View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by CERN Document Server

UNIVERSITÀ DEGLI STUDI DI ROMA
"LA SAPIENZA"

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

TESI DI LAUREA IN FISICA

Controllo e prestazioni di reti a commutazione di pacchetto nell'ambito di studi per il trigger di 2° livello dell'esperimento ATLAS

Relatore interno:
Prof. Lucia Zanello

Relatore esterno:
Dr. Speranza Falciano

Laureando: Roberto Cesari
Matr. 11091891

Anno Accademico 1997-1998

TESINE

Misura della posizione di elementi ottici in rivelatori gravitazionali interferometrici.

Relatore: Dott. P. Rapagnani

Calcolatori quantistici.

Relatore: Prof. G. Diambrini-Palazzi

A mia madre

INTRODUZIONE

Nonostante le numerose conferme sperimentali fino ad oggi ottenute, manca ancora una completa verifica del Modello Standard. In esso è prevista infatti l'esistenza del bosone di Higgs (con massa inferiore al Tev) connesso alla rottura spontanea di simmetria. Nel nuovo anello di accumulazione per fasci di protoni che è in costruzione al CERN, il Large Hadron Collider, si potrà raggiungere un'energia nel centro di massa che potrebbe permettere la scoperta del bosone di Higgs o suggerire meccanismi diversi da questo. Nella prima ipotesi LHC fornirà prove sperimentali valide per l'ultimo elemento mancante alla verifica completa del Modello Standard.

Attraverso LHC sarà possibile inoltre ottenere un'alta produzione di coppie $b\bar{b}$ e $t\bar{t}$ attraverso cui si potranno investigare fenomeni fisici quali la violazione di CP, ed avviare una comprensione più approfondita della fisica al di là del Modello Standard.

ATLAS rappresenta uno dei principali esperimenti in fase di realizzazione ad LHC; grazie ad uno spettrometro per muoni di elevate prestazioni, nonché un efficiente sistema di trigger, sarà possibile rilevare leptoni altamente energetici che rappresentano una segnatura fondamentale per l'individuazione del bosone di Higgs.

Il sistema di trigger sarà necessario per operare una selezione sugli eventi interessanti prodotti nelle collisioni protone-protone, attraverso una ricostruzione delle tracce e depositi di energia all'interno dei calorimetri. Il flusso di informazione derivante da questi processi di selezione è tale che l'intero sistema di trigger sarà strutturato in tre livelli, ognuno dei quali opererà su diversi volumi di dati: su questi verranno applicati degli algoritmi per la ricostruzione degli eventi, la cui complessità crescerà con il passaggio a livelli di trigger sempre più alti.

In particolare per il 2° livello sono state proposte delle architetture nelle quali l'informazione proveniente dal 1° livello dovrà essere trasferita, attraverso una o più reti di interconnessione, a dei processori (chiamati *Global Processor*) che applicheranno opportuni algoritmi per rigettare od accettare l'evento; questo meccanismo dovrà introdurre latenze non superiori ai 10 ms. Con tali valori si può intuire come la rete d'interconnessione debba essere estremamente efficiente, e garantire minimi ritardi nello instradamento dell'informazione da sorgenti a destinazioni (*Global Processor*).

Il lavoro di tesi è stato mirato a riprodurre in laboratorio piccole reti d'interconnessione, con una tecnologia commerciale già indicata da alcuni progetti pilota [13] nell'ambito di collaborazioni all'esperimento ATLAS, per valutarne prestazioni e capacità di distribuzione del traffico di dati. Questa tecnologia denominata DS-Link ed utilizzata principalmente per progetti di calcolo parallelo, segue lo standard di comunicazione seriale IEEE 1355.

L'organizzazione del lavoro è la seguente.

Il primo capitolo è dedicato ad una descrizione dell'esperimento ATLAS e del rivelatore, introducendo poi le attuali proposte di architetture del 2° livello.

Nel secondo si fornirà una descrizione dello standard IEEE 1355 e della sua implementazione nella tecnologia DS-Link, illustrando tra l'altro i dispositivi utilizzati nei diversi test di misura.

Nel terzo capitolo verranno descritte delle possibili topologie di reti di interconnessione, fornendo una trattazione teorica per un modello che possa essere utilizzato nella determinazione del traffico nella rete.

Con il quarto capitolo s'illustrerà il lavoro svolto per il controllo delle reti d'interconnessione.

Infine nel quinto capitolo verranno esaminate le diverse misure effettuate, le prestazioni ottenute e le possibili conferme sperimentali ai modelli di traffico ipotizzati.

Capitolo 1

L' esperimento ATLAS ed il sistema di trigger di livello 2

1.1 Introduzione

Il Large Hadron Collider (LHC) è il nuovo anello per collisioni protone-protone che sarà costruito entro il 2004, sfruttando il tunnel dell'esistente collider LEP (Large Electron-Positron) realizzato al CERN.

Il futuro collider LHC sarà costituito da due anelli paralleli di raggio medio 4.2 Km, percorsi ognuno da un fascio di protoni confinati attraverso un campo magnetico dipolare di 8.5 T.

Tali valori di campo saranno ottenuti con dei magneti superconduttori dipolari, titanio-niobio, immersi in un sistema criogenico contenente He liquido che garantirà temperature attorno ai 2°K: è stimato che il solo consumo del sistema di raffreddamento sarà di circa 140 kW.

L'energia del centro di massa sarà di $\sqrt{s}$ = 14 TeV e la luminosità avrà un valore iniziale di $10^{33} cm^{-2} s^{-1}$ (bassa luminosità), per arrivare poi a $10^{34} cm^{-2} s^{-1}$ (alta luminosità).

L'attuale acceleratore del CERN, costituito da un *linac* da 50 MeV, un *booster* da 1 GeV, un *ProtoSincrotrone* da 26 GeV ed un *SuperProtoSincrotone* da 450 GeV, costituirà il complesso (vedi figura 1.1) per l'accelerazione dei protoni fino alla energia di iniezione di LHC. I protoni saranno immessi nell'anello principale in pacchetti (*bunches*) composti da circa 10^{11} particelle ciascuno. Nella macchina circoleranno simultaneamente 3600 pacchetti con una frequenza di rivoluzione pari a 1.1 $10^{4} Hz$: si otterrà così un incrocio o " *bunch-crossing*" ogni 25 ns.

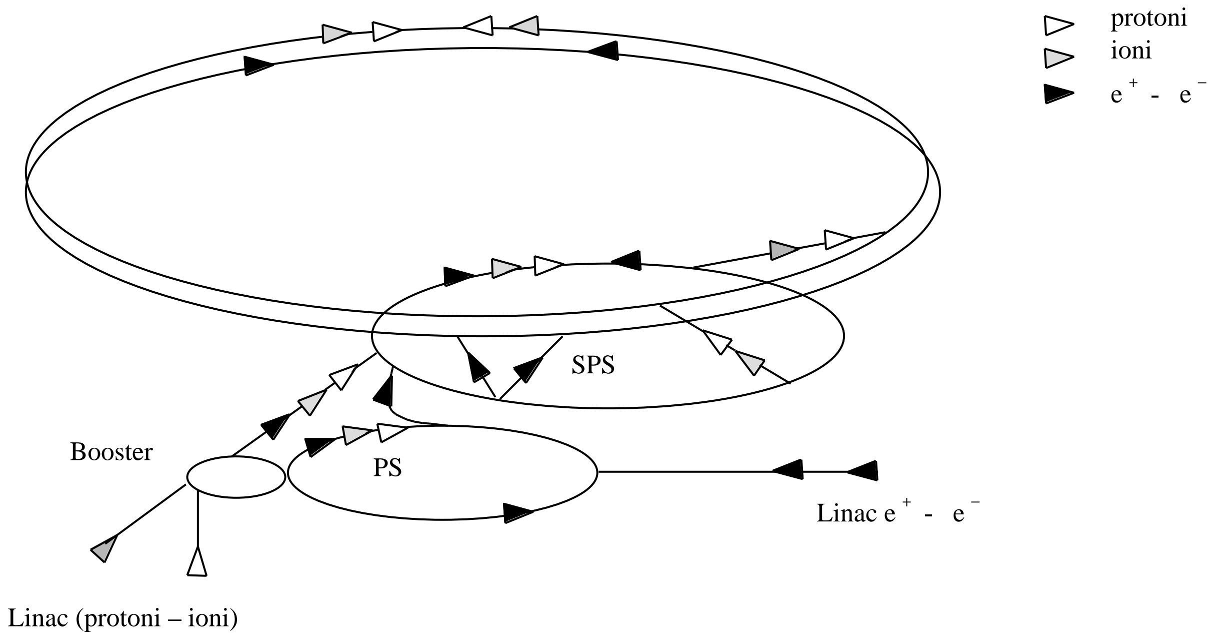


Figura 1.1: Struttura del futuro collider LHC con i suoi elementi principali.

1.2 La fisica ad LHC

Le alte energie raggiungibili con il nuovo collider offriranno un vasto spettro di possibili ricerche fisiche; in particolare la scoperta del bosone di Higgs costituirebbe la conferma dell'ultimo tassello mancante nelle previsioni del Modello Standard. La comprensione attuale delle particelle ed i fenomeni di interazione tra esse, sono infatti contemplati nel Modello Standard. Esso descrive la teoria quantistica delle interazioni forti QCD (cromodinamica quantistica) e la teoria unificata delle interazioni deboli ed elettromagnetiche (elettrodeboli): la gravità non è contemplata nel Modello Standard.

I fermioni secondo il Modello Standard sono strutturati in due gruppi: *leptoni* e *quark*. I *leptoni* interagiscono con forze elettromagnetiche e deboli e sono raggruppati in 3 famiglie, chiamate anche generazioni, che sono:

$$e \qquad \mu$$

$$_{e} \qquad _{\mu}$$

I *quark*, che possono interagire sia mediante forze deboli, elettromagnetiche o forti, sono anch'essi raggruppati in 3 famiglie; a differenza dei *leptoni* essi non esistono come particelle libere ma sono i costituenti degli adroni (*mesoni* e *barioni*).

Le 3 famiglie di *quark* sono:

u	c	t
d	s	b

Tutte e tre le forze con cui interagiscono i fermioni sono trasmesse attraverso uno o più bosoni: $W^{\pm}$ e Z^0 (anche detti bosoni vettoriali) che mediano le interazioni elettrodeboli, i *fotoni* che mediano le interazioni elettromagnetiche ed i *gluoni* le interazioni forti. Nel tentativo di incorporare i campi elettromagnetici e deboli in una unica teoria, *elettrodebole* per l'appunto, sono state riscontrate inizialmente grosse difficoltà, poiché la "*underlying theory*" di gauge prevede che tutti i bosoni della teoria unificata abbiano massa nulla: non soltanto il fotone ma anche i W e lo Z^0. Si è usciti da questa difficoltà mediante la teoria della "*rottura spontanea di simmetria*", attraverso cui si riesce a mantenere uguale a zero la massa del fotone ma a dare massa ai W ed allo Z^0. Tale meccanismo richiede l'introduzione di un ulteriore campo scalare (campo di Higgs) che porta all' esistenza di una nuova particella, il bosone di Higgs. Quindi l'esistenza del bosone di Higgs (ancora sperimentalmente non dimostrata) è un punto cruciale per la autoconsistenza dell' intero Modello Standard.

I bosoni di Higgs potrebbero essere anche più di uno: se ad esempio si considera il MSSM (Minimal Supersymmetric Standard Model), un'estensione del Modello Standard, allora il singolo bosone di Higgs verrebbe rimpiazzato da un insieme di 5 bosoni: $H^{\pm}, h, H^0, A$.

Circa il valore da attribuire alla massa del bosone di Higgs, non sussiste una relazione precisa ma si suppone solamente che debba rientrare in un intervallo compreso tra:

$$80 \text{ GeV} < m_H < 1 \text{ TeV}$$

Altri campi di interesse per la fisica che possono essere esplorati ad LHC sono:

- **Top quark**

Già dai primi anni di funzionamento del collider a bassa luminosità, sarà possibile osservare un alto numero di coppie $t\bar{t}$ ($\sim 10^7$) fornendo così una migliore accuratezza, circa ± 2GeV, alla misura della massa di $m_t \sim 170$ GeV.

- **Fisica dei B**

Un importante campo di studio verrà offerto dal decadimento dei mesoni B neutri. Il sistema $B^0 - \overline{B^0}$ è l' unico altro sistema, oltre quello $K^0 - \overline{K^0}$, in cui è possibile studiare la violazione di CP.

- **Particelle supersimmetriche**

È prevista nell' estensione supersimmetrica del Modello Standard, una grande varietà di nuove particelle (partners supersimmetrici) con masse comprese in un range tra 1 e 4 TeV. Eventi con alti valori di energia trasversa mancante E_T^{miss} permetteranno ricerche per stabilire la presenza di queste nuove particelle.

- **Fisica oltre il Modello Standard**

Con le energie in gioco ad LHC sarà possibile osservare l'esistenza, o meno, dei bosoni di gauge pesanti W' e Z' con masse comprese tra 5 e 6 TeV, contemplati nel modello di supersimmetria SUSY.

1.3 Il rivelatore ATLAS ed i suoi sottosistemi

ATLAS è uno dei due principali esperimenti in corso di realizzazione ad LHC. Il rivelatore dell'esperimento è progettato per garantire una profonda capacità d'indagine, in un'ampia gamma di processi fisici nei diversi regimi di luminosità a cui opererà LHC (sia alta che bassa): quindi individuare con buona efficienza le segnature di processi interessanti rigettando, in modo altrettanto efficiente, il livello di fondo. Per ottenere ciò il rivelatore dovrà misurare con precisione l'energia dei fotoni, elettroni e muoni e jet, nonché l'energia trasversa mancante E_T^{miss}. In figura 1.2 è riportata una visione tridimensionale del rivelatore dell' esperimento ATLAS.

I criteri guida nella costruzione del rivelatore sono :

- Un ottimo calorimetro elettromagnetico che dovrà essere in grado di identificare e misurare elettroni e fotoni e, grazie ad un elevata ermeticità, permettere misure di energia trasversa mancante.
- Efficiente sistema di tracciamento centrale ad alta luminosità per la misura del momento dei leptoni, e per una buona discriminazione tra fotoni ed elettroni.

- Uno spettrometro di muoni che ne misurerà il momento sia ad alta che a bassa luminosità.

Per la descrizione del rivelatore è stato scelto un sistema di riferimento in cui l'asse z risulta coassiale all' asse di simmetria della macchina e l' angolo polare, indicato con , è quello formato dalla direzione delle particelle emesse rispetto al fascio principale. Spesso invece dell'angolo si preferisce impiegare la pseudorapidità così definita:

$$= -\ln tg \frac{}{2}$$

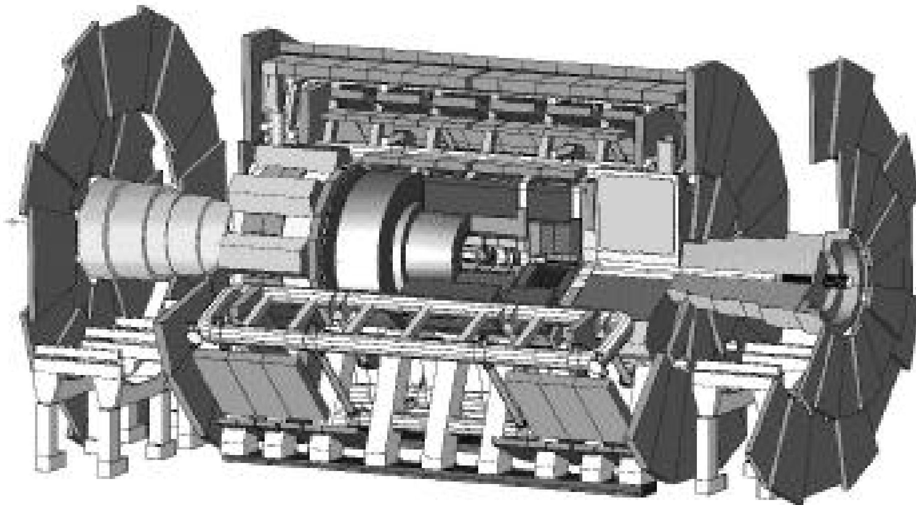


Figura 1.2: Spaccato tridimensionale dell' intero rivelatore ATLAS.

1.3.1 Rivelatore interno

Esso è collocato nella cavità cilindrica più interna ed è limitato ai bordi dai calorimetri elettromagnetici. Il rivelatore interno è disposto assialmente ad un solenoide superconduttore che fornisce un campo di 2T integrato nel criostato del calorimetro. Meccanicamente il rivelatore interno è diviso in tre parti: una prima sezione lunga circa 80 cm (chiamata *barrel region*) che si estende lungo l' asse del fascio e due ulteriori sezioni che si estendono per tutta la lunghezza del rivelatore interno; nella figura 1.3 si osserva una vista in sezione dell' insieme.

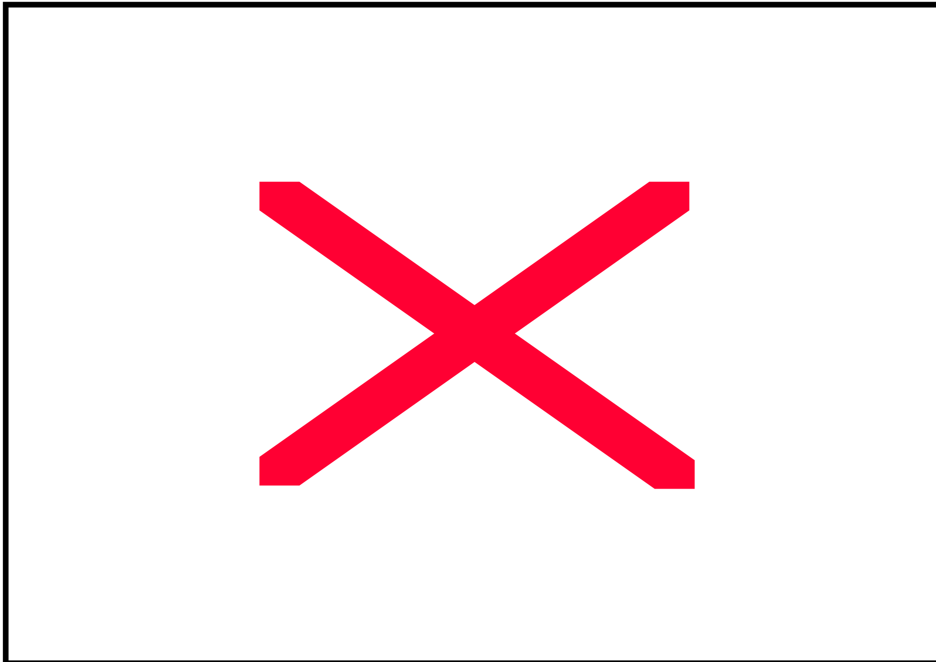


Figura 1.3: Vista tridimensionale dello Inner Detector con l' insieme dei sub-rivelatori.

Nella prima sezione i rivelatori sono disposti in cilindri concentrici e coassiali all' asse del fascio, mentre nelle altre due sezioni sono posti in direzione perpendicolare al fascio. Le diverse tecnologie impiegate per i rivelatori sono:

- **Pixel detector** utilizzati nella parte più vicina al fascio e basati su rivelatori al silicio. Essi sono situati nei primi due strati della sezione *barrel* e forniscono informazioni in due dimensioni spaziali.
- **Semiconductor tracker** (**SCT**) che impiegano strisce di semiconduttore per ottenere maggiore granularità e precisione nella direzione . Nelle zone ad alto livello di radiazione il silicio è sostituito con arseniuro di gallio.

- **Transition radiation tracker** (**TRT**) basati su sottili strati di tubi aventi diametro di 4mm. Ogni tubo è intervallato da un sottile foglio di polietilene con funzione di radiatore che emette raggi X al passaggio di particelle relativistiche. I TRT permettono di seguire la traiettoria delle particelle in modo continuo e contribuiscono inoltre al miglioramento dell'identificazione degli elettroni.

La combinazione di rivelatori di traccia ad alta precisione ed elementi di tracciamento continuo, forniscono un potente strumento d'indagine per le particelle in un range di $| \quad | < 2.5$.

1.3.1 Sistema calorimetrico

Il calorimetro è formato da una prima sezione interna cilindrica (sezione *barrel interna*) ed una terminale (*end-cap*) ove è impiegata una tecnologia basata sull' Argon liquido (LAr) come elemento attivo per la sua resistenza alle radiazioni. Nella seconda sezione cilindrica più esterna è invece utilizzato un tipo di scintillatore di nuova tecnologia; nell' apparato sono presenti principalmente calorimetri adronici ed elettromagnetici. La figura 1.4 offre una vista globale di tutto il sistema calorimetrico.

Nella regione *barrel* più interna è situato un primo calorimetro elettromagnetico, racchiuso in un criostato, che risulta separato in due metà: tra gli spazi degli assorbitori, realizzati in piombo, viene immesso dell'Argon liquido. Il calorimetro elettromagnetico dovrà fornire la misura di posizione ed energia di elettroni e fotoni in un ampio intervallo di energie. La sua profondità minima, in questa regione, è pari a ~ 26 X_0 (lunghezza di radiazione) per $\quad = 0$.

Nella regione *end-cap* il calorimetro ha una zona formata da due identici "terminatori", aventi ognuno 4 moduli coassiali a forma di cuneo: in questa regione, detta *forward Lar calorimeter*, la profondità del calorimetro è di 28 X_0 per tutti i valori di . Il calorimetro adronico *end-cap*, situato in posizione coassiale rispetto a questi moduli e formato da un insieme di 32 moduli realizzati con la tecnologia dell' Argon liquido, dovrà invece identificare i jet e misurarne energia e direzione.

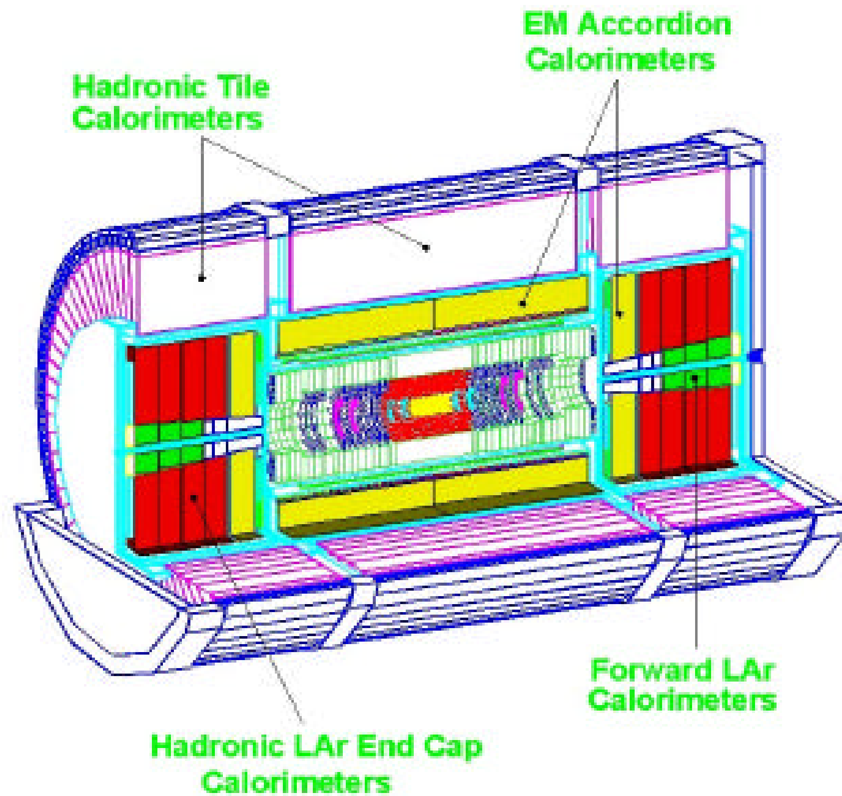


Figura 1.4: Vista complessiva del sistema calorimetrico, con indicati i diverse tipi di calorimetri.

Il calorimetro adronico a scintillatore è basato su una tecnologia realizzata con assorbitori in ferro e piastre di plastica con funzione di scintillatori. Le piastre sono disposte perpendicolarmente rispetto alla direzione del fascio, e l' informazione è trasmessa su fibra ottica.

Il sistema calorimetrico così strutturato sarà in grado di fornire, ad alte prestazioni, misure complete delle energie di elettroni, fotoni e getti adronici, ed una buona efficienza per la misura dell' energia trasversa mancante. Tali caratteristiche sono state simulate in modo estensivo e confermate da risultati di test-beam.

La risoluzione in energia offerta dal calorimetro elettromagnetico può essere descritta, in buona approssimazione, tramite la relazione:

$$\frac{}{E} = \frac{0.1}{\sqrt{E[GeV]}} + \frac{400MeV}{E} + 0.003$$

Per il calorimetro adronico la risoluzione in energia, per pioni carichi, è data dalla relazione:

$$\frac{\quad}{E} = \frac{0.45}{\sqrt{E[GeV]}} + 0.015$$

1.3.3 Lo spettrometro dei muoni

Il criterio di realizzare uno strumento che potrà fornire un' elevata risoluzione nella misura dei muoni, è alla base del progetto dello spettrometro di μ dell' esperimento ATLAS: questa elevata risoluzione, tra l'altro, sarà garantita in un range di momento trasverso p_T da 5 GeV a valori superiori ai 1000 GeV e in un intervallo di pseudorapidità pari a $| \ | \ \ 3$. Tali proprietà permetteranno di ottenere, con alte risoluzioni, i decadimenti rari quali $H \quad ZZ^* \quad$ μμ che quelli più frequenti quali J/ $\quad \mu^+\mu^-$.

Lo spettrometro è basato su di un sistema di magneti superconduttori in aria di forma toroidale suddivisi in due parti: una centrale (*barrel*) ed una più esterna (*end-cap*). Il toroide centrale sarà lungo 26 m con diametro esterno di 19.5 m ed interno di 9.5 m, mentre i toroidi degli *end-cap* avranno una lunghezza di 5.6 m per un diametro interno di 1.26 m. Questo sistema di toroidi sarà in grado di fornire valori di campo B dell' ordine dei 3T, che permetterà di minimizzare lo scattering coulombiano multiplo, incrementando così la precisione nella determinazione del momento dei muoni. Nella figura 1.5 è riportata una veduta azimutale dello spettrometro.

Ogni toroide è formato da otto bobine superconduttrici disposte simmetricamente attorno al fascio: le sezioni *end-cap* risultano ruotate rispetto al toroide interno. Mentre le bobine del toroide centrale sono racchiuse singolarmente in un criostato, quelle degli *end-cap* sono invece contenute tutte assieme in un unico criostato.

Le camere dei muoni, necessarie alla misura della loro traiettoria, sono poste direttamente sui toroidi: nella sezione centrale (*barrel*) le camere sono disposte su tre strati. Il primo strato, quello più interno, è posto a ridosso del calorimetro adronico, mentre quello più esterno è posto direttamente sul muro della caverna. Tale disposizione permetterà di determinare sagitta ed impulso delle tracce cariche che hanno attraversato lo spettrometro.

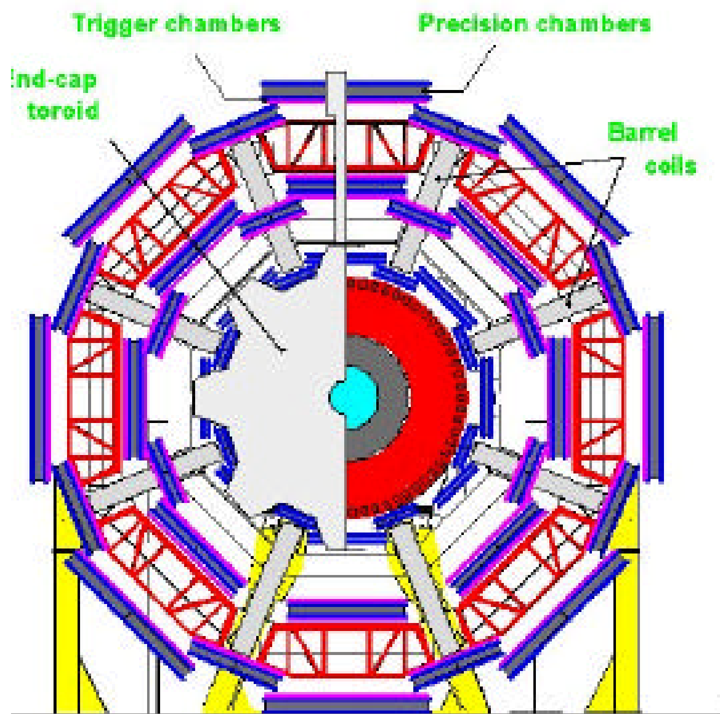


Figura 1.5: Vista azimutale dello spettrometro.

Le camere della zona *end-cap* sono invece montate in posizione verticale e disposte davanti e dietro le facce dei criostati.

Complessivamente le camere offriranno una copertura dell' intervallo di pseudorapidità $| \quad |$ 3 su tutto l' angolo ; esse saranno realizzate impiegando due diverse tecnologie di rivelatori e precisamente:

- **Monitored drift tube** (MDT) consistenti in tubi a deriva di alluminio, pressurizzati (~ 4 atmosfere) e aventi un diametro dell' ordine di 30 mm, con cui è possibile risalire alla posizione della traccia carica che ha attraversato il rivelatore, misurando il tempo di transito degli elettroni di ionizzazione prodotti dalla particella primaria nel tubo . Le camere MDT saranno costituite da due strati, aventi ciascuno 3 o 4 tubi, posti ad una distanza di 30-50 cm tale che sia anche possibile ottenere localmente un' informazione della direzione della particella. Le camere presenti sia nella regione *barrel* che *end-cap*, con la loro superficie di $\sim 5000 \ m^2$, daranno luogo alla presenza di $\sim 300 \quad 10^3$ canali di lettura. Il

problema dell' allineamento delle camere sui tre piani sarà risolto per mezzo di un sistema di allineamento ottico, permettendo così la necessaria precisione nella misura della sagitta nella regione magnetica, per la determinazione del momento trasverso p_t del muone.

- **Cathode strip chambers** (CSC) composte da sezioni multi cavo con un catodo centrale segmentato e due anodi da esso equidistanti. L' alta precisione nella misura della posizione è ottenuta determinando il centro di gravità della carica indotta, attraverso effetto valanga, su uno dei segmenti del catodo: si hanno in questo caso risoluzioni dell' ordine delle decine di μm. Con questa tecnologia verranno realizzate le camere nelle regioni ad alti valori di pseudorapidità, incrementando e raffinando le informazioni delle camere MDT. L' insieme delle camere CSC fornirà ~ 100 10^3 canali.

La figura 1.6 fornisce una visione laterale dello spettrometro.

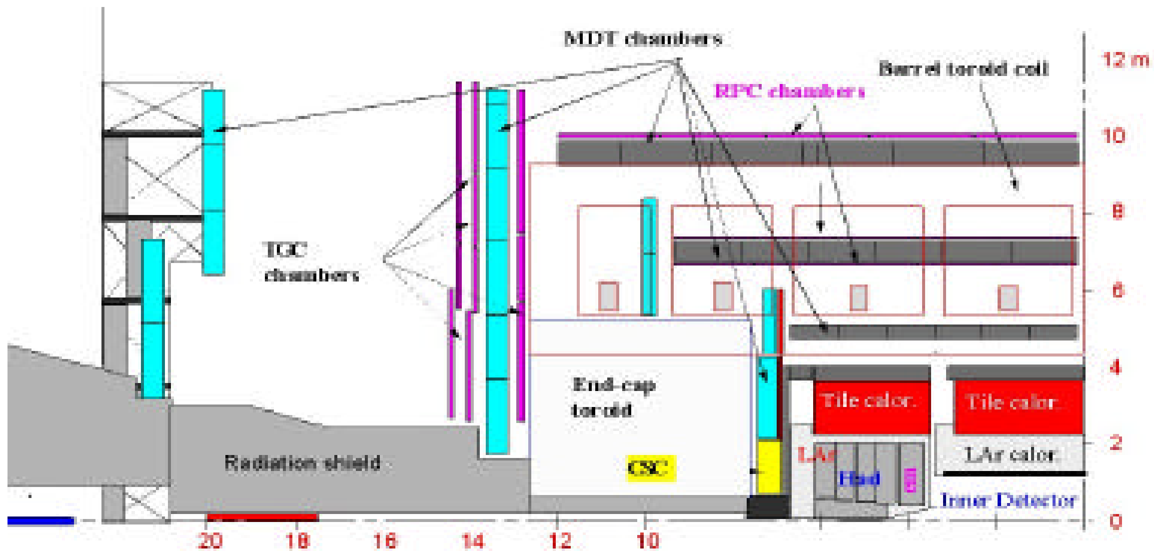


Figura 1.6: Spaccato laterale di una metà dello spettrometro dei muoni, con evidenziate le diverse camere di precisione e di trigger.

Nella camere di trigger della regione *barrel* verranno utilizzati dei rivelatori a gas RPC (Resistive Plate Counters) a piatti piani, resistivi e paralleli, con una lettura capacitiva ed una risoluzione temporale di ~ 2 ns. Nelle regioni degli end-cap verranno

invece utilizzati dei rivelatori del tipo TGC (Thin Gap Counters) che sono delle camere a fili operanti in modalità saturata, e sempre a lettura capacitiva, che presentano però una peggiore risoluzione temporale (5 ns), ma sono in grado di sostenere flussi di particelle molto intensi (come è il caso delle regioni in avanti).

Il sistema di trigger del primo livello dei muoni si baserà su una selezione veloce, e semplificata, di muoni ad alto o basso momento trasverso p_t; lo schema proposto utilizza due stazioni, la prima composta da due piani di RPC, quella esterna da tre piani di RPC. La figura 1.7 dà un idea del meccanismo su cui si basa l'algoritmo del trigger muonico.

Il trigger a basso p_t (5-6 GeV e per la fisica del B) richiede che ci sia una coincidenza sui due piani che compongono la prima stazione, mentre quello ad alto p_t (20-40 GeV e per alte luminosità) richiede una coincidenza sulla prima e sulla seconda stazione.Lo stesso criterio verrà adottato per la regione degli *end-cap*.

L' insieme delle camere di trigger si estenderà su una superficie di ~ 7000 m^2 ed i canali di lettura saranno ~ 900 10^3.

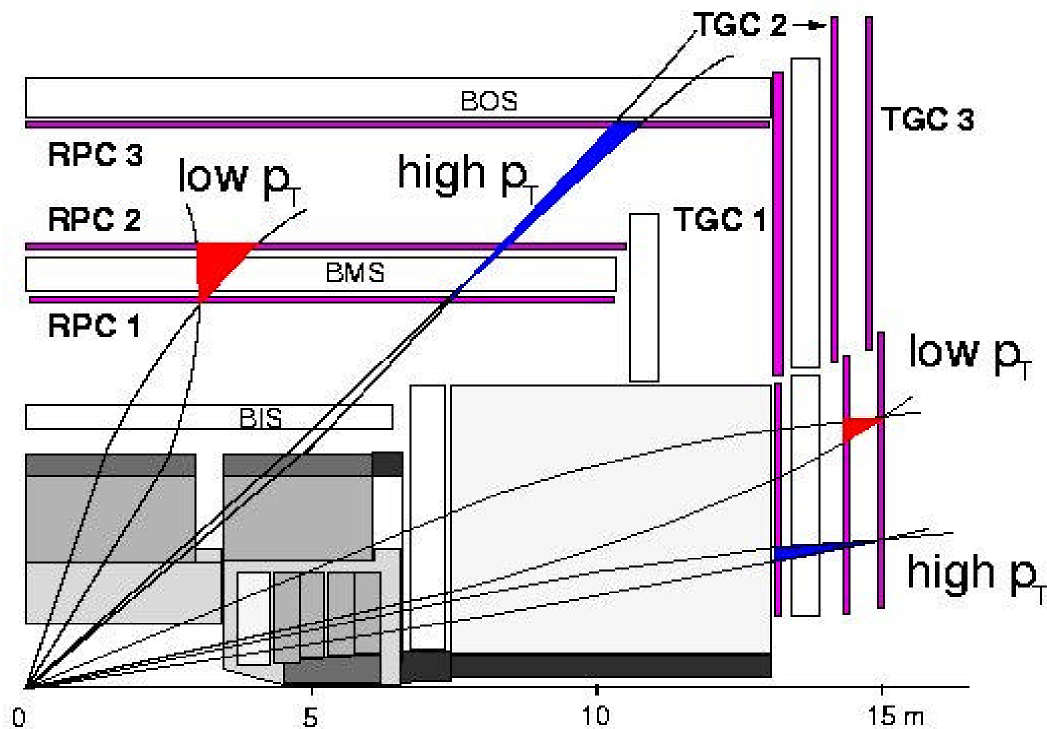


Figura1.7: Schema del trigger di livello 1 nello spettrometro muonico.

1.4 Il trigger dell'esperimento ATLAS

1.4.1 Le strategie di trigger

Dall'insieme dei numerosi canali previsti per il rivelatore interno (decine di milioni), più quelli per il calorimetro (alcune centinaia di migliaia) e più quelli provenienti dallo spettrometro (qualche milione di canali), è stato stimato che ogni evento avrà una dimensione di 1 ~ Mbyte; tenendo conto che si ha un bunch-crossing ogni 25 ns, il flusso totale dei dati proveniente dall'elettronica di front-end (quella più vicina al rivelatore) e prima di ogni tipo di selezione (trigger) sarà dell'ordine di ~ 40 Tbyte/s.

Questo flusso dovrà ovviamente essere ridotto ad un tasso ragionevole affinché possa essere processato e quindi memorizzato in dispositivi di massa, senza creare un considerevole tempo morto. Per questo scopo è necessario un sistema di riduzione dei dati altamente sofisticato che, oltre ad offrire una selezione di eventi interessanti, non faccia perdere informazioni su quella che potrebbe essere una fisica oltre il Modello Standard. Nell'esperimento ATLAS il sistema di trigger e di acquisizione dati è basato su un'architettura a tre livelli, ognuno con un alto grado di parallelismo per eliminare congestioni nel trasporto dei dati e ridurre i tempi di processamento dell'evento; tra l'altro il tipo stesso di tecniche di rivelazione dell'esperimento induce ad un uso massiccio di strutture parallele. Il primo livello è sincrono con la macchina (40 MHz) mentre gli altri due sono completamente asincroni.

Il trigger di livello 1 (LVL1) acquisisce i dati provenienti dai calorimetri e dallo spettrometro di muoni alla frequenza del bunch-crossing, ovvero di 40 MHz, con una bassa granularità ed una latenza di 2 μs. Attraverso il confronto dei valori misurati per momento trasverso di elettroni, fotoni, muoni, jet ed energia trasversa mancante con diverse soglie predefinite, il trigger dovrà selezionare una frazione degli eventi prodotti, identificare bunch crossing ed indirizzare i dati al livello successivo: grazie a questo processo di selezione degli eventi, il flusso dei dati verrà ridotto di circa un fattore 400 ovvero si avrà un valore di ~ 100 kHz. In realtà le simulazioni di fisica [5] prevedono una frequenza di trigger di 40 kHz con un massimo valore di 75 kHz innalzabile fino a 100 kHz ; questo limite è importante perché fissa i parametri per il disegno dell'elettronica di lettura dai rivelatori.

A causa dell'elevata complessità degli eventi presenti nel rivelatore interno, a questo livello le informazioni che esso fornisce non vengono utilizzate; in tal modo si riesce a mantenere un basso valore per la latenza del processo del LVL1. Inoltre simulazioni accurate mostrano che l'uso di questo rivelatore al livello 1 non aiuta molto a diminuire la frequenza di trigger.

Il trigger di primo livello non solo riduce il flusso iniziale di un fattore 400 ma fornisce inoltre informazioni sulle cosidette RoI (Region of Interest) associate alla accettazione di un evento; le RoI, su cui poi si baserà il trigger di secondo livello, sono quelle aree dei rivelatori, date in coordinate e e individuate dal LVL1, caratterizzate dal tipo di particelle (muoni, elettroni/fotoni etc.) che esse rappresentano.

Il trigger di livello 2 (LVL2) utilizza la massima granularità dei dati provenienti dai calorimetri, dallo spettrometro dei muoni e dal rivelatore interno, per affinare le decisioni prese al livello precedente. La latenza massima prevista per questo livello di trigger è pari a 10 ms che consente rispetto al LVL1, di applicare algoritmi di decisione più sofisticati. Il processo globale di trigger LVL2 permetterà una riduzione del rate di eventi di un fattore 100.

I dati relativi agli eventi selezionati dal LVL2 passeranno così, ad una frequenza di 1 kHz, al livello 3 (Event Filter, EF) il quale, utilizzando processori più potenti che analizzano l'evento completo (e non solo le RoI) sulla base di algoritmi di ricostruzione impostati dal software offline, deciderà quale evento dovrà essere memorizzato: esso ridurrà il flusso dati di un ulteriore fattore 10 con latenze, che in questo caso non saranno un parametro critico, oscillanti tra 1 e 10 s.

1.4.2 Organizzazione del trigger e acquisizione dati

Gli ingressi provenienti dal sistema di trigger del calorimetro sono segnali analogici con celle a granularità ridotta di $= 0.1 \quad 0.1$. Per la sezione del trigger relativa allo spettrometro di muoni, si rimanda al precedente paragrafo 1.3.3 e riferimento grafico annesso della figura 1.7 .

La figura 1.8 mostra l'architettura del completo sistema di trigger. Durante la fase di lavoro del LVL1 i dati provenienti da tutti i rivelatori (calorimetro e spettrometro) vengono memorizzati in memorie pipeline: queste potranno essere analogiche o digitali. Le memorie dovranno avere inoltre sufficiente capacità (buffer per l'accodamento dei

dati) per mantenere i dati per un tempo di 2 μs, che è la latenza dell'operazione di trigger e pari ad 80 bunch-crossing.

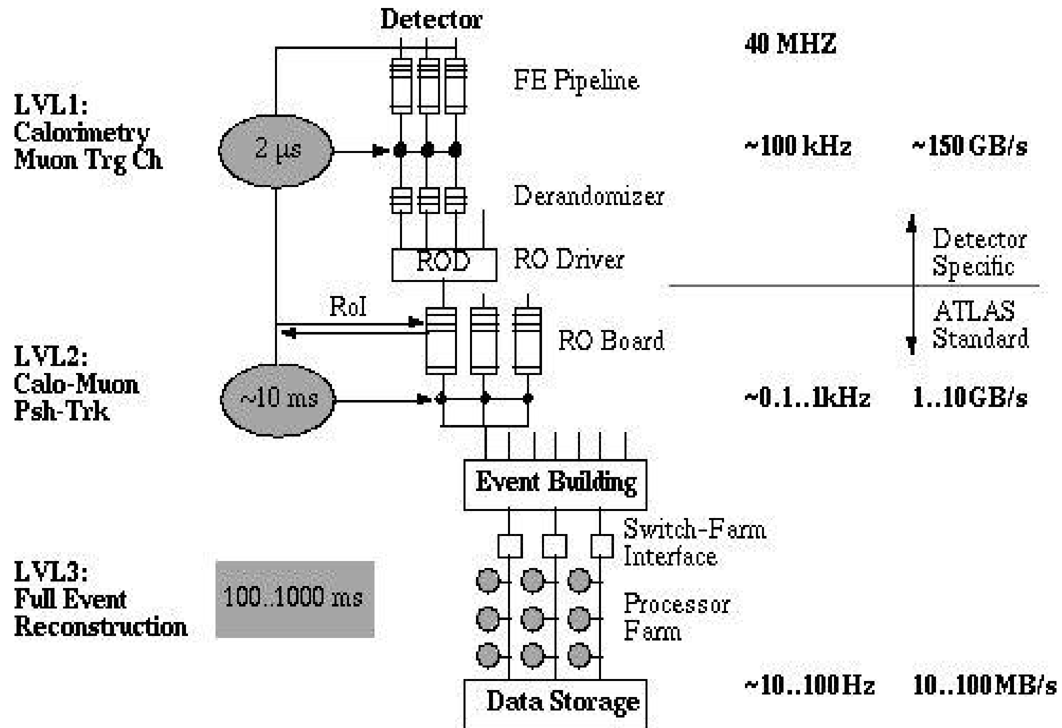


Figura 1.8: Schema a blocchi del sistema di trigger e acquisizione dati dell' esperimento.

Quando l'evento è accettato dal LVL1, tutti i dati contenuti nelle pipeline sono posti in uscita su dei buffer "derandomizzatori" le cui dimensioni debbono essere opportunamente calcolate per evitare una parziale perdita dati. Tali buffer permettono di accordare le velocità delle pipeline con la banda di lavoro dei Read Out Drivers (ROD): la trasmissione dei dati tra i buffer ed i ROD dovrebbe avvenire tramite fibra ottica, proprio per garantire in sicurezza alti rate di traffico. I ROD accorpano su un unico readout link, con una procedura di multiplexing, più di un flusso dati proveniente dalle memorie pipeline: anch'essi per la trasmissione saranno basati sull'uso di fibre ottiche: il traffico sul readout link non supera 1 Gbit/s.

L'insieme dei dispositivi fin qui analizzati, come anche l'elettronica di front-end, può essere sviluppato impiegando tecnologie tra loro diverse, ma il discorso cambia per il LVL2 dal momento che le tecnologie utilizzate dovrebbero essere omogenee: questo per

avere una certa facilità di costruzione ed un appropriato controllo del sistema di acquisizione dati di ATLAS.

Gli eventi selezionati dal LVL1 e concentrati dai ROD sono così diretti a dispositivi Read Out Buffers (ROB) del sistema di trigger di LVL2 e infine da essi allo EF; questo processo di trasporto dati dai ROB all' EF è chiamato *event building*.

Prima dell'operazione di *event building* ciascun evento è composto da diversi frammenti, ognuno di essi memorizzato in un ROB: dopo lo *event building* l'evento completo viene memorizzato in una singola memoria su cui lavorerà un processore dello Event Filter.

Verrà ora analizzata con maggiore dettaglio il trigger di livello 2.

1.5 Il trigger di livello 2

Riferendoci a quanto illustrato nei paragrafi precedenti, il trigger LVL2 è basato sull'uso delle RoI identificate [8] dal LVL1 e sul processamento di una piccola frazione del totale dei dati provenienti dai rivelatori, circa il 5%.

Il processo di trigger è suddiviso in tre fasi:

- *Conferma delle RoI di livello 1*, sulla base dei dati dei singoli rivelatori aventi precisione e granularità superiore a quelle utilizzate dal LVL1. Questa fase associa alle tracce e ai cluster di energia trovati, quantità fisiche (quali energia, impulso, rapidità, azimut, etc.) ricavate con gli algoritmi programmabili e sviluppati ad hoc[1].

- *Costruzione di oggetti fisici* (e, μ, , etc.), ove le caratteristiche ricavate dai diversi rivelatori relativi ad una RoI sono combinate assieme per fornire un'ipotesi più solida sul tipo di particella ed i suoi parametri.

[1] Simulazioni Montecarlo mostrano che già questa prima fase è in grado di ridurre notevolmente la frequenza di trigger. Ad esempio per il trigger di muoni, il taglio in impulso trasverso applicato alle tracce ricostruite nello spettrometro usando le camere di precisione (MDT) invece che quelle di trigger (RPC e TGC) come fa il LVL1, riduce la frequenza di trigger di un fattore due.

- *Selezione del tipo di trigger*, per cui tutti gli oggetti trovati nell'evento vengono confrontati con gli *item* di un menu di trigger: quest'ultimo corrisponde ad un certo numero di canali di fisica che si vogliono selezionare. In tal modo si arriva ad una decisione di carattere globale sul particolare evento.

Per la realizzazione del sistema di trigger di secondo livello sono state proposte diverse architetture: attualmente tre di esse risultano essere le più congeniali ed adeguate al tipo di problema. In tutte queste architetture comunque sono presenti dei blocchi comuni: i Readout Out Buffer (**ROB**), un **Supervisor** e un **RoI Builder**, una **rete di connessione** ed un insieme di **processori** per eseguire gli algoritmi di trigger. Il modello generico di architettura del LVL2 è mostrato nella figura 1.9.

I dati relativi ad ogni evento accettato dal LVL1 vengono trasferiti, con le velocità di flusso del trigger di primo livello, dai Read Out Driver ai ROB[2]: il numero stimato di ROB è di ~ 1700. Le informazioni fornite dal LVL1 e strutturate sotto forma di pacchetti, conterranno sia il numero identificativo dell'evento che del bunch-crossing a cui appartiene.

È stimato che per ogni evento ci sia mediamente una RoI associata ai muoni, agli elettroni e ai , mentre il numero sale a tre nel caso dei jet. Assumendo un flusso di eventi provenienti dal LVL1 pari a 100 kHz, si avrebbe un flusso su singolo link, da LVL1 a LVL2, di circa 20 Mbyte/s.

Per quanto concerne la struttura dei ROB è previsto che conterranno delle memorie multiporta con almeno due di esse per l'ingresso e l'uscita dei dati ad alta velocità; inoltre saranno raggruppati assieme attraverso una architettura bus-based che potrebbe essere lo standard VME.

La principale funzione di questi dispositivi è memorizzare i dati, relativi agli eventi accettati dal LVL1, per tutto il tempo necessario a:

- rigettare o accettare l'evento da parte del trigger di LVL2.
- trasferire al sistema di acquisizione, associato con lo Event Filter, l'intero corpo dei dati.

---

[2] Per i prototipi d'interfaccia tra LVL1 e LVL2, ovvero tra i ROD ed i ROB è stato proposto lo standard S-Link (specifica CERN). È comunque da pochi anni che lo standard è passato alla sua fase operativa, attraverso implementazioni su bus locale PCI.

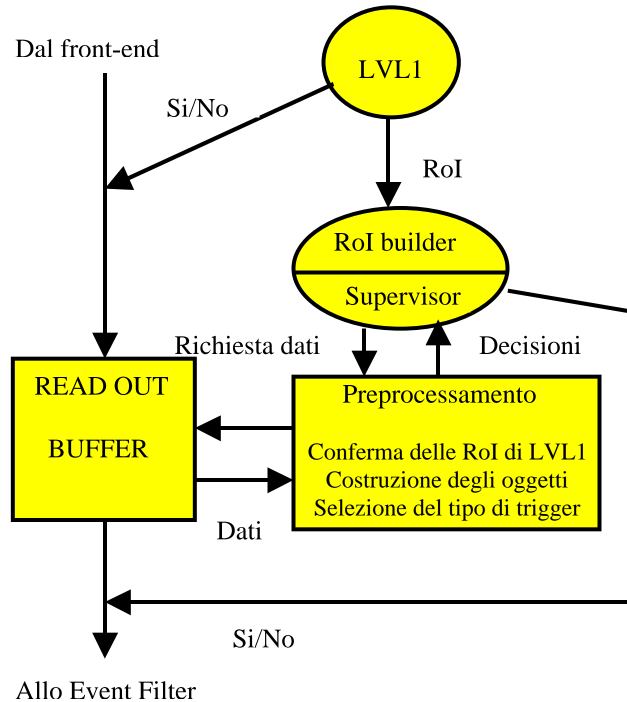


Figura 1.9: Schema funzionale del trigger di livello 2.

Per le comunicazioni con il sistema di trigger LVL2 (Supervisor + RoI builder), i ROB dovranno essere dotati di interfacce di controllo e di trasmissione dei dati. Attraverso l'interfaccia di controllo il ROB riceve le richieste per inviare dati, il quali vengono poi trasmessi tramite la seconda interfaccia dedicata. Il LVL2 riceve dal LVL1 informazioni sulle RoI ad una velocità di 100 kHz: subito dopo avvia le tre fasi del processo di trigger (*conferma delle RoI*, *costruzione degli oggetti, selezione del tipo di trigger*) dopo una operazione iniziale di raccolta e concentrazione dei dati nella memoria di un processore ed eventuale loro preprocessamento. In seguito alla fase di trigger il LVL2 prenderà la decisione di abilitare, oppure no, i diversi ROB alla trasmissione dei dati verso lo Event Filter.

Gli studi sino ad oggi effettuati [5] [8] non hanno ancora portato alla definizione finale dell'architettura del sistema di trigger di LVL2. Esistono ancora tre opzioni interessanti, ciascuna delle quali presenta vantaggi diversi risultando più appropriata in determinate condizioni sperimentali o per l'uso di alcune tecnologie.

1.5.1 Architetture del LVL2

Per comodità denotiamo con le lettere A, B, C le possibili realizzazioni dell'architettura di LVL2.

Nel modello A, mostrato nella figura 1.10, le informazioni relative alle RoI vengono inviate dal Supervisor ai dispositivi RoIC (Collettori RoI) i quali estraggono i dati delle RoI dai ROB e li inoltrano a dei processori veloci (Data Driven, DD) per l'estrazione delle caratteristiche fisiche delle RoI.

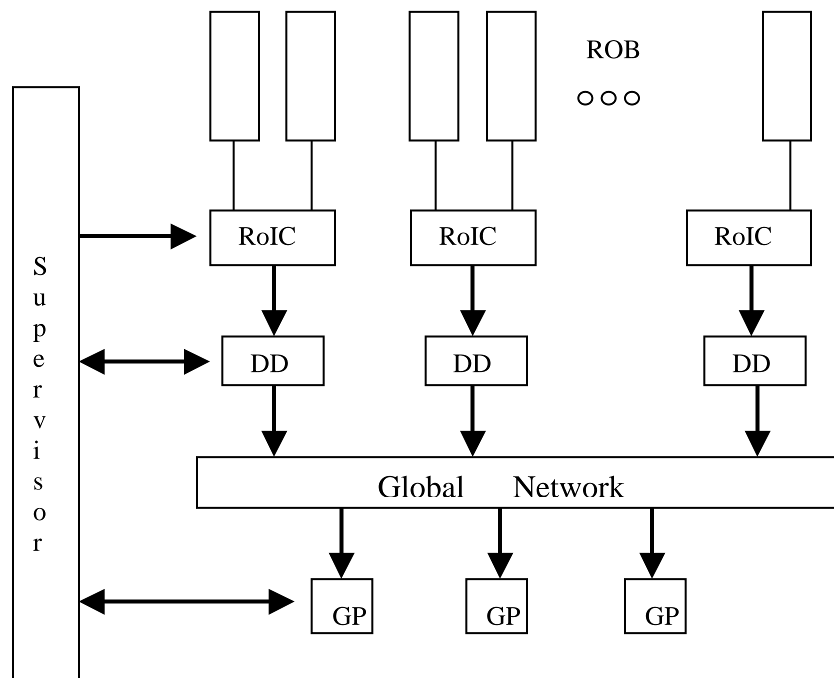


Figura 1.10: Schema dell'architettura A.

Dopo quest'operazione una rete di interconnessione trasferisce tutti i dati, provenienti da più DD, ad un processore GP (Global Processor) per un particolare evento. Anche quest'ultima fase è controllata dal Supervisor, nel senso che esso dà delle "direttive" ai DD per trasferire i loro dati ad un solo e determinato GP. Una volta che il Global Processor ha preso la decisione sull'evento, esso la comunica al Supervisor che abiliterà i ROB alla trasmissione dei dati verso lo Event Filter e informerà il Supervisor del livello 3 della decisione di trigger.

Questo schema fa uso di componenti non commerciali, bensì RoIC e Processori basati su Field Programmable Gate Arrays (FPGA)[3] con le quali si ottengono delle ottime prestazioni in termini di latenze introdotte.

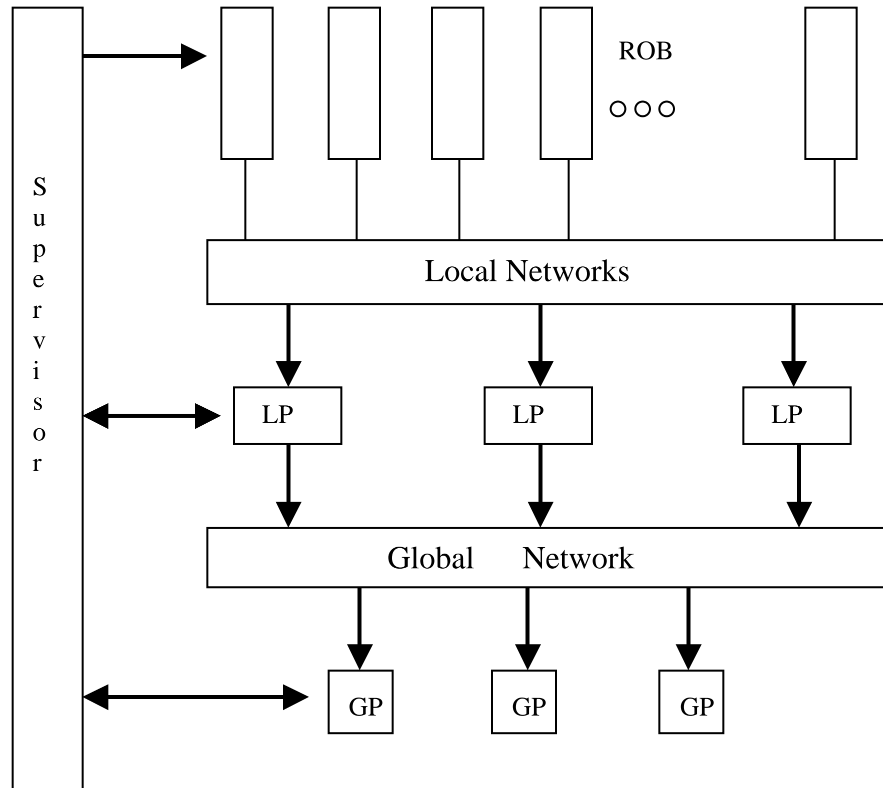


Figura 1.11: Architettura B.

Il secondo modello B, illustrato in figura 1.11, è invece strutturato con due diverse reti di interconnessione. In tale architettura le informazioni sulle RoI vengono comunicate ai ROB dal Supervisor, il quale controlla l'intero flusso dei dati. Su richiesta del Supervisor i ROB, dopo un'eventuale operazione di preprocessamento, inviano i dati relativi alle RoI a dei processori LP (Local Processor) per mezzo di una prima rete di interconnessione. I processori LP effettuano l'*estrazione delle caratteristiche delle RoI* ed attraverso un seconda rete inoltrano le proprie informazioni ad uno dei processori GP; la

[3] Logica programmabile con funzionalità aggiuntive. Essa viene programmata attraverso l'uso di sofisticati tool di sviluppo che incorporano linguaggi dedicati (implementati dalle stesse case costruttrici) ed apposite librerie con cui è possibile riprodurre al suo interno numerosi componenti elettronici, nonché crearne di nuovi. Le FPGA offrono prestazioni elevate in termini di velocità con cui vengono fornite le variabili di uscita.

decisione presa da questi processori viene comunicata al Supervisor il quale avvierà le procedure già viste nella precedente architettura. L'architettura B è un'architettura fortemente parallela (il parallelismo è insito nel meccanismo di processamento contemporaneo degli algoritmi della parte locale sui dati provenienti dai vari subdetector) ed è basata completamente su componenti commerciali (network, processori).

Nel modello C, mostrato nella figura 1.12, è presente una sola rete di interconnessione ed un numero minore di dispositivi: questo a carico però di una maggiore complessità del Supervisor nonché della rete stessa d'interconnessione.

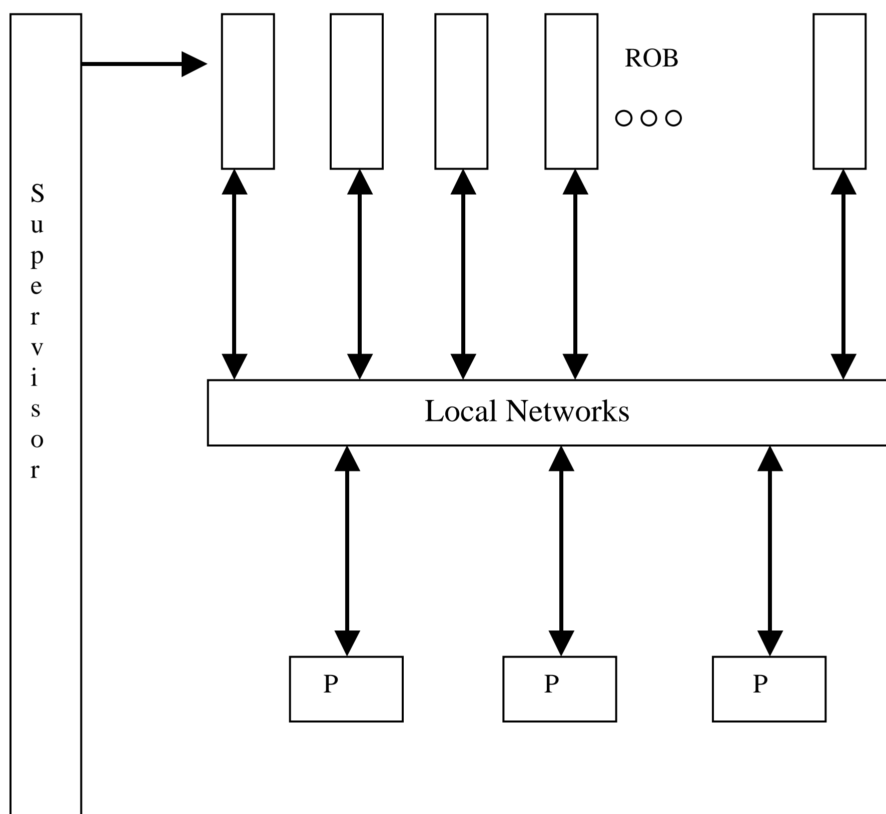


Figura 1.12: Architettura C.

In tale architettura il Supervisor è sempre delegato alla gestione del flusso delle informazioni: esso assegna l'intero evento ad un singolo processore P e gli comunica le informazioni sulle RoI. Questi preleva i dati delle RoI dai ROB nella sequenza che l'algoritmo gli richiede finché non si manifesta la condizione per la quale l'evento può essere rigettato. A questo punto il processamento e quindi il trasferimento ulteriore di dati

s’interrompe. Dopo la fase di elaborazione il processore P inoltra al Supervisor la propria decisione sull’evento: nuovamente il Supervisor attiverà l’insieme di procedure per il passaggio di informazione al livello 3. Rispetto alle precedenti architetture tutte le comunicazioni, dal trasferimento dati alle procedure di controllo, fluiscono a mezzo di una singola rete d’interconnessione. Tale architettura, basata su procedure di tipo sequenziale, è molto simile a quella adottata per l’EF (a parte l’organizzazione della farm di processori) ed è il candidato ideale per studiare un’eventuale unificazione dei due livelli di trigger (LVL2 e EF) che comunque richiederebbe una rete d’interconnesione di prestazioni eccezionali. Anche quest’ultima architettura è incentrata sull’uso di componenti commerciali.

1.5.2 Le reti per l’ interconnessione

In tutti e tre i modelli di possibili architetture sono sempre presenti una o due reti d’interconnessione (*switching network*); siccome i dati trasmessi sono sotto forma di pacchetti, le reti a commutazione di pacchetto (*packet switching network*) rappresentano la naturale scelta per le reti d’interconnessione; inoltre esse permettono di trattare in modo adeguato gli alti volumi del flusso dei dati. Nel Capitolo 3 verranno ripresi ampiamente questi aspetti fornendo un’analisi più esaustiva delle reti di interconnessione.

Per studiare il comportamento delle *packet switching network* nelle diverse architetture di LVL2, sono state indicate delle moderne tecnologie [15] [17] [18] [19]: ATM (Asynchronous Transfer Mode), SCI (Scalable Coherent Interface), DS-Link (Data Strobe Link). Tutte queste tecnologie forniscono alte velocità trasmissive, anche se l’ultima risulta inadeguata per i flussi previsti in ATLAS ma estremamente vantaggiosa in termini di costo (è già pronta comunque una versione più veloce denominata HS-Link).

Nello sviluppo di questo lavoro di tesi, imperniato proprio sullo studio di *packet switching network*, è stata impiegata la tecnologia DS-Link che rientra nello standard IEEE 1355 [20]. Nel capitolo seguente verrà dapprima illustrato lo standard IEEE 1355 e successivamente descritta nei dettagli la tecnologia DS-Link, sia per quanto riguarda i dispositivi fisici veri e propri sia per quanto concerne il protocollo di comunicazione implementato nella tecnologia.

Capitolo 2

Lo standard IEEE 1355

2.1 Introduzione

Nei sistemi di comunicazione l'aspetto dell'interconnessione tra dispositivi costituisce un campo di studio ampio ed in continua evoluzione, in cui la scelta delle tecniche e delle tecnologie da utilizzare assume un aspetto rilevante, poiché sempre più spesso esse devono garantire la transazione di alti volumi di traffico informativo, elevate velocità di trasmissione ed ottime proprietà di rivelazione-correzione di errori. Tra le tecniche attualmente utilizzate quella basata su *bus*[1] risulta essere la più diffusa ma, rispetto ad esempio alla tecnica punto-punto, presenta alcune limitazioni e svantaggi; lo standard IEEE 1355 [20] su cui è basata la tecnologia DS-Link, ed in cui sono previste trasmissioni di tipo asincrono e full-duplex, è stato sviluppato per la tecnica punto-punto e per garantire quelle esigenze sempre più sentite di alte velocità di comunicazione.

Le comunicazioni punto-punto coinvolgono soltanto due processi[2]: un processo che spedisce un messaggio ed un altro che lo riceve; la connessione così realizzata prende il nome di *canale*. Se confrontate con comunicazioni basate su bus, le punto-punto in un sistema con più dispositivi offrono principalmente:

- Assenza di contesa nel meccanismo di comunicazione.
- Nessuna diminuzione della capacità di carico con l'aggiunta di altri dispositivi al sistema; in questo modo inoltre la larghezza di banda di comunicazione non viene saturata: anzi, crescendo il numero di dispositivi, essa aumenta ulteriormente.

Nella comunicazione tra due processi può accadere che il processo trasmittente (*sender*) invii un insieme di dati, ad una determinata velocità, e un processo ricevente

[1] Come ad esempio la tecnologia dominante per connettere la CPU con la memoria ed i sottosistemi di input-output (I/O), oppure architetture di comunicazione basate su bus VME etc.

[2] L'accezione di processo è molto generale; esso può essere visto come una "scatola nera" con uno stato interno, che può dialogare con altri processi usando un canale di comunicazione.

(*receiver*) il quale li acquisirà ad una propria velocità di lavoro: si suppone quindi che ci siano velocità diverse tra i due processi.

Inserendo una FIFO[3] tra gli stessi, il *sender* potrà inviare dati finché risulta spazio disponibile nella memoria: una volta satura il processo *sender* interromperà il suo lavoro. Dall'altro lato il *receiver* acquisendo i dati attraverso la FIFO ne garantirà lo svuotamento, permettendo così la riattivazione del processo trasmittente: ovviamente in questo modo la velocità di trasmissione sarà governata dal processo più lento. Vedremo più avanti che questo meccanismo, in alcune versioni dello standard IEEE 1355, viene ulteriormente perfezionato con l'introduzione di un'altro meccanismo chiamato *flow control token*.

Le comunicazioni bidirezionali, come prevede lo standard, sono garantite da due coppie di FIFO come in figura 2.1.

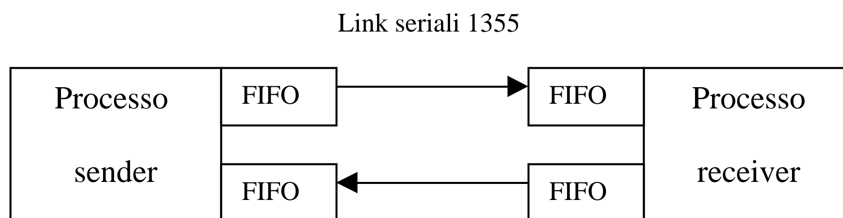


Figura 2.1 : Modello in cui sono implementate FIFO in trasmissione e ricezione per ogni processo, con l'impiego di link d' interconnessione seriali.

I dati trasmessi sono strutturati sotto forma di pacchetti ove il primo, o i primi byte, ne formano la testa che permette di indirizzare l'informazione verso uno specifico processo ricevente. Il pacchetto è poi formato dal corpo dei dati vero e proprio (messaggio) ed infine da uno speciale carattere (terminatore) indicante il tipo di pacchetto trasmesso. Il formato di questa struttura è mostrato in figura 2.2.

Poiché nello standard la lunghezza dell'intero pacchetto non è vincolata a particolari dimensioni, sia la testa che i dati possono assumere qualsiasi misura permettendo così l'utilizzo di pacchetti secondo innumerevoli formati standard [20]: MPEG, Ethernet, ATM etc.

[3] Memoria, ove FIFO è l'acronimo di **F**irst **I**n **F**irst **O**ut.

Figura 2.2 : Semplice struttura di un pacchetto con HDR = Testa, EOP = Fine del pacchetto e DATI = dati del messaggio. La freccia rappresenta la direzione di "spostamento" del pacchetto.

Ad esempio la cella ATM è realizzabile utilizzando 5 byte per la testa (ATM Hdr) altri 48 byte per il dato (ATM payload) ed un terminatore di fine pacchetto (EOP) . Ne è riportato un esempio in figura 2.3.

Figura 2.3: Pacchetto di tipo ATM realizzabile con lo standard IEEE 1355.

Consideriamo ora delle interconnessioni in cui vengono utilizzati più processori, aventi localmente dei processi attivi. Se con la testa del pacchetto IEEE 1355 identifichiamo il processo-destinazione a cui deve pervenire il messaggio del processo-sorgente, allora attraverso dei dispositivi chiamati *routing switch*, in grado di stabilire dinamicamente a quale processo remoto l'informazione è destinata, è possibile realizzare interconnessioni punto-punto tra qualsiasi processo; il *routing switch* realizza un indirizzamento del messaggio (*routing*) interpretando la testa del pacchetto.

Per piccoli sistemi di interconnessione [39] è comunque possibile indirizzare messaggi tra processi locali e remoti impiegando dei dispositivi[4] che svolgano un minimo lavoro di *routing*: interpretando la testa del pacchetto, essi instradano il messaggio verso la destinazione.

La figura 2.4 mostra per semplicità soltanto 2 processori, ognuno dei quali presenta 2 processi attivi ed un *routing switch* intermedio che permette d'instradare il messaggio qualora sussista una fase di comunicazione tra processo A e processo B (vedi linea tratteggiata). È importante osservare che non sono stati finora posti dei limiti

[4] Citiamo ad esempio i transputer T9000 di cui si darà più avanti una breve descrizione.

alle connessioni fisiche ovvero si suppone che ad ogni interconnessione punto-punto sia associato un distinto collegamento fisico.

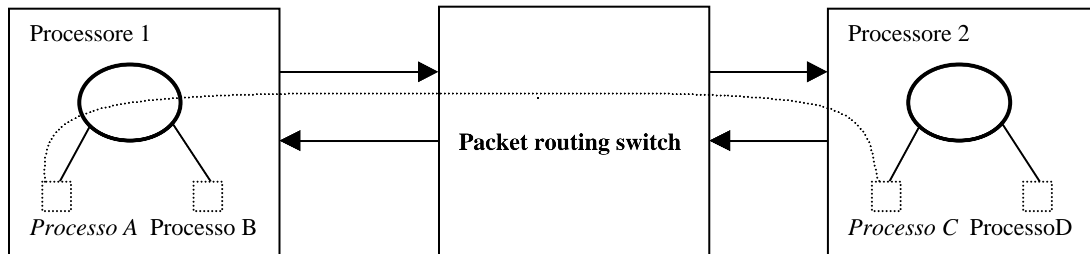


Figura 2.4: Connessione tra processori remoti attraverso un routing switch. Con la linea tratteggiata è rappresentata la comunicazione stabilita tra processi A ed E.

Nel momento in cui si hanno più processi che debbono essere interconnessi e la risorsa fisica è il singolo link, lo standard IEEE 1355 prevede l'uso di tecniche di multiplexing: queste ultime dovrebbero essere implementate direttamente sui processori che permettono l'attivazione dei processi di comunicazione[5].

Lo standard IEEE 1355 è strutturato secondo un insieme di regole sia per quanto concerne lo scambio di informazioni (strato logico), che per il mezzo fisico impiegato (strato fisico).

Lo strato fisico può essere implementato impiegando differenti tecnologie la cui identificazione è data secondo il formato a tre parametri :

SC-TM-dd

I possibili valori di questi parametri sono dati dalla *Tabella 2.1*. Se quindi ci riferiamo ad esempio alla tecnologia DS-DE-02, essa sarà realizzata con:

- Codifica Data/Strobe
- Trasmissione di tipo differenziale
- Velocità max di 200 Mbaud[6]

---

[5] E il caso del VCP (Virtual Channel Processor) implementato sul già menzionato transputer T9000.
[6] Il baud rappresenta quante volte in un secondo il segnale cambia il suo valore, mentre bit/s rappresenta il numero di bit trasmessi in un secondo.

PARAMETRO	*VALORE*	*DESCRIZIONE*
SC	**DS**	codifica Data/Strobe
	TS	codifica Three-of-six
	HS	codifica High Speed
TM	**SE**	Trasmissione Single ended
	DE	Trasmissione Differenziale
	FO	Trasmissione Fibra ottica
dd	Due cifre che rappresentano la massima velocità operativa	
	in unità di 100 Mbaud	

Tabella 2.1: Valore dei parametri per identificare le diverse tecnologie.

Un'ulteriore caratterizzazione delle tecnologie è riportata nella Tabella 2.2: per ogni tecnologia sono specificate in modo dettagliato le diverse implementazioni.

Tecnologia	Baud Rate	Massimo Rate dei Dati (Bidirezionale)	Supporto di Trasmissione	Distanza max.	Specifiche Dei connettori
Unità	MBaud	Mbyte/s	-	Metri	-
DS-SE-02	1-200	35	4 cavi (Tracce su PCB)	1	Nessuna
TS-FO-02	250	35	2 fibre multimodo	300	IEC 1754-6
DS-DE-02	1 – 200	35	8 cavi	30	IEC 48B
HS-SE-10	100 – 1000	160	2 cavi coax.	5	IEC 1076-4-107
HS-FO-10	700 – 1000	160	2 fibre multimodo	100	IEC 1754-6
			2 fibre monomodo	500	IEC 1754-6

Tabella 2.2: Caratteristiche ulteriori di alcune tecnologie implementate.

Lo strato logico, definito anche come *Protocol Stack,* è suddiviso in livelli ad ognuno dei quali sono assegnate ulteriori regole. Per quanto concerne i livelli, essi verranno illustrati più avanti quando saranno descritte le tecnologie DS-SE-02 e DS-DE-02.

2.2 Lo standard IEEE 1355 nelle versioni DS-SE e DS-DE

Queste due tecnologie con velocità massime di trasmissione pari a 100 Mbit/s, sono state implementate dalla SGS-THOMSON con la dicitura DS-Link™, relegando il tipo DS-SE a comunicazioni punto-punto su singolo PCB e la versione DS-DE per comunicazioni su più lunghe distanze[7].

Per capire meglio i meccanismi di gestione delle comunicazioni, vedremo ora in dettaglio lo strato fisico ed i livelli del *Protocol Stack* relativi al DS-Link: questi ultimi differiscono leggermente, come terminologie adottate, da quelli illustrati per lo standard IEEE 1355-DS-DE / SE, [20] [34] .

2.2.1 Lo strato fisico

I *drivers* di uscita dei dispositivi DS-Link forniscono segnali TTL compatibili, adeguati per connessioni di lunghezza vicina al metro, realizzate su PCB, con impedenze dell' ordine di 100 . Per riuscire a coprire distanze maggiori di 1 metro (DS-DE), vengono impiegati dei buffer differenziali AT&T della serie 41 (High Performance Line Drivers, Receivers, Transceivers) che permettono flussi d'informazione fino a 400 Mbit/s. Essi convertono un segnale di tipo TTL in uno differenziale (con livelli pseudo-ECL), garantendo una sufficiente rimozione dei potenziali di massa flottante tra due dispositivi DS-Link. Per la propagazione dei segnali vengono utilizzati dei cavi a 10 fili, schermati singolarmente, della Madison cable (modello 2791) di cui otto servono per le trasmissioni DS-Link di tipo differenziale, uno per la massa ed uno per impieghi speciali. I cavi presentano un ritardo per unità di lunghezza pari a circa 0.15 ns/m: con lunghezze dello ordine delle decine di metri, intervengono ritardi di propagazione non più trascurabili, che possono portare ad errori sui segnali trasportati. Da prove eseguite presso il Cern [14] è emerso che l' andamento del *Max bit-rate* (max. velocità di trasmissione esente da errori), in funzione della lunghezza dei cavi utilizzati, rimane costante e pari a 100 Mbit/s per distanze di circa 10 metri. Oltre queste lunghezze i segnali cominciano a degradarsi, facendo abbassare il *Max bit-rate*: infatti per distanze attorno ai 15 metri si hanno valori del *Max bit-rate* di circa 75 Mbit/s.

---

[7] Vedi Tabella 2.2 .

2.2.2 Lo strato logico

I livelli del *Protocol Stack* sono in totale 4, e sono enumerati secondo una gerarchia:

- **Livello 1** Bit level
- **Livello 2** Token level
- **Livello 3** Packet level
- **Livello 4** Higher level

Bit level

Come visto in precedenza il parametro DS indica una codifica dei segnali detta Data/Strobe. Per Data si considera il segnale binario che trasporta informazione vera e propria cadenzato da un clock. Invece con Strobe è da intendere un segnale binario, che varia di stato qualora il Data permanga ad uno stesso livello per più di un ciclo di clock. Trasmettendo informazione in questo modo, il *receiver* può estrarre da questi due segnali sia l'informazione vera e propria (Data) che il segnale di clock: questo ultimo semplicemente eseguendo un'operazione XOR sui due segnali. Si è in tal modo evitato il problema di avere una sincronizzazione tra *sender* e *receiver*; ovvero la trasmissione è di tipo asincrono.

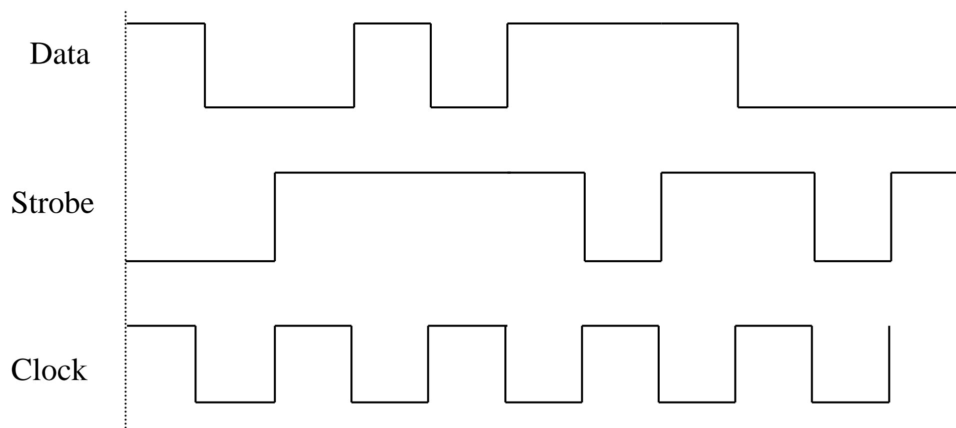


Figura 2.5: Formato del Data, Strobe e del Clock ricostruito da essi con operazione XOR.

Come già premesso la velocità di trasmissione su DS-Link è pari a 100 Mbit/s, oppure esprimendola in altra unità pari a 100 MBaud. La figura 2.5 mostra la forma dei segnali su una singola coppia di connessione: ovviamente se si considera la versione DS-SE ci sarà un Data e uno Strobe su entrambe le coppie di connessione.

Token level

La sequenza di bit trasmessi tra dispositivi è suddivisa in token[8] di tipo diverso: *Data token* che contengono bit d' informazione, *Control Token* aventi informazioni di controllo ed ulteriormente suddivisi in 4 sottotipi come mostrato nella Tabella 2.3. Entrambi i token presentano un bit di controllo che ne identifica il tipo ed un bit di parità per il controllo degli errori. I *Data token* contengono 8 bit di *data*, mentre i *Control token* ne contengono solamente 2 (che chiameremo per comodità bit *info*) attraverso cui vengono identificati i 4 sottotipi: un totale di 10 bit per i *Data token* e 4 bit per i *Control token*.

Il campo che compete ad ogni bit di parità di ciascun token riguarda:

- il bit di parità stesso
- i bit di *data* o *info* del token precedente
- il proprio bit di controllo

Nella figura 2.6 è illustrata una stringa di bit che mostra le porzioni di bit incluse dal bit di parità.

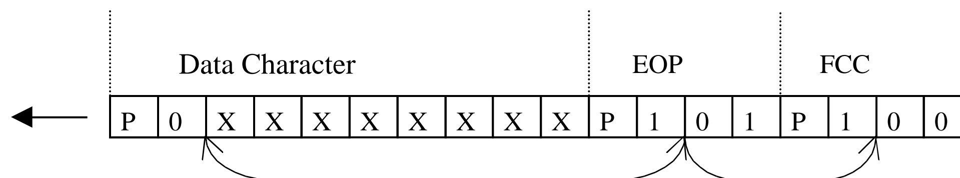


Figura 2.6: Porzioni di bit incluse dalla parità. Con X sono indicati dei possibili valori dei data-bit e con P il valore del bit di parità.

---

[8] Ovvero un carattere o simbolo.

Sottotipo	Acronimo	Valore (bit)
Flow control token	FCT	P100
End of packet	EOP	P101
End of message	EOM	P110
Escape token	ESC	P111

Tabella 2.3: Codice rappresentativo dei 4 sottotipi di Control token.

Questo meccanismo di lavoro del bit di parità permette di rilevare un singolo errore nei token (*Data o Control*) sebbene essi abbiano lunghezze diverse; la parità impiegata è di tipo dispari.

Un ultimo token, *NULL token*, è utilizzato per stabilire una connessione tra dispositivi DS-Link: dal momento che un'uscita DS-Link è attivata, essa trasmette in modo continuativo dei *NULL token* finché un altro dispositivo ad essa connesso non viene attivato. Il NULL token è formato da un FCT ed un EOP come nella figura 2.7.

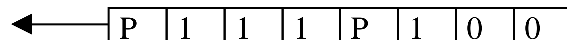


Figura 2.7: Composizione di un NULL token.

Per il **Token level** è previsto anche un meccanismo di controllo del flusso non visibile a più alto livello di protocollo. Come è stato osservato in precedenza lo standard IEEE 1355 è basato sull'uso di memorie FIFO; per evitare di saturare il buffer di memoria di un *receiver* a seguito di trasmissioni di un *sender*, è stato pensato di far scambiare tra essi dei messaggi di controllo denominati FCT (*flow control token*).

In questo meccanismo il *sender* invia una quantità di dati pari ad 8 token (indistintamente data o control) e poi si pone in condizione di attesa. Nel contempo il *receiver* acquisisce i token trasmessi e, nell' istante in cui il suo buffer ha spazio per almeno 8 token, invia al *sender* un FCT. Quest' ultimo ricevendo un FCT riprende la sua trasmissione per altri 8 token, ponendosi successivamnente in condizione di attesa. In questo modo si evitano perdite di dati dovute a riempimento dei buffer.

Le dimensioni di questi ultimi sono comunque di 20 token, ben superiori, per ovvie garanzie di sicurezza, agli 8 token previsti dal meccanismo del controllo di flusso. Nella figura 2.8 è riportato uno schema del flusso di FCT nel caso di una trasmissione unidirezionale.

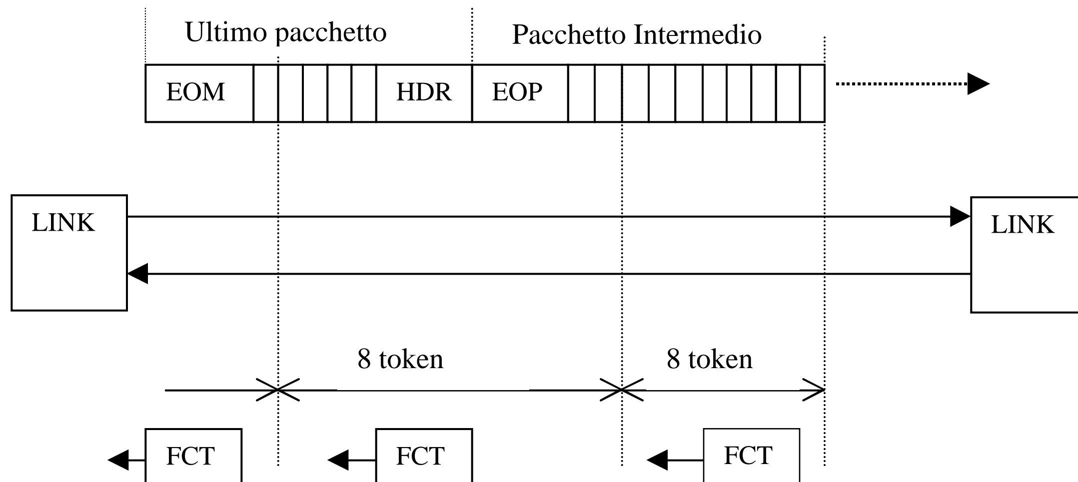


Figura 2.8: Schema del flusso di FCT per una trasmissione di tipo unidirezionale. In alto si osserva un messaggio composto da più pacchetti, terminato da un EOM. In basso sono riportati gli FCT, che viaggiano in direzione opposta al messaggio, inviati ogni 8 token ricevuti.

Per trasmissioni di tipo bidirezionale il flusso degli FCT è inserito sempre ogni 8 token ricevuti, ma ora viaggia assieme ai pacchetti di dato nel senso opposto a quelli ricevuti.

Packet level

Per esso si rimanda alla precedente definizione di pacchetto, secondo lo standard IEEE 1355.

Higher level

Anche in questo caso si fa riferimento a quanto già visto in precedenza, ovvero data la particolare duttilità offerta dallo IEEE 1355 (e quindi dalla tecnologia DS-Link) circa le dimensioni dei pacchetti, essi possono essere utilizzati come “meccanismi”

di trasporto per altri standard come Ethernet, ATM, Fibre Channel etc. Per lo specifico DS-Link è stato pensato un protocollo ad alto livello denominato *Virtual Channel Protocol* [39] [38]. Con esso messaggi di qualsiasi dimensione dovrebbero essere suddivisi in pacchetti di dimensioni pari a 32 byte (dati), seguiti da un terminatore di tipo EOP e così sino all' ultimo pacchetto, del messaggio in toto, che sarà invece terminato da un EOM avente dimensioni variabili da 1 a 32 byte, in funzione della lunghezza del messaggio stesso.

Nel *Virtual Channel Protocol*, dopo un primo pacchetto dati non può essere spedito un ulteriore pacchetto, se prima non è stato ricevuto un pacchetto di risposta (ACK) formato dalla sola testa e da un terminatore di tipo EOP. Il protocollo garantisce in tal modo che nel caso di messaggi lunghi un singolo processo non monopolizzi il link di connessione.

In realtà questo protocollo era stato implementato sul transputer[9] T9000, prodotto dalla stessa SGS-THOMSON, che grazie ad un multiplexer comandato da un processore interno (il già menzionato VCP) ed un controllore DMA per ognuno dei 4 link di trasmissione-ricezione, permetteva la gestione di comunicazioni multi-processo. Infatti attraverso il multiplexer lo stesso link fisico veniva condiviso da più processi, ai quali era assegnato un certo tempo di lavoro, evitando così l'uso di numerose connessioni fisiche per ciascun processo: i canali di comunicazione così stabiliti venivano chiamati "canali virtuali" [10].

2.3 Il Packet routing switch STC 104

2.3.1 Tecniche di routing ed elementi funzionali

Abbiamo notato precedentemente che per la connessione di più dispositivi di comunicazione, impiegando interconnessioni di tipo *punto-punto*, il *routing switch* rappresenta l'elemento focale per l'instradamento dei messaggi. Lo switch utilizzato nel presente lavoro è stato lo STC 104 [35] prodotto dalla SGS-THOMSON, su singolo chip VLSI, realizzato in tecnologia CMOS, e che presenta 32 DS-Link per interconnessioni *punto-punto*, più altri 2 DS-Link dedicati al suo controllo e configurazione.

---

[9] Microcomputer con funzioni di calcolo, integrato su singolo chip ed avente un certo numero di link (4), dedicati alla comunicazione, per garantire interconnessioni con altri dispositivi di calcolo o altri transputer.
[10] Per informazioni più specifiche riguardanti il Virtual Channel Protocol ed il transputer T9000 vedi [8].

Per quanto riguarda la sua configurazione esso dispone di numerosi registri che, come vedremo nei capitoli successivi, permettono ampie variazioni dei diversi parametri nonché degli algoritmi di instradamento dei dati (o pacchetti).

Esso implementa delle tecniche che operano in modo congiunto: il *wormhole routing* che riduce le latenze di attraversamento dei pacchetti, e lo *interval routing* che minimizza le decisioni di instradamento. Nella tecnica *wormhole routing*, lo switch legge soltanto la testa del pacchetto: una volta presa la "decisione" per l'instradamento (*interval routing*), la testa è posta direttamente su un link d'uscita, ed il resto del pacchetto è spedito direttamente dall'ingresso all'uscita senza essere memorizzato[11] nello switch (come è invece il caso degli switch Ethernet): ciò implica, evidentemente, una bassa latenza.

Il pacchetto può quindi attraversare numerosi switch nel medesimo tempo, e la sua testa, nel passare lungo una serie di switch, crea un "circuito" temporaneo (*wormhole*), su cui fluiscono i dati veri e propri: non appena il terminatore del pacchetto (sia EOP che EOM) attraversa l'uscita di uno switch, il "circuito" svanisce parzialmente, per poi cancellarsi definitivamente nell'istante in cui il terminatore attraversa l'ultimo switch.

Questo metodo fa sì che la latenza di attraversamento dei pacchetti, oltre ad essere minimizzata, sia anche indipendente dalla lunghezza degli stessi.

Con la tecnica dello *interval routing*[12], lo switch legge la testa del pacchetto in ingresso e ne compara il contenuto con un insieme di valori, *interval*, ognuno dei quali può essere riferito ad una particolare uscita dello switch. Mantenendo questi riferimenti fissi e variando solo il contenuto della testa è possibile indirizzare il pacchetto a qualsiasi uscita dello switch. Se si è quindi nella condizione di dover trasferire l'informazione tra due dispositivi (DS-Link), una volta connessi allo switch ed impostati gli *interval* di quest'ultimo occorre solo immettere la "giusta" testa affinché il messaggio passi da un dispositivo all'altro. Inoltre lo switch permette l'utilizzo di un altro meccanismo chiamato *header deletion* che effettua la cancellazione della testa solo quando il pacchetto è in uscita da un link. In tal caso se un pacchetto in ingresso contiene 2 byte di testa, ed è attivo lo *header deletion*, una volta uscito dallo switch esso conterrà un solo byte di testa, come illustrato nella figura 2.9.

---

[11] Questo permetterebbe, ad esempio, di fare dei controlli sui dati che attraversano lo switch.
[12] Esso rappresenta effettivamento il processo che garantisce l' instradamento del pacchetto.

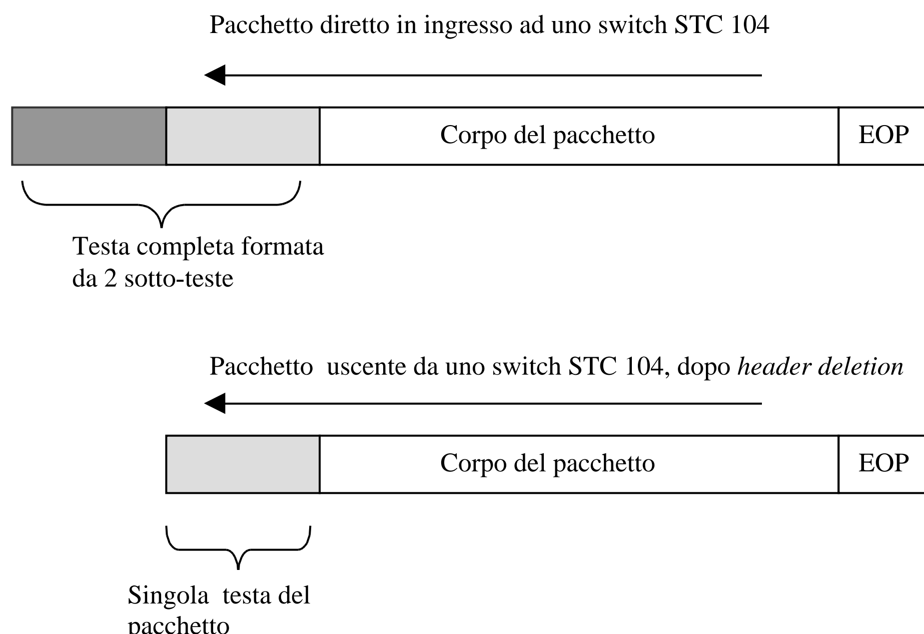


Figura 2.9: Effetto dello header deletion su un pacchetto avente due byte di testa

In questo modo, aggiungendo più *header* ed implementando lo *header deletion* si possono creare degli instradamenti molto articolati, costruire reti di switch con una gerarchia, aumentare considerevolmente il numero di canali di comunicazione e, non ultimo, diminuire il numero di *interval* diversi[13]. Purtoppo il prezzo che si paga è che aumentando il numero di *header*, si ha una quantità minore di informazione trasmessa a parità di tempo.

Nello STC 104 il flusso totale di dati provenienti dalle 32 porte d'ingresso attraversa una rete a stella ed arriva ad un unico punto di *routing*, il *crossbar switch*, (descritto più avanti) e da qui riparte verso 32 porte di uscita. Ogni singolo DS-Link dello switch implementa sia una porta d'ingresso che una di uscita, dei buffer e degli elementi con funzioni di routing. Questo insieme verrà chiamato *Link Slice*.

Ogni *Link Slice* presenta dei parametri di configurazione che sono modificabili attraverso dei registri (*configuration register*) ed accessibili dall'utilizzatore attraverso un'unità di controllo detta appunto *control unit*, e che li gestisce direttamente lasciando

[13] E' evidente che per garantire ad ogni dispositivo un indirizzo unico (in assenza di *header deletion*), occorrerebbero tanti *interval* diversi quanto è il numero totale dei dispositivi.

all’ utilizzatore uno spazio di configurazione pari a circa 28 kbits[14]. Gli elementi principali dello switch, esposti nella figura 2.10, sono:

- Crossbar switch 32_32
- Control unit
- 32 Link Slice

Il Crossbar switch è il “cuore” dello STC 104, è formato da una serie di switch elettronici e presenta 32 porte di ingresso ed altrettante di uscita (porte di I/O), inoltre presenta al suo interno un clock con frequenza variabile tra 30 e 50 Mhz .

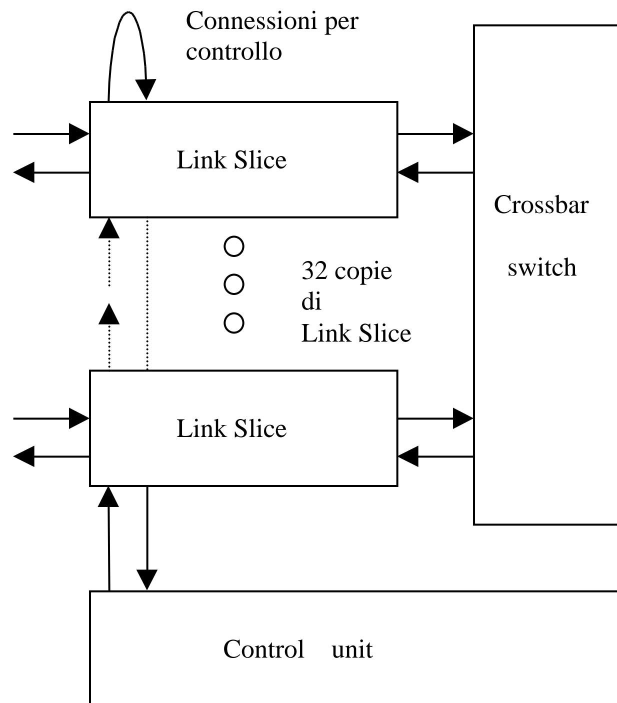


Figura 2.10: Elementi principali, costituenti lo switch STC 104

Una volta che il crossbar ha rilevato dal Link-Slice la richiesta per una precisa porta di uscita (*interval routing*), come essa è libera il crossbar realizza rapidamente una connessione tra ingresso ed uscita, facendo fluire i pacchetti di dati. Mentre la

[14] Pari al numero totale dei bit disponibili per ogni registro di tutti e 32 i *Link Slice*.

connessione rimane attiva è garantito il funzionamento del meccanismo di *flow control token*: ma all'arrivo di un token di terminazione la connessione viene annullata.

Nel caso che da più *Link Slice* arrivi la richiesta per la stessa porta di output il crossbar effettua un arbitraggio connettendo un sola porta di input alla volta, lasciando le altre in condizione di attesa fino a che la prima non abbia completato la transazione del suo pacchetto. L' arbitraggio è fatto secondo la tecnica *round-robin,* la quale assicura che tutte le porte di input saranno servite dopo un certo tempo.

Il crossbar switch inoltre implementa il meccanismo di *grouping* (che sarà analizzato in seguito) fornendo un altro livello di arbitraggio ma su gruppi di porte in uscita.

Vediamo ora in maggiore dettaglio la struttura di un *Link Slice.* Esso è decomponibile in due parti: dal DS-Link al crossbar (*input side*), e dal crossbar al DS-Link (*output side*).

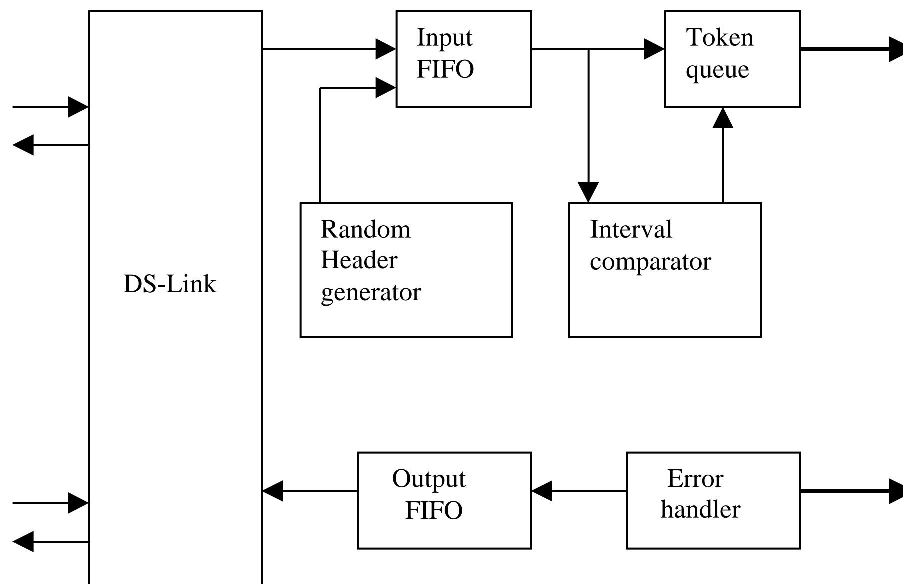


Figura 2.11: Elementi fondamentali del Link Slice di un STC 104.

I suoi elementi basici, visibili nella figura 2.11, possono essere raggruppati in quattro blocchi nello input side (*input* FIFO, *random header generator*, *interval comparator, token queue*) e altri due blocchi nello output side (*error handler, output* FIFO).

Input FIFO: il token proveniente dal DS-Link arriva nella *input FIFO*, subisce un solo ciclo di latenza (di clock del crossbar), dopodiché viene spedito fuori e la FIFO risulta così pronta ad accettarne di nuovi. Nel momento però che si presentano due pacchetti a due rispettive input FIFO, e che richiedono entrambi la stessa porta di uscita, allora una delle due FIFO va in stallo, continuando però ad accettare almeno altri 20 token dal DS-Link, una quantità pari alla sua profondità.

Random header generator: vedremo in seguito l'utilizzo dello, *universal routing*, algoritmo che permette di aumentare le prestazioni di flusso dati in casi particolari. Per il suo impiego occorre un elemento che è appunto il *random generator*, in grado di generare degli *header* pseudo-casuali. Esso fornisce uno o due byte di testa, che vengono aggiunti alla testa vera e propria del pacchetto, il cui valore è programmabile in una finestra:

$$base \quad header < base + range$$

dove *base* e *range* sono interi a 16 bit.

Interval comparator: il meccanismo dello *interval routing* è determinato da due elementi che sono appunto lo *Interval comparator* ed il *token queue*. Lo *Interval comparator* confronta ogni testa del pacchetto con una "tavola" di indirizzamento, di tipo programmabile, che associa ad ogni valore di *header* una certa porta di uscita.

Lo *Interval comparator* implementa due ulteriori sotto-elementi, *Invalid* e *Discard*, che forniscono delle garanzie per il routing. Il primo segnala pacchetti la cui testa non è confrontabile con nessun valore della "tavola", mentre il secondo garantisce la cancellazione della testa aggiunta dal *random generator*. La figura 2.12 fornisce un esempio di *interval routing*.

Token queue: esso interpreta il pacchetto uscente dallo *Interval comparator* e ne rivela errori nella struttura dei token. Il pacchetto esente da errori è allora passato al crossbar per essere indirizzato all'uscita opportuna, oppure cancellato nel caso in cui il *token queue* abbia rilevato errori.

Lo *output side* presenta invece due soli blocchi di seguito brevemente illustrati.

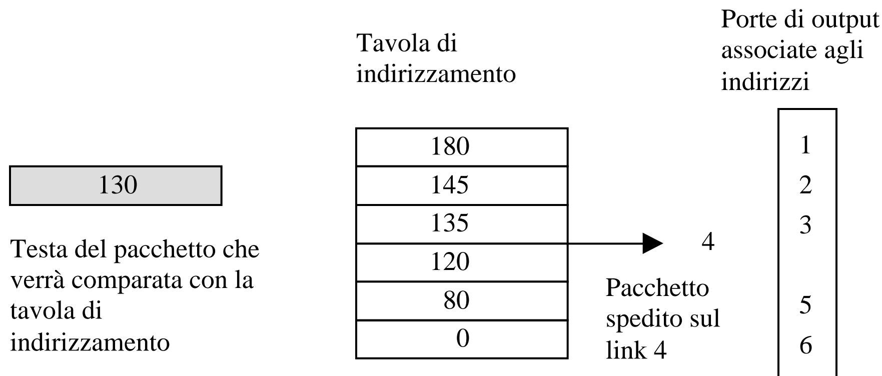


Figura 2.12: Meccanismo dello interval routing.

Error handler: nelle normali operazioni, esso prende semplicemente i dati dal crossbar e li spedisce alla FIFO di uscita, senza latenza alcuna. Nel caso che un errore sia stato rivelato dai precedenti blocchi, lo *Error handler* si porrà in condizione di stallo, o meno (per altri pacchetti provenienti dal crossbar) a seconda di come è stato programmato.

Nel caso in cui esso vada in stallo, tutti i pacchetti indirizzati al DS-Link relativo saranno completamente annullati .

Output FIFO: il suo meccanismo di funzionamento è perfettamente identico alla input FIFO, così pure la possibilità di acquisire nel suo buffer fino a 20 token. A differenza della *input*, la *output FIFO* implementa anche la funzione di *header deletion*.

Per quanto concerne le dimensioni complete dei buffer del singolo link dello STC 104 sono riportati, in *Tabella 2.4*, il numero di *token* per tutti i vari blocchi sin qui analizzati.

Input DS-Link	20
Input Link Slice FIFO	20
Token queue	3
Crossbar switch	4
Output Link Slice FIFO	20
Output DS-Link	3
Totale	70

Tabella 2.4: Dimensioni, in termini di token, dei buffer di un DS-Link completo.

Le frequenze di clock del crossbar switch sono variabili in un range compreso tra 30 e 50 Mhz: questo clock deriva da un phase-locked-loop (PLL) che moltiplica per un fattore selezionabile un clock esterno fisso a 5 Mhz.

Un secondo PLL genera invece un clock variabile in un range tra 10 e 100 Mhz, che viene impiegato per spedire i pacchetti in uscita da tutti i 32 DS-Link, mentre in ingresso il segnale di clock è ricostruito a partire dai segnali Data-Strobe stessi.

2.3.2 Registri di configurazione e proprietà dello switch

Lo STC 104 è un dispositivo altamente programmabile [35] i cui parametri di configurazione[15], possono essere impostati per mezzo di registri (*configuration register*) accessibili dall' utilizzatore attraverso la *control unit*.

La *control unit* ha due DS-Link dedicati: uno di essi deve essere connesso ad un *Controlling process*[16] mediante cui ricevere le istruzioni, o comandi, per i *configuration register,* l'altro deve essere utilizzato per propagare queste istruzioni ad altri possibili switch. Questo meccanismo di routing per il controllo di più switch verrà affrontato specificamente nel Capitolo 4, in cui si descriverà il lavoro svolto, imperniato sulla configurazione di uno o più switch STC104.

Osserviamo ora le caratteristiche di traffico dello switch: se consideriamo un pacchetto formato da un solo byte di testa ed un terminatore (*acknowledge packet*), esso è formato in totale da 14 bit. Con i DS-Link che lavorano a 100Mbit/s si ha che ogni bit ha un tempo di propagazione pari a 10 ns, e quindi un *acknowledge packet* occupa il link per una durata pari a 140 ns. Se consideriamo una trasmissione di tipo bidirezionale occorre ricordarsi del meccanismo, a basso livello DS-Link, del *flow control token* ovvero bisogna tener conto di 4 bit dello FCT; in questo caso il link sarà occupato per un tempo medio maggiore, e cioè per 140+40 = 180 ns.

Poiché il nostro switch possiede 32 DS-Link bidirezionali ed ammettendo che su tutti link siano inviati *acknowledge packet*, il totale di questi *packet* in un secondo è pari a $32/(180 \quad 10^{-9}) \quad 178 \text{Mpacket/s}$. Questo calcolo è stato fatto però nelle condizioni in cui sullo stesso link debba viaggiare sia il dato che lo FCT ossia nel caso di trasmissioni bidirezionali. Se invece consideriamo trasmissioni unidirezionali, con FCT che viaggia

[15] Ovvero la programmabilità di tutti i blocchi visti in precedenza, più altre funzioni speciali.
[16] Un host con interfaccia DS-Link.

sul link opposto, si ha un flusso di *acknowledge packet* pari a 230Mpacket/s e quindi un aumento delle prestazioni del 22 % circa.

Un altro parametro importante è la latenza che impone lo switch su un pacchetto che in esso transita; poiché il packet switch STC 104 utilizza la tecnica del *wormhole routing*, la latenza che esso impone ai pacchetti è legata, sostanzialmente, al tempo necessario alla lettura dello *header* ed il successivo instradamento. Per determinarne il valore della latenza occorre considerare che lo STC 104 ha due differenti regimi di clock: il clock del crossbar switch ovvero "system cycle" (fissato ad esempio a 50 Mhz), e quello del DS-Link, "link cycle" variabile da 10 a 100 Mhz.

Nel caso ad esempio di 2 byte di testa la latenza del DS-Link è quantificabile [42] in 4 system cycle + 17 link cycle in ingresso ed 1 system cycle + 22 link cycle per la uscita. A questi occorre aggiungere il contributo del singolo *Link Slice*, pari a 9 system cycle, quindi un totale di: 14 system cycle + 39 link cycle. Nella Tabella 2.5 il computo delle latenze considerate viene suddiviso per elementi del *Link Slice*.

Input Link Slice FIFO	1 system cycle
Token queue	4 system cycle
Crossbar	3 system cycle
Output Link Slice FIFO	1 system cycle

Tabella 2.5: Suddivisione dei vari system cycle.

Con le latenze precedentemente determinate, con il clock del crossbar switch a 50 Mhz e quindi un system cycle pari a 20 ns, con 100Mhz per il clock DS-Link cioè un link cycle pari a 10 ns, si ha una latenza totale di 670 ns.

Infine grazie all'implementazione di due metodi, lo *adaptive grouping routing* e lo *universal routing*, lo STC104 può garantire alte prestazioni nel contesto di reti formate da più switch DS-Link.

Lo *adaptive grouping routing* permette di assegnare un certo numero di porte di uscita come un unico gruppo: ad un unico interval della tavola degli indirizzamenti corrisponde un insieme di link di uscita e non più uno solo. In questo modo i messaggi con medesima testa di indirizzamento sono diretti ora non ad un singolo ma ad un

gruppo di link, con l' effetto che se uno dei link è occupato il messaggio verrà instradato dallo switch sul primo link libero del gruppo.

In tal modo la contesa sulla porta di uscita risulta essere drasticamente ridotta, sebbene dipenda dal numero dei link formanti il gruppo e dalle quantità di messaggi che richiedono contemporaneamente la stessa uscita. La figura 2.13 mostra un insieme di 4 switch connessi sfruttando il *grouped adaptive* più un altro dispositivo DS-Link con una singola connessione.

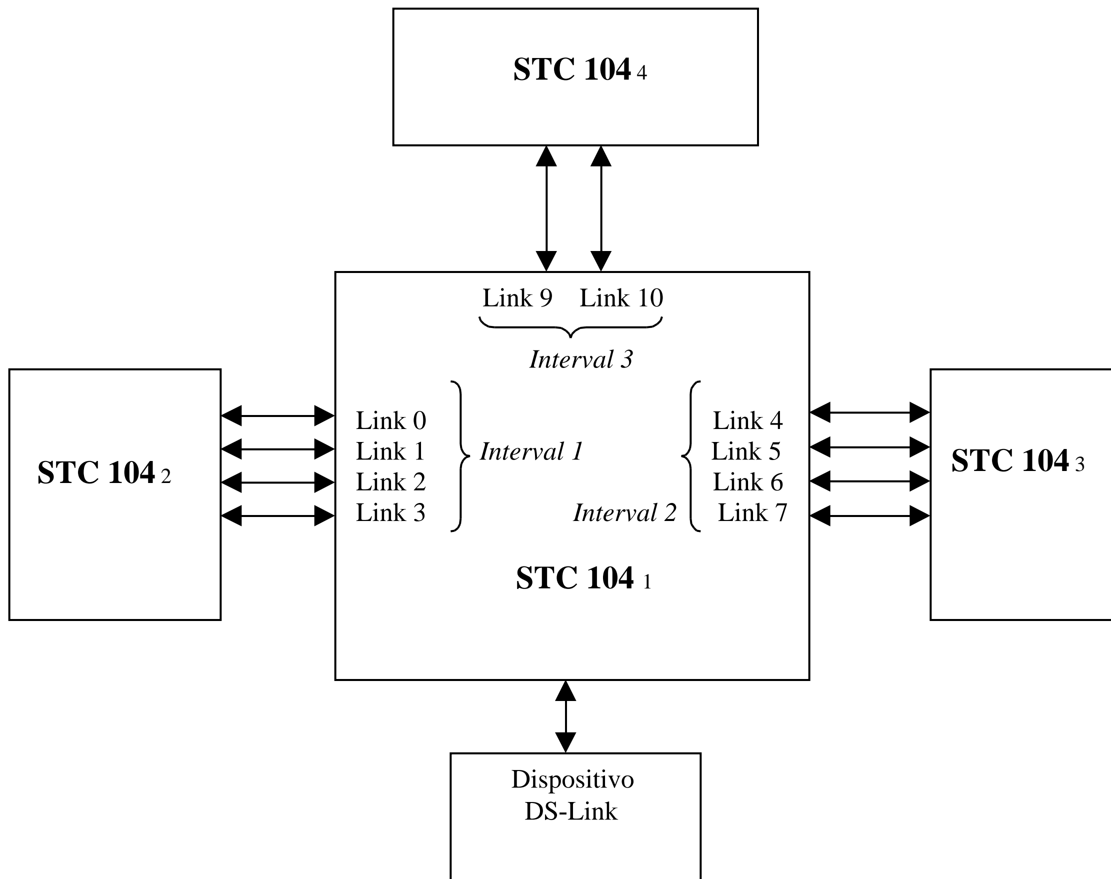


Figura 2.13: Implementazione dello adaptive grouping, sui link di connessione tra 4 switch. Per andare ad esempio dallo switch STC1041 allo switch STC 1042, tramite tutti e 4 i link di connessione, è sufficiente intestare i pacchetti con un header 0.

L' impiego dello *adaptive routing* risulta molto efficace nell'uso di architetture di switch di tipo *multistage-indirect network* ed incrementa il flusso di traffico informativo.

L' altro metodo, lo *universal routing*, chiamato anche *two-phase routing*, permette di evitare o comunque diminuire problemi di contesa link nel caso di reti di switch più o

meno complesse. Infatti ad ogni pacchetto in arrivo ad una porta dello switch, il random generator aggiunge una testa casuale, secondo lo schema già visto (la prima fase del routing), il cui valore deve essere in un range compatibile con le tavole di indirizzamento degli altri switch che si vogliono raggiungere. Raggiunto in modo casuale uno switch, la testa random del pacchetto subisce la cancellazione (seconda fase di routing), e così il pacchetto viene indirizzato all'uscita finale relativa alla "vera" testa.

Entrambe le tecniche e le relative modalità di programmazione dello switch verrano riprese più avanti quando ci occuperemo di studiare i flussi di dati di alcune topologie di reti di switch.

2.4 L' adattatore parallelo / DS-Link STC 101.

Per utilizzare la tecnologia DS-Link su piattaforme quali PC, Workstation et simili, la SGS-Thomson ha realizzato un dispositivo, Parralel DS-Link Adaptor STC 101, che permette l'interfacciamento [34] ad alte prestazioni tra il bus di I/O del sistema e la tecnologia stessa. Nello sviluppo del lavoro sono state utilizzate delle schede elettroniche che presentano lo STC 101 interconnesso tramite apposito controllore su bus locale PCI[17].

Questo dispositivo è realizzato in tecnologia VLSI e converte i dati provenienti dal bus locale da formato parallelo a seriale DS-Link e viceversa. Le sue caratteristiche principali sono:

- accesso al bus parallelo sia a 16 che a 32 bit.
- Possibilità di garantire trasmissioni bidirezionali DS-Link, con velocità di 100 Mbit/s su ciascuna direzione.
- Pacchettizzazione secondo le specifiche dello strato logico DS-Link e, per garantire ottimizzazioni del processo, impiego di memorie FIFO (64 byte in ingresso e 64 byte in uscita).
- Possibilità di utilizzare dei segnali d'interrupt, per gestire in modo efficiente i messaggi da trasmettere o ricevere.

[17] Bus di I/O che può garantire il trasferimento sia a 32 che 64 bit [26], con frequenze di clock variabili da 25 a 33 MHz (anche se sviluppi futuri prevedono versioni a 64 bit e 66 MHz di clock).

L'adattatore parallelo può essere utilizzato in due differenti modalità: con pacchettizzazione abilitata (è lo stesso STC 101 che implementa la struttura del *protocol stack*, fino al livello di pacchetto), oppure con pacchettizzazione disabilitata, detta anche *transparent mode*, che permette di effettuare trasmissioni tra dispositivi DS-Link senza lo impiego di un routing switch (non c'è più la testa e il terminatore).

Con la pacchettizzazione abilitata si possono trasmettere pacchetti di qualsiasi dimensione, ma qualora si voglia suddividere un messaggio di grosse dimensioni in una serie di messaggi più piccoli, lo STC 101 deve essere "pilotato" opportunamente: è evidente che una tale operazione risulta efficiente, in termini di latenze, solo se realizzata con macchine "intelligenti"[18]. Il funzionamento in *trasparent mode* può essere utile per effettuare delle semplici prove di trasmissione DS-Link.

Lo STC 101 garantisce un funzionamento in quattro modalità differenti: *16 bit processor interface* (figura 2.14a), *32 bit processor interface* (figura 2.14b*)*, *16 bit processor interface + token* (figura 2.14c*), 16 bit processor interface + multiplexed token* (figura 2.14d*)*.

Si precisa subito che queste funzionalità possono essere fattivamente realizzate solo con delle connessioni hardware ad hoc, e quindi da prevedere in fase di progettazione della scheda elettronica: nel caso dello hardware specificamente utilizzato in questo lavoro, è stato possibile utilizzare solamente la prima modalità e la *16 bit processor interface + token*.

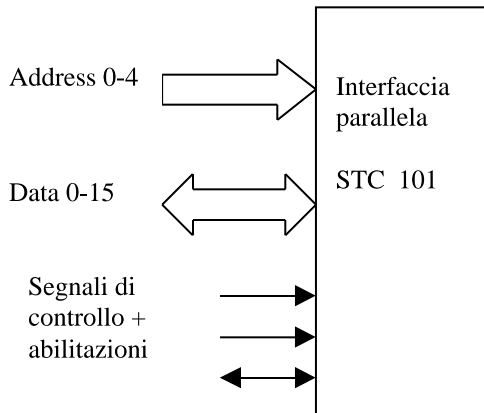
Con la funzionalità *16 bit processor interface*, l'accesso al bus parallelo di I/O avviene direttamente con dati a 16 bit, mentre con *32 bit processor interface* l'accesso è ovviamente a 32 bit. Nella modalità *16 bit processor interface + token* ci sono 2 porte ad 8 bit dedicate unicamente alla trasmissione-ricezione dati. Grazie a queste 2 porte (*token port*) i bit di programmazione inviati allo STC 101 ed i dati d' informazione, viaggiano su percorsi indipendenti: i primi accedono al bus a 16 bit, i secondi, ripetiamo, sulle *token port*. L'utilizzo della modalità *16 bit processor interface + token,* permette di raggiungere alte velocità di trasmissione.

Come per lo switch STC104, la programmazione dello STC101 avviene tramite dei registri interni a 16 bit[19], i quali possono essere di sola lettura, sola scrittura o di lettura/scrittura.

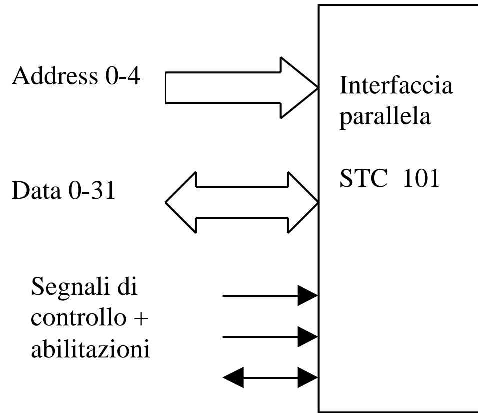
---

[18] Nuovamente citiamo il caso dei processori VCP presenti nel T9000.

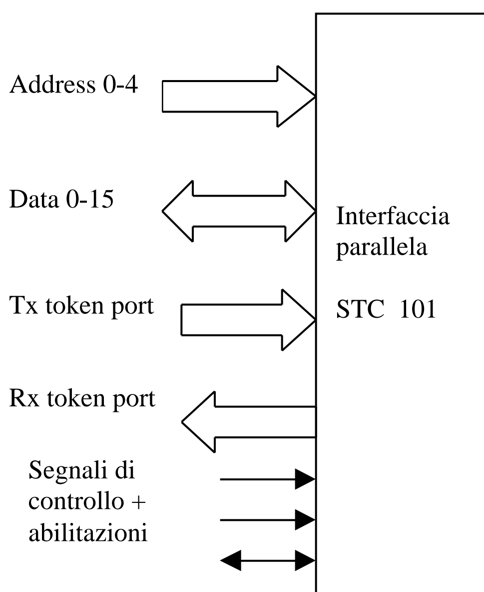
[19] Nelle modalità, senza uso di token, ci sono dei registri per trasmettere e ricevecere i pacchetti dati, le cui "dimensioni" arrivano comunque fino a 32 bit.



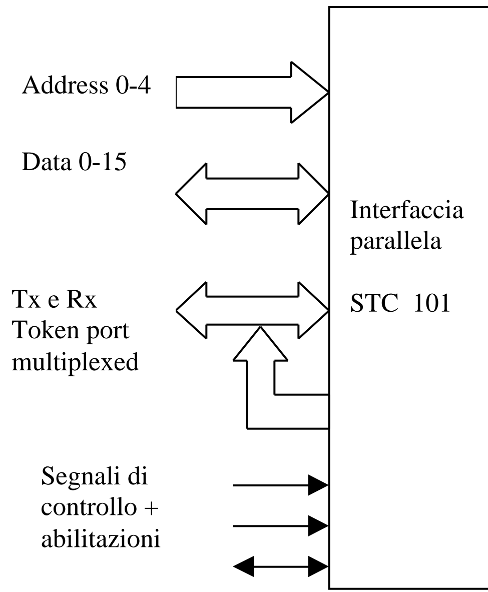
(a) 16 bit processor interface



(b) 32 bit processor interface



(c) 16 bit processor interface + token



(d) 16 bit processor interface + token multiplexed

Figura 2.14: Modalità operative dello STC 101. (a) 16 bit processor interface, (b) 32 bit processor interface, (c) 16 bit processor interface + token, (d) 16 bit processor interface + token multiplexed.

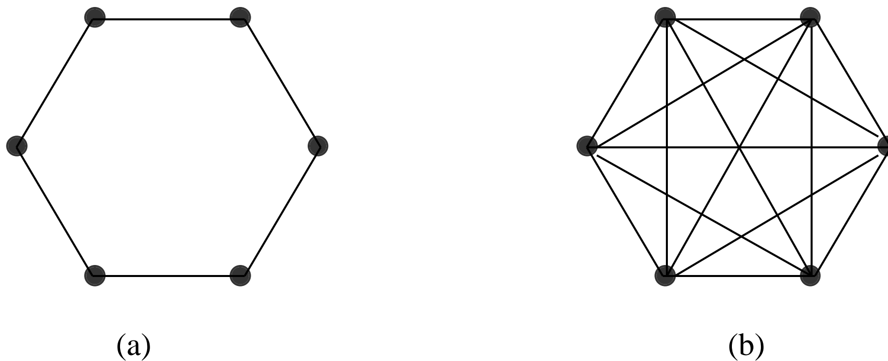
Capitolo 3

Le reti di interconnessione

3.1 Introduzione

Le larghezze di banda e l'alta connettività richieste dal trigger di 2° livello di ATLAS determinano in modo inequivocabile l' impiego di reti a commutazione di pacchetto[1] con elevato numero di nodi[2], le cui prestazioni sono tuttora diffusamente studiate nel contesto dello scambio di messaggi in architetture parallele e multi-processore. L'elemento topico nella realizzazione di una rete di comunicazione ad alte prestazioni, è la bassa *latenza* nel trasferimento dell' informazione da nodi sorgenti a nodi destinazione. Contribuiscono ad una bassa latenza non solo le caratteristiche tecnologiche degli elementi ma anche l'architettura e le dimensioni della rete. Descriviamo ora i parametri che caratterizzano una rete.

Due nodi sono *contigui* se c'è un link che li connette direttamente ed il *grado* di un nodo è rappresentato dal numero di quanti ad esso sono *contigui*. Il *diametro* di una rete rappresenta invece il più lungo tra i cammini minimi [10]: nel caso di una topologia *ring*, mostrata in figura 3.1, avente **n** nodi, il *diametro* è pari ad **n/2**, mentre per una topologia *fully connected* esso rimane costante e pari a **2**.



(a) (b)

Figura 3.1: Topologie di reti. a) ring, b) fully connected. I punti rappresentano i diversi nodi, mentre con le linee sono indicati i link di connessione.

[1] Nella commutazione di pacchetto non sussistono connessioni fisiche, al contrario della commutazione di circuito, tra trasmettitore e ricevitore. Come abbiamo visto nel paragrafo 2.1 il percorso tra trasmettitore e ricevitore viene attuato da un dispositivo, il *routing switch*, in base al valore presente nella testa del pacchetto.

[2] Con esso indicheremo sia un processore, una memoria oppure uno switch.

D’ altra parte però una connessione a *ring* può essere espansa indefinitamente ed il suo *grado* rimane costante, mentre ciò non è più vero per una *fully connected*.

Altre interessanti topologie sono: *ipercubo*, *mesh*, *albero*, *stella*, *multistage interconnect networks (MIN)*. Nella figura 3.2 sono illustrati un *ipercubo*, a 3 e a 4 dimensioni. Nella costruzione di questa topologia si parte da un elemento basico, ad esempio 2 dimensionale, e si congiungono i 4 nodi (originali) con altri 4 nodi (copie) ottenendo così un cubo (o ipercubo 3D). Con la medesima procedura si costruiscono *ipercubi* di dimensioni maggiori.

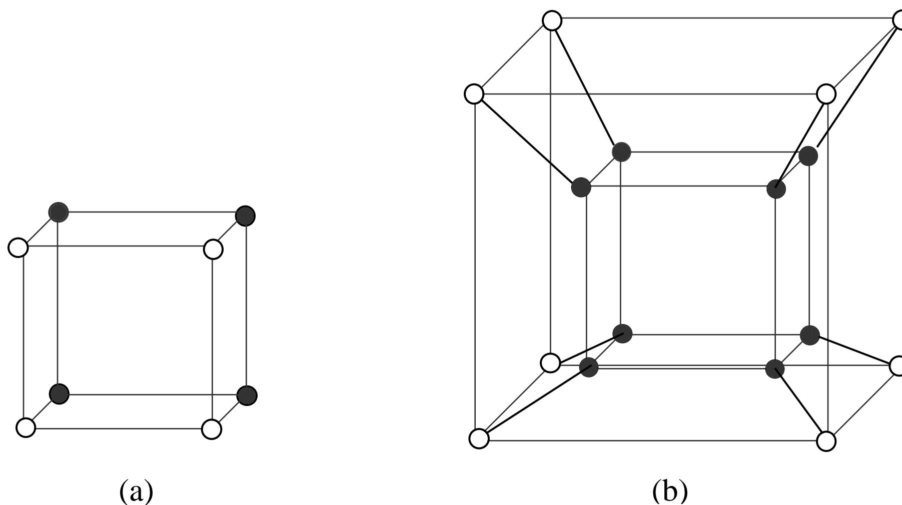


Figura 3.2: a) ipercubo 3D, b) ipercubo 4D. In nero sono contrassegnati i nodi dello ipercubo n-1 dimensionale, mentre in bianco quelli della copia.

Si può dimostrare che un ipercubo avente 2^n nodi, presenta un diametro **n** ed un grado pari ad **n**. La *scalabilità* di questa topologia è completa, nel senso che il volume del flusso dati su un singolo link rimane costante al variare del numero dei nodi. Inoltre i suoi nodi hanno la proprietà di essere *simmetrici*, ossia ognuno di essi ha le medesime caratteristiche: chiaramente tale proprietà è da intendere valida per tutte le architetture.

Le altre topologie possibili, *mesh*, *stella* ed *albero* sono riportate in figura 3.3.

Capitolo 4

Controllo e configurazione di reti di switch

Nel presente capitolo si affronterà inizialmente il meccanismo con cui si possono controllare uno o più switch STC104 per poi descrivere come è stato realizzato il lavoro che permette di gestire tali dispositivi tramite delle schede originali.

Infine verrà illustrata una tecnica per sviluppare del software che permetta di accedere ai dispositivi periferici, in un sistema operativo UNIX compatibile.

4.1 L' insieme dei comandi e loro propagazione

La presenza di registri (*configuration register*) all'interno del packet switch STC104, come asserito nel paragrafo 2.3.1, permette di ottenere un'ampia duttilità d'impiego del dispositivo stesso. Si è sottolineato inoltre che l'accesso all'insieme dei registri da parte dell'utilizzatore avviene con l'invio di opportuni comandi tramite una connessione DS-Link diretta alla *control unit* (la sezione di controllo del dispositivo); questo particolare link è denominato **Clink**0.

La *control unit* è provvista di un altro link, **Clink**1, con il quale è possibile propagare i segnali di comando (spediti sotto forma di pacchetti) ad altri switch tra loro interconnessi mediante la tecnica di routing, secondo la quale ogni dispositivo deve essere "marcato" in modo univoco e senza ambiguità per rendere possibile il suo riconoscimento dagli altri. Per questo motivo il pacchetto di comando contiene una testa che indica l'esatto switch a cui deve essere inviata la serie di istruzioni per i *configuration register*: se il pacchetto che giunge ad un STC104 contiene una testa che non gli compete, il dispositivo lo lascerà passare automaticamente, attraverso il **Clink**1, per farlo così proseguire verso lo switch di destinazione. Sia il **Clink**0 che il **Clink**1 sono denominati *control link* e presentano le stesse caratteristiche elettriche degli altri 32 link, indicati invece come *data link.*

Per la propagazione dei comandi si può impiegare una semplice connessione a *daisy-chaining* tra i *control link* dei dispositivi. Nella figura 4.1 è riportato un esempio di

funzioni di comunicazione trasferendo o ricevendo i dati su due semplici variabili, anziché inviare o ricevere i dati attraverso delle aree di memoria dedicata: questa strategia è stata adottata con il solo fine di valutare le prestazioni, in termini di velocità, sia della tecnologia DS-Link implementata su bus locale PCI, che delle reti di switch.

Poiché il *device driver* interagisce direttamente con il kernel è possibile far gestire a quest'ultimo [31] [22], nella fase di comunicazione, o altri processi o solamente le trasmissioni DS-Link. Nel primo caso si introduce un tempo di attesa per la comunicazione DS-Link, durante il quale il kernel può operare l'attività di "schedulazione" sugli altri processi; abbreviando tale tempo si riesce ad aumentare la priorità della comunicazione DS-Link rispetto agli altri processi, incrementandone le prestazioni. Nella realizzazione delle comunicazioni DS-Link si è scelta proprio questa modalità di lavoro perché per sfruttare una particolare configurazione, come vedremo negli ultimi paragrafi, deve essere garantita l'attività delle trasmissioni attraverso la rete Ethernet. Tra l'altro da misure effettuate il guadagno in termini di prestazioni che si ottiene, bloccando gli altri processi che non siano comunicazioni DS-Link, è molto modesto.

Si riporta un breve esempio, solo di trasmissione, di una funzione **txrx_ds** (impiegata per una tecnica di misura, *round trip time*, che tra poco vedremo) utilizzata dallo spazio utente per trasmettere e ricevere in modo sequenziale dei pacchetti le cui dimensioni e quantità devono essere preliminarmente impostate; essa è implementata nel *device driver* attraverso uno dei vari *case* presenti nell'istruzione **pcidsl_ioctl** (vedi paragrafo 4.3).

Nel codice relativo alla **txrx_dsl** si è indicato in grassetto il parametro (`WAIT_FOR`) che modifica i tempi di attesa della comunicazione DS-Link, e quindi la priorità della funzione stessa.

```
case TXRX_DS:
{
       struct TXRX_ds ioctlTXRX;
       unsigned int start, stop, i, j, time;
       unsigned int Transmit;

     ret = verify_area (VERIFY_WRITE, (void *) arg,
                          sizeof (struct TXRX_ds));
     if (ret)
        return ret;
     ret = verify_area (VERIFY_READ, (void *) arg,
                          sizeof (struct TXRX_ds));
     if (ret)
        return ret;
```

```
            copy_from_user (&ioctlTXRX,(struct TXRX_ds *) arg,
                            sizeof (struct TXRX_ds));



    /* Inizializzazione variabile di trasmissione */
              Transmit = 0x1234567;
              start = stop = 0;

            rdtsc(start); /* start lettura del TSC */

            for (i=0;i<ioctlTXRX.cicli;i++)
            {
               /* Attivazione trasmissione */
               outw(TERMINATOR_ENABLE|EOP_TERMINATOR|
                    HEADER_ENABLE|HEADER_SELECT_0|
                    ioctlTXRX.numbyte,adr1|TX_SEND_PACKET);

               /* Check per trasmissione */
               do
               {
               }while(((inw(adr1|ISR))&
                            TX_FIFO_EMPTY)==0x0L);

                for(j=0;j<((ioctlTXRX.numbyte)/4);j++)
                      outl(Transmit, adr0);

                current->state = TASK_INTERRUPTIBLE;
                current->timeout = jiffies+WAIT_FOR;
                schedule();

             }
                 ..............................

               rdtsc(stop); /* stop lettura del TSC */

               ioctlTXRX.cnt = (stop-start);

               copy_to_user ((struct TXRX_ds *) arg,
                             &ioctlTXRX,sizeof (struct TXRX_ds));

               break;
           }
```

Nella prima parte del *case* relativo alla **txrx_dsl**() viene utilizzata due volte la funzione *verify_area()*; riferendoci sempre al paragrafo 4.2, questa sub-funzione controlla se vi sia area di memoria-kernel disponibile, per essere poi usata nel trasferire (`VERIFY_WRITE`) o ricevere informazione dallo spazio utente (`VERIFY_READ`): in questo caso *verify_area()* è usata in entrambe le modalità. Infatti nella **txrx_dsl()** è presente sia la istruzione *copy_from_user()*, con cui viene trasferito dallo spazio utente allo spazio

kernel il numero dei pacchetti da inviare e le loro dimensioni, che l'istruzione *copy_to_user()* con cui viene restituito allo spazio utente una misura del tempo impiegato dalla **txrx_dsl()**.

Infine con le assegnazioni:

```
current->state = TASK_INTERRUPTIBLE;
current->timeout = jiffies+WAIT_FOR;
schedule();
```

si è posto che il processo in atto possa essere interrotto (`TASK_INTERRUPTIBLE`), che deve andare in attesa per tempi che vanno dal centesimo di secondo (`jiffies`), imposto dal kernel, ad un valore predefinibile (`jiffies+WAIT_FOR`), e che altri processi possano essere "schedulati" dal kernel (`schedule()`).

Utilizzando opportunamente le diverse funzioni per la comunicazione, sono stati realizzati dei programmi di test tra più PC contenenti ognuno un'interfaccia PCI-DSLink. Uno di tali test, effettuato usando due soli PC e che verrà illustrato nel prossimo paragrafo, è basato sulla tecnica *round-trip-time* [40]; su ogni host è presente un processo che di volta in volta può essere trasmittente o ricevente. Nella prima fase uno dei due host trasmette all'altro un determinato flusso di dati, mentre nella seconda fase lo host che ha ricevuto i dati li ritrasmette al primo; per misurare la velocità di trasferimento dei dati viene preso il tempo, sul primo host, tra l'inizio della trasmissione e la fine della ricezione. Considerando la quantità di byte trasferiti (in una sola direzione) e la metà esatta del tempo intercorso, si perviene alla velocità del flusso dati tra i due host: in questo modo la misura non viene influenzata dal fatto che un host risulti più veloce dell'altro nella fase di ricezione, o di trasmissione.

5.2 Velocità di trasmissione delle interfacce PCI-DSLink

In questo paragrafo si riporteranno dei test di misura per studiare le prestazioni delle interfacce, e le massime velocità di trasferimento ottenibili.

Per garantire una certa uniformità nella misura, sono stati utilizzati due PC equipaggiati con microprocessore Pentium con uguale frequenza di clock (100 MHz), aventi le medesime caratteristiche tecniche: ossia stessa scheda madre, stesse periferiche

e controllori. Tale scelta è dettata dal fatto che diverse istruzioni contenute nel *device driver*, risentono delle caratteristiche hardware della macchina residente.

In questo ambito di misure le schede sono state impostate, via software, per funzionare in *Token-mode* (vedi par. 2.4 e 4.2.1), permettendo così di sfruttare la loro massima velocità operativa. Con il programma realizzato **test_mirror.c** sono state effettuate delle misure lavorando di volta in volta con pacchetti di dimensioni diverse (espresse in byte); dal momento che nel funzionamento in *Token-mode* è possibile leggere/scrivere parole di 32 bit (4 byte), le dimensioni dei pacchetti potranno avere solo valori multiplo di 4.

Sono state effettuate una decina di misure per 8 diversi tipi di pacchetto: 4096, 2048, 1024, 512, 256, 128, 64 e 32 byte. In ogni sessione di misura sono stati trasmessi e ricevuti 15000 pacchetti: questo alto valore permette di ridurre gli errori casuali. I tempi di comunicazione sono stati presi utilizzando un particolare registro a 64 bit del microprocessore Pentium chiamato TSC (*Time Stamp Counter*) [43]; in tale registro i bit vengono incrementati seguendo il regime di clock. Mediante una macro scritta in assembler i 32 bit meno significativi del TSC vengono trasferiti in una variabile: leggendo quest'ultima prima e dopo una funzione di comunicazione, si ha effettivamente il numero di cicli di clock intercorsi e quindi una misura, con precisione dipendente dal clock del sistema, del tempo impiegato nella fase di comunicazione.

Si riporta il codice della macro realizzata **rdtsc()** in cui *time* è la variabile utilizzata:

```
#define rdtsc(time) ( {__asm  (".byte 0x0f; .byte 0x31" \
        : "=eax" (time)  ) ; } )
```

Per quanto concerne gli errori associati alle velocità (calcolati attraverso la semidispersione massima), essi risultano di entità modesta; a ciò concorre, indubbiamente, il fatto che è stata utilizzata un'alta priorità nell'uso delle funzioni di comunicazione e quindi una sufficiente indipendenza da altri processi gestiti dal kernel. Nella figura 5.1 è riportato lo schema di collegamento tra i due PC.

Figura 5.1: Collegamento tra i due PC con raffigurati i due processi di trasmissione e ricezione; si noti che la trasmissione è half-duplex dal momento che i processi sono sequenziali.

Nella tabella 5.1 sono indicati i valori di velocità ottenuti in funzione delle diverse dimensioni dei pacchetti; si può osservare che per pacchetti di 4096 byte si raggiunge la massima velocità. Sapendo che la massima velocità di trasferimento delle informazioni su supporto DS-Link è di ~ 9.53 Mbyte/s, la velocità raggiunta di 9.56 Mbyte/s ci pone addirittura lievemente oltre il limite teorico. Probabilmente ci sono da considerare delle tolleranze nelle prestazioni massime della tecnologia, anche se ciò non è esplicitamente dichiarato dal costruttore.

Velocità (Mbyte/s)	**Dimensione pacchetti (byte)**
9.56 ± 0.03	4096
9.47 ± 0.04	2048
9.40 ± 0.02	1024
8.41 ± 0.02	512
7.21 ± 0.01	256
5.14 ± 0.01	128
3.27 ± 0.01	64
1.81 ± 0.02	32

Tabella 5.1: Valori delle velocità misurate per diversi tipi di pacchetti.

Si noti che le prestazioni delle schede diminuiscono riducendo le dimensioni dei pacchetti. Questo andamento è da imputare sostanzialmente a due fattori: il primo legato

al fatto che le dimensioni della testa, per piccoli pacchetti, cominciano ad avere rilevanza sul corpo completo, il secondo, e forse il più importante, dovuto al controllo software introdotto per evitare la saturazione delle FIFO e che nel programma deve essere ripetuto per ogni pacchetto. Quest'ultimo è evidentemente legato al kernel (tramite il *device driver*) ed i ritardi introdotti potrebbero essere minimizzati solamente con una sua implementazione nello hardware: in sostanza ciò che avviene sulle schede di comunicazione Ethernet.

Per meglio visualizzare le velocità di comunicazione dell'interfaccia, nella successiva figura 5.2 sono riportati in grafico i valori della precedente Tabella 5.1.

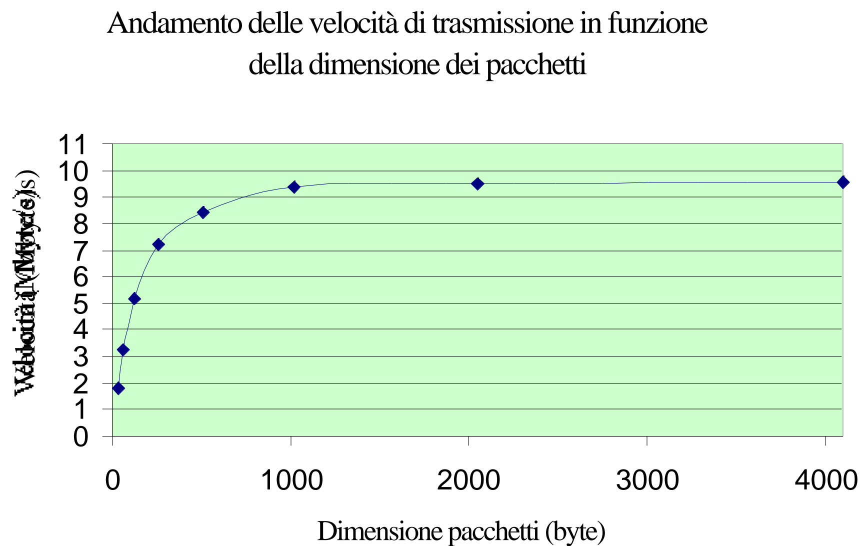


Figura 5.1: Curva rappresentativa della velocità di trasmissione tra PC1 e PC2 per diverse dimensioni dei paccheti di dato.

Ovviamente la precisione con cui sono state prese le misure è forse eccessiva, dal momento che cambiando configurazione del sistema e cioè altri modelli di PC, le velocità di trasmissione variano notevolmente. È però pur vero che le massime prestazioni non potranno mai superare i limiti oggettivi della tecnologia DS-Link, e tali valori sono stati ampiamente raggiunti nell'attuale configurazione.

5.3 Verifica del modello per la contesa nello switch

Nel paragrafo 3.3 è stato proposto un modello grazie al quale si forniscono gli strumenti per studiare il flusso informativo in diverse architetture di packet switch; è infatti attraverso l'uso di tale modello che siamo giunti nel paragrafo 3.4 alla determinazione teorica del *throughput* nelle topologie *banyan* e *Clos*.

La verifica sperimentale del modello è ovviamente legata alle disponibilità hardware, ovvero 4 interfacce PCI-DSLink dotate ciscuna di un singolo link. Sfruttando le possibilità della tecnologia DS-Link di permettere comunicazioni bidirezionali, ogni singola scheda, come già visto nel par. 3.4, ha la duplice funzionalità di sorgente o destinazione dei dati. In realtà dalle prove effettuate si riscontra che i ritardi introdotti dalle necessarie operazioni software, dallo host stesso e, non ultimo, dal meccanismo bidirezionale, non forniscono dei risultati apprezzabili.

Il problema è stato allora affrontato utilizzando le schede con una singola funzionalità: o sorgente o destinazione dei dati. Si sottolinea comunque che la configurazione precedente sarà utile nel contesto di misure per le reti di packet switch.

Le interfacce PCI-DSLink sono state inserite in 4 PC: due di essi sono quelli già utilizzati nel precedente paragrafo, con medesime caratterische, mentre gli altri due sono di prestazioni diverse (clock a 90 MHz e 166 MHz). Per agevolare la loro individuazione chiameremo PC1 e PC2 i precedenti host con clock a 100 MHz, mentre con PC3 lo host a 90 MHz ed infine PC4 quello a 166 MHz.

Gli host PC1 e PC2 sono stati scelti come sorgenti dei dati, mentre PC3 e PC4 come destinazioni; è stato inoltre generato del traffico con impostazione casuale delle destinazioni, e questo proprio in accordo con il modello. Le possibili "fonti" di generatori di numeri casuali (i valori da immetere nelle teste dei pacchetti) erano le funzioni ANSI C presenti nel complilatore GCC oppure lo switch stesso: tra le due non sono state riscontrate differenze in termini di prestazioni[1]. Per utilizzare il generatore di numeri casuali, **rand(),** del compilatore, è stata realizzata una semplice funzione chiamata **num_rand**().

---

[1] Sono stati fatti dei test sullo switch immettendo le teste random nei pacchetti, con entrambi i generatori, e non ci sono state variazioni nelle misure effettuate. Per questo motivo si vedrà più avanti che nella verifica delle diverse topologie, è stato indistintamente utilizzato o l' uno o l' altro generatore.

```
int num_rand(int range, int base)
{
     int val = 0;

     val = base + (int)((float)(range)*rand()/(RAND_MAX));
     return val;
}
```

Nella figura 5.2 è riportata la distribuzione su 5000 campioni dei numeri casuali compresi in un intervallo da 0 a 1000: come si può osservare tali numeri sono distribuiti in modo sufficientemente uniforme e quindi accettabili per i nostri scopi. In questo contesto di misure i tempi di comunicazione sono stati rilevati senza sfruttare la tecnica del *round trip time*, perché nel modello per la contesa sono previste comunicazioni unidirezionali: da sorgenti a destinazioni o viceversa ma non entrambi.

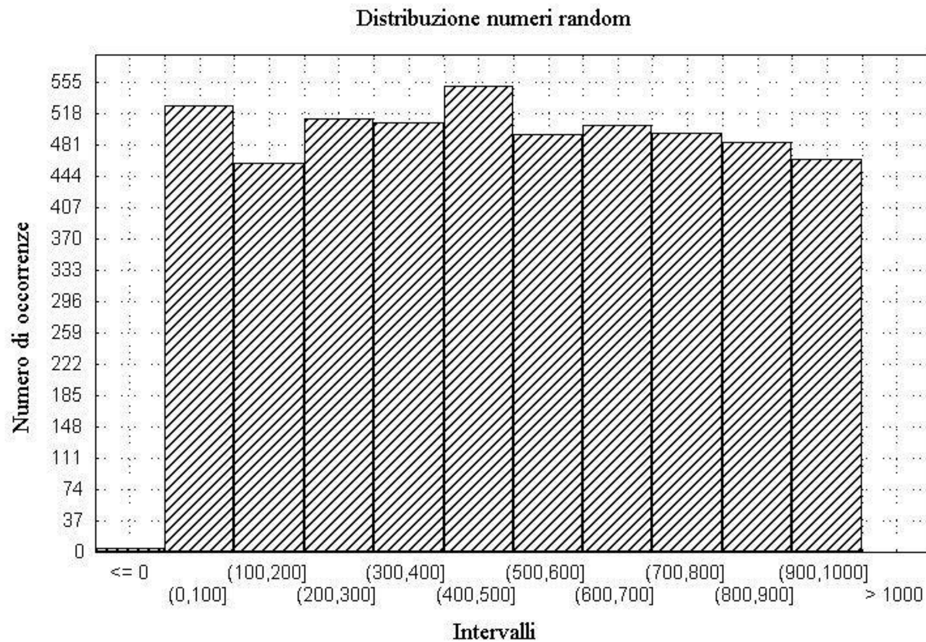


Figura 5.2: Distribuzione dei numeri da 0 a 1000 forniti dal generatore random in 5000 diversi campioni. In ascissa sono riportati i diversi intervalli tra 0 e 1000, mentre in ordinata i conteggi delle loro occorrenze.

Poiché le 4 schede PCI-DSLink sono state tutte utilizzate nei test, lo switch è stato configurato mediante una scheda ISA-DSLink di valutazione (B108 della SGS-Thomson)

attiva sul sistema operativo MS-DOS [25]. Sfruttando un programma di test associato alla scheda, è stato possibile configurare lo switch con un file di configurazione precedentemente ottenuto dal compilatore NDL (vedi par. 4.2.2). Nella figura 5.3 è illustrato lo schema di collegamento dei diversi host attraverso un unico switch STC 104, controllato da una scheda B108.

Come test iniziale è stata misurata la velocità di trasmissione di PC1 e PC2, singolarmente, avendo impostato le teste dei pacchetti per un traffico random sulle due destinazioni e ponendo queste ultime, PC3 e PC4, in ricezione continua. Il valore rilevato sui due PC è stato leggermente diverso ed è stato preso facendo 10 misure diverse, per 15000 pacchetti di 2048 byte .

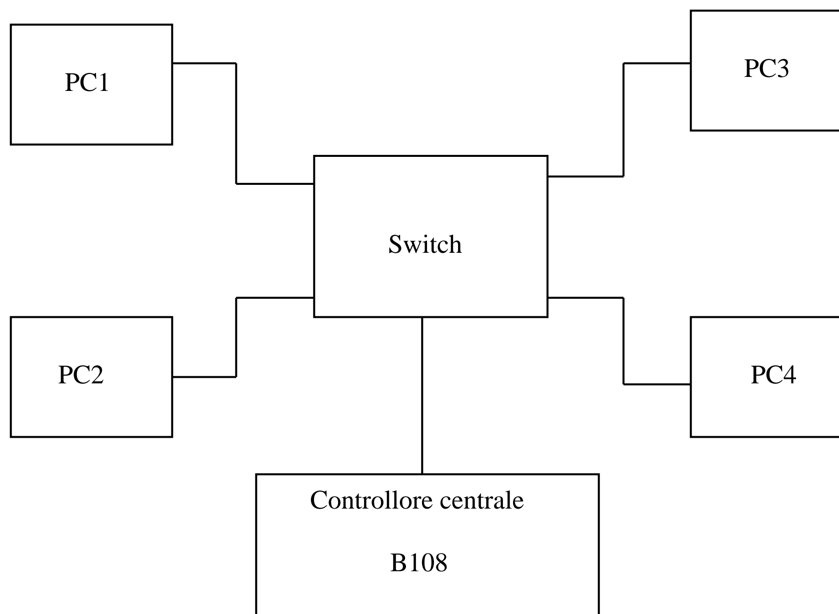


Figura 5.3: Collegamento dei 4 host al packet switch STC 104.

Di seguito si riportano le velocità ottenute, V(PC1) e V(PC2):

$$V(\text{PC1}) = 9.46 \pm 0.01 \ \text{Mbyte/s} \quad (1)$$

$$V(\text{PC2}) = 9.51 \pm 0.01 \ \text{Mbyte/s} \quad (2)$$

A questo punto le due sorgenti, PC1 e PC2, sono state poste in trasmissione contemporaneamente per misurarne le velocità in presenza di contesa sullo switch. I valori rilevati, sempre con lo stesso volume di dati precedente, sono mostrati di seguito:

$$V_{cont}\ (\text{PC1}) = 6.79 \pm 0.07 \text{ Mbyte/s}$$
$$V_{cont}\ (\text{PC2}) = 7.11 \pm 0.08 \text{ Mbyte/s}$$

Il modello teorico prevede che nel caso in cui lo switch presenti 2 ingressi e 2 uscite in regime di comunicazione, la probabilità di trovare un'uscita occupata sarà data dalla relazione (3) del par. 3.3:

$$P(occupata) = 1 - (1 - \frac{1}{2})^2$$

utilizzando la (4) dello stesso par. 3.3 si ottiene una velocità teorica per la contesa

$$V'_{cont}\ (PCx) = P(occupata)\quad V(PCx) \qquad (3)$$

dove *V(PCx)* è una delle due velocità di trasmissione, senza contesa, precedentemente calcolate. Sostituendo nella (1) i valori citati si ottiene:

$$V'_{cont}\ (\text{PC1}) = 7.10 \pm 0.01 \quad \text{Mbyte/s}$$
$$V'_{cont}\ (\text{PC2}) = 7.13 \pm 0.01 \quad \text{Mbyte/s}$$

Confrontando questi ultimi valori teorici con quelli sperimentali, derivati dalla contesa nello switch, si ha uno scarto di circa il 4-5% che ci permette di convalidare ed accettare il modello teorico. Ovviamente per una maggiore attendibilità del modello sarebbero necessarie ulteriori prove con un numero maggiore di sorgenti e destinazioni.

Si è pensato di verificare ulteriormente il modello della contesa in una configurazione di due switch disposti in cascata, come illustrato in figura 5.4. In tale condizione, giacché le uscite del primo switch costituiscono al tempo stesso gli ingressi del secondo, usando la relazione (9) del par. 3.4, si ha:

$$P_2(occupata) = 1 - (1 - \frac{1}{2})^{P_1\ 2}$$

dove P_1 è la probabilità già vista precedentemente.

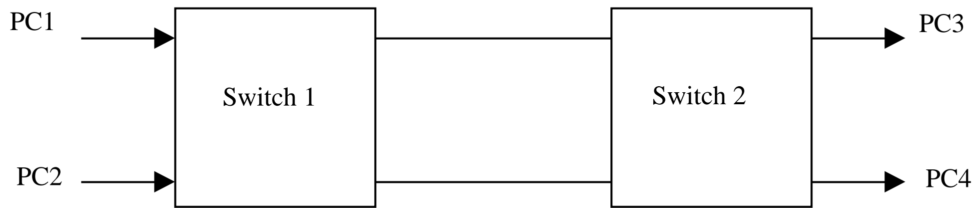


Figura 5.4: Switch disposti in cascata.

Sono state nuovamente misurate le velocità in assenza di contesa V(PC1) e V(PC2), ed i valori rilevati non si discostano da quelli ottenuti precedentemente. Il fatto che in presenza di un'altro switch le velocità non cambiano, indica anche che per pacchetti di dimensioni di 2048 byte il tempo di routing dello switch risulta trascurabile; si ricordi infatti che la latenza imposta dallo switch STC104 si ha solo per l'operazione di instradamento della testa del pacchetto, ed il corpo susseguente non subisce ritardi; la cosa ovviamente cambia qualora si considerino pacchetti di piccole dimensioni, in cui il tempo di routing ha il suo peso.

Nel caso in cui si ha possibilità di contesa negli switch le velocità misurate sono state:

$$V_{cont}\ (PC1)^* = 5.89 \pm 0.03 \quad \text{Mbyte/s}$$
$$V_{cont}\ (PC2)^* = 6.02 \pm 0.04 \quad \text{Mbyte/s}$$

mentre le velocità teoriche, inserendo nella (3) la probabilità P_2 sarebbero:

$$V'_{cont}\ (PC1)^* = 6.11 \pm 0.01 \quad \text{Mbyte/s}$$
$$V'_{cont}\ (PC2)^* = 6.13 \pm 0.01 \quad \text{Mbyte/s}$$

Ancora una volta gli scarti tra i due valori sono molto lievi, circa il 5%, confermando la validità e la buona approssimazione del modello teorico.

5.4 Prestazioni delle architetture banyan e Clos

Nei paragrafi successivi si farà riferimento ad una serie di misure effettuate sulle architetture viste nel paragrafo 3.4. Nell' ottica di uno studio puramente qualitativo sulle differenze di traffico tra diverse topologie, sono state utilizzate le interfacce nella duplice modalità sorgente/destinazione dei dati.

5.4.1 Il modello banyan con traffico random

Riportiamo in figura 5.5 il modello della *banyan network* di tipo 4   4 realizzata con switch di tipo 2   2. Il software realizzato, **txrx_rand_C104.c**, presenta una particolare procedura per la fase di ricezione.

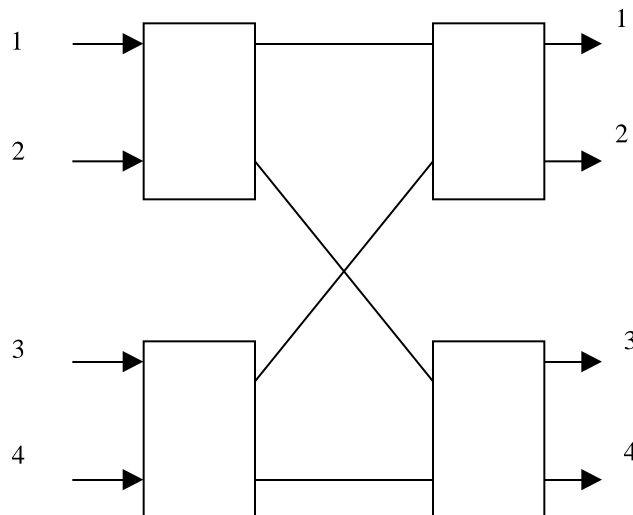


Figura 5.5: Topologia banyan a due stadi e di tipo 4   4 realizzata con 4 switch 2   2.

L' organizzazione del processo di comunicazione è il seguente:

- L' interfaccia, nella <u>fase sorgente</u>, trasmette un primo pacchetto con testa random affinché possa essere instradato casualmente, prima su una delle due uscite di uno switch del primo stadio e poi su una delle due uscite di uno switch del secondo stadio.

- Spedito il singolo pacchetto viene avviata la ricezione, fase destinazione, che dovrà contemplare, dato il tipo di traffico casuale, un minimo di 0 pacchetti (nessuna sorgente ha scelto questa destinazione) fino ad un massimo di 4 pacchetti (le sorgenti hanno scelto tutte la stessa destinazione).

I PC a disposizione, avendo caratteristiche diverse tra loro, hanno capacità diseguali di acquisire o trasmettere i dati (pacchetti) causando situazioni di stallo nel traffico; introducendo dei ritardi nel software degli host più veloci si diminuiscono le velocità di lavoro, evitando però la fase critica di blocco completo del sistema[2] (interfacce + switch). Si può osservare di seguito una parte del programma relativa alla fase di trasmissione (sorgente) e a quella di ricezione (destinazione).

```
    /* Impostazione(random) testa del pacchetto */
    Header[i] = (linknum1[i] + (linknum2[i] << 8));

    my_outl(Header[i] , c101_or_pld|TX_PACKET_HEADER_LOWER_0);
    my_outl(Header[i] /*per default*/,
            c101_or_pld|TX_PACKET_HEADER_UPPER_0);

    /* Fase sorgente */
    my_outldslink(DataTx, address, numbyte);
    delay_old(xxxx);

    while(Temp<5)
    {
             /* Check per Rx */
       if((my_inw(c101_or_pld|ISR)&(PACK_RECEIVED))!=0)
       {
           /* Fase destinazione */
          delay_old(xxxx);
          DataRx = my_inldslink(address, numbyte);
          Temp = 0;
       }

      Temp++;
}
```

La realizzazione della *banyan network*, mediante 2 dei 3 switch a disposizione e con la possibilità di suddividerli ognuno in sotto-switch di tipo 2 2, ha richiesto la programmazione di un particolare file di configurazione. Il primo stadio della topologia è stato realizzato con una sezione di entrambi gli STC 104 e così pure il secondo stadio: in

[2] Sono stati fatti diversi tentativi per arrivare al compromesso: minimo ritardo ma completa operatività del sistema.

quest'ultimo è stato utilizzato il generatore random presente negli switch. Nella programmazione del file di configurazione si è dovuto tener conto, ovviamente, che i possibili percorsi ed instradamenti garantissero il traffico random.

Sfruttando nuovamente la scheda ISA-DSLink per la configurazione degli switch, si è passati alla misura vera e propria del traffico sulla rete. È stata valutata la velocità su ogni singolo host, mettendo i rimanenti in una situazione di traffico continuo e spedendo da quello sotto test, lo stesso volume di dati: misure con 15000 pacchetti ciascuno di 2048 byte ripetute per 10 volte.

I risultati ottenuti sono stati i seguenti:

Host	Velocità (Mbyte/s)
PC1	3.27 ± 0.04
PC2	3.27 ± 0.05
PC3	3.53 ± 0.03
PC4	2.83 ± 0.05

Si può osservare che l'introduzione dei ritardi porta il PC meno "brillante" in termini di prestazioni, PC3, ad avere la più alta velocità di lavoro, mentre il più veloce, PC4, ha subìto una pesante diminuzione delle proprie prestazioni; tali ritardi però, ripetiamo, garantiscono che il sistema non entri in condizione di stallo.

5.4.2 Il modello Clos con traffico random

La configurazione di questa rete è stata leggermente più laboriosa della *banyan*, in quanto introducendo altri 2 sotto-switch, occorreva garantire lo stesso instradamenti di tipo random. Il modello è stato realizzato ancora una volta utilizzando 2 soli switch STC 104 e suddividendoli in modo opportuno. Nella figura 5.6 è illustrata la *Rearrangeable Clos network* a tre stadi nel caso 4   4 con indicato lo switch a cui appartiene ogni sotto-switch; tra parentesi sono riportati gli *interval* assegnati ai link di ogni sotto-switch.

L'impostazione degli *interval* è legata al meccanismo scelto per distribuire il traffico. È stato sviluppato un file di configurazione per utilizzare pacchetti con 2 byte di testa, in cui il valore dell' ultimo byte era impostato in modo casuale in un intervallo tra

26 e 27. Inoltre, in accordo con quanto affermato nel paragrafo 3.4, si è fatto uso dello *adaptive grouping routing* al primo stadio della Clos; sia al primo che al secondo stadio della rete viene effettuata la cancellazione della testa, mentre al terzo stadio viene invece usato il generatore random dello switch. Vediamo come è stata resa possibile la scelta di cammini casuali.

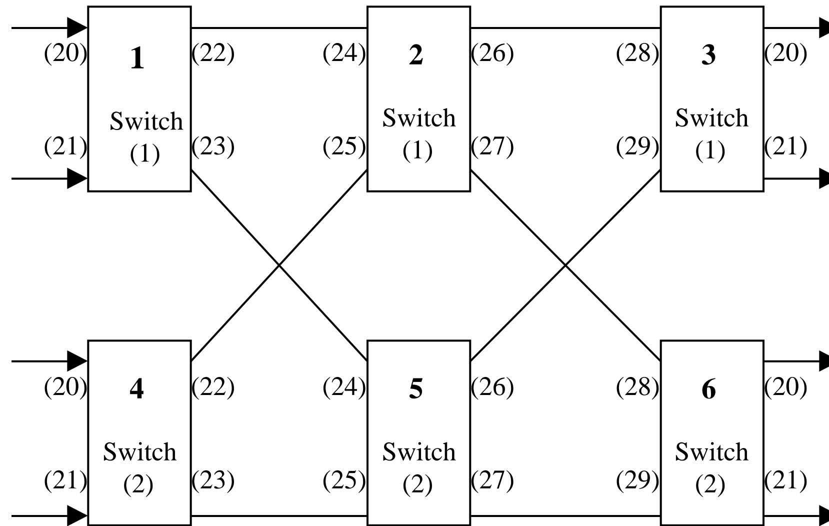


Figura 5.6: Rearrangeable Clos network a tre stadi di dimensione 4 4.

Una sorgente locata al primo stadio invia un pacchetto che, in base allo *adaptive grouping routing*, viene posto dal sotto-switch su uno dei due link che risulta libero: in questo stadio non si ha nessuna contesa ed il pacchetto, sottomesso a cancellazione del primo byte di testa, può andare al secondo stadio. Quello che era il secondo byte di testa con scelta random tra 26 e 27, diventa ora, al secondo stadio, il primo byte di testa; il pacchetto può così essere indirizzato al link associato allo *interval* 26 o al 27 e da qui ad uno dei sotto-switch del terzo stadio. Ricordando che al secondo stadio è attiva la cancellazione della testa, e ciò porterebbe il pacchetto ad esserne sprovvisto, viene aggiunta una testa random, con valore tra 20 e 21, dai sotto-switch del terzo stadio: lo instradamento è garantito ed il pacchetto può giungere ad una destinazione. Chiaramente con questo meccanismo è assicurata la possibilità che sia al secondo che al terzo stadio avvenga una contesa, proprio perché due diversi pacchetti possono richiedere la medesima uscita.

Per ottenere delle misure del traffico sono stati utilizzati gli stessi programmi della *banyan network*, con il medesimo volume di traffico.

I risultati ottenuti vengono di seguito indicati:

Host	Velocità (Mbyte/s)
PC1	3.3
PC2	3.3
PC3	3.5
PC4	2.8

Infine è stato studiato il modello della *Non-blocking Clos network*, che come visto nel paragrafo 3.4, permette di aumentare il numero dei possibili cammini all'interno degli stadi della rete, fornendo così un incremento delle velocità di traffico rispetto alla precedente *Rearrangeable Clos network*. Si riporta di seguito nella figura 5.7 lo schema della sua realizzazione mediante due soli STC104 partizionati in più switch di tipo 2 2 e 3 2.

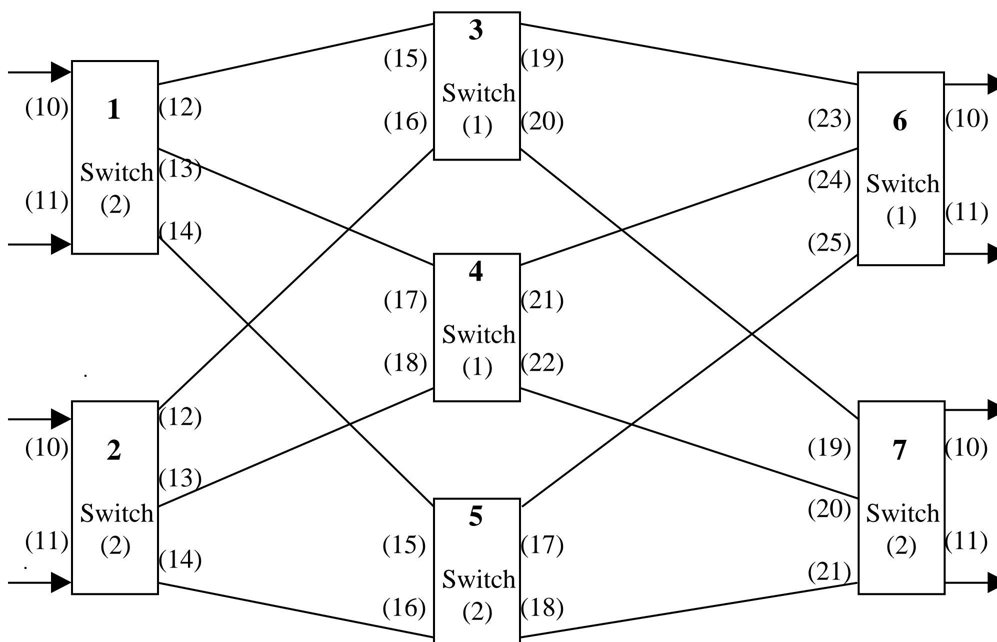


Figura 5.7: Non-blocking Clos network a tre stadi di dimensione 4 4.

Per questa topologia la fase di configurazione è stata sicuramente più laboriosa delle altre, proprio per garantire che tutti gli instradamenti fossero casuali. Anche in questo caso le velocità ottenute sui vari PC erano le medesime della precedente *Rearrangeable Clos network*. In questo caso, in base alla (10) del paragrafo 3.4, ci saremmo dovuti aspettare un leggero aumento del *throughput,* ma a causa dei già citati ritardi e del funzionamento bidirezionale delle interfacce tale minima variazione non si è riscontrata.

In sostanza i valori di velocità delle due reti di tipo *Clos* confrontati con quelli ottenuti precedentemente, *banyan network*, non mostrano grosse differenze; si può asserire quindi che quanto previsto nella trattazione teorica è stato confermato e che inoltre si è dimostrata indirettamente la validità e l'efficienza dello *adaptive grouping routing*.

5.5 Controllo da remoto delle comunicazioni DS-Link

Riprendendo un aspetto accennato in chiusura del Capitolo 4, si illustrerà ora il lavoro sviluppato per attuare una gestione da remoto delle sorgenti e destinazioni presenti in una semplice switching network, mediante un'interconnessione attraverso la rete Ethernet. Alcuni degli host su cui è alloggiata l'interfaccia PCI-DSLink, dispongono di una ulteriore scheda per la rete Ethernet (o più propriamente lo standard IEEE 802.3) a 10 Mbit/s con il necessario *device driver* per il sistema operativo Linux. È così possibile sfruttare l'interconnessione Ethernet per trasferire informazioni di controllo o di servizio, mentre i dati ad alta velocità possono viaggiare attraverso la rete di switch DS-Link. Nel presente lavoro per lo scambio delle informazioni attraverso la rete Ethernet è stato sfruttato il protocollo UDP (*User Datagram Protocol*), che è uno dei due di tipo end-to-end[3] forniti dall'architettura TCP/IP.

Questo approccio, di fornire cioè un' ulteriore meccanismo d' interconnessione, può essere utile per introdurre una possibile simulazione del controllo di architetture del 2° livello di ATLAS mediante un Supervisor[4].

---

[3] In una accezione più generale, senza riferimento a precedenti citazioni, da intendere end-to-end come comunicazione tra un host-destinazione ed un host-ricezione.
[4] In tal caso occorrerebbe parlare più propriamente delle architetture A e B, dal momento che nella architettura C le informazioni del supervisor dovrebbero viaggiare sulla stessa *Global Network*.

Nel prossimo paragrafo verrà brevemente descritta l'architettura TCP/IP, o *Internet Protocol Suite*, nonché il protocollo UDP stesso.

5.5.1 Internet Protocol Suite

Il progetto di ricerca che diede origine all'attuale architettura fu Arpanet [12], e fu finanziato dal Ministero della Difesa americano per creare una rete di comunicazione tra computer estremamente affidabile e costantemente attiva. L'architettura TCP/IP fu introdotta per integrare i diversi tipi di rete che venivano via via sviluppati, e permetterne quindi la completa interconnessione.

TCP/IP presenta dei livelli [40] [36] [12], o strati logici, che vengono riportati nella figura 5.8, ove il più alto è il livello Applicazione.

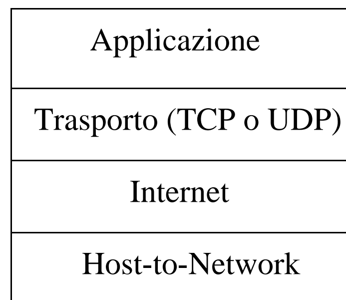


Figura 5.8: Livelli dello Internet Protocol Suite.

Livello Applicazione

In esso sono contemplati tutti i protocolli di alto livello per le applicazioni reali, quali **Telnet**, **FTP** (*File Transfer Protocol*) etc.

Livello Trasporto

Progettato per garantire passaggio di informazioni (pacchetti) end-to-end in cui sono definiti due protocolli: **TCP** (*Transmission Control Protocol*) e **UDP** di cui si è già accennato.

Livello Internet

Il livello che tiene assieme l'intera architettura permettendo il trasferimento di pacchetti tra host ubicati su stessa o diversa rete, garantendo nel contempo il corretto instradamento dell'informazione; il protocollo ufficiale per i pacchetti è **IP** (*Internet Protocol*). Su questo livello gravano incombenze di routing e di controllo della congestione.

Livello Host-to-network

Questo livello, il più basso, è relegato allo hardware: l'architettura prevede solamente che lo host abbia capacità di inviare pacchetti **IP** sulla rete.

Dopo questa premessa su TCP/IP analizziamo meglio le caratteristiche del protocollo UDP.

Esso è strutturato, al contrario del protocollo TCP, per fornire dei servizi non orientati alla connessione: ogni messaggio fornito dell'indirizzo completo della destinazione, è instradato nella rete in modo indipendente da altri messaggi. Poiché tra host-destinazione ed host-trasmittente non c'è connessione (come avviene nel sistema telefonico), può accadere che i messaggi spediti ad una destinazione arrivino in un ordine diverso da quello originario [28].

Però, rispetto a TCP, proprio per il motivo che UDP non è orientato alla connessione, il protocollo UDP ha il vantaggio di fornire una maggiore velocità di trasferimento dei dati [36]. Inoltre sviluppare del software con UDP piuttosto che con TCP comporta una sensibile riduzione di lavoro.

5.5.2 Struttura dell' interconnessione con Ethernet

Come organizzazione del lavoro, con uno sguardo rivolto al Supervisor del 2° livello, si è pensato di introdurre una configurazione di tipo Master-Slave, con un minimo di protocollo suppletivo, per lo scambio di informazioni: queste ultime sono ovviamente lontanissime dal simulare quelle probabilmente presenti in una futura architettura di 2° livello.

La configurazione completa del sistema, visibile in figura 5.9, è stata così strutturata:

- Due schede PCI-DSLink con funzione di destinazione e sorgente dei dati ad alta velocità (pensate ad esempio come interfacce per i *Data Driven* ed i *Global Processor* in un'architettura tipo A di LVL2, o *Local Processor* e *Global Processor* per l'architettura di tipo B), e funzione Slave nell'ambito della interconnessione Ethernet.

- Una scheda PCI-DSLink con duplice funzione: *Controlling process* per il controllo e configurazione di uno switch STC 104, e Master per la gestione e scambio di informazioni da remoto con le schede Slave (funzioni queste, tipiche del supervisor).

- Uno switch STC 104.

- Tre schede Ethernet alloggiate nei tre diversi host in cui trovano posto le interfacce PCI-DSLink.

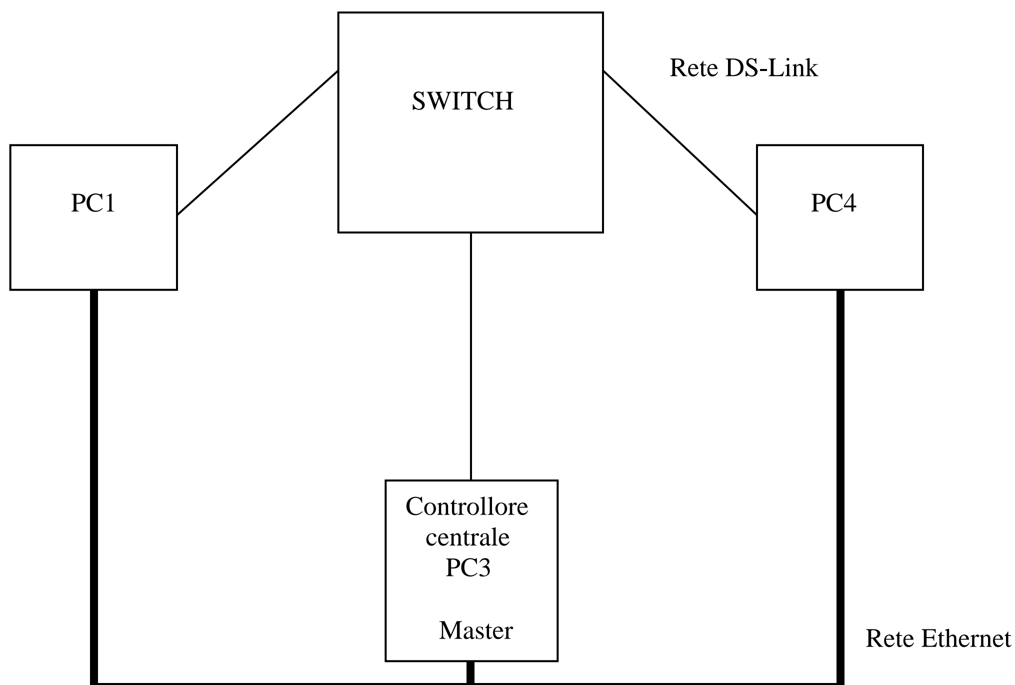


Figura 5.9: Configurazione del sistema.

Per la realizzazione dell'applicazione che permette la trasmissione e ricezione delle informazioni, sono state utilizzate le funzioni della *Berkeley Software Distribution* (BSD), la quale fornisce una tra le interfacce più comunemente usate per la programmazione di applicazioni (API): tali funzioni sono incorporate nel compilatore GCC. Il lavoro è stato condotto sfruttando i *socket*[5], secondo la sequenza di figura 5.10 che è poi la tipica procedura adottata utilizzando protocolli non orientati alla connessione [36].

Nella prima fase il Master attiva la chiamata **socket()** con cui viene creata una struttura dati all'interno del sistema operativo in uno spazio chiamato (*address family*) e con cui si avvia l'utilizzo del particolare protocollo (TCP opp. UDP); mediante la **bind()** viene assegnato un nome alla struttura precedentemente creata, mentre con **recvfrom()** e **sendto()** vengono rispettivamente ricevuti ed inoltrati pacchetti. Il meccanismo è analogo per lo Slave.

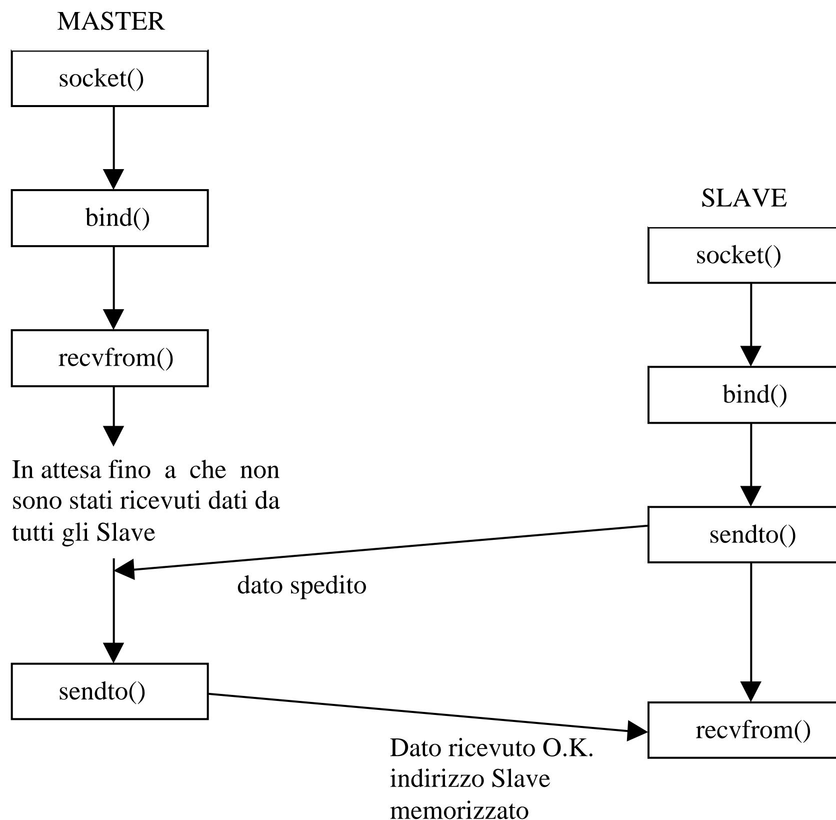


Figura 5.10: Sequenza delle chiamate per la connessione tra Master e Slave.

[5] Canali di carattere generale per comunicazioni tra processi locali o remoti.

La fase iniziale di ricezione del Master è stata strutturata per permettere a più di uno Slave di connettersi ad esso. Grazie all' uso di altre funzioni per i socket presenti nel GCC, quali ad esempio **gethostbyaddr**(), il Master in questa fase riconosce qual'è lo identificativo IP (unico) di ogni Slave, che viene automaticamente memorizzato in un array di puntatori. In tal modo il Master può colloquiare in ogni momento con un qualsiasi Slave: questo rientra tra l' altro nella politica di azione del supervisor, secondo cui esso può mandare informazioni o comandi ai vari processori senza che questi ne abbiano fatto richiesta.

Nelle fasi successive Il Master comunica allo Slave se dovrà lavorare in modalità destinazione o sorgente DS-Link: in quest'ultimo caso il Master inoltrerà allo Slave il volume del traffico da trasferire, le dimensioni dei pacchetti nonché la testa necessaria all' instradamento attraverso lo switch.

Quando sono terminate le procedure assegnate agli Slave, questi spediscono al Master la loro abilitazione ad una successiva operazione di flusso dei dati: in questo caso il Master deciderà se continuare o interrompere l' intero processo e quindi anche la interconnessione con la rete Ethernet, inviando un opportuno comando agli Slave.

Per distinguere i vari comandi che il Master deve inviare agli Slave è stato introdotto un semplice protocollo come illustrato in figura 5.11. Il primo byte identifica il comando mentre i successivi contengono l' informazione; nel caso in cui comando sia la interruzione della connessione Ethernet, questi ultimi byte sono vuoti.

Comando	N° Link	N° Pacchetti	Dim. Pacchetti

Figura 5.11: Struttura del pacchetto spedito dal Master.

Sono stati infine effettuati dei test di velocità sulle schede PCI-DSLink per appurare se l'introduzione di un'interconnessione Ethernet procurava qualche degradamento nelle prestazioni. Le misure sono stato prese tra PC1 e PC4 (dotati di schede di rete veloci per bus PCI) con la tecnica già vista del *round-trip-time*, ponendo PC4 come host per la ritrasmissione dei pacchetti. I dati sono stati fatti fluire attraverso un switch STC 104 configurato dallo host PC3, per un volume di traffico di 15000

pacchetti da 2048 byte. Si riportano di seguito i valori misurati nel caso di controllo da remoto o locale.

La velocità, con un controllo locale delle interfacce ed indicate con V_L, è la seguente:

$$V_L = 9.4 \pm 0.1 \quad \text{Mbyte/s}$$

Il valore della velocità V_R in presenza dell' interconnessione Ethernet è invece:

$$V_R = 9.43 \pm 0.07 \text{ Mbyte/s}$$

Le due velocità sono senza dubbio identiche e quindi il controllo tramite i *socket* non perturba minimamente la velocità di lavoro delle interfacce PCI-DSLink. Questo conferma tra l'altro le scelte, in fase di sviluppo del software, di tenere indipendenti i due meccanismi: flusso dei dati e controllo da remoto.

Ci sarebbero da fare considerazioni circa la scelta del protocollo; con UDP non si ha la completa sicurezza che il messaggio una volta spedito giunga al destinatario (cosa che non avviene per TCP). Per ovviare a ciò si può realizzare un minimo protocollo con lo scambio di messaggi tra UDP-destinazione ed UDP-sorgente, aumentando però il traffico su Ethernet.

Tale aumento è in misura sicuramente inferiore [36] [40] rispetto a quello che si presenta con il protocollo TCP, il quale garantisce l'inoltro delle informazioni al destinatario, nonché l'ordine con il quale sono state spedite, grazie alla presenza dei riscontri (*ack*) provenienti dalla destinazione stessa; il surplus di informazione che viene introdotta appesantisce così il traffico in entrambe le direzioni di comunicazione.

Come ultimo "esperimento" sull'uso dei socket, è stata realizzata una versione del programma, **config_tcp.c**, per la configurazione e controllo da remoto dello switch in ambiente Linux. Con il software realizzato si è in grado, sfruttando in questo caso il protocollo TCP, di configurare ed individuare il numero degli switch che compongono la rete, su un host remoto. Il programma è strutturato nella modalità Client-Server, ove il Client risiede su una macchina remota ed il Server sulla macchina provvista dell'interfaccia PCI-DSLink necessaria alla configurazione degli switch.

CONCLUSIONI

Durante lo sviluppo di questo lavoro di tesi sono stati affrontati diversi aspetti correlati tra loro, che hanno permesso di acquisire teoricamente e poi con esperienza diretta numerose informazioni.

Come primo punto è stato affrontato e risolto il problema della configurazione dello switch, attraverso l'uso di un'interfaccia dalle elevate prestazioni che, in quanto realizzata per bus locale PCI, ha fornito un ottimo mezzo per apprendere e maturare esperienze su questo bus di moderna concezione. Inoltre il software realizzato per la configurazione è stato testato al CERN, sia nella versione con il controllo in locale che in quella Client-Server realizzata mediante l'uso del protocollo TCP; i risultati sono stati positivi e non sono stati riscontrati errori.

Il secondo aspetto è stato l'acquisizione e successiva verifica, di concetti importanti nella realizzazione di *device driver* in un sistema operativo UNIX compatibile (Linux), con cui è stato possibile sviluppare delle istruzioni che interagendo direttamente con il kernel, particolarmente nelle fasi di comunicazione, hanno permesso di ottenere prestazioni elevate.

Attraverso lo studio dello switch, visto come elemento di una rete d'interconnessione complessa come il trigger di 2° livello di ATLAS, si è arrivati a trattare prima teoricamente e poi praticamente una serie di modelli per reti di switch, probabili candidati alle architetture stesse dell'esperimento ATLAS. Realizzando le topologie *banyan* e *Clos*, e con una simulazione di traffico di tipo random, si è cercato di valutare quell'architettura che garantiva prestazioni migliori: in questo caso la ridotta disponibilità delle risorse a disposizione ha limitato l'indagine, ma risultati interessanti sono stati comunque rilevati.

Infine si è cercato di introdurre gli aspetti di comunicazione che dovranno legare il Supervisor ai processori locali e globali, attraverso la realizzazione di un sistema di tipo Master-Slave, sfruttando i *socket* ed il protocollo UDP per lo scambio di informazioni e la rete di STC104 per la trasmissione dati ad alta velocità.

Durante i diversi test di misura si è potuto constatare realmente l'alta configurabilità offerta dal packet switch STC104; tale prerogativa ha infatti permesso di realizzare delle topologie diverse tra loro. Ognuna di esse ha richiesto solamente l'adozione di un appropriato file di configurazione. È forse proprio nella generazione dei file di configurazione che si sono riscontrati degli aspetti negativi, perché nel caso di topologie di una certa complessità la costruzione di questi file (tramite il compilatore NDL) è risultata molto laboriosa e pesante.

In questo caso, come tra l'altro è stato già detto, sarebbe molto più comodo poter disporre di strumenti a più alto livello per la configurazione dello switch, fornendo al tempo stesso maggiore libertà all'utilizzatore.

L'impiego del linguaggio C per la programmazione si è rivelato molto soddisfacente, giacché l'insieme delle librerie necessarie per la realizzazione del *device driver* è scritto per il linguaggio C, così come in C sono scritti i codici sorgente del kernel stesso di Linux.

Inoltre l'aver adottato un sistema UNIX compatibile come Linux si è dimostrato essere una scelta molto valida, in quanto per l'utilizzo dei *socket* è disponibile una vasta e copiosa letteratura nonché una notevole quantità di librerie.

APPENDICE A

Esempi del software realizzato

Si riportano di seguito alcune parti, in codice sorgente, dei principali programmi realizzati.

Questo primo programma chiamato **PCIdsl.c,** si riferisce alla parte principale del *character device driver*.

```
#ifndef __KERNEL__
# define __KERNEL__
#endif

#include <linux/config.h>     /* per CONFIG_PCI */

#ifdef MODULE
# include <linux/module.h>
# include <linux/version.h>
#else
# define MOD_INC_USE_COUNT
# define MOD_DEC_USE_COUNT
#endif

#if (LINUX_VERSION_CODE < 0x020100)
# include <linux/mm.h>
# define copy_from_user memcpy_fromfs
# define copy_to_user   memcpy_tofs
# define ioremap  vremap
# define iounmap  vfree
#else
# include <asm/uaccess.h>
#endif

#include <linux/fs.h>
#include <linux/bios32.h>
#include <asm/io.h>
#include <linux/malloc.h>
#include <asm/pgtable.h>

#include "PCIdsl_int.h"
#include "amcc.h"
#include "dslink.h"

#define MAX_NODES 1
#define WAIT_FOR  0.05  /* 0.05 tempo di attesa */

#define rdtsc(time) ( {__asm  (".byte 0x0f; .byte 0x31" \
        : "=eax" (time)  ) ; } )
```

connessione a *daisy-chaining* tra più STC104 comandati da un *Controlling process*[1] (in taluni casi per brevità verrà usato semplicemente *Controllore*). Questo tipo di connessione è utilizzabile qualora il numero di dispositivi non sia molto elevato (piccole switching network); per reti più complesse è da preferire una connessione di tipo ad albero attraverso la quale si riducono notevolmente le velocità di propagazione dei segnali.

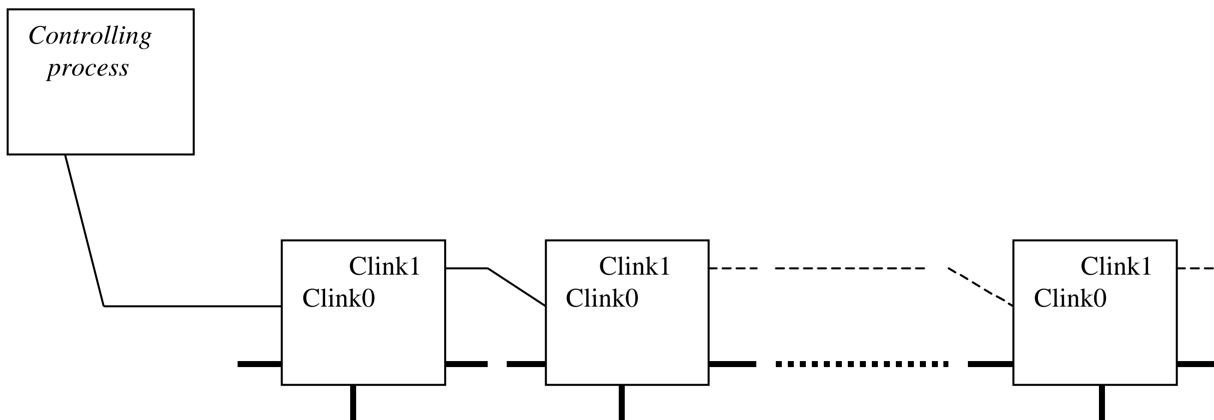


Figura 4.1 : Connessione a daisy-chaining per i Clink0 e Clink1 di più STC 104. In grassetto sono indicati i data-link, il cui numero è puramente indicativo.

Ad ogni pacchetto di comando che il *Controlling process* spedisce allo switch, questi risponde inviando dei messaggi contenenti delle informazioni. Si instaura così tra switch e *Controllore* un processo di comunicazione che è basato su di un protocollo a due livelli [35]: esso permette di evitare possibili problemi di riempimento delle FIFO (vedi Input Link Slice FIFO paragrafo 2.3.1) nonché la presenza di deadlock[2]. Il primo livello del protocollo di comunicazione è caratterizzato dall'uso di *acknowledge packet* (ricordiamo che sono così denominati i pacchetti formati dalla sola testa ed un EOP, senza quindi nessun dato trasportato), che debbono essere scambiati prima della trasmissione di *data packet* (pacchetti contenenti il dato). Se lo Host deve inviare dei

[1] Ossia un processo con funzione di controllo, residente su un host (PC, Workstation od altro), che può comunicare con l'esterno attraverso un opportuna interfaccia DS-Link.
[2] Una condizione di stallo nella quale due nodi si scambiano perennemente lo stesso pacchetto. Il deadlock è una proprietà negativa che discende dalla topologia delle reti e dall'algoritmo di routing utilizzato.

comandi allo STC104, o viceversa, prima dovrà essere spedito un *acknowledge packet* e poi il corpus del comando, ovvero un *data packet*.

Occorre precisare che gli *acknowledge packet* spediti dal *Controllore* presentano due byte di testa, al contrario di quelli spediti dallo switch che ne presentano invece uno solo.

Per il secondo livello sono presenti due classi di *data packet* chiamati *command packet* ed *handshake packet*. Lo scambio dei messaggi prevede che ad ogni *command packet*, spedito in una direzione dal *Controllore*, lo switch risponda con un *handshake packet* nella direzione opposta. Il meccanismo deve verificarsi anche nella direzione contraria di comunicazione, ovvero il dispositivo invia un *command packet* e ad esso lo Host deve rispondere con un *handshake packet*. Questo scambio simmetrico di un *handshake* per ogni *command packet*, evita la presenza di deadlock qualora sia lo switch che il *Controllore*, spediscano un *command packet* nello stesso istante.

Nella figura 4.2 è illustrato il meccanismo completo del protocollo a due livelli per il caso di un singolo STC104.

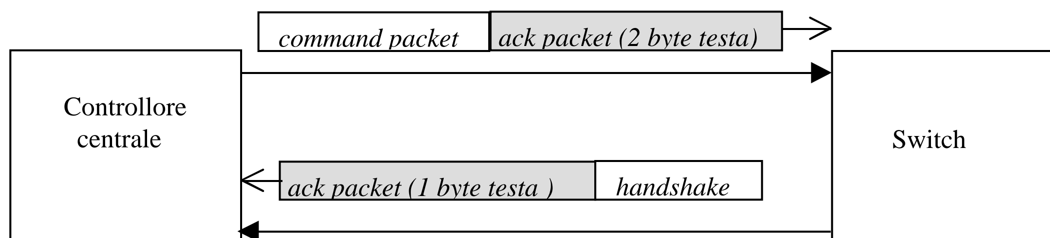


Figura 4.2 : Scambio dei messaggi tra Controlling process e switch, secondo il protocollo a due livelli. Si noti come ogni command, o handshake packet, sia preceduto da un ack packet (primo livello) ed inoltre come le due informazioni possano viaggiare su vie distinte, grazie alla bidirezionalità del DS-Link.

La struttura dei *command packet* è basata su:

- due byte di testa
- un terminatore di tipo EOM
- un byte (*command code*) identificativo del particolare comando

Nella figura 4.3 sono riportati i 6 *command packet* utilizzabili dal *Controllore* verso lo switch, più uno singolo che invece segue il percorso contrario.

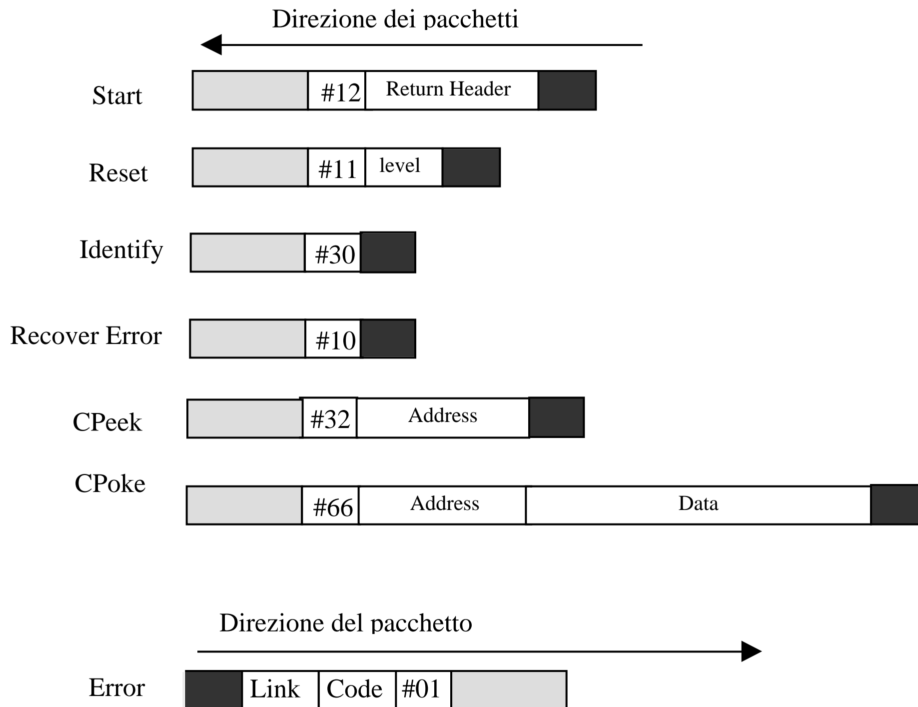


Figura 4.3 : Struttura dei 6 command packet diretti dal Controlling process allo switch. Con il colore chiaro sono indicati i byte di testa, mentre in nero sono rappresentati gli EOM. In basso è riportato l' unico command packet diretto dallo switch al Controllore.

Start

Questo è il primo comando che deve essere inviato allo switch ed è composto di 5 byte. Grazie ai primi due byte (la testa del pacchetto) si etichetta lo switch in modo definitivo cosicché i successivi *command packet*, aventi medesima testa, saranno indirizzati tutti allo stesso dispositivo. Il byte presente nel pacchetto, successivo alla testa, è il codice identificativo del comando stesso (che nel caso specifico è uguale a #12)[3] ed

[3] L' epressione # indica una rapresentazione esadecimale del numero.

infine gli ultimi 2 byte permettono di impostare il *Return header*, che saranno poi i due byte di testa di ogni *handshake packet,* che lo switch ritorna al *Controllore*.

Reset

Il comando è formato da 4 byte e garantisce un reset a 2 diversi livelli che differiscono lievemente tra loro.

Con esso è possibile annullare tutti i valori dei *configuration register* e cancellare eventuali errori presenti. Utilizzando questo comando si ha però la perdita completa dei dati in transito sui *data-link*, proprio perché lo STC104 non presenta più nessun tipo di "configurazione". Il *ResetHandshake* contiene un byte, Status, il cui valore indica se l'operazione di Reset è avvenuta correttamente oppure no.

Identify

L'invio di questo comando determina la risposta da parte dello switch con un numero, 384, che identifica in modo univoco tutti i dispositivi STC 104. Infatti rammentiamo che ci sono altri dispositivi DS-Link, quali lo STC 101 ed il transputer T9000, che posseggono ognuno un proprio numero identificativo. Lo Identify può essere utilmente impiegato per procedure di test, durante le comunicazioni con lo Host.

Recover Error

Permette di ripristinare possibili errori presenti sui *control links*.

CPeek

Questo *command packet*, formato da 5 byte, permette di leggere il contenuto di qualsiasi registro dello switch. Presenta due byte, *Address*, in cui deve essere immesso lo indirizzo del particolare registro che si vuole visualizzare. Nello *CPeekHandshake* relativo sarà contenuto il valore del particolare registro "sondato", inoltre nel byte Status verrà indicata la corretta o meno operazione di lettura .

CPoke

È il comando con cui è possibile scrivere sui bit dei *configuration register*. In tutto presenta 9 byte, di cui 2 sono da per specificare l' indirizzo del particolare registro e

4 che contengono il valore secondo cui i singoli bit del registro debbono essere impostati; per il byte Status dello *CPokehandshake* valgono le stesse considerazioni precedenti.

Error

Tale *command packet* è spedito dallo switch al *Controllore* qualora vi sia un errore sullo STC104. Nel byte indicato come *Code* è riportato il tipo di errore avvenuto, mentre sul byte *Link* si ha un' indicazione del *data link* su cui tale errore è presente.

L' insieme degli *handshake packet* è riportato nella figura 4.4, completi della direzione di spostamento. Anch'essi presentano un byte, *command code*, che è in corrispondenza a quello impiegato nel corrispondente *command packet*; in effetti il *code* dello *handshake packet* ha, rispetto al *command packet*, il primo bit invertito. Ad esempio il *command code* dello Start è pari a #12 ovvero 00010010 in rappresentazione binaria, invertendo il bit più significativo si ha 10010010 pari proprio a #92.

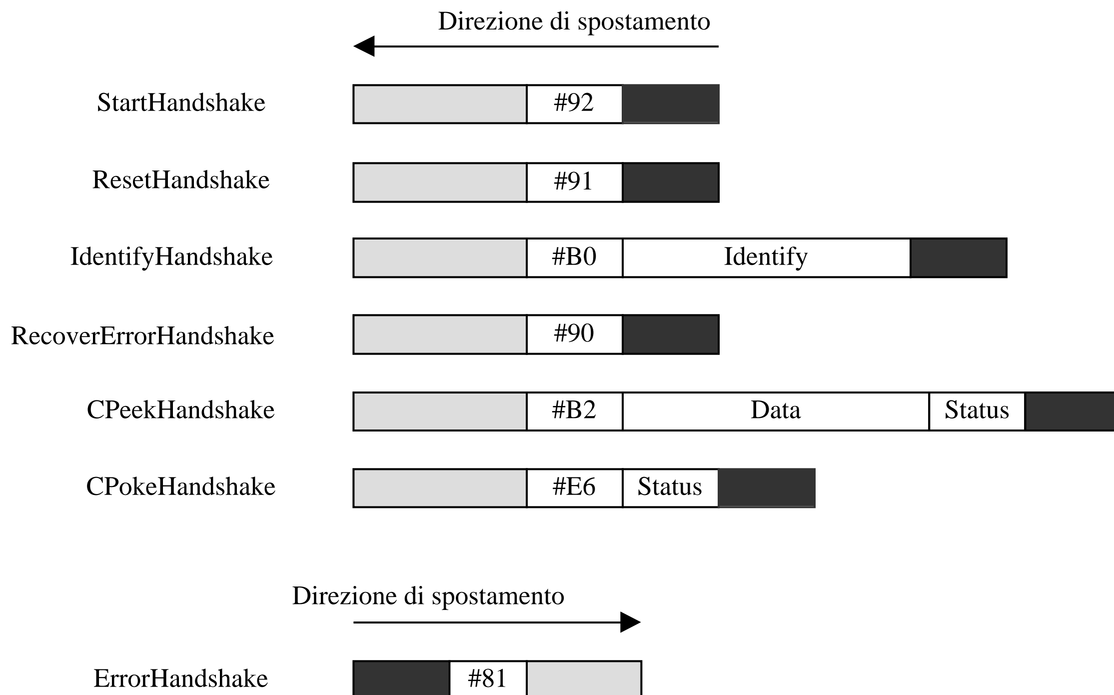


Figura 4.4: Handshake packet. I primi 6 sono spediti dallo switch, attraverso il Clink0, al Controllore, mentre l' ultimo handshake packet è l' unico spedito in senso inverso.

Dopo questa iniziale premessa, sul modo con cui si è in grado di controllare uno switch STC104, proseguiamo ora nella descrizione di come sia possibile implementare in modo software le funzionalità di un *Controlling process*.

4.2 Il Controllore per le switching network

Nei successivi paragrafi si darà spazio alla descrizione del software progettato per la realizzazione del *Controlling process*; si offrirà infine un'analisi delle funzioni create per ottenere degli strumenti di diagnosi e controllo, di una rete qualsiasi di packet switch STC104.

4.2.1 L'interfaccia di controllo

Per ottenere il controllo ed il comando di dispositivi STC104 è stata utilizzata una interfaccia PCI-DSLink [37] realizzata presso i laboratori della Università di Zeuthen, che presenta un singolo adattatore parallelo-seriale STC101 e che garantisce un'alta velocità di trasferimento dei dati grazie alle elevate prestazioni offerte dal bus locale PCI. La interconnessione tra lo STC101 ed il bus è garantita da un controllore prodotto dalla AMCC: lo S5933 [1].

Descrivendo lo STC101 nel precedente paragrafo 2.4, si è illustrata la capacità di funzionamento in quattro modalità differenti: *16 bit processor interface*, *32 bit processor interface*, *16 bit processor interface + token, 16 bit processor interface + multiplexed token*. L'interfaccia PCI-DSLink permette, per la natura del suo progetto, lo utilizzo dell'adattatore STC101 solo nella prima e terza modalità ovvero *16 bit processor interface* e *16 bit processor interface + token*, che per brevità chiameremo rispettivamente *Register-mode* e *Token-mode*.

Utilizzando la scheda in *Register-mode* si ha la possibilità di poter accedere allo insieme dei registri di configurazione dello STC101, tranne però che ai registri Tx/Rx Data24 e Tx/Rx Data32, i quali permettono di scrivere/leggere parole di 24 o 32 bit nelle FIFO del dispositivo stesso. Quindi per la trasmissione/ricezione di pacchetti DS-Link, in *Register-mode*, è possibile utilizzare solamente i registri Tx/Rx Data8 e Tx/Rx Data16.

Nella modalità *Token-mode* (ove ricordiamo che per la trasmissione/ricezione ci sono 2 porte dedicate, le *token port* appunto) sono programmabili ugualmente tutti i registri dello STC 101 mentre per la trasmissione/ricezione è possibile usare solamente parole a 32 bit.

È possibile impostare le diverse modalità delle schede PCI-DSLink attraverso comandi softawre, cosiccome tutte le altre funzioni realizzabili dallo STC101. In aggiunta alle memorie FIFO dello STC 101 il controllore AMCC S5933 contiene altri 32 byte di memoria FIFO in trasmissione ed in ricezione, ed in più sono presenti *on-board* ulteriori 2x16 kbyte di memoria FIFO. La presenza di una NVRAM (Non Volatile RAM) e di una PLD (Programmable Logic Device) AMD MACH445-12 [37] opportunamente programmate dal costruttore, assicurano l'identificazione di tutti i registri dello STC101 [34] e di numerosi altri registri di controllo e di stato presenti sulle schede; questi ultimi possono essere suddivisi in 4 categorie:

- **PCI Configuration Registers** (r/w): sono dei registri di configurazione comuni a tutte le schede elettroniche connesse sul bus PCI. La loro principale funzione è quella di memorizzare gli indirizzi base dei vari dispositivi presenti sulla interfaccia PCI-DSLink.
- **PCI Bus Operation Registers** (r/w): a questi registri è demandata la funzione di controllo dall'interfaccia PCI.
- **C101 Registers** (r/w): sono i registri dello STC101 presente sulla scheda.
- **Interface Control/Status Registers** (r/w): sono tre registri molto importanti ai fini della gestione delle comunicazioni, in quanto permettono di effettuare controlli sul flusso dei dati.

Poiché come già visto i *command packet* sono di diversa lunghezza con numeri pari o dispari di byte, per la realizzazione del *Controlling process* è stata scelta la modalità *Register-mode* che con i suoi 8 e 16 bit in uscita permette la costruzione di pacchetti di qualsiasi misura (ovviamente espressa in byte); è implicito sottolineare che la modalità "pacchettizzazione" risulta attiva.

4.2.2 La strategia di lavoro

Il punto di partenza nello sviluppo di questa parte del lavoro è stato l'utilizzo di un particolare applicativo **NDL** (*Network Descritpion Language*) facente parte di un pacchetto software denominato D7394a Toolset [38], messo a punto dalla SGS-THOMSON e realizzato per lo ambiente MS DOS[4]. Lo **NDL** offre la possibilità di impostare ad alto livello i valori dei *configuration register* il cui spazio di configurazione è pari a circa 28 kbits. È evidente che in presenza di un elevato numero di STC104 lo spazio di tutti i *configuration register* assume dimensioni rilevanti, e poiché ogni variazione di un singolo bit può determinare un diverso comportamento del dispositivo, si capisce che la loro impostazione debba essere svolta con estrema cautela e con procedure automatiche.

Il software NDL ha una propria sintassi che permette di rappresentare ad un sufficiente livello di astrazione, numerose topologie ed architetture di connessione di switching network. Con esso è possibile indicare delle connessioni *data link – data link* sia su un singolo switch che tra più di essi, impostare l'insieme degli *interval* dello switch per garantire l'instradamento di *data packet* su opportuni link d' uscita; inoltre si possono variare le velocità di lavoro sia dei *data* che dei *control link* e così via.

Una volta scritto in formato testuale il codice atto a rappresentare la particolare topologia di switch, si passa ad una fase di compilazione durante la quale si è avvisati di possibili errori nella stesura del codice che possono essere di natura sintattica, di scelta della topologia o di presenza di eventuali deadlock. Il risultato della compilazione dello **NDL** è un file in formato binario, in cui sono riportati tutti i valori dei singoli registri della completa switching network. Per la corretta inizializzazione dei *configuration register* (secondo una precisa topologia), una volta in possesso dell'appropriato file binario, occorre trasferire alla rete tramite lo Host i valori in esso contenuti.

Per la realizzazione del *Controllore* mediante software ad hoc, si è optato per il linguaggio C che è moderno, efficiente e permette tra l'altro di utilizzare chiamate in assembler. Come piattaforma è stato utilizzato un PC di tipo IBM compatibile dotato di slot PCI su cui connettere l'interfaccia PCI-DSLink. Mentre il sistema operativo adottato è stato Linux che in quanto clone di Unix ne riflette tutte le caratteristiche, incluso

[4] Esiste anche una versione per la piattaforma SUN-Solaris4.x che però non è stata utilizzata nel presente lavoro.

multitasking, memoria virtuale, librerie condivise, utilizzo di TCP/IP[5] ed altro. Dal momento che Linux è stato sviluppato essenzialmente da "volontari" programmatori, esso è distribuito gratuitamente attraverso numerosi siti Internet e sono disponibili i codici sorgenti del kernel[6], di librerie, e di molti applicativi. Infine per questo ambiente di sviluppo si è utilizzato il compilatore GCC[7].

Per assegnare i corretti valori ai registri dello STC104, l'informazione contenuta nel file binario deve essere interpretata e successivamente trasferita allo switch. È stato creato un file binario estremamente semplice tramite il compilatore NDL per un singolo switch, in cui sono impostate solo le velocità di funzionamento del *data link* 0 e del *control link*; in seguito ne è stato realizzato un altro che, oltre ai parametri delle velocità (le stesse del precedente), conteneva un'informazione di routing: *interval* del *data link* 0 impostato con il valore 0. A seguire ne sono stati realizzati degli altri aumentando progressivamente il numero dei *data link*, ed impostandone i relativi *interval* in modo lineare[8].

Per mezzo di funzioni di lettura del linguaggio C (**fseek** e **fread**) è stato possibile individuare nei dati dei raggruppamenti di numeri facilmente associabili a gruppi di registri[9]; in particolare una struttura di numeri che rappresentava fedelmente i diversi *interval* precedentemente impostati. Procedendo nella costruzione dei file binari in questo modo, cioè aggiungendo di volta in volta una singola "variabile" (come ad esempio la presenza di *header deletion*, oppure delle serie di link con *header deletion* o magari con *adaptive grouping routing* etc.) si è riusciti a scoprire l'organizzazione completa dell' informazione contenuta nel file stesso; nel caso poi di più switch l' analisi è stata banale, perché nel file binario erano presenti grossi ed evidenti blocchi di numeri, pari proprio al numero di dispositivi impostati nello NDL.

Di seguito sono riportati due file scritti nel linguaggio NDL: il primo in cui nessun *interval* è impostato ed il secondo in cui lo sono tutti.

[5] Il protocollo di comunicazione, largamente diffuso, per le connessioni tra reti eterogenee.
[6] L'insieme delle procedure che implementano i processi elementari, o basici, di un sistema operativo e le comunicazioni tra essi.
[7] Il compilatore fornito dalla GNU project, la fondazione per lo sviluppo e diffusione di software libero.
[8] Link 0 con lo *interval* impostato al valore 0, il link 1 con lo *interval* impostato al valore 1 e così via.
[9] Infatti i *configuration register* sono catalogati in gruppi: **Interval** per i registri relativi agli *interval*, **Random** per i registri relativi all' utilizzo del *two-phase-routing* etc.

```
#INCLUDE "c:\tools\libs\stdndl.inc" --libreria per le funzioni usate


CONTROLPORT Host :
NODE switch :
NETWORK
  DO
    SET DEFAULT(link.speed.multiply  := 20)--impostazione della velocità
    SET DEFAULT(link.speed.divide    := [1])--velocità dei data-link
    SET DEFAULT(control.speed.divide := [2])--velocità dei control-link

    SET switch (type                 := "C104")


 CONNECT Host[control]TO switch[control.up]--Connessione tra Host e Clink0
:
```

```
#INCLUDE "c:\tools\libs\stdndl.inc"


CONTROLPORT host :
NODE switch :
NETWORK
  DO
    SET DEFAULT(link.speed.multiply  := 20)
    SET DEFAULT(link.speed.divide    := [1])
    SET DEFAULT(control.speed.divide := [2])

    SET switch (type                 := "C104")
    SET switch (interval.separator[0]:= 0)--Impostazione per lo interval 0

    DO i = 0 FOR 32 --Impostazione degli altri 32 interval
      DO
        SET switch (interval.separator[i+1]   := i+1)
        SET switch (delete[i]                 := FALSE)--Assenza di
header deletion che invece, con l'opzione TRUE, risulta attiva

    CONNECT host[control]         TO switch[control.up]
:
```

Da notare che in questo linguaggio i commenti sono indicati con il simbolo “**--**“.

In possesso della “chiave” di lettura dei file binari si è passati a costruire in C le funzioni per spedire i *command packet* e gli *acknowledge packet*, strutturando così il Controllore secondo il protocollo a 2 livelli.

4.2.3 La realizzazione software del Controllore

Il sistema operativo Linux rappresenta i dispositivi fisici come file speciali localizzati nel direttorio **/dev** attraverso i quali l'utilizzatore può accedere alla risorsa hardware vera e propria. I dispositivi (e quindi i file speciali ad essi associati) sono suddivisi in due categorie principali[10]: carattere e blocco. Sul primo è possibile scrivere e leggere delle sequenze di caratteri (il caso, ad esempio, di un terminale oppure di una porta parallela), mentre sul secondo è permessa la scrittura e la lettura dei dati con dimensioni di blocchi (è il caso dei dischi fissi, con dimensioni dei blocchi che spesso sono dell' ordine di 512 o 1024 byte); la nostra interfaccia, dovendo leggere dei dati di dimensioni molto disomogenee, può quindi essere facilmente accostata a dispositivi a carattere.

Il problema sarà poi quello di passare dalla risorsa hardware alla sua "rappresentazione" mediante file speciale: questo aspetto lo affronteremo più avanti in uno specifico paragrafo, quando si parlerà della realizzazione di un *character device driver* [31], ossia lo "strato" software che permette l'interazione tra l'utilizzatore ed il kernel, e tra il kernel e la risorsa hardware. Si suppone per il momento che attraverso il *character device driver* l'interfaccia PCI-DSLink sia accessibile da parte dell'utilizzatore ed il bus locale PCI correttamente inizializzato [31] [32].

I diversi *command packet* ed i relativi *handshake packet* sono stati costruiti con delle funzioni apposite. La funzione *cpoke* di seguito riportata, permette di costruire e spedire un pacchetto secondo la specifica del relativo *command packet CPoke*: 9 byte più un terminatore di tipo EOM (vedi figura 4.3). Ricordiamo che tale *command packet* permette di scrivere sui bit dei *configuration register*.

Il codice relativo alla funzione è il seguente:

```
void poke (unsigned short HEADER, const unsigned short address,const
long int Data)
{

        int dim = 10;
        unsigned char Payload[dim];
        int  j;
        int  numbyte = 9; /*Num. di bytes necessari per il CPoke command */
```

[10] Spesso però le schede di rete (Ethernet o altro) vengono considerate come categoria a parte di dispositivi; si potrebbe quindi parlare di un totale di tre tipi diversi di categorie.

```
 /*Abilitazione del registro del C101 TX SEND PACKET, che permette di costruire
pacchetti di 9 byte + un EOM */
          my_outw(numbyte|TERMINATOR_ENABLE|
          EOM_TERMINATOR, c101_regs_base|TX_SEND_PACKET);

/* Assegnazione dei valori opportuni ai bytes del poke command */
          Payload[0] = (HEADER & 0x00ff);
          Payload[1] = (HEADER & 0xff00) >> 8;
          Payload[2] = 0x66; /* codice del CPoke */
          Payload[3] = (address & 0x00ff);
          Payload[4] = (address & 0xff00) >> 8;
          Payload[5] = (Data & 0x000000ff);
          Payload[6] = (Data & 0x0000ff00) >> 8;
          Payload[7] = (Data & 0x00ff0000) >> 16;
          Payload[8] = (Data & 0xff000000) >> 24;

          for (j=0; j<numbyte; j++)
          {
             if((my_inw((c101_regs_base|TX_INTERRUPT_STATUS)&
                 C101_TX_LEVEL)== 0) /*  Controllo necessario affinché la
FIFO in Tx sia libera prima di poter trasmetter dati */

              {
                 my_outb(Payload[j],c101_regs_base|
                 TX_DATA_8_BIT); /* trasmissione su DS-Link tramite il
registro TX Data ad 8 bit */

              }
          } /* Fine del ciclo for */

}/* Fine della funzione poke */
```

Le tre variabili della funzione **poke**, **HEADER**, **address**, **Data**, sono da assegnare in base allo switch ed al suo registro da configurare. Infatti con **HEADER** si indirizza il comando al preciso switch, **address** rappresenta la locazione del particolare registro mentre nella variabile **Data** è contenuto il valore a cui debbono essere impostati i suoi bit.

Nella funzione sono presenti delle istruzioni del linguaggio C, mentre altre sono state realizzate appositamente quali, **my_outb**, **my_inw**; queste ultime permettono di accedere alle periferiche dallo spazio utente[11], per mezzo del *character device driver*, rispettivamente in output ad 8 bit ed in input a 16 bit. All'interno della funzione **poke** le variabili **HEADER**, **address**, e **Data** vengono suddivise in byte e distribuite secondo il formato del *command packet CPoke* (vedi figura 4.3).

[11] Tale accezione deriva dal fatto che si è soliti parlare di *spazio kernel* per indicare lo spazio ove è attivo il *character device driver*, ed invece con *spazio utente* dove sono attive le applicazioni.

Dopo che nella funzione **poke** viene attivato il registro per la trasmissione, TX_SEND_PACKET, tramite l'istruzione

```
my_outw(numbyte|TERMINATOR_ENABLE|EOM_TERMINATOR,
        c101_regs_base|TX_SEND_PACKET);
```

si passa alla trasmissione vera e propria del pacchetto per mezzo del registro TX_DATA_8_BIT. In questa fase è importante che la FIFO in trasmissione sia libera, perché una scrittura sulla FIFO mentre questa non è libera e pronta a trasmettere dati, può provocare lo stallo indefinito del sistema operativo, poiché le funzioni di I/O operano in modalità cosiddetta "a basso livello".

La funzione **poke** come si può notare trasmette solo il *command packet* relativo a *CPoke*; occorrerà quindi un' altra funzione che riceva lo *handshake packet* spedito dallo switch a seguito del comando *CPoke*. Non si riporta l'intera funzione **receive** che realizza tale operazione, ma basterà ricordare che essa è stata strutturata in modo da soddisfare le caratteristiche del protocollo a due livelli: inizialmente essa leggerà quanti pacchetti sono stati ricevuti e la loro lunghezza, poi distinguendo tra *ack packet* ed *handshake packet* spediti dallo switch, rileverà il contenuto dello *handshake packet* trasferendolo in una variabile.

Il prototipo della funzione **receive** si presenta nel modo seguente:

```
void receive ( unsigned char Info[ ]);
```

Nell'array **Info[ ]** è contenuto il valore dello *handshake packet*; realizzando la funzione **receive** si è tenuto conto che dovrà essere in grado di ricevere tutti i tipi di *handshake packet*.

Con le due funzioni **poke** e **receive** si è poi costruito la funzione completa **CPoke,** che effettua anche un test per vedere se lo *handshake packet* è stato ricevuto correttamente o se c'è qualche errore (per mezzo del byte Status che indica la corretta, o meno, operazione di scrittura sul registro).

La funzione completa **CPoke** ha allora la seguente forma:

```
int CPoke(unsigned short HEADER,const unsigned short address,const long
int Data)
{
   unsigned short Packet[2];

   ack_packet(HEADER); /* Questa funzione è lo ack packet mandato dal
Controllore verso lo switch, secondo il primo livello del protocollo */

   poke(HEADER,address,Data);
   receive(Packet);

/* Check per vedere se il valore dello handshake packet contiene sia il giusto
command code(#e6) che il corretto valore di Status, il quale in caso di
operazione riuscita deve essere uguale a zero */

   if ((Packet[0] != 0xe6) || (Packet[1] != 0x00 ))
   {
       printf(" ATTENZIONE ERRORE NELLO HANDSHAKE PACKET :\n");
       printf(" Packet[0] = %x\n",Packet[0]);
       printf(" Packet[1] = %x\n",Packet[1]);

       return 1;
   }
   return 0;

}/* Fine della funzione */
```

Poiché la funzione ritorna un valore intero, questo varrà 0 in caso di operazione **CPoke** riuscita, altrimenti assumerà il valore 1 e verranno visualizzati, dello *handshake packet*, sia il contenuto del *command code* che del byte Status.

Seguendo la struttura del protocollo a 2 livelli, sono state costruite tutte le altre funzioni, ognuna relativa ad un diverso *command* ed *handshake packet*. In possesso di questo insieme di funzioni è stato poi possibile trasferire i dati contenuti nel file prodotto dallo **NDL** ed avviare così la completa procedura di inizializzazione dei *configuration register* dello switch. La lettura del file binario è stata strutturata per gruppi di registri, ed ogni singolo valore di essi, con il relativo indirizzo, è inviato allo switch tramite la funzione **CPoke**.

Date le dimensioni del programma (circa 1600 linee di codice), si è pensato di organizzare il codice in più file, collegati ad unico file header, contenente i prototipi delle funzioni. Con essi è stata creata una libreria, ***libPCIdsl.a***, che viene utilizzata in fase di compilazione nel programma principale (**config.c**). Tale programma utilizza delle funzioni a più alto livello, quale ad esempio la funzione **Configure**, che ingloba sia le

funzioni **CPoke** e **CPeek** che altre, con cui si dà avvio alla completa operazione di configurazione. Il codice relativo alla lettura del file binario è riportato nella APPENDICE A. Si riportano di seguito alcune delle funzioni prototipo contenute nello header file chiamato *mypci.h*.

Quelle che seguono sono alcune delle funzioni derivate dal *character device driver*, utili sia per l'inizializzazione del bus PCI che per l'accesso in modo utente, tramite le funzioni di I/O, alla scheda PCI-DSLink.

```
int my_pcibios_present(unsigned short *bas_adr0, unsigned short  *bas_adr1);
unsigned short my_inw(unsigned short add);
void my_outb(unsigned char data, unsigned short address);
```

Funzioni utili alla configurazione della rete di STC104.

```
void Configure(int *NumberDev,const int head);
int Start( unsigned char Data[]);
int Reset(unsigned short HEADER,const int level);
int Identify(unsigned short HEADER);
int Recover_error( unsigned short HEADER);
int CPeek (unsigned short HEADER);
int CPoke(unsigned short HEADER,const unsigned short address,const long int Data);
int Error_Handshake(unsigned short HEADER);
int ack_packet(unsigned short HEADER);
int receive(unsigned char Info[]);
```

Sono state realizzate due ulteriori funzioni, **Start_network** ed **ISPY**, che sono di diagnosi e check della rete di STC104. Il loro prototipo è il seguente:

```
int  Start_Network(void);
void ISPY(unsigned short Number);
```

Nel prossimo paragrafo verranno descritte queste due ultime funzioni.

4.2.4 Le funzioni di diagnosi per una switching network

La funzione **Start_network**, che nella realizzazione del *Controlling process* deve essere impiegata prima di tutte le altre, “indaga” la rete in una connessione a *daisy-chaining*, individuando il numero di dispositivi in essa presenti.

La **Start_Network** spedisce uno *Start command* (che nella fase iniziale, vedi paragrafo 4.1, deve precedere ogni altro comando) al primo STC104 a cui viene assegnato il valore 0 per il *Return header* ed etichettato con lo stesso valore 0. Dopo questa fase viene spedito un comando di attivazione del **Clink**1 (da notare che all’atto della accensione dello switch il **Clink**0 si porta subito in uno stato attivo, trasmettendo una serie continua di NULL TOKEN): come questi è attivato, allo switch giunge un nuovo comando di **Start** che presenta però un valore di header uguale a 1; lo **Start** *command* viene perciò direttamente posto dal dispositivo al suo **Clink**1, per essere propagato ad un ulteriore secondo switch. Se è presente un secondo, un terzo STC104, e così via, questo meccanismo viene ripetuto fino allo n-esimo.

Poiché al **Clink**1 dell’ultimo switch della catena non ne è connesso nessuno altro, questi non propagherà nessun handshake di ordine n+1: la funzione **Start_network**, in base agli handshake ricevuti, riporterà il numero totale di dispositivi presenti. In questo modo si ha inoltre la catena di STC104 già etichettata in modo lineare e progressivo.

Nella figura 4.3 è riportato un diagramma di flusso che esemplifica il meccanismo di lavoro della funzione **Start_Network**.

Nel caso invece che nessuno dispositivo sia connesso al *Controlling process*, la funzione indicherà un valore nullo. Sono stati poi preparati tre diversi file binari di configurazione (poiché tale è il numero di dispositivi utilizzati) molto semplici, relativi ad uno, due o tre componenti. La funzione **Configure** in base al numero di packet routing individuato dalla **Start_network** provvede alla loro configurazione.

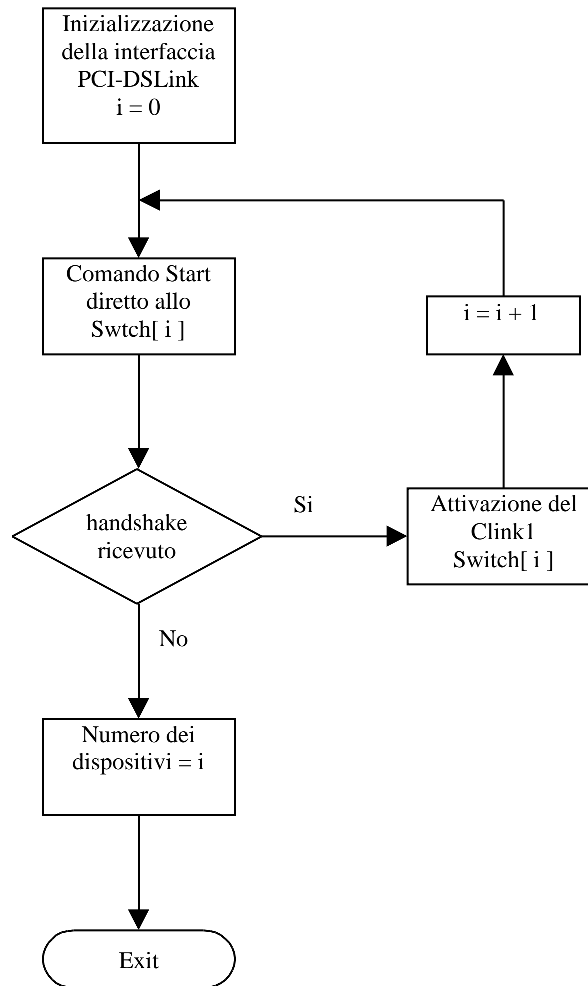


Figura 4.3: Diagramma di flusso della funzione Start_network.

Nel corso del lavoro si è osservato che potrebbe essere interessante sviluppare dei meccanismi di configurazione più “intelligenti” e con un minore intervento da parte dell’utilizzatore: si potrebbero dare ad esempio al Controllore delle indicazioni ancora più ad alto livello, fornendo solo una mappa della topologia da realizzare (magari grazie all’ interazione con dei tool grafici), relegando al Controllore stesso (ovvero al software che lo gestisce) compiti quali: connessioni dei *data-link* sfruttando o il *grouped adaptive routing* oppure lo *universal routing*, con *interval* degli stessi link realizzato, ad esempio, in modo progressivo e senza pericolo di deadlock e così via; è altresì evidente che ciò richiederebbe uno sforzo di progettazione e realizzazione per nulla banale.

La seconda funzione, **ISPY**, realizza un’ ispezione di tutte le connessioni tra *data-link* di uno o più switch, indicando il numero dei link e dei dispositivi interessati. Di

seguito si riporta un esempio della schermata che appare nel caso di due STC104, i quali presentano delle connessioni tra diversi *data-link.*

```
*************************************************
************   SINGLES   CONNECTIONS ************
*************************************************

                  C104 NUMBER 1

    LINK 10        <----------->       LINK 12


*************************************************
************    ALL   CONNECTIONS    ************
*************************************************

  C104 1                                  C104 2

    LINK  8        <----------->       LINK  2
    LINK 20        <----------->       LINK 30
```

L' indicazione è abbastanza esaustiva: il primo componente, quello etichettato con 1, presenta i link 10 e 12 connessi assieme, inoltre i link 8 e 20 del primo sono connessi rispettivamente con i link 2 e 30 del secondo. La funzione **ISPY** è molto utile per individuare tutte le connessioni di una rete comunque complessa, sia nel caso che il loro numero risulti così alto da non essere immediatamente evidente, sia che si voglia costruire un file di configurazione partendo dalle connessioni esistenti. Se il Controllore viene gestito da remoto, ad esempio tramite una connessione Ethernet, oppure ove non è possibile una ispezione fisica delle connessioni, avere a disposizione un programma di collaudo e diagnosi, quale **ISPY**, risulta estremamente utile.

ISPY si basa sul fatto che nel momento in cui un link dello STC104[12] viene attivato, comincia a trasmettere un serie continua di NULL TOKEN (vedi paragrafo 2.2.2) : se a questo link ne è connesso un altro, nell'istante in cui anche quest'ultimo viene attivato si realizza la comunicazione DS-Link secondo il meccanismo di scambio degli FCT.

[12] In generale ciò vale per qualsiasi link di tipo DS.

In tale situazione se si leggono i registri LINK_STATUS, relativi ai due link, si osserva la presenza di un token ricevuto. Se si attiva prima un link, e ponendosi in lettura del relativo LINK_STATUS si attivano in sequenza lineare e numerata gli altri; come viene rivelata la presenza di un token, viene bloccato il processo di attivazione dei link.

È quindi immediato risalire al link incognito in connessione con il primo semplicemente osservando ove si è fermata la sequenza di attivazione.

4.3 Il Character Device Driver

Dal momento che il sistema operativo Linux gestisce i dipositivi fisici tramite file speciali, contenuti nel direttorio **/dev**, cerchiamo adesso di ripercorrere il meccanismo che porta alla loro realizzazione.

Il *character device driver*, o più brevemente **modulo**, è composto sostanzialmente di istruzioni che permettono di interagire con il kernel, e da altre che permettono di utilizzare, secondo il precipuo impiego, il particolare dispositivo. Nella figura 4.4 è illustrato il meccanismo d'interazione delle applicazioni attive nello spazio utente ed i dispositivi fisici.

Questi ultimi vengono sottoposti ad una prima fase di compilazione da cui si ottiene il codice oggetto, per arrivare, attraverso il linker[13] **insmod,** alla fase di "caricamento" nel kernel che durerà per tutto il tempo in cui quest' ultimo è attivo (quindi fino al successivo reboot o ... eventuale crash). È possibile comunque, tramite il comando **rmmod,** disinstallare il **modulo** dal kernel anche se questi risulta ancora attivo.

Nella scrittura di un **modulo** vengono usate delle funzioni che non sono presenti nelle librerie standard del linguaggio C: ad esempio la funzione **printf** che nell' usuale scrittura di programmi permette di visualizzare variabili, o semplici caratteri di testo, nel **modulo** viene invece sostituita da **printk** che, tra l' altro, è molto utile in fase di *debugging* del codice; il **modulo** è a tutti gli effetti una parte del kernel ed occorre prestare molta attenzione alla sua scrittura, ed un *debugger* come il programma **gdb** è molto utile.

[13] Il Linker è un programma che collega assieme il codice oggetto (object modules) e le librerie per formare un singolo e coerente programma. Gli object modules non sono che codice macchina in uscita da un compilatore e contengono il codice macchina eseguibile e dati che assieme vengono "manipolati" dal linker per produrre un programma.

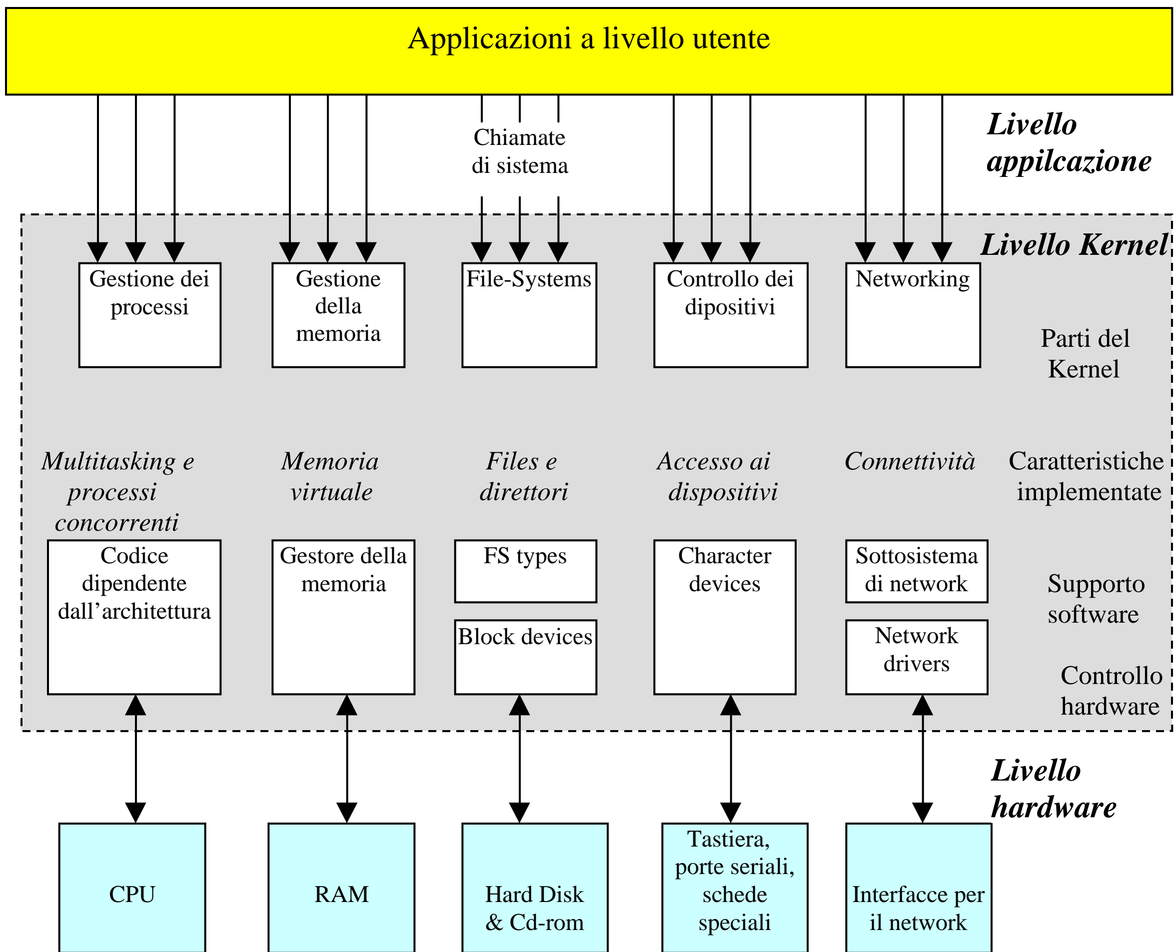


Figura 4.4: Schema esemplificativo dell'interazione tra applicazioni ed hardware attraverso il kernel.

Una volta che il **modulo** è stato inserito nel kernel, con il linker **insmod**, ad esso sarà associato un numero identificativo, *Major Number*, che sarà unico e varrà come riferimento del particolare *character driver*. Questo numero è visibile nel file */proc/devices* in cui sono presenti tutti i moduli al momento inseriti nel kernel, sia di tipo *carattere* che *blocco*.

La funzione **mknod** permette infine di allocare dinamicamente il **modulo** nei file di tipo speciale (quelli nel direttorio /dev per intenderci): questa funzione può essere impiegata, tramite un'opzione, anche per creare file speciali nel caso si utilizzino dispositivi di tipo a *blocco*. Inoltre grazie alla funzione **mknod** possono essere creati più file speciali (partendo da un unico **modulo**), sfruttando il fatto che essi contengono sia un

Major che un *minor number* e quest' ultimo può essere impostato proprio dalla funzione **mknod**. Avere ad esempio dei file speciali con stesso *Major number* ma diverso *minor number*, è utile qualora si abbiano più schede dello stesso tipo da utilizzare.

Vediamo ora come è strutturato basicamente il **modulo** per la scheda PCI-DSLink. Le prime linee di codice includono degli header file che, in questo caso, sono quasi tutti file di sistema. Solitamente le prime righe si presentano nel seguente modo e sono comuni a tutti i **moduli**:

```
#define  _KERNEL_  /*definisce il codice come parte del kernel */
#define   MODULE
#include <linux/module.h> /*libreria per caricare il modulo nel kernel */
#include <linux/version.h> /*contiene la versione del kernel */
```

Seguono poi altre librerie per utilizzare le funzioni di input ed output, oppure per le funzioni di accesso al bus PCI, o ancora per l' accesso alla memoria dello spazio kernel e così via, ed infine delle librerie che contengono i prototipi delle funzioni create ad hoc per la scheda DS-Link, oppure delle dichiarazioni di costanti. Affinché il **modulo** sia effettivamente caricato nel kernel, occorre che esso sia inizializzato (la registrazione nell' appropriata *kernel table*) inserendo nel codice la funzione **init_module()**; si riporta di seguito il codice relativo:

```
int init_module( void)
{
       int stat;
       if(stat=register_chrdev(0,"Nome_modulo",&pcidev_fops))
       {
            printk(" Impossibile ottenere il Major number \n");
            return -1;
       }
       printk(" Major number = %d\n",stat);
   return 0;
}
```

La funzione `register_chrdev` è quella che registra il **modulo** nel kernel. Il primo argomento è il tipo di allocazione richiesto, ed in questo caso con 0 si ha l'assegnazione del *Major number* di tipo dinamico, scelto cioè in base ad altri già presenti. Il secondo argomento è il nome del *character driver* che, per essere in "tema" con l' interfaccia utilizzata, è stato chiamato **PCIdsl**. Il terzo argomento di `register_chrdev` è una

struttura[14] che è utilizzata per chiamare tutte le funzioni costruite e presenti nel **modulo**. Nel caso che `register_chrdev` fallisca nel tentativo di registrare il *character driver* nel kernel, si è avvisati tramite la visualizzazione del relativo messaggio.
Accanto alla funzione `init_module` c'è anche un'altra istruzione che serve a rimuoverlo dalla *kernel table* e liberare quindi risorse; se ne riporta di seguito la forma:

```
void  cleanup_module(void)
{
   unregister_chrdev(Major number, "Nome_modulo");
}
```

La funzione *unregister_chrdev* realizza per l'appunto la rimozione del **modulo** dal kernel: risultano sufficientemente chiari quali sono i parametri utilizzati.

Si è detto precedentemente che la funzione *register_chrdev* presenta un parametro legato ad una struttura: *pcidev_fops*. Essa è di tipo *file_operations* (struttura di puntatori a funzioni) ed è impiegata nel VFS[15] (Virtual Filesystem Switch) per accedere al file di un determinato filesystem [31] [22]. La figura 4.5 schematizza l'attività del VFS.

La struttura *pcidev_fops* assume allora la forma:

```
static struct file_operations pcidsl_fops = {

            NULL,           /* per lseek()   */
            NULL,           /* per read()    */
            NULL,           /* per write()   */
            NULL,           /* per readdir() */
            NULL,           /* per select()  */
            pcidsl_ioctl,   /* per ioctl()   */
            NULL,           /* per mmap()    */
            pcidsl_open,    /* per open()    */
            pcidsl_release, /* per release() */

};
```

[14] Secondo la terminologia del linguaggio C.
[15] Il filesystem è una struttura logica che permette di organizzare un supporto fisico per memorizzare dati in una gerarchia di direttori e file. L' utilizzatore ed i programmi sono così in grado di accedere ai dati non più indirizzando direttamente il supporto fisico, bensì utilizzando il livello più astratto, indipendente dal dispositivo, fornito dal filesystem. Il VFS presente in Linux rappresenta poi un' ulteriore astrazione del concetto di filesystem. Esso fornisce un' interfaccia unificata a tutti i tipi di filesystem che il kernel è in grado di supportare, quali ad esempio: VFAT (Windows 95), ISO-9660 (filesystem per i cd-rom) etc.

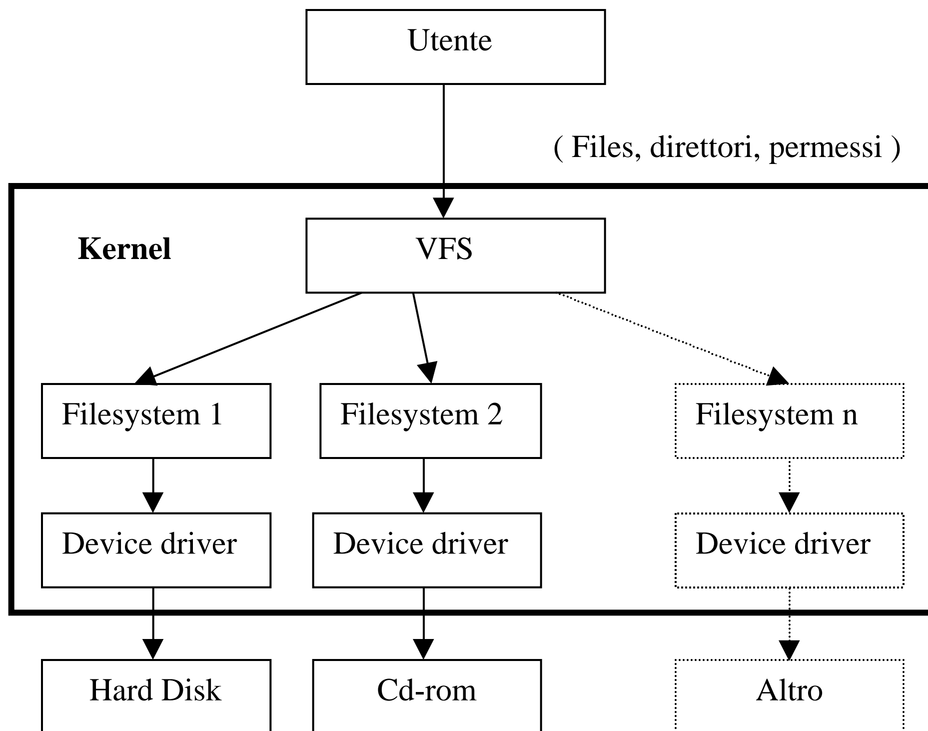


Figura 4.5: Meccanismo d' interazione tra lo spazio utente e lo hardware, mediante il Virtual File System.

L'accesso a queste funzioni da parte dell'utente (propriamente chiamato spazio utente) avviene con le chiamate di sistema[16] **ioctl()**, **open()**, **release()**, contenute nelle librerie standard. Vediamo ora con maggiore dettaglio le tre funzioni create.

La funzione *pcidsl_open* è molto semplice e permette la preparazione del **modulo** per altre operazioni, nonché garantisce che esso non venga rimosso da altre applicazioni quando è in uso. La funzione *pcidsl_release* serve a deallocare la precedente funzione e quindi permettere la rimozione del *character driver* dal kernel. L'ultima funzione, *pcidsl_ioctl*, è invece abbastanza articolata ed è quella che è stata utilizzata per inizializzare lo "spazio di configurazione" del bus PCI, ed inoltre permette di trasferire dati alla scheda attraverso le funzioni già viste di I/O.

[16] Mediamente in un sistema UNIX-like ci sono circa 60 chiamate di sistema.

Vediamone una versione semplificata:

```
static int pcidsl_ioctl(struct inode *inode, struct file *file,
                        unsigned int cmd, unsigned long arg)
{
    int ret;

    switch (cmd) {

    case PCIBIOS_PRESENT:
    {
      .............
      break;
    }

    case DSINW:
    {
      .............
      break;
    }
    case DSOUTB:
    {
      .............
      break;
    }
      default: break;

       return -EINVAL;
    }
   return 0;
}
```

Per brevità sono stati enumerati solo 3 *case* di un totale di 10. Nel primo *case* è riportata la sub-funzione che permette l'inizializzazione del bus PCI e ritorna nello spazio utente gli indirizzi di I/O con cui è possibile accedere alla scheda. Gli altri due *case*, *DSINW* e *DSOUTB*, sono relativi alle operazioni di utilizzo rispettivamente delle istruzioni *inw* e *outb* (scelte puramente a titolo indicativo).

Queste due ultime istruzioni potrebbero essere utilizzate senza necessità di realizzare un **modulo** [31]: ma ciò sarebbe possibile solo con i privilegi di super-user e previo utilizzo di una funzione **iopl**[17]; nel caso poi sia necessario un trasferimento e ricezione di dati ad alta velocità, questo tipo di accesso allo spazio di I/O è comunque da

[17] Esiste anche un' altra funzione *ioperm* che da spazio utente permette di accedere soltanto ad un numero limitato di porte di I/O.

evitare essendo di prestazione inferiore rispetto ad un accesso mediante *character driver*[18]. Si riporta ora, come semplice esempio, la versione completa del *case DSINW*.

```
case DSINW:
      {

        int ret;
        struct inb_ds ioctlinb;

        ret = verify_area (VERIFY_WRITE, (void *) arg,
                           sizeof (struct inb_ds));
        if (ret)
            return ret;

        ret = verify_area (VERIFY_READ, (void *) arg,
                           sizeof (struct inb_ds));
        if (ret)
            return ret;

        copy_from_user (&ioctlinb,(struct inb_ds *) arg,
                      sizeof (struct inb_ds));

        ioctlinb.data = inw(ioctlinb.address);

        copy_to_user ((struct inb_ds *) arg, &ioctlinb,
                      sizeof (struct inb_ds));
        break;
      }
```

In questa sub-funzione l'istruzione *verify_area()*, presente prima della *inw* segnata in grassetto, permette di verificare se ci sia area disponibile di memoria-kernel, per essere poi usata nel trasferire informazione all'utente, o in lettura o in scrittura. Infatti con la istruzione *copy_from_user()* viene trasferita, dallo spazio utente allo spazio kernel, la informazione relativa all'indirizzo di I/O a cui la *inw* deve accedere (ioctlinb.address). Viceversa con l'istruzione *copy_to_user()* è il valore letto da *inw* (ioctlinb.data) che si rende disponibile all'utente.

Questo meccanismo è stato utilizzato per tutte le funzioni realizzate per garantire il passaggio di dati, o informazioni, tra spazio utente e spazio kernel: come si vedrà nel prossimo Capitolo 5, anche le funzioni per le comunicazioni DS-Link sono strutturate in tal senso.

---

[18] In questo caso si realizzerano delle istruzioni *read* e *write* da implementare nella struttura *file_operations*.

L'istruzione **pcidsl_ioctl** è stata decomposta in vari *case* di modo che, quando da utente si richiama la funzione **ioctl(),** sia possibile sfruttare tutte le sotto-funzioni presenti, cambiando semplicemente un parametro nella **ioctl()** stessa**,** e che sarà relativo ai vari *case* della funzione **pcidsl_ioctl**.

Realizzato il *character device driver* è stato sviluppato un programma chiamato **config.c**, che a livello utente utilizza l'insieme di funzioni che offre il *driver* più quelle relative al Controllore della switching network. Di seguito si riporta la pagina di avvio iniziale con il menu disponibile.

```
*************************************************************
*** BASIC CONFIGURATION FOR  NETWORK BASED ON C104 SWITCH ***
*************************************************************
***                                                       ***
***                  Command :                            ***
***                                                       ***
***  CONFIGURATION          (Press   c)                   ***
***                                                       ***
***                                                       ***
***  ISPY  LINK  CONNECT    (Press   i)                   ***
***                                                       ***
***                                                       ***
***  STATUS REGISTER        (Press   s)                   ***
***                                                       ***
***                                                       ***
***  RESET                  (Press   r)                   ***
***                                                       ***
***                                                       ***
***  DATA TRANSMISSION      (Press   d)                   ***
***                                                       ***
***                                                       ***
***  EXIT                   (Press   e)                   ***
***                                                       ***
*************************************************************
```

Premendo il tasto **c** si configura la rete di switch; con il tasto **s** si può visualizzare lo stato di un qualsiasi registro di uno STC104 della rete, e così via. Da notare la presenza dell' opzione DATA TRANSMISSION; vedremo nel successivo Capitolo 5 che all'interno del **modulo** sono state realizzate delle funzioni che permettono di utilizzare la scheda PCI-DSlink per il trasferimento dei dati attraverso il packet routing STC104; tali processi di comunicazione sono gestiti tramite una connessione remota di tipo Ethernet.

Capitolo 5

Prestazioni di switching network

In questo capitolo si darà un'ampia descrizione delle diverse misure effettuate mediante le interfacce PCI-DSLink nonché il packet switch STC104, per valutare il traffico e le velocità di comunicazione in più modalità:

- attraverso una semplice configurazione che prevede l'utilizzo di due sole interfacce.
- Attraverso un singolo switch ma con l'attivazione da remoto, tramite una connessione Ethernet, dei processi di comunicazione DS-Link.
- Mediante interconnessioni più complesse in cui entrano in gioco le reti di switch sia con il modello della *banyan* che della *Clos network*, sfruttando quattro interfacce.

Si forniranno infine dei risultati per convalidare il modello teorico della contesa in uno switch, secondo la trattazione sviluppata nel paragrafo 3.3 .

5.1 Comunicazioni DS-Link su device driver

Per poter utilizzare l'interfaccia PCI-DSLink come supporto con cui effettuare delle trasmissioni di dati ad alta velocità e non solamente nella "veste" di configuratore (come già descritto nel paragrafo 4.2), è stato necessario apportare delle modifiche al *character device driver* il quale, ripetiamo, è lo strato software che permette di far interagire l'utilizzatore, attraverso il kernel, con le interfacce stesse. Sono state introdotte delle funzioni che, invocate dall'applicazione attiva nello spazio utente, permettono la ricezione e trasmissione di pacchetti di qualsiasi dimensione. Per limitare al massimo i ritardi che inevitabilmente introdurrebbe il sistema operativo, si è pensato di strutturare le

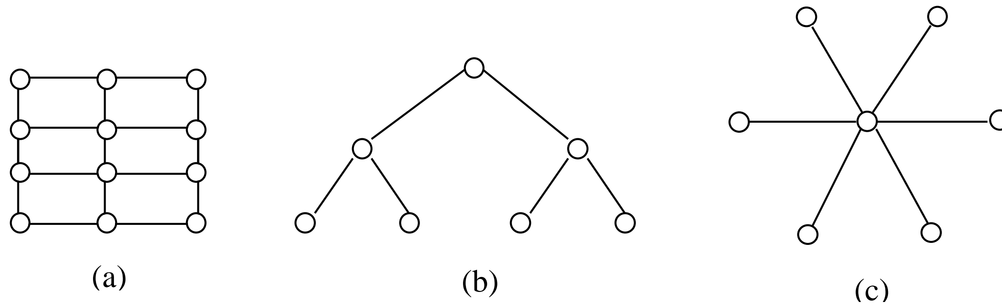


Figura 3.3: a) mesh bidimensionale, c) albero, d) stella.

Le architetture di tipo MIN sono quelle più interessanti per i nostri scopi perché permettono l'interconnessione di un alto numero di processori ed elevate velocità di trasmissione. In generale non hanno un'ottima scalabità, hanno però dei costi di realizzazione contenuti ed una ridotta complessità.

Esempi di MIN sono la *banyan network,* la *Benes*[1] *network* e la *Clos*[2] *network* che verranno descritte nel prossimo paragrafo.

3.2 Le reti di tipo MIN

Vediamo le caratteristiche della topologia *banyan network*. Essa permette di connettere N nodi di calcolo (processori) in ingresso con altrettanti nodi (processori oppure memorie) in uscita.

La sua struttura è rappresentata in figura 3.4 ove per facilità sono stati impiegati switch 2   2: con essi si è costruita una rete 8   8 composta da tre stadi (pensando gli switch come elementi di una colonna) numerati in modo crescente partendo da sinistra. La caratteristica ed anche il <u>limite</u> di questa topologia è che il percorso tra un determinato ingresso ed una uscita è <u>unico</u>.

---

[1] Da V. Benes teorico dei modelli di studio per la teoria dell' interconnessione.
[2] Nome derivato da C. Clos, ricercatore dei Bell labs., che studiò il problema di switching network nello ambito della telefonia.

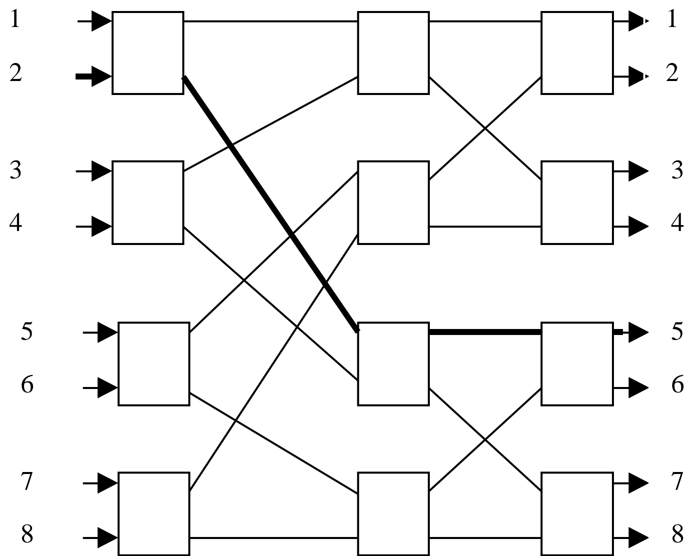


Figura 3.4: Rappresentazione di una banyan network 8   8 formata da switch 2   2. In grassetto è indicato un percorso che connette l' ingresso 2 con l' uscita 5 e come si può notare è un cammino unico, senza possibili alternative.

Purtroppo, dal momento che i percorsi sono unici, c'è la possibilità che negli stadi intermedi avvenga un conflitto tra comunicazioni, detto anche *internal blocking* [11]: ad esempio, riferendoci alla figura 3.4, nell' ipotesi che l'ingresso 1 debba essere connesso all'uscita 6 e l' ingresso 2 con l' uscita 5, sul primo switch in alto a sinistra si creerà inevitabilmente una contesa.

Il totale degli stadi impiegati in una generica *banyan network* N   N è pari a $\log_m N$, dove **m** è il numero di link in ingresso del singolo switch; ogni stadio presenta N/2 switch.

Per ridurre il problema dello *internal blocking* possono essere utilizzate altre architetture, tra cui la *Benes* e la *Clos network*, le quali presentano più di un possibile percorso tra ingressi ed uscite; la *Benes network*, essendo derivata dalla *Clos*, non verrà presa in esame.

La figura 3.5 mostra una *Clos*(N,n,m) *network* a tre stadi, avente **N** ingressi totali ed **N** uscite totali. Questa topologia è caratterizzata dal fatto che lo stadio intermedio fornisce **m** diversi percorsi tra gli ingressi e le uscite, facendo così abbassare la

probabilità di avere contesa negli stadi intermedi: per questo la *Clos* viene anche detta *non-blocking network*.

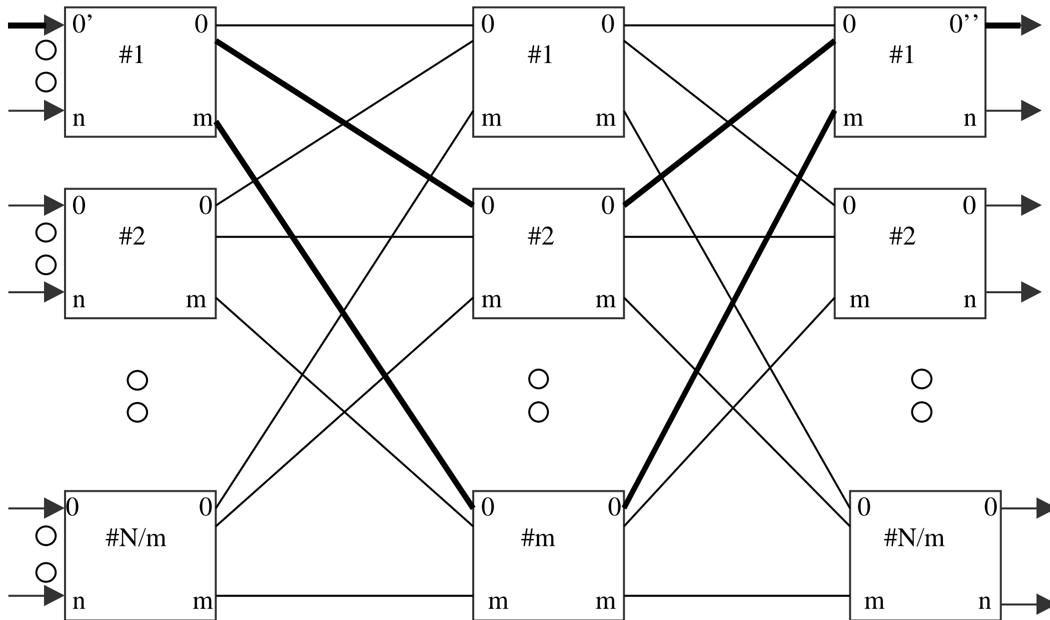


Figura 3.5: Clos(N,n,m) network a 3 stadi. Il primo e terzo stadio hanno degli switch n   m, mentre lo stadio intermedio è realizzato con switch m   m: della rete, sono riportate solo alcune connessioni. In grassetto sono segnati due percorsi alternativi per connettere il nodo 0’ (in alto a sinistra) al nodo 0’’ (in alto a destra).

Le possibili contese tra gli stadi intermedi possono essere ulteriormente ridotte introducendo quelle tecniche già illustrate nel Capitolo2, ovvero il *two phase routing* e lo *adaptive grouping routing*.

Nei prossimi paragrafi verranno analizzate, da un punto di vista teorico, la *banyan* e la *Clos network* [2] .

3.3 Un modello per la contesa in uno switch

Abbiamo osservato nel precedente paragrafo come in taluni casi sia possibile, per due diversi messaggi non diretti necessariamente alla stessa uscita, contendere un link nei rispettivi percorsi. In pratica questo si ripercuote direttamente in una condivisione su uno

switch degli stadi intermedi; questo si troverà così ad effettuare un arbitraggio tra due o più ingressi che richiedono la stessa porta di uscita. Vediamo di analizzare tale aspetto nel caso di switch STC104 e sotto le seguenti condizioni:

- comunicazioni di tipo unidirezionali, da sorgente a destinazione.
- Traffico di tipo casuale cioè con un messaggio che può richiedere qualsiasi link d'uscita.

Indichiamo con **S** il tempo che occupa l' intero pacchetto ad attraversare lo switch dalla testa sino al terminatore: per semplicità consideriamo pacchetti aventi dimensioni fisse di 2048 byte.

Il tempo **S** sarà dato da:

$$\mathbf{S} \quad h + k \quad b$$

dove *h* è il tempo di routing dello switch, k è la dimensione in bit del pacchetto (che deve però includere i 2 bit, parità e tipo token) e *b* è il tempo di transito di un bit ed approssimabile a 10 ns (per velocità del DS-Link di 100 Mbit/s): nel computo di **S**, si è trascurato il tempo di ritardo legato al terminatore, il cui apporto è irrilevante rispetto alla lunghezza del pacchetto.

Un' uscita dello switch risulterà libera se non è utilizzata da nessun ingresso: associamo a questa condizione una probabilità così definita:

$$p(libera) = \frac{N_L}{N_T} \qquad (1)$$

N_L è il numero di casi possibili in cui l' uscita risulta libera, mentre con N_T si indicano tutti i casi. Detto **m** il numero di ingressi ed **n** il numero di uscite, con semplici considerazioni si ricava:

$$N_T = n^m$$

mentre

$$N_L = (n-1)^m$$

quindi la *p(libera)* sarà:

$$p(libera) = (1 - \frac{1}{n})^m . \qquad (2)$$

La probabilità associata al fatto che invece l' uscita sia occupata, sarà evidentemente:

$$p(occupata) = 1 - (1 - \frac{1}{n})^m . \qquad (3)$$

Da queste quantità possiamo determinare un parametro importante: il *throughput* (bit/s) ossia la quantità di dati passanti in un link d' ingresso per unità di tempo.

$$T = S^{-1} \quad p(occupata) \quad r \quad k' \qquad (4)$$

dove **r** $=$ **n**/**m**, mentre **k'** è il numero di bit dati effettivamente trasmessi.

Considerando ad esempio uno switch avente i suoi 32 link distribuiti in 24 ingressi ed 8 uscite ed un tempo di routing h di circa 700ns (2 byte di testa), si ha un *throughput* approssimativamente pari a 3.0 Mbyte/s: un risultato coerente, essendo quasi 1/3 della velocità massima del flusso di dati di un DS-Link, e cioè 9.5 Mbyte/s.

Si sottolinea che in questo semplice modello non si è tenuto conto del meccanismo *flow control token*, dal momento che il decremento che esso apporta alle prestazioni, in termini di *throughput*, è di circa il 5% (nel caso però che siano trasmessi solo *data token*).

La (3) può essere ulteriormente semplificata con una relazione asintotica, infatti sappiamo che:

$$\lim_{i \quad \times} (1 - \frac{x}{i})^i = e^{-x} \qquad (5)$$

ponendo la (2) in altra forma, e nel limite di **m** grande si può scrivere:

$$(1 - \frac{1}{n})^m = (1 - \frac{1}{r \quad m})^m \quad e^{-\frac{1}{r}} \qquad (6)$$

e la nuova espressione per il *throughput* è data da

$$T^{\text{asint}} = S^{-1} \quad (1 - e^{-\frac{1}{r}}) \quad r \quad k' \qquad \text{bit/s}. \qquad (7)$$

La (6) può essere utile per dei calcoli semplici ed approssimati, per studiare il flusso di dati attraverso uno o più switch, aventi un alto numero di link utilizzati.

3.4 Modelli di reti realizzabili

In possesso degli strumenti per modellizzare il meccanismo della contesa, e quindi calcolare il *throughput*, passiamo a costruire dei modelli per topologie reali, considerando le risorse hardware effettivamente disponibili. Con 4 schede PCI-DSLink

fornite ciascuna di un link e con la possibilità di effettuare comunicazioni bidirezionali, si può disporre di un totale di 4 sorgenti e 4 destinazioni: certamente, e già lo abbiamo dimostrato, le velocità di trasferimento dati saranno più basse rispetto al caso in cui si abbiano effettivamente 8 distinti dispositivi con trasmissione unidirezionale.

Con questo numero di sorgenti e destinazioni possiamo costruire solo reti 4   4 impiegando switch 2   2. Dal momento che lo STC104 possiede 32 link e risulta partizionabile, possiamo utilizzarlo come un insieme di 8 switch del tipo 2   2.

Considerando che sono a disposizione 3 STC104 si avrebbe un totale di 24 diversi sub-switch 2   2: occorre però considerare il numero dei cavi fisici utilizzabili, che in realtà limita lo sviluppo di architetture complesse.

Restringendoci al caso di topologie MIN, sarà interessante confrontare le *banyan* e le *Clos network* per vedere se, ad un aumento di complessità di queste ultime, corrisponda un aumento del *throughput*.

Nelle due reti impiegheremo un tipo di traffico *random* (in accordo con quanto premesso per il modello della contesa dello switch) ovvero la destinazione di ogni pacchetto è scelta in modo casuale; questo tra l'altro è un buon criterio per distribuire in modo uniforme il traffico sulla rete.

Per la generazione del traffico ci sarebbero diversi modelli, ma da studi fatti [9] [23] quello *random* risulta molto efficiente, e tra l'altro facilmente realizzabile dal momento che lo switch presenta al suo interno un generatore *random* per le teste dei pacchetti (vedi Capitolo 2).

Impostiamo ora il modello per il calcolo del *throughput*, nel caso delle due reti di figura 3.6 di dimensione 4   4. Nel caso della *banyan*, dal momento che viene generato traffico *random*, c'è la possibilità che nel primo stadio ci sia contesa, e quindi la probabilità che un' uscita dello switch sia occupata è data da:

$$P_1(occupata) = 1 - (1 - \frac{1}{2})^2 . \qquad (8)$$

Ma lo stesso tipo di contesa è presente anche nel secondo stadio e quindi, considerando che le uscite del primo switch sono gli ingressi del secondo, la probabilità associata al dispositivo del secondo stadio sarà

$$P_2(occupata) = 1 - (1 - \frac{1}{2})^{P_1 \;\; 2} . \qquad (9)$$

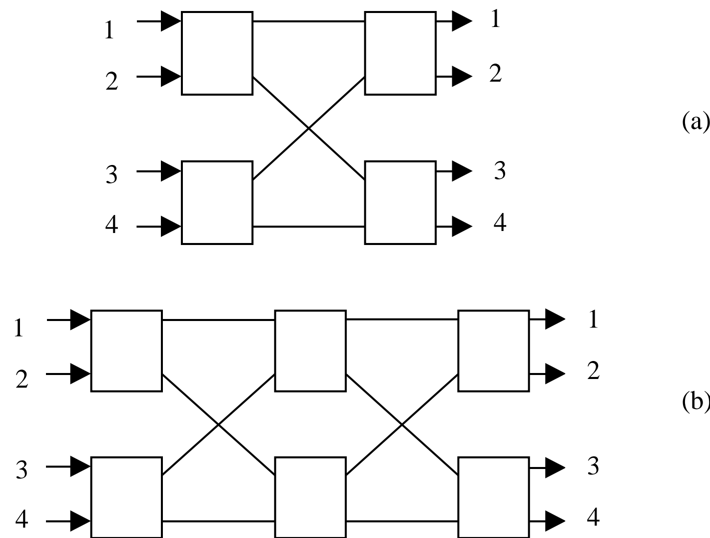


Figura 3.6: a) banyan network 4 4, b) Clos network 4 4 a 3 stadi.

Se poi consideriamo una velocità di trasmissione DS-Link di 9.5 Mbyte/s, e con un rapporto **r** = 1, il *throughput* per questa topologia avrà un valore pari a P_2 9.5 6.2 (Mbyte/s).

Per la *Clos network* il discorso è diverso: dal momento che la contesa può avvenire su tutti e tre gli stadi, la probabilità finale associata all' ultimo stadio è data da:

$$P_3(occupata) = 1 - (1 - \frac{1}{2})^{P_2 \;\; 2}$$

è evidente che in questo caso si ottiene un *throughput* finale più basso rispetto alla *banyan* poiché $P_2 < P_3$.

Nell' ipotesi però che si utilizzi lo *adaptive grouping routing* sul primo stadio della *Clos*, la contesa non sussiste più (sempre che il numero degli ingressi sia al numero delle uscite) e pertanto il *throughput* avrà lo stesso valore della precedente *banyan*.

Questi calcoli approssimati dimostrano che in linea teorica, e sotto opportune condizioni, le due topologie presentano medesime caratteristiche di traffico, ma un numero diverso di dispositivi: scegliere una *Clos* sembrerebbe inefficiente dal momento che, oltre a richiedere dei costi maggiori rispetto alla *banyan*, non offre incrementi di prestazioni. Occorre precisare però che una topologia *Clos* può essere realizzata in diversi

modi e quella che abbiamo trattato adesso è strutturata con il seguente criterio: il numero di switch dello stadio intermedio è pari al numero di porte di ingresso del singolo switch del primo stadio. In figura 3.5 si avrebbe **n** = **m**; la rete così costruita prende il nome di *Rearrangeable Clos*.

Se invece si costruiscono *Clos network* con un numero di switch intermedi pari a **m** = 2**n** –1, si ottiene la cosidetta *Non-blocking Clos network* (vedi figura 3.7); con essa è allora possibile aumentare il *throughput* rispetto alla *banyan network*.

Infatti con questa topologia scegliendo **n** = 2 e quindi **m** = 3, ed utilizzando il *grouping* sempre al primo stadio, si ottiene per $P'_1(occupata)$

$$P'_1(occupata) = 1 - (1 - \frac{1}{3})^2$$

per $P'_2(occupata)$

$$P'_2(occupata) = 1 - (1 - \frac{1}{2})^{P'_1 \;\; 2}$$

mentre per la $P'_3(occupata)$ si avrà

$$P'_3(occupata) = 1 - (1 - \frac{1}{2})^{P'_2 \;\; 3} \qquad (10)$$

ed un *throughput* uguale a P'_3 9.5 6.4 (Mbyte/s), leggermente superiore a quello offerto dalla *banyan network*.

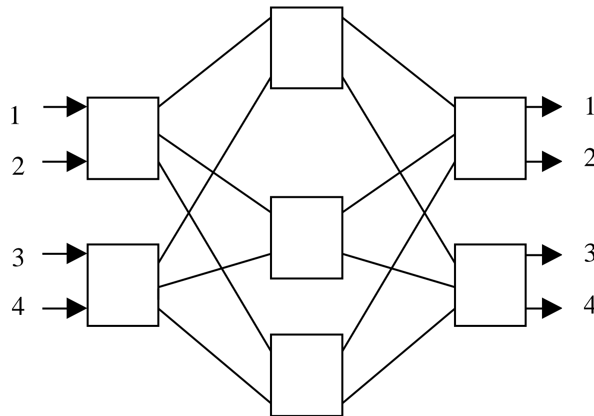


Figura 3.7: Rappresentazione di una Non-blocking Clos network di tipo 4 4, avente 3 switch nello stadio intermedio.

Da questi semplici calcoli si evince che la topologia *Non-blocking Clos network* sia quella che riesce ad offrire, per un traffico di tipo casuale, le migliori prestazioni. Per contro c'è da considerare l'aumento del numero di switch e delle connessioni da realizzare che tale topologia comporta. Se spostiamo queste considerazioni verso le reti d'interconnessione per il trigger di 2° livello (nelle diverse architetture), la scelta della topologia ottimale coinvolge anche altri diversi aspetti e parametri di valutazione: numero dei pacchetti, loro frequenza, loro dimensioni variabili. Un ulteriore problema è che non si conosce tuttora il modello del traffico attraverso quella che sarà la rete del trigger di 2° livello.

Nel presente lavoro si vuole comunque fornire una serie di informazioni, che possano concorrere a trovare un soddisfacente modello teorico nella trattazione di alcune architetture di tipo MIN. In tal senso sarà interessante vedere quanto il modello di contesa adottato simuli il comportamento reale delle diverse reti di switch; nel Capitolo 5 verrà affrontato proprio questo aspetto, durante l'analisi del traffico su piccole switching network realizzate in laboratorio.

funzioni di comunicazione trasferendo o ricevendo i dati su due semplici variabili, anziché inviare o ricevere i dati attraverso delle aree di memoria dedicata: questa strategia è stata adottata con il solo fine di valutare le prestazioni, in termini di velocità, sia della tecnologia DS-Link implementata su bus locale PCI, che delle reti di switch.

Poiché il *device driver* interagisce direttamente con il kernel è possibile far gestire a quest'ultimo [31] [22], nella fase di comunicazione, o altri processi o solamente le trasmissioni DS-Link. Nel primo caso si introduce un tempo di attesa per la comunicazione DS-Link, durante il quale il kernel può operare l'attività di "schedulazione" sugli altri processi; abbreviando tale tempo si riesce ad aumentare la priorità della comunicazione DS-Link rispetto agli altri processi, incrementandone le prestazioni. Nella realizzazione delle comunicazioni DS-Link si è scelta proprio questa modalità di lavoro perché per sfruttare una particolare configurazione, come vedremo negli ultimi paragrafi, deve essere garantita l'attività delle trasmissioni attraverso la rete Ethernet. Tra l'altro da misure effettuate il guadagno in termini di prestazioni che si ottiene, bloccando gli altri processi che non siano comunicazioni DS-Link, è molto modesto.

Si riporta un breve esempio, solo di trasmissione, di una funzione **txrx_ds** (impiegata per una tecnica di misura, *round trip time*, che tra poco vedremo) utilizzata dallo spazio utente per trasmettere e ricevere in modo sequenziale dei pacchetti le cui dimensioni e quantità devono essere preliminarmente impostate; essa è implementata nel *device driver* attraverso uno dei vari *case* presenti nell'istruzione **pcidsl_ioctl** (vedi paragrafo 4.3).

Nel codice relativo alla **txrx_dsl** si è indicato in grassetto il parametro (`WAIT_FOR`) che modifica i tempi di attesa della comunicazione DS-Link, e quindi la priorità della funzione stessa.

```
case TXRX_DS:
{
       struct TXRX_ds ioctlTXRX;
       unsigned int start, stop, i, j, time;
       unsigned int Transmit;

     ret = verify_area (VERIFY_WRITE, (void *) arg,
                          sizeof (struct TXRX_ds));
     if (ret)
        return ret;
     ret = verify_area (VERIFY_READ, (void *) arg,
                          sizeof (struct TXRX_ds));
     if (ret)
        return ret;
```

```
struct pcidsl_physarea
{
    unsigned long pci_addr;
    size_t size;
    pid_t pid;
    struct pcidsl_physarea *next;
    struct pcidsl_physarea *prev;

};

struct pcidsl_dev_info
{
    struct pcidsl_physarea *physhead;
};

/* major number del driver */
static int pcidsl_major;

/* informazioni per il device */
static struct pcidsl_dev_info dev_info[MAX_NODES];
static int numnodes=1;
static unsigned short adr0, adr1 = 0;
static const char devname[]="PCIdsl";
static inline int clog(long x)
{
    const int fs=ffz(~x);
    if ((1<<fs)==x)
        return fs;
    else
        return fs+1;
}

static int physfree(struct pcidsl_dev_info *dip, struct pcidsl_physarea
*pa)
{
    free_pages((unsigned long)bus_to_virt(pa->pci_addr),
                clog(pa->size)-PAGE_SHIFT);

    if (pa->next)
        pa->next->prev=pa->prev;
    if (pa->prev)
        pa->prev->next=pa->next;
    else if (pa==dip->physhead)
        dip->physhead=pa->next;

    kfree(pa);

    return 0;
}

static int pcidsl_ioctl(struct inode *inode, struct file *file,
                        unsigned int cmd, unsigned long arg)
{
    int ret;
    unsigned int minor=MINOR(inode->i_rdev);
```

```
    if (minor>=numnodes)
        return -ENODEV;

    switch (cmd) {

    case PCIBIOS_PRESENT:
    {
       struct pci_address ioctladd;
       char bus_num,dev_fun;


       cli();
       if (pcibios_present())
       {
         if (pcibios_find_device(0x1234, 0x5678, 0x00,
             &bus_num,&dev_fun) == 0)
         {

           ret = verify_area (VERIFY_WRITE, (void *) arg,
                         sizeof (struct pci_address));
           if (ret)
            return ret;

           pcibios_read_config_word(bus_num, dev_fun,
                                PCI_CS_BASE_ADDRESS_0,&ioctladd.bus0);

           pcibios_read_config_word(bus_num, dev_fun,
                                PCI_CS_BASE_ADDRESS_1,&ioctladd.bus1);

           adr0 = (ioctladd.bus0 & 0xfffc)|AMCC_OP_REG_FIFO;
           adr1 = ioctladd.bus1 & 0xfffc;

           ioctladd.val = 0; /*per ritornare la corretta operazione di
lettura */

           copy_to_user ((struct pci_address *) arg, &ioctladd,
                    sizeof (struct pci_address));

           pcibios_write_config_word(bus_num, dev_fun,
                                 PCI_CS_COMMAND, IO_ACCESS_ENABLE);

       }
       else if(pcibios_find_device(0x1234, 0x5678, 0x00,
                                  &bus_num, &dev_fun)!= 0)
       {
           printk (KERN_INFO" DEVICE NOT FOUND \n ");

           ret = verify_area (VERIFY_WRITE, (void *) arg,
                         sizeof (struct pci_address));
           if (ret)
            return ret;

           ioctladd.val = 1; /*per ritornare errore nella operazione di
lettura */
```

```
     copy_to_user ((struct pci_address *) arg, &ioctladd,
              sizeof (struct pci_address));
 }

}
sti();
break;
}
case DSINB:
{
   struct inb_ds ioctlinb;

   ret = verify_area (VERIFY_WRITE, (void *) arg,
                   sizeof (struct inb_ds));
   if (ret)
      return ret;
   ret = verify_area (VERIFY_READ, (void *) arg,
                   sizeof (struct inb_ds));
   if (ret)
      return ret;

   copy_from_user (&ioctlinb,(struct inb_ds *) arg,
              sizeof (struct inb_ds));

   ioctlinb.data = inb(ioctlinb.address);

   copy_to_user ((struct inb_ds *) arg, &ioctlinb,
              sizeof (struct inb_ds));
   break;
 }

 case DSOUTB:
 {
    struct outb_ds ioctloutb;

    ret = verify_area (VERIFY_READ, (void *) arg,
                   sizeof (struct outb_ds));
    if (ret)
       return ret;

    copy_from_user (&ioctloutb,(struct outb_ds *) arg,
              sizeof (struct outb_ds));

    outb(ioctloutb.data, ioctloutb.address);

    break;
 }
 case INLDSLINK:
 {
       struct inlDS_ds ioctlinds;
       int count;

       ret = verify_area (VERIFY_WRITE, (void *) arg,
                   sizeof (struct inlDS_ds));
       if (ret)
            return ret;
```

```
 ret = verify_area (VERIFY_READ, (void *) arg,
            sizeof (struct inlDS_ds));
 if (ret)
    return ret;

 copy_from_user (&ioctlinds,(struct inlDS_ds *) arg,
       sizeof (struct inlDS_ds));

 for(count=0;count<((ioctlinds.numbyte)/4);count++)
        ioctlinds.data = inl(ioctlinds.address);

 copy_to_user ((struct inlDS_ds *) arg, &ioctlinds,
             sizeof (struct inlDS_ds));
  break;
}
case OUTLDSLINK:
{
  struct outlDS_ds ioctloutds;
  int count;

  ret = verify_area (VERIFY_READ, (void *) arg,
            sizeof (struct outlDS_ds));
  if (ret)
      return ret;

  copy_from_user (&ioctloutds,(struct outlDS_ds *) arg,
       sizeof (struct outlDS_ds));
  for(count=0;count<((ioctloutds.numbyte)/4);count++)

  outl(ioctloutds.data, ioctloutds.address);

   break;
  }
  case TXRX_DS:
  {
    struct TXRX_ds ioctlTXRX;
    unsigned int start, stop, i, j/*, time*/;
    unsigned int Transmit, Receive =0;

    ret = verify_area (VERIFY_WRITE, (void *) arg,
            sizeof (struct TXRX_ds));
    if (ret)
        return ret;
    ret = verify_area (VERIFY_READ, (void *) arg,
            sizeof (struct TXRX_ds));
    if (ret)
        return ret;

    copy_from_user (&ioctlTXRX,(struct TXRX_ds *) arg,
       sizeof (struct TXRX_ds));

    Transmit = 0x1234567;
    start = stop = 0;
    outw(ACK_EOM_RXED|ACK_EOP_RXED|ACK_ACK_RXED|
         ACK_HDR_VALID,adr1|RX_ACKNOWLEDGE);
    cli();
```

```
rdtsc(start); /* avvio del TSC */

for (i=0;i<ioctlTXRX.cicli;i++)
{
      /* Attivazione trasmissione */
   outw(TERMINATOR_ENABLE|EOP_TERMINATOR|
        HEADER_ENABLE|HEADER_SELECT_0|
        ioctlTXRX.numbyte,adr1|TX_SEND_PACKET);

      /* Check per trasmissione */
    do
    {
    } while(((inw(adr1|ISR))&
                TX_FIFO_EMPTY)==0x0L);

    current->state = TASK_INTERRUPTIBLE;
    current->timeout = jiffies+WAIT_FOR;
    schedule();

    for(j=0;j<((ioctlTXRX.numbyte)/4);j++)
       outl(Transmit, adr0);

    }
    for (i=0;i<ioctlTXRX.cicli;i++)
    {
         /* Check per ricezione */
      do
      {
      } while(!(inw(adr1|ISR)&
         (HEADER_VALID|PACK_RECEIVED)));

       current->state = TASK_INTERRUPTIBLE;
       current->timeout = jiffies/*+JIFFIES_TO_WAIT*/;
       schedule();

       for(j=0;j<((ioctlTXRX.numbyte)/4);j++)
            Receive = inl(adr0);

        /* Bit di Ack */
        outw(ACK_EOM_RXED|ACK_EOP_RXED|ACK_ACK_RXED|
             ACK_HDR_VALID, adr1|RX_ACKNOWLEDGE);


   }
   rdtsc(stop); /* stop lettura del TSC */

   ioctlTXRX.cnt = (stop-start);

   copy_to_user ((struct TXRX_ds *) arg, &ioctlTXRX,
          sizeof (struct TXRX_ds));

  sti();

  break;
}
```

```
                default : break;

        return -EINVAL;
    }
   return 0;
}

static int pcidsl_open(struct inode * inode, struct file * filp)
{
    printk(" %s: open\n",devname);

    MOD_INC_USE_COUNT;

    return 0;
}

static void pcidsl_release(struct inode * inode, struct file * file)
{
   printk ("%s: close\n",devname);

   MOD_DEC_USE_COUNT;
}

static struct file_operations pcidsl_fops = {
    NULL, /* per lseek  */
    NULL, /* per read   */
    NULL, /* per write  */
    NULL, /* per readdir*/
    NULL, /* per select */
    pcidsl_ioctl,
    NULL,
    pcidsl_open,
    pcidsl_release,

};

static void pcidsl_init_dev(struct pcidsl_dev_info *dip)
{
    dip->physhead=NULL;

}

static inline void pcidsl_deinit_dev(struct pcidsl_dev_info *dip)
{
    while (dip->physhead)
        physfree(dip, dip->physhead);
}

/* funzione per inizializzazione del driver */
int pcidsl_init(void)
{
    int stat;
    int node;
    struct pcidsl_dev_info *dip=dev_info;
```

```
    printk("%s: PCI Character Device Driver", devname);

    if ((stat=register_chrdev(0, devname, &pcidsl_fops))<0) {
        printk(": unable to get major (%d)\n", stat);
        return -1;
    }
    pcidsl_major=stat;

    printk(" (Major = %d)\n", pcidsl_major);

    memset(dev_info, 0, sizeof(dev_info));

    for (node=0; node<numnodes; ++node, ++dip)
        pcidsl_init_dev(dip);

    return 0;
}

static int pcidsl_deinit(void)
{
    int node;
    struct pcidsl_dev_info *dip=dev_info;

    for (node=0; node<numnodes; ++node, ++dip)
        pcidsl_deinit_dev(dip);

    unregister_chrdev(pcidsl_major, devname);

    return 0;
}

#ifdef MODULE

int init_module(void)
{
    pcidsl_init();
    return 0;
}

void cleanup_module(void)
{
    pcidsl_deinit();
    printk("%s: removed\n", devname);
}

#endif
```

Il programma seguente è utilizzato per sfruttare le funzioni presenti nel *character device driver*, ma dallo spazio utente.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <sys/mman.h>

#include "PCIdsl_int.h"
#include "mypci.h"

int Major_number;

int pcidsl_init(void)
{
    Major_number=open("/dev/PCIdsl", O_RDWR);

    if (Major_number == -1)
        return -1;

    return 0;
}

int my_pcibios_present(unsigned short *bas_adr0, unsigned short
*bas_adr1)
{
    struct pci_address pp;

    if (ioctl(Major_number,PCIBIOS_PRESENT, &pp))
       printf(" pcibios_present ioctl failed \n");

    *bas_adr0 = pp.bus0;
    *bas_adr1 = pp.bus1;

    return pp.val;
}
unsigned char my_inb(unsigned short address)
{
     struct inb_ds DS;

     DS.address = address;

     if (ioctl(Major_number, DSINB, &DS))
        printf(" DSinb ioctl failed \n");

     return DS.data;
}
```

```
unsigned short my_inw(unsigned short address)
{
     struct inw_ds DS;

     DS.address = address;
     if (ioctl(Major_number, DSINW, &DS))
        printf(" DSinw ioctl failed \n");

     return DS.data;
}

unsigned long my_inl(unsigned short address)
{
     struct inl_ds DS;

     DS.address = address;
     if (ioctl(Major_number, DSINL, &DS))
        printf(" DSinl ioctl failed \n");

     return DS.data;
}
void my_outb(unsigned char data, unsigned short address)
{
     struct outb_ds DS;

     DS.address = address;
     DS.data = data;

     if (ioctl(Major_number, DSOUTB, &DS))
        printf( "DSoutb ioctl failed \n");

}
void my_outw(unsigned short data, unsigned short address)
{
     struct outw_ds DS;

     DS.address = address;
     DS.data = data;

     if (ioctl(Major_number, DSOUTW, &DS))
        printf( "DSoutw ioctl failed \n");

}
void my_outl(long int data, unsigned short address)
{
     struct outl_ds DS;

     DS.address = address;
     DS.data = data;

     if (ioctl(Major_number, DSOUTL, &DS))
        printf( "DSoutl ioctl failed \n");

}
```

```
unsigned long int my_inldslink(unsigned short address, unsigned short
numbyte)
{
     struct inlDS_ds inl;

     inl.address = address;
     inl.numbyte = numbyte;

     if (ioctl(Major_number,INLDSLINK, &inl))
        printf( "inldslink ioctl failed \n");

     return  inl.data;
}
unsigned long int my_outldslink(long int data, unsigned short address,
                  unsigned short numbyte)
{
     struct  outlDS_ds outl;

     outl.address = address;
     outl.data = data;
     outl.numbyte = numbyte;

     if (ioctl(Major_number,OUTLDSLINK, &outl))
        printf( "outldslink ioctl failed failed \n");

     return  outl.data;
}

void txrx_ds(unsigned short numbyte, unsigned short cicli,
           unsigned int *cnt)
{
     struct TXRX_ds TXRX;

     TXRX.numbyte = numbyte;
     TXRX.cicli = cicli;

     if (ioctl(Major_number,TXRX_DS, &TXRX))
        printf( "TXRX_DS ioctl failed \n");

     *cnt = TXRX.cnt;
}
void rxtx_ds(unsigned short numbyte, unsigned short cicli)
{
     struct RXTX_ds RXTX;

     RXTX.numbyte = numbyte;
     RXTX.cicli = cicli;

     if (ioctl(Major_number,RXTX_DS, &RXTX))
        printf( "RXTX_DS ioctl failed \n");

}
```

Questo codice, chiamato **config.c**, è relativo al Controlling process che permette la configurazione, ed il controllo di switching network, nonchè la funzione Master per l'interconnessione con Ethernet.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <asm/io.h>
#include <sys/time.h>
#include <errno.h>
#include <string.h>
#include <signal.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <string.h>
#include <sys/wait.h>
#include <ctype.h>
#include <netdb.h>
#include "amcc.h"
#include "mypci.h"
#include "dslink.h"


#define MYPORT 4000
#define NOT_NUMBER_DEVICES  -1


void main(void)
{
   int           NumberOfDevices;
   unsigned short ii =0;
   int               val=0;
   unsigned short base_adr0, base_adr1;
   int             op_regs_base,c101_regs;
   int             k,Status,Temp;
   char            read[1], input[10];
   int           sock,WhichHost;
   struct   sockaddr_in my_addr;
   struct   sockaddr_in their_addr[10];
   struct   hostent *host;
   int               addr_len, numbyterx,numbytetx,i,j;
   int               NumberSlave;
   unsigned short buf[100],data[10][100];
   char     insert[10];


   /* Apertura dello special file /dev/PCIdsl */
   if(pcidsl_init())
   {
     printf(" \n\n\n ");
     printf(" Error File: Unable to open the special file ->
              /dev/Pcidsl \n\n\n");
     exit(1);
```

```
}

/* Inizializzazione spazio PCI e lettura base address */
if(my_pcibios_present(&base_adr0,&base_adr1))
{
   printf(" \n\n\n ");
   printf(" Error PCI: (maybe PCIdriver is not loaded),
            or DSLINK-CARD NOT FOUND   \n\n\n");
   exit(1);
}

op_regs_base = base_adr0 & 0xfffc;
c101_regs = base_adr1 & 0xfffc;

printf(" \n\n\n");
printf ("****   Initial value   **** \n\n");
printf ("BASE ADR 0 = %x\n", base_adr0);
printf ("BASE ADR 1 = %x\n", base_adr1);
regs_base(c101_regs); /* Funzione che ritorna il base address + offset */

/* Trasferisce l'indirizzo del c101 al file txrx.c */
Device_Id();

/* S5933 - Reset */
my_outl(ADD_ON_RESET, op_regs_base|AMCC_OP_REG_MCSR);
my_outl(0, op_regs_base|AMCC_OP_REG_MCSR);

my_outw(TOKEN_DISABLE, c101_regs|ICR);

my_outw(RESET_LINK, c101_regs|LINK_COMMAND);

my_outw(WRITE_1|_50MBIT|SPEED_SELECT|
        LOCALIZE_ERROR, c101_regs|LINK_MODE);

/* Abilitazione Interrupts */
my_outw(INTERRUPTS_ENABLED, c101_regs|ENABLE_INTERRUPTS);

my_outw(SEND_PACKET|C101_TX_LEVEL, c101_regs|TX_INTERRUPT_ENABLE);
my_outw(TX_LEVEL_HIGH|TX_LEVEL_62, c101_regs|TX_LEVEL);

/* Attivazione pacchettizzazione */
my_outw(SUPPRESS_RX_EOX|ENABLE_PACKETIZATION|RX_HEADER_LENGTH_1,
c101_regs|DEVICE_CONFIG);

my_outw(START_LINK, c101_regs|LINK_COMMAND);

if((sock = socket(AF_INET,SOCK_DGRAM,0)) == -1)
{
   perror("socket");
   exit(1);
}

 my_addr.sin_family = AF_INET;
 my_addr.sin_port = htons(MYPORT);
 my_addr.sin_addr.s_addr = INADDR_ANY;
 bzero(&(my_addr.sin_zero),8);
```

```
    if(bind(sock,(struct sockaddr *)&my_addr,
        sizeof(struct sockaddr)) == -1)
    {
      perror("bind");
      exit(1);
    }

    Status = 0; /* Variabile usata per rimanere o uscire dalla
                    trasmission session */
    ii =0;
    NumberOfDevices = 0;
    do
    {
      read[0] = 'x';
      printf("\n\n");
      gets(read);

      /******************************/
      /* Start configuration  C104 */
      /******************************/

      if (read[0] == 'c')
       {
          ii++;
          if(ii ==1)
          {
            NumberOfDevices = Start_Network(); /*ritorna il numero di
devices */
          if(NumberOfDevices == 0)
            {
              printf(" No devices in this network \n");
              read[0] = 'e';
            }
            else if(NumberOfDevices == NOT_NUMBER_DEVICES)
            {
              read[0] = 'e';
            }

            Configure(&NumberOfDevices,
                      NOT_NUMBER_DEVICES); /* configura i o il C104 */

          else if (ii >1)
          {
            ack_packet(0);
            delay_old(1000);
            Configure(&NumberOfDevices,NOT_NUMBER_DEVICES);
        }
          if((NumberOfDevices > 0) &
             (NumberOfDevices != NOT_NUMBER_DEVICES))
          {
            printf("\n\n");
            printf("CONFIGURATION O.K. & NUMBER OF
                    DEVICES IS %d\n",NumberOfDevices);
          }
      }
```

```
/******************************/
/* ISPY link configuration   */
/******************************/

if (read[0] == 'i')
 {

      ISPY(NumberOfDevices);  /* descrive le connessioni
link-link */

 }

/******************************/
/* Status register      C104 */
/******************************/
if (read[0] == 's')
{
   if(NumberOfDevices ==1)
   {
    CPeek(0) ;
   }
   else if (NumberOfDevices >1)
   {
     do
     {
       printf("Insert STC 104's number,
                Max is %d\n",NumberOfDevices);
       gets(input);
     ii = atoi(input);

     if((ii <= NumberOfDevices)&& ii !=0)
     {
         CPeek((ii -1));
         val = 0;
       }
       else if(ii > NumberOfDevices )
       {
         printf("The number is out of the range \n\n");
         val =1;
       }
     }while(val!=0);
   }

  }
```

```
  /***********************************/
  /* Data Transmission with IP-UDP   */
  /***********************************/

printf(" \n\n\n");
printf("  ********   YOU ARE INSIDE THE TRANSMISSION **** \n\n\n");
printf(" How many slaves ? \n");
gets(insert);
HowManySlave = atoi(insert);

/*Fase iniziale di riconoscimento degli slave */
while(NumberSlave<HowManySlave)
{
   for(j=0;j<100;j++)
      buf[j] = 0;


   addr_len = sizeof(struct sockaddr);
   if((numbyterx=recvfrom(sock,buf,6,0,(struct sockaddr*)&
   their_addr[NumberSlave],
   &addr_len)) == -1)
   {
     perror("recvfrom");
   exit(0);
   }

   if ((host = gethostbyaddr((char*)&
        their_addr[NumberSlave].sin_addr,sizeof
        their_addr[NumberSlave].sin_addr,AF_INET)) == NULL)
   {
     perror("gethostbyaddr");
   }

   if((numbytetx=sendto(sock,buf,numbyterx,0,(struct sockaddr *)&
   their_addr[NumberSlave],sizeof(struct sockaddr))) == -1)
   {
     perror("sendto");
     exit(1);
   }

   printf("THE HOST (NUMBER) %d  IS -> %s \n",(NumberSlave+1),
           host->h_name);

   NumberSlave++;
   delay_old(3000000);

}/*chiusura del while */


WhichHost = EndWhile =  Status = ZeroHost = FALSE;
Transmit = EndTotalWhile = TRUE;

while(EndTotalWhile != FALSE)
{

 while(WhichHost<HowManySlave)
 {
```

```
do
{
  if ((host = gethostbyaddr((char *)&
      their_addr[WhichHost].sin_addr,
      sizeof their_addr[WhichHost].sin_addr,AF_INET)) == NULL)
  {
    perror("gethostbyaddr");
  }
  printf("\n\n\n");
if((Status > 0) & (WhichHost == 0))
{
  printf("VUOI CONTINUARE CON LA TRASMISSIONE (y/n)? \n");
  gets(insert);

  if(insert[0] == 'y')
  {
    Transmit = TRUE;
    EndTotalWhile = TRUE;
  }
  else if(insert[0] == 'n')
  {
    for(i =0;i<HowManySlave;i++)
    {
       data[i][4] = 5;
    }
    EndTotalWhile = FALSE;
    Transmit = FALSE;
    EndWhile = TRUE;
    WhichHost = HowManySlave;
  }

}
if (Transmit == TRUE)
{
    printf(" LO SLAVE DA CONFIG. %s \n\n\n",host->h_name);
    printf("Trasmissione o ricezione ? (1 opp. 9) \n");
    gets(insert);
    data[WhichHost][0] = atoi(insert);
    printf("INSERIRE IL NUM. DEL LINK SCELTO \n");
    gets(insert);
    data[WhichHost][1] = atoi(insert);

  if(WhichHost > 0)
  {
    data[WhichHost][2] = data[ZeroHost][2];
    data[WhichHost][3] = data[ZeroHost][3];
  }
  else if(WhichHost == 0)
  {
    printf("INSERIRE IL NUM. DEI BYTES DA SPEDIRE (MAX 64) \n");
      gets(insert);
    data[WhichHost][2] = atoi(insert);
    printf("INSERIRE IL NUM. DEI CICLI \n");
      gets(insert);
      data[WhichHost][3] = atoi(insert);
  }
  printf("CONFERMI I DATI (y/n) \n");
```

```
          gets(insert);

          if(insert[0] == 'y')
          {
             EndWhile = TRUE;
             data[WhichHost][4] = 7; /*Per continuare la trasmissione */
          }
          else if(insert[0] == 'n')
             EndWhile = FALSE;

         }/*Fine dell' IF (Transmit) */

        }while(EndWhile != TRUE);/* Fine della procedura do-while */

        WhichHost++;

      }/*Chiusura del while per trasferire i valori agli slaves */

       if(EndTotalWhile != FALSE)
       {
         printf("Press Enter for start transmission ...... \n");
         getchar();
       }

       WhichHost = 0;

       /* Trasmissione delle informazioni agli slaves */
       while(WhichHost<HowManySlave)
       {

         if((numbytetx=sendto(sock,data[WhichHost],10,0,(struct sockaddr
*)&
         their_addr[WhichHost],sizeof(struct sockaddr))) == -1)
         {
           perror("sendto");
           exit(1);
         }

         WhichHost++;
       }/* Fine del while per trasmissione delle informazioni */


       for(i=0;i<HowManySlave;i++)
       {
          for(j=0;j<10;j++)
             data[i][j] = 0;
       }

       WhichHost = 0;
       EndWhile  = FALSE; /*Permette di riprendere la trasmissione*/
       Status++; /* Permette di rimanere nella trasmissione */

     } /* Fine del while "inside transmission" */
  printf("\n\n\n");
  close (sock);

}
```

Bibliografia

[1] AMCC, *S5933 PCI Matchmaker Controller Data Book*, Applied Micro Circuit Corporation, 1996.

[2] *A Survey of ATM switching Techniques*, http://www.cis.ohio-state.edu/~jain/cis788/atm_switching/

[3] *ATLAS DAQ, EF, LVL2 and DCS Technical Progress Report*, CERN/LHCC/98-16, 1998.

[4] *ATLAS Technical Proposal*, CERN/LHCC 94-43, 1944.

[5] *ATLAS Trigger Performance Status Report*, CERN/LHCC/98-15, 1998.

[6] Ruggero Adinolfi, *Reti di computer*, Mc Graw-Hill Libri Italia srl, 1994.

[7] AT&T Microelettronics, *The 41 Series of High Performance Line Drivers, Receivers, Transceivers*, Application Note, 1992.

[8] *ATLAS Level-2 Trigger User Requirements Document*, ATLAS DAQ note79, 1997.

[9] Naba Barkakati, *I segreti di Linux*, Apogeo, 1996.

[10] D. Bertsekas, R. Gallager, *Data Networks*, Prentice Hall, 1992.

[11] C. Bouras, J. Garofalakis et al., *A General Performance Model for Multistage Interconnection Networks*, Proc. of the ACM SIGMETRICS Conference, short paper, 1990.

[12] D.E. Comer, D.L. Stevens, *Internetworking with TCP/IP - Vol. III*, Prentice Hall, 1994.

[13] R.W. Dobinson et al., *Evaluation of the Level-2 trigger, architecture B, on the Macrame Testbed,* ATLAS DAQ note 102, 1998.

[14] R.W. Dobinson, S. Haas, *Electrical and Optical Transmission Media (CERN)*, ftp://stdsbbs.ieee.org/pub/p1355/articles-and-data/fiber.ps/ .

[15] D. Francis, R.W. Dobinson et al., *A Local-Global Implementation of a Vertical Slice of the ATLAS Second Level Trigger*, ATLAS DAQ note 81, 1998.

[16] B.W. Kernighan, D.M. Ritchie, *Linguaggio C*, Edizioni Jackson, 1989.

[17] A. Kugel et al., *ATLAS Level-2 Trigger Demonstrator-A Activity Report – Part 2: Demonstrator Program*, ATLAS DAQ note 84, 1998.

[18] A. Kugel et al., *ATLAS Level-2 Trigger Demonstrator-A Activity Report – Part 2: Paper Model*, ATLAS DAQ note 101, 1998.

[19] R.E. Hughes-Jones et al., *Using SCI to Implement the Local-Global Architecture for the ATLAS Level 2 Trigger*, ATALS DAQ note 11, 1998.

[20] IEEE std. 1355, *Standard for Heterogeneous Interconnect (HIC) (Low Cost Low Latency Scalable Serial Interconnect for Parallel System Construction)*, IEEE inc., 1995.

[21] *Level-1 Trigger Technical Design Report*, CERN/LHCC 98-14, 1998.

[22] Michael K. Johnson, *The Linux Kernel Hackers' Guide*, Alpha version 0.6, http://linux.caspur.it/webpages/docsf.html, 1995.

[23] M.D. May, P.W. Thompson, P.H. Welch, *Networks, Routers and Transputers*, SGS-Thomson, 1993.

[24] Rachel Morgan, Henry McGilton, *Unix System V*, McGraw-Hill, 1988.

[25] *PC HTRAM Motherboard – IMS B108*, Data Sheet, SGS Thomson, 1994.

[26] PCI Special Interest Group, *PCI Local Bus*, http://www.pcisig.com/ .

[27] D.H. Perkins, *Introduction to High Energy Physics*, Addison-Wesley, 1987.

[28] J. Postel, *User Datagram Protocol*, RFC 768, 1980.

[29] Red Hat Development Team, *Red Hat Linux 4.2 Users Guide*, Red Hat Software Inc., 1996.

[30] Red Hat Development Team, *Red Hat Linux 5.1 Users Guide*, Red Hat Software Inc., 1998.

[31] A. Rubini, *Linux device drivers*, O'Reilly & Associates inc., 1998.

[32] D.A. Rusling, *The LINUX Kernel*,http://linux.caspur.it/webpages/docsf.html, 1998.

[33] Emilio Segrè, *Nuclei e Particelle*, Zanichelli, 1982.

[34] *STC101 Parallel DSLink Adaptor*, Data Sheet, SGS Thomson, 1997.

[35] *STC104 Asynchronous Packet Switch*, Data Sheet, SGS Thomson, 1995.

[36] W.R. Stevens, *UNIX Network Programming*, Prentice Hall, 1990.

[37] K.H. Sulanke, *PCI to Data Strobe Link - Description*, Data Sheet, Rev. 2.4 .

[38] *T9000 ToolSet Hardware Configuration Manual*, SGS Thomson, 1994.

[39] *The T9000 transputer*, Hardware Reference Manual, SGS Thomson, 1993.

[40] A.S. Tanenbaum, *Reti di Computer - Seconda edizione*, Edizioni Jakson, 1993.

[41] A.S. Tanenbaum, *Progettazione e sviluppo dei Sistemi Operativi*, Edizioni Jackson, 1988.

[42] P.W. Thomposon J.D. Lewis, *The STC104 Packet Routing Chip*, SGS-Thomson, 1995.

[43] *Using the RDTSC Instruction for Performance Monitoring*, http://developer.intel.com/drg/pentiumII/appnotes/RDTSCPM1.HTM#4.0 .

[44] J.F. Wakerly, *Microcomputer Architecture and Programming*, J. Wiley, 1981.

[45] Matt Welsh, *The Linux Bible*, Yggdrasil Computing Inc., 1995.

RINGRAZIAMENTI

In questa pagina vorrei rivolgere un pensiero, per me naturale, di gratitudine a coloro che hanno contribuito a fornirmi idee e suggerimenti per rendere questa tesi più interessante, e forse più vicina a come desideravo che fosse.

Le relatrici del lavoro, la Professoressa Lucia Zanello e la Dottoressa Speranza Falciano, sono senza dubbio le prime persone a cui vanno i miei più cari ringraziamenti. Sono state una preziosa fonte di stimoli, di spunti interessanti nonché tanta disponibilità, che forse non sempre un laureando può facilmente trovare.

Merita un dovuto ringraziamento il dottor Franco Marzano che mi ha fornito numerosi consigli ed idee molto interessanti, ed il dottor Matteo Mascagni.

Non potevano mancare poi i tanti cari amici, che mi hanno offerto un supporto non indifferente.

Alfredo e Luciano (vero *deus ex machina*) con cui ho diviso tanti momenti divertenti e… di puro "sacrificio". Sveva che da anni, e non so come, mi sopporta; Grazia che sdrammatizza ogni mio dramma; Elisabetta che mi mette sempre di buon umore (forse dovrei decidermi a restituirle i libri…). Ci sono poi Lucia, Fabiola, Alessandro (Cavagnoli) ed Alessandro (DiMattia).

Infine un grazie di cuore e denso di affetto alle due persone che in questi anni mi hanno sostenuto e che ho sempre avuto vicino: Papà e Valeria.

```
            copy_from_user (&ioctlTXRX,(struct TXRX_ds *) arg,
                             sizeof (struct TXRX_ds));



    /* Inizializzazione variabile di trasmissione */
              Transmit = 0x1234567;
              start = stop = 0;

            rdtsc(start); /* start lettura del TSC */

            for (i=0;i<ioctlTXRX.cicli;i++)
            {
               /* Attivazione trasmissione */
               outw(TERMINATOR_ENABLE|EOP_TERMINATOR|
                    HEADER_ENABLE|HEADER_SELECT_0|
                    ioctlTXRX.numbyte,adr1|TX_SEND_PACKET);

               /* Check per trasmissione */
               do
               {
               }while(((inw(adr1|ISR))&
                             TX_FIFO_EMPTY)==0x0L);

                for(j=0;j<((ioctlTXRX.numbyte)/4);j++)
                      outl(Transmit, adr0);

                current->state = TASK_INTERRUPTIBLE;
                current->timeout = jiffies+WAIT_FOR;
                schedule();

             }
                 ..............................

               rdtsc(stop); /* stop lettura del TSC */

               ioctlTXRX.cnt = (stop-start);

               copy_to_user ((struct TXRX_ds *) arg,
                             &ioctlTXRX,sizeof (struct TXRX_ds));

               break;
           }
```

Nella prima parte del *case* relativo alla **txrx_dsl**() viene utilizzata due volte la funzione *verify_area()*; riferendoci sempre al paragrafo 4.2, questa sub-funzione controlla se vi sia area di memoria-kernel disponibile, per essere poi usata nel trasferire (`VERIFY_WRITE`) o ricevere informazione dallo spazio utente (`VERIFY_READ`): in questo caso *verify_area()* è usata in entrambe le modalità. Infatti nella **txrx_dsl()** è presente sia la istruzione *copy_from_user()*, con cui viene trasferito dallo spazio utente allo spazio

kernel il numero dei pacchetti da inviare e le loro dimensioni, che l'istruzione *copy_to_user()* con cui viene restituito allo spazio utente una misura del tempo impiegato dalla **txrx_dsl()**.

Infine con le assegnazioni:

```
current->state = TASK_INTERRUPTIBLE;
current->timeout = jiffies+WAIT_FOR;
schedule();
```

si è posto che il processo in atto possa essere interrotto (`TASK_INTERRUPTIBLE`), che deve andare in attesa per tempi che vanno dal centesimo di secondo (`jiffies`), imposto dal kernel, ad un valore predefinibile (`jiffies+WAIT_FOR`), e che altri processi possano essere "schedulati" dal kernel (`schedule()`).

Utilizzando opportunamente le diverse funzioni per la comunicazione, sono stati realizzati dei programmi di test tra più PC contenenti ognuno un'interfaccia PCI-DSLink. Uno di tali test, effettuato usando due soli PC e che verrà illustrato nel prossimo paragrafo, è basato sulla tecnica *round-trip-time* [40]; su ogni host è presente un processo che di volta in volta può essere trasmittente o ricevente. Nella prima fase uno dei due host trasmette all'altro un determinato flusso di dati, mentre nella seconda fase lo host che ha ricevuto i dati li ritrasmette al primo; per misurare la velocità di trasferimento dei dati viene preso il tempo, sul primo host, tra l'inizio della trasmissione e la fine della ricezione. Considerando la quantità di byte trasferiti (in una sola direzione) e la metà esatta del tempo intercorso, si perviene alla velocità del flusso dati tra i due host: in questo modo la misura non viene influenzata dal fatto che un host risulti più veloce dell'altro nella fase di ricezione, o di trasmissione.

5.2 Velocità di trasmissione delle interfacce PCI-DSLink

In questo paragrafo si riporteranno dei test di misura per studiare le prestazioni delle interfacce, e le massime velocità di trasferimento ottenibili.

Per garantire una certa uniformità nella misura, sono stati utilizzati due PC equipaggiati con microprocessore Pentium con uguale frequenza di clock (100 MHz), aventi le medesime caratteristiche tecniche: ossia stessa scheda madre, stesse periferiche

e controllori. Tale scelta è dettata dal fatto che diverse istruzioni contenute nel *device driver*, risentono delle caratteristiche hardware della macchina residente.

In questo ambito di misure le schede sono state impostate, via software, per funzionare in *Token-mode* (vedi par. 2.4 e 4.2.1), permettendo così di sfruttare la loro massima velocità operativa. Con il programma realizzato **test_mirror.c** sono state effettuate delle misure lavorando di volta in volta con pacchetti di dimensioni diverse (espresse in byte); dal momento che nel funzionamento in *Token-mode* è possibile leggere/scrivere parole di 32 bit (4 byte), le dimensioni dei pacchetti potranno avere solo valori multiplo di 4.

Sono state effettuate una decina di misure per 8 diversi tipi di pacchetto: 4096, 2048, 1024, 512, 256, 128, 64 e 32 byte. In ogni sessione di misura sono stati trasmessi e ricevuti 15000 pacchetti: questo alto valore permette di ridurre gli errori casuali. I tempi di comunicazione sono stati presi utilizzando un particolare registro a 64 bit del microprocessore Pentium chiamato TSC (*Time Stamp Counter*) [43]; in tale registro i bit vengono incrementati seguendo il regime di clock. Mediante una macro scritta in assembler i 32 bit meno significativi del TSC vengono trasferiti in una variabile: leggendo quest'ultima prima e dopo una funzione di comunicazione, si ha effettivamente il numero di cicli di clock intercorsi e quindi una misura, con precisione dipendente dal clock del sistema, del tempo impiegato nella fase di comunicazione.

Si riporta il codice della macro realizzata **rdtsc()** in cui *time* è la variabile utilizzata:

```
#define rdtsc(time) ( {__asm  (".byte 0x0f; .byte 0x31" \
        : "=eax" (time)  ) ; } )
```

Per quanto concerne gli errori associati alle velocità (calcolati attraverso la semidispersione massima), essi risultano di entità modesta; a ciò concorre, indubbiamente, il fatto che è stata utilizzata un'alta priorità nell'uso delle funzioni di comunicazione e quindi una sufficiente indipendenza da altri processi gestiti dal kernel. Nella figura 5.1 è riportato lo schema di collegamento tra i due PC.

Figura 5.1: Collegamento tra i due PC con raffigurati i due processi di trasmissione e ricezione; si noti che la trasmissione è half-duplex dal momento che i processi sono sequenziali.

Nella tabella 5.1 sono indicati i valori di velocità ottenuti in funzione delle diverse dimensioni dei pacchetti; si può osservare che per pacchetti di 4096 byte si raggiunge la massima velocità. Sapendo che la massima velocità di trasferimento delle informazioni su supporto DS-Link è di ~ 9.53 Mbyte/s, la velocità raggiunta di 9.56 Mbyte/s ci pone addirittura lievemente oltre il limite teorico. Probabilmente ci sono da considerare delle tolleranze nelle prestazioni massime della tecnologia, anche se ciò non è esplicitamente dichiarato dal costruttore.

Velocità (Mbyte/s)	**Dimensione pacchetti (byte)**
9.56 ± 0.03	4096
9.47 ± 0.04	2048
9.40 ± 0.02	1024
8.41 ± 0.02	512
7.21 ± 0.01	256
5.14 ± 0.01	128
3.27 ± 0.01	64
1.81 ± 0.02	32

Tabella 5.1: Valori delle velocità misurate per diversi tipi di pacchetti.

Si noti che le prestazioni delle schede diminuiscono riducendo le dimensioni dei pacchetti. Questo andamento è da imputare sostanzialmente a due fattori: il primo legato

al fatto che le dimensioni della testa, per piccoli pacchetti, cominciano ad avere rilevanza sul corpo completo, il secondo, e forse il più importante, dovuto al controllo software introdotto per evitare la saturazione delle FIFO e che nel programma deve essere ripetuto per ogni pacchetto. Quest'ultimo è evidentemente legato al kernel (tramite il *device driver*) ed i ritardi introdotti potrebbero essere minimizzati solamente con una sua implementazione nello hardware: in sostanza ciò che avviene sulle schede di comunicazione Ethernet.

Per meglio visualizzare le velocità di comunicazione dell'interfaccia, nella successiva figura 5.2 sono riportati in grafico i valori della precedente Tabella 5.1.

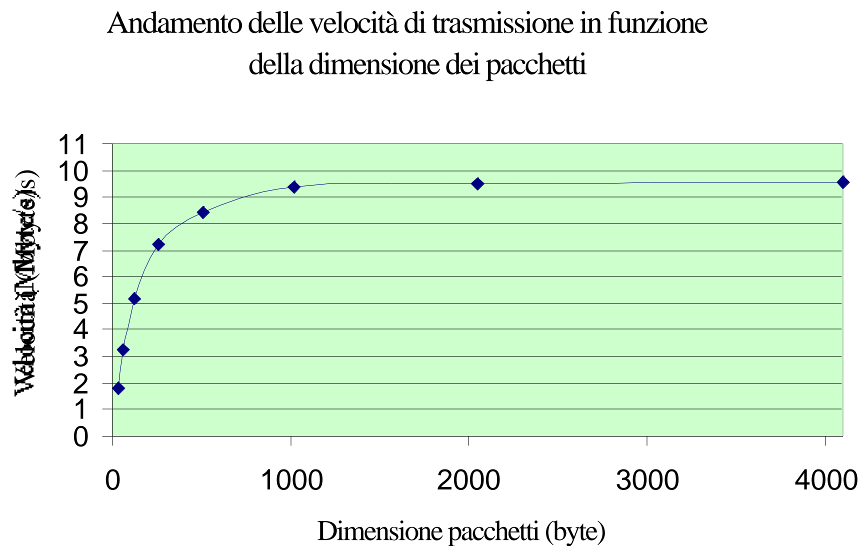


Figura 5.1: Curva rappresentativa della velocità di trasmissione tra PC1 e PC2 per diverse dimensioni dei paccheti di dato.

Ovviamente la precisione con cui sono state prese le misure è forse eccessiva, dal momento che cambiando configurazione del sistema e cioè altri modelli di PC, le velocità di trasmissione variano notevolmente. È però pur vero che le massime prestazioni non potranno mai superare i limiti oggettivi della tecnologia DS-Link, e tali valori sono stati ampiamente raggiunti nell'attuale configurazione.

5.3 Verifica del modello per la contesa nello switch

Nel paragrafo 3.3 è stato proposto un modello grazie al quale si forniscono gli strumenti per studiare il flusso informativo in diverse architetture di packet switch; è infatti attraverso l'uso di tale modello che siamo giunti nel paragrafo 3.4 alla determinazione teorica del *throughput* nelle topologie *banyan* e *Clos*.

La verifica sperimentale del modello è ovviamente legata alle disponibilità hardware, ovvero 4 interfacce PCI-DSLink dotate ciscuna di un singolo link. Sfruttando le possibilità della tecnologia DS-Link di permettere comunicazioni bidirezionali, ogni singola scheda, come già visto nel par. 3.4, ha la duplice funzionalità di sorgente o destinazione dei dati. In realtà dalle prove effettuate si riscontra che i ritardi introdotti dalle necessarie operazioni software, dallo host stesso e, non ultimo, dal meccanismo bidirezionale, non forniscono dei risultati apprezzabili.

Il problema è stato allora affrontato utilizzando le schede con una singola funzionalità: o sorgente o destinazione dei dati. Si sottolinea comunque che la configurazione precedente sarà utile nel contesto di misure per le reti di packet switch.

Le interfacce PCI-DSLink sono state inserite in 4 PC: due di essi sono quelli già utilizzati nel precedente paragrafo, con medesime caratterische, mentre gli altri due sono di prestazioni diverse (clock a 90 MHz e 166 MHz). Per agevolare la loro individuazione chiameremo PC1 e PC2 i precedenti host con clock a 100 MHz, mentre con PC3 lo host a 90 MHz ed infine PC4 quello a 166 MHz.

Gli host PC1 e PC2 sono stati scelti come sorgenti dei dati, mentre PC3 e PC4 come destinazioni; è stato inoltre generato del traffico con impostazione casuale delle destinazioni, e questo proprio in accordo con il modello. Le possibili "fonti" di generatori di numeri casuali (i valori da immetere nelle teste dei pacchetti) erano le funzioni ANSI C presenti nel complilatore GCC oppure lo switch stesso: tra le due non sono state riscontrate differenze in termini di prestazioni[1]. Per utilizzare il generatore di numeri casuali, **rand(),** del compilatore, è stata realizzata una semplice funzione chiamata **num_rand**().

[1] Sono stati fatti dei test sullo switch immettendo le teste random nei pacchetti, con entrambi i generatori, e non ci sono state variazioni nelle misure effettuate. Per questo motivo si vedrà più avanti che nella verifica delle diverse topologie, è stato indistintamente utilizzato o l' uno o l' altro generatore.

```
int num_rand(int range, int base)
{
     int val = 0;

     val = base + (int)((float)(range)*rand()/(RAND_MAX));
     return val;
}
```

Nella figura 5.2 è riportata la distribuzione su 5000 campioni dei numeri casuali compresi in un intervallo da 0 a 1000: come si può osservare tali numeri sono distribuiti in modo sufficientemente uniforme e quindi accettabili per i nostri scopi. In questo contesto di misure i tempi di comunicazione sono stati rilevati senza sfruttare la tecnica del *round trip time*, perché nel modello per la contesa sono previste comunicazioni unidirezionali: da sorgenti a destinazioni o viceversa ma non entrambi.

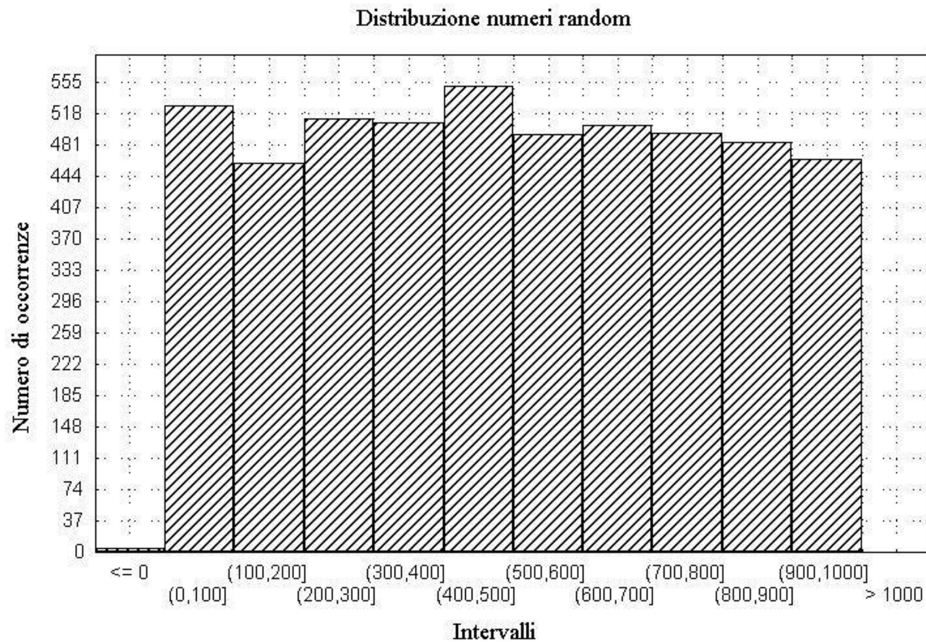


Figura 5.2: Distribuzione dei numeri da 0 a 1000 forniti dal generatore random in 5000 diversi campioni. In ascissa sono riportati i diversi intervalli tra 0 e 1000, mentre in ordinata i conteggi delle loro occorrenze.

Poiché le 4 schede PCI-DSLink sono state tutte utilizzate nei test, lo switch è stato configurato mediante una scheda ISA-DSLink di valutazione (B108 della SGS-Thomson)

attiva sul sistema operativo MS-DOS [25]. Sfruttando un programma di test associato alla scheda, è stato possibile configurare lo switch con un file di configurazione precedentemente ottenuto dal compilatore NDL (vedi par. 4.2.2). Nella figura 5.3 è illustrato lo schema di collegamento dei diversi host attraverso un unico switch STC 104, controllato da una scheda B108.

Come test iniziale è stata misurata la velocità di trasmissione di PC1 e PC2, singolarmente, avendo impostato le teste dei pacchetti per un traffico random sulle due destinazioni e ponendo queste ultime, PC3 e PC4, in ricezione continua. Il valore rilevato sui due PC è stato leggermente diverso ed è stato preso facendo 10 misure diverse, per 15000 pacchetti di 2048 byte .

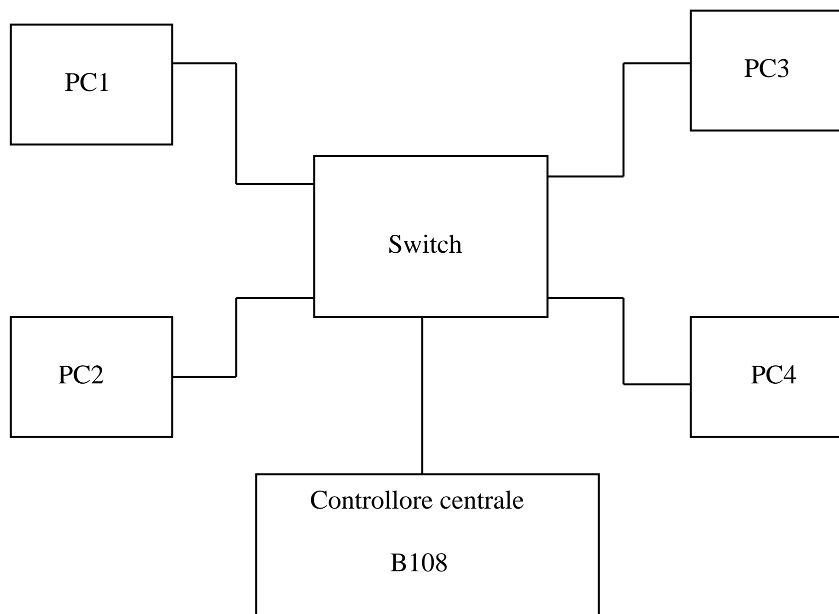


Figura 5.3: Collegamento dei 4 host al packet switch STC 104.

Di seguito si riportano le velocità ottenute, $V(\text{PC1})$ e $V(\text{PC2})$:

$$V(\text{PC1}) = 9.46 \pm 0.01 \text{ Mbyte/s} \quad (1)$$

$$V(\text{PC2}) = 9.51 \pm 0.01 \text{ Mbyte/s} \quad (2)$$

A questo punto le due sorgenti, PC1 e PC2, sono state poste in trasmissione contemporaneamente per misurarne le velocità in presenza di contesa sullo switch. I valori rilevati, sempre con lo stesso volume di dati precedente, sono mostrati di seguito:

$$V_{cont}\ (PC1) = 6.79 \pm 0.07 \ \text{Mbyte/s}$$
$$V_{cont}\ (PC2) = 7.11 \pm 0.08 \ \text{Mbyte/s}$$

Il modello teorico prevede che nel caso in cui lo switch presenti 2 ingressi e 2 uscite in regime di comunicazione, la probabilità di trovare un'uscita occupata sarà data dalla relazione (3) del par. 3.3:

$$P(occupata) = 1 - (1 - \frac{1}{2})^2$$

utilizzando la (4) dello stesso par. 3.3 si ottiene una velocità teorica per la contesa

$$V'_{cont}\,(PCx) = P(occupata) \quad V(PCx) \qquad (3)$$

dove *V(PCx)* è una delle due velocità di trasmissione, senza contesa, precedentemente calcolate. Sostituendo nella (1) i valori citati si ottiene:

$$V'_{cont}\,(PC1) = 7.10 \pm 0.01 \quad \text{Mbyte/s}$$
$$V'_{cont}\,(PC2) = 7.13 \pm 0.01 \quad \text{Mbyte/s}$$

Confrontando questi ultimi valori teorici con quelli sperimentali, derivati dalla contesa nello switch, si ha uno scarto di circa il 4-5% che ci permette di convalidare ed accettare il modello teorico. Ovviamente per una maggiore attendibilità del modello sarebbero necessarie ulteriori prove con un numero maggiore di sorgenti e destinazioni.

Si è pensato di verificare ulteriormente il modello della contesa in una configurazione di due switch disposti in cascata, come illustrato in figura 5.4. In tale condizione, giacché le uscite del primo switch costituiscono al tempo stesso gli ingressi del secondo, usando la relazione (9) del par. 3.4, si ha:

$$P_2(occupata) = 1 - (1 - \frac{1}{2})^{P_1 \ 2}$$

dove P_1 è la probabilità già vista precedentemente.

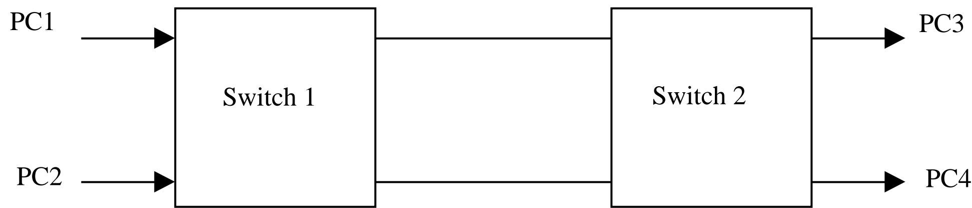


Figura 5.4: Switch disposti in cascata.

Sono state nuovamente misurate le velocità in assenza di contesa V(PC1) e V(PC2), ed i valori rilevati non si discostano da quelli ottenuti precedentemente. Il fatto che in presenza di un'altro switch le velocità non cambiano, indica anche che per pacchetti di dimensioni di 2048 byte il tempo di routing dello switch risulta trascurabile; si ricordi infatti che la latenza imposta dallo switch STC104 si ha solo per l'operazione di instradamento della testa del pacchetto, ed il corpo susseguente non subisce ritardi; la cosa ovviamente cambia qualora si considerino pacchetti di piccole dimensioni, in cui il tempo di routing ha il suo peso.

Nel caso in cui si ha possibilità di contesa negli switch le velocità misurate sono state:

$$V_{cont}\ (\text{PC1})^* = 5.89 \pm 0.03 \quad \text{Mbyte/s}$$
$$V_{cont}\ (\text{PC2})^* = 6.02 \pm 0.04 \quad \text{Mbyte/s}$$

mentre le velocità teoriche, inserendo nella (3) la probabilità P_2 sarebbero:

$$V'_{cont}\ (\text{PC1})^* = 6.11 \pm 0.01 \quad \text{Mbyte/s}$$
$$V'_{cont}\ (\text{PC2})^* = 6.13 \pm 0.01 \quad \text{Mbyte/s}$$

Ancora una volta gli scarti tra i due valori sono molto lievi, circa il 5%, confermando la validità e la buona approssimazione del modello teorico.

5.4 Prestazioni delle architetture banyan e Clos

Nei paragrafi successivi si farà riferimento ad una serie di misure effettuate sulle architetture viste nel paragrafo 3.4. Nell' ottica di uno studio puramente qualitativo sulle differenze di traffico tra diverse topologie, sono state utilizzate le interfacce nella duplice modalità sorgente/destinazione dei dati.

5.4.1 Il modello banyan con traffico random

Riportiamo in figura 5.5 il modello della *banyan network* di tipo 4   4 realizzata con switch di tipo 2   2. Il software realizzato, **txrx_rand_C104.c**, presenta una particolare procedura per la fase di ricezione.

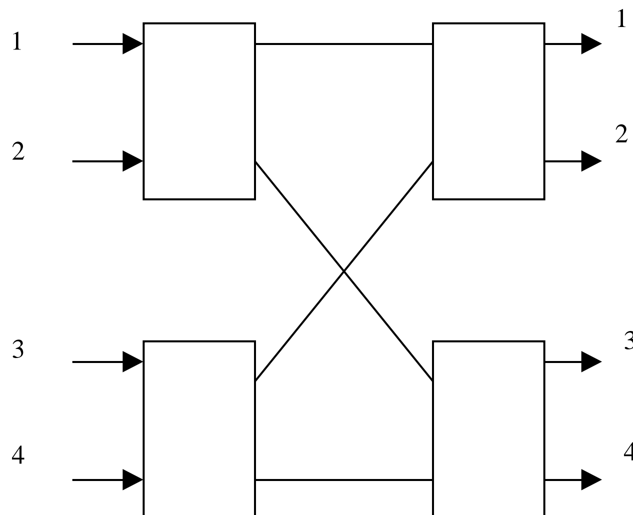


Figura 5.5: Topologia banyan a due stadi e di tipo 4   4 realizzata con 4 switch 2   2.

L' organizzazione del processo di comunicazione è il seguente:

- L' interfaccia, nella <u>fase sorgente</u>, trasmette un primo pacchetto con testa random affinché possa essere instradato casualmente, prima su una delle due uscite di uno switch del primo stadio e poi su una delle due uscite di uno switch del secondo stadio.

- Spedito il singolo pacchetto viene avviata la ricezione, fase destinazione, che dovrà contemplare, dato il tipo di traffico casuale, un minimo di 0 pacchetti (nessuna sorgente ha scelto questa destinazione) fino ad un massimo di 4 pacchetti (le sorgenti hanno scelto tutte la stessa destinazione).

I PC a disposizione, avendo caratteristiche diverse tra loro, hanno capacità diseguali di acquisire o trasmettere i dati (pacchetti) causando situazioni di stallo nel traffico; introducendo dei ritardi nel software degli host più veloci si diminuiscono le velocità di lavoro, evitando però la fase critica di blocco completo del sistema[2] (interfacce + switch). Si può osservare di seguito una parte del programma relativa alla fase di trasmissione (sorgente) e a quella di ricezione (destinazione).

```
/* Impostazione(random) testa del pacchetto */
Header[i] = (linknum1[i] + (linknum2[i] << 8));

my_outl(Header[i] , c101_or_pld|TX_PACKET_HEADER_LOWER_0);
my_outl(Header[i] /*per default*/,
        c101_or_pld|TX_PACKET_HEADER_UPPER_0);

/* Fase sorgente */
my_outldslink(DataTx, address, numbyte);
delay_old(xxxx);

while(Temp<5)
{
         /* Check per Rx */
   if((my_inw(c101_or_pld|ISR)&(PACK_RECEIVED))!=0)
   {
       /* Fase destinazione */
      delay_old(xxxx);
      DataRx = my_inldslink(address, numbyte);
      Temp = 0;
   }

  Temp++;
}
```

La realizzazione della *banyan network*, mediante 2 dei 3 switch a disposizione e con la possibilità di suddividerli ognuno in sotto-switch di tipo 2 2, ha richiesto la programmazione di un particolare file di configurazione. Il primo stadio della topologia è stato realizzato con una sezione di entrambi gli STC 104 e così pure il secondo stadio: in

[2] Sono stati fatti diversi tentativi per arrivare al compromesso: minimo ritardo ma completa operatività del sistema.

quest'ultimo è stato utilizzato il generatore random presente negli switch. Nella programmazione del file di configurazione si è dovuto tener conto, ovviamente, che i possibili percorsi ed instradamenti garantissero il traffico random.

Sfruttando nuovamente la scheda ISA-DSLink per la configurazione degli switch, si è passati alla misura vera e propria del traffico sulla rete. È stata valutata la velocità su ogni singolo host, mettendo i rimanenti in una situazione di traffico continuo e spedendo da quello sotto test, lo stesso volume di dati: misure con 15000 pacchetti ciascuno di 2048 byte ripetute per 10 volte.

I risultati ottenuti sono stati i seguenti:

Host	Velocità (Mbyte/s)
PC1	3.27 ± 0.04
PC2	3.27 ± 0.05
PC3	3.53 ± 0.03
PC4	2.83 ± 0.05

Si può osservare che l'introduzione dei ritardi porta il PC meno "brillante" in termini di prestazioni, PC3, ad avere la più alta velocità di lavoro, mentre il più veloce, PC4, ha subìto una pesante diminuzione delle proprie prestazioni; tali ritardi però, ripetiamo, garantiscono che il sistema non entri in condizione di stallo.

5.4.2 Il modello Clos con traffico random

La configurazione di questa rete è stata leggermente più laboriosa della *banyan*, in quanto introducendo altri 2 sotto-switch, occorreva garantire lo stesso instradamenti di tipo random. Il modello è stato realizzato ancora una volta utilizzando 2 soli switch STC 104 e suddividendoli in modo opportuno. Nella figura 5.6 è illustrata la *Rearrangeable Clos network* a tre stadi nel caso 4   4 con indicato lo switch a cui appartiene ogni sotto-switch; tra parentesi sono riportati gli *interval* assegnati ai link di ogni sotto-switch.

L'impostazione degli *interval* è legata al meccanismo scelto per distribuire il traffico. È stato sviluppato un file di configurazione per utilizzare pacchetti con 2 byte di testa, in cui il valore dell' ultimo byte era impostato in modo casuale in un intervallo tra

26 e 27. Inoltre, in accordo con quanto affermato nel paragrafo 3.4, si è fatto uso dello *adaptive grouping routing* al primo stadio della Clos; sia al primo che al secondo stadio della rete viene effettuata la cancellazione della testa, mentre al terzo stadio viene invece usato il generatore random dello switch. Vediamo come è stata resa possibile la scelta di cammini casuali.

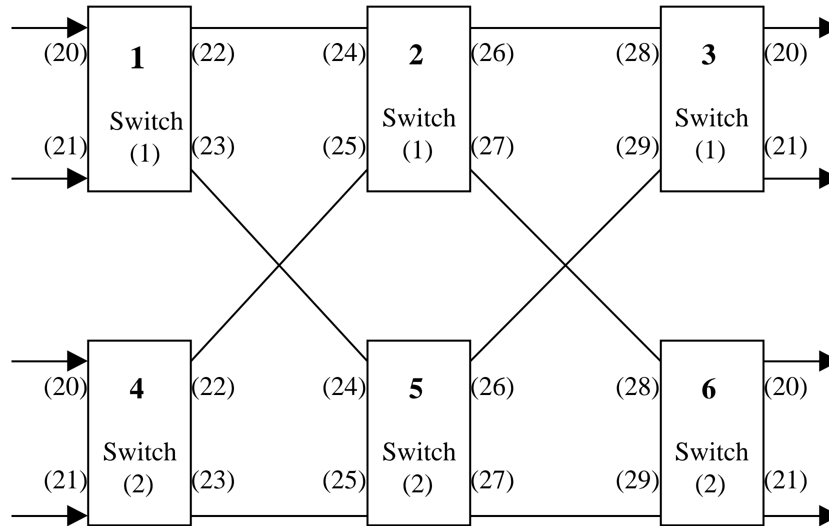


Figura 5.6: Rearrangeable Clos network a tre stadi di dimensione 4 4.

Una sorgente locata al primo stadio invia un pacchetto che, in base allo *adaptive grouping routing*, viene posto dal sotto-switch su uno dei due link che risulta libero: in questo stadio non si ha nessuna contesa ed il pacchetto, sottomesso a cancellazione del primo byte di testa, può andare al secondo stadio. Quello che era il secondo byte di testa con scelta random tra 26 e 27, diventa ora, al secondo stadio, il primo byte di testa; il pacchetto può così essere indirizzato al link associato allo *interval* 26 o al 27 e da qui ad uno dei sotto-switch del terzo stadio. Ricordando che al secondo stadio è attiva la cancellazione della testa, e ciò porterebbe il pacchetto ad esserne sprovvisto, viene aggiunta una testa random, con valore tra 20 e 21, dai sotto-switch del terzo stadio: lo instradamento è garantito ed il pacchetto può giungere ad una destinazione. Chiaramente con questo meccanismo è assicurata la possibilità che sia al secondo che al terzo stadio avvenga una contesa, proprio perché due diversi pacchetti possono richiedere la medesima uscita.

Per ottenere delle misure del traffico sono stati utilizzati gli stessi programmi della *banyan network*, con il medesimo volume di traffico.

I risultati ottenuti vengono di seguito indicati:

Host	Velocità (Mbyte/s)
PC1	3.3
PC2	3.3
PC3	3.5
PC4	2.8

Infine è stato studiato il modello della *Non-blocking Clos network*, che come visto nel paragrafo 3.4, permette di aumentare il numero dei possibili cammini all'interno degli stadi della rete, fornendo così un incremento delle velocità di traffico rispetto alla precedente *Rearrangeable Clos network*. Si riporta di seguito nella figura 5.7 lo schema della sua realizzazione mediante due soli STC104 partizionati in più switch di tipo 2 2 e 3 2.

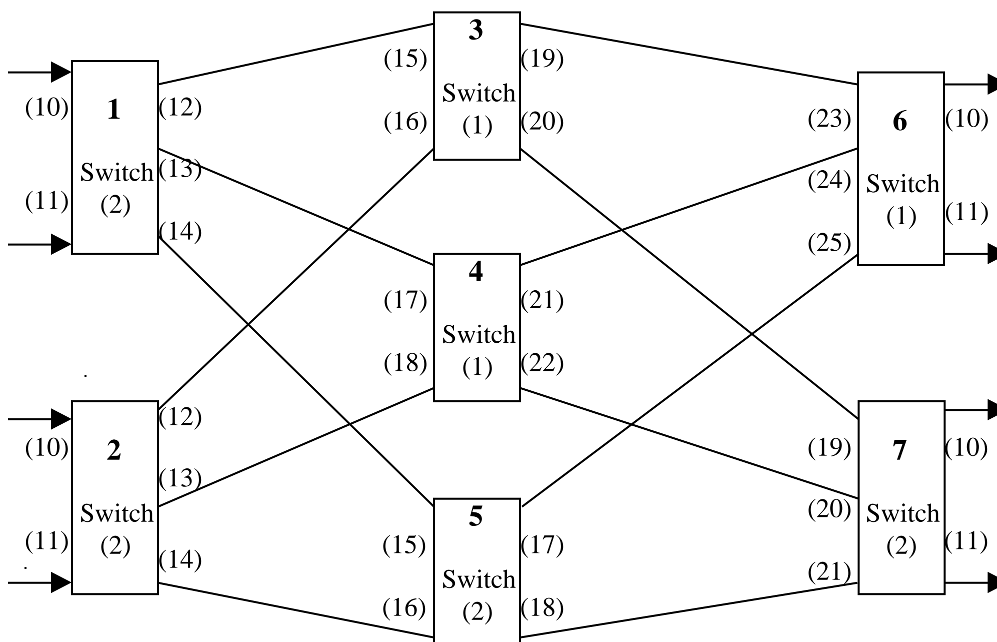


Figura 5.7: Non-blocking Clos network a tre stadi di dimensione 4 4.

Per questa topologia la fase di configurazione è stata sicuramente più laboriosa delle altre, proprio per garantire che tutti gli instradamenti fossero casuali. Anche in questo caso le velocità ottenute sui vari PC erano le medesime della precedente *Rearrangeable Clos network*. In questo caso, in base alla (10) del paragrafo 3.4, ci saremmo dovuti aspettare un leggero aumento del *throughput,* ma a causa dei già citati ritardi e del funzionamento bidirezionale delle interfacce tale minima variazione non si è riscontrata.

In sostanza i valori di velocità delle due reti di tipo *Clos* confrontati con quelli ottenuti precedentemente, *banyan network*, non mostrano grosse differenze; si può asserire quindi che quanto previsto nella trattazione teorica è stato confermato e che inoltre si è dimostrata indirettamente la validità e l'efficienza dello *adaptive grouping routing*.

5.5 Controllo da remoto delle comunicazioni DS-Link

Riprendendo un aspetto accennato in chiusura del Capitolo 4, si illustrerà ora il lavoro sviluppato per attuare una gestione da remoto delle sorgenti e destinazioni presenti in una semplice switching network, mediante un'interconnessione attraverso la rete Ethernet. Alcuni degli host su cui è alloggiata l'interfaccia PCI-DSLink, dispongono di una ulteriore scheda per la rete Ethernet (o più propriamente lo standard IEEE 802.3) a 10 Mbit/s con il necessario *device driver* per il sistema operativo Linux. È così possibile sfruttare l'interconnessione Ethernet per trasferire informazioni di controllo o di servizio, mentre i dati ad alta velocità possono viaggiare attraverso la rete di switch DS-Link. Nel presente lavoro per lo scambio delle informazioni attraverso la rete Ethernet è stato sfruttato il protocollo UDP (*User Datagram Protocol*), che è uno dei due di tipo end-to-end[3] forniti dall'architettura TCP/IP.

Questo approccio, di fornire cioè un' ulteriore meccanismo d' interconnessione, può essere utile per introdurre una possibile simulazione del controllo di architetture del 2° livello di ATLAS mediante un Supervisor[4].

[3] In una accezione più generale, senza riferimento a precedenti citazioni, da intendere end-to-end come comunicazione tra un host-destinazione ed un host-ricezione.
[4] In tal caso occorrerebbe parlare più propriamente delle architetture A e B, dal momento che nella architettura C le informazioni del supervisor dovrebbero viaggiare sulla stessa *Global Network*.

Nel prossimo paragrafo verrà brevemente descritta l'architettura TCP/IP, o *Internet Protocol Suite*, nonché il protocollo UDP stesso.

5.5.1 Internet Protocol Suite

Il progetto di ricerca che diede origine all'attuale architettura fu Arpanet [12], e fu finanziato dal Ministero della Difesa americano per creare una rete di comunicazione tra computer estremamente affidabile e costantemente attiva. L'architettura TCP/IP fu introdotta per integrare i diversi tipi di rete che venivano via via sviluppati, e permetterne quindi la completa interconnessione.

TCP/IP presenta dei livelli [40] [36] [12], o strati logici, che vengono riportati nella figura 5.8, ove il più alto è il livello Applicazione.

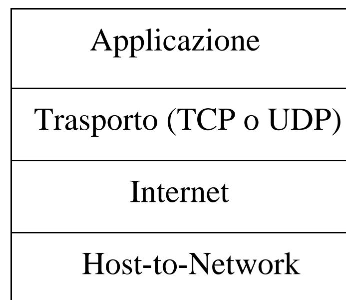


Figura 5.8: Livelli dello Internet Protocol Suite.

Livello Applicazione

In esso sono contemplati tutti i protocolli di alto livello per le applicazioni reali, quali **Telnet**, **FTP** (*File Transfer Protocol*) etc.

Livello Trasporto

Progettato per garantire passaggio di informazioni (pacchetti) end-to-end in cui sono definiti due protocolli: **TCP** (*Transmission Control Protocol*) e **UDP** di cui si è già accennato.

Livello Internet

Il livello che tiene assieme l'intera architettura permettendo il trasferimento di pacchetti tra host ubicati su stessa o diversa rete, garantendo nel contempo il corretto instradamento dell'informazione; il protocollo ufficiale per i pacchetti è **IP** (*Internet Protocol*). Su questo livello gravano incombenze di routing e di controllo della congestione.

Livello Host-to-network

Questo livello, il più basso, è relegato allo hardware: l'architettura prevede solamente che lo host abbia capacità di inviare pacchetti **IP** sulla rete.

Dopo questa premessa su TCP/IP analizziamo meglio le caratteristiche del protocollo UDP.

Esso è strutturato, al contrario del protocollo TCP, per fornire dei servizi non orientati alla connessione: ogni messaggio fornito dell'indirizzo completo della destinazione, è instradato nella rete in modo indipendente da altri messaggi. Poiché tra host-destinazione ed host-trasmittente non c'è connessione (come avviene nel sistema telefonico), può accadere che i messaggi spediti ad una destinazione arrivino in un ordine diverso da quello originario [28].

Però, rispetto a TCP, proprio per il motivo che UDP non è orientato alla connessione, il protocollo UDP ha il vantaggio di fornire una maggiore velocità di trasferimento dei dati [36]. Inoltre sviluppare del software con UDP piuttosto che con TCP comporta una sensibile riduzione di lavoro.

5.5.2 Struttura dell' interconnessione con Ethernet

Come organizzazione del lavoro, con uno sguardo rivolto al Supervisor del 2° livello, si è pensato di introdurre una configurazione di tipo Master-Slave, con un minimo di protocollo suppletivo, per lo scambio di informazioni: queste ultime sono ovviamente lontanissime dal simulare quelle probabilmente presenti in una futura architettura di 2° livello.

La configurazione completa del sistema, visibile in figura 5.9, è stata così strutturata:

- Due schede PCI-DSLink con funzione di destinazione e sorgente dei dati ad alta velocità (pensate ad esempio come interfacce per i *Data Driven* ed i *Global Processor* in un'architettura tipo A di LVL2, o *Local Processor* e *Global Processor* per l'architettura di tipo B), e funzione Slave nell'ambito della interconnessione Ethernet.

- Una scheda PCI-DSLink con duplice funzione: *Controlling process* per il controllo e configurazione di uno switch STC 104, e Master per la gestione e scambio di informazioni da remoto con le schede Slave (funzioni queste, tipiche del supervisor).

- Uno switch STC 104.

- Tre schede Ethernet alloggiate nei tre diversi host in cui trovano posto le interfacce PCI-DSLink.

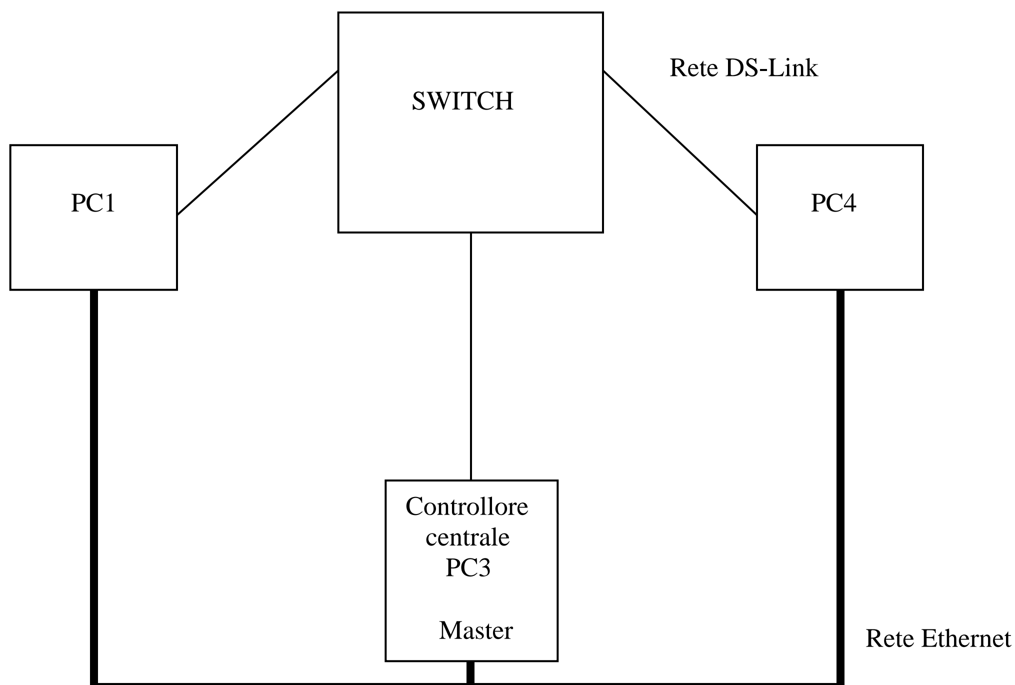


Figura 5.9: Configurazione del sistema.

Per la realizzazione dell'applicazione che permette la trasmissione e ricezione delle informazioni, sono state utilizzate le funzioni della *Berkeley Software Distribution* (BSD), la quale fornisce una tra le interfacce più comunemente usate per la programmazione di applicazioni (API): tali funzioni sono incorporate nel compilatore GCC. Il lavoro è stato condotto sfruttando i *socket*[5], secondo la sequenza di figura 5.10 che è poi la tipica procedura adottata utilizzando protocolli non orientati alla connessione [36].

Nella prima fase il Master attiva la chiamata **socket()** con cui viene creata una struttura dati all'interno del sistema operativo in uno spazio chiamato (*address family*) e con cui si avvia l'utilizzo del particolare protocollo (TCP opp. UDP); mediante la **bind()** viene assegnato un nome alla struttura precedentemente creata, mentre con **recvfrom()** e **sendto()** vengono rispettivamente ricevuti ed inoltrati pacchetti. Il meccanismo è analogo per lo Slave.

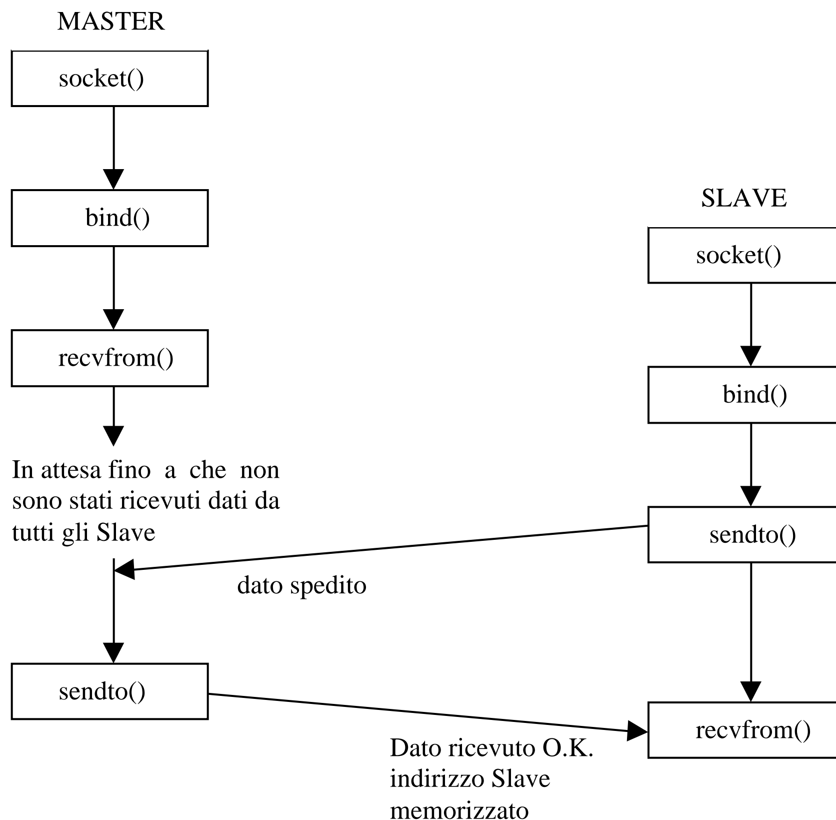


Figura 5.10: Sequenza delle chiamate per la connessione tra Master e Slave.

[5] Canali di carattere generale per comunicazioni tra processi locali o remoti.

La fase iniziale di ricezione del Master è stata strutturata per permettere a più di uno Slave di connettersi ad esso. Grazie all' uso di altre funzioni per i socket presenti nel GCC, quali ad esempio **gethostbyaddr**(), il Master in questa fase riconosce qual'è lo identificativo IP (unico) di ogni Slave, che viene automaticamente memorizzato in un array di puntatori. In tal modo il Master può colloquiare in ogni momento con un qualsiasi Slave: questo rientra tra l' altro nella politica di azione del supervisor, secondo cui esso può mandare informazioni o comandi ai vari processori senza che questi ne abbiano fatto richiesta.

Nelle fasi successive Il Master comunica allo Slave se dovrà lavorare in modalità destinazione o sorgente DS-Link: in quest'ultimo caso il Master inoltrerà allo Slave il volume del traffico da trasferire, le dimensioni dei pacchetti nonché la testa necessaria all' instradamento attraverso lo switch.

Quando sono terminate le procedure assegnate agli Slave, questi spediscono al Master la loro abilitazione ad una successiva operazione di flusso dei dati: in questo caso il Master deciderà se continuare o interrompere l' intero processo e quindi anche la interconnessione con la rete Ethernet, inviando un opportuno comando agli Slave.

Per distinguere i vari comandi che il Master deve inviare agli Slave è stato introdotto un semplice protocollo come illustrato in figura 5.11. Il primo byte identifica il comando mentre i successivi contengono l' informazione; nel caso in cui comando sia la interruzione della connessione Ethernet, questi ultimi byte sono vuoti.

Comando	N° Link	N° Pacchetti	Dim. Pacchetti

Figura 5.11: Struttura del pacchetto spedito dal Master.

Sono stati infine effettuati dei test di velocità sulle schede PCI-DSLink per appurare se l'introduzione di un'interconnessione Ethernet procurava qualche degradamento nelle prestazioni. Le misure sono stato prese tra PC1 e PC4 (dotati di schede di rete veloci per bus PCI) con la tecnica già vista del *round-trip-time*, ponendo PC4 come host per la ritrasmissione dei pacchetti. I dati sono stati fatti fluire attraverso un switch STC 104 configurato dallo host PC3, per un volume di traffico di 15000

pacchetti da 2048 byte. Si riportano di seguito i valori misurati nel caso di controllo da remoto o locale.

La velocità, con un controllo locale delle interfacce ed indicate con V_L, è la seguente:

$$V_L = 9.4 \pm 0.1 \quad \text{Mbyte/s}$$

Il valore della velocità V_R in presenza dell' interconnessione Ethernet è invece:

$$V_R = 9.43 \pm 0.07 \ \text{Mbyte/s}$$

Le due velocità sono senza dubbio identiche e quindi il controllo tramite i *socket* non perturba minimamente la velocità di lavoro delle interfacce PCI-DSLink. Questo conferma tra l'altro le scelte, in fase di sviluppo del software, di tenere indipendenti i due meccanismi: flusso dei dati e controllo da remoto.

Ci sarebbero da fare considerazioni circa la scelta del protocollo; con UDP non si ha la completa sicurezza che il messaggio una volta spedito giunga al destinatario (cosa che non avviene per TCP). Per ovviare a ciò si può realizzare un minimo protocollo con lo scambio di messaggi tra UDP-destinazione ed UDP-sorgente, aumentando però il traffico su Ethernet.

Tale aumento è in misura sicuramente inferiore [36] [40] rispetto a quello che si presenta con il protocollo TCP, il quale garantisce l'inoltro delle informazioni al destinatario, nonché l'ordine con il quale sono state spedite, grazie alla presenza dei riscontri (*ack*) provenienti dalla destinazione stessa; il surplus di informazione che viene introdotta appesantisce così il traffico in entrambe le direzioni di comunicazione.

Come ultimo "esperimento" sull'uso dei socket, è stata realizzata una versione del programma, **config_tcp.c**, per la configurazione e controllo da remoto dello switch in ambiente Linux. Con il software realizzato si è in grado, sfruttando in questo caso il protocollo TCP, di configurare ed individuare il numero degli switch che compongono la rete, su un host remoto. Il programma è strutturato nella modalità Client-Server, ove il Client risiede su una macchina remota ed il Server sulla macchina provvista dell'interfaccia PCI-DSLink necessaria alla configurazione degli switch.

CONCLUSIONI

Durante lo sviluppo di questo lavoro di tesi sono stati affrontati diversi aspetti correlati tra loro, che hanno permesso di acquisire teoricamente e poi con esperienza diretta numerose informazioni.

Come primo punto è stato affrontato e risolto il problema della configurazione dello switch, attraverso l'uso di un'interfaccia dalle elevate prestazioni che, in quanto realizzata per bus locale PCI, ha fornito un ottimo mezzo per apprendere e maturare esperienze su questo bus di moderna concezione. Inoltre il software realizzato per la configurazione è stato testato al CERN, sia nella versione con il controllo in locale che in quella Client-Server realizzata mediante l'uso del protocollo TCP; i risultati sono stati positivi e non sono stati riscontrati errori.

Il secondo aspetto è stato l'acquisizione e successiva verifica, di concetti importanti nella realizzazione di *device driver* in un sistema operativo UNIX compatibile (Linux), con cui è stato possibile sviluppare delle istruzioni che interagendo direttamente con il kernel, particolarmente nelle fasi di comunicazione, hanno permesso di ottenere prestazioni elevate.

Attraverso lo studio dello switch, visto come elemento di una rete d'interconnessione complessa come il trigger di 2° livello di ATLAS, si è arrivati a trattare prima teoricamente e poi praticamente una serie di modelli per reti di switch, probabili candidati alle architetture stesse dell'esperimento ATLAS. Realizzando le topologie *banyan* e *Clos*, e con una simulazione di traffico di tipo random, si è cercato di valutare quell'architettura che garantiva prestazioni migliori: in questo caso la ridotta disponibilità delle risorse a disposizione ha limitato l'indagine, ma risultati interessanti sono stati comunque rilevati.

Infine si è cercato di introdurre gli aspetti di comunicazione che dovranno legare il Supervisor ai processori locali e globali, attraverso la realizzazione di un sistema di tipo Master-Slave, sfruttando i *socket* ed il protocollo UDP per lo scambio di informazioni e la rete di STC104 per la trasmissione dati ad alta velocità.

Durante i diversi test di misura si è potuto constatare realmente l'alta configurabilità offerta dal packet switch STC104; tale prerogativa ha infatti permesso di realizzare delle topologie diverse tra loro. Ognuna di esse ha richiesto solamente l'adozione di un appropriato file di configurazione. È forse proprio nella generazione dei file di configurazione che si sono riscontrati degli aspetti negativi, perché nel caso di topologie di una certa complessità la costruzione di questi file (tramite il compilatore NDL) è risultata molto laboriosa e pesante.

In questo caso, come tra l'altro è stato già detto, sarebbe molto più comodo poter disporre di strumenti a più alto livello per la configurazione dello switch, fornendo al tempo stesso maggiore libertà all'utilizzatore.

L'impiego del linguaggio C per la programmazione si è rivelato molto soddisfacente, giacché l'insieme delle librerie necessarie per la realizzazione del *device driver* è scritto per il linguaggio C, così come in C sono scritti i codici sorgente del kernel stesso di Linux.

Inoltre l'aver adottato un sistema UNIX compatibile come Linux si è dimostrato essere una scelta molto valida, in quanto per l'utilizzo dei *socket* è disponibile una vasta e copiosa letteratura nonché una notevole quantità di librerie.

APPENDICE A

Esempi del software realizzato

Si riportano di seguito alcune parti, in codice sorgente, dei principali programmi realizzati.

Questo primo programma chiamato **PCIdsl.c,** si riferisce alla parte principale del *character device driver.*

```
#ifndef __KERNEL__
# define __KERNEL__
#endif

#include <linux/config.h>     /* per CONFIG_PCI */

#ifdef MODULE
# include <linux/module.h>
# include <linux/version.h>
#else
# define MOD_INC_USE_COUNT
# define MOD_DEC_USE_COUNT
#endif

#if (LINUX_VERSION_CODE < 0x020100)
# include <linux/mm.h>
# define copy_from_user memcpy_fromfs
# define copy_to_user   memcpy_tofs
# define ioremap  vremap
# define iounmap  vfree
#else
# include <asm/uaccess.h>
#endif

#include <linux/fs.h>
#include <linux/bios32.h>
#include <asm/io.h>
#include <linux/malloc.h>
#include <asm/pgtable.h>

#include "PCIdsl_int.h"
#include "amcc.h"
#include "dslink.h"

#define MAX_NODES 1
#define WAIT_FOR  0.05  /* 0.05 tempo di attesa */

#define rdtsc(time) ( {__asm  (".byte 0x0f; .byte 0x31" \
        : "=eax" (time)  ) ; } )
```