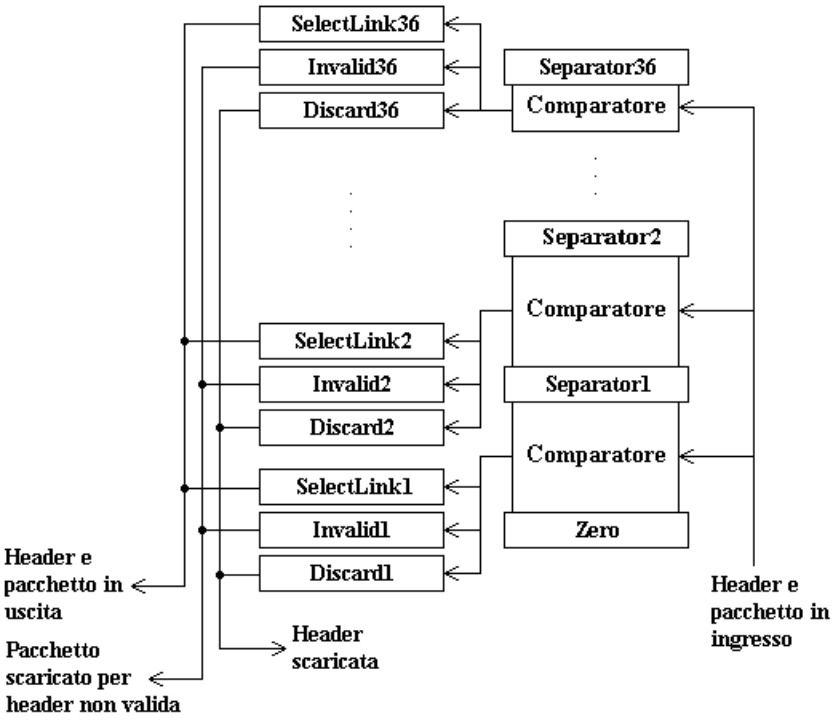

Figura 2.6: Registri mediante i quali viene effettuato il routing dei pacchetti.

E' evidente la necessità di avere degli intervalli crescenti e senza intersezioni per evitare che vi siano dei pacchetti con delle destinazioni multiple o ambigue. Si noti inoltre che il pacchetto può essere inviato sullo stesso link di ingresso formando così un collegamento con se stesso (i link sono bidirezionali). Questa tecnica, nella quale ad ogni link di uscita è assegnato un intervallo mediante delle etichette, è denominata *interval labelling*. Nei registri Interval1-36 sono inoltre presenti 2 bit denominati **Discard** ed **Invalid** mediante i quali si può riconoscere se la testa di un pacchetto non è nell'intervallo, ed eventualmente scaricarla insieme al pacchetto stesso.

2.2.3 *Two-phase routing*

In strutture d'interconnessione nelle quali molti host "attivi" sono collegati tramite più STC104, vi possono essere dei link soggetti ad un alto numero di conflitti che costringono i pacchetti a delle attese elevate. Per *switching network* affette dalla presenza di tali punti, detti *hot spot*, la maggior parte delle volte non è possibile effettuare una stima della latenza, in quanto questa non dipende soltanto dal traffico globale della rete, bensì dal traffico sul punto caldo. Sotto queste condizioni le prestazioni delle reti possono diminuire drasticamente. Il *two-phase routing* è un algoritmo che può essere utilizzato al fine di eliminare il problema degli hot spot; consiste nello spedire il pacchetto ad una destinazione intermedia (scelta in maniera pseudo-casuale tra gruppi di link predefiniti) dalla quale poi i dati vengono instradati verso il link finale. Con questa tecnica, le percentuali di occupazione dei diversi percorsi tendono ad equivalersi. Nella documentazione [SGS95a] tale algoritmo è indicato con il nome *Universal Routing*.

Scegliere delle destinazioni in modo pseudo-casuale con la tecnologia descritta è possibile grazie ai registri **PacketMode0-31**, **RandomBase0-31** e **RandomRange0-31** dello STC104. Nel primo è presente un bit, denominato "Randomize" che, se posto ad "1", aggiunge una testa al pacchetto che lo attraversa, in base alle modalità impostate negli altri 2 registri. In particolare la generazione pseudo-casuale delle teste avviene nel range RandomBase ÷ (RandomBase + RandomRange –1).

2.2.4 *Grouped adaptive routing*

Il *grouped adaptive routing* è una ulteriore implementazione del protocollo IEEE 1355 realizzabile tramite lo STC104. Consiste nel formare dei gruppi di link di uscita consecutivi tali che, fornendo alla testa di un pacchetto l'indirizzo del primo link del gruppo, si ottiene il suo instradamento sul primo canale libero del gruppo stesso. Supponiamo di voler connettere tra loro due STC104 come in figura 2.7:

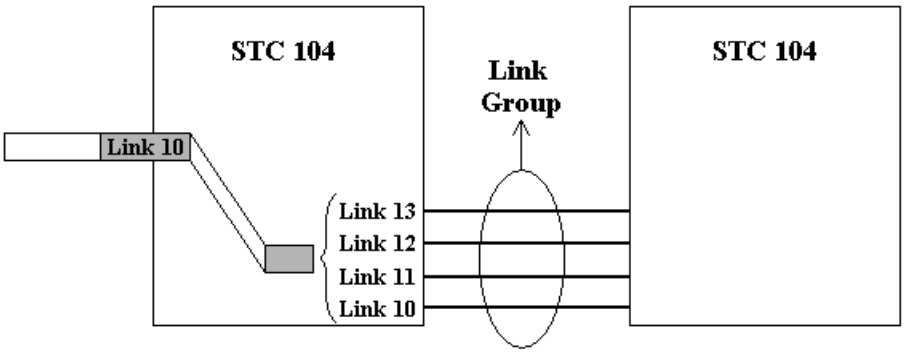

Figura 2.7: Esempio di 4 link appartenenti ad un gruppo. Per accedere al gruppo si deve fornire ai pacchetti la testa 10.

Invece di specificare i singoli percorsi tra i due dispositivi, si configura un gruppo formato dai link 10, 11, 12 e 13 (figura 2.7) e, ai vari pacchetti che debbono passare da uno switch all'altro, si fornisce l'indirizzo del link 10.

L’impostazione di tale funzione avviene tramite il bit **ContinueGroup** del registro **PacketMode** che ogni link possiede. Se questo bit viene posto a “0”, il link corrispondente è il primo di un gruppo, altrimenti è *uno qualsiasi* del gruppo. Se sono posti tutti a “0” il grouping è disabilitato. Una volta effettuate tutte le impostazioni, per utilizzare questo algoritmo di instradamento si deve agire sull’unico bit del registro **ConfigComplete**.

Uno dei difetti del *grouped adaptive routing* è che esso non è affatto *fault tolerant*: se c’è un errore su uno dei link compresi nel gruppo, tutto il gruppo diviene inutilizzabile.

2.3 L’interfaccia STC101

Per inviare e ricevere i pacchetti di comando di figura 2.4 nella rete di STC104, attraverso il link **Clink0**, viene utilizzato un personal computer provvisto di una scheda di interfacciamento IMS B108 [SGS94a] sul bus di I/O parallelo ISA che è fornita di due STC101. Come visto, il DSLink è un protocollo seriale: l’interfaccia tra il formato parallelo dei dati ed il DSLink è costituita appunto dallo STC101. Il suo diagramma a blocchi è mostrato nella figura 2.8:

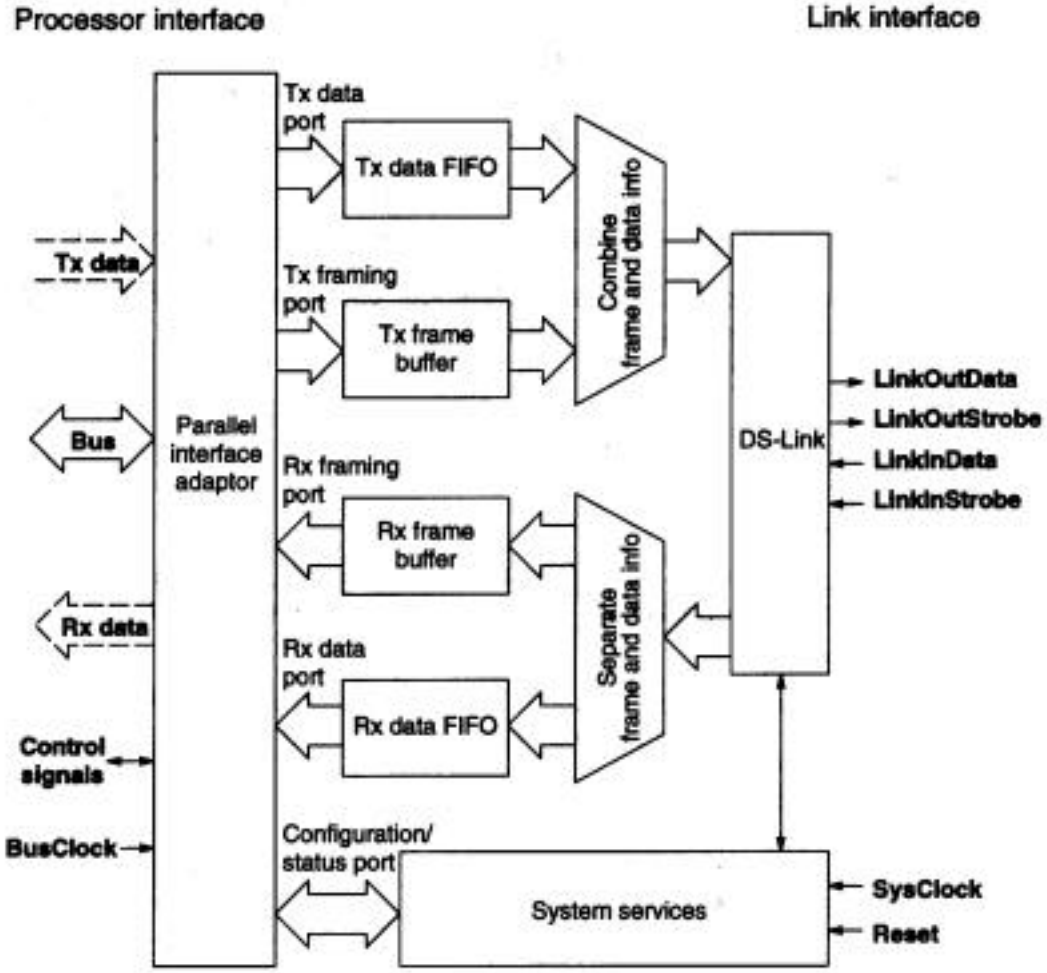

Figura 2.8: Diagramma a blocchi del C101.

Questo dispositivo VLSI, che gestisce in modo hardware il controllo di flusso fino al 3° livello (token) e converte i dati dal formato parallelo a seriale DSLink e viceversa, ha le seguenti caratteristiche fondamentali:

- può operare in modo seriale bidirezionale con una banda passante totale di 19 Mbyte/s;
- il bus parallelo è accessibile sia a 16 che a 32 bit;
- è in grado di raggruppare i dati in pacchetti di diverse lunghezze e di aggiungere e rimuovere la testa agli stessi: fornisce cioè gli strumenti per la gestione dei pacchetti e dei messaggi (4° e 5° livello del protocollo DSLink);
- in accordo con il protocollo IEEE 1355 contiene delle memorie FIFO (First In First Out) al suo interno (64 byte in ingresso e 64 byte in uscita);
- ha una interfaccia parallela opzionale denominata **Token Interface** con la quale si possono gestire in multiplexing le porte di ingresso e di uscita, con conseguente aumento delle prestazioni.

La porta denominata *Tx data* fornisce i dati da spedire alla FIFO di trasmissione, mentre la porta *Tx framing* invia le informazioni di contorno (header e terminator). Operazioni analoghe svolgono le porte *Rx data* ed *Rx framing*. Le informazioni per il routing ed i dati vengono combinate e separate da una logica dedicata come mostrato in figura 2.8. La porta *configuration/status* permette di leggere e scrivere i registri di configurazione. Il *parallel interface adaptor* effettua invece il multiplexing delle porte logiche (quelle interne al dispositivo) nelle porte fisiche.

Come per lo STC104, la programmazione dello STC101 avviene tramite dei registri interni i quali possono essere di sola lettura, sola scrittura o di lettura/scrittura. Molti dei registri sono simili a quelli presenti nel C104,

pertanto non vengono descritti in questo paragrafo, rimandando il lettore alla bibliografia per ulteriori specifiche [SGS97].

Capitolo 3

Comunicazioni tra i processi remoti e l'agente di controllo

Per ridurre la complessità, il software di comunicazione delle reti di computer è organizzato in livelli o strati, ciascuno costruito sopra il suo predecessore. Il numero degli strati ed il nome, il contenuto e la funzione di ciascuno di essi differisce da una rete all'altra. Comunque, in tutte le reti lo scopo di ciascuno strato è quello di offrire servizi agli strati superiori.

Lo strato *n* su una macchina dialoga con lo strato *n* di un'altra macchina. Le regole di tale conversazione, che definiscono il formato dei dati ed il modo con cui due processi si scambiano le informazioni, sono note come **protocollo** dello strato *n*. Ovviamente, nessun dato viene trasferito direttamente dallo strato *n* di una macchina allo strato *n* di un'altra macchina, a meno che non si parli del cavo fisico su cui viaggiano i segnali; tra ciascuna coppia di strati adiacenti, esiste un'interfaccia che definisce le operazioni primitive ed i servizi offerti dallo strato inferiore a quello superiore.

In questo capitolo verrà descritto il modello generale, noto come modello *client-server*, con il quale è stato gestito il dialogo tra un processo (*client*),

residente su una macchina, che richiede un determinato tipo di servizio ad un altro processo (*server*) residente su un'altra macchina. Come vedremo dettagliatamente in seguito, quest'ultimo permetterà al *client* di poter configurare e monitorare in remoto una *switching network* di STC104 attraverso l'invio di opportuni comandi. Lo scambio di informazioni tra i due processi si basa sul servizio offerto dall'insieme di protocolli *Transmission Control Protocol/Internet Protocol* (TCP/IP) che rappresenta di fatto lo standard più diffuso per lo sviluppo di applicazioni in rete.

3.1 Il modello OSI

Nella figura 3.1, tratta da [Tan95], si illustra il modello di riferimento OSI, basato su di una proposta sviluppata dalla organizzazione internazionale degli standard (*International Standards Organization*, ISO) come primo passo verso la standardizzazione internazionale dei vari protocolli:

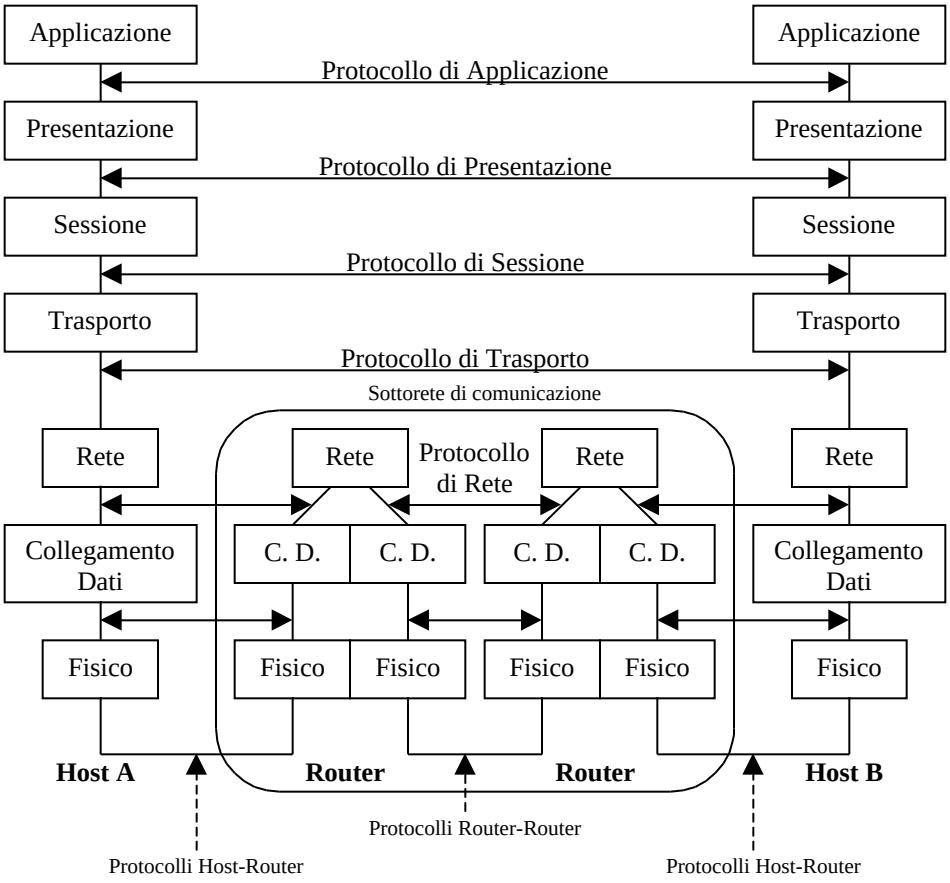

Fig. 3.1: Il modello OSI a sette strati

Tale modello non definisce esattamente un'architettura di rete, intesa come l'insieme dei servizi e dei protocolli da utilizzare in ciascun strato, ma ciò che ciascuno strato dovrebbe fare. Si consideri, inoltre, che tale modello di standardizzazione fu proposto quando già esistevano numerose reti che impiegavano differenti algoritmi ai vari livelli.

Uno strato può offrire allo strato immediatamente superiore due diversi tipi di servizio:

- orientato alla connessione;
- senza connessione.

Il **servizio orientato alla connessione** è modellato come il sistema telefonico. L' utente del servizio stabilisce dapprima una connessione, la utilizza ed infine la termina. L' aspetto essenziale di una connessione è che essa garantisce al ricevitore di estrarre le informazioni nel medesimo ordine con cui sono state inviate dal trasmettitore.

Per contro, il **servizio senza connessione** è modellato come il sistema postale. Ciascun messaggio trasporta l'indirizzo di destinazione completo e viene instradato attraverso la rete indipendentemente dagli altri messaggi; può quindi accadere, contrariamente al servizio orientato alla connessione, che due messaggi diretti verso la medesima destinazione arrivino in ordine inverso.

Come si nota dalla figura 3.1, il protocollo dello strato di trasporto è il primo vero protocollo, a partire dallo strato più basso, definito tra le macchine finali di provenienza e destinazione, o **end-to-end**, mentre negli strati inferiori i protocolli sono tra ciascuna macchina ed i punti di accesso alla sottorete di comunicazione, e quindi non si svolgono tra le macchine finali di provenienza e destinazione vere e proprie. Lo scopo dello strato di rete è infatti quello di instradare i pacchetti dall'estremità di provenienza a

quella di destinazione (*routing*) e di controllare il congestionamento della rete, fornendo così allo strato di trasporto un canale di comunicazione virtuale *end-to-end* tra l'host di provenienza e quello di destinazione.

Come già descritto nell'introduzione al presente capitolo, il flusso effettivo dei dati non è "orizzontale" tra uno strato n di una macchina e lo strato corrispondente di un'altra macchina. Nella figura 3.2 che segue [BG92] si illustra la tecnica con la quale i pacchetti di dati vengono trasmessi, omettendo i tre strati superiori di figura 3.1[1]:

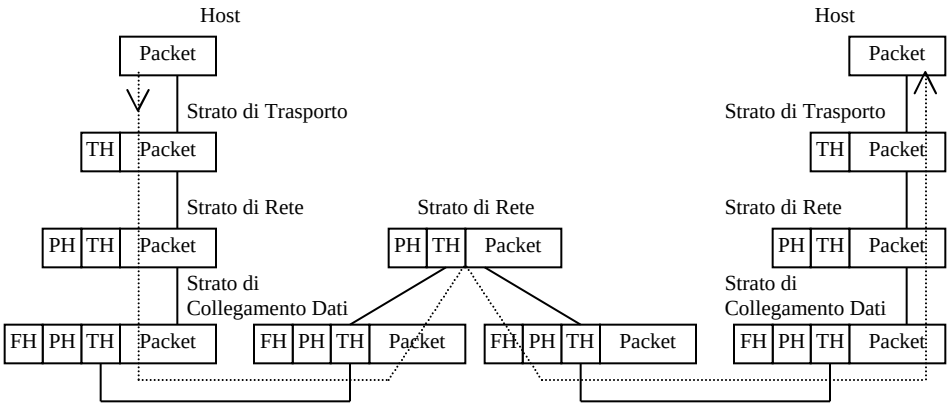

Fig. 3.2: Effettivo percorso di trasmissione dei dati

L'host di provenienza ha un pacchetto di dati che desidera inviare all'host di destinazione; lo strato di trasporto combina questi dati con una sua intestazione (TH) e passa l'intero blocco allo strato di rete. Tale processo si ripete sino al raggiungimento del canale di comunicazione fisico. Alla

[1] Come vedremo in seguito, il software sviluppato per il progetto accede direttamente allo strato di trasporto.

destinazione, le varie intestazioni vengono eliminate man mano che l'intero messaggio attraversa i vari strati.

3.2 I protocolli TCP/IP

Il termine TCP/IP si riferisce ad un insieme di protocolli che, in termini di paragone con il modello OSI visto nel paragrafo precedente, si riferiscono rispettivamente allo strato di trasporto e a quello di rete.

L'idea originale alla base del TCP/IP era di definire un insieme di procedure standard che permettessero a reti individuali di essere interconnesse con la rete *Arpanet*, una rete sviluppata dal Ministero della Difesa statunitense, che non è basata sul modello OSI (l'ha preceduto di circa un decennio); questo processo di interconnessione tra reti si è sviluppato fino a costituire l'attuale rete mondiale *Internet*.

I livelli TCP/IP hanno questa relazione con quelli OSI [Mar94]:

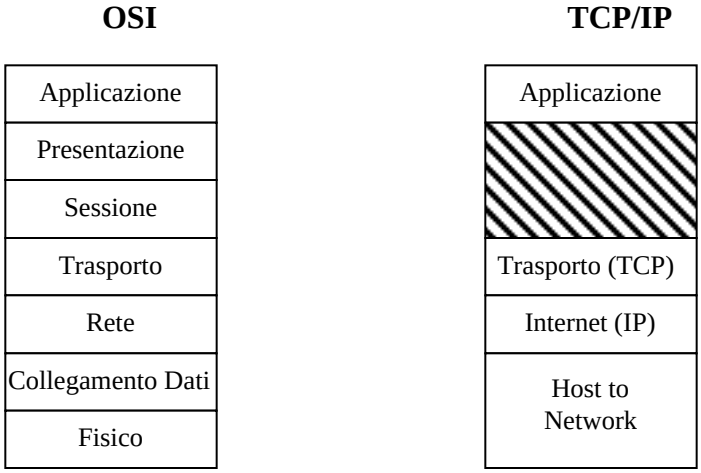

Fig. 3.3: Relazione fra i livelli OSI e TCP/IP

Come si può notare dalla figura 3.3, non ci sono i livelli sessione e presentazione definiti nel modello OSI (non furono ritenuti necessari; l'esperienza ha mostrato che questa visione è condivisibile). Sopra il livello di trasporto c'è direttamente il livello applicazione, che contiene tutti i protocolli di alto livello che vengono usati dalle applicazioni reali. I primi protocolli furono:

- ***Telnet***: terminale virtuale;
- ***FTP*** (***F****ile* ***T****ransfer* ***P****rotocol*): trasferimento di file,

ai quali sono stati successivamente aggiunti altri protocolli, tra i quali:

- ***DNS*** (***D****omain* ***N****ame* ***S****ervice*): mapping fra nomi di host e indirizzi IP numerici;
- ***HTTP*** (***H****yper* ***T****ext* ***T****ransfer* ***P****rotocol*): la base del *World Wide Web*.

Nella *suite* di protocolli TCP/IP, l'intero indirizzo di una sorgente (o di una destinazione) è organizzato gerarchicamente con la sequenza: identificatore di sottorete, identificatore dell'host nella sottorete e identificatore di processo. Quest'ultimo, chiamato anche *porta*, appartiene all'intestazione del pacchetto TCP, mentre i primi due identificatori appartengono all'intestazione dell' IP, come si può notare nelle figure 3.4 e 3.5, tratte da [Tan95]:

Fig. 3.4: Formato dell'intestazione dei dati nel protocollo IP

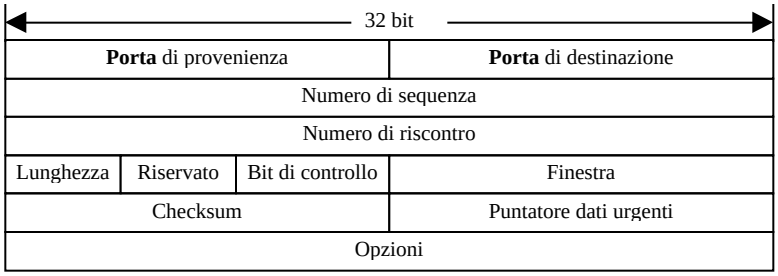

Fig. 3.5: Formato dell'intestazione dei dati nel protocollo TCP

Con questa struttura, tutte le sessioni stabilite da un nodo sorgente ad un nodo destinazione hanno lo stesso indirizzo nell' intestazione IP, ma saranno riconosciute singolarmente al livello TCP attraverso la *porta*.

Lo strato di trasporto TCP offre un servizio affidabile ed orientato alla connessione; accetta messaggi arbitrariamente lunghi dai processi utente, li suddivide in segmenti di lunghezza non superiore a 64Kbyte, e invia ciascun segmento come un distinto datagramma[2] tramite lo strato IP. Lo strato di rete IP, per contro, offre un servizio senza connessione, per cui non vi è alcuna garanzia che i datagrammi saranno consegnati correttamente e

[2] Con datagramma si identifica la coppia (*intestazione*,*dati*) del protocollo IP.

nell'ordine con cui sono stati trasmessi; è compito del TCP riassemblare i messaggi nella sequenza corretta e chiedere la ritrasmissione di quelli affetti da errore. A tale scopo, l'algoritmo utilizzato per garantire all'host di destinazione di ricevere i messaggi esenti da errore e con lo stesso ordine con il quale sono stati trasmessi è del tipo "a finestra scorrevole con ripetizione selettiva". Ciò significa che ciascun byte dei dati trasmessi ha un suo numero di sequenza (figura 3.5), in modo che l'host trasmittente, in risposta ai dati inviati, riceverà dalla destinazione un riscontro (*Ack*), anch'esso numerato, sull'avvenuta ricezione. Il meccanismo di riscontro implementato fa sì che un Ack con numero di sequenza *n* indichi all' host trasmittente che tutti i byte trasmessi con numero di sequenza $m \leq n-1$ sono stati correttamente ricevuti. Il trasmettitore ed il ricevitore gestiscono quindi entrambi una finestra di numeri di sequenza accettabili che scorre quando i dati vengono trasmessi e correttamente riscontrati.

Tale algoritmo è ampiamente illustrato in [Pos81] e [BG92] per una analisi della efficienza e della correttezza.

3.2.1 Attivazione di una connessione

Nella figura 3.6, tratta da [Pos81], si illustra la procedura tramite la quale due processi, indicati genericamente con le sigle **A** e **B**, stabiliscono una connessione. I segnali etichettati con CTRL sono dei bit di controllo presenti nell'intestazione dei segmenti del protocollo TCP (figura 3.5), mentre quelli etichettati con SEQ sono i numeri di sequenza tramite i quali i due processi numerano i byte trasmessi :

	TCP **A**		TCP **B**
1.	CLOSED		LISTEN
2.	TX SYN	→ <SEQ=n><CTRL=SYN> →	RX SYN
3.	CONNESSO	← <SEQ=m><ACK=n+1><CTRL=SYN,ACK> ←	RX SYN
4.	CONNESSO	→ <SEQ=n+1><ACK=m+1><CTRL=ACK> →	CONNESSO

Fig. 3.6: Procedura per l'attivazione di una connessione

Il processo A inizia trasmettendo un segmento SYN e comunicando che userà i numeri di sequenza a partire da *n* (riga 2). Il processo B a sua volta trasmette un segmento SYN riscontrando, con il bit di controllo ACK, il SYN del processo A (riga 3). Come si nota, il processo B comunica anche che i segmenti trasmessi inizieranno con il numero di sequenza *m* e, inoltre, che si attende dal processo A un nuovo segmento con numero di sequenza *n+1*. Come ultimo passo della procedura, il processo A riscontra il segmento SYN ricevuto (riga 4). Al termine della procedura, entrambi i processi avranno stabilito la connessione.

Questa procedura è anche nota con il nome *handshake a tre vie* in quanto l'ACK del processo B alla richiesta di inizio connessione proveniente dal processo A viene trasmesso insieme alla propria richiesta di connessione (riga 3 di figura 3.6), generando quindi uno scambio di tre messaggi.

3.3 Il modello Client-Server

Il modello tipico per sviluppare applicazioni in rete è quello noto come *Client-Server*. Un *Server* è un processo in attesa di essere "contattato" da un altro processo, il *Client*, in modo da poter soddisfare una richiesta.

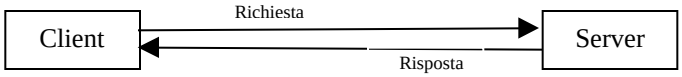

Fig. 3.7: Il client emette una richiesta a cui il server spedisce una risposta

La comunicazione assume sempre la forma di coppie richiesta-risposta, sempre avviate dai client, mai dai server. Uno scenario tipico può essere il seguente:

- il processo *server* è attivato su una qualsiasi macchina. Si inizializza, quindi attende di essere contattato da un processo *client*;
- un processo *client* è attivato, sulla stessa macchina del *server* oppure su un' altra macchina connessa attraverso una rete, e trasmette una richiesta al processo *server* richiedendo un servizio di un particolare tipo. Alcuni esempi tipici del servizio che un server può offrire sono:
  - stampare un file su una stampante per il *client*;
  - leggere o scrivere un file sulla macchina del *server* per il *client*;
  - trasmettere il tempo di sistema al *client*;
  - eseguire un comando per il *client* sulla macchina del *server*.
- quando il processo *server* ha concluso il suo servizio per il *client*, ritorna di nuovo ad attendere il prossimo processo *client* per soddisfarne le richieste.

I processi *server* possono essere suddivisi in due categorie [Ste90]:

Server iterativi

la richiesta proveniente dal processo *client* può essere soddisfatta dal *server* in un intervallo di tempo relativamente breve e noto a priori; il processo *server* stesso espleta il servizio. Tipicamente, la trasmissione del tempo di sistema è gestita in questo modo.

Server concorrenti

il tempo necessario a soddisfare una richiesta dipende dal tipo della richiesta stessa; il processo *server* gestisce le attività in modo *concorrente*, invocando un altro processo per soddisfare ogni richiesta proveniente dal *client*, così che il *server* originale ritorna in attesa di successive richieste da parte di altri *client*. Come sarà mostrato in seguito, l'agente di controllo sviluppato per il progetto è di questo tipo.

3.4 Descrizione del sistema

Lo scopo del progetto è quello di poter configurare, monitorare e controllare in modo remoto una *switching network* comunque complessa basata sullo STC104.

Ispirandosi al modello *client-server* appena descritto, possiamo suddividere il sistema nelle seguenti componenti:

- un **agente di controllo** (il processo *server*) della rete di STC104, in grado di accettare le richieste di connessione provenienti dai *client* e di eseguire localmente, per loro conto, le funzioni di controllo della rete stessa;

· un **processo remoto** (il processo *client*) tramite il quale un utente può inoltrare all'agente di controllo, via TCP/IP, le richieste di esecuzione delle suddette funzioni ed attendere i relativi dati di risposta.

Nella figura 3.8 che segue si illustra il sistema in esame, considerando la potenziale situazione in cui diversi utenti siano connessi con l'agente di controllo:

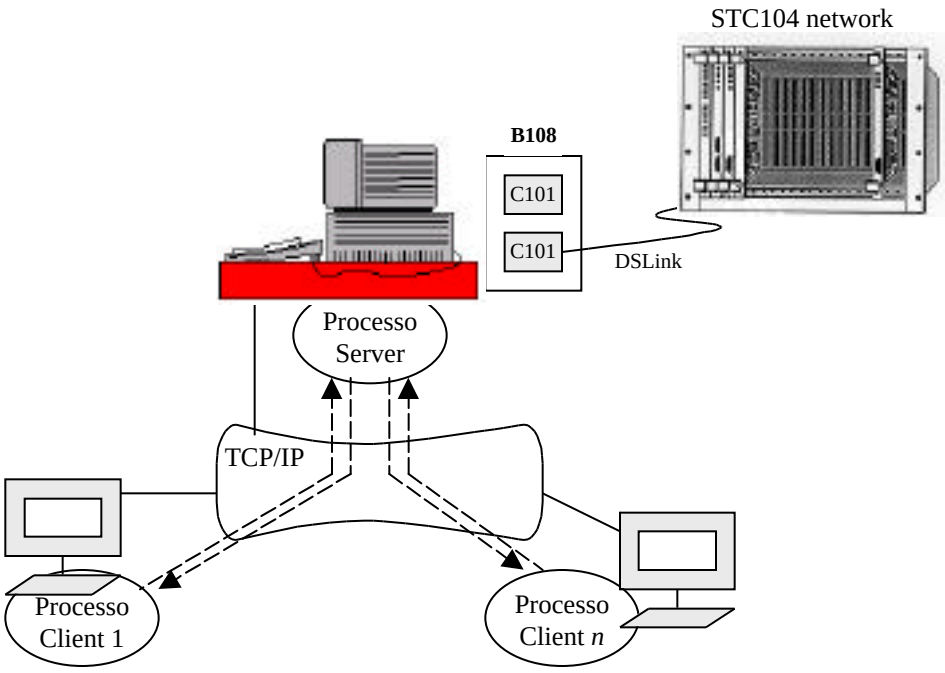

Fig. 3.8: Ipotesi di sviluppo: più processi client, su diversi host, richiedono al processo server, tramite una connessione TCP/IP, l'esecuzione di specifici comandi per il controllo ed il monitoraggio della rete di STC104.

Il cavo DSLink in uscita dall'interfaccia C101 (figura 3.8) rappresenta la connessione fisica tra l'agente di controllo e la rete di STC104; come descritto nel capitolo 3, l'invio dei comandi di inizializzazione e di controllo dei registri del singolo STC104 avviene tramite il link di controllo **Clink0**, mentre tramite il **Clink1** è possibile instradare i pacchetti di comando ai vari *switch* di cui è composta la rete:

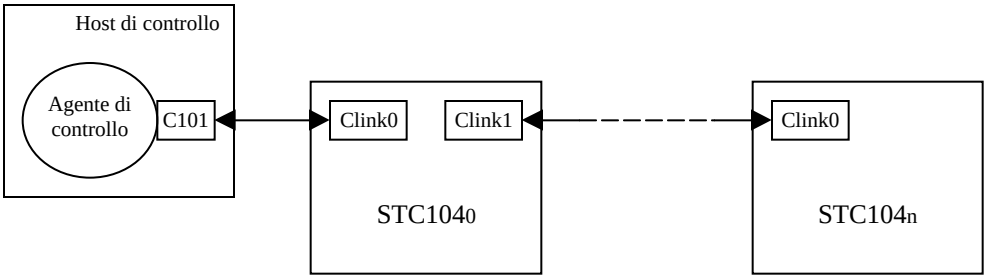

Fig. 3.9: Daisy-chaining dei link di controllo di una switching network composta da n+1 STC104

Dal punto di vista dell'agente di controllo, non è noto a priori quanto tempo verrà speso da ciascun utente (*client*) per espletare il proprio servizio; si può ragionevolmente supporre che un utente rimanga connesso per diverse ore al fine di controllare il corretto funzionamento della *switching network* durante la fase di trigger dei dati. Per questo motivo, facendo riferimento alle due categorie descritte nel precedente paragrafo, il server sviluppato è di tipo concorrente.

3.4.1 Il protocollo al livello di applicazione

Si vuole ora analizzare come è stato gestito il dialogo, a livello di applicazione della architettura TCP/IP, tra un processo remoto e l'agente di controllo, descrivendo cioè le funzioni con le quali quest'ultimo accede alla rete di STC104, il formato dei messaggi di richiesta tramite i quali un processo *client* invoca l'esecuzione di tali funzioni ed il formato dei messaggi di risposta trasmessi dal *server* al *client*.

La struttura generale di un messaggio, sia di richiesta che di risposta, è illustrata nella figura 3.10 che segue:

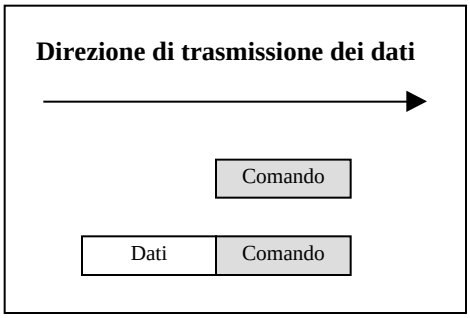

Fig. 3.10: Formato dei messaggi

Con questa struttura, l'agente di controllo può identificare la funzione richiesta testando il valore del campo Comando e, se la particolare funzione lo richiede, estrarre anche i Dati; il *client*, con la stessa tecnica, può correttamente processare i Dati di risposta relativi ad un Comando precedentemente trasmesso.

Nella figura 3.11 che segue si riportano i formati dei messaggi di richiesta che possono essere trasmessi al *server*:

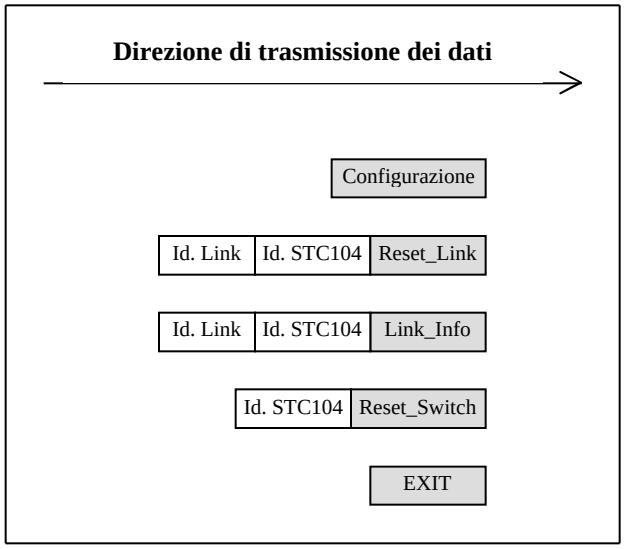

Fig. 3.11: Formato dei messaggi di richiesta che un processo client trasmette al server

Il significato di ciascuno di essi è facilmente intuibile:

- con il comando Configurazione si richiede all'agente di controllo di configurare appunto ciascuno STC104 di cui è composta la *switching-network*;
- il comando Reset_Link provoca l'esecuzione dell'operazione di *Reset* di uno dei 32 Data Link (Id. Link in figura 3.11) di uno specifico *switch* (Id. STC104);
- con Link_Info, la cui struttura è analoga al precedente messaggio, il *client* può richiedere la lettura dei bit dei registri di configurazione di

uno dei 32 Data Link (Id. Link in figura 3.11) di uno specifico *switch* (Id. STC104);

- il comando Reset_Switch invoca invece l'esecuzione dell'operazione di *Reset* su tutti i Data Link di un determinato *switch* (Id. STC104 in figura 3.11). Come sarà descritto in seguito, questo tipo di *Reset* permette, rispetto al comando di Reset_Link, un migliore ripristino della operatività degli STC104 a seguito di condizioni di errore sui Data Link interessati da un flusso di pacchetti;
- con EXIT il *client* richiede la disconnessione.

I messaggi che l'agente di controllo trasmetterà al *client*, in risposta al comando ricevuto, sono illustrati in figura 3.12:

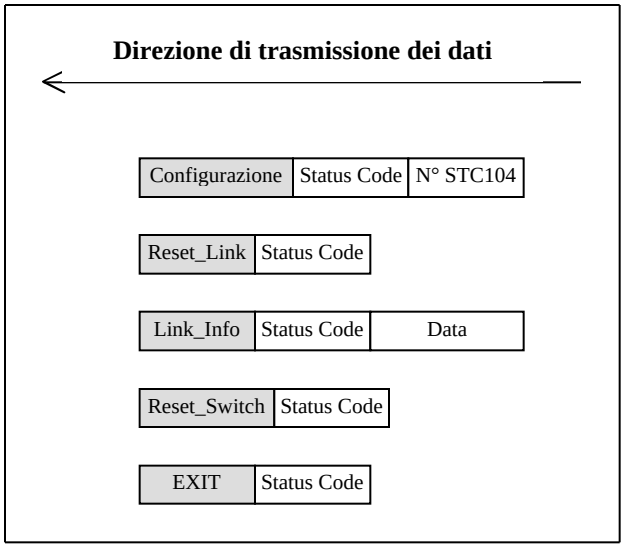

Fig. 3.12: Formato dei messaggi di risposta inviati dal server per ciascun comando ricevuto dal client

Per ogni richiesta ricevuta dal *client*, il *server*, dopo aver eseguito il relativo comando, trasmette in risposta la codifica del comando stesso seguita da uno Status Code che ne rivela il successo (Status Code = 9) od il fallimento (Status Code = 1). In caso di successo, in risposta ai comandi di Configurazione e Link_Info viene trasmesso rispettivamente il numero di STC104 di cui è composta la rete (N° STC104 in figura 3.12) ed il valore dei singoli bit (Data in figura 3.12) dei registri di configurazione del Data Link richiesto.

L'esecuzione di ciascun comando sulla macchina del *server* sarà descritta nel capitolo 6, quello dedicato alla descrizione del software sviluppato.

Di seguito si riporta la codifica, utilizzata nei programmi *client* e *server* sviluppati, del campo Comando presente in ciascun messaggio di richiesta e di risposta :

Configurazione :: = 1;

Reset_Link :: = 3;

Link_Info :: = 5;

Reset_Switch :: = 6;

EXIT ::= 9;

Come vedremo, ciascuno di essi verrà rappresentato con un *byte* (8-bit).

3.4.2 Monitoraggio della rete di STC104

Come già accennato nel paragrafo 3.2, i registri **Link0-31Status** sono stati scelti al fine di monitorare continuamente, in stile *polling*, lo stato di ciascun Data Link, e trasmettere conseguentemente al *client* i valori letti. Intuitivamente, si potrebbero scegliere due diverse implementazioni:

- secondo il modello *client-server* vero e proprio, e quindi interrogare ciclicamente, da remoto, lo stato della rete di STC104 attraverso una richiesta di lettura dei **Link0-31Status** e attesa dei relativi dati in risposta dal *server*;
- demandare al *server* il compito di trasmettere tali dati qualora lo stato dei Data Link fosse cambiato durante la fase di *polling*.

Nel primo caso, il monitoraggio degli STC104 avrebbe luogo con modalità analoghe a quelle viste per i messaggi di richiesta inviabili al *server*. Senza dubbio tale soluzione è corretta, il codice per sviluppare il *server* sarebbe più semplice, ma richiederebbe, per ogni *client* connesso, un flusso pressoché continuo di richieste e risposte tra le macchine di provenienza e destinazione, a prescindere se i valori dei registri **Link0-31Status** siano variati o meno rispetto al precedente ciclo di interrogazione.

Nel secondo caso, che rappresenta poi la soluzione adottata, il processo *client* gestisce in modo *asincrono* le informazioni provenienti dal *server* relative allo stato dei Data Link; in questo modo, il *server* decide autonomamente se, ad ogni ciclo di *polling* effettuato localmente,

comunicare o meno tali dati al *client*, basandosi sui valori letti e confrontandoli con quelli memorizzati al ciclo precedente.

Questo tipo di soluzione, oltre a complicare leggermente il codice del processo *server*, ha delle ripercussioni sul protocollo di comunicazione precedentemente definito. Si immagini ad esempio la seguente situazione, nella quale un *client* emette una richiesta di esecuzione di un generico comando, ponendosi quindi in attesa dei relativi dati di risposta dal *server*:

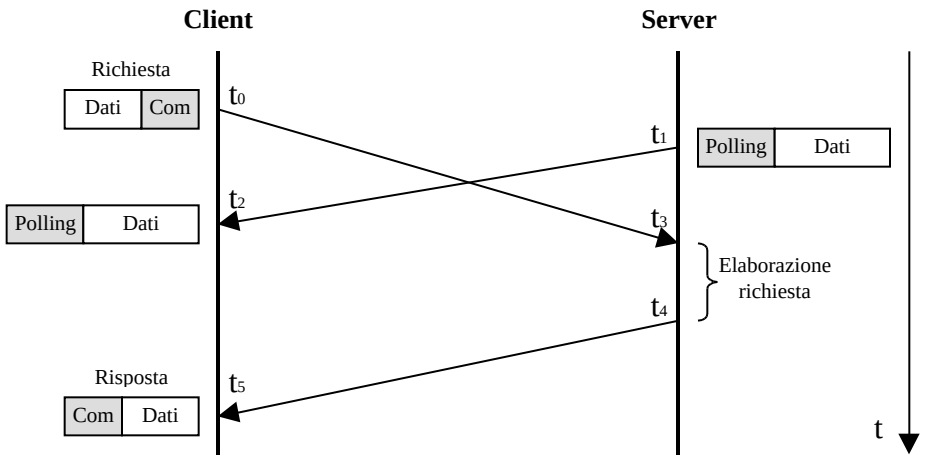

Fig. 3.13: Il client, nell'istante di tempo t_0*, emette un messaggio di richiesta la cui risposta giunge in* t_5*. Nell'istante* $t_0 \leq t_2 \leq t_5$*, il client riceve, asincronamente, un messaggio di polling trasmesso dal server in* t_1.

Sostanzialmente bisogna prevedere nel processo *client*, in fase di lettura dei dati, la possibilità di ricevere informazioni relative al cambiamento di stato dei Data Link oppure l'effettivo messaggio di risposta al comando richiesto.

Grazie al supporto per il *multithreading* offerto dall'ambiente di programmazione Java, con il quale è stato sviluppato il processo *client*, si è potuto realizzare, nell'ambito del *client* stesso, un processo in grado di trasmettere le richieste al server e riceverne le risposte, ed un secondo processo in grado di gestire il monitoraggio degli STC104.

Possiamo a questo punto già anticipare i problemi legati alla sincronizzazione dei processi operanti in regime di concorrenza, in quanto:

- nel *client*, i processi di monitoraggio e richieste di esecuzione di comandi condividono lo stesso canale di comunicazione con l'agente di controllo della *switching-network*;
- nel *server*,che si ricorda è di tipo **concorrente**, i processi ai quali si demanda il compito di soddisfare le richieste provenienti dal *client* condividono la medesima rete di STC104. Inoltre, il processo di *polling* dei **Link0-31Status** condivide con essi il canale di comunicazione con i *client*.

3.5 L'interfaccia di servizio del protocollo TCP

In questo paragrafo intendo illustrare gli strumenti software con i quali è stato possibile interfacciare l'applicazione *client-server* sviluppata per la tesi con il protocollo TCP. L'interfaccia di programmazione delle applicazioni (API) più comunenemente usata è stata sviluppata dalla *Berkeley Software Distribution* (BSD), ed è disponibile in molti sistemi operativi Unix e non.

D' importanza fondamentale per l'interfaccia di servizio è il concetto di *socket*. I socket sono punti terminali a cui le connessioni possono essere

collegate “dal fondo”, ossia dal lato del sistema operativo, ed a cui i processi possono essere collegati “dalla cima”, quindi dal lato dell’utente.

In base a quanto analizzato nel precedente paragrafo 3.2.1, la trasmissione dei dati tra due processi è possibile solo dopo aver attivato una connessione tra essi. Inoltre il processo client deve essere a conoscenza dell’indirizzo IP dell’host sul quale il *server* è in esecuzione, nonché della specifica porta che quest’ultimo utilizza nell’intestazione del protocollo TCP.

Possiamo rappresentare la connessione tra due processi attraverso la seguente 5-upla:

{protocollo, indirizzo locale, porta locale, indirizzo remoto, porta remota}

Con la chiamata di sistema *socket( )*, il *server* crea una struttura dati all’interno del sistema operativo nella quale, con la chiamata *bind( )*, viene memorizzato il protocollo, il suo indirizzo e la sua porta locali; il *client*, con la stessa chiamata di sistema *socket( )*, specifica il protocollo, l’indirizzo e la porta remoti del *server*. In entrambi i processi, la 5-upla vista rimane pertanto incompleta fino a quando non viene di fatto stabilita la connessione.

Nella figura 3.14 che segue viene illustrata la sequenza completa di operazioni che tipicamente vengono eseguite per un trasferimento di dati, tra un processo *client* ed un *server*, basato su un servizio orientato alla connessione come quello offerto dal protocollo TCP:

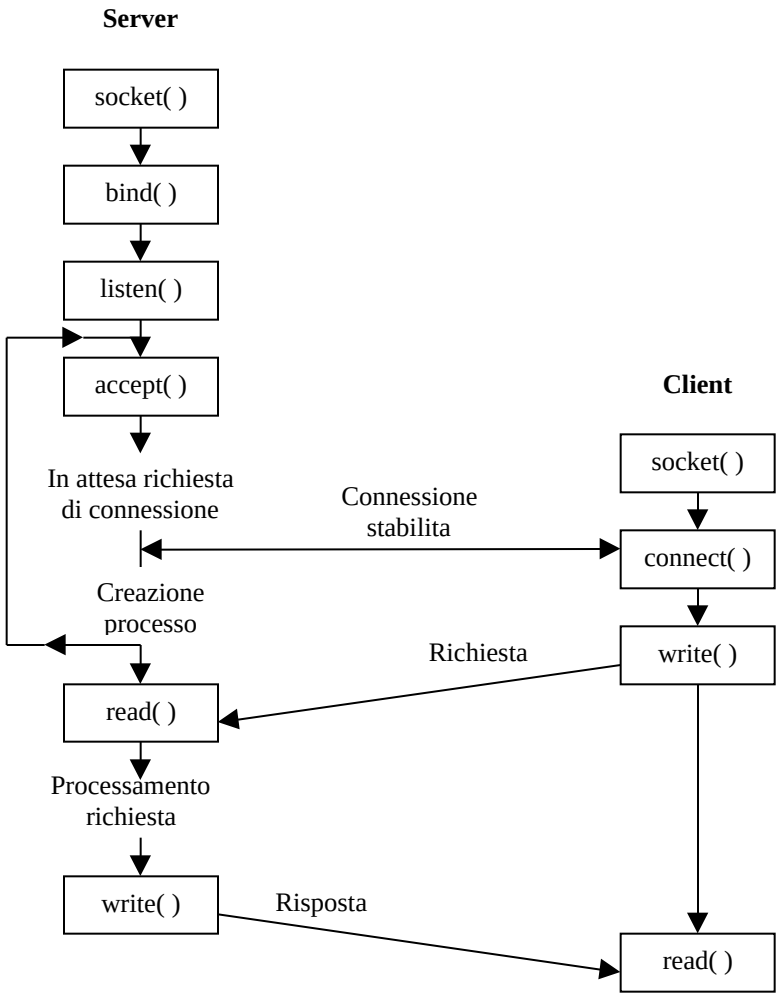

Fig. 3.14: Sequenza delle chiamate di sistema effettuate da un processo client ed un server per un protocollo orientato alla connessione

Il *server*, con la chiamata *accept( )*, si pone in uno stato di attesa di richiesta di connessione da parte di un *client*; quando quest'ultimo invoca la chiamata *connect( )*, ha inizio la procedura di attivazione della connessione descritta

al paragrafo 3.2.1 . Durante questa fase, il nucleo del sistema operativo sul quale viene eseguito il *client* trasmette, assieme ai dati del *server* specificati dall'utente, l'*indirizzo locale* ed una *porta locale* tra quelle disponibili nel sistema. Tali dati, quando ricevuti dal *server*, completeranno la 5-upla in entrambi i processi. L'aspetto interessante da sottolineare, in merito alla chiamata *accept( )*, è che il valore da essa ritornato, a meno di condizioni di errore, si riferisce alla nuova 5-upla appena completata, e risulta essere una struttura diversa dal *socket* inizialmente creato. In questo modo, è possibile implentare un *server* concorrente, demandando ad un altro processo il compito di gestire le comunicazioni con il *client*, e ritornando ad attendere nuove richieste di connessione da parte di altri *client* sul *socket* iniziale.

Nel capitolo successivo verrà data una descrizione dell'ambiente di programmazione Java, con il quale è stato realizzato il pannello di visualizzazione grafica, monitoraggio e controllo da remoto della rete di STC104, ossia il processo *client* di figura 3.14.

Capitolo 4

L’ambiente di programmazione JAVA

Java fu introdotto dalla Sun Microsystems nel 1995 e immediatamente diede un nuovo senso alle possibilità degli utenti di interagire con il *Web* (detto anche *WWW*), il sistema grazie al quale si può accedere ad un enorme insieme di documenti (detti *pagine Web*) sparsi su milioni di elaboratori. Fu espressamente progettato per l’uso in ambienti distribuiti, come la rete mondiale *Internet* o una qualsiasi rete di calcolatori, e sostanzialmente fornisce una soluzione al problema di eseguire applicazioni trasportando programmi con un formato indipendente dalla piattaforma.

Poichè quest’ultimo aspetto è in relazione al codice compilato di un generico programma Java, e considerata la molteplicità delle architetture hardware delle piattaforme connesse ad una rete di calcolatori, tale codice viene interpretato ed eseguito dalla cosiddetta macchina virtuale Java (*Java Virtual Machine*, **JVM**) sulla specifica piattaforma.

Questo capitolo descrive gli aspetti ritenuti essenziali del linguaggio di programmazione Java, della macchina virtuale Java e delle librerie utilizzate

per lo sviluppo del pannello grafico di visualizzazione e controllo remoto della rete di STC104.

4.1 Descrizione generale

Una pagina Web contiene un riferimento ad un programma scritto in Java, detto ***applet***; quando un utente, tramite il *Web-browser*[1], accede a tale pagina, assieme ad essa viene recuperato anche il codice eseguibile (detto ***bytecode***) dello *applet*. Dopo essere stato caricato, esso viene mandato in esecuzione "all'interno" del *browser*, quindi sulla piattaforma dell'utente. In questo modo, disponendo degli strumenti forniti dall'ambiente di programmazione Java (suono, grafica, gestione dei flussi di input/output ed altri) viene offerta quella alta interattività che prima mancava al Web.

Le finalità alle quali un *applet* tende possono essere di varia natura. Un esempio può essere quello citato nel documento [Sar97], un progetto in corso di sviluppo presso l'Università di Cambridge teso a sfruttare la capacità di calcolo delle piattaforme che si connettono ad un *server* centrale tramite un *browser* che "ospita" la JVM, al fine di distribuire i processi di calcolo di un'applicazione parallela sotto forma di *applet*; inoltre, alcune grandi organizzazioni commerciali, proprietarie di reti private, stanno riprogettando le interfacce utente ai dati sfruttando le nuove potenzialità offerte dagli *applet* all'interno di tali *browser*.

[1] I *browser* (sfogliatore) sono le applicazioni che permettono ad un utente di reperire e visualizzare le informazioni contenute in una pagina Web; esempi tipici ne sono *Netscape Navigator* ed il *Microsoft Internet Explorer*.

Nel caso dell'esperimento ATLAS, questa tecnica apre tutta una serie di possibilità di interazione e controllo remoto dell'esperimento per mezzo di un'interfaccia grafica (lo *applet*), prescindendo così dal sistema operativo dello host *client* ed, inoltre, senza dover installare tale software su determinate piattaforme, dal momento che si può ragionevolmente supporre che laddove ci sia un calcolatore connesso in rete sia presente anche un *Web browser*.

Nel presente lavoro si è sviluppato un tale sistema per la riconfigurazione di uno switching network come quello descritta nel capitolo 1.

4.2 La piattaforma Java

Come detto nell'introduzione al presente capitolo, un generico programma scritto con il linguaggio Java viene tradotto staticamente, mediante compilazione, in un linguaggio intermedio e, successivamente, interpretato dalla implementazione della JVM sulla specifica piattaforma.

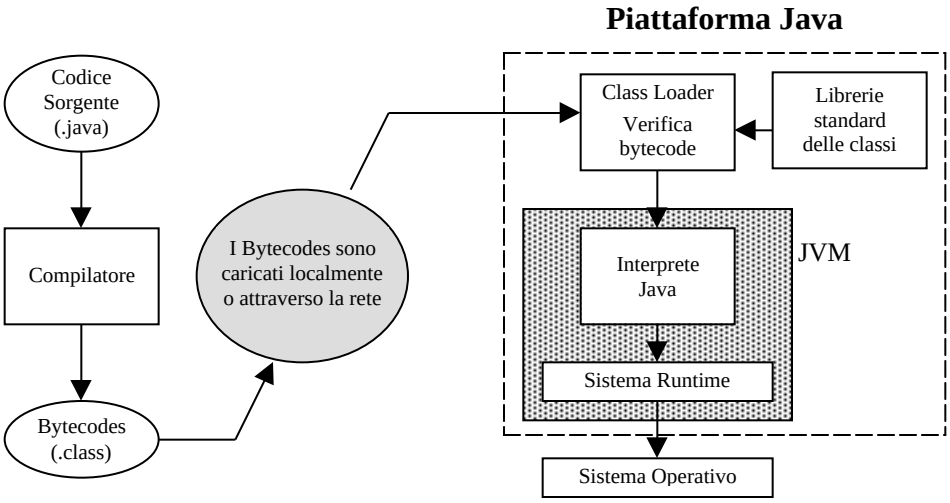

Fig. 4.1: Compilazione e interpretazione di un sorgente java

Il compilatore traduce i file sorgenti scritti in Java (aventi l'estensione .java) in codice eseguibile, contenuto in uno o più file caratterizzati dall'estensione .class. Sono questi ultimi file che vengono eseguiti localmente dalla piattaforma Java oppure, come illustrato in figura 4.1, possono viaggiare attraverso la rete ed essere caricati dai Web *browser* per essere eseguiti.

La JVM è il cuore di tutto il meccanismo. Essa fornisce un dispositivo di calcolo astratto, capace di eseguire il codice contenuto nei file .class e dotato di:

- un set di istruzioni macchina;
- un insieme di registri, un program counter, ecc.

Si noti che attualmente i file .class si ottengono a partire da codice Java, ma ciò non fa parte delle specifiche della macchina virtuale. Non è escluso che in futuro si possano utilizzare altri linguaggi sorgente corredati di opportuni

compilatori. Inoltre, anziché interpretare i *bytecodes*, come illustrato in figura 4.1, è possibile compilarli ulteriormente, a tempo d'esecuzione, in codice macchina della specifica piattaforma mediante l'utilizzo di compilatori detti Just-In-Time (JIT), ottenendo un miglioramento nelle prestazioni di un fattore 10 rispetto al codice interpretato, come indicato nella bibliografia [CQ96].

L'interprete Java, od opzionalmente il compilatore JIT, operano nel contesto del Sistema Runtime (threads, memoria ed altre risorse di sistema), come illustrato in figura 4.1.

Per eseguire i bytecode su una macchina reale, bisogna realizzare un ***emulatore*** della macchina virtuale, il cui compito è tradurre i bytecode in istruzioni native della piattaforma Hardware/Software ospite. Vari modi sono possibili per ottenere emulatori della macchina virtuale Java:

- all'interno di un Web *browser*: in questo caso l'emulatore della macchina virtuale risiede all'interno del codice eseguibile del *browser* stesso;
- all'interno del Sistema Operativo.

Nel secondo caso, la JVM è in grado di eseguire non soltanto gli *applet*, ma anche applicazioni vere e proprie scritte in Java (*stand-alone*), cioè programmi del tutto autonomi che possono risiedere permanentemente sulla macchina ospite ed essere eseguiti senza l'utilizzo del Web *browser*[2].

[2] Sono in corso di sviluppo progetti volti a realizzare un chip che implementi la JVM direttamente in hardware, con ovvi benefici dal punto di vista delle prestazioni.

L'indipendenza dei *bytecode* dalla specifica piattaforma risulta dall'aver adottato lo standard *Unicode* per la rappresentazione dei caratteri, lo standard *IEEE 754* per i numeri in virgola mobile e per aver assunto lunghezze fisse, in termini di bit, per rappresentare gli interi[3].

Il codice .class indicato nella figura 4.1 è composto, come detto, da un insieme di moduli oggetto in relazione al numero di file .java definiti nel programma sorgente. Ciascun modulo viene memorizzato in un file di formato binario come una sequenza di byte (8-bit) che rappresentano appunto i *bytecode*, mantenendo i riferimenti sia interni che esterni al modulo in forma simbolica. In fase di esecuzione, la JVM invoca un *ClassLoader* (una classe definita nelle librerie standard) che, dinamicamente, carica nello spazio di memoria assegnato alla JVM i moduli .class e verifica i *bytecode* di cui sono composti. La macchina su cui è implementata la JVM interpreta i *bytecode* rispondenti, in termini di rappresentazione, agli standard sopra citati, e li esegue rispettandone la semantica.

Il processo di verifica dell'integrità dei *bytecode* garantisce che la struttura di ciascun modulo sia sintatticamente corretta ed in accordo con il formato atteso. Tale processo introduce una ridondanza sui controlli fatti durante la compilazione, ma ne giustifica la presenza il fatto che il codice dei moduli .class stessi potrebbe essere ricevuto da piattaforme remote tramite rete e direttamente eseguito.

[3] Tali scelte sono dettate dal fatto che le CPU più diffuse condividono e possono facilmente supportare tali caratteristiche.

In relazione a quest'ultimo concetto, occorre sottolineare che alcuni dei metodi dichiarati nelle classi appartenenti alle librerie standard, e tra esse la suddetta classe *ClassLoader* ad esempio, necessitano di fatto una implementazione scritta in codice nativo, dipendente pertanto dalla piattaforma[4], come illustrato in figura 4.2:

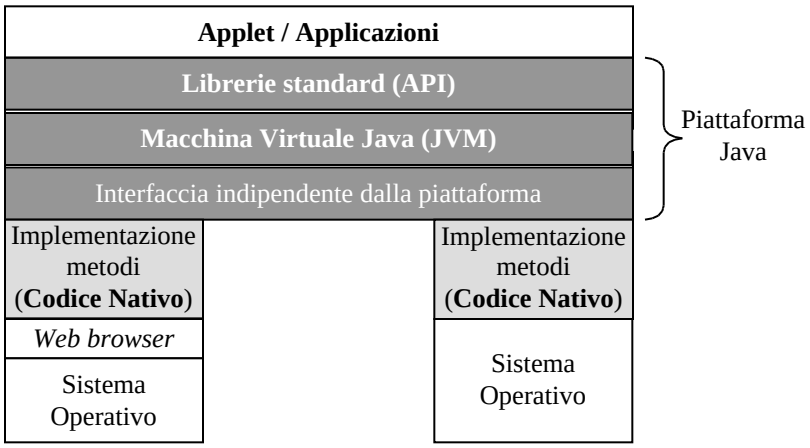

Fig. 4.2: Strutturazione a livelli della piattaforma Java

Pertanto, al fine di eseguire un insieme di moduli *.class* (ricevuti tramite rete o generati localmente) su una qualsiasi macchina fisica, occorre:

- definire una implementazione della JVM;
- disporre localmente delle suddette librerie standard, come indicato anche in figura 4.1;
- definire un insieme di librerie, scritte per quella particolare piattaforma e quindi in codice nativo, che implementano tali metodi (figura 4.2).

[4] Questo giustifica il motivo per il quale le librerie standard delle classi appartengono alla piattaforma Java, e sono quindi anch'esse inserite nei Web *browser*.

4.3 Il linguaggio di programmazione Java

Java è un linguaggio di programmazione orientato agli oggetti. Il termine "orientato agli oggetti" significa rappresentare le componenti del problema da risolvere, nel caso specifico un'interfaccia grafica per il controllo da remoto di una rete di STC104, strutturando il software come un insieme di elementi, gli oggetti appunto. Ciascun oggetto è caratterizzato da proprie variabili ed operazioni: quest'ultime possono essere effettuate sulle medesime variabili o su quelle di un altro oggetto, invocando le operazioni in esso definite.

La struttura di un programma scritto in Java si basa sui concetti di classe, interfaccia e libreria [GJS96]:

- dichiarare una classe (*class*) significa definire un nuovo tipo con diritti d'accesso e variabili (*fields*) che rappresentano gli attributi della classe, e implementare inoltre i metodi (*methods*) che operano su tali variabili. La classe è sostanzialmente una *classificazione* di oggetti aventi le stesse caratteristiche;
- una interfaccia (*interface*) definisce un nuovo tipo, con costanti che ne rappresentano gli attributi, e i cui metodi sono dichiarati ma non implementati;
- una libreria (*package*) è un insieme di classi raggruppate secondo uno specifico criterio.

Un esempio di dichiarazione di una classe è illustrato nella figura seguente:

```
public class Link {
   int Index;          /* Attributo */
   public Link (int numero){     /* Costruttore */
      Index = numero;
   }
   public int get_index(){     /* Metodo */
      return index;
   }
   public String text(){     /* Metodo */
      return ("Indice Link"+toString(get_index()));
   }
}
```

Fig. 4.3: Codice sorgente Java per la dichiarazione di una classe

La classe **`Link`** potrebbe ragionevolmente rappresentare un link dello switch STC104 il cui attributo è un intero, `Index`, che identifica l'indice del link nei 32 che compongono lo switch. Il metodo `get_index()` restituisce un intero che identifica l'indice stesso, mentre il metodo `text()` restituisce una stringa di caratteri che potrebbe descrivere il link. La creazione e l'inizializzazione di un oggetto di tipo **`Link`**, che istanzia cioè tale classe, si ottiene invocando il costruttore della classe che, nel linguaggio Java, è un metodo il cui nome coincide con il nome della classe stessa

(figura 4.3). Una generica classe può eventualmente dichiarare più costruttori e/o metodi con lo stesso nome ma con segnatura[5] differente.

In merito alla implementazione dei metodi, è possibile scrivere programmi in codice nativo (*native methods*), utilizzando ad esempio linguaggi come il "C" , al fine di realizzare particolari funzioni che, per la loro natura, sono specifiche della piattaforma su cui l'applicazione Java deve essere eseguita. All'interno della classe scritta in Java tali metodi sono dichiarati con la parola chiave `native`, ma non implementati:

```
. . .
public native int Metodo(. . .){};
. . .
```

Il programma scritto in codice nativo, preventivamente tradotto con lo specifico compilatore, viene caricato ed eseguito con modalità dipendenti dalla specifica piattaforma.

Il modello di generalizzazione delle classi è quello noto come *single inheritance*, secondo il quale una classe (*subclass*) può direttamente ereditare ed arricchire, con la definizione di nuove variabili e/o metodi, le caratteristiche di una sola classe (*superclass*).

Ad esempio, potremmo generalizzare la classe **`Link`** arricchendola con un attributo che ne identifichi lo switch di appartenenza nel caso volessimo rappresentare il link stesso in una rete di STC104:

[5] La segnatura di un metodo è composta dal suo nome, dal numero e tipo dei parametri formali e dal tipo del valore di ritorno.

```
public class NetLink extends Link {
  int numSwitch;
  public NetLink (int switch, int indice){
    numSwitch = switch;
    super(indice); /* Invocazione Costruttore superclass */
  }
  public int get_switch(){
    return numSwitch;
  }
  public String text(){
    return ( "Indice Switch"+toString(get_switch())+
           "Indice Link"+toString(get_index()) );
  }
}
```

Fig. 4.4: Generalizzazione di una classe

Ogni oggetto di tipo **NetLink** è un **Link** perfezionato con l'attributo `numSwitch` di tipo intero, ed eredita dalla classe che estende gli attributi ed i metodi in essa definiti. Una *subclass* che definisce un metodo con la stessa segnatura di un metodo della *superclass* che estende, come `text()` di figura 4.4, ne sostituisce (*overriding*) l'implementazione.

La scelta del modello *single inheritance*, sebbene restringa la gerarchia delle classi ad un albero [RB+91], evita potenziali problemi di ambiguità nella ereditarietà degli attributi e delle implementazioni dei metodi[6].

La soluzione adottata dal linguaggio Java [GJS96] consiste nel permettere ad una classe di implementare direttamente anche i metodi di una o più interfacce (*implements*), con la condizione di implementare tutti i metodi dichiarati da ciascuna di esse.

Di conseguenza, l'oggetto che istanzia una classe è anche una istanza della classe estesa e di tutte le interfacce eventualmente implementate, permettendo così agli oggetti stessi di supportare caratteristiche comuni a più classi senza peraltro condividere alcuna implementazione.

Possiamo ora delineare la sequenza delle attività che un utente intraprende qualora volesse controllare da remoto una rete di STC104:

- tramite il *Web-browser*, l'utente accede ad un documento *Web* che in realtà contiene un *link* ad un programma scritto in Java, lo *applet*. Quest'ultimo è costituito da un insieme di classi che definiscono l'interfaccia grafica che rappresenta la rete stessa e che viene visualizzata all'interno del *browser* stesso;
- sulla medesima piattaforma di provenienza dello *applet* è in esecuzione l'agente di controllo dello switching network, il quale accede direttamente allo hardware da controllare eseguendo i

[6] Una classe potrebbe estendere direttamente più di una superclass con lo stesso attributo o con un metodo la cui segnatura è uguale ma implementato in maniera differente.

comandi contenuti nei messaggi di richiesta definiti al capitolo 3. In questa fase delle attività, non è stata ancora attivata alcuna connessione TCP sulla *porta* di ascolto dell'agente di controllo;

- sfruttando il supporto al *networking* offerto dall'ambiente di programmazione Java, lo *applet*, su richiesta dell'utente, attiva una connessione TCP con l'agente di controllo al fine di monitorare e controllare la rete di STC104 basandosi sul protocollo definito al capitolo 3.

Si vuole cioè sottolineare il fatto che la ricezione e conseguente visualizzazione nel *browser* del programma Java (lo *applet*), che costituisce il pannello di controllo da remoto, è una attività "trasparente" all'agente di controllo degli *switch*; inoltre, l'agente di controllo stesso, sviluppato nel presente lavoro, non risponde solamente a richieste di connessione provenienti da uno *applet*, ma da un qualunque processo remoto, implementato con un qualsiasi linguaggio di programmazione, che richieda una connessione TCP sulla medesima porta di ascolto del *server*.

4.4 Librerie standard

L'ambiente di programmazione Java include un insieme di librerie di classi e di interfacce generiche che possono essere utilizzate direttamente oppure generalizzate dall'utente per definire classi specifiche, avvalendosi così di una struttura già esistente. Al fine di comprendere quanto è stato sviluppato

per realizzare il pannello di controllo, si darà nei seguenti paragrafi una descrizione delle librerie standard di supporto nei seguenti settori:

- Input/Output;
- Comunicazioni di rete;
- *Multithreading*;
- Interfacciamento grafico.

La definizione di tutti i metodi dichiarati all'interno delle classi appartenenti alle librerie standard è disponibile in [GY+96].

4.4.1 Gestione dell'Input/Output

L' I/O in Java è definito in termini di *stream* (flussi). Gli *stream* sono un'astrazione di alto livello per rappresentare la connessione a un canale di comunicazione orientato al *byte*. Esso può essere stabilito tra entità molto diverse, le più importanti delle quali sono:

- un file;
- una connessione di rete, come ad esempio TCP/IP;
- un buffer in memoria.

Uno stream rappresenta di fatto un punto terminale di un canale di comunicazione **unidirezionale**, e può leggere dal canale (InputStream) o scrivervi (OutputStream):

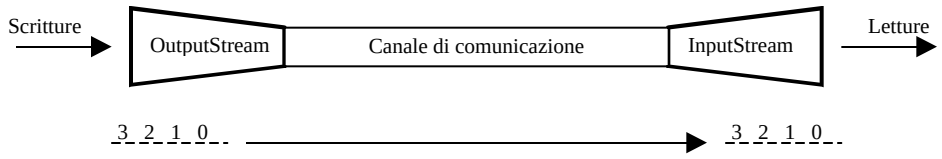

Fig. 4.5: Stream di input e output

Tutto ciò che viene scritto da un OutputStream viene letto dall'altra parte dal corrispondente InputStream. Le loro principali proprietà sono:

- disciplina FIFO, per cui ciò che viene scritto da un OutputStream viene letto nello stesso ordine dal corrispondente InputStream;
- uno *stream* consente la lettura o la scrittura, non entrambe le cose. Nel caso delle connessioni di rete, dove chiaramente si richiede l'accesso sia in lettura che in scrittura al Socket, si ottengono infatti due distinti *stream* per le due modalità d'accesso;
- generano eccezioni in caso di errore, che possono essere gestite, come tutte le eccezioni in Java, con la clausola

```
try { blocco di operazioni; }
catch ( Exception e) { gestione eccezione; }
```

- sono bloccanti: una operazione di lettura (read( )) blocca il chiamante finché i dati non sono disponibili, mentre una di scrittura (write( )) lo blocca finché non è completata.

4.4.2 Comunicazioni di rete

Come descritto nel capitolo 4, la vita di una connessione TCP si articola in tre fasi:

- apertura della connessione;
- dialogo per mezzo della connessione;
- chiusura della connessione.

Anche un'applicazione Java, com'è lecito aspettarsi, segue lo stesso percorso. Attraverso la classe Socket, che rappresenta l'estremità locale di una connessione TCP, si può aprire una connessione con un *server* in ascolto su una porta nota a priori. La creazione di un oggetto Socket, con opportuni parametri, stabilisce automaticamente una connessione TCP con l'host remoto specificato, a meno di condizioni di errore, nel qual caso viene generata un'eccezione (host sconosciuto, nessun processo *server* in ascolto su quella porta, ecc.).

A titolo d'esempio, si riporta di seguito un frammento di programma scritto in C, secondo quanto descritto nel paragrafo 3.5, per stabilire una connessione con l'host dove attualmente e' installato il *server* sviluppato:

```
                          /* Indirizzo IP 141.108.2.65 */
#define SERV_ADDR    ( (unsigned long int) 0x8d6c0241)

#define SERV_PORT    6884
struct sockaddr_in server;
struct sockaddr *name;

if ( (sock = socket (AF_INET, SOCK_STREAM, 0) >0 ) {
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = htonl (SERV_ADDR);
  server.sin_port = htons (SERV_PORT);
  name = (struct sockaddr *) &server;
  if ( (con = connect (sock, name, sizeof (*name))) >0 ) {
    /* Connessione stabilita */
    . . .
    . . .
  } else {
    /* Errore chiamata connect */
  }
} else {
  /* Errore chiamata socket */
}
```

Fig. 4.6: Set di istruzioni C del client per stabilire una connessione con un server in ascolto sulla porta 6884

In Java, l'interfaccia al protocollo TCP è notevolmente semplificata. La connessione viene stabilita, come detto, creando un oggetto di tipo Socket:

```
import java.net.*;
. . .
try {
  sock = new Socket (host,6884);
} catch ( Exception e) {
  gestione eccezione;
}
. . .
```

Fig. 4.7: Attivazione di una connessione TCP in Java: `host` *può specificare il nome(DNS) o l'indirizzo IP in dotted decimal notation*

Una volta che l'oggetto `sock` è stato creato, da esso si possono ottenere, attraverso i metodi getInputStream( ) e getOutputStream( ) definiti nella classe Socket stessa, gli *stream* per leggere e scrivere i dati.

Relativamente al progetto, l'agente di controllo della rete di STC104 è stato sviluppato utilizzando il linguaggio C con sistema operativo Linux, per cui in corrispondenza dello OutputStream (InputStream) del *client* non esiste un InputStream (OutputStream) in Java del *server*, ma un descrittore di socket TCP sul quale il processo *server* legge e scrive i dati:

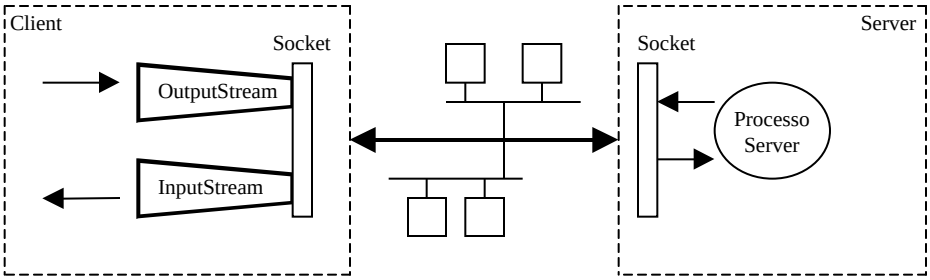

Fig.4.8: Connessione tra un client Java ed un server

Nel caso dello sviluppo di *applet*, è possibile attivare una connessione con il solo host di provenienza dello *applet* stesso. Ciò non costituisce una limitazione per i nostri scopi, in quanto l'host di provenienza dello *applet*, che costituisce il pannello di controllo, è il medesimo host sul quale viene eseguito l'agente di controllo della rete di STC104.

4.4.3 *Multithreading*

L'ambiente di programmazione Java fornisce il supporto al *multithreading* attraverso la classe Thread. Ogni oggetto che istanzia tale classe costituisce un flusso separato di elaborazione[7]. Una volta creato, l'attivazione e la terminazione del flusso di elaborazione avviene invocando rispettivamente i metodi start( ) e stop ( ) sull'oggetto. Il codice che specifica l'elaborazione vera e propria del *thread* si definisce all'interno del metodo run( ). Una limitazione imposta ad un oggetto che estende la classe Thread è che in esso

[7] Un thread, a differenza di un vero processo, è un contesto di esecuzione il cui spazio di indirizzamento viene ricavato all'interno di quello del processo che lo ha generato. Spesso si usa come sinonimo di thread il termine *lightweight process* ("processo leggero").

non possono essere definiti più flussi di elaborazione. Per risolvere tale problema si utilizza l'interfaccia Runnable:

```
public CommClass implements Runnable {
. . .
}
```

e definire al suo interno il metodo run( ) .

Per gestire il dialogo sulla connessione TCP, è stata definita una classe che si occupa della comunicazione dei dati con il *server* e che implementa appunto l'interfaccia Runnable. La ricezione dei dati relativi al *polling* dei **Link0-31Status** avviene in un *thread* separato tramite la definizione del metodo run( ), costituito da un ciclo infinito in cui si ricevono tali dati dal *server* (si legge cioé dallo InputStream del Socket) e si provvede alla loro elaborazione, mentre il flusso di esecuzione principale invia (scrive sullo OutputStream) i messaggi di richiesta specificati dall'utente e attende le relative risposte (sempre dallo InputStream).

Come si nota, e come è già stato descritto nel paragrafo 4.4.1, i due flussi di elaborazione condividono lo stesso InputStream. In Java, per risolvere i problemi legati alla concorrenza, è stata incorporata nella classe Object, la radice di tutte le classi, la capacità di funzionare come un *monitor* (Hoare, 1974); tale funzionalità si attiva ricorrendo ai metodi sincronizzati:

```
public void synchronized Metodo(. . .){
  . . .
}
```

Quando un *thread* chiama un metodo sincronizzato di un oggetto, acquisisce un lucchetto su quell'oggetto. Di conseguenza, qualsiasi altro *thread* di esecuzione che invoca lo stesso od un altro metodo sincronizzato sullo stesso oggetto viene messo in attesa finché il lucchetto non viene rilasciato.

In tale ottica, i metodi dichiarati all'interno della classe responsabile delle comunicazioni con il *server* sono tutti sincronizzati:

- quando l'utente invia un messaggio di richiesta al *server*, il metodo relativo a tale richiesta si pone in attesa di ricevere o i dati di risposta oppure quelli relativi al *polling*; in quest'ultimo caso il pannello di controllo del *client* viene aggiornato, e ci si pone nuovamente in attesa di ricevere il messaggio di risposta. Solo quando ciò accade il metodo termina rilasciando il lucchetto sull'oggetto;
- il *thread* responsabile del monitoraggio della risorsa verifica, con un intervallo di tempo settabile da programma e invocando un metodo sincronizzato sullo stesso oggetto, se sullo InputStream del Socket ci sono dei dati disponibili (attraverso il metodo available( ) si può eseguire sostanzialmente una *receive* non bloccante). In caso positivo, per quanto detto, possono essere solo dati relativi al polling della rete di STC104 e, conseguentemente, viene aggiornato il pannello di controllo.

4.4.4 Interfacciamento grafico

Per implementare, ad esempio, l'interfaccia grafica (*Graphical User Interface*, **GUI**) di un programma scritto con un linguaggio che non sia Java sul sistema operativo Linux, occorre conoscere le librerie che implementano le risorse GUI di *Xwindow*[8], l'interfaccia grafica di Linux e le relative funzioni di libreria della particolare implementazione del linguaggio sotto Linux. Analogamente dicasi per l'interfacciamento grafico dello stesso o di un altro programma che debba girare su Microsoft Windows. Poiché i due ambienti sono incompatibili, occorrerebbe riscrivere l'interfaccia grafica se si desiderasse portarla da un sistema operativo all'altro. In Java, i dettagli per la gestione delle risorse GUI sono uguali, dal punto di vista del programmatore, per qualunque sistema operativo; tali risorse sono implementate attraverso i metodi e le interfacce dello *Abstract Window Toolkit* (AWT), che fa parte appunto delle librerie standard .

Lo AWT di Java fornisce molti componenti (Component) standard per realizzare le GUI, come bottoni, menu, liste, aree di testo ed altri. Tali componenti possono essere racchiusi in vari Containers e la loro disposizione sullo schermo può essere gestita utilizzando dei LayoutManagers. Ad esempio, lo *applet* stesso è tecnicamente un pannello (Panel), che a sua volta estende la classe Container, visualizzabile all'interno dei *Web browser*.

Quando l'utente pigia il tasto del mouse o digita sulla tastiera, il sistema operativo produce un evento. Questo evento viene passato alla piattaforma Java, che a sua volta identifica il componente per il quale è stato generato

[8] *Xwindow* è il sistema di interfacciamento grafico a finestre per gli ambienti Unix.

l'evento e lo presenta all'applicazione attraverso un oggetto che istanzia la classe Event.

Le azioni da svolgere in risposta ad un particolare evento (MOUSE_DOWN, MOUSE_UP, KEY_PRESS, KEY_RELEASE ed altri) vengono specificate dal programmatore definendo il metodo handleEvent (Event e, int x, int y) all'interno della classe:

```
. . .
public boolean handleEvent(Event e, int x, int y){
        if (e.id == Event.MOUSE_DOWN) {
           . . .
           return true;
        }
}
. . .
```

dove gli interi `x` ed `y` sono le coordinate del punto in cui l'evento si è verificato.

In alternativa, l'evento può essere direttamente intercettato definendo il metodo mouseDown(Event e, int x, int y), o i corrispondenti di altri tipi di evento. Ad esempio, nel codice che implementa il *client* è stata definita la classe Link, che rappresenta di fatto un DataLink dello STC104, come una estensione della classe standard Canvas[9]; in essa è stato definito il metodo

[9] Un *canvas* rappresenta un'area rettangolare dello schermo sulla quale un'applicazione può effettuare delle operazioni grafiche e alla quale lo AWT trasmette gli eventi generati dall'utente.

mouseDown nel quale, controllando il valore x , viene inviato all'agente di controllo della switching network un messaggio di richiesta di reset del DataLink stesso oppure di lettura dei dati dei registri con i quali è stato configurato.

Capitolo 5

Descrizione del software sviluppato

In questo capitolo intendo descrivere i programmi sviluppati per il presente lavoro che realizzano le funzioni di comunicazione TCP/IP e controllo della rete di STC104 dell'agente di controllo, e delle classi Java per la realizzazione del pannello di controllo remoto della rete stessa. I codici sorgente di entrambi i programmi, opportunamente commentati, sono riportati nella Appendice A di questa tesi.

5.1 L'agente di controllo

Il software che realizza le funzioni di controllo e monitoraggio della rete di STC104 e delle comunicazioni con i processi remoti è stato sviluppato in linguaggio C su Sistema Operativo Linux-RedHat 4.1; il programma è attualmente installato ed in funzione su un Personal Computer IBM-compatibile con CPU 80486/25Mhz dotato di 16 Mbyte di memoria RAM, il cui nome identificativo è **pcape1.roma1.infn.it**, (indirizzo IP : **141.108.2.65**).

Lo schema generale delle attività svolte dal programma, denominato **C104_ctrl**, è illustrato nella figura seguente:

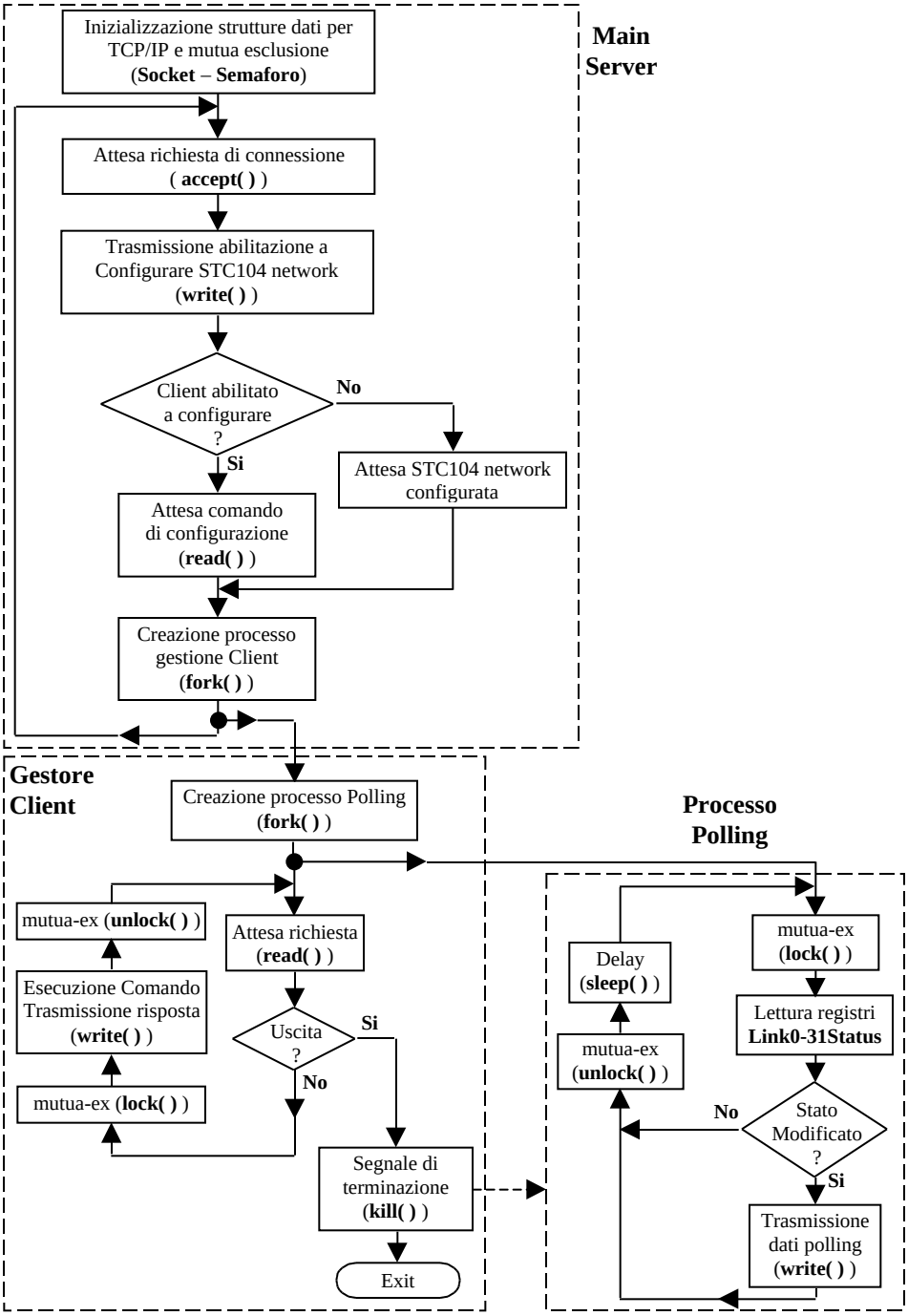

Fig. 5.1: Diagramma di flusso dell'agente di controllo

All’interno dei blocchi di figura 5.1 sono indicate, in **grassetto**, le principali procedure e chiamate di sistema che saranno descritte nel seguito del capitolo.

Le distinzioni del processo **Main Server** di figura 5.1 rispetto al caso più generale descritto nel capitolo 3 (figura 3.14) sono dovute alle seguenti specifiche:

- ad ogni istante, solo una delle connessioni attivate con i processi client è abilitata alla configurazione della rete di STC104;
- tutti i processi clienti devono poter accedere, almeno in lettura, ai registri di configurazione dei link di ciascuno STC104 della rete stessa.

Per soddisfare tali requisiti, il processo **Main Server** abilita all’ operazione di configurazione il processo remoto la cui richiesta di connessione è la prima accettata in ordine di tempo tra quelle presenti nel sistema: poiché la chiamata *accept( )*, come già descritto nel capitolo 3, blocca il flusso di esecuzione in attesa di richieste di connessione e ritorna il nuovo *socket* quando la fase di attivazione di una connessione è completata, si tiene traccia del numero di connessioni presenti nel sistema incrementando una variabile globale (`NumOfClientsConnected` nel programma) non appena la *accept( )* ritorna, e viene trasmesso al processo remoto il valore di tale variabile prima di creare il processo **Gestore Client** (figura 5.1) al quale verrà demandato il compito di servirne le richieste. Per tale ragione, solo il *client* il cui **Gestore Client** avrà

`NumOfClientsConnected = 1` sarà abilitato a configurare la rete di STC104; tale messaggio di risposta alla richiesta di connessione provocherà conseguentemente la abilitazione/disabilitazione del pulsante di configurazione che appare, come vedremo, nel pannello grafico di controllo remoto. Abbiamo quindi due possibili situazioni:

Client abilitato a configurare

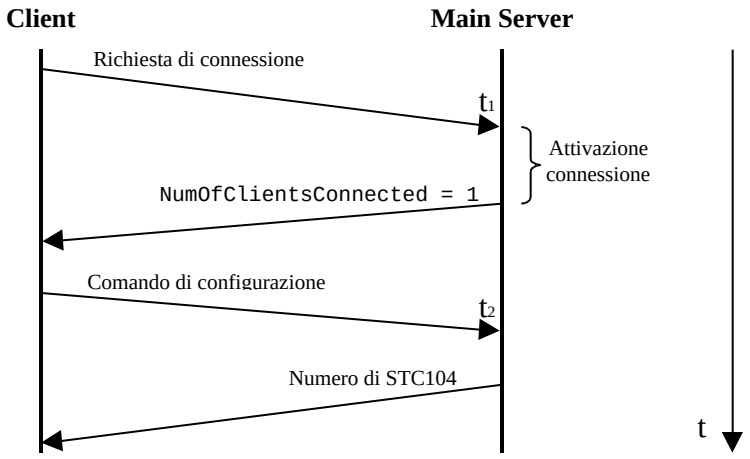

Fig. 5.2: Fase iniziale: client abilitato a configurare

Nell'intervallo di tempo $t_1<t<t_2$ di figura 5.2, ogni altra richiesta di connessione rimane pendente in modo che qualsiasi client non abilitato alla operazione di configurazione acceda alla rete di STC104 quando essa è già stata configurata;

Client non abilitato a configurare

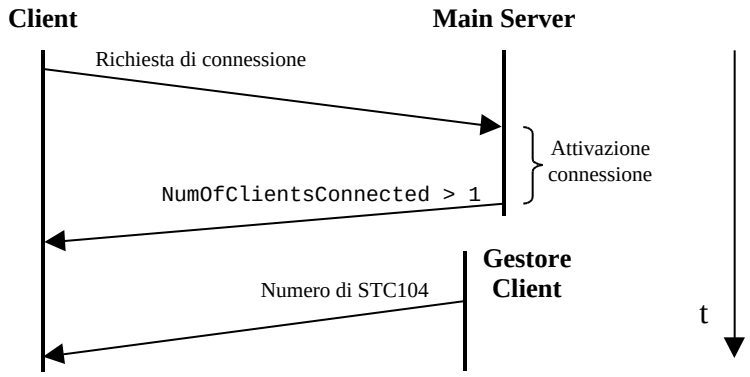

Fig. 5.3: Fase iniziale: client non abilitato a configurare

Main Server prosegue nella sua esecuzione con la creazione del **Gestore Client** che, come prima operazione, trasmette al *client* Java il numero di switch STC104 di cui è composta la switching network, "simulando" quindi un messaggio di risposta ad un comando di configurazione.
In entrambi i casi illustrati, la trasmissione dello stato iniziale della risorsa, ossia dei registri **Link0-31Status**, avrà luogo alla prima interrogazione del processo **Polling** di figura 5.1.

L'operazione di configurazione, i cui dettagli saranno descritti nel successivo paragrafo, e' eseguita tramite una funzione che restituisce un puntatore, inizialmente di valore indefinito, ad una struttura dati tramite la quale, una volta configurata la rete di STC104, si accede alla parte di controllo degli switch. La creazione di un nuovo processo viene realizzata, per replicazione, tramite la chiamata di sistema *fork( )* (figura 5.1) che copia l'immagine del processo genitore, ossia il processo che la invoca,

nella memoria allocata per il processo figlio. Per questa ragione, e' fondamentale che la configurazione della rete di switch sia eseguita all'interno del **Main Server** prima di creare un nuovo **Gestore Client**; in caso contrario, le successive richieste di connessione sarebbero gestite da processi il cui puntatore alla suddetta struttura dati avrebbe valore indefinito, nonostante la risorsa sia già stata configurata.

Discendendo nel diagramma di flusso di figura 5.1, una volta che la risorsa è configurata, il **Gestore Client** crea un ulteriore processo, il processo **Polling**, al fine di monitorare, con un intervallo di tempo settabile da programma (Delay in figura 5.1), il valore dei registri **Link0-31Status** dei 32 data-link di ciascuno STC104: è sostanzialmente un ciclo infinito nel quale la lettura di tali registri viene confrontata con la lettura effettuata al ciclo precedente; il confronto tra le due determina o meno la trasmissione del nuovo stato al client Java. Come descritto nel capitolo 2, questi registri hanno 6 bit accessibili in lettura che indica lo stato del DataLink: il processo **Polling** dichiara al suo interno due matrici di *byte* (8-bit) che contengono i valori dei 32 DataLink di ciascuno STC104 per effettuare il confronto tra due letture successive, per cui se la switching network è composta da n STC104, ciascuna matrice occupa $(n * 32)$ byte di memoria per un totale di $(2 * n * 32)$ byte. Se si fosse implementato anche il monitoraggio dello switching network secondo il modello client-server, il processo **Polling** non avrebbe ragione di esistere, e le richieste di "polling" verrebbero gestite dal processo **Gestore Client**.

All’atto della ricezione del comando di uscita dal client Java, il **Gestore Client** invia un segnale di terminazione forzata (kill( ) in figura 5.1) al processo **Polling** prima di terminare egli stesso. Nel **Main Server** è stata dichiarata una funzione per intercettare la terminazione dei processi generati [CS94] con il duplice scopo di:

- tenere traccia del numero di connessioni presenti nel sistema, decrementando la variabile `NumOfClientsConnected`, così da poter assegnare ad una nuova connessione l’abilitazione a configurare la rete di switch;
- impedire che essi, all’atto della loro terminazione, entrino in un particolare stato denominato *stato zombie*, così detto in quanto il processo, sebbene sia terminato, rimane allocato nella tabella dei processi del Sistema Operativo finché il processo che lo ha generato non ne intercetti la terminazione.

5.1.1 Il software di accesso allo STC104

Il programma **C104_ctrl** utilizza alcune funzioni presenti nelle librerie incluse nel ToolSet D7394A della INMOS [SGS95b]. Nella tabella 5.1 che segue vengono riportate quelle utilizzate per le operazioni di configurazione, controllo e monitoraggio della rete di STC104:

Nome funzione	**Descrizione**
InitializeDeviceNoStart	Inizializza i registri di entrambi I C101 presenti sulla scheda B108.
Configure	Configura la rete di STC104
ResetC104Link	Specificati un STC104 ed un link, ne effettua il reset.
BacoReset	Specificati un STC104 ed un livello (1-2), ne effettua il reset a quel livello.
Cpeek	Specificati un STC104 ed un registro, ne legge il contenuto.

Tabella 5.1: Funzioni di configurazione ed accesso ai registri con relativa descrizione

La funzione **InitializeDeviceNoStart**, oltre ad inizializzare i C101 come riportato nella Tabella 5.1, specifica l'indirizzo base con il quale si accede allo spazio di indirizzamento di I/O del PC relativo alla scheda B108; le funzioni di seguito descritte effettuano pertanto le operazioni di lettura/scrittura dei registri specificando uno sfasamento (*offset*) da sommare all'indirizzo base per raggiungere i componenti hardware presenti sulla scheda.

L'operazione di configurazione viene effettuata dalla funzione **Configure** interpretando un file binario di descrizione della topologia della rete. Quando si devono realizzare delle strutture d'interconnessione con molti componenti e con topologie complesse, la loro configurazione (inizializzazione dei registri, impostazione delle etichette per lo *Interval Labelling*, ecc…) può risultare molto laboriosa e complessa. Al fine di rendere più semplice questa operazione, si utilizza un tool della INMOS comprendente un linguaggio ad alto livello di descrizione dello hardware

*(**N**etwork **D**escription **L**anguage*, **NDL**) ed il relativo compilatore [SGS94b]. Quest'ultimo, oltre a fornire un file binario **NIF** (***N**etwork **I**nitialization **F**ile*), effettua dei controlli sulla giusta connessione tra gli STC104 e riconosce se la rete è potenzialmente affetta da deadlock.

Il file NIF generato può essere così interpretato dalla funzione **Configure** invocata dal programma **C104_ctrl**; ciascuno STC104 viene indicizzato con un intero $i \in [0, n-1]$ per una rete composta di n switch, in modo da poter identificarli univocamente per le successive operazioni di controllo.

La funzione **ResetC104Link** marca come vuoto il buffer dei data token del link ed azzera lo stato dei bit di parità e di riconoscimento token (capitolo 2).

BacoReset effettua invece l'operazione di reset dello STC104:

- al livello 1, vengono disabilitati tutti i DataLink;
- al livello 2, esegue il reset di tutti i 32 DataLink con le stesse modalità del **ResetC104Link**.

In entrambi i casi, i dati memorizzati nei buffer vengono persi, ma i link di controllo **Clink0** e **Clink1** rimangono operativi; per tale ragione è possibile riconfigurare dinamicamente l'intera switching network tramite la funzione **Configure**.

La funzione **CPeek** è utilizzata sia dal processo **Polling** per monitorare i registri **Link0-31Status**, sia dal processo **Gestore Client** per l'esecuzione del comando Link_Info (capitolo 3) relativo alla lettura dei registri di configurazione dei DataLink. Nella figura 5.4 si riporta, a titolo d'esempio, il codice in C per leggere i bit del registro **Link0-31Status** del DataLink **12** dello STC104 con indice **0**:

```
. . .
ControlNetwork->CurrentVirtualLink = 0;
Address = ((0x80+12)<<8) | C104_LINK_STATUS);
Cpeek (Address,&result,ControlNetwork);

if (ControlNetwork->Status == 0) {
  /* Operazione eseguita con successo */
} else {
  /* Errore */
}
. . .
```

*Fig. 5.4: Set di istruzioni C per la lettura del registro **LinkStatus***

La variabile `Address` contiene i due byte utilizzati nel pacchetto di comando Cpeek inviabile allo STC104 (capitolo 2 – figura 2.4), che è costituito dalla somma dell'identificativo del link e del registro scelto[1] (`C104_LINK_STATUS` in figura 5.4), mentre `ControlNetwork` è il puntatore, restituito dalla funzione **Configure**, con il quale si accede alla parte di controllo degli switch.

Il controllo degli errori, con il quale si imposta poi il campo StatusCode dei messaggi di risposta dell'agente di controllo ai processi remoti, viene gestito tramite la variabile `Status` in figura 5.4, il cui valore si riferisce a sua volta al byte di stato contenuto nei pacchetti di handshake dello STC104 (capitolo 2).

[1] Gli indirizzi per specificare i registri di configurazione di un link sono definiti, come costanti, nel file "C104.h" delle librerie del Toolset INMOS.

5.1.2 Il software di comunicazione

La ricezione dei messaggi di richiesta provenienti dallo *applet* Java e la relativa trasmissione delle risposte è stata realizzata utilizzando le chiamate di sistema introdotte nel capitolo 3 (**accept( )**, **write( )** e **read( )** in figura 5.1).

Per ogni connessione stabilita, la **accept( )** ritorna un *descrittore di socket* (un intero a 32 bit); è sufficiente specificare tale intero nei parametri delle chiamate **read( )** e **write( )** dei processi **Gestore Client** e **Polling** associati a quella connessione al fine di, rispettivamente, leggere e scrivere i dati sul canale di comunicazione TCP attivato con il processo remoto uniformandosi al protocollo di applicazione definito al capitolo 3.

Dal punto di vista del programmatore, la scrittura/lettura di dati sul canale di comunicazione è simile ad una operazione di I/O; ad esempio, il processo **Polling** trasmette i nuovi valori dei registri **Link0-31Status** con la seguente istruzione:

write (sock_fd, &buffer, count);

in cui il primo parametro identifica il descrittore di socket ritornato dalla **accept( )**, il secondo specifica l'indirizzo di partenza del buffer dei dati da trasmettere (la matrice dei **Link0-31Status** descritta precedentemente) e count è il numero di byte (8-bit) da trasmettere. Analogamente, con l'istruzione:

read (sock_fd, &Command, 1);

il processo **Gestore Client** può, ad esempio, leggere nella variabile Command il byte che codifica il comando contenuto nel messaggio di richiesta del

processo remoto, eventualmente ricevendo gli ulteriori dati del messaggio di richiesta, ed effettuare la relativa operazione sulla rete di STC104.

Se il dato da trasmettere/ricevere è, anziché un byte come nell'esempio, un valore a 16 o 32-bit (Id. STC104, nei messaggi definiti al capitolo 3, è un intero a 32-bit), possono sorgere dei problemi a causa del modo con cui differenti sistemi memorizzano i dati. Alcuni processori (Intel e Alpha ad esempio) memorizzano i valori associati alle variabili in ordine *little-endian*:

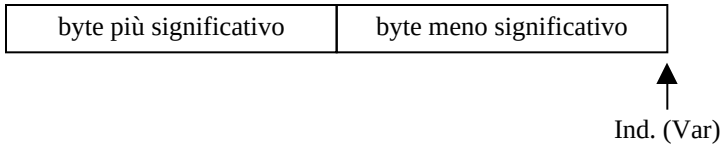

Fig. 5.5: Little-endian: l'indirizzo di memoria di una variabile punta al byte meno significativo

mentre altri (Motorola 68000) memorizzano tali valori in ordine *big-endian*:

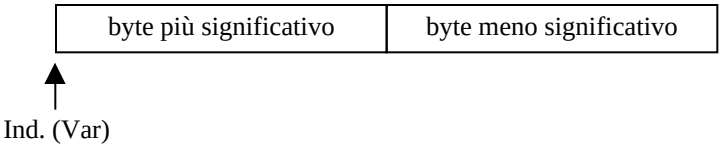

Fig. 5.6: Big-endian: l'indirizzo di memoria di una variabile punta al byte più significativo

Per risolvere tale problema, molti protocolli di comunicazione, e tra essi TCP/IP, utilizzano il formato *big-endian*, detto anche *network byte order*,

per mantenere i dati; la conversione del dato da trasmettere, in C, viene effettuata dalle funzioni di libreria **htonl( )** (**h**ost **to n**etwork **l**ong) e **htons( )** (**h**ost **to n**etwork **s**hort)[2].

Entrambe le chiamate **read( )** e **write( )** ritornano il valore –1 in caso di errore, oppure un intero positivo che contiene il numero di byte effettivamente letti o scritti; a causa dei ritardi di propagazione della rete, può infatti succedere che le due istruzioni ritornino un valore positivo inferiore al parametro count specificato dall'utente. Si ovvia a ciò inserendo le istruzioni in un ciclo che termina solo quando, a meno di condizioni di errore, viene letto/scritto il numero di byte specificato dall'utente.

5.1.3 Sincronizzazione dei processi

Ci troviamo nella condizione in cui due processi distinti, il **Gestore Client** ed il **Processo Polling** di figura 5.1, operano concorrentemente per l'accesso alla rete di switch ed al descrittore di *socket* TCP; il primo per eseguire i comandi contenuti nei messaggi di richiesta provenienti dal client Java, mentre il secondo per monitorare (leggere) il valore dei registri **Link0-31Status** e trasmetterli, qualora cambino, al client Java stesso. Il regime di concorrenza è ancora più spinto, sulla rete di STC104, quando ci sono più client Java connessi con l'agente di controllo, in quanto le coppie **Gestore Client**-**Processo Polling** di ciascuna connessione operano sulla medesima risorsa.

[2] Su un sistema con memorizzazione *big-endian* tali funzioni non fanno nulla, ma è buona norma utilizzarle per garantire la portabilità del programma.

Come gia' descritto, l'invio dei pacchetti di comando allo STC104 sono una sequenza:

- scrittura del comando sul bus di I/O ISA, attraverso le funzioni di Tabella 5.1;
- lettura del risultato.

Bisogna pertanto coordinare gli accessi dei vari processi allo *switching-network* ed al descrittore di socket per garantire la correttezza di tali operazioni.

La soluzione adottata consiste nell'utilizzo di un particolare tipo di semafori, introdotti da Dijkstra nel 1965, al fine di realizzare tali operazioni in regime di mutua esclusione [Tan88]: i ***semafori binari***. Un semaforo binario ha un valore associato, inizializzato nel nostro caso ad 1, che indica sostanzialmente che la risorsa è disponibile, ed una coda di processi. Quando un processo intende accedere alla risorsa, verifica se il valore del semaforo è maggiore di zero (**lock( )** in figura 5.1), e in tal caso decrementa il valore, effettua le operazioni sulla risorsa ed infine ripristina il valore del semaforo ad 1 (**unlock( )** in figura 5.1); in caso contrario il processo viene inserito nella coda associata al semaforo e sospeso in attesa di essere risvegliato dalla **unlock( )** del processo che in quel momento è nella sua sezione critica[3]. Le operazioni di verifica e modifica del valore del semaforo, ed eventualmente sospensione del processo, sono gestite dal nucleo del Sistema Operativo in modo indivisibile, come una singola azione

[3] Si definisce sezione critica la parte di programma in cui un processo accede a dati condivisi.

atomica; ciò garantisce che quando ha inizio una operazione sul semaforo, nessun altro processo può essere schedulato ed accedere al semaforo finché tale operazione non è completata. Per quanto detto è fondamentale che ciascun processo, prima di entrare e uscire dalla propria sezione critica, invochi tali procedure sul semaforo: se, a titolo d'esempio, un processo non invoca l'operazione di **unlock** prima di lasciare la propria sezione critica, tutti i processi che cooperano attraverso tale semaforo verranno messi in attesa non appena intendono accedere alla risorsa condivisa, raggiungendo infine una condizione di *deadlock* in quanto nessun processo sarà in grado di sbloccarli.

Se è vero che, in fase di stesura del codice, è compito del programmatore invocare le procedure di **lock** ed **unlock** rispettivamente all'ingresso ed all'uscita di una sezione critica, è altrettanto vero che un processo, nel caso specifico il **Processo Polling** di figura 5.1, potrebbe essere ucciso quando è all'interno della propria sezione critica; il Sistema Operativo Linux risolve tale situazione riportando il semaforo nello stato in cui era prima di eseguire l'operazione [Rus98].

Le procedure **lock( )** e **unlock( )** sono state realizzate utilizzando la primitiva semop( ) dello ***I**nter**P**rocess **C**ommunication* (**IPC**) di Linux. La descrizione dettagliata della creazione e delle operazioni di controllo dei semafori è stata tratta da [GVdM+95] e [Ste90].

5.2 Il pannello di controllo remoto

Le classi Java che implementano il pannello grafico di visualizzazione e controllo da remoto della rete di STC104 sono state realizzate con lo ***J**ava **D**evelopment **K**it* (**JDK**) versione 1.0.2 su sistema operativo Linux-RedHat 4.1; sono attualmente memorizzate sullo stesso PC in cui risiede l'agente di controllo degli switch, nella porzione del file system riservata ai documenti *Web*[4].

Nella figura 5.7 che segue si riporta, in notazione *Object Modeling Technique* (OMT) tratta da [RB+91], il diagramma delle classi che costituiscono lo *applet* omettendo, per una migliore leggibilità del diagramma stesso, le classi estese appartenenti alle librerie standard di Java utilizzate per la stesura del presente lavoro[5]:

[4] Il direttorio è */home/httpd/html/C104*.

[5] Con il simbolo "◊" si indica la *aggregazione* di più oggetti, i quali rappresentano le componenti di un oggetto che li contiene. Il simbolo "_" numerato indica quante istanze (oggetti) di una stessa classe sono associate ad un oggetto che le contiene.

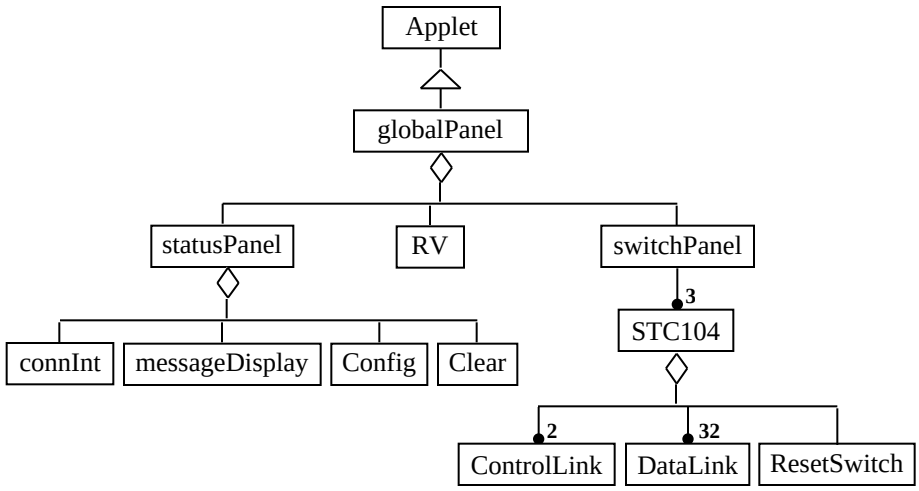

Fig. 5.7: Diagramma di relazione tra le classi

GlobalPanel è la classe che contiene l'intera struttura che costituisce il client Java. Estende la classe standard Applet ed incapsula due pannelli grafici, **statusPanel** e **switchPanel** in figura 5.7, e la classe **RV**, la quale implementa l'interfaccia *Runnable* e contiene i metodi sincronizzati per le comunicazioni con l'agente di controllo della rete di STC104. In relazione ai comandi definiti al paragrafo 3.4.1, essi sono:

```
synchronized int configNetwork();
synchronized void resetLink(int numSwitch, int index);
synchronized void askLinkInfo(int numSwitch, int index);
synchronized void resetSwitch(int numSwitch);
synchronized void sendExit();
synchronized void get_polling();
```

Fig. 5.8: Metodi sincronizzati per l'invio richieste/ricezione risposte al/dal server

Gli *stream* di input e output utilizzati per accedere alla connessione TCP sono due classi standard che estendono InputStream e OutputStream descritte nel capitolo 4, rispettivamente DataInputStream e DataOutputStream: esse forniscono al programmatore dei metodi per leggere e scrivere dati ad un livello di astrazione più elevato (interi, reali, stringhe, ecc.) su un canale di comunicazione orientato al *byte*, ed i valori numerici vengono scritti in *network byte order* (byte più significativo per primo), uno standard universalmente accettato.

Il metodo `get_polling()` è invocato automaticamente, con un intervallo di tempo settabile da programma, dal *thread* d'esecuzione separato da quello principale, e condivide con i restanti metodi elencati in figura 5.8 il DataInputStream associato al Socket.

statusPanel è il pannello grafico nel quale sono stati posizionati un interruttore (**connInt** in figura 5.7) per attivare/disattivare la connessione, un bottone (**Config** in figura 5.7) per trasmettere il comando di configurazione ed una area di testo (**messageDisplay** in figura 5.7) che visualizza i messaggi di risposta dal server.

switchPanel è il pannello contenente la rappresentazione grafica dei tre STC104. Ognuno di essi è composto da un componente grafico per la trasmissione del comando di *Reset* (**ResetSwitch** in figura 5.7) dello STC104, i due link di controllo **Clink0** e **Clink1** (**ControlLink** in figura 5.7) ed i 32 DataLink (**DataLink** in figura 5.7): questi ultimi sono logicamente e graficamente suddivisi in due parti, relative ai comandi di *reset* del DataLink e di lettura dei registri di configurazione. I componenti

appena descritti intercettano il click sul mouse da parte dell'utente definendo il metodo mouseDown( ), al cui interno si invocano i metodi sincronizzati illustrati in figura 5.8 al fine di trasmettere il relativo messaggio di richiesta.

La visualizzazione del pannello di controllo all'interno del *Web browser* è illustrata in figura 5.9:

```
/*********************************************/
/* La figura è memorizzata nel file "appletC104.doc" */
/* Il capitolo prosegue nel file "capitolo5_b".          */
/*********************************************/
```

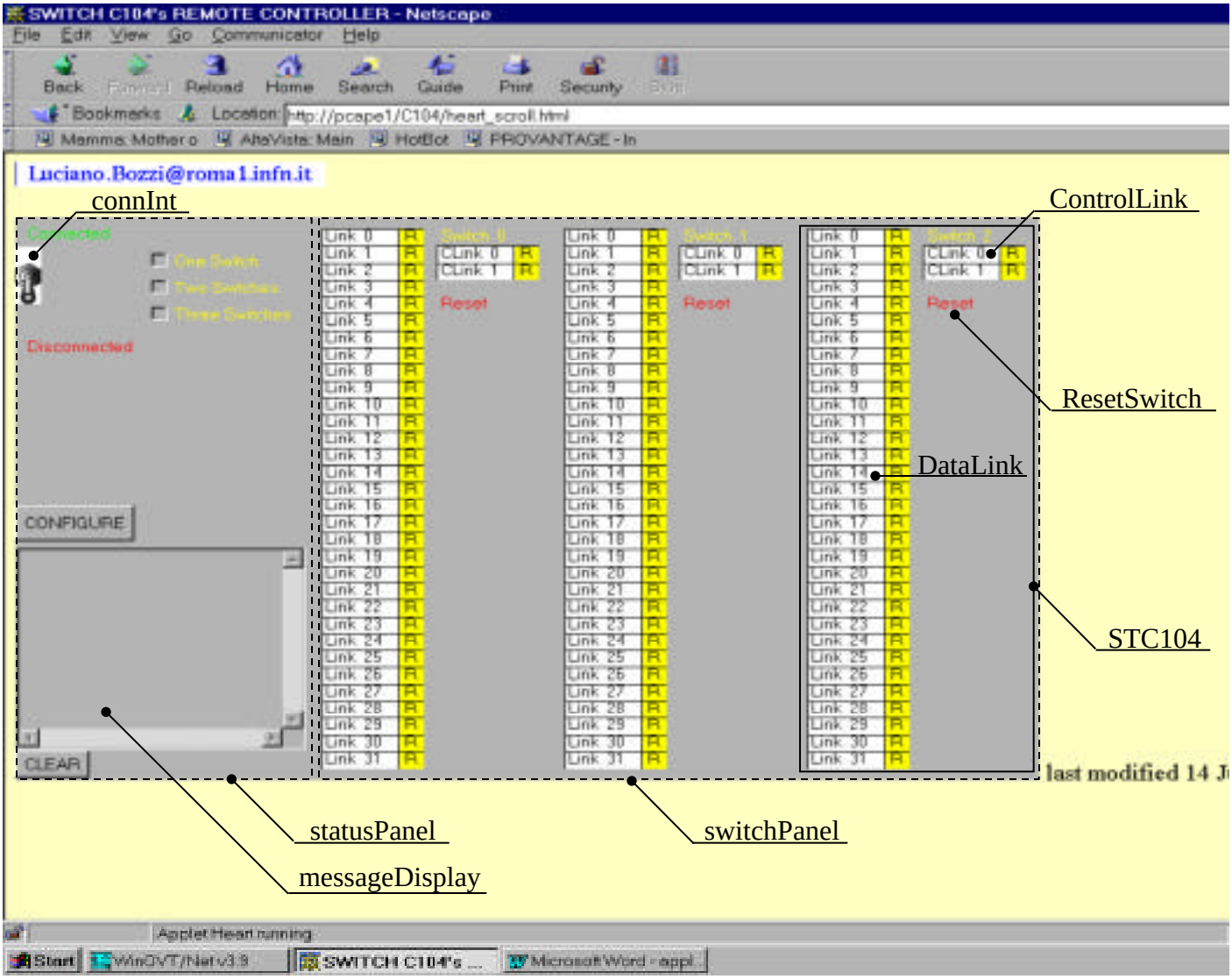

Fig. 5.9: Visualizzazione del pannello di controllo nel Web browser con indicazione delle classi

le classi illustrate, ad eccezione del bottone di configurazione, estendono la classe standard *Canvas* in modo da permettere, oltre all'intercettazione degli eventi, una rappresentazione grafica rispondente all'ambiente di esercizio del pannello di controllo e che possa guidare l'utente nell'utilizzo delle funzionalità del pannello stesso.

Nella soluzione dei problemi riguardanti le interazioni con l'utente, come nel caso di una interfaccia grafica, assumono una particolare importanza quegli aspetti che sono in relazione con i cambiamenti di stato[6] degli oggetti in risposta ad un evento generato dall'utente [RB+91]: è possibile pertanto rappresentare tale relazione, come mostrato nella figura 5.10 che segue, attraverso un grafo i cui nodi sono gli stati ed i cui archi, etichettati dal tipo di evento e dall'oggetto che lo intercetta, rappresentano le transizioni di stato[7]:

[6] Lo stato di un oggetto può essere rappresentato dai valori assunti dai suoi attributi.
[7] Come detto, nel progetto è intercettata la pressione del tasto sinistro del *mouse*. Nel diagramma degli stati, per questa ragione, le frecce sono etichettate con il solo oggetto che intercetta l'evento senza specificarne il tipo.

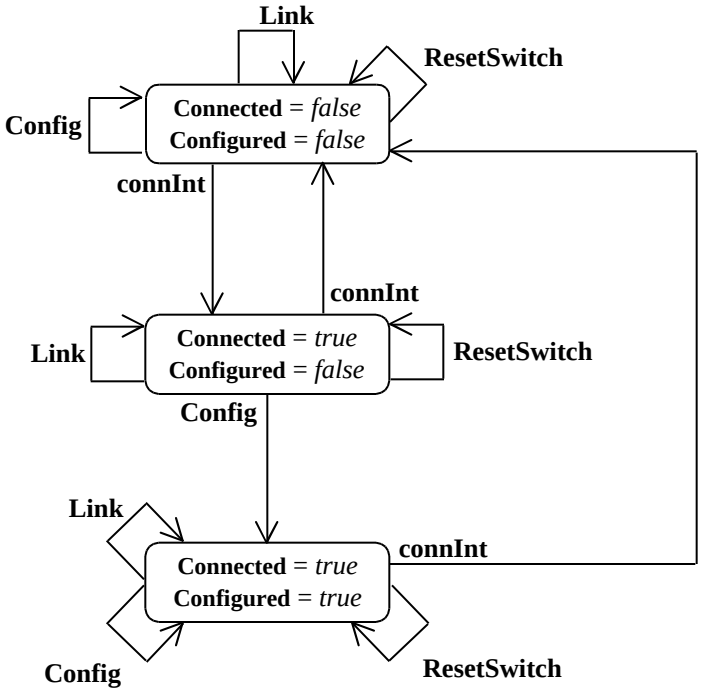

Fig. 5.10: Diagramma degli stati – client abilitato a configurare

Gli attributi **connected** e **configured** di figura 5.10 sono due variabili booleane definite in **globalPanel** e indicano rispettivamente se il sistema è connesso con l'agente di controllo e se la rete di STC104 è configurata.

Dallo stato iniziale, in cui entrambe le variabili hanno valore *false*, si giunge allo stato successivo (**connected** = *true*/**configured** = *false* in figura 5.10) tramite l'interruttore **connInt**; in qualunque altro caso l'utente viene avvertito visualizzando un messaggio sull'area di testo **messageDisplay**.

Ciò avviene anche qualora si richieda, ad esempio, la lettura dei registri di un **Link** nello stato **connected** = *true*/**configured** = *false*, poichè la rete di STC104 non è configurata. Nel caso in cui non si è abilitati a configurare la

risorsa, gli ultimi due stati di figura 5.10 coincidono; all'atto della connessione con l'agente di controllo, si riceve immediatamente, come descritto nel precedente paragrafo, il numero di STC104 di cui è composta la rete, il bottone di configurazione **config** viene disabilitato e un messaggio sull'area di testo avvisa l'utente che il controllo della risorsa in *scrittura* è assegnato ad un altro client.

La visualizzazione dei registri di un **Link** è realizzata tramite la creazione di un *Frame*, una finestra grafica che appare sul pannello di controllo, nel quale vengono riportati i valori dei registri di configurazione in risposta al metodo sincronizzato `askLinkInfo(..)`definito nella classe **RV**. Nella figura 5.11 si mostra una stampa, effettuata durante una prova reale, della visualizzazione dei registri di configurazione di un **DataLink**:

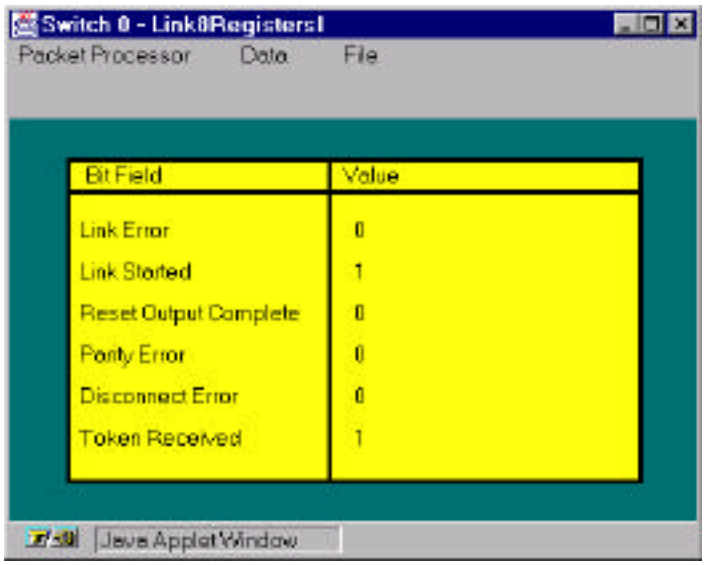

Fig. 5.11: Visualizzazione dei bit del registro LinkStatus del link 8 dello STC104 con indice 0

I bit in figura 5.11 corrispondono al registro **Link0-31Status** dello STC104, ossia il gruppo di registri che vengono interrogati al fine di monitorare la switching network. A tali valori è stato associato un colore di sfondo per ciascun **Link** sul pannello di controllo:

Blu → Link Started;
Verde → Link Started, Token Received;
Rosso → Error.

Nel metodo `get_polling()`, definito nella classe **RV,** viene invocato l'aggiornamento dello sfondo di ciascun **Link** ad ogni cambiamento dei registri **Link0-31Status**.

Attualmente, in risposta al comando di richiesta Link_Info, l'agente di controllo trasmette i bit dei registri **Link0-31Status** e **Packet0-31Mode** (Capitolo 2), visualizzabili tramite il menu che appare in figura 5.11. La tabella dei bit è stata implementata con una classe, **RegTab**, il cui costruttore riceve come parametri un array di stringhe (BitField in figura 5.11) ed i rispettivi valori (value in figura 5.11); se si volesse incrementare il numero di registri da inviare al client, è quindi sufficiente creare un ulteriore oggetto di tipo **RegTab**, specificando nel costruttore i parametri BitField e value.

Conclusioni

Dal punto di vista del monitoring, lo switch STC104 offre senza dubbio il vantaggio di non influenzare il flusso dei dati attraverso i 32 DataLink grazie alla completa indipendenza della sezione di controllo (**Clink0**-**Clink1**) dalla rete dati; malgrado non sia possibile, attraverso il link di controllo, avere una misura del *data rate* effettivo sui DataLink, si è comunque in grado di controllare il corretto funzionamento dello switching network senza pregiudicarne le prestazioni, anche con una frequenza di polling sufficientemente alta.

L'adozione dell'ambiente di programmazione Java, unita all'utilizzo dei browser Web, è particolarmente indicata per quanto riguarda possibili modifiche sia al protocollo di comunicazione dell'applicazione che ai componenti software che realizzano il sistema di controllo remoto (*client* e *server*); infatti, essendo essi residenti sulla medesima piattaforma, si riduce il lavoro di modifica dei programmi che altrimenti dovrebbe essere ripetuto su ciascuna macchina in cui tali programmi vengono eseguiti. Per contro, una siffatta organizzazione prevede la presenza di una "infrastruttura" già esistente, ossia un demone http sulla piattaforma dell'agente di controllo e la presenza di un browser Web sugli host dai quali si intende controllare lo switching network.

Per quanto riguarda lo sviluppo dell'agente di controllo degli switch, si è proceduto in maniera graduale partendo dapprima da uno studio delle funzioni incluse nelle librerie software per accedere allo switching network, passando successivamente al loro utilizzo in "locale", ossia da tastiera;

grazie alla ampia e gratuita disponibilità di documentazione ed informazioni riguardanti il Sistema Operativo Linux ed il linguaggio C, si è potuto dapprima comandare i dispositivi via rete attraverso l'interfaccia di servizio del protocollo TCP, e successivamente affrontare e risolvere il problema degli accessi in regime di concorrenza da parte di più client utilizzando le librerie per la gestione dei semafori.

Durante la fase di sviluppo dello *applet*, invece, si è riscontrato quanto sia importante utilizzare i LayoutManager al fine di posizionare i componenti grafici sullo schermo; solo in questo modo si ha la garanzia che la visualizzazione del pannello di controllo, così come di una applicazione o *applet* Java nel caso più generale, sia interamente indipendente dal Sistema Operativo e dalla particolare versione del browser Web nei quali risiede la JVM.

Appendice A: Codici sorgente

In questa appendice si riportano i listati dei programmi sviluppati nel corso del presente lavoro. Nella Appendice A.1 si illustra il codice sorgente, scritto in C, dell'agente di controllo della rete di STC104 ed una libreria di funzioni da me realizzata (**sem-sock.c**) per la inizializzazione e l'uso dei semafori e dei *socket* TCP. Nella Appendice A.2 vengono invece riportati i listati, scritti in codice Java, che definiscono le principali classi con le quali si è realizzato il pannello di controllo remoto.

Appendice A.1: L'agente di controllo

Il seguente programma, denominato **C104_ctrl**, permette la configurazione, il controllo ed il monitoring di una rete di STC104 da uno host remoto tramite una connessione di tipo TCP/IP sulla porta 6884.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/in.h>
#include <time.h>
#include <asm/io.h>
#include <signal.h>
#include <sys/time.h>
#include <sys/wait.h>
#include "c101.h"
#include "txt9mixv.h"
#include "c104.h"
#include "control.h"
#include "sem-sock.h"

extern struct sockaddr_in client;

VCPLINKBLOCK        *ControlNetwork, TempVcplinkptr;
Channel *FromControlLink, *ToControlLink;
```

```
unsigned long int   ConfigurationStatus;
unsigned long int   NumberOfDevicesInControlNetwork = 0;

int LIVE = 0;
int RegisterAddress = 0x180; /* Indirizzo di I/O della B108
*/
int sem_id, sock;
char seed = 'l';
char buf[10];
unsigned long int NumOfClientsConnected = 0;

void Free_Resources() {

  printf("Freeing Resources..\n");
  close(sock);
  dispose_sem(sem_id);
  exit(0);
}

/* Attesa terminazione figli */
void sig_chld(signal_type)
int signal_type;
{
  int pid;
  int status;

  while( (pid=wait3(&status, WNOHANG, NULL)) > 0);
  NumOfClientsConnected-=1;
  printf("Num. Clients still connected =
         %d\n",NumOfClientsConnected);
}

int request_handler (int sock) {
  int i, k, j, IdSwitch, ind, IdLink;
  int tonet;
  unsigned long int stat,mode;

  do {
    if (sock_read(sock,buf,1)>0) {
      j = (int) buf[0];
      printf("Comando ricevuto %d\n",j);
      switch(j)
      {


/*****************************************************/
       /*  Configurazione del C104 (lettura del file .NIF)
*/

/*****************************************************/
       case 1:
```

```
lock(sem_id);
iopl(3);
InitializeDevicesNoStart(RegisterAddress);
printf("Inizializzazione della rete....\n");
StartLink(&TempVcplinkptr);
ControlNetwork =
   Configure (FromControlLink, ToControlLink, buf,
              &NumberOfDevicesInControlNetwork,
              &ConfigurationStatus);
iopl(0);
if (ControlNetwork->LinkStatus == LINKSTATUSOKAY) {
  ControlNetwork->CurrentVirtualLink = 0 ;
  printf ("Configurazione della rete  OK \n");
  printf("La rete e' costituita da %d C104.\n",
  NumberOfDevicesInControlNetwork);
  buf[0] = HEAD_CONFIG;
  buf[1] = PRT_SUCCESS;
  tonet = htonl(NumberOfDevicesInControlNetwork);
  putbuf(&buf[2],&tonet);
  sock_write(sock, buf, 6);
} else {
  printf("Configurazione della rete NOT OK \n");
  buf[0] = HEAD_CONFIG;
  buf[1] = PRT_ERROR;
  sock_write(sock, buf, 2);
}
unlock(sem_id);
break;

/************************************/
/* reset del singolo link di un C104 */
/************************************/
case 3:
lock(sem_id);
iopl(3);
sock_read(sock, buf, 5);
putbuf(&tonet,buf);
IdSwitch = ntohl(tonet);
printf("Id. STC104: %d\n",IdSwitch);
ControlNetwork->CurrentVirtualLink = IdSwitch;
IdLink = (int) buf[4];
printf("Id. Link: %d\n",IdLink);
ResetC104link (IdLink, ControlNetwork);
if (ControlNetwork->LinkStatus == LINKSTATUSOKAY) {
  buf[0] = HEAD_RSTLINK;
  buf[1] = PRT_SUCCESS;
  sock_write(sock, buf, 2);
} else {
  buf[0] = HEAD_RSTLINK;
  buf[1] = PRT_ERROR;
  sock_write(sock, buf, 2);
```

```
 }
 iopl(0);
 unlock(sem_id);
 break;

/*****************************************************
**/
/* Comando LinkInfo: Trasmetti i bit dei registri di
*/
/* configurazione del link del C104 richiesto
*/
/*****************************************************
**/
 case 5:
 lock(sem_id);
 iopl(3);
 sock_read(sock, buf, 5);
 putbuf(&tonet,buf);
 IdSwitch = ntohl(tonet);
 printf("Id. STC104: %d\n",IdSwitch);
 ControlNetwork->CurrentVirtualLink = IdSwitch;
 IdLink = (int) buf[4];
 printf("Id. Link: %d\n",IdLink);
 ind = ((0x80 + IdLink)<< 8)+C104_LINK_STATUS;
 CPeek(ind, &stat, ControlNetwork);
 ind = ((0x80 + IdLink)<< 8)+C104_PACKET_MODE;
 CPeek(ind, &mode, ControlNetwork);
 if (ControlNetwork->LinkStatus == LINKSTATUSOKAY) {
   buf[0] = HEAD_LINKINFO;
   buf[1] = PRT_SUCCESS;
    buf[2] = (char) stat; /* Link0-31Status ha 6 bit
    */
    buf[3] = (char) mode; /* Packet0-31Mode ha 5 bit
    */
   sock_write(sock, buf, 4);
 } else {
   buf[0] = HEAD_LINKINFO;
   buf[1] = PRT_ERROR;
   sock_write(sock, buf, 2);
 }
 iopl(0);
 unlock(sem_id);
 break;

 /******************/
 /* Reset del C104 */
 /******************/
 case 6:
 lock(sem_id);
 iopl(3);
 sock_read(sock, buf, 4);
```

```
        putbuf(&tonet,buf);
        IdSwitch = ntohl(tonet);
        printf("Id. STC104: %d\n",IdSwitch);
        ControlNetwork->CurrentVirtualLink = IdSwitch;
        BacoReset(1,ControlNetwork);
        if (ControlNetwork->LinkStatus == LINKSTATUSOKAY) {
          buf[0] = HEAD_RSTSWITCH;
          buf[1] = PRT_SUCCESS;
          sock_write(sock, buf, 2);
        } else {
          buf[0] = HEAD_RSTSWITCH;
          buf[1] = PRT_ERROR;
          sock_write(sock, buf, 2);
        }
        iopl(0);
        unlock(sem_id);
        break;
      }
    }
    else {
      return -1;
    }
  } while (j!=9);
  buf[0] = HEAD_EXIT;
  buf[1] = PRT_SUCCESS;
  sock_write(sock, buf, 2);
  return PRT_SUCCESS;
}

void main () {

  int newsock,n,i,j,k,tonet,polling,gestoreClient;
  struct sigaction act, oldact;

  sigemptyset(&act.sa_mask);
  act.sa_flags=0;
  act.sa_handler = sig_chld;
  sigaction(SIGCHLD, &act, &oldact);

  if (((sock = create_sock()) != -1) &&
       ((sem_id = create_sem(seed)) != -1)){

    signal(SIGINT,Free_Resources);
    i = 0;
    LIVE = 1;
     printf("Ready to connect! \n");
     while (LIVE) {
      if ((newsock = my_accept(sock)) != -1) {
        NumOfClientsConnected+=1;
        printf("I'm %d^ forked\n",NumOfClientsConnected);
        /*****************************************/
```

```
/* Trasmetto NumOfClientsConnected al Client */
/*******************************************/
tonet = htonl(NumOfClientsConnected);
sock_write(newsock,&tonet,sizeof(tonet));
if (NumOfClientsConnected == 1) {
  do { sock_read(newsock,buf,1);
  } while ( buf[0] != HEAD_CONFIG);
  FromControlLink  = (Channel *)CONTROL_LINK ;
  ToControlLink    = (Channel *)CONTROL_LINK ;
   ControlNetwork = NULL ;
  TempVcplinkptr.ToNetwork = ToControlLink ;
  iopl(3);
  InitializeDevicesNoStart(RegisterAddress);
  printf("Inizializzazione della rete....\n");
  StartLink(&TempVcplinkptr);
  ControlNetwork =
    Configure (FromControlLink, ToControlLink, buf,
               &NumberOfDevicesInControlNetwork,
                &ConfigurationStatus);
  iopl(0);
  if (ControlNetwork->LinkStatus == LINKSTATUSOKAY) {
    ControlNetwork->CurrentVirtualLink = 0 ;
    printf ( "Configurazione della rete  OK \n");
    printf("La rete e' costituita da %d C104.\n",
    NumberOfDevicesInControlNetwork);
    buf[0] = HEAD_CONFIG;
    buf[1] = PRT_SUCCESS;
    tonet = htonl(NumberOfDevicesInControlNetwork);
    putbuf(&buf[2],&tonet);
    sock_write(newsock, buf, 6);
  } else {
    printf("Configurazione della rete NOT OK \n");
    buf[0] = HEAD_CONFIG;
    buf[1] = PRT_ERROR;
    sock_write(newsock, buf, 2);
    close(newsock);
    Free_Resources();
    exit(1);
  }
}
gestoreClient = fork();
        /*************************/
        /* Processo gestoreClient */
        /*************************/
if (gestoreClient == 0) {
  int temp;
  signal(SIGINT, SIG_DFL);
  if (NumOfClientsConnected > 1) {
    /*****************************************/
    /*              gestoreClient            */
    /* Trasmetto il numero di STC104 al Client */
```

```
            /****************************************/
            temp = htonl(NumberOfDevicesInControlNetwork);
            sock_write(newsock,&temp,4);
          }
          polling = fork();
          if (polling == -1) {
            printf("Fork Error...\n");
            LIVE = 0;
          } else if (polling == 0) {
             /*******************/
             /* Processo Polling */
             /*******************/
            int changed;
            char pollingMatTemp[3][32],pollingMatrix[3][32];
            close(sock);
            signal(SIGINT, SIG_DFL);
            while (1) {
              if (ControlNetwork != NULL) {
                lock(sem_id);
                iopl(3);
                for
(k=0;k<NumberOfDevicesInControlNetwork;k++)
                 {
                  ControlNetwork->CurrentVirtualLink = k;
                  PollingC104LinkStatus(ControlNetwork,k,
                                        &pollingMatrix[k][0]);
                 }
                changed = 0;

for(i=0;i<NumberOfDevicesInControlNetwork;i++) {
                  for(j=0;j<32;j++) {
                    if (pollingMatrix[i][j] !=
                          pollingMatTemp[i][j]) {
                      pollingMatTemp[i][j] =
pollingMatrix[i][j];
                      changed = 1;
                    }
                  }
                }
               if (ControlNetwork->LinkStatus ==
LINKSTATUSOKAY){
                if (changed) {
                int
count=(NumberOfDevicesInControlNetwork*32);
                  buf[0] = HEAD_POLLING;
                  buf[1] = PRT_SUCCESS;
                  sock_write(newsock, buf, 2);
                  sock_write(newsock,&pollingMatrix[0][0],
                             count);
                 }
               } else {
```

```
                buf[0] = HEAD_POLLING;
                buf[1] = PRT_ERROR;
                 sock_write(newsock, buf, 2);
              }
              iopl(0);
              unlock(sem_id);
              }
              sleep(1);
            }
          } /* FINE Processo Polling */

    /***********************************/
    /*          gestoreClient          */
    /* Esecuzione richieste provenienti */
    /* dal Client                      */
    /***********************************/
          n = request_handler(newsock);
          my_kill(polling);
          close(newsock);
          printf("%d client
exited..\n",NumOfClientsConnected);
          return;
        }
      }
    }
    close(sock);
    dispose_sem(sem_id);
  }
}
```

Si riporta di seguito l'insieme di funzioni, memorizzate nel file **sem-sock.c**, per la gestione dei semafori e dei *socket*.

```
#include "sem-sock.h"

struct sembuf lock_flg={0, -1, 0}; /* Decrementa di 1 il
                                      valore associato al semaforo
                                      */
struct sembuf unlock_flg={0, 1, 0}; /* Incrementa di 1 il
                                      valore associato al semaforo
                                      */
union semun initialize;

struct sockaddr_in server,client;
struct sockaddr *name;

              /*******************************/
              /*   -- SEMAPHORE SECTION --   */
```

```
              /*******************************/

int create_sem (char c) {
  int sd,i;
  key_t key;
  if ((key=ftok(".",c)) == -1) {
    perror("ftok");
    return -1;
  }
/*************************************************************
/
/* La funzione semget crea(IPC_CREAT) con successo un nuovo
*/
/* (IPC_EXCL) semaforo e ne restituisce il descrittore (sd)
*/
/*************************************************************
/
  if ((sd = semget(key,1,IPC_CREAT|IPC_EXCL|0660)) == -1) {
    perror("semget");
    return -1;
  }
  initialize.val = 1;
  if ( (i = semctl(sd,0,SETVAL,initialize)) == -1) {
    perror("semctl");
    return -1;
  }
  return sd;
}



/* Attesa passiva se la risorsa non e' disponibile */
void lock (int sem) {
  int i;

  if ( (i = semop(sem,&lock_flg,1)) == -1) {
    perror("semop");
    exit(1);
  }
}

void unlock (int sem) {
  int i;

  if ( (i = semop(sem,&unlock_flg,1)) == -1) {
    perror("semop");
    exit(1);
  }
}
```

```
void dispose_sem (int sem) {
  int i;

  if ( (i = semctl(sem,0,IPC_RMID,0))== -1) {
    perror("semctl");
    exit(1);
  };
}

                /****************************/
                /*   -- SOCKET SECTION --   */
                /****************************/

int create_sock() {
  int sock,n,bi,i;
  if ((sock = socket(AF_INET,SOCK_STREAM, 0)) == -1) {
    perror("socket");
    return -1;
  }
  bzero ((char *) &server, sizeof(server));
  server.sin_family = AF_INET;
  server.sin_addr.s_addr = htonl(INADDR_ANY);
  server.sin_port = htons (6884);
  name = (struct sockaddr *) &server;
  if ((bi = bind(sock,name, sizeof(*name))) == -1) {
    perror("bind");
    if ((i = close(sock))== -1){
      perror("close");
    }
    return -1;
  } else {
    printf(" bind OK \n");
    printf ("Nr. descrittore di socket %d \n", sock);
    if ((i = listen (sock, 1)) == -1){
      perror("listen");
      return -1;
    }
    return sock;
  }
}

int my_accept(int sock) {
  int cli_len, newsock;
  cli_len = sizeof(client);
  if ((newsock = accept(sock, (struct sockaddr *)&client,
                        &cli_len)) == -1) {
    if (errno != EINTR) {
      perror("accept");
    }
  }
  return newsock;
```

```
}

int sock_read (int sockfd, char *buffer, int count) {
  int bytes_read = 0;
  int temp_read;

  while (bytes_read < count) {
    temp_read = read (sockfd, buffer, count - bytes_read);
    if (temp_read < 0)
      /* Error */
      return temp_read;
    bytes_read += temp_read;
    buffer += temp_read;
  }
  return count;
}

int sock_write (int sockfd, char *buffer, int count) {
  int bytes_sent = 0;
  int temp_write;

  while (bytes_sent < count) {
    temp_write = write (sockfd, buffer, count - bytes_sent);
    if (temp_write < 0)
      /* Error */
      return temp_write;
    bytes_sent += temp_write;
    buffer += temp_write;
  }
  return count;
}

void putbuf (char *dest, char *src) { /* Copia i 4 byte che
              compongono un intero in un vettore di byte */
  int i;
  for (i=0;i<4;i++) {
    *dest = *src;
    dest++;
    src++;
  }
}

void my_kill ( int prn ) { /* Terminazione forzata di un
                              processo */
  int i;

  if ((i = kill(prn,SIGKILL)) == -1) {
    perror("kill");
  }
}
```

Appendice A.2: Le classi Java dello *applet*

In questa appendice, per questioni di brevità, verranno illustrate le classi RV, STC104 e DataLink che modellano rispettivamente la connessione TCP con l'agente di controllo, lo switch ed il *data link*.

Di seguito si mostrano le righe di codice della classe **RV**:

```
import java.net.*;
import java.io.*;
import java.util.*;

public class RV implements Runnable{
  int port = 6884;

  /** Application protocol Commands **/
  private byte Head_Configure = 1;
  private byte Head_RstLink = 3;
  private byte Head_Polling = 4;
  private byte Head_LinkInfo = 5;
  private byte Head_RstSwitch = 6;
  private byte Head_EXIT = 9;
  private byte ERROR = 1;
  private byte SUCCESS = 9;

  int rx,tx;
  String host;
  globalPanel gP;
  Socket sock;
  DataInputStream is;
  DataOutputStream os;
  boolean OK = false;
  Thread ear = null;
  int numConfSw;
```

```
public RV (globalPanel gP) { /* Costruttore */
  this.gP = gP;
  OK = false;
  numConfSw = 0;
}

public void parti() {
  int i;
  try{
      host = gP.getCodeBase().getHost();
      sock = new Socket(host,port);
      is = new DataInputStream(sock.getInputStream());
      os = new DataOutputStream (sock.getOutputStream());
      gP.printMessage("Sei connesso con : "+host);
      gP.connected = true;
      gP.open.forceOff();
      OK = true;
     }
  catch(IOException e){
    gP.printMessage("Socket Error: "+e.getMessage());
    gP.printMessage("The Server may be down...");
    this.stop();
  }
  System.out.println("OK = "+OK);
  System.out.println("Connected = "+gP.connected);
  if (OK) {
    System.out.println("Prima di readInt in RV.class");
    try {
      rx = is.readInt();
      System.out.println("You are the" + rx +"^ Client");
      if (rx > 1) {
        /* Ricevi Numero di Switch */
        i = is.readInt();
        this.numConfSw = i;
        gP.numConfSw = i;
        gP.setConfiguration();
        gP.config.disable();
        gP.printMessage("Somebody is using the Network.");
        gP.printMessage("Plase wait, the Switch status");
        gP.printMessage("is reaching you....");
        this.startPolling();
      }
    } catch (EOFException e) {
        gP.printMessage("END REACHED: "+e.getMessage());
        this.stop();
    }
    catch (IOException e) {
        gP.printMessage("ERROR: "+e.getMessage());
        this.stop();
    } catch (NullPointerException e) {
```

```
          gP.printMessage("You `d better reload the
Applet...");
          this.stop();
      }
    } else stop();
  }

  public void startPolling() {
    if (ear != null) {
      try {
        ear.stop();
        ear = null;
      } catch (SecurityException e) {
        System.out.println("Thread stopping Error");
        gP.printMessage(e.getMessage());
      }
    }
    ear = new Thread(this);
    ear.start();
  }

  public void stop() {
    if (gP.connected) {
      gP.connected = false;
      gP.configured = false;
      gP.open.forceOff();
      gP.config.enable();
    }

    if (is != null) {
      try {
        is.close();
        is = null;
        gP.printMessage("Input Stream chiuso");
      } catch (IOException e) {
        gP.printMessage("Closing Input I/O Error:");
        gP.printMessage(e.getMessage());
      }
    }
    if (os != null) {
      try {
        os.close();
        os = null;
        gP.printMessage("Output Stream chiuso");
      }
      catch (IOException e) {
        gP.printMessage("Closing Output I/O Error:");
        gP.printMessage(e.getMessage());
      }
    }
    if (sock != null) {
```

```
      try {
        sock.close();
        gP.printMessage("Socket chiuso");
        sock = null;
      }
      catch (IOException e) {
        gP.printMessage("Socket closing Error:");
        gP.printMessage(e.getMessage());
      }
    }
    if (ear != null) {
      try {
        ear.stop();
        ear = null;
        System.out.println("Thread stoppato");
      } catch (SecurityException e) {
        System.out.println("Thread stopping Error");
        gP.printMessage(e.getMessage());
      }
    }
  }

  private void refreshPanel() {
    byte [][] buf = new
byte[numConfSw][STC104.numOfDataLink];
    gP.printMessage("Updating Switch...");
    for (int j=0;j<numConfSw;j++) {
      try {
        is.readFully(buf[j]);
        gP.fullNetwork[j].refreshLinks(buf[j]);
      } catch (IOException e) {
        gP.printMessage(e.getMessage());
        this.stop();
        return;
      }
    }
    gP.printMessage("Done!");
  }

  public synchronized int configNetwork() {
    byte [] buf = new byte[2];
    int i = -1;
    boolean again = true;
    /* Invio dati di richiesta */
    try {
      os.writeByte(Head_Configure);
    } catch (IOException e) {
      gP.printMessage(e.getMessage());
      stop();
    }
    while(again) {
```

```
        /* Ricezione Risposta */
      try {
        is.readFully(buf, 0, 2);
      } catch (IOException e) {
        gP.printMessage(e.getMessage());
        again = false;
        stop();
      }
      if (buf[0] == Head_Configure) {
        if (buf[1] == SUCCESS) {
          /* Ricevi Numero di Switch */
          try {
            i = is.readInt();
            this.numConfSw = i;
            if (gP.configured == false) {
              this.startPolling();
            }
          } catch (IOException e) {
            gP.printMessage(e.getMessage());
            again = false;
            stop();
          }
        } else gP.printMessage("Error Receiving
Configuration");
        again = false;
      } else if (buf[0] == Head_Polling) {
        if (buf[1] == SUCCESS) {
      /* Aggiorna Schermo e torna a Ricevere
         (N° STC104 o un altro Polling)*/
          this.refreshPanel();
        } else {
          gP.printMessage("Error during Polling");
          again = false;
        this.stop();
        }
      }
    }
    return i;
  }

  public synchronized void resetLink(int numSwitch,int index)
{
    byte [] buf = new byte[2];
    boolean again = true;
    /* Invio dati di richiesta*/
    try {
      os.writeByte(Head_RstLink);
      os.writeInt(numSwitch);
      os.writeByte(index);
    } catch (IOException e) {
      gP.printMessage(e.getMessage());
```

```
    stop();
    return;
  }
  while (again) {
    /* Ricezione risposta */
    try {
      is.readFully(buf, 0, 2);
    } catch (IOException e) {
      gP.printMessage(e.getMessage());
      again = false;
      stop();
      return;
    }
    if (buf[0] == Head_RstLink) {
      if (buf[1] == SUCCESS) {
        gP.printMessage("Reset Link "+index+" Switch " +
                        numSwitch + " OK");
      } else gP.printMessage("Error during Reset Link");
      again = false;
    } else if (buf[0] == Head_Polling) {
      if (buf[1] == SUCCESS) {
      /* Aggiorna Schermo e torna a Ricevere
         (Risposta-Reset o un altro Polling) */
        this.refreshPanel();
      } else {
        gP.printMessage("Error during Polling");
        again = false;
        this.stop();
        return;
      }
    }
  }
}


public synchronized void askLinkInfo(int numSwitch,int
index) {
  byte [] buf = new byte[2];
  int i;
  boolean again = true;
  /* Invio dati di richiesta */
  try {
    os.writeByte(Head_LinkInfo);
    os.writeInt(numSwitch);
    os.writeByte(index);
  } catch (IOException e) {
    gP.printMessage(e.getMessage());
    stop();
    return;
  }
  while(again) {
```

```
      /* Ricezione Risposta */
    try {
      is.readFully(buf, 0, 2);
    } catch (IOException e) {
      gP.printMessage(e.getMessage());
      again = false;
      stop();
      return;
    }
    if (buf[0] == Head_LinkInfo) {
      if (buf[1] == SUCCESS) {
        /* Ricevi Dati-Registri, Mostra Tabella e esci */
        try {
          is.readFully(buf, 0, 2);
        } catch (IOException e) {
          gP.printMessage(e.getMessage());
          again = false;
          stop();
          return;
        }
        (new InfoFrame(gP,buf,"Switch "+numSwitch+" – “+
                    "Link"+index+"Registers")).show();
      } else gP.printMessage("Error asking LinkInfo");
      again = false;
    } else if (buf[0] == Head_Polling) {
      if (buf[1] == SUCCESS) {
     /* Aggiorna Schermo e torna a Ricevere
        (Risposta-LinkInfo o un altro Polling) */
        this.refreshPanel();
      } else {
        gP.printMessage("Error during Polling");
        again = false;
        this.stop();
        return;
      }
    }
  }
}

public synchronized void resetSwitch(int numSwitch) {
  byte [] buf = new byte[2];
  boolean again = true;
  /* Invio dati di richiesta */
  try {
    os.writeByte(Head_RstSwitch);
    os.writeInt(numSwitch);
  } catch (IOException e) {
    gP.printMessage(e.getMessage());
    stop();
    return;
  }
```

```
  while (again) {
    /* Ricezione risposta */
    try {
      is.readFully(buf, 0, 2);
    } catch (IOException e) {
      gP.printMessage(e.getMessage());
      again = false;
      stop();
      return;
    }
    if (buf[0] == Head_RstSwitch) {
      if (buf[1] == SUCCESS) {
        gP.printMessage("Reset Switch "+numSwitch +" OK");
      } else gP.printMessage("Error during Reset Switch");
      again = false;
    } else if (buf[0] == Head_Polling) {
      if (buf[1] == SUCCESS) {
       /* Aggiorna Schermo e torna a Ricevere
         (Risposta-Reset o un altro Polling) */
        this.refreshPanel();
      } else {
        gP.printMessage("Error during Polling");
        again = false;
        this.stop();
        return;
      }
    }
  }
}

public synchronized void sendExit() {
  byte [] buf = new byte[2];
  boolean again = true;
  /* Invio dati di richiesta */
  try {
    os.writeByte(Head_EXIT);
  } catch (IOException e) {
    gP.printMessage(e.getMessage());
    stop();
    return;
  }
  while (again) {
    /* Ricezione risposta */
    try {
      is.readFully(buf, 0, 2);
    } catch (IOException e) {
      gP.printMessage(e.getMessage());
      again = false;
      stop();
      return;
    }
```

```
    if (buf[0] == Head_EXIT) {
      if (buf[1] == SUCCESS) {
        gP.printMessage("Disconnecting....");
        stop();
      } else gP.printMessage("Error during Disconnection");
      again = false;
    } else if (buf[0] == Head_Polling) {
      if (buf[1] == SUCCESS) {
       /* Aggiorna Schermo e torna a Ricevere
          (Risposta-Exit o un altro Polling) */
        this.refreshPanel();
      } else {
        gP.printMessage("Error during Polling");
        again = false;
        stop();
        return;
      }
    }
  }
}

public synchronized void get_Polling() {
  int i;
  byte [] buf = new byte[2];
  try {
    if (is.available() > 0) {
      /* Ricezione risposta */
      try {
        is.readFully(buf, 0, 2);
      } catch (IOException e) {
        gP.printMessage(e.getMessage());
        stop();
        return;
      }
      if (buf[0] == Head_Polling) {
        if (buf[1] == SUCCESS) {
         /* Aggiorna Schermo */
          this.refreshPanel();
        } else {
          gP.printMessage("Error during Polling");
          this.stop();
          return;
        }
      }
    }
  } catch (IOException e) {
      stop();
  } catch (NullPointerException e) {
      stop();
  }
}
```

```
  public void run() {
    for(;;){
      get_Polling();
      try {Thread.sleep(2000);}
      catch (InterruptedException e){};
    }
  }
}
```

Di seguito si mostrano le righe di codice della classe **STC104**:

```
public class STC104 {
  int switchIndex;
  static final int numOfDataLink = 32;
  static final int numOfControlLink = 2;
  DataLink[] DataGrid = new DataLink[numOfDataLink];
  ControlLink[] ControlGrid = new
ControlLink[numOfControlLink];
  SwReset reset;
  globalPanel gP;
  Swlab myLabel;

  public STC104 (int switchIndex, globalPanel gP) {
    this.switchIndex = switchIndex;
    this.gP = gP;
    for (int i=0;i<numOfDataLink;i++) {
      DataGrid[i] = new DataLink(switchIndex,i,gP);
    }
    for (int i=0;i<numOfControlLink;i++) {
      ControlGrid[i] = new ControlLink(switchIndex,i,gP);
    }
    reset = new SwReset(switchIndex,gP);
    myLabel = new Swlab(switchIndex,gP);
  }

/* Aggiorna il colore di sfondo dei link a seguito di una
variazione dei valori del registro LinkStatus */
  public void refreshLinks (byte[] val) {
    for (int i=0;i<numOfDataLink;i++) {
      DataGrid[i].update(DataGrid[i].getGraphics(),val[i]);
    }
  }
```

```
/* Disabilita le funzionalità dello switch */
  public void disable () {
    for (int i=0;i<numOfDataLink;i++) {
      DataGrid[i].disable();
    }
    for (int i=0;i<numOfControlLink;i++) {
      ControlGrid[i].disable();
    }
    reset.disable();
  }
}
```

Di seguito si mostrano le righe di codice della classe **DataLink**:

```
import java.awt.*;
public class DataLink extends Canvas {
  int linkIndex, bg,switchIndex;
  Dimension d;
  globalPanel gP;
  String lbl;
  Color Connected = new Color(120,255,0); /* Link Connected
*/
  Color Error = new Color(255,0,0); /* Link Error */
  Color Started = new Color(0,150,255); /* Link Started */

  public DataLink(int switchIndex,int linkIndex,globalPanel
gP){
    this.switchIndex = switchIndex;
    this.linkIndex = linkIndex;
    this.gP = gP;
    bg = 0;
  }

  public int get_index () {
    return linkIndex;
  }

  public Dimension preferredSize() {
    return new Dimension(100,15);
  }

  /* Rappresentazione grafica del data link */
  public synchronized void paint(Graphics g) {
    d = size();
    g.setColor(Color.black);
    g.drawRect(0,0,d.width-1,d.height);
    switch (bg) {
      case 0:
        g.setColor(Color.white);
```

```
        break;
      case 34:
        g.setColor(Connected);
        break;
      case 2:
        g.setColor(Started);
        break;
      default:
        g.setColor(Error);
        break;
    }
    g.fillRect(1,1,d.width-2,d.height-1);
    g.setColor(Color.black);
    g.drawLine(d.width-20,1,d.width-20,d.height-1);
    g.setColor(Color.yellow);
    g.fillRect(d.width-19,1,d.width-3,d.height-1);
    g.setColor(Color.black);
    Font f = gP.getFont();
    FontMetrics fm = gP.getFontMetrics(f);
    g.setFont(f);
    lbl = "Link  "+String.valueOf(this.get_index());

g.drawString(lbl,3,(int)((d.height/2)+fm.getDescent()+1));
    g.drawString("R",d.width-15,
                 (int)((d.height/2)+fm.getDescent()+1));
  }

  public synchronized void update (Graphics g,int i) {
    bg = i;
    paint(g);
  }

  public boolean mouseDown(Event evt,int x,int y) {

    if (gP.configured) {
      if (x<d.width-20) { /* Richiesta lettura dei registri
*/
        gP.printMessage("Asking Info on
Link"+this.get_index());
        gP.rv.askLinkInfo(switchIndex,this.get_index());
      }  else { /* Richiesta reset del link */
        gP.printMessage("Asking Reset on Link"+
                        this.get_index());
        gP.rv.resetLink(switchIndex,this.get_index());
      }
    } else {
      gP.printMessage("Configure first...");
      System.out.println("configured = "+gP.configured);
    }
    return true;
  }
```

```
}
```

Bibliografia

[AG96] K. Arnold, J. Gosling. “The Java Programming Language”, Addison-Wesley.

[ATL94] ATLAS Collaboration. “ATLAS Technical Proposal for a General Purpose p-p Experiment at the Large Hadron Collider at CERN”.

[BG92] D. Bertsekas, R. Gallager. “Data Networks”, Prentice Hall.

[CQ96] M. Caprini, Z. Qian. “Java language evaluation for DAQ status display”, CERN - Nota Tecnica 010. |http://atddoc.cern.ch/Atlas/Notes/010/Note010-1.html|

[CS94] D.E. Comer, D.L. Stevens. “Internetworking with TCP/IP” - Vol. III, Prentice Hall.

[Dad96] R. Dadda. “Java e i browser Java”, Jackson.

[Flan96] D. Flanagan. “Java in a Nutshell”, O’Reilly & Associates Inc.

[GVdM+95] S. Goldt, S. Van der Meer, S. Burkett, M. Welsh. ”The LINUX Programmer’s Guide”. |http://linosa.caspur.it/htmldocs|

[GJS96] J. Gosling, B. Joy, G. Steele. “The Java Language Specification”, Addison-Wesley.

[GY+96] J. Gosling, F. Yellin, The Java team. “The Java Application Programming Interface” – Vol. I e II, Addison-Wesley.

[KR89] B.W. Kernighan, D.M. Ritchie. “Linguaggio C”, Jackson.

[LY96] T. Lindholm, F. Yellin. “The Java Virtual Machine Specification”, Addison-Wesley.

[Mar94] J. Martin. “TCP/IP Networking: Architecture, Administration, and Programming”, Prentice Hall.

[MTW93] M.D. May, P.W. Thompson, P.H. Welch. “Networks, Routers and Transputers”, SGS-Thomson.

[Per87] D.H. Perkins. “Introduction to High Energy Physics”, Addison-Wesley.

[Pos81] J. Postel. “Transmission Control Protocol”, *Request For Comments* 793.

[RB+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. “Object-Oriented Modeling and Design”, Prentice Hall.

[Rus98] D.A. Rusling. “The LINUX Kernel”.

[Sar97] L.F.G. Sarmenta. “Bayanihan: Web-Based Volunteer Computing using Java”, MIT Laboratory for Computer Science. |http://www.cag.lcs.mit.edu/bayanihan|

[SGS94a] SGS-Thomson. “PC HTRAM Motherboard – IMS B108”.

[SGS94b] SGS-Thomson. “T9000 Toolset Hardware Configuration Manual”.

[SGS95a] SGS-Thomson. “STC104 Asynchronous Packet Switch – Engineering Data”.

[SGS95b] SGS-Thomson. “DS-Link Evaluation Kit Installation and User Guide”.

[SGS97] SGS-Thomson. “STC101 Parallel DSLink Adaptor”.

[Ste90] W.R. Stevens. “UNIX Network Programming”, Prentice Hall.

[Str97] C.W. Strevens. “An Introduction to IEEE 1355”, |http://www.1355-association.org|

[Tan88] A.S. Tanenbaum. “Progettazione e sviluppo dei Sistemi Operativi” – Edizione Italiana, Jackson.

[Tan95] A.S. Tanenbaum. “Reti di Computer” – Seconda edizione, Prentice Hall.

[vHSS96] A. van Hoff, S. Shaio, O. Starbuck. “Hooked on Java”, Addison-Wesley.