UNIVERSITÀ DEGLI STUDI DI PARMA
FACOLTÀ DI INGEGNERIA
CORSO DI LAUREA IN INGEGNERIA ELETTRONICA

INTEGRAZIONE DELLA VIRTUAL INTERFACE ARCHITECTURE (VIA) ALL'INTERNO DELLA LIBRERIA DI MESSAGE PASSING LAM/MPI

Relatore:
Chiar.mo Prof. Ing. GIANNI CONTE

Correlatori:
Dott. Ing. MASSIMO BERTOZZI
Dott. Ing. MONICA REGGIANI

Tesi di laurea di:
FRANCO BOSELLI

ANNO ACCADEMICO 1997-1998

Alla mia famiglia

Desidero rivolgere un sentito ringraziamento al Prof. Ing. Gianni Conte per avere gettato il seme da cui si è sviluppato un progetto interessante e promettente.

Vorrei ringraziare l'Ing. Monica Reggiani per la disponibilità che ha dimostrato in ogni momento del mio lavoro di tesi, per la capacità di affrontare con competenza i diversi problemi che sono sorti, per l'entusiasmo che quotidianamente ha distribuito nel laboratorio di ParMa^2 rendendo piacevole il duro lavoro.

Un particolare ringraziamento all'Ing. Massimo Bertozzi che mi ha condotto alla scoperta del sistema operativo Linux e per questo ha dovuto puntualmente rispondere alle mie numerose domande.

Un ringraziamento ai Dott. Emanuele e Alessandro Fava, storiche colonne del laboratorio di ParMa^2, per l'amichevole accoglienza che mi hanno riservato.

Ringrazio anche i colleghi del corso di Sistemi di Elaborazione Michele Belloli, Cristiano Fornasari, Monica Rossi e Federica Sacchi che hanno sperimentato insieme a me i pregi di M-VIA e le difficoltà legate alla sua programmazione.

Un ringraziamento va anche a tutti coloro che mi hanno accompagnato e sostenuto durante il mio cammino universitario, soprattutto nei momenti più impegnativi.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Architetture parallele

Una architettura parallela è costituita da un insieme di elementi di elaborazione che comunicano e collaborano per la risoluzione efficiente e veloce di un singolo problema dalla elevata complessità.

Il disegno di architetture parallele si occupa di cercare le soluzioni più convenienti affinchè i componenti di cui è costituito il calcolatore parallelo cooperino in modo da assicurare i risultati migliori, ottenibili attraverso sistemi equilibrati, privi quindi di colli di bottiglia.

Le tradizionali architetture parallele commerciali [1] [2] sono formate da un elevato numero di processori collegati tra loro da una struttura di interconnessione dedicata, progettata appositamente per assicurare comunicazioni a bassa latenza e ampia banda. Lo scopo di questi calcolatori è quello di ottenere le migliori prestazioni possibili in termini di capacità di calcolo complessiva: per raggiungere questo obiettivo, una volta si abbiano a disposizione unità di elaborazione tanto più potenti quanto consentito dalla tecnologia microelettronica, bisogna progettare con cura la rete di connessione tra i componenti della macchina parallela.

Attraverso opportune scelte progettuali [3] queste macchine parallele sono in grado di svolgere un singolo e complesso programma parallelo con le prestazioni più elevate permesse dalla tecnologia, trovando applicazione soprattutto nel campo del calcolo scientifico, ad esempio per la simulazione di fenomeni fisici molto complicati.

Tuttavia queste macchine non sono esenti da difetti. In primo luogo proprio perché la costruzione di una macchina parallela commerciale implica uno studio attento e un accurato progetto della architettura per la comunicazione tra i vari componenti [1], spesso intercorre un ritardo tra il momento in cui un calcolatore parallelo viene immesso sul mercato rispetto al momento in cui le tecnologie utilizzate sono disponibili: per esempio il Cray T3D, equipaggiato con microprocessori Alpha a 150 MHz, fu immesso sul mercato nel 1993, quando tale processore era già stato rimpiazzato da una omologa CPU con frequenza di clock maggiore. Questo aspetto che è stato indicato come ritardo di ingegnerizzazione [4] è accentuato dal ritmo di crescita della tecnologia microelettronica.

Non tutti i componenti presentano però lo stesso tasso di crescita: in particolare i microprocessori aumentano le loro prestazioni con un ritmo superiore di un ordine di grandezza rispetto all'incremento delle performance delle memorie e dei sottosistemi di Input/Output. Ciò complica ulteriormente il progetto dei sistemi paralleli in quanto un modello architetturale non sempre mantiene la sua validità al variare dei rapporti tra le prestazioni dei diversi sottosistemi.

Per ottenere comunicazioni efficienti tra i componenti di un sistema è quindi necessario riprogettare spesso le reti di interconnessione: per tale ragione il loro prezzo si mantiene elevato dal momento che i costi per lo studio e la realizzazione sono sfruttati solo per un numero limitato di macchine vendute.

Inoltre spesso l'utilizzo di queste macchine richiede una specifica conoscenza del sistema operativo e del software creati ad hoc per ogni tipo di architettura; ciò contribuisce a innalzare i costi di gestione e di manutenzione del sistema, rendendone più complicato e oneroso l'aggiornamento.

Lo svantaggio maggiore delle architetture parallele commerciali, però è forse costituito da altre considerazioni economiche: non solo il prezzo di acquisto è molto elevato e la manutenzione richiede ulteriori significativi e continui costi, ma l'incremento del rapporto prestazioni-prezzo per anno mostra che, mentre per i Personal Computer si attesta su valori prossimi al 100 per cento, le architetture parallele commerciali non sono in grado di beneficiare dei vantaggi delle economie di scala [4].

Un ulteriore problema è costituito dal numero di unità di elaborazione disponibili nei modelli più diffusi di macchine parallele commerciali: solitamente i nodi di calcolo sono numerosi, dal momento che le reti di interconnessione sono progettate per essere sfruttate pienamente con molti nodi, ma non è economicamente vantaggioso utilizzare questi tipi di connesione con un piccolo numero di unità di elaborazione. In questo modo si riduce ulteriormente il mercato di queste macchine parallele.

Da queste considerazioni è nata nel 1994 l'idea di costruire architetture parallele con un rapporto prestazioni-prezzo decisamente più elevato, con l'introduzione dei cluster di workstation [4].

1.2 Network di Workstation (NOW)

Si definisce cluster di workstation un insieme di nodi costituiti da Personal Computer o workstation, che cooperano in parallelo per la risoluzione di un problema. Un cluster di workstation o semplicemente cluster è quindi un sottoinsieme delle architetture parallele in quanto presenta una struttura formata da più unità di elaborazione con l'obiettivo di svolgere computazioni in parallelo per giungere alla soluzione di un unico problema. La singola unità di elaborazione dei cluster è costituita da un calcolatore a tutti gli effetti indipendente e autonomo. Per la sua natura flessibile un cluster si presta anche all'utilizzo per la gestione di software distribuito, come database: il famoso motore di ricerca web Altavista è supportato da una architettura hardware costituita da un cluster di workstation Digital Alpha.

Molti termini vengono utilizzati per descrivere l'architettura definita in precedenza, ognuno dei quali mette in risalto specifiche sfumature. Con il termine Network di workstation (NOW) [4], sinonimo di cluster di workstation, si enfatizza l'uso di nodi dotati di una elevata potenza di calcolo, superiore a quella comunemente messa a disposizione dai PC, riflettendo l'obsoleta distinzione tra workstation e PC; oggi grazie allo sviluppo dei PC tale distinzione è più sfumata, e si conferisce l'attributo NOW a sistemi i cui nodi non siano dedicati eclusivamente al calcolo parallelo, bensì siano solitamente usati come stazioni di lavoro per l'utenza comune e il solo tempo non occupato dagli utenti sia lasciato al calcolo parallelo. Con il termine di Pile of PC (PoP) si sottolinea invece l'uso di Personal Computer, acquistabili facilmente sul mercato dell'elettronica di consumo, come unità di elaborazione di sistemi dedicati al calcolo parallelo. Similmente il progetto denominato Beowulf [5] sviluppato ai laboratori della NASA a partire dall'estate del 1994, ha sempre posto la sua attenzione alla creazione di cluster di PC con l'uso di componenti standard, prodotti per il mercato consumer di modo che si potesse beneficiare della riduzione di prezzo dovuta alle economie di scala; oggi, grazie ai miglioramenti della tecnologia, Beowulf è sinonimo di Pile of

Componente	Incidenza sul costo totale
Processore	0.38
Memoria	0.25
Hard disk	0.12
Comunicazione	0.25

Tabella 1.1: Rapporti tra i costi per un nodo di un cluster di classe 1 (regola di Tron).

PC o di cluster di PC. D'ora in poi, quindi, si utilizzeranno i termini di cluster, network di workstation, cluster di Personal Computer per indicare una architettura parallela come definita all'inizio del paragrafo.

A prescindere dalla non univoca terminologia usata per indicare un cluster, una distinzione corrente divide i cluster in due categorie: alla prima, chiamata classe 1, appartengono i cluster i cui componenti siano tutti reperibili sul mercato di massa dei Personal Computer[1]; della seconda, detta classe 2, fanno parte quei cluster che pur mantenendo l'architettura formata da nodi di elaborazione indipendenti connessi tra loro per svolgere lavoro in parallelo, utilizzano anche componenti non standard o comunque non reperibili sul mercato di massa. Tale differenziazione tuttavia non è ben definita in quanto non è univocamente determinabile quali componenti appartengano al mercato di massa dei Personal Computer; alcuni definiscono per esempio come componenti off-the-shelf quelli che sono acquistabili almeno dal catalogo di tre diversi rivenditori di livello nazionale o mondiale. Altre possibili definizioni sono basate sulla quota minima di mercato che un componente deve avere per potere essere considerato parte del mercato di massa, oppure sul costo di un elemento del cluster che non deve eccedere determinate percentuali in rapporto ai costi degli altri elementi (si veda la tabella 1.1).

Qualunque sia il criterio di scelta adottato per discriminare i componenti off-the-shelf dagli altri, la distinzione tra cluster di classe 1 e classe 2 non vuole separare i cluster in due categorie per definire la scelta migliore: la decisione tra quali componenti utilizzare per un cluster deve tenere conto delle applicazioni che il cluster dovrà gestire, dalle prestazioni che è lecito attendersi e dal budget a disposizione. Generalmente infatti cluster di classe 2 offrono performance migliori rispetto alla classe 1 a spese di un maggior

[1]Questi componenti sono indicati con il termine *off-the-shelf* (COTS).

costo, della dipendenza dal o dai pochi produttori dei componenti custom o non standard che si traduce spesso in maggiori difficoltà nella configurazione e mantenimento del sistema e un supporto a livello di driver e di software minore.

Un cluster di PC si compone di tre elementi: un *nodo* costituito da un calcolatore separato e autonomo con capacità di elaborazione, di memorizzazione e di I/O; l'hardware di interconnessione tra i nodi capace di costituire un canale fisico per la comunicazione tra i nodi separati, chiamato genericamente *rete*; il *protocollo di comunicazione* che definisce le modalità secondo cui avvengono gli scambi di dati. Ciascuno di questi elementi deve cercare di massimizzare una specifica grandezza: la potenza di calcolo per il nodo, la banda passante per la rete di interconnessione e la latenza per il protocollo di comunicazione. Un cluster è infatti caratterizzato anche dalla distanza tra i nodi che deve essere limitata, di modo che si possano usare reti di interconnessione con ampia banda e protocolli con un numero minimo di livelli software che assicurino bassa latenza: spesso piuttosto che di Local Area Network (LAN) si parla in questo caso di System Area Network (SAN), per sottolineare le caratteristiche di rete dedicata esclusivamente alla connessione dei nodi di un cluster senza ulteriore traffico.

È utile a questo punto introdurre e definire alcuni parametri per caratterizzare in modo univoco le grandezze relative alle comunicazioni all'interno di un cluster. Si utilizza a tale scopo il modello LogP [6], che definisce quattro parametri:

Latenza L: ritardo che intercorre tra la trasmissione e la ricezione di un messaggio di dimensioni tendenti a zero.

Overhead o: tempo impiegato dal processore per svolgere tutte le operazioni richieste per la trasmissione o ricezione di un pacchetto di dati; durante questo tempo il processore non può effettuare alcun altro compito.

Gap g: minimo intervallo di tempo tra due consecutive trasmissioni o ricezioni di un byte; il suo reciproco è la banda passante.

Numero di processori P: il numero di processori presenti nel cluster.

Secondo tale modello il *ritardo di trasmissione* per un messaggio di N byte risulta:

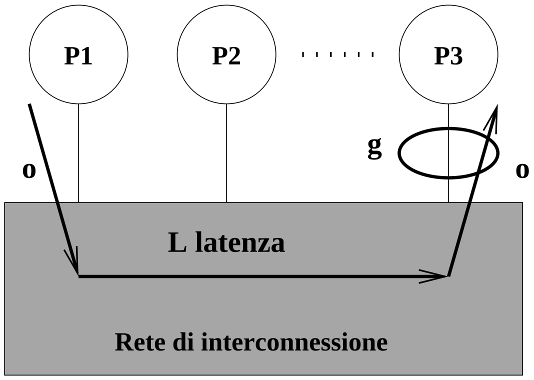

Figura 1.1: Modello LogP.

$$T = L + 2 * o + g * N$$

Uno dei pregi di questo modello è la distinzione tra la latenza dovuta al ritardo introdotto dalla trasmissione dei dati sulla rete, causata dalla scheda di rete, dalla interconessione e dal protocollo di rete, e dall'overhead causato dalle operazioni che il processore deve svolgere in conseguenza alle azioni richieste dal protocollo di comunicazione. Purtroppo la misura separata della latenza e dell'overhead risulta piuttosto complicata, per cui solitamente, come avverrà anche nei successivi capitoli, si esprime e determina solo un tipo di tempo, che indicheremo impropriamente latenza, comprensivo della latenza come definita dal modello LogP e di due volte l'overhead.

L'obiettivo di un cluster può essere riassunto nella capacità di svolgere computazioni in parallelo ottenendo un rapporto prezzo-prestazioni migliore rispetto alle tradizionali architetture utilizzate per il calcolo parallelo. I cluster di PC cercano di intervenire su entrambi i fattori sfruttando il continuo aumento della potenza di calcolo dei microprocessori destinati al mercato dei Personal Computer, offerta a costi decisamente ragionevoli. Per quanto riguarda la potenza computazionale, il principale vantaggio dei cluster di PC risiede proprio nello straordinario sviluppo della tecnologia VLSI che consente, secondo quanto previsto dalla legge di Moore e sostanzialmente verificato, un raddoppio del numero di componenti ospitati su di un singolo chip ogni 18 mesi, a cui corrisponde quasi un raddoppio della velocità di clock dei microprocessori. L'abbattimento dei costi è consentito invece dall'enorme diffusione dei Personal Computer che ha portato la loro quota di mercato a coprire la quasi totalità di tutti calcolatori (server, workstation, mainframe e supercomputer) venduti; la diffusione dei PC a sua volta contribuisce enormemente all'abbassamento dei prezzi, dal momento che, come ha sintetizzato Gordon Bell, il raddoppio del volume di produzione dei PC riduce il loro costo unitario del 90 per cento, a causa degli enormi investimenti in ricerca tecnologica che le industrie microelettroniche devono affrontare, innescando in questo modo un circolo virtuoso del quale gli utenti possono beneficiare. I vantaggi delle economie di scala si traducono nel costante miglioramento non solo delle CPU, ma anche degli altri componenti dei Personal Computer.

Il frenetico incremento delle velocità delle CPU per PC, permette inoltre di acquistare in ogni momento i componenti più evoluti tecnologicamente: in altre parole, al contrario di quanto avviene per i supercomputer che soffrono di un ritardo tra il momento in cui la macchina è immessa sul mercato rispetto ai componenti più evoluti dovuto al tempo di ingegnerizzazione, con i cluster di PC si è in grado di seguire il mercato ottenendo componenti con le massime prestazioni possibili.

Un ulteriore vantaggio nel costruire cluster di PC con componenti off-theshelf è l'indipendenza da un particolare produttore o rivenditore di hardware o software, di modo che da un lato si possa beneficiare della concorrenza, dall'altro si possa fruire di prodotti standardizzati testati da un gran numero di utenti e, per quanto riguarda i componenti hardware, dotati di ampio supporto software.

Per quanto riguarda il software una scelta economicamente vantaggiosa è quella di ricorrere a sitemi operativi e applicativi disponibili gratuitamente; tale codice spesso non è affatto inferiore per qualità a quello venduto dalle varie software house; consente inoltre di disporre dei sorgenti, rendendo possibili eventuali modifiche, e dell'aiuto di un vasto numero di utenti e sviluppatori che solitamente sono pronti a collaborare e a condividere esperienze e risultati.

Riassumendo le caratteristiche più significative dei cluster di PC più diffusi sono:

- rapporto prezzo-prestazioni inferiore alle architetture parallele commerciali,
- l'uso di componenti ampiamente disponibili sul mercato,
- disponibilità di processori dedicati al calcolo parallelo,

- interconnesione tra i nodi costituita da una rete privata (SAN),
- l'uso di un sistema operativo e applicativi gratuiti,
- la capacità di aggiornare facilmente l'architettura seguendo gli sviluppi tecnologici,
- i limitati costi di manutenzione hardware e software.

1.3 Sommario

Questa tesi si occupa dello sviluppo di un cluster di PC con l'obiettivo di massimizzarne il rapporto prestazioni-prezzo, iniziato con il progetto dell'hardware per costruire un singolo nodo e per l'interconessione fino alla scelta della libreria per gestire software parallelo e alla sua modifica per migliorarne le prestazioni, percorrendo per livelli successivi le tappe per ottimizzare il cluster. L'attenzione maggiore è stata tuttavia rivolta al protocollo di comunicazione e alla sua integrazione negli strumenti che presentano una maggiore efficacia per l'uso in cluster, cioé la libreria di Message Passing che consente la scrittura e l'esecuzione di codice parallelo.

Il **secondo capitolo** inizia la trattazione relativa allo sviluppo di un cluster dal livello più basso, con lo studio dei componenti hardware che costituiscono il nodo di calcolo: prima vengono elencati gli elementi di un nodo, quindi sono analizzate le possibili tipologie di interconnessione. Infine viene descritto il cluster costruito presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Parma, chiamato **ParMa^2**.

Il **terzo capitolo** costituisce una approfondita analisi della **Virtual Interface Architecture**, l'interfaccia di comunicazione proposta come standard da Intel, Microsoft e Compaq, tre importanti industrie a livello mondiale. Tale studio teorico è basato sui diversi documenti di specifica rilasciati al pubblico e copre tutti gli aspetti definiti dalle specifiche. Vengono quindi descritti i componenti dell'architettura, le loro funzioni e relazioni tra loro, spiegando sinteticamente e con molta precisione lo svolgimento di un generico trasferimento di dati tra processi, al fine di comprendere le differenze con un modello tradizionale di comunicazione interprocesso, come possono essere le socket BSD. In seguito si approfondiscono le varie fasi necessarie per lo scambio di dati con VIA, a partire dalla creazione delle strutture necessarie al funzionamento dell'architettura, per giungere ai modelli di connessione tra processi. Sono poi presentati i due modelli di comunicazione supportati da VIA, il modello Send/Receive e il modello basato su operazioni di lettura e scrittura remota (Remote Direct Memory Access o RDMA). Infine vengono descritte alcune funzionalità accessorie di VIA.

Il **quarto capitolo** si occupa di valutare quali siano i possibili benefici sulle prestazioni ottenibili con l'uso della Virtual Interface Architecture: si è sperimentata e testata una particolare realizzazione della Virtual Interface Architecture per il sistema operativo Linux, chiamata **M-VIA**, sul sottocluster B di ParMa^2 in precedenza costruito. Di M-VIA sono rilevate e studiate le caratteristiche e la sua integrazione nel sistema operativo. Si procede con l'analisi delle prestazioni di M-VIA in relazione ai tempi di latenza e ai ritardi di trasmissione di pacchetti di diverse dimensioni ottenuti sul cluster paragonandoli a quelli forniti dai più diffusi protocolli TCP e UDP. Dalla valutazione di tutti i risultati sperimentali si deducono quindi i pregi e i difetti di M-VIA.

Proseguendo idealmente con la costruzione del cluster si sale ulteriormente di livello descrivendo l'interfaccia a disposizione dell'utente per sfruttare i singoli nodi del cluster come una architettura parallela reale, in grado quindi di risolvere un unico complesso problema attraverso la cooperazione di più unità di elaborazione; tra le possibili opportunità si è scelto il Message Passing, di cui il **quinto capitolo** descrive caratteristiche fondamentali e differenze con altri modelli di utilizzo dei nodi di un cluster. Viene quindi introdotto il **Message Passing Interface (MPI)**, uno standard molto diffuso per il Message Passing. Infine vengono valutate con risultati sperimentali due diverse realizzazioni di MPI, motivando la scelta della libreria LAM MPI.

Dal momento che l'obiettivo primario di questa tesi è consentire a una libreria di Message Passing di utilizzare M-VIA per trasmettere i messaggi, il **sesto capitolo** esegue una operazione di reverse engineering analizzando il codice della libreria **LAM MPI** al fine di comprenderne il funzionamento e valutare la possibilità della sostituzione dei meccanismi di trasporto dei messaggi.

Il **settimo capitolo** si occupa interamente del progetto del nuovo livello destinato al trasporto dei messaggi. Prima viene affrontato il problema di creare un protocollo di comunicazione idoneo per MPI avendo a disposizione le sole primitive fornite da M-VIA. Quindi si introduce nel protocollo un ulteriore livello di controllo di flusso per evitare l'esaurimento delle risorse di comunicazione. Poi si definisce il funzionamento vero e proprio dell'interfaccia di trasporto per LAM con l'uso di M-VIA. Infine si confrontano le prestazioni delle funzioni MPI che sfruttano M-VIA con quelle ottenibili dagli attuali meccanismi di comunicazione realizzati da LAM MPI.

L'**ottavo capitolo** trae le conclusioni riguardo tutto il lavoro svolto, quindi principalmente in relazione all'uso di M-VIA e di VIA come protocollo di comunicazione a bassa latenza in un cluster; ma anche riguardo alle modalità di progetto di un cluster di PC, e ai suoi componenti. Infine si valutano i risultati ottenuti dalle primitive di comunicazione MPI della versione di LAM MPI riscritta integrando al suo interno M-VIA, tracciando le linee per il futuro sviluppo di questo progetto.

Capitolo 2

Struttura hardware di un cluster di PC

2.1 Il nodo

In un cluster di PC il singolo nodo è costituito da un completo e autonomo Personal Computer con capacità di calcolo, possibilità di svolgimento di operazioni di I/O e interfaccia di connessione verso qualche tipologia di rete; esso quindi non presenta peculiarità specifiche che lo differenziano dai Personal Computer che vengono utilizzati dall'utenza domestica o Small Office Home Office (SOHO), per cui l'architettura di base di un PC usato come nodo di un cluster non risulta dissimile a quella dei calcolatori distribuiti dalle comuni reti commerciali: le differenze risiedono in alcuni componenti che determinano la tipologia di funzione che il PC meglio si adatta a svolgere, come ad esempio server o stazione di lavoro. Si descrive di seguito molto succintamente l'architettura dei PC basati sui microprocessori Intel x86, usati come nodi nella maggior parte dei cluster, senza entrare nelle problematiche generali relative a ogni componente, ma soffermandosi su quelli di interesse per un nodo di un cluster.

I componenti minimi contenuti in ogni Personal Computer sono:

Il case: è il contenitore passivo di gran parte dei componenti di un PC, e al suo interno devono essere presenti le circuiterie fondamentali, mentre i dispositivi periferici, insieme ai device di interfaccia con l'utente quali tastiere e monitor, si trovano all'esterno del case.

L'alimentatore: ha il compito di convertire il segnale elettrico proveniente dalla rete di distribuzione dell'energia in un segnale di tensione opportuna da distribuire a tutti i componenti elettronici.

La scheda madre: la scheda madre, motherboard o mainboard, è la scheda destinata a contenere i principali componenti di un PC quali la CPU, la memoria, porte di I/O, il BIOS e i connettori o slots in cui andranno inserite le schede di espansione.

La funzione più importante che deve svolgere è quella di permettere una comunicazione efficiente tra tutti i componenti che alloggiano sulla motherboard stessa: a questo scopo sono dedicati vari tipi di bus, cioè canali che permettono lo scambio di segnali elettrici con opportune codifiche.

Il processore: il processore o Central Processing Unit è il sofisticato circuito elettronico che costituisce la parte intelligente del PC, in quanto si occupa di acquisire e leggere le istruzioni che gli vengono fornite, di raccogliere i dati eventualmente necessari ed eseguire le operazioni richieste.

Le CPU più utilizzate nei nodi dei cluster sono quelle della famiglia Intel x86, proprio per le motivazioni presentate nel primo capitolo: l'evoluzione tecnologica permessa dai grandi volumi di vendita di questi microprocessori produce CPU sempre più potenti a prezzi contenuti, consentendo anche un costante aggiornamento dei nodi del cluster.

La memoria di sistema: è costituita da circuiti elettronici con il compito di immagazzinare dati in maniera casuale (RAM Random Access Memory), cioè consentendo l'accesso ad un singolo dato specificandone l'indirizzo. E' realizzata con memorie a semiconduttore che, grazie all'alta integrabilità, raggiungono tempi di accesso dell'ordine delle decine di nanosecondi; sono però memorie volatili, cioè perdono il loro contenuto all'atto dello spegnimento dell'alimentazione.

Tutti i tipi di memoria di un calcolatore sono organizzati gerarchicamente: al livello più alto stanno le memorie più veloci, di capacità più limitata e di costo maggiore; la CPU nel momento in cui ha bisogno di un dato interroga per prima la memoria di primo livello, la quale nel caso non abbia disponibile il dato la richiede alla memoria di secondo livello e così fino all'ultimo livello.

Il più recente sviluppo della tecnologia microelettronica ha lanciato sul mercato di massa memorie di tipo DRAM sincrona con il bus di sistema, in grado cioé di eseguire un trasferimento di dati a ogni ciclo del segnale di clock del bus; i nodi più recenti utilizzano questo tipo di memorie.

La memoria di massa: è costituita da hard disk che hanno il compito di memorizzare in maniera permanente dati. Essi costituiscono il gradino più basso della gerarchia delle memorie, essendo le più lente con tempi di accesso dell'ordine della decina di millisecondi, e le più capienti, fino a decine di gigabyte. Le ragioni di queste differenze risiedono anche nella tecnologia utilizzata: mentre le RAM sono memorie a semiconduttore, veloci ma costose, gli hard disk sono dispositivi basati su materiali ferrromagnetici, più economici. E' importante osservare che mentre le memorie a semiconduttore nel corso degli ultimi anni si sono evolute enormemente, ciò non è accaduto per gli hard disk, che hanno incrementato molto la capacità ma relativamente poco i tempi di accesso e le velocità di rotazione dei dischi.

Per quanto riguarda i nodi di un cluster spesso essi, o alcuni di essi sono privi di dispositivi di memorizzazione permanente dei dati[1]: il caricamento del sistema operativo così come delle applicazioni avviene di solito via Network File System (NFS).

Il sottosistema video: è composto da una serie di circuiti alloggiati su schede che possono anche includere alcuni megabyte di memoria RAM, dei processori di accelerazione o dedicati alle immagini in tre dimensioni; si occupa di inviare al monitor tutte le informazioni che devono essere viste dall'utente. I nodi di un cluster solitamente sono sprovvisti di schede grafiche e monitor; solo alcuni usati come front end dispongono di questi dispositivi.

Nel seguito si accennano alcune considerazioni sulla scheda madre, componente che riveste una particolare importanza per i nodi di un cluster.

[1] Si dicono diskless.

2.1.1 La motherboard

La motherboard è il supporto che in un calcolatore ospita tutti gli altri componenti e permette le comunicazioni tra di loro, quindi può essere considerata la parte più importante dell'intero calcolatore. Infatti le funzioni che deve svolgere sono di primaria importanza:

Organizzazione: direttamente attraverso una porta o uno slot, indirettamente attraverso un controller, tutti i componenti sono connessi alla motherboard, quindi il modo con cui la scheda madre è disegnata, gli slots e le porte con cui è equipaggiata influiscono sull'organizzazione dell'intero PC e sulle funzionalità che esso può offrire.

Controllo: la motherboard attraverso il BIOS ed il chipset effettua un controllo su qualunque tipo di dato transiti all'interno del calcolatore.

Comunicazione: è una delle funzioni più importanti, anche perché il collo di bottiglia tra i vari componenti spesso risiede nella velocità con cui i sottosistemi comunicano tra loro; è importante quindi organizzare efficientemente le comunicazioni suddividendo i bus in tipi e stabilendo tra di essi una gerarchia o utilizzando linee dedicate; la motherboard e il suo chipset si devono occupare di gestire tutte le problematiche relative alla comunicazione.

Supporto al processore: ogni scheda madre ospita un particolare connettore detto zoccolo o socket che costituisce l'interfaccia tra la scheda madre ed il processore. Sfortunatamente ogni famiglia di processori presenta una piedinatura diversa e pertanto necessita di un apposito socket; la scelta di una scheda madre vincola quindi, seppur entro certi limiti, la scelta del processore.

Supporto alle periferiche: la scheda madre determina in gran parte il tipo di periferica che potrà essere collegato, in quanto le periferiche stesse richiedono una appropriata interfaccia con la motherboard; esistono dei bus e degli slot di diversi standard per ospitare le varie periferiche.

Prestazioni: la motherboard influenza le prestazioni del sistema per due ragioni: la prima è che essa discrimina i tipi di processore, la quantità di memoria massima installabili, il tipo e la velocità dei bus e questi parametri determinano le prestazioni dell'intero sistema. In secondo luogo la circuiteria stessa della motherboard e in particolare il chipset, in quanto gestori delle comunicazioni, influenzano le performance.

Espandibilità: la capacità di un calcolatore di espandersi relativamente a processori diversi o stessi processori di frequenze diverse o con tensioni di funzionamento diverse, a quantità di memoria diverse o tipi diversi di memoria o tagli diversi di memoria, a tipi e numero di periferiche dipendono quasi esclusivamente dalla scheda madre.

Per tutti questi motivi i nodi per cluster sono costruiti con le migliori schede madri disponibili in commercio, prodotte da costruttori noti e affidabili.

Per ottenere prestazioni migliori, alcuni sistemi operativi, come Windows NT, alcune versioni di UNIX, Linux, possono operare con più di una CPU per macchina, che lavorano in parallelo dividendosi il carico delle istruzioni. Si possono realizzare sistemi con 2 , 4 o più CPU; tuttavia per poter sfruttare efficacemente un sistema multiprocessore è necessario che il sistema operativo sia predisposto al supporto per tale caratteristica [7]; infatti l'impiego di una coppia o più di processori non migliora minimamente le prestazioni del PC se non viene utilizzato un sistema operativo adeguato.

Alcuni chipset forniscono un supporto nativo per il multiprocessing, ad esempio la famiglia di chipset Intel 440 supporta il Symmetric Multi Processing (SMP). E' inoltre possibile installare una sola CPU su una mainboard dual processor.

Sistemi multiprocessore sono di ampio utilizzo nei cluster di PC poichè la caratteristica più importante richiesta a un nodo è la potenza di calcolo: le CPU addizionali in un nodo aumentano la capacità di elaborazione se adeguatamente supportate dal sistema operativo, senza provocare un analogo incremento nel costo complessivo del nodo, cosicchè il costo per istruzioni al secondo diminuisce.

L'elemento funzionale più importante di una scheda madre è il chipset: esso è costituito da uno o solitamente più circuiti integrati progettati e costruiti per gestire le comunicazioni tra le varie parti di un PC (CPU, memorie, bus, schede aggiuntive). Inizialmente vi era un singolo chip per ogni funzione, ad esempio uno per la cache, uno per la RAM, uno per il DMA, poi si passò a 4 chip ed infine oggi un chipset consiste di soli 2 chip: un System Controller si occupa dei bus di sistema e PCI; questo chip in particolare si occupa delle comunicazioni tra CPU e memoria, e ha una notevole influenza sulle prestazioni in quanto è deputato alla gestione della gerarchia della memoria. L'altro chip è il PCI-ISA bridge che gestisce lo scambio di dati tra la RAM e i dispositivi connessi al bus ISA e alle porte seriali e parallele; fornisce inoltre due canali EIDE ognuno dei quali può gestire due periferiche, quindi è responsabile dello scambio di dati tra gli hard disk e la RAM; svolge inoltre la funzione di bridge grazie alla quale l'host bus non percepisce le differenze tra i vari tipi bus o porte.

Una significativa novità che ha contribuito ad aumentare le performace dei PC è stata introdotta con il chipset 440BX prodotto da Intel a partire dall'aprile del 1998: il bus di sistema ha innalzato la sua frequenza di funzionamento da 66 a 100 MHz; ciò ha notevolmente velocizzato gli accessi del processore alla memoria, tanto che per un nodo di elevate prestazioni questa caratteristica è indispensabile.

2.2 La rete di interconnessione

Fin dalla nascita delle architetture di tipo cluster il collo di bottiglia è costituito dalla interconnessione tra i nodi, soprattutto in relazione alla differenza tra la banda passante disponibile per la comunicazione tra i dispositivi interni al nodo, i quali colloquiano per mezzo dei bus disposti sulla scheda madre, e la banda per le comunicazioni internodo. Per tale motivo la scelta di Ethernet e di Fast Ethernet non sembra appropriata, dal momento che la massima velocità di trasmissione teorica è di 12.5 MegaByte al secondo contro i 132 MegaByte al secondo del bus PCI (32 bit e 33 MHz). La ragione della sua adozione per i cluster risiede soprattutto nella diffusione di questo tipo di protocollo nelle LAN[2]: questo implica prezzi di acquisto bassi, conoscenza diffusa dei problemi e delle soluzioni, supporto a livello di driver nei sistemi operativi. Queste ragioni, unitamente alla possibilità di realizzare diverse topologie, con limitati vincoli sulla lunghezza dei segmenti, hanno indotto molti cluster a adottare Ethernet e poi Fast Ethernet come tecnologia di connessione tra i nodi.

Si ricordano in breve le caratteristiche principali del protocollo Ethernet [8]:

[2]Stime recenti indicano nel 80 per cento la quota di mercato di Ethernet sul totale delle LAN.

- protocollo che specifica la parte Medium Access Control (MAC) del livello OSI data link e il livello fisico;
- meccanismo di controllo di accesso al canale fisico di tipo CSMA/CD;
- in caso di collisione lo slot in cui ritentare la trasmissione viene calcolato con un algoritmo chiamato Truncated Binary Exponential Back-off, che determina l'attesa prima della ritrasmissione in base alla realizzazione di una variabile aleatoria;
- topologia a bus;
- numero massimo di stazioni per ogni rete 1024;
- indirizzi a 48 bit;
- lunghezza massima di un pacchetto 1514 byte.

Per quanto riguarda l'interconnessione dei nodi di un cluster che utilizzi Ethernet o Fast Ethernet come protocollo di comunicazione, la topologia più semplice è costituita da un unico cavo che realizza il bus che connette tutti i nodi: la banda disponibile è condivisa tra tutti i nodi.

Operano allo stesso modo anche i dispositivi detti *hub* o *concentratori*: dispongono di una porta per ogni stazione che si vuole collegare; nel momento in cui una stazione trasmette un pacchetto questo è ricevuto dall'hub che lo invia su tutte le porte collegate eccetto quella di provenienza del pacchetto. Un hub quindi non suddivide le stazioni connesse in diversi collision domain: la topologia rimane quella di un bus condiviso, con il vantaggio di potere creare il bus mediante collegamenti punto-punto da una stazione all'hub e di fungere da ripetitore di pacchetti.

Una topologia caratterizzata da virtuali collegamenti punto-punto con protocollo Ethernet può essere realizzata con l'ausilio di uno *switch* [9].

Lo switch esegue delle funzioni più complesse rispetto a quelle dell'hub: esso infatti riceve un pacchetto, ne legge l'indirizzo MAC, e lo trasmette solo alla porta di destinazione, evitando di occupare porte non interessate alla comunicazione; lo switch quindi spezza il collision domain in tanti segmenti quante sono le sue porte. La banda che teoricamente uno switch mette a disposizione è la somma della banda di tutte le porte, dal momento che permette la comunicazione simultanea tra più nodi.

Gli switch possono essere distinti in diversi tipi in base alle funzionalità che essi offrono e al modo con cui le svolgono: esistono switch *statici* e *dinamici*, *switch di segmento* e *switching hub*.

Gli switch statici fanno semplicemente in modo che alcune porte vengano raggruppate e definite come appartenenti a un unico segmento. Tutto il traffico provienente da ogni porta del gruppo viene riportato a ogni altra appartenente al medesimo gruppo e a nessun altra. Uno switch statico quindi ha forti analogie con un hub. Lo switch dinamico invece apprende gli indirizzi delle stazioni collegate alle sue porte ogni volta che una di queste venga connessa. In questo modo ogni pacchetto in arrivo viene inviato solo alla porta corretta eliminando lo spreco di banda; se poi una stazione viene spostata da un segmento a un altro, lo switch riconfigura immediatamente e automaticamente la sua tabella degli indirizzi.

La differenza invece fra switch di segmento e switching hub consiste nel fatto che uno switch di segmento è in grado di gestire un segmento intero di rete per ogni porta, mentre uno switching hub può gestire solo un'unica stazione per porta.

Gli switching hub sono sovradimensionati per la maggior parte di reti con un nodo per ogni porta, tuttavia per la connesione di nodi di un cluster è consigliabile disporre uno switching hub dinamico, per poter offrire a ogni nodo una connessione con ciascuno degli altri disponendo della intera banda.

Una delle caratteristiche di maggiore interesse per uno switch è la modalità con cui i pacchetti in ingresso vengono immagazzinanti in una zona di memoria dello switch stesso prima di essere inviati: si distinguono switch *cut through* e *store and forward*. Uno switch cut through consente connessioni a latenza minima fra i vari segmenti di rete, in quanto incomincia a trasmettere un pacchetto alla porta di destinazione prima che esso sia stato interamente ricevuto: lo switch legge l'indirizzo di destinazione, immagazzina i primi 64 byte del pacchetto Ethernet per assicurarsi che non si verifichi una collisione e poi inizia a trasmettere il messaggio.

Un dispositivo store and forward invece attende che tutto il pacchetto venga ricevuto prima di cominciare a trasmettere, immagazzinando i dati in arrivo in opportuni buffer. È evidente che gli switch store and forward introducono una latenza sicuramente maggiore rispetto ai cut through, tuttavia questi ultimi non sono utilizzabili in dispositivi che colleghino porte dalla velocità differente: le porte veloci devono attendere che tutto il pacchetto venga ricevuto da una porta più lenta per non essere costretti a sospenderne la trasmissione ad alta velocità; analogamente una porta lenta non è in grado di trasmettere un pacchetto alla velocità di ricezione di una porta veloce.

Per capire su quale porta inviare il pacchetto, lo switch è dotato di un dispositivo di apprendimento che gli permette di capire, sulla base dei pacchetti ricevuti in ingresso dalle diverse porte quale o quali host si trovano collegati a ciascuna porta.

Uno switch inoltre offre la modalità di comunicazione full-duplex, modalità impossibile da realizzarsi con un hub, in quanto solo un utente può accedere al dispositivo in un dato istante senza provocare collisioni.

L' architettura interna di uno switch può essere scomposta essenzialmente in tre blocchi funzionali:

Controllo di flusso: gestisce i pacchetti instradandoli verso l' opportuna porta di destinazione.

Buffering: tratta la memorizzazione dei pacchetti dal momento in cui essi arrivano allo switch fino a quando non sono ritrasmessi.

Address resolution: si occupa della decodifica degli indirizzi.

Lo switch utilizzato per la connessione dei nodi del cluster ParMa^2 è un Cisco Micro Switch 1548 10/100; esso offre otto porte 10 BaseT/100 Base TX che possono essere usate sia per le connessioni Ethernet che per quelle Fast-Ethernet. È uno switch che può svolgere funzioni sia di switching hub che di switch di segmento, e utilizza una modalità di ritrasmissione di pacchetti di tipo store and forward; per tali motivi Cisco lo ritiene ideale per connessioni ad alte prestazioni di workstation e server. Ciascuna porta supporta azioni di autosensing e autonegoziazione per impostare correttamente la velocità di comunicazione e la modalità full-duplex tra i dispositivi connessi.

2.3 Sviluppi futuri delle System Area Network

Per risolvere il problema della scarsità della larghezza di banda nelle reti usate per connettere i nodi di un cluster, alcuni costruttori di cluster di classe 2 hanno sperimentato reti fisiche diverse dalla Fast Ethernet; alcune di queste, benchè nate per scopi diverse, hanno trovato applicazione in alcuni cluster, come ad esempio ATM, sviluppata per reti con necessità di un efficiente

Tecnologia	Prezzo/Porta anno 96(dollari)	Prezzo/Porta anno 99(dollari)	Variazione (per cento)
Fast Ethernet bus condiviso	137	85	-39
Switched Fast Ethernet	780	390	-50
FDDI bus condiviso	835	650	-22
Switched FDDI	4000	1860	-54
ATM 622 Mbps	6600	4800	-27
Gigabit Ethernet bus condiviso		700	
Switched Gigabit Ethernet		1500	

Tabella 2.1: Confronto tra i prezzi per porta per diverse tipolgie di interconnessione tra gli anni 1996 e 1999. Fonte Dall'Oro Group [10].

supporto per traffico distinto in base alle esigenze di Qualità del Servizio, con l'integrazione di dati, di flusso multimediale e di pacchetti con stringenti esigenze di real-time. Dall'estensione del protocollo Ethernet alle velocità di 1 Gigabit al secondo i cluster possono trarre grande vantaggio, sia per la possibilità di migrare agevolmente da Fast a Gigabit Ethernet, sia per l'uso di uno standard ampiamente supportato. Sono state sviluppate anche delle soluzioni proprietarie, tra cui la più diffusa per i cluster è Myrinet, che consentono ottime prestazioni dal punto di vista della banda e della latenza; gli svantaggi risiedono nella dipendenza da un solo produttore con la mancanza di concorrenza e della conseguente diminuzione dei prezzi.

La tabella 2.1 riporta il confronto tra i prezzi per porta per diverse tecnologie di interconnessione.

Gigabit Ethernet

Gigabit Ethernet [10] è uno standard per comunicazioni a larga banda su reti Ethernet, approvato dalla IEEE con il nome di 802.3z nel giugno 1998; per ora il supporto fisico richiesto è la fibra ottica, ma sono in già in corso di standardizzazione le specifiche[3] per utilizzare velocità di trasmissione di 1 Gigabit al secondo su cavi di rame, in particolare su 4 UTP di classe 5.

Per quanto riguarda il protocollo Gigabit Ethernet non si differenzia sostanzialmente rispetto a Fast Ethernet: infatti oltre alla velocità di comunicazione, i cambiamenti riguardano alcuni parametri relativi alle temporizzazioni delle rivelazioni di collisione e all'introduzione di un nuovo algoritmo per l'invio di pacchetti di piccole dimensioni.

È inoltre in fase di cambiamento il supporto per messaggi dalla diversa Qualità del Servizio (QoS), che è stato ampliato con le specifiche IEEE 802.1p.

Gigabit Ethernet supporta qualsiasi tipo di topologia, purchè si rispettino le specifiche sulle lunghezze dei segmenti e sul numero di nodi: switch con capacità di routing, switch di livello 2, hub e buffered distributors[4] consentono il collegamento dei nodi.

Le caratteristiche del protocollo usato da Gigabit Ethernet consentono una migrazione indolore da Fast Ethernet a Gigabit Ethernet con la sola modifica dei mezzi di interconnessione fisica, senza cambiare per esempio i router di livello 3.

Il prezzo per l'aggiornamento di reti Fast Ethernet a Gigabit Ethernet oggi si mantiene abbastanza sostenuto, ma si prevede che il trend dei prezzi di questa nuova tecnologia segua l'andamento di quello di Switched Ethernet, con una diminuzione del 50 per cento in tre anni.

Myrinet

Myrinet [11] è una tecnologia di proprietà della Myricon per l'interconnessione di cluster di workstation o di PC o di single board computer. Offre una comunicazione a commutazione di pacchetto dalle performance molto elevate con la possibilità di utilizzare qualsiasi tipo di topologia anche attraverso l'uso di appositi meccanismi di switching.

Le caratteristiche peculiari di Myrinet sono:

- collegamenti full-duplex con banda passante di 1.28 Gigabyte al secondo per ogni direzione;

[3]IEEE 802.3ab.

[4]È un dispositivo che come un hub non esegue filtering sugli indirzzi ma si limita a inviare il pacchetto su ogni link; a differenza di un hub può bufferizzare i messaggi in arrivo.

- controllo di flusso, controllo di errore e monitoraggio continuo dei link;
- switching a bassa latenza secondo una modalità cut-through;
- possibilità di qualsiasi topologia di interconnessione tra i nodi;
- alta scalabilità fino a un migliaio di nodi;
- comunicazione diretta tra i processi in spazio utente da parte degli host che si occupano della gestione dell'interfaccia di rete al fine di contenere l'overhead[5];
- diverse implementazioni del livello fisico.

I pacchetti di dati usati da Myrinet possono essere di qualsiasi lunghezza, possono incapsulare altri tipi di pacchetti, come ad esempio quelli IP; ogni pacchetto è identificato da un tipo in modo che Myrinet possa trasportare pacchetti di protocolli diversi in modo concorrente.

Le latenze che tipicamente si ottengono nello scambio di messaggi con Myrinet sono dell'ordine dei 30 μs in sistemi UNIX.

Una completa rete Myrinet è composta da:

- l'interfaccia di rete tra un host e il canale di interconnessione;
- la rete di interconnessione realizzata tramite appositi switch;
- cavi e adattatori per collegare i nodi agli switch;
- Myrinet software.

Per quanto riguarda il tipo di cablaggio necessario, Myricom offre tre possibilità:

SAN. Necessita del tipo di cablaggio più economico; consente di connettere nodi a una distanza massima di 3 metri con cavi che non confinano le interferenze elettromagnetiche e non possono essere calpestati: questo tipo di cablaggio è adatto solo per Sytem Area Network per cluster in-cabinet.

[5]Questo importante concetto è alla base anche dell'interfaccia di comunicazione VIA, che verrà presentata nel prossimo capitolo.

Interfaccia	Memoria	Prezzo
Myrinet-LAN/PCI	512KB	1400
Myrinet-LAN/PCI	1MB	1600
Myrinet-SAN/PCI	512KB	1300
Myrinet-SAN/PCI	1MB	1500

Tabella 2.2: Prezzi delle interfacce Myrinet al produttore nel marzo 1999.

Switch	Porte	Prezzo
Myrinet-LAN	4	1200
Myrinet-LAN	8	2400
Myrinet-LAN/SAN	4/4	1600
Myrinet-LAN/SAN	8/8	5000
Myrinet-LAN	16	6000

Tabella 2.3: Prezzi degli switch Myrinet al produttore nel marzo 1999.

LAN. Utilizza cavo schermato, di robustezza maggiore e segnali differenziali; si possono raggiungere distanze di 10.7 metri.

Fibre ottiche. Con l'uso di cavi in fibra ottica si possono creare link di lunghezze da 550 metri fino a 10 Kilometri.

Le interfacce di rete sono fornite in genere sia per LAN che per SAN, mentre per i cablaggi in fibra occorre acquistare oltre a una delle due precedenti interfacce un apposito adattatore.

Myricom rende pubbliche le specifiche dei propri prodotti, e ciò ha reso possibile la scrittura di driver anche per il sistema operativo Linux.

Per quanto riguarda i costi, come si vede dalla tabella, le schede di rete sono superiori di un ordine di grandezza rispetto a schede Fast Ethernet, mentre gli switch hanno prezzi confrontabili.

2.4 ParMa^2

ParMa^2 è il nome del cluster di PC costruito e sviluppato presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Parma. Il progetto ebbe inizio nel 1995 con l'acquisto di 4 PC con processore Intel Pentium 100, che in seguito furono affiancati da due Pentium Pro a 200 MHz. Obiettivo

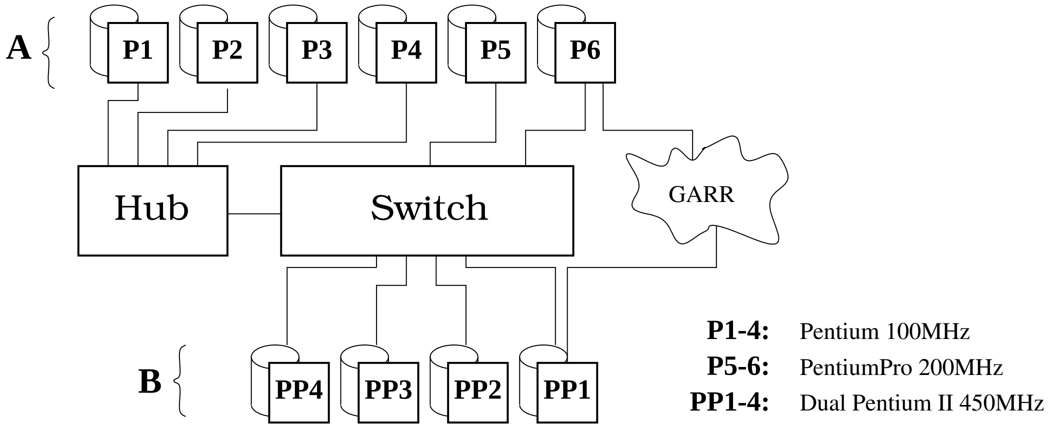

Figura 2.1: Attuale struttura di ParMa^2.

del progetto è quello di realizzare un sistema parallelo capace di ottenere un rapporto prezzo-prestazioni inferiore a quello delle macchine parallele dedicate, utilizzando solo componenti off-the-self; ParMa^2 appartiene quindi alla classe 1 dei cluster di PC. ParMa^2 è inoltre usato per l'analisi e lo sviluppo di programmi paralleli.

L'architettura attuale, come riportato nella figura 2.1, è il risultato dell'evoluzione di un primo cluster sperimentale, che ora è definito come sottocluster A, costituito dai primi sei nodi, a cui sono stati aggiunti nell'ottobre del 1998 quattro nuovi nodi con macchine basate su Pentium II Dual Processor a 450 MHz, che formano il sottocluster B e portano il totale dei nodi a dieci:

- 4 nodi Dual Pentium II a 450 MHz - 256 Mbyte RAM;
- 2 nodi PentiumPro 200 MHz - 64 Mbyte RAM;
- 4 nodi Pentium 100 MHz - 64 Mbyte RAM;

Il sistema è accessibile all'esterno attraverso uno dei nodi costituiti da Pentium Pro ed ora anche da uno dei nuovi nodi con Pentium II.

Prossimamente anche i nodi più vecchi costituiti da Pentium 100 verrano connessi tra loro attraverso uno switch identico a quello usato per gli altri nodi.

È stato installato il sistema operativo Linux nella distribuzione Debian 2.0 con una dotazione standard di applicazioni, e inoltre sono funzionanti anche due diversi tipi di librerie a scambio di messaggi per la programmazione parallela MPI, cioé MPICH 1.1 e LAM 6.1, le cui prestazioni verranno analizzate nel capitolo 6.

2.4.1 Sottocluster A

È costituito da sei nodi eterogenei dal punto di vista dell'hardware, omogenei dal punto di vista del sistema operativo; due nodi sono connessi tramite uno switch Fast Ethernet, gli altri per ora sono collegati a un hub Fast Ethernet. Nel complesso quindi il sottocluster A è costituito da:

- 2 nodi Pentium Pro 200 MHz - 256 Kbyte cache asincrona on chip - 64 Mbyte RAM - sceda di rete 3Com con chipset Digital Tulip 21140;
- 4 nodi Pentium 100 MHz - 512 Kbyte cache sincrona - 64 Mbyte RAM - scheda di rete 3Com con chipset Digital Tulip 21140;
- 1 hub Fast Ethernet 3Com;
- 1 scheda di rete 3Com Vortex Ethernet residente su un Pentium Pro per l'esterno.

2.4.2 Costruzione del sottocluster B

Il sottocluster B consiste di 4 nodi omogenei con le seguenti caratteristiche:

- doppio processore Pentium II a 450 MHz con cache L1 di 32 Kbyte e L2 di 512 Kbyte all'interno del contenitore della CPU;
- motherboard ASUS P2B-DS con bus a 100 MHz;
- memoria RAM di 256 Mbyte in 2 DIMM PC100;
- scheda di rete D-LINK DFE-500TX basata su chipset DEC Tulip 21140;
- scheda video S3 Virge GX2 AGP con 4Mbyte di RAM;
- architettura SCSI con controller integrato sulla motherboard di tipo Ultra2 Wide SCSI realizzato con il chip Adaptec AIC 7890;
- hard disk Ultra Wide SCSI da 4.3 Gbyte;
- switch Cisco Fast Ethernet del tipo Micro Switch 1548.

Inoltre un nodo è equipaggiato con una scheda di rete Fast Ethernet 3Com per la connessione con la rete GARR.

Come si vede l'architettura dei nodi del sottocluster B è basata su alcune caratteristiche introdotte recentemente sul mercato dei Personal Computer (ad esempio il bus processore-memoria a 100 MHz) che sono state oggetto di un attento studio e di indagini di mercato al fine di determinare i componenti più adatti all'utilizzo in ambiente Linux Symmetric Multi Processor e alla programmazione parallela, prestando attenzione al rapporto prestazioni-prezzo.

Il primo componente preso in esame per la costruzione del nuovo sottocluster è stata la scheda madre, poichè essa, come descritto in precedenza, influenza notevolmente l'organizzazione e l'espandibilità del nuovo sistema e può essere causa di malfunzionamenti ed incompatibilità in ambiente Linux SMP.

Le caratteristiche richieste sono state:

- supporto per Dual Pentium II con frequenze di almeno 400 MHz;
- bus processore-memoria a 100 MHz;
- utilizzo del chipset Intel BX;
- supporto di almeno 512 Mbyte di RAM in DIMM;
- supporto al bus AGP;
- conformità al protocollo PCI 2.1;
- dotazione standard di porte seriali, parallela, PS/2.

Dopo una ricerca tra più di trenta produttori di motherboard sono state elencate circa venti schede madri soddisfacenti ai requisti[6], tra le quali è stata operata una scelta in base ai seguenti criteri:

- Esperienze documentate di utilizzo in ambiente LINUX SMP;
- Reperibilità sul mercato italiano;
- Disponibilità di più modelli di uno stesso produttore, ad esempio con e senza controller SCSI;

[6] La documentazione a riguardo è reperibile presso la pagina web all'indirizzo `ftp://ftp.ce.unipr.it/pub/ParMa2/mb.doc.gz`.

- Notorietà ed affidabilità del produttore;
- Prezzo;
- Esperienze varie di utilizzo.

Si è ritenuto di optare per una architettura SCSI[7] in quanto nettamente superiore all'IDE/EIDE soprattutto in ambiente multitasking e multiutente, per cui si è proceduto all'acquisto di motherboard ASUS P2B-DS con le seguenti caratteristiche:

- supporto di uno o due Pentium II da 233 a 450 MHz;
- velocità del bus impostabile a 66 o 100 MHz;
- utilizzo del chipset Intel 440BX;
- supporto fino a 1 Gbyte di SDRAM in DIMM di tipo PC 100;
- slot AGP;
- controller Adaptec AIC 7890 Ultra2 Wide SCSI;
- conformità al protocollo PCI 2.1 che garantisce PCI bus mastering concorrente e supporto all'UltraDMA/33;
- funzionalità di controllo dello stato dell'hardware (DMI), di risparmio energetico e antivirus.

2.4.3 Il sistema operativo

Tutti i nodi utilizzano Linux, un sistema operativo gratuito appartenente alla famiglia di sistemi operativi UNIX-type, per architetture Intel x86, Sparc, Alpha e alcune altre, come nella tradizione dei cluster di classe 1, per molteplici ragioni:

- Linux è un sistema operativo completo, affidabile e robusto;
- Linux richiede solamente una licenza di tipo GNU; è quindi un prodotto software gratuito, ridistribuibile e modificabile dal momento che vengono forniti anche i sorgenti;

[7] In appendice sono riportate alcune specifiche di questo standard.

- il numero di applicazioni disponibili per tale sistema operativo è molto elevato e la maggior parte sono gratuite;
- lo sviluppo del kernel di Linux e delle applicazioni è condotta da una vasta comunità di ricercatori, che producono senza sosta nuove versioni per migliorare le prestazioni o risolvere problemi e adattare il sistema operativo ai nuovi componenti;
- Linux è conforme allo standard POSIX;
- Linux possiede una tra le più efficienti implementazioni del protocollo TCP/IP;
- i device driver offrono un supporto esteso a un gran numero di perifiriche.

L'insieme del sistema operativo, dei tools di utilità e delle applicazioni costituisce una *distribuzione*: è stata scelta la distribuzione Debian 2.0 in quanto offre una utility di gestione dei pacchetti applicativi particolarmente flessibile ed efficiente e include solitamente le versioni più recenti dei pacchetti applicativi.

Il kernel del sistema operativo è stato continuamente aggiornato, partendo dalla ultima versione stabile disponibile al momento della costruzione del cluster, la 2.0.35[8] che offre il supporto per sistemi SMP in modo sperimentale. Si è quindi deciso di installare un kernel della famiglia 2.1.12x[9] con l'opzione SMP che è invece supportata ufficialmente; tali kernel, benchè in fase di sviluppo, non hanno evidenziato seri problemi di stabilità e gestiscono in maniera efficace i due processori suddividendo in modo automatico il carico di lavoro.

[8]L'utilizzo delle versioni del kernel 2.0.3x sono possibili a patto di utilizzare una specifica patch per il controller SCSI Adaptec della serie 7xxx.

[9]La seconda cifra dispari indica che si tratta di kernel sperimentali.

Capitolo 3

Virtual Interface Architecture

Nel capitolo precedente ci si è occupati di due dei tre componenti che costituiscono un cluster di PC, cioé del nodo di calcolo e della rete di interconnessione: grazie soprattutto ai notevoli e costanti miglioramenti della tecnologia i PC hanno incrementato in modo significativo la loro potenza computazionale, rendendo la soluzione di costruire un sistema distribuito secondo il modello cluster di PC ancora più competitiva rispetto ai sistemi dedicati. Tuttavia un buon sistema deve essere prima di tutto equilibrato, evitando per quanto possibile colli di bottiglia che inficiano le prestazioni dell'intero sistema. Nelle architetture di tipo cluster di PC attualmente il fattore limitante è costituito dalla comunicazione tra i nodi, che avviene per mezzo di una rete di comunicazione costituita da elementi hardware e software, utilizzando un protocollo di comunicazione che stabilisce le modalità di trasferimento dei dati. Per quanto riguarda le rete di connessione tra i nodi, come si è visto, sono disponibili a costi accessibili tecnologie in grado di fornire una banda passante vicino al Gigabit al secondo: il collo di bottiglia viene in questo caso spostato sul protocollo di comunicazione che non sempre è in grado di sfruttare pienamente le risorse messe a disposizione dai canali fisici di connessione.

Il protocollo di comunicazione utilizzato in un cluster di PC deve assicurare le seguenti caratteristiche [12]:

- bassa latenza, secondo la definizione data nel modello LogP;
- banda passante elevata tendente al massimo permesso dalla rete di comunicazione;
- overhead limitato;

- scarso intervento della CPU nel trasferimento dei dati.

Si analizzano brevemente le modalità con cui il sistema operativo Linux implementa il protocollo TCP/IP al fine di capirne le principali limitazioni [13] [14]. Linux realizza la famiglia di protocolli Internet come una serie di livelli software tra loro connessi, come si vede dalla figura 3.1. Un processo che desideri comunicare deve utilizzare una socket[1], che rappresenta una estremità di un canale di comunicazione: la socket deve essere creata, associata a un indirizzo, connessa a un'altra socket attraverso una procedura di tipo client-server. Queste operazioni richiedono che il kernel del sistema operativo intervenga per creare e inizializzare opportunamente una serie di strutture dati destinate a contenere le informazioni necessarie per condurre la comunicazione; bisogna rilevare come ogni livello mantenga le proprie strutture dati, che devono essere aggiornate durante l'avanzamento della comunicazione. Nel momento in cui un processo deve inviare o ricevere dati esso invoca una system call che provoca l'intervento del kernel il quale trasmette i messaggi attraverso i vari livelli aggiornando tutte le strutture dati relative, fino ad arrivare al livello dei device driver; ogni device driver registra una serie di routine di callback che i livelli del protocollo di networking invocano per effettuare le operazioni di trasferimento dei dati.

I fattori quindi che introducono overhead nelle comunicazioni sono [15]:

- il numero di livelli software che devono essere attraversati per l'effettuazione di una comunicazione;

- gli interrupts che devono essere obbligatoriamente generati in alcune fasi della comunicazione;

- il context switch che avviene ogni volta che il sistema operativo deve intervenire per portare avanti la comunicazione;

- le copie dei dati che vengono effettuate passando da un livello al successivo o da un processo al sistema operativo.

Proprio per cercare di evitare le operazioni descritte in precedenza sono state sviluppate possibili alternative allo stack TCP, con lo scopo soprattutto di diminuire la latenza e l'overhead connesso al trasferimento dei dati: esperienze come U-Net [16], Active Message (AM) [17], Fast Messagge (FM)

[1] L'interfaccia BSD socket supporta anche la comunicazione interprocesso.

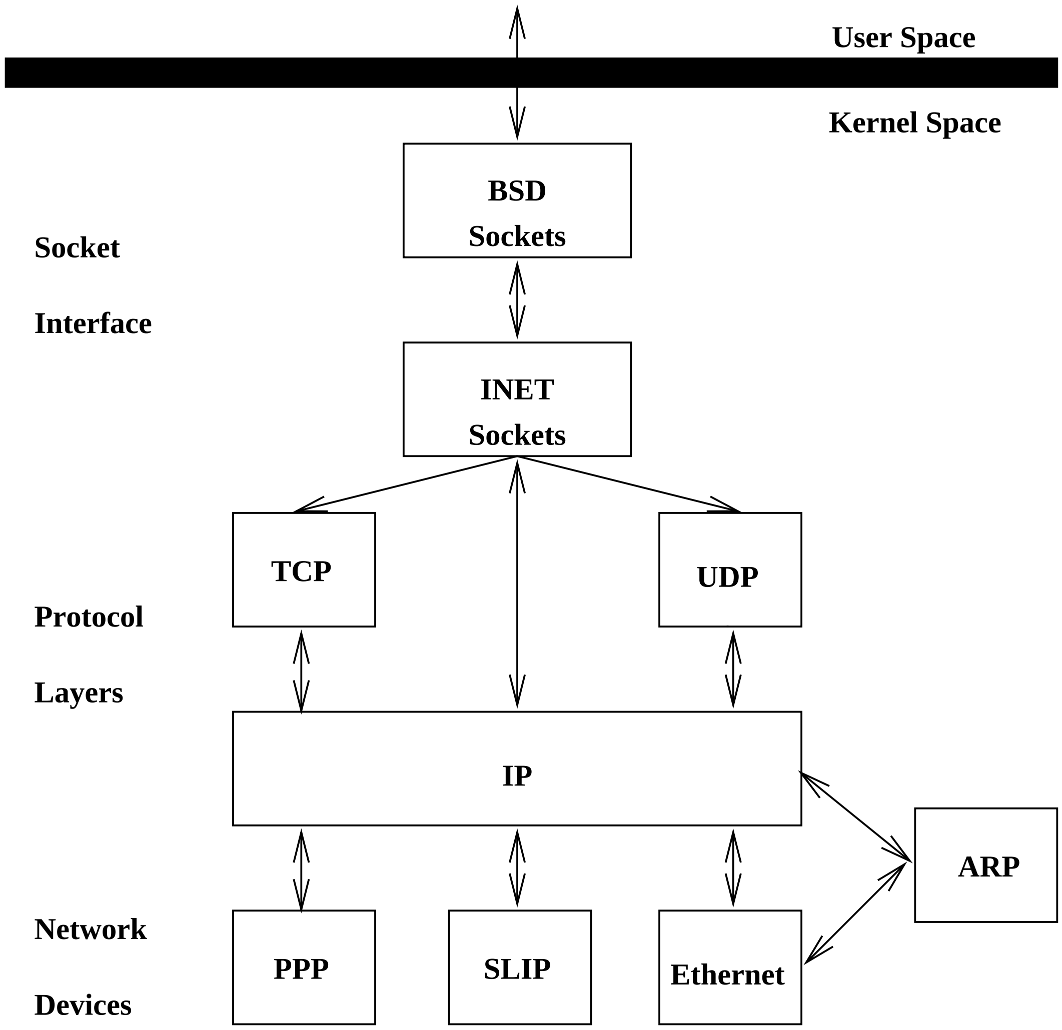

Figura 3.1: Livelli del networking in Linux.

[18], sono alcuni esempi che hanno ottenuto buoni risultati dal punto di vista delle prestazioni ma sono rimaste esperienze limitate all'ambito della ricerca con scarsa interazione con il sistema industriale; tuttavia alcuni dei loro concetti fondamentali sono stati ripresi dai progettisti della Virtual Interface Architecture (VIA) [19], uno standard per comunicazioni a bassa latenza promosso da Intel [20], Microsoft e Compaq, con l'obiettivo di risolvere in modo organico le problematiche in precedenza enunciate [21] [22].

Il concetto alla base di VIA è di offrire a un processo utente accesso diretto all'interfaccia con la rete, senza bisogno di intervento da parte del sistema operativo e in modo tale che il processo abbia l'illusione di possedere la scheda di rete per eseguire tutti gli scambi di dati necessari per i quali VIA definisce un insieme di funzioni, delle strutture dati ed una relativa semantica.

3.1 Il modello architetturale di VIA

Virtual Interface Architecture (figura 3.2) si compone di quattro elementi essenziali [21]:

- un VI Provider,
- uno o più VI Consumer,
- una o più Virtual Interface,
- una o più Completion Queues,

che interagiscono tra loro per assicurare la comunicazione.

3.1.1 VI Provider

Il VI Provider è l'insieme dei componenti hardware e software che consentono la creazione di tutte le strutture necessarie alla comunicazione, in particolare le Virtual Interface (VI), le Completion Queues (CQ), le aree di memoria coinvolte nel trasferimento dei dati e le strutture dati usate dalle VI. Il VI Provider è composto dal Network Interface Controller (NIC) e dal Kernel Agent.

Il *VI NIC* costruisce le VI e le CQ. Attualmente, poichè il VI NIC è completamente emulato via software, le VI e le CQ sono realizzate utilizzando

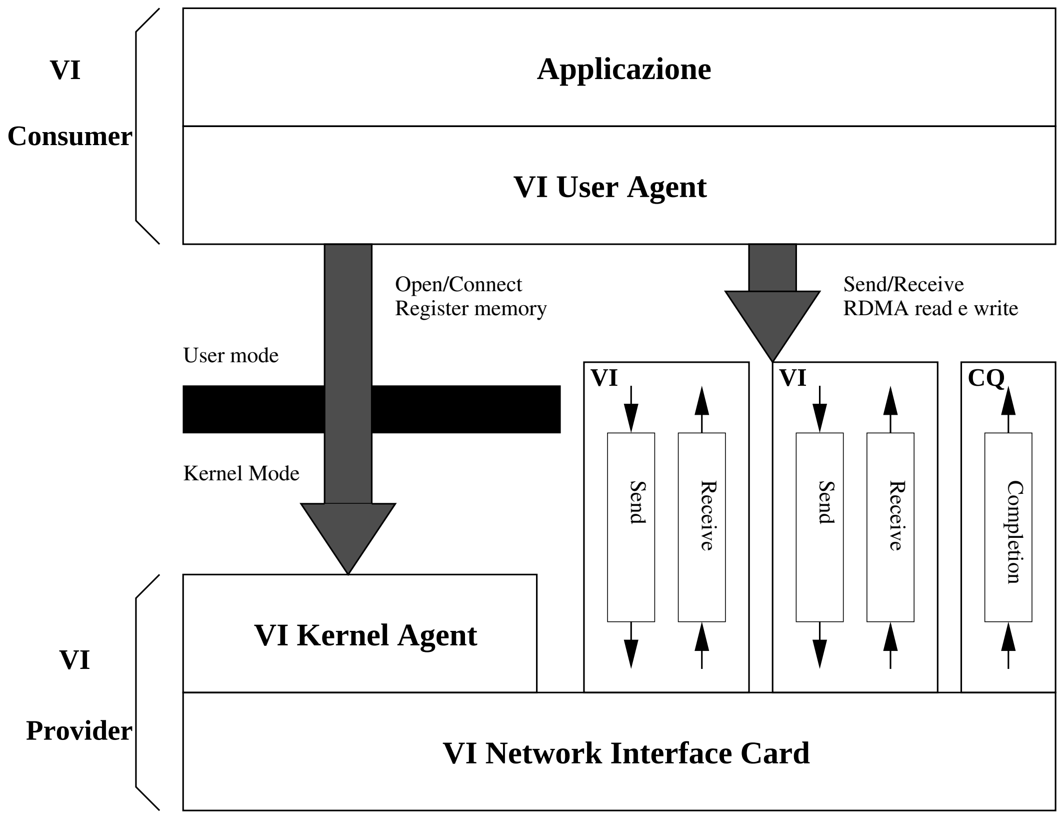

Figura 3.2: Modello architetturale di VIA.

opportune zone della memoria di sistema, ma si prevede che con la futura introduzione di schede di rete per VIA[2], le VI e le CQ potranno utilizzare registri inseriti sulla VI NIC stessa.

Dopo la creazione delle VI e delle CQ, il VI NIC deve farsi carico di provvedere al trasferimento dei dati operando in modo asincrono rispetto al resto dei componenti di VIA, cioé senza il loro ulteriore intervento una volta che una delle Virtual Interface abbia inizializzato una richiesta di scambio di dati.

Il *Kernel Agent* è una parte del sistema operativo che fornisce tutte le funzioni necessarie per la gestione delle Virtual Interface tra i VI consumer e la VI NIC.

Nelle operazione di creazione, distruzione, connessione e disconnessione di VI, gestione della memoria usata dal VI NIC e dai VI Consumer per le operazioni di trasferimento dati vengono utilizzate funzioni del Kernel Agent. Mentre le operazioni di scambio di dati (Send/Receive e RDMA) non utilizzano il Kernel Agent ma coinvolgono solo il VI Consumer ed il VI NIC che si occupa della trasmissione fisica dei bit di dati. In questo modo il ricorso al Kernel Agent, e quindi l'intervento del kernel del sistema operativo, è necessario solo nel momento di creazione, modifica e distruzione della VI, mentre le operazioni di trasmissione e ricezione dei dati non richiedono invece l'intervento del kernel come invece avviene con i tradizionali protocolli di comunicazione.

Il Kernel Agent svolge un ruolo simile a quello dei device driver facendosi inoltre carico di tener traccia di tutte le risorse associate all'uso di un VI NIC da parte di un VI Consumer.

3.1.2 VI Consumer

VI Consumer è un generico utilizzatore di una Virtual Interface. Il VI Consumer si avvale di VIA usando le funzioni definite dallo *User Agent*, una interfaccia che rende trasparenti all'utilizzatore del protocollo di comunicazione i dettagli realizzativi delle altre tre componenti di VIA.

In questo modo l'evolversi della realizzazione di VIA che porterà dalla sua emulazione software alla sua realizzazione definitiva con schede hardware dedicate sarà indolore per chi vuole già da ora sfruttare le potenzialità di VIA.

[2]Alcune schede di rete per VIA sono state rilasciate da Giganet, Finisar e Tandem.

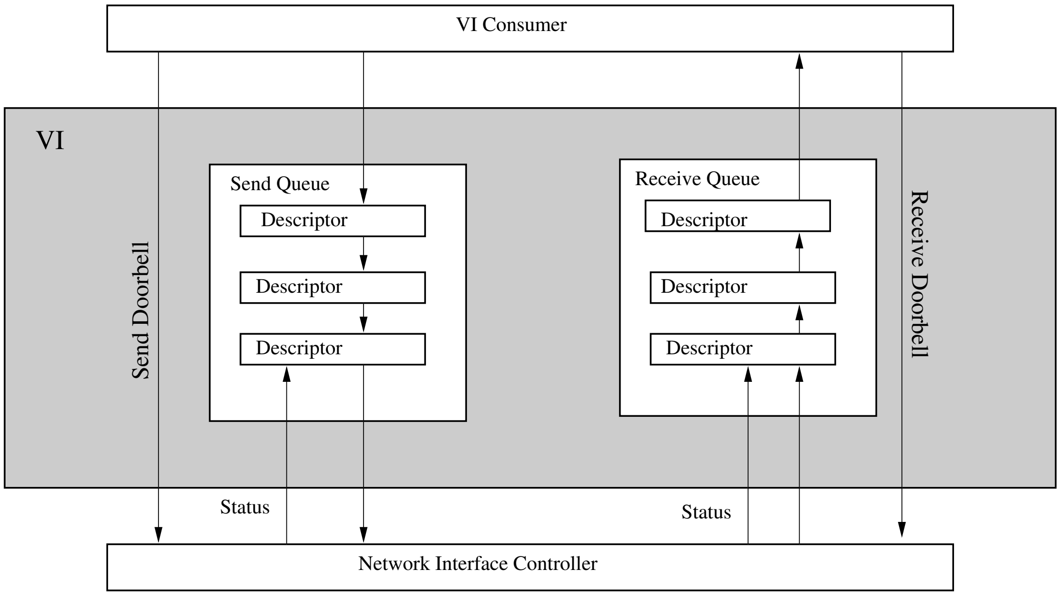

Figura 3.3: Virtual Interface.

Lo User Agent definisce quindi l'insieme delle funzionalità offerte da VIA che un programmatore può utilizzare nelle sue applicazioni.

3.1.3 Virtual Interface

La Virtual Interface (VI) è il meccanismo con cui un VI Consumer accede al VI NIC per eseguire i trasferimenti dei dati. La Virtual Interface crea l'illusione che ogni processo sia proprietario dell'interfaccia di rete evitando il multiplexing tra i processi da parte del sistema operativo per abilitare un processo al trasferimento di dati, come invece avviene ad esempio nel caso di TCP.

Una VI (figura 3.3) è composta da due code (*Send Queue* e *Receive Queue*) e da due *Doorbell*. Ogni coda è costituita da una lista concatenata di descrittori. Ogni *descrittore* è una complessa struttura dati che contiene le informazioni necessarie al VI Provider per effettuare i trasferimenti dei dati.

Il funzionamento della Virtual Interface può essere brevemente così descritto. Quando un processo desidera comunicare deve prima di tutto creare tramite il Kernel Agent una VI. A questo punto può accodare (operazione di *posting*) un descrittore in una delle due code: la Send Queue se desidera mandare dati o la Receive Queue se desidera riceverli. In conseguenza dell'accodamento viene segnalato al VI NIC la richiesta di una nuova operazione di trasferimento attraverso la modifica della Doorbell associata alla coda. La *Doorbell* è un registro del VI NIC che non richiede l'intervento del kernel quando viene modificato.

Per consentire al VI NIC di effettuare lo scambio di dati in modo asincrono la proprietà del descrittore accodato viene trasferita dal processo al VI NIC. Da questo momento sarà il VI NIC che si occuperà di gestire il descrittore andando a modificare opportunamente le informazioni di stato senza fare intervenire il kernel del sistema operativo e quindi senza causare interrupt che provocano inevitabilmente overhead che contribuisce ad aumentare il tempo di trasmissione.

Al VI Consumer viene comunque lasciata la possibilità di controllare lo stato del descrittore e di toglierlo dalla coda quando l'operazione di comunicazione sia terminata, anche se essa non sia andata a buon fine.

Per far ciò il VI Consumer ha a disposizione due diverse semantiche per rilevare il completamento di un descrittore: polling e operazioni bloccanti. Nel *polling* il processo esamina lo stato del descrittore in cima alla coda: nel caso in cui sia stato completato lo toglie dalla coda, altrimenti prosegue la sua esecuzione. Al contrario quando il VI Consumer utilizza *operazioni bloccanti* si ferma in attesa che l'operazione di comunicazione associata al descrittore venga completata.

3.1.4 Completion Queues

Le Completion Queue (CQ) consentono ad un processo di unire in una coda le notifiche di completamento delle operazioni di comunicazione associate ai descrittori che appartengono a code dello stesso tipo appartenenti a diversi VI. Quando un descrittore postato su di una coda associata ad una CQ viene completato il VI NIC pone un puntatore al descrittore sulla CQ.

Se al momento della creazione di una VI si specifica di associare una coda ad una CQ, la verifica del completamento di un descrittore, sia essa di tipo bloccante o a polling, non deve più avvenire sulla coda ma bensì sulla CQ. Anche in questo caso la notifica e la verifica del completamento avvengono senza interrupts.

3.2 Modello di comunicazione con VIA

Quando un processo instaura una comunicazione utilizzando VIA attraversa una serie di fasi che verrano qui brevemente descritte. Nei paragrafi successivi verrano riprese in maggiore dettaglio includendo anche la spiegazione della sintassi delle istruzioni associate.

Creazione di un handle per un VI NIC: questa operazione consente di specificare il VIA device driver da utilizzare per la comunicazione, associando il processo al VI NIC; nelle operazioni successive per individuare univocamente il VI sarà sufficiente utilizzare l'handle restituito al processo dal VI NIC tramite il Kernel Agent.

Creazione di un protection tag: durante questa fase un protection tag viene associato a una VI; la protection tag è un meccanismo di sicurezza che consente a una o più VI di accedere ad una zona di memoria solo se i rispettivi protection tag coincidono.

Registrazione della memoria: ogni zona di memoria utilizzata per i trasferimenti di dati (descrittori o buffer di memoria) deve essere preventivamente registrata inviando una richiesta di registrazione al Kernel Agent che restituisce un Memory Handle; è possibile specificare diversi attributi e una protection tag.

Creazione di una Completion Queue: la creazione della CQ viene effettuata dal VI Provider che si occupa dell'allocazione in memoria nello spazio degli indirizzi e della notifica al VI NIC della locazione nella zona riservata alla CQ. Una CQ deve essere creata prima della VI che desideri utilizzarla.

Creazione di una VI: durante la creazione di una VI da parte del VI Provider secondo gli attributi specificati nella richiesta del VI Consumer, vengono creati una Send Queue, una Receive Queue e una Doorbell associata ad ogni coda; lo stato finale della VI è di non connessione (*idle*).

Connessione di una VI: questa operazione è necessaria perché VIA utilizza un modello di comunicazione connection-oriented, per cui due VI appartenenti a processi diversi possono scambiarsi dati solo se si trovano in uno stato di connessione, che può essere instaurata secondo due modalità, client-server o peer-to-peer.

Inizializzazione dei descrittori: è la fase di preparazione dei descrittori precedentemente registrati, durante la quale vengono indicati l'indirizzo e il Memory Handle del buffer contenente i dati che si intende inviare.

Posting dei descrittori: i descrittori inizializzati con le corrette informazioni vengono postati su una delle due code di una VI connessa. Equivale ad una generica operazione di send o receive non bloccante.

Sincronizzazione con il completamento dei descrittori: durante questa fase il processo verifica il completamento (con successo o con fallimento) del descrittore dell'operazione richiesta utilizzando una semantica bloccante o di polling sulle code di Send o di Receive o, nel caso esista, su una Completion Queue.

Disconnessione della VI: quando il processo ha terminato le operazioni di comunicazione utilizzando la VI, essa deve essere disconnessa.

Distruzione della VI: dopo la disconnessione il processo deve richiedere la distruzione delle VI disconnesse e con le code vuote al VI che dealloca le zone di memoria riservate alle code e ai Doorbell.

Distruzione della Completion Queue: quando tutte le VI che utilizzino una CQ sono state distrutte, si passa alla fase di rimozione della CQ.

Deregistrazione della memoria: in questa fase la memoria non più utilizzata viene deregistrata dal Kernel Agent.

Chiusura del VI NIC nel processo: per concludere l'uso di VIA, l'handle del VI NIC viene distrutto e il VI NIC chiuso.

3.3 Accesso a un VI NIC

Tutte le funzioni della VI architecture, dalle operazioni di registrazione della memoria alla creazione delle Completion Queue e delle Virtual Interface dall'ottenimento di informazioni sul NIC alla gestione degli errori, richiedono un identificatore (*handle*) del VI NIC che si vuole utilizzare. Per questo la prima operazione da svolgere consiste nella richiesta di un handle da parte del VI Consumer al VI Provider, che viene fornito con le schede VIA o che viene emulato in modo software. La richiesta viene compiuta con la seguente istruzione:

VipOpenNic(
    IN **const** VIP_CHAR ∗DeviceName,
    OUT VIP_NIC_HANDLE ∗NicHandle
)

Il processo indica in ∗DeviceName il nome simbolico del device driver che intende utilizzare, quindi il Kernel Agent, dopo aver verificato l'esistenza dei device restituisce in ∗NicHandle l'identificatore che rappresenta il VI NIC prescelto. È possibile scegliere tra un device di *loopback*, da utilizzare per processi che risiedono sullo stesso nodo, e un device di rete.

Un processo può ottenere accesso a diversi VI NIC contemporaneamente, ma le risorse relative ai diversi VI NIC non possono in alcun modo essere condivise.

3.4 Gestione della memoria

Molti sistemi operativi richiedono che le pagine di memoria usate per contenere i dati per le comunicazioni siano bloccate nella memoria fisica e il loro indirizzo virtuale sia tradotto in quello fisico prima che la scheda di rete possa accedervi; questa operazione viene ripetuta per ogni dato trasmesso o ricevuto, generando overhead.

Anche VIA richiede che la memoria utilizzata per il trasferimento dei dati, sia buffer che descrittori, sia bloccata nella memoria fisica: questa operazione viene chiamata registrazione della memoria, che avviene però una sola volta per ogni buffer che può essere riusato nelle comunicazioni e al di fuori delle operazioni di trasmissione dei dati. Il procedimento di registrazione della memoria viene richiesto da un VI Consumer a un VI Provider il quale riserva per il buffer da registrare una o più pagine fisiche contigue, indicate come una regione di memoria, blocca le pagine in memoria fisica e inserisce un entry in una Page Table, gestita dal Kernel Agent per contenere la traduzione tra indirizzo virtuale e fisico assieme alle informazioni di protezione; infine restituisce un handle per la regione di memoria registrata.

Al VI Consumer è permesso di specificare in modo arbitrario l'allineamento e le lunghezze delle regioni di memoria che devono essere registrate, ma la traduzione e gli attributi di memoria della regione sono applicati a ogni pagina fisica completa anche se il buffer da registrare ha una dimensione inferiore.

Quando una regione di memoria è registrata, quindi, ogni pagina all'interno della regione è bloccata nella memoria fisica: questo garantisce al VI NIC che la regione di memoria sia sempre fisicamente disponibile e che la traduzione da indirizzo virtuale a fisico rimanga fissa mentre il VI NIC sta elaborando le richieste che si riferiscono a quella regione.

L'handle restituito dal VI Provider identifica univocamente la regione registrata. Il VI Consumer ogni volta che fa riferimento a un dato coinvolto in operazioni di comunicazione, deve associare a ogni indirizzo virtuale il corrispondente handle restituito dall'operazione di registrazione, affinchè, una volta che il dato sia inviato o ricevuto, il VI Provider possa risalire al suo indirizzo fisico senza bisogno di intervento del kernel.

La memoria registrata da ciascun processo è accessibile da ogni altro thread dello stesso processo, ma un altro processo non può accedere a una zona di memoria non sua usando il gestore del processo proprietario: questo tentativo porterebbe ad un errore di protezione della memoria.

La registrazione della memoria può fallire per l'incapacità del VI Provider di trovare un entry della Page Table o una zona di memoria fisica grande abbastanza per contenere la regione di memoria; in questo caso la memoria non viene registrata.

Lo stesso intervallo di indirizzi virtuali può essere registrato più di una volta, generando multipli gestori della memoria.

Un VI Consumer deve deregistrare una regione di memoria una volta che non abbia più necessità di usare tale memoria: questo consente al Kernel Agent di rimuovere l'entry corrispondente nella Page Table e di sbloccare le pagine riservate nella memoria fisica. I descrittori postati contenuti in una zona di memoria deregistrata vengono completati con un errore di violazione di protezione.

Dal momento che VIA offre un modello di comunicazione orientato alla connessione, gli scambi di dati in diverse regioni di memoria sono consentite solo tra una coppia di VI connesse.

Tuttavia dal momento che un processo può possedere numerose VI è bene disporre della possibilità di limitare gli accessi a specifiche regioni di memoria registrate dal processo: per questo scopo VIA dispone di due meccanismi, i tag di protezione della memoria (*protection tag*) e gli attributi di protezione della memoria.

I tag di protezione della memoria sono degli identificativi univoci costituiti da un numero intero che vengono associati sia a una VI che alle regioni di memoria; ogni tag di protezione della memoria deve essere unico all'interno di un VI Provider che può servire numerosi processi; è quindi compito del VI Provider assicurarsi che i tag di protezione associati a VI o a regioni di memoria registrata siano validi solo per il VI Consumer che ha creato il tag. Un VI Provider deve essere in grado di fornire almeno un protection tag per ogni VI che viene creata.

Il VI Consumer richiede l'assegnazione un tag di protezione della memoria alla regione di memoria quando essa viene registrata e a una VI quando questa viene creata; tale tag può essere rimpiazzato successivamente con un tag diverso cambiando gli attributi della regione di memoria o del VI.

Un qualunque accesso a un dato in memoria è consentito da un VI NIC solo se il tag di protezione della memoria con cui il dato è stato registrato è identico a quello della VI che vuole accedere a tale dato; gli accessi che violano queste regole causano un errore di protezione della memoria e nessun dato può essere trasferito. La convalida dei tag di protezione viene applicata sia alle regioni di memoria registrate che sono utilizzate per contenere descrittori, sia alle regioni di memoria che contengono dati.

Nel caso in cui un VI Consumer non desideri discriminare le aree di memoria da usare con diverse VI, si può utilizzare lo stesso protection tag per tutti i dati e i descrittori di tutte le VI.

Gli attributi di protezione della memoria realizzano la seconda modalità di protezione della memoria offerta da VIA. Essi, come i protection tag, sono associati sia alle regioni di memoria che alle VI e sono usati per controllare il tipo di operazioni che può essere svolto su una determinata zona di memoria abilitando RDMA Read e RDMA Write. Anche in questo caso gli attributi della memoria sono assegnati alle zone di memoria quando esse sono registrate e alle VI quando vengono create e possono essere modificati successivamente cambiando gli attributi delle regioni di memoria o quelli delle VI.

Gli attributi di protezione della memoria hanno un duplice utilizzo: innanzitutto in fase di connessione possono collegarsi con successo solo VI con i medesimi attributi di protezione della memoria. In secondo luogo, al momento dell'effettuazione di una operazione di trasferimento di dati di tipo RDMA vengono controllati gli attributi del dato e delle VI implicate nella comunicazione: se gli attributi coincidono l'operazione viene svolta, nel caso invece in cui gli attributi di protezione della memoria siano in disaccordo, l'attributo che viene utilizzato è quello che garantisce la protezione più stringente; ciò può significare che l'operazione non può essere eseguita, come ad esempio il caso di una VI con RDMA Read abilitato che voglia leggere tramite RDMA Read un dato con solo RDMA Write abiltato.

Anche in questo caso una operazione RDMA che viola i permessi consentiti dagli attributi provoca un errore di protezione della memoria e nessun dato può essere trasferito.

3.5 Creazione e distruzione di una VI

Una VI viene creata da un VI Provider su richiesta di un VI Consumer.

Una VI consiste di una coppia di Work Queue, una Send Queue e una Receive Queue, e una coppia di Doorbell, una per ogni Work Queue: all'atto della creazione di una VI è dunque necessario che il VI Provider riservi la memoria per costruire queste strutture. Per quanto riguarda infatti le Work Queue, esse sono allocate nella memoria virtuale di un processo dal VI Provider, che svolge operazioni simili alla registrazione di una zona di memoria: blocca le pagine di memoria fisica, effettua la traduzione tra indirizzo fisico e virtuale e notifica al VI NIC la locazione delle strutture create.

I Doorbell invece sono stati pensati come una risorsa hardware collocata nel VI NIC; è compito ancora del Kernel Agent di mappare queste risorse nello spazio di indirizzi virtuali di un processo usando le operazioni standard del sistema operativo. Dal momento che lo spazio usato per creare un Doorbell è limitato, ma è comunque necessario riservare una intera pagina di memoria, per evitare di consumare la maggior parte dello spazio di indirizzi virtuali di un VI Consumer, è conveniente che il VI Provider mappi Doorbell multipli in una singola pagina se un VI Consumer crea più di una VI; Doorbell che appartengono a differenti processi devono invece essere mappati in pagine diverse.

Solo dopo la creazione una VI può essere connessa, e nessun dato viene trasferito finchè una VI non è correttamente connessa con un'altra VI.

Quando un processo non ha più bisogno di una VI per comunicare effettua l'operazione di distruzione della VI, in cui un VI Consumer incarica il VI Provider di deallocare e liberare tutte le strutture dati create, le Work Queue e i Doorbell. Una VI non può essere distrutta se rimangono dei pacchetti in fase di trasmissione, nè se si trova in uno stato diverso da *idle* (si veda la figura 3.5).

3.6 Modelli di connessione

VIA fornisce un servizio di trasferimento dati di tipo connection-oriented; all'atto della sua creazione, una VI non risulta essere associata ad alcun'altra VI e pertanto, per rendere possibile il trasferimento dei dati, è necessario attuare un procedimento specifico per la connessione con un'altra VI; alla fine del trasferimento dei dati è necessario disconnettere la VI precedentemente connessa.

I modelli di connessione possibili sono due: un modello di tipo client-server e uno di tipo peer-to-peer (con questo termine verrà indicata d'ora in poi qualunque tipo di connessione in cui non sia riscontrabile una relazione gerarchica, tipo client-server, tra i processi coinvolti). Questi due modelli di connessione risultano essere completamente distinti e le applicazioni coinvolte debbono utilizzare lo stesso modello su entrambe le parti della medesima connessione. Nei limiti in cui rimane rispettato questo vincolo una applicazione può utilizzare indifferentemente entrambi i modelli.

3.6.1 Modello client-server

Il modello di connessione client-server realizza una semantica esclusivamente di tipo bloccante. Sia per il client che per il server il VI Consumer deve effettuare una richiesta al VI Provider per ottenere la connessione tra la sua Virtual Interface e una VI remota; il VI Provider si occupa di realizzare protocolli di connessione affidabili e robusti.

Il processo di connessione, il cui schema è riportato in fig.3.4 segue le seguenti fasi: il server si pone in attesa di una richiesta di connessione e quando questa arriva da un client che ha precedentemente creato la propria VI, la analizza e controlla che essa risulti accettabile per gli attributi che specifica; se questo si verifica crea una VI locale che diventerà uno degli estremi della connessione con il client e spedisce l'accettazione della richiesta

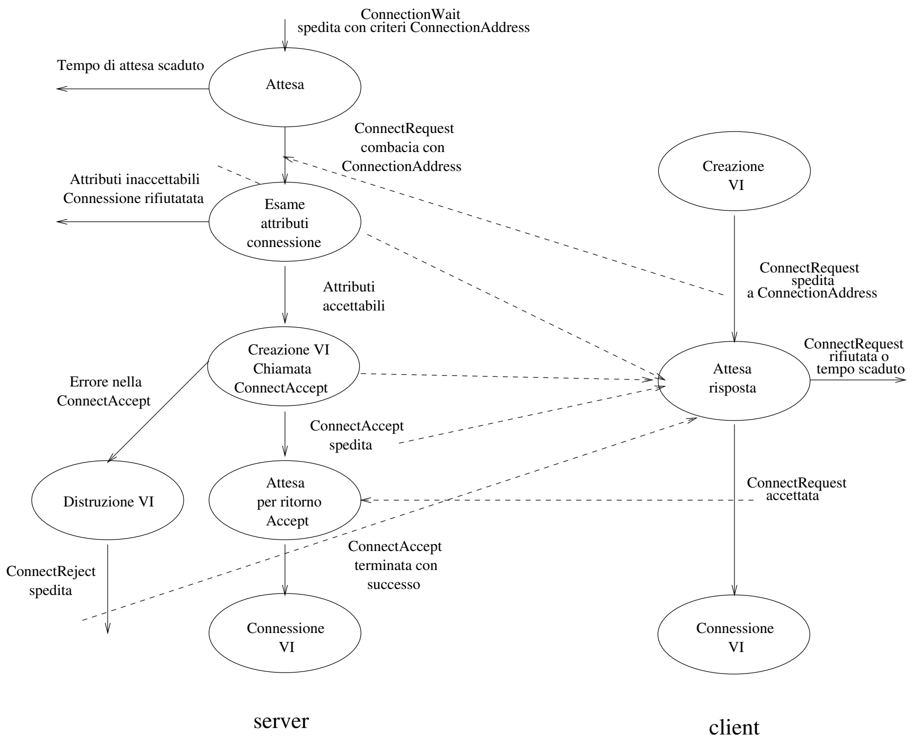

Figura 3.4: Modello client-server: fasi della connessione.

al client; una volta che il client ha terminato la sua richiesta e ne ha dato conferma al server, può essere creata la connessione tra i due VI. Il modello è di tipo bloccante perché il client è costretto ad attendere una risposta dal server alla sua richiesta e può uscire da questo stato solo quando tale richiesta risulta soddisfatta o respinta o quando è scaduto il tempo stabilito per l'attesa della risposta.

Nel seguito le operazioni necessarie alla connessione verranno analizzate in dettaglio. Prima di tutto il VI Consumer del processo server effettua una richiesta di tipo ConnectWait al suo VI Provider. Questa richiesta contiene il valore del discriminator[3] (discrimination portion nell'indirizzo di rete) che rende possibile individuare quali richieste dovranno essere accettate dal VI Consumer del server. A secondo del valore del discriminator il VI consumer sarà in grado di accettare una connessione da qualsiasi locazione remota oppure da un sito remoto ben preciso. Oltre a questo una richiesta di tipo ConnectWait deve anche contenere una struttura dati idonea per l'immagaz-

[3]Si veda in appendice per l'esatto significato di questa variabile.

zinamento delle informazioni del VI remoto che sta richiedendo una connessione, un handle per la gestione della futura connessione (esso sarà restituito nelle operazioni di ConnectAccept e ConnectReject) e una specifica temporale che indica la massima durata della ConnectWait allo scadere della quale si ha il ritorno del controllo al chiamante. Mentre il VI Consumer del server aspetta la connessione, il VI Consumer del client effettua una richiesta di tipo ConnectRequest al suo VI Provider; questa richiesta specifica a quale VI locale deve essere connesso, l'indirizzo (sotto forma di idonea struttura) del VI remoto a cui si vuole collegare, una struttura dati specifica usata per ricevere informazioni riguardanti la VI del server quando la connessione va a buon fine ed un tempo massimo il cui significato è analogo a quello della ConnectWait per il server.

Le richiesta da parte del client si può risolvere in due modi:

- se la connessione con la VI remota fallisce, perché la VI remota non esiste, non è raggiungibile, non è in uno stato idoneo oppure il suo discriminator non combacia con quello del client, il VI Provider inoltra un messaggio di errore al VI Consumer;

- se la VI remota specificata nella richiesta è disponibile, allora la ConnectWait del VI Consumer del server termina correttamente e riceve tutte le informazioni sulla VI del client e un identificatore univoco per la richiesta di connessione che è stata inoltrata.

Il server può a questo punto decidere se accettare la richiesta che gli è stata inoltrata.

Accettazione della richiesta. Se il server decide di accettare la richiesta deve avere a disposizione una VI: per questo scopo il VI Consumer può sfruttare una VI precedentemente creata oppure costruirne una nuova dotata delle caratteristiche necessarie per la connessione: il livello di affidabilità e la qualità del servizio (Quality of Service), la dimensione massima del pacchetto trasferibile sul mezzo fisico (*MTU Maximum Transfer Unit*) della nuova VI devono coincidere con quelli della VI remota. Come passo successivo è necessario che il VI Consumer del server effettui una richiesta di ConnectAccept, sfruttando l'handle della connessione precedentemente creato e specificando la VI locale usata nella connessione, che si deve trovare nello stato *idle* (fig. 3.5) e non sia il punto terminale di un'altra connessione.

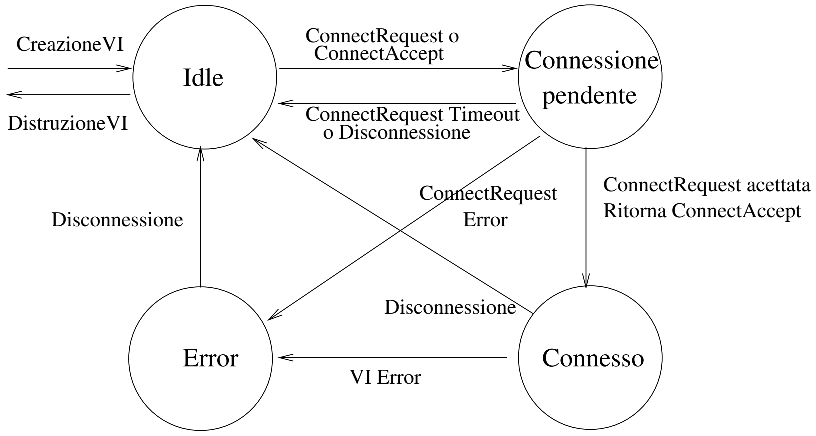

Figura 3.5: Stati di una VI.

A questo punto viene compiuta la verifica sulle caratteristiche della VI locale perché solo se questi combaciano con quelli richiesti dalla VI remota è possibile stabilire una connessione; nel caso in cui l'operazione di confronto sia negativa, la ConnectAccept restituisce un errore. In questo caso, la richiesta di connessione rimane pendente e il VI Consumer del server, se non vuole rifiutare la connessione, deve produrre una nuova richiesta di ConnectAccept che specifichi una VI con degli attributi corretti.

Quando la ConnectAccept ritorna senza produrre alcun messaggio di errore la VI locale, estremo della connessione sul lato del server, passa dallo stato *pendente* a quello *connesso*; analogamente dal lato client, se la ConnectRequest del client risulta accettata, la VI locale del client passa allo stato *connesso*.

Rifiuto della richiesta. Se il server decide di rifiutare la connessione, il VI Consumer deve inoltrare una richiesta di ConnectionReject al suo VI Provider, specificando naturalmente l'handle della connessione creato con la ConnectWait.

In questo caso il VI Consumer del client riceve un opportuno messaggio di errore prodotto dalla ConnectRequest che informa del rifiuto da parte del server di connettersi. Un messaggio di errore analogo viene ricevuto anche quando il tempo per il quale il client è disposto ad aspettare una risposta alla sua richiesta di connessione scade.

In entrambi i casi lo stato della VI passa da *connessione pendente* a *idle*.

Nel seguito di questo paragrafo verranno descritte le funzioni necessarie per questo tipo di connessione.

Istruzioni del Server:

VIP_RETURN **VipConnectWait** (
IN VIP_NIC_HANDLE NicHandle,
IN VIP_NET_ADDRESS ∗LocalAddr,
IN VIP_ULONG Timeout,
OUT VIP_NET_ADDRESS ∗RemoteAddr,
OUT VIP_VI_ATTRIBUTES ∗RemoteViAttribs,
OUT VIP_CONN_HANDLE ∗ConnHandle
)

Questa funzione è utilizzata dal server per porsi in attesa delle richieste di connessione provenienti dall'indirizzo locale di rete ∗LocalAddr per un tempo massimo di attesa pari a Timeout. Se un client presenta una richiesta di connessione che rispetta i parametri espressi nella porzione *discriminator* dell'indirizzo di rete, la connessione viene stabilita e la funzione restituisce in ∗RemoteAddr l'indirizzo di rete da cui è provenuta la richiesta, in ∗RemoteViAttribs gli attributi della VI remota e in ∗ConnHandle lo handle per la connessione necessario per la successiva funzione che il server deve eseguire.

VIP_RETURN **VipConnectAccept** (
IN VIP_CONN_HANDLE ConnHandle,
IN VIP_VI_HANDLE NicHandle
)

Una volta ricevuta la richiesta di connessione il server può accettarla con la funzione *VipConnectAccept*. Il VI Provider si fa carico di verificare la corrispondenza tra gli attributi dei due estremi della connessione, provvedendo eventualmente alla modifica degli attributi del VI locale per poi ritentare l'accept.

VIP_RETURN **VipConnectReject** (
    IN VIP_CONN_HANDLE ConnHandle
)

Con questa funzione il server può rifiutare una richiesta di connessione informandone il client.

Istruzioni del client:

VIP_RETURN **VipConnectRequest** (
    IN VIP_VI_HANDLE ViHandle,
    IN VIP_NET_ADDRESS *LocalAddr,
    IN VIP_NET_ADDRESS *RemoteAddr,
    IN VIP_ULONG Timeout,
    OUT VIP_VI_ATTRIBUTES *RemoteViAttribs,
)

Il lato client (con indirizzo *LocalAddr) utilizza questa funzione per richiedere una connessione con il punto remoto avente l'indirizzo pari a *RemoteAddr. Se la richiesta viene accettata, gli attributi del punto remoto vengono consegnati al chiamante permettendogli così di verificare se le operazioni di RDMA possono essere eseguite sulla connessione appena creata.

Fase di disconnessione

Per disconnettere una VI è necessario che il VI Consumer effettui una richiesta di disconnessione al suo VI Provider, producendo in questo modo una caduta unilaterale della connessione. Quando l'estremo locale di connessione è disconnesso, si interrompe l'elaborazione dei descrittori attualmente postati e tutti quelli non terminati vengono completati con un errore. Il VI Consumer della VI disconnessa da remoto non riceve alcun segnale di errore indicante la disconnessione da parte del suo VI Provider, ma lo stato della VI passa a *Errore* non appena tenti di eseguire una operazione di invio di dati.

VIP_RETURN **VipDisconnect** (
    IN VIP_VI_HANDLE ViHandle
)

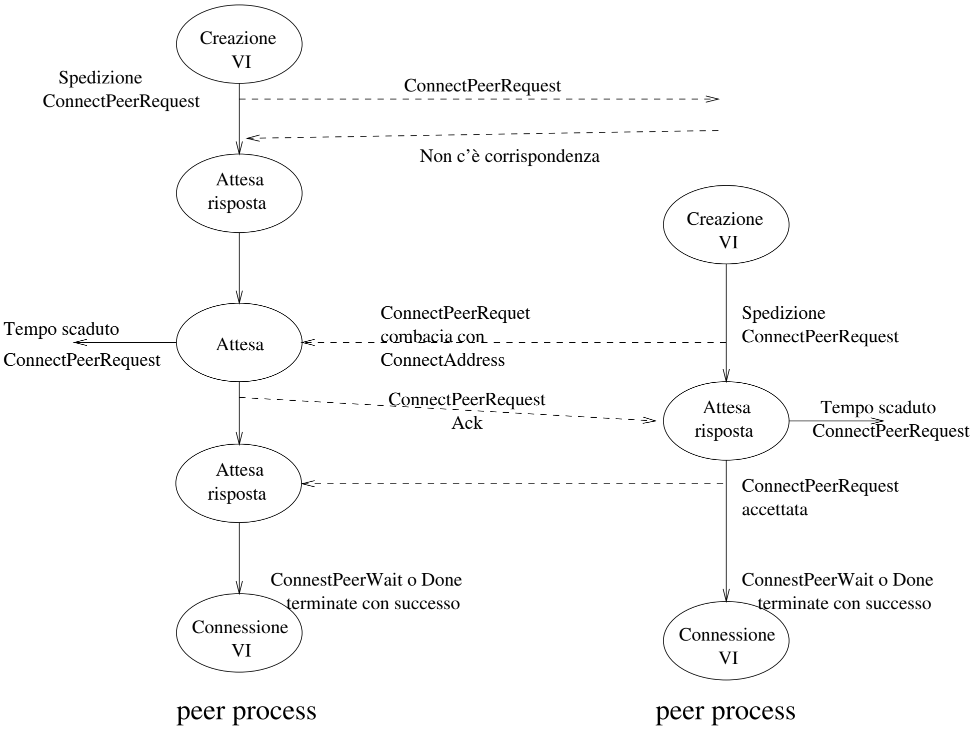

Figura 3.6: Modello peer-to-peer: connessione.

La disconnessione avviene chiamando la funzione VipDisconnect. Entrambi i VI Consumer interessati nella connessione dovranno chiamare questa funzione per portarsi dallo stato di *Errore* o di *Connessione* allo stato di *Idle*.

3.6.2 Modello peer-to-peer

Il modello di connessione peer-to-peer realizza una semantica sia di tipo bloccante che di tipo non bloccante, secondo il diagramma degli stati di figura 3.6.

La connessione peer-to-peer inizia con la richiesta da parte di uno dei due VI Consumer al VI Provider. Una volta inizializzata la richiesta di connessione, il controllo ritorna al chiamante che può decidere se controllare lo stato della richiesta di connessione con un test non bloccante oppure porsi in attesa bloccante fino a che la fase di inizializzazione della connessione ha termine. Da questo punto in poi il modello peer-to-peer è identico al modello client-server. Qualunque sia la strada scelta, bloccante o non bloccante, se la connessione viene stabilita con successo le informazioni del VI remoto della connessione vengono restituite al chiamante.

Una differenza fondamentale tra questo modello e quello client-server riguarda i parametri che vengono confrontati dai due elementi che si vogliono connettere per decidere se accettare o meno la connessione; infatti nel peer-to-peer è necessaria una corrispondenza non solo tra la *discrimination portion* ma anche tra la *host portion* contenuti nell'indirizzo di rete così come specificato nelle funzioni chiamanti.

Esaminiamo ora le chiamate necessarie per questo tipo di connessione.

VIP_RETURN **VipConnectPeerRequest** (
IN VIP_VI_HANDLE ViHandle,
IN VIP_NET_ADDRESS ∗LocalAddr,
IN VIP_NET_ADDRESS ∗RemoteAddr,
IN VIP_ULONG Timeout
)

VipConnectPeerRequest richiede la connessione tra una VI locale ed una VI remota. Se la connessione viene stabilita con successo vengono ricevuti gli attributi della VI remota.

VIP_RETURN **VipConnectPeerDone** (
IN VIP_VI_HANDLE ViHandle,
OUT VIP_VI_ATTRIBUTES ∗RemoteViAttribs,
)

VipConnectPeerDone consente di testare il risultato di una VipConnectPeerRequest precedentemente effettuata su un particolare *ViHandle* senza bloccare il processo chiamante. Se la connessione è stabilita con successo, vengono fatti pervenire al chiamante gli attributi del punto remoto in *RemoteViAttribs*; questi attributi consentono di stabilire se e quali operazioni di RDMA possano essere effettuate sulla connessione appena creata.

VIP_RETURN **VipConnectPeerWait** (
IN VIP_VI_HANDLE ViHandle,
OUT VIP_VI_ATTRIBUTES ∗RemoteViAttribs,
)

Questa funzione testa il risultato di una VipConnectPeerRequest precedentemente effettuata su un particolare *ViHandle* bloccando il processo chiamante finchè il risultato della connessione non è disponibile. Come per la funzione *VipConnectPeerDone* se la connessione è stabilita con successo, vengono fatti pervenire al chiamante gli attributi del punto remoto (in *RemoteViAttribs*).

La fase di disconnessione del modello peer-to-peer è identica al modello client-server.

3.7 Modelli di trasferimento dei dati

Dopo che due VI sono state connesse un VI Consumer può accedere direttamente all'interfaccia di rete attraverso una Virtual Interface per effettuare le operazioni di trasferimento dati. Una Virtual Interface è formata da una coppia di Work Queue: una coda per le operazioni di spedizione, la Send Queue, e una coda, la Receive Queue, per le operazioni di ricezione dei dati. Ogni coda è costituita da una lista concatenata di descrittori di lunghezza variabile[4]. Il VI Consumer inoltra le richieste di spedizione e di ricezione inserendo, ovvero *postando* sulle Work Queue opportuni descrittori. A sua volta il VI Provider processa in modo asincrono i descrittori in precedenza postati sulle code e aggiorna il loro stato attraverso alcuni campi[5].

Quando l'operazione di trasferimento è stata completata il VI Consumer rimuove il descrittore dalla coda relativa e per poterlo riutilizzare nelle richieste successive. Una estensione delle strutture per la comunicazione con VIA consente al VI Consumer di utilizzare una coda aggiuntiva, la Completion Queue, per verificare il completamento dei descrittori appartenenti a più Work Queue dello stesso tipo ma di VI diverse.

VIA consente di utilizzare due modelli di trasferimento dei dati:

- Send/Receive;
- Remote Direct Memory Access (RDMA).

[4]In appendice si riportano i dettagli di queste e altre strutture dati.

[5]Diremo che un descrittore è completato se il suo *done bit* è posto a uno.

3.7.1 Send/Receive

Questo modello si basa sui meccanismi solitamente usati per scambiare dati tra due entità comunicanti: una esegue operazioni di invio, spedendo dati in suo possesso, mentre l'altra riceve in una zona di memoria in suo possesso i dati spediti.

Infatti un VI Consumer deve sempre specificare una sola locazione dei dati: nel caso in cui desideri mandare dati, il processo specifica la regione di memoria che contiene i dati da spedire; nel caso voglia ricevere, il processo indica la regione di memoria dove i dati in arrivo saranno immagazzinati.

Ciascuna operazione di ricezione o spedizione di dati, avviene mediante il posting sulla coda corretta di un descrittore che contiene, tra le sue informazioni, l'indirizzo virtuale della regione di memoria da inviare o destinata a contenere i dati in arrivo assieme agli handle creati durante la fase di registrazione della memoria.

Un aspetto importante di questo modello di trasferimento dei dati riguarda la sincronizzazione tra i processi: una volta che due VI siano connesse, se uno dei due processi desidera inviare un dato postando un descrittore sulla propria Send Queue, si deve preventivamente assicurare che sulla Receive Queue del processo a lui connesso sia postato almeno un descrittore pronto a ricevere i dati; inoltre il buffer il cui indirizzo è indicato nel descrittore destinato a ricevere i dati deve essere di dimensione sufficiente a contenere tutti i dati in arrivo. Se queste condizioni non sono verificate, cioé il descrittore sulla cima della Receive Queue del processo ricevente non ha una dimensione sufficiente per i dati in arrivo, o la coda è vuota, viene segnalato un errore denominato *asyncronous error*; in tal caso se la connessione è di tipo affidabile[6] essa termina e le VI entrano in uno stato di *errore* dal quale l'unica via di uscita è la disconnessione della VI (figura 3.5); se la connessione non è affidabile le VI possono ancora essere usate per le comunicazioni.

Questo modello fornisce ai VI Consumer su entrambi i lati della comunicazione la notifica del completamento del descrittore: il descrittore usato per spedire dati è completato una volta che i dati da mandare siano usciti dalla interfaccia di rete, mentre un descrittore su una Receive Queue è completato quando il dato è disponibile.

Il modello Send/Receive quindi richiede stretta sincronizzazione prima di potere effettuare una operazione di invio di dati e fornisce una sincronizza-

[6] Si veda il paragrafo relativo ai livelli di affidabilità.

zione sul completamento di una operazione.

La modalità di verifica del completamento di una operazione consente una semantica sia bloccante sia non bloccante (polling); nel caso bloccante una funzione controlla se vi sia un descrittore completato sulla cima della coda: in tal caso esso viene rimosso dalla coda e l'indirizzo del descrittore è ritornato al processo chiamante, altrimenti la funzione blocca il processo chiamante fino a quando il descrittore in cima alla coda non è completato, o fino a quando non è trascorso il tempo massimo specificato alla chiamata della funzione. Nel caso non bloccante una funzione, diversa dalla precedente, controlla ancora se il descrittore sulla cima della coda sia completato: se lo è esso viene rimosso dalla coda e l'indirizzo del descrittore è ritornato al processo chiamante, altrimenti la funzione restituisce un codice di errore e la cima della coda non viene modificata.

Nel seguito verranno analizzate in dettaglio le funzioni per le operazioni di Send e quelle per le corrispondenti Receive.

VIP_RETURN **VipPostSend** (
  IN VIP_VI_HANDLE ViHandle,
  IN VIP_DESCRIPTOR *DescriptorPtr,
  IN VIP_MEM_HANDLE MemoryHandle
)

VipPostSend aggiunge il descrittore puntato da *DescriptorPtr* sulla Send Queue della VI specificata da *ViHandle*, avvisa il VI NIC che un nuovo descrittore è stato postato sulla Send Queue attraverso la Doorbell, e ritorna immediatamente al processo chiamante. *MemoryHandle* contiene l'handle del descrittore ottenuto al momento della registrazione del descrittore.

VIP_RETURN **VipSendDone** (
  IN VIP_VI_HANDLE ViHandle,
  OUT VIP_DESCRIPTOR **DescriptorPtr
)

VipSendDone controlla che il descrittore in cima alla Send Queue della VI specificata da *ViHandle* sia stato completato. Se il descrittore è completato viene rimosso dalla Send Queue ed il suo indirizzo è ritornato dalla funzione tramite la variabile *DescriptorPtr*, altrimenti se l'operazione di completamento è terminata in modo scorretto o se la Send Queue è vuota la funzione ritorna NULL nel *DescriptorPtr*. VipSendDone è una funzione non bloccante.

VIP_RETURN **VipSendWait** (
    IN VIP_VI_HANDLE ViHandle;
    IN VIP_ULONG TimeOut;
    OUT VIP_DESCRIPTOR **DescriptorPtr;
}

VipSendWait ha un funzionamento del tutto analogo alla precedente VipSendDone, con la differenza che essa realizza una semantica bloccante; quindi se in descrittore in cima alla Send Queue non è stato completato, VipSendWait blocca il processo chiamante fino a quando il descrittore sia stato completato, o sia trascorso il tempo massimo specificato in millisecondi in *TimeOut*. Se non si desidera alcun timeout, questo parametro deve essere posto a VIP_INFINITE. Questa funzione non può essere usata se la Send Queue è stata associata ad una Completion Queue, in quanto la sincronizzazione deve avvenire sulla CQ stessa.

Lo strumento per mezzo del quale ogni comunicazione in VIA viene effettuata è il descrittore; per comprendere in quale modo VIA elimini il kernel del sistema operativo dal percorso critico delle comunicazioni è necessario seguire il cammino che un descrittore compie all'interno dei componenti di VIA. Durante la comunicazione la proprietà del descrittore passa dal VI Application alla VI Library (VIPL) e quindi al VI NIC, senza transitare attraverso il Kenel Agent, per poi compiere il percorso inverso[7].

VI Application. (figura 3.7)

La VI Application, cioé il processo utente che faccia uso di VIA, inizializza alcuni campi della prima parte del descrittore, il *Control Segment*, che contiene le informazioni di stato del descrittore: essa modifica il campo per il tipo di operazione richiesta (*control*), per il numero complessivo di parti di cui il descrittore si compone (*segment count*), per la dimensione totale in byte dei dati inviati o dei buffer per la ricezione, e per altre informazioni accessorie. Inoltre la VI Application deve

[7] In appendice sono riportati con esattezza i formati delle strutture dati coinvolte, in particolare il descrittore.

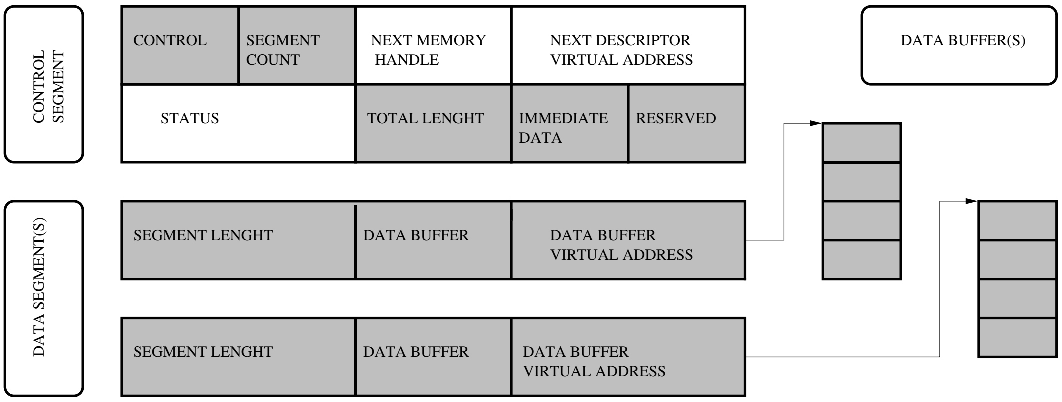

Figura 3.7: Descrittore inizializzato dalla VI Application. I campi ombreggiati sono quelli modifcati in ciascuna fase.

specificare gli indirizzi dei dati da spedire o dei buffer destinati a contenere i dati in arrivo: per questa funzione un descrittore contiene anche uno o più *Data Segment* in cui la VI Application indica la dimensione totale dei buffer di dati, il loro indirizzo virtuale e l'handle creato durante la loro registrazione. Il descrittore passa dalla VI Application alla VI Library quando è invocata una delle due funzioni VipPostSend e VipPostReceive che postano un descrittore su di una coda.

VIPL. (figura 3.8)

La VI Library, si occupa di inserire il descrittore sulla coda opportuna, aggiornando i campi del descrittore relativi al *next descriptor*. La VIPL inoltre deve settare il campo *status* a zero. A questo punto la VIPL modifica il Doorbell per indicare al VI NIC che un nuovo descrittore da processare è pronto.

VI NIC. (figura 3.9)

Il VI NIC processa in modo asincrono il descrittore: esegue le operazioni di trasferimento dei dati richieste, modificando i contenuti dei buffer indicati nel data segment; lo stato del descrittore viene quindi aggiornato e quando è il *done bit* è posto a uno, cioé il descrittore è completato, esso è passato di nuovo alla VIPL.

VIPL. (figura 3.10)

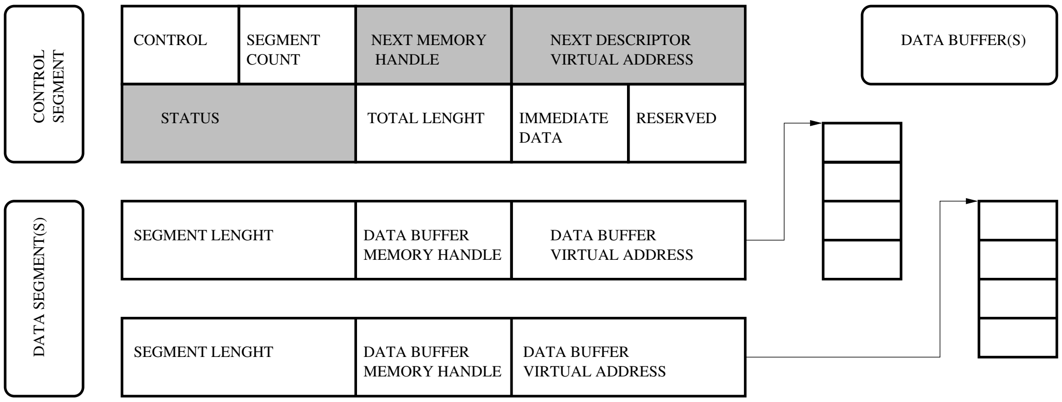

Figura 3.8: Descrittore modificato dalla VIPL.

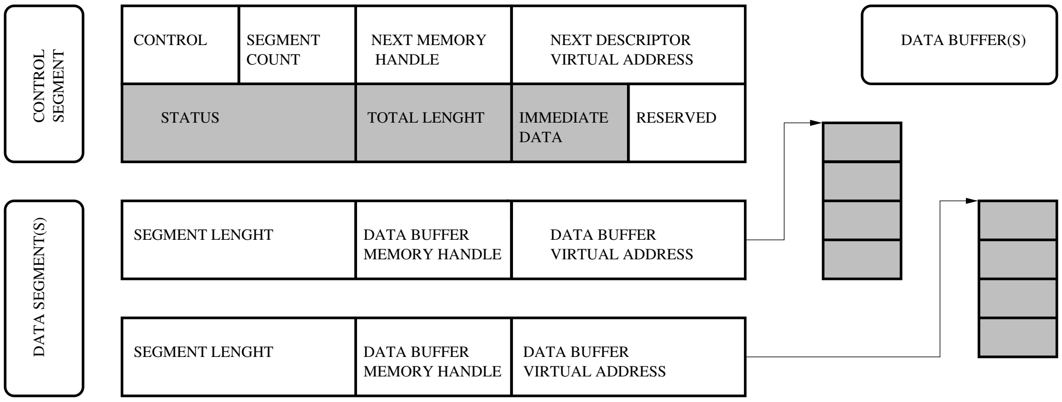

Figura 3.9: Descrittore processato dal VI NIC.

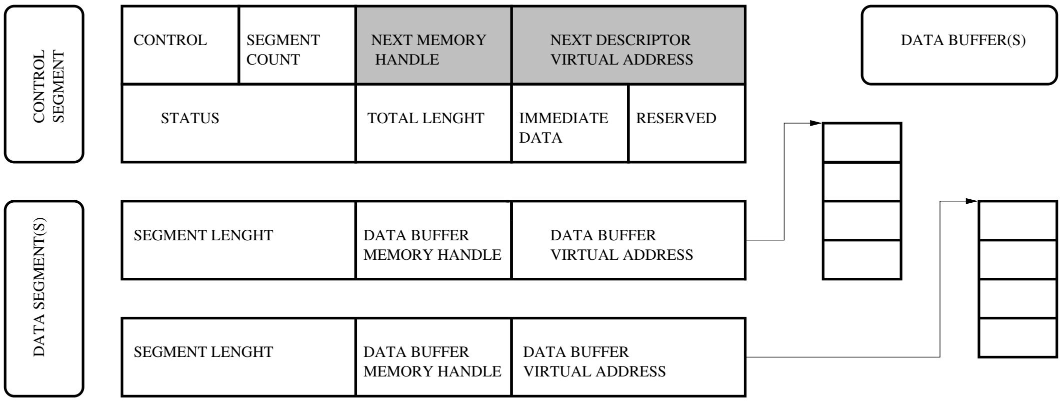

Figura 3.10: Descrittore restituito alla VIPL.

Quando la VI Application chiama VipSendDone o VipRecvDone, la VI Library controlla *done bit*, toglie dalla coda il descrittore se esso è completato e ritorna alla VI Application il descrittore come modificato nelle fasi precedenti. La VI Application non può fare assunzioni sul contenuto dei campi*Next Handle* e*Next Descriptor Virtual Address* del *Control Segment*.

VI Application. La VI Application diventa nuovamente proprietaria del descrittore quando il puntatore ad esso è ritornato dalle funzioni VipSendDone o VipRecvDone.

3.7.2 Remote Direct Memory Access

Nel modello RDMA il processo che intende trasferire i dati, specifica sia l'indirizzo della zona di memoria sorgente che della zona di memoria destinazione; da questa caratteristica deriva il nome di accesso diretto da parte di un processo alla memoria di un secondo processo che si può trovare anche su un nodo diverso da quello del primo processo.

Questo modello di trasferimento dei dati differisce dal precedente anche per quanto riguarda l'aspetto della sincronizzazione tra i processi comunicanti: il processo che inizi una operazione di RDMA non ha bisogno di sapere se l'altro processo abbia postato alcun descrittore sulla propria Receive Queue, in quanto le operazioni di RDMA non consumano descrittori in ricezione; questo però non consente neppure di stabilire una sincronizzazione successiva al completamento delle operazioni di tipo RDMA: nè il processo mittente nè il ricevente hanno notifica del completamento dell'operazione. È quindi un modello di comunicazione asincrono.

Le richieste per entrambi i tipi di operazioni RDMA, Write e Read, sono sempre realizzate tramite descrittori, in cui il campo indicante il tipo di operazione riporta RDMA, postati sulla Send Queue di una VI in stato di connesione.

Un aspetto importante delle operazioni RDMA è la necessità che il processo che effettui questo tipo di comunicazioni sia già stato informato dal VI Consumer remoto su quale sia l'indirizzo virtuale della memoria del buffer di destinazione del processo remoto e che il buffer stesso sia già stato abilitato per questo tipo di operazione attraverso una registrazione di memoria durante la quale il Kernel Agent fornisce al VI Consumer un handle associato univocamente alla memoria registrata. Prima di ogni operazione di RDMA quindi i processi devono scambiarsi tramite Send e Receive gli indirizzi virtuali delle regioni di memoria che intendono mettere a disposizione per RDMA.

Esistono due tipi di operazioni RDMA: RDMA Write e RDMA Read.

RDMA Write

Per questa operazione il VI Consumer specifica la sorgente dei dati in una delle sue regioni di memoria locali precedentemente registrata, e la destinazione dei dati dentro ad una regione di memoria remota.

Un descrittore può contenere un solo indirizzo di una regione di memoria remota, mentre il VI Consumer può specificare più regioni di memoria locali in ogni descrittore per RDMA Write: in questo modo sono possibili operazioni di *gather* in cui i dati da inviare tramite un RDMA Write possono essere specificati come un insieme di buffer anche non contigui, mentre la destinazione deve essere una singola, contigua regione di memoria registrata.

Si diceva che una operazione di RDMA Write non consuma alcun descrittore sulla Receive Queue del processo destinatario. Esiste una importante eccezione, RDMA Write con Immediate Data: questa operazione ha un codice di controllo da inserire nel descrittore diverso rispetto alla RDMA Write senza Immediate Data e consente di riportarsi al modello di sincronizzazione di Send/Receive. Infatti prima di potere inviare dati tramite RDMA Write con Immediate Data, un processo deve assicurarsi che un descrittore sia già stato postato sulla Receive Queue del processo destinatario; tale descrittore viene completato una volta che il trasferimento dei dati abbia avuto termine, fornendo in tal modo la notifica al processo destinatario dell'avvenuta scrittura nella propria memoria.

RDMA Read

Specularmente a quanto accade per la RDMA Write, nella RDMA Read, il VI Consumer specifica la sorgente dei dati da leggere sul nodo remoto, e la destinazione, cioé il buffer predisposto a contenere i dati letti sul nodo remoto, in una regione di memoria registrata localmente. La sorgente di dati deve essere un buffer singolo, contiguo identificato da un unico indirizzo virtuale, mentre la destinazione può essere specificata come un insieme di buffer diversi e non contigui; sono quindi possibili operazioni di *scatter*, in cui una singola informazione letta in remoto viene spezzettata in tanti buffer locali.

Anche RDMA Read impone che prima del trasferimento dati, il VI Consumer sul nodo remoto abbia informato il processo, che dovrebbe iniziare l'operazione, sulla locazione del buffer sorgente.

Un'operazione RDMA Read non consuma alcun descrittore sulle code del nodo remoto, ma necessita solo di un descrittore da postare sulla Send Queue della VI del processo che voglia leggere i dati sul processo remoto. Con RDMA Read non è consentito l'uso di Immediate Data, per cui essa è sempre una operazione asincrona.

Mentre l'operazione di RDMA Write è una caratteristica richiesta dalle specifiche dall'architettura VIA, il supporto di operazioni RDMA Read è una caratteristica opzionale. Un VI Provider dovrebbe fornire un meccanismo con il quale un VI Consumer può determinare se sono supportate le operazioni del tipo RDMA Read.

3.8 Completion Queue

Le Completion Queue (CQ) consentono di fondere in una singola coda gli avvisi di completamento di descrittori appartenenti a più Work Queue dello stesso tipo. L'associazione della Work Queue alla Completion Queue avviene nel momento della creazione della VI. Ogni Work Queue può essere associata ad una sola Completion Queue. Una volta che una Work Queue sia stata

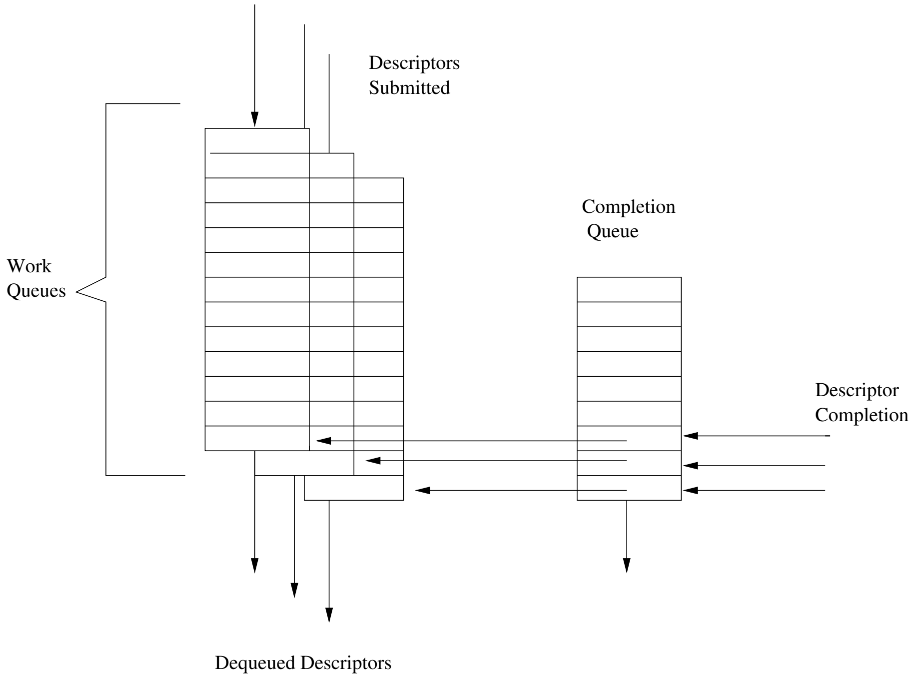

Figura 3.11: Completion Queue.

associata a una Completion Queue ogni verifica di completamento di descrittori postati sulla Work Queue deve aver luogo sulla Completion Queue e non sulla Work Queue, sia nel caso di semantica bloccante che non bloccante (figura 3.11).

Per creare una Completion Queue il VI Provider alloca una zona di memoria nello spazio degli indirizzi virtuali del VI Consumer riservandola per la Completion Queue ed informando il VI NIC del suo indirizzo. Se la quantità di memoria richiesta non può essere allocata la CQ non viene creata e viene segnalato un errore.

La dimensione di una Completion Queue, indicata al momento della sua creazione, può essere modificata dinamicamente dal VI Provider.

Una Completion Queue può essere distrutta solo quando tutte le VI con code associate sono state distrutte. Se un processo viene terminato dal sistema operativo il VI Provider si deve fare carico della distruzione della CQ ancora associata a quel processo.

In seguito si analizzano alcune delle funzioni che VIA mette a disposizione per gestire le Completion Queue.

VIP_RETURN **VipCreateCQ** (
    IN VIP_NIC_HANDLE NicHandle,
    IN VIP_ULONG EntryCount,
    OUT VIP_CQ_HANDLE *CQHandle
)

VipCreateCQ crea una nuova Completion Queue con un numero di ingressi pari a *EntryCount*, associata al NIC *NicHandle* e ritorna l'identificatore in *CQHandle*.

VIP_RETURN **VipDestroyCQ** (
    IN VIP_CQ_HANDLE CQHandle
)

VipDestroyCQ distrugge la Completion Queue *CQHandle*. Se sono ancora presenti delle Work Queue associate alla Completion Queue, quest'ultima non è distrutta e la funzione ritorna un errore.

VIP_RETURN **VipResizeCQ** (
    IN VIP_CQ_HANDLE CQHandle,
    IN VIP_ULONG EntryCount
)

VipResizeCQ modifica la dimensione della Completion Queue *CQHandle* specificando in *EntryCount* il nuovo numero di ingressi.

Le funzioni seguenti sono usate per avere la notifica del completamento di un descrittore:

VIP_RETURN **VipCQDone** (
    IN VIP_CQ_HANDLE CQHandle,
    OUT VIP_VI_HANDLE *VIHandle,
    OUT VIP_BOOLEAN *RecvQueue
)

VipCQDone esmina la Completion Queue *CQHandle* per verificare se esista un descrittore completato. In caso affermativo ritorna l'identificatore della VI associata in *VIHandle* e un flag per indicare se il descrittore risiede sulla Receive o sulla Send Queue (VIP_TRUE allora la Completion Queue è associata alla Receive Queue, VIP_FALSE altrimenti). Questa funzione non è bloccante.

Essa non ha il compito di togliere dalla Work Queue il descrittore eventualmente completato, ma solo di rimuovere l'entry corrispondente dalla Completion Queue; successivamente il processo chiamante deve invocare le funzioni appropriate, VipSendDone o VipRecvDone, per togliere dalla Work Queue il descrittore completato.

VIP_RETURN **VipCQWait** (
IN VIP_CQ_HANDLE CQHandle,
IN VIP_ULONG TimeOut,
OUT VIP_VI_HANDLE ∗VIHandle,
OUT VIP_BOOLEAN ∗RecvQueue
)

VipCQWait, come VipCQDone, cerca sulla Completion Queue *CQHandle* una entry per un descrittore completato. In caso affermativo ritorna l'identificatore di VI in *VIHandle* ed il flag per indicare se il descrittore risiede sulla Receive o sulla Send Queue. Se non è stata trovata nessuna entry completata, il processo chiamante si blocca fino a quando non viene completato un descrittore, o il tempo massimo *timeout* si esaurisce. Come per la funzione precedente anche questa non rimuove il descrittore dalla sua Work Queue, ma solo la entry nella Completion Queue.

3.9 Livelli di affidabilità

Virtual Interface Architecture supporta tre diversi livelli d'affidabilità [23]:

- Unreliable Delivery.
- Reliable Delivery.
- Reliable Reception.

Fra questi il livello Unreliable Delivery è sempre richiesto, mentre gli altri due sono opzionali, anche se sono fortemente raccomandati poichè permettono la costruzione al di sopra di VIA di protocolli di comunicazione più semplici rispetto al caso di Unreliable Delivery.

I livelli d'affidabilità sono attribuiti a un'istanza VI nella fase di creazione e sono estremamente importanti perché possono essere connesse solo VI che abbiano lo stesso livello d'affidabilità. Il livello di affidabilità di una VI può essere modificato con la funzione *VipSetViAttributes*.

3.9.1 Unreliable Delivery

Questo livello di affidabilità garantisce che un'operazione di Send oppure di RDMA Write sia consegnata al più una sola volta all'istanza VI ricevente, cioé quindi che per effetto di una Send sulla Receive Queue del processo destinatario venga consumato uno o nessun descrittore. Di conseguenza una Send o un RDMA Write possono non giungere mai a destinazione risultando persi.

Eventuali errori di corruzione dei dati vengono rilevati ma questo non comporta la ritrasmissione dei dati: in questo caso una operazione di Send o di RDMA Write con Immediate Data consumano un descrittore sulla Receive Queue del processo destinatario completato con un errore.

Non vi è alcuna garanzia che un dato qualsiasi sia ricevuto e nemmeno che l'ordine con il quale i dati sono stati spediti corrisponda a quello di ricezione: è compito del VI Provider di costruire un protocollo che consenta la ricezione ordinata, se necessaria.

La rivelazione di un errore non provoca la transizione della VI in stato di *errore*, ma solo il dato corrotto viene perso, e le comunicazioni possono continuare.

L'operazione di RDMA Read non è supportata da una VI Unreliable: richiedere questo tipo d'operazione causa il completamento con errore del descrittore.

3.9.2 Reliable Delivery

Questo livello di affidabilità garantisce che tutti i dati trasferiti arrivino a destinazione esattamente una sola volta, integri, nell'ordine in cui sono stati spediti in assenza di errori. Ogni qualvolta invece si verifichi un trasferimento perso, corrotto oppure l'ordinamento dei dati ricevuti non è corretto il VI Provider inoltra un errore al VI Consumer. Altre situazioni che provocano errore sono la mancanza o la insufficiente dimensione del descrittore per la ricezione di una Send o RDMA Write con Immediate Data.

Proprietà	Unreliable	Reliable Delivery	Reliable Reception
Rilevati dati corrotti	Sì	Sì	Sì
Dati consegnati al più una volta	Sì	Sì	Sì
Dati consegnati esattamente una volta	No	Sì	Sì
Dati consegnati in ordine	No	Sì	Sì
Rilevata la perdita di dati	No	Sì	Sì
Connessione caduta per un errore	No	Sì	Sì
Supporto per RDMA Read	No	Opzionale	Opzionale
Supporto per RDMA Write	Sì	Sì	Sì

Tabella 3.1: Garanzie offerte dai livelli di affidabilità.

Quando viene rilevato un errore la VI transita in uno stato di *errore*, la connessione cade e tutti i descrittori in precedenza postati sulle code sono completati con uno stato d'errore.

Nella Reliable Delivery i descrittori usati per una Send o una RDMA Write sono considerati completati quando i dati associati ad essi sono trasmessi sulla rete mentre un descrittore per RDMA Read è completato con successo non appena i dati richiesti sono stati scritti sul buffer di destinazione.

In questo caso quindi un descrittore usato per inviare dati viene completato con un errore solo se l'errore si verifica sul processo mittente, ad esempio un errore di protezione della memoria locale; quando invece l'errore si manifesta dopo che un descrittore per Send o RDMA Write con Immediate Data è stato completato, cioé il trasferimento dei dati è stato iniziato ma non concluso, questo viene notificato al VI Consumer in modo asincrono.

3.9.3 Reliable Reception

È il livello che garantisce la maggiore affidabilità. Questo livello d'affidabilità offre le stesse garanzie del livello Reliable Delivery ma, in aggiunta a queste, consente che un descrittore sia indicato come completato con successo solo quando i dati sono stati consegnati nella locazione di memoria a destinazione.

Se avviene un errore che non permette la trasmissione corretta dei dati, perché non in ordine o non integri, questo è indicato nello stato del descrittore; in tal caso il VI Provider garantisce che nessun descrittore successivo a quello che ha causato l'errore sia processato.

Come nel livello Reliable Delivery, ogni errore è responsabile della rottura della connessione, e fa transitare l'istanza VI nello stato di *errore* e completa con insuccesso tutti i descrittori precedentemente postati.

3.10 Richiesta di informazioni

L'interfaccia della VI Architecture fornisce alcune funzioni per l'acquisizione e la modifica dei valori degli attributi di alcuni componenti del sistema. In caso di errore durante l'operazione di modifica, tutti gli attributi rimangono inalterati.

VIP_RETURN **VipQueryNic** (
    IN VIP_NIC_HANDLE NicHandle,

OUT VIP_NIC_ATTRIBUTES *NicAttribs
)

Fornisce gli attributi del NIC individuato da *NicHandle* e li memorizza in *NicAttribs*.

VIP_RETURN **VipSetViAttributes** (
IN VIP_VI_HANDLE ViHandle,
OUT VIP_VI_ATTRIBUTES *VIAttribs
)

Permette di modificare gli attributi della VI *ViHandle*, secondo i valori contenuti in *VIAttribs*. Quando la VI ViHandle si trova nello stato di Idle restituisce un errore.

VIP_RETURN **VipQueryVi** (
IN VIP_VI_HANDLE ViHandle,
OUT VIP_VI_STATE *State,
OUT VIP_VI_ATTRIBUTES *ViAttribs,
OUT VIP_BOOLEAN *ViSendQEmpy,
OUT VIP_BOOLEAN *ViRecvQEmpy,
)

Restituisce gli attributi, lo stato e la situazione delle code del VI *ViHandle*.

VIP_RETURN **VipSetMemAttributes** (
IN VIP_VI_HANDLE ViHandle,
IN VIP_PVOID Address,
IN VIP_MEM_HANDLE MemHandle,
VIP_MEM_ATTRIBUTES *MemAttribs
)

Modifica gli attributi della zona di memoria il cui indirizzo di base della regione è contenuto in *Address* e l'identificatore in *MemHandle* secondo i valori specificati in *MemAttribs*.

VIP_RETURN **VipQueryMem** (
IN VIP_VI_HANDLE NicHandle,

IN VIP_PVOID Address,
IN VIP_MEM_HANDLE MemHandle,
VIP_MEM_ATTRIBUTES *MemAttribs
}

Restituisce in *MemAttrbs* gli attributi della regione di memoria individuata da *Address* e *MemHandle*.

VIP_RETURN **VipQuerySystemManagementInfo** (
IN VIP_NIC_HANDLE NicHandle,
IN VIP_ULONG InfoType,
OUT VIP_PVOID SysManInfo
)

Restituisce le proprietà della gestione del sistema del NIC *NicHandle*.

3.11 Name service

In un sistema distribuito, per rendere possibile la comunicazione tra i vari elementi, risulta fondamentale disporre di un servizio di *name service* per individuare univocamente nomi e indirizzi nel dominio di utilizzo di VIA, cioé reti locali o meglio System Area Network.

Per immagazzinare le informazioni che le applicazioni richiedono attraverso le funzioni di traduzione di nomi con indirizzi e viceversa, VIA può utilizzare diverse fonti per il name service come quelli di rete tipo DNS, NIS o basati su file come in Unix con i file /etc/hosts; tale scelta compete a chi si incarica di realizzare VIA su uno specifico sistema operativo. Le operazioni invece che l'utente svolge sono indipendenti dalla loro realizzazione e si dividono in due categorie:

- traduzione di nomi sotto forma di stringhe in indirizzi di rete;
- traduzione di indirizzi in un host name.

Le funzioni fondamentali per la risoluzioni dei nomi e degli indirizzi sono le seguenti:

VIP_RETURN **VipNSInit** (
    IN VIP_NIC_HANDLE NicHandle,
    IN VIP_PVOID NSInitInfo
)

Inizializza il name service collegato al *NicHandle* secondo i parametri specificati in *NSInitInfo*.

VIP_RETURN **VipNSGetHostByName** (
    IN VIP_NIC_HANDLE NicHandle,
    IN VIP_CHAR *Name,
    IN OUT VIP_NET_ADDRESS *Address,
    IN VIP_ULONG NameIndex
)

Traduce per il *NicHandle* i nome contenuti in *Name* in forma di stringa in indirizzi di rete, restituendoli tramite la variabile *Address*. *NameIndex* consente di associare più indirizzi di NIC con lo stesso nome su di una System Area Network. Per ogni nuovo nome è necessario richiamare questa funzione specificando un valore diverso per il paramentro *NameIndex*.

VIP_RETURN **VipNSGetHostByAddr** (
    IN VIP_NIC_HANDLE NicHandle,
    IN VIP_NET_ADDRESS *Address,
    OUT VIP_CHAR *Name,
    IN OUT VIP_ULONG NameLen
)

È la funzione duale della precedente. A partire da *Address* restituisce in *Name* il nome a lui associato dal name service. *NameLen* specifica la lunghezza dell'array contenente il nome ed è settato dal chiamante

VIP_RETURN **VipNSShutdown** (
    IN VIP_NIC_HANDLE NicHandle
)

Questa funzione permette di specificare che non verrà effettuata alcuna nuova richiesta al name service.

3.12 Gestione degli errori

Gli errori che si possono verificare nell'uso di VIA possono essere classificati in due categorie:

- errori sincroni,
- errori asincroni.

Gli errori sincroni sono quelli che vengono riportati al VI Consumer quando è in corso di esecuzione un'operazione oppure al completamento di un descrittore.

Gli errori asincroni sono quelli che non possono essere riportati al VI Consumer per mezzo di un descrittore; alcuni tipici esempi sono:

- errore nel postare un descrittore;
- interruzione della connessione;
- Receive Queue vuota;
- VI Overrun: il VI Consumer tenta di postare troppi descrittori su di una Work Queue;
- RDMA Write Protection Error: una RDMA Write sta tentando di scrivere sul processo remoto in una zona di memoria non consentita, provocando un errore di protezione;
- RDMA Write Data Error: compromissione dei dati in arrivo da parte di una operazione di RDMA Write;
- RDMA Read Protection Error: una RDMA Write sta tentando di leggere in una zona di memoria non consentita, provocando un errore di protezione;
- Completion Protection Error: una zona di memoria viene deregistrata prima che sia stato completato il relativo descrittore;
- Catastrofic Error: errore di hardware o di configurazione.

Gli errori asincroni sono riportati dal VI Provider al VI Consumer per mezzo di un opportuno handler di default; l'utente può cambiare il comportamento del VI Provider nel caso si verifichi un errore asincrono, registrando un nuovo handler che si sostituisce al precedente nella gestione degli errori asincroni. Questo meccanismo è realizzato dalla funzione alla VipErrorCallback:

```
VIP_RETURN VipErrorCallBack (
        IN VIP_NIC_HANDLE NicHandle,
        IN VIP_PVOID Context,
        IN void(*Handler)(
            IN VIP_PVOID Context,
            IN VIP_ERROR_DESCRIPTOR *ErrorDesd;
        )
)
```

VipErrorCallback registra come nuovo gestore degli errori asincroni per il NIC identificato da *NicHandle* la funzione*Handler*.

La distinzione tra gli errori di tipo sincrono e quelli che invece vengono riportati in modo asincrono dal VI Provider dipendono dal livello di affidabilità della connessione.

Gestione degli errori per connessioni non affidabili

La gestione degli errori in modo asincrono per connessioni non affidabili viene invocata solo in caso di errori hardware come una coda inutilizzabile, errori relativi al NIC o perdita della connessione.

Gestione degli errori per connessioni affidabili in spedizione

La gestione degli errori per connessioni affidabili in fase di spedizione viene invocata in caso di errori hardware, come una coda corrotta, errori relativi al NIC o perdita di connessione, oppure per errori nel trasporto quali perdita di pacchetti. Al verificarsi di un errore asincrono la VI viene posta nello stato di *errore*. Il meccanismo di gestione degli errori asincroni deve essere attivato dopo che uno o più descrittori sono stati completati dopo il verificarsi dell'errore.

Gestione degli errori per connessioni affidabili in ricezione

Come nel caso precedente anche la gestione degli errori per connessioni affidabili in fase di spedizione viene attivata in caso di errori hardware, come una coda corrotta, errori relativi al NIC o perdita della connessione, oppure per errori nel trasporto quali perdita di pacchetti. La differenza consiste nel fatto che in questo caso la funzione di gestione deve essere attivata al massimo dopo che un solo descrittore è stato completato dopo il verificarsi dell'errore. Un errore asincrono anche in questo caso provoca la transizione in stato di *errore* della VI.

Capitolo 4

VIA per Linux: M-VIA

4.1 Caratteristiche

M-VIA costituisce un pacchetto software che mette a disposizione una realizzazione della Virtual Interface Architecture per Linux[1].

È stata prodotta dai ricercatori del National Energy Research Scientific Center (NERSC) presso i National Laboratories Lawrence Berkeley [24].

Fino alla fine del marzo 1999 sono state rilasciate due versioni del pacchetto completo di M-VIA che comprende alcuni moduli caricabili dal sistema operativo, alcuni driver e una libreria utilizzabile da parte dell'utente[2].

Bisogna sottolineare come le due versioni di M-VIA attualmente disponibili sono delle beta release e come tali non ancora testate in modo esaustivo. Inoltre il limitato numero di schede di rete supportate consente di effettuare comunicazioni con i seguenti dispositivi:

- Device di loopback per comunicazioni all'interno della stessa macchina.
- Schede di rete Ethernet/Fast Ethernet basate sul chip DC21x4x (Tulip) della Digital.
- Schede di rete Gigabit Ethernet Packet Engines GNIC-I (Yellowfin) e GNIC-II (Hamaci)[3].

[1] Vi è un'altra implementazione di VIA per Linux per reti Myrinet; esistono altre implementazioni di VIA per il sistema operativo Windows NT.

[2] È un insieme di funzioni scritte in linguaggio C che definiscono l'interfaccia tra il VI User e il VI Provider.

[3] Le GNIC-II sono supportate a partire dalla seconda beta release di M-VIA del dicembre 1998.

È inoltre necessario dotarsi di un kernel particolare[4] nel quale i driver delle periferiche di rete installate sulla macchina non devono essere compilati all'interno del kernel, bensì come moduli o non compilati affatto.

Le caratteristiche di M-VIA sono, come dichiarate dai produttori:

- VI Provider costituito da moduli caricabili dal sistema operativo, da cui il nome di Modular VIA[5].
- Modularità nel disegno del software che costituisce M-VIA, caratteristica che dovrebbe facilitare l'estensione del codice esistente a nuovi driver di schede di rete.
- Possibilità di utilizzo con l'hardware appositamente prodotto per VIA (VI NIC) oppure con le schede di rete già supportate e emulazione via software delle funzionalità di VIA.
- Coesistenza di M-VIA con i protocolli di rete tradizionali, come TCP/IP, sulla stessa rete fisica.
- Conformità agli standard VIA come attesta il superamento dei severi test di conformità di Intel; questa caratteristica dovrebbe assicurare la compatibilità del codice prodotto per M-VIA con altre realizzazioni di VIA e quindi altri sistemi operativi.
- Alto livello di prestazioni ottenuto con l'uso di fast trap al posto delle tradizionali system call.
- Possibilità di ottenere gratuitamente il codice sorgente e ridistribuirlo.

Gli sviluppatori di M-VIA hanno cercato di mantenersi fedeli all'idea che sta alla base di VIA, cioé quella di concedere all'utente accesso diretto alla scheda di rete senza l'intervento del kernel del sistema operativo nelle operazioni di comunicazione. Un altro sforzo realizzativo ha cercato di ridurre le copie intermedie tra i dati scambiati: infatti una operazione di send per inviare dati invoca una fast trap, per evitare l'overhead di una system call e

[4]La prima versione beta abbisogna di un kernel di versione compresa tra la 2.1.125 e la 2.1.128, mentre la seconda versione beta necessita di un kernel 2.1.x con x maggiore o uguale di 129; nessuna delle due release è supportata dalla nuova serie di kernel 2.2.

[5]Per il concetto di modulo del sistema operativo si consulti l'appendice.

non compie alcuna copia di dati (zero-copy) in buffer intermedi; per quanto riguarda la receive invece i dati sono ricevuti direttamente per mezzo di una procedura di interrupt hardware di basso livello che svolge tutte le operazioni necessarie sui dati in arrivo compresa una copia dei dati nella memoria del processo utente destinatario della comunicazione, inevitabile nel caso di comunicazioni protette senza un apposito supporto hardware.

Per quanto riguarda l'uso in sistemi SMP come il sottocluster B di ParMa[2], l'uso del device di loopback dovrebbe teoricamente permettere prestazioni molto elevate: una comunicazione tra processi all'interno di un nodo SMP richiede solo una copia di dati, dalla memoria del processo mittente direttamente a quella del processo ricevente. Solitamente invece questo è possibile solo se le zone di memoria dei due processi coinvolti nella comunicazione si trovano sullo stesso segmento condiviso, altrimenti si rende necessaria una copia in un buffer intermedio.

Per potere sviluppare correttamente applicazioni basate su M-VIA bisogna osservare che alcune delle caratteristiche della Virtual Interface Architecture non sono ancora state realizzate dai progettisti di M-VIA, anche se indicate come prossimi sviluppi:

- Dei tre livelli di affidabilità contemplati in VIA solo uno è disponibile ed è ovviamente l'Unreliable Delivery, poichè è una caratteristica richiesta per la conformità allo standard VIA.

- Le operazioni di RDMA Read non possono essere utilizzate, per quanto riguarda le modalità di comunicazione, in conseguenza alla mancanza di livelli di affidabilità più alti dell'Unreliable [23].

- Il modello di connessione peer-to-peer non è stato realizzato; diventa quindi obbligatorio usare l'altra modalità di connessione, quella di tipo client-server.

- L'attributo dei descrittori relativo alla qualità del servizio viene ignorato ed è necessario impostarlo a zero per evitare un errore.

Oltre alle caratteristiche che non sono ancora state rese diponibili, M-VIA presenta alcuni limiti:

- Le risorse a disposizione di ogni VI NIC sono finite: è limitato il numero di VI che è possibile creare, il numero di Completion Queue.

- Dal momento che ogni descrittore e ogni buffer per poter essere trasmessi o ricevuti devono essere registrati e l'operazione di registrazione di una zona di memoria implica che tale area non possa essere soggetta a swapping ma debba essere bloccata in memoria fisica, in sistemi con dimensioni di memoria RAM limitate è possibile esaurire le risorse messe a disposizione dalla macchina. Limitato è anche il numero di buffer che è possibile registrare anche a causa della dimensione della Page Table che il VI NIC possiede per tenere traccia delle zone registrate.

- La gestione di messaggi di lunghezza molto breve non è molto efficiente a causa di due motivi: il primo è che per messaggi di pochi byte il numero di byte di informazione inviati è molto inferiore al numero di byte effettivamente coinvolti nel trasferimento in quanto la dimensione di un descrittore è di 64 byte. In secondo luogo per una operazione di invio di dati sono comunque necessari due trasferimenti di dati tra la memoria e il NIC, oltre a quello dei buffer di dati: quello del descrittore e quello relativo al Doorbell.

- La richiesta di risorse per i Doorbell è molto onerosa: dal momento che a ogni coppia di Doorbell per ogni VI creata da un processo deve essere riservata una zona della memoria della scheda di rete e tale memoria deve essere mappata nella memoria del processo in modo da evitare un interrupt ogni volta che il VI NIC acceda a questa memoria, è indispensabile bloccare un'intera pagina in memoria fisica per ogni coppia di Doorbell; questo consuma oltre che memoria fisica, gran parte della quale non utilizzata, anche la memoria della periferica di rete e spazio di indirizzamento del processo.

- Il continuo monitoraggio dei Doorbell per verificare se qualche descrittore sia stato postato su di una coda è una operazione dispendiosa in termini di tempo.

- Manca un qualunque supporto per comunicazioni di tipo broadcast.

- La massima dimensione di pacchetto trasferibile è di 32KByte; per messaggi più grandi è il VI Consumer che si deve occupare di segmentazione e riassemblamento.

Gli sviluppatori di M-VIA al NERSC hanno pensato di superare alcuni di questi limiti appena enunciati, e nei loro progetti suggeriscono dei miglioramenti alla attuale struttura di VIA con la definizione di una versione “intelligente” di M-VIA (INVIA) [25], che dovrebbe includere queste caratteristiche:

Active Doorbell. Per meglio supportare messaggi brevi si propone il concetto di Doorbell attivo che fornisca tutte le informazioni necessarie al NIC per eseguire direttamente il trasferimento dei dati eliminando in tal modo la necessità dell'uso di un descrittore.

Fast descriptor. In aggiunta o in alternativa agli active Doorbell si può pensare a dei particolari descrittori che integrino un descrittore standard e dati. Questo già in parte accade con l'uso di Immediate Data nei descrittori, ma la dimensione dei dati che possono essere trasferiti è limitata a quattro byte, mentre sarebbe auspicabile potere inviare alcune decine di byte assieme a un descrittore.

Broadcast VI. Il supporto per il multicast aumenterebbe enormemente la complessità della struttura di VIA che deriva parte della sue performance dalla semplicità dovuta al modello di comunicazione orientato alla connessione tra due sole Virtual Interface connesse. Si potrebbe pensare a un tipo alternativo di Virtual Interface conesso a un gruppo di VI, nella quale ogni descrittore postato su di una coda venga trasmesso a tutte le VI appartenenti al gruppo.

4.2 Confronto di prestazioni con TCP e UDP

Le prestazioni ottenute da M-VIA su ParMa^2, in particolare sul sottocluster B, sono state paragonate ai risultati dei protocolli di comunicazione TCP e UDP, per valutare i miglioramenti conseguibili con l'uso dell'interfaccia di comunicazione VIA.

Le misure sono state effettuate con semplici programmi per le prestazioni di M-VIA e con benchmark standard per TCP e UDP, mantenendo inalterato il modello di valutazione delle prestazioni: a rete scarica due processi comunicano tra loro scambiandosi messaggi di lunghezza diversa secondo una modalità di tipo pingpong, in cui un processo invia un messaggio e si pone in attesa di un analogo messaggio proveniente dal secondo processo, il quale prima attende il messaggio, poi lo spedisce indietro al primo processo. Misurando con apposite procedure il tempo trascorso per un gran numero di scambi di messaggi secondo queste modalità e mediando si ottiene il round-trip-time, cioé il tempo impiegato per trasmettere e ricevere un messaggio; dividendo per due si determina il tempo di trasmissione di un messaggio, corrispondente, secondo il modello LogP presentato nel primo capitolo, alla somma di due volte l'overhead, della latenza e dell'inverso della banda per la lunghezza del messaggio. Con queste misure quindi, si è in grado di stimare la latenza, considerando il ritardo di trasmissione per messaggi di lunghezza molto piccola, anche se questo valore comprende due volte l'overhead; quindi si può parlare di latenza in modo improprio.

Per quanto riguarda la banda invece, essa può essere ottenuta dividendo la dimensione di un messaggio per il suo tempo di trasmissione, usando messaggi di dimensioni molto grandi affinchè la latenza e l'overhead diventino trascurabili rispetto al tempo per la trasmissione sul canale fisico.

Il confronto tra i tempi di trasmissione ottenuti con i tre protocolli studiati con interconnessione dei nodi tramite switch è riportato nella tabella 4.1.

Per questi primi valori è possibile notare come il TCP/IP presenti una latenza maggiore rispetto all' UDP/IP spiegabile con un numero maggiore di operazioni che devono essere applicate al pacchetto prima della sua spedizione. VIA presenta le prestazioni migliori per piccoli pacchetti.

A partire da dimensioni di circa 2Kbyte il tempo di trasmissione di UDP/IP supera quella di TCP/IP; questo comportamento è spiegabile con la perdita di pacchetti che occasionalmente affligge UDP in quanto unreliable.

Analoghe misure sono state eseguite con l'hub come dispositivo di interconnessione tra i nodi del cluster: i tempi di trasmissione sono inferiori rispetto allo switch per dimensioni basse dei pacchetti, dal momento che lo switch store and forward introduce una latenza significativa, e per UDP che non richiede messaggi di acknowledge. Per pacchetti TCP grandi invece, entrano in gioco le più frequenti collisioni provocate dalla banda condivisa dell'hub.

I test di prestazioni sui tempi di trasmissione sono stati svolti anche in locale, mostrando come M-VIA sia notevolmente più veloce: in questo caso i risultati non dipendono più dalle caratteristiche della rete di interconnessione, ma solo dalla realizzazione software del protocollo di comunicazione. Bisogna

Bytes	TCP/IP (μs)	UDP/IP (μs)	VIA (μs)	Bytes	TCP/IP (μs)	UDP/IP (μs)	M-VIA (μs)
1	100,2	84,8	61	512	200,2	184,6	152
2	100,6	85	61	515	201	185,4	153
3	101	84,6	61	765	249,2	234	200
4	101	84,8	61	768	250	234,4	199
6	101,4	85	61	771	250	235,4	200
8	102	84,8	61	1021	298,8	282	248
12	102,8	85	61	1024	299	282,2	246
13	102,8	85	61	1027	299,6	283,2	249
16	103,8	85	61	1533	399,8	404,6	332
19	104,6	85,2	62	1536	400	405	331
21	105	86,4	62	1539	400	406,4	332
24	105,6	86,8	62	2045	433,4	457,8	380
27	106	87,8	62	2048	433,4	457,6	380
29	106,4	87,8	63	2051	433,4	458,2	380
32	106,4	88,2	62	3069	541	547,4	460
35	109	89	64	3072	541,2	547,8	458
45	111	91,2	66	3075	541,4	548	459
48	111,8	92	66	4093	597	645,8	550
51	112,6	93	68	4096	597,2	645,6	549
61	114,6	96,6	69	4099	597,8	646,2	550
64	115	97,2	69	6141	811,8	851,2	705
67	115,8	98	71	6144	811,8	851	704
93	120,4	103,8	76	6147	812,2	851,6	705
96	120	104,2	76	8189	962,8	1089,4	894
121	121	104,8	77	8192	962,8	1089,8	894
125	126	109,2	82	8195	963	1090	894
128	126	110	82	12285	1320	1468	1232
131	127	111	83	12288	1319,8	1467,8	1230
189	138	121,4	94	12291	1320	1468,2	1231
192	139	122	94	16381	1644,8	1931,6	1557
195	139	123	94	16384	1645	1931	1555
253	150,8	134	106	16387	1646,2	1933,4	1556
256	151	134,6	105	24573	2350	2695,2	2258
259	151,6	135,6	106	24576	2350	2694,6	2257
381	175	159	129	24579	2350,4	2696,8	2266
384	175,8	159,2	129	32765	3056,2	3517,4	2949
387	176,2	160,6	130	32768	3056,4	3517,4	2947
509	200	184	153				

Tabella 4.1: Tempi di trasmissione con TCP, UDP e M-VIA con switch.

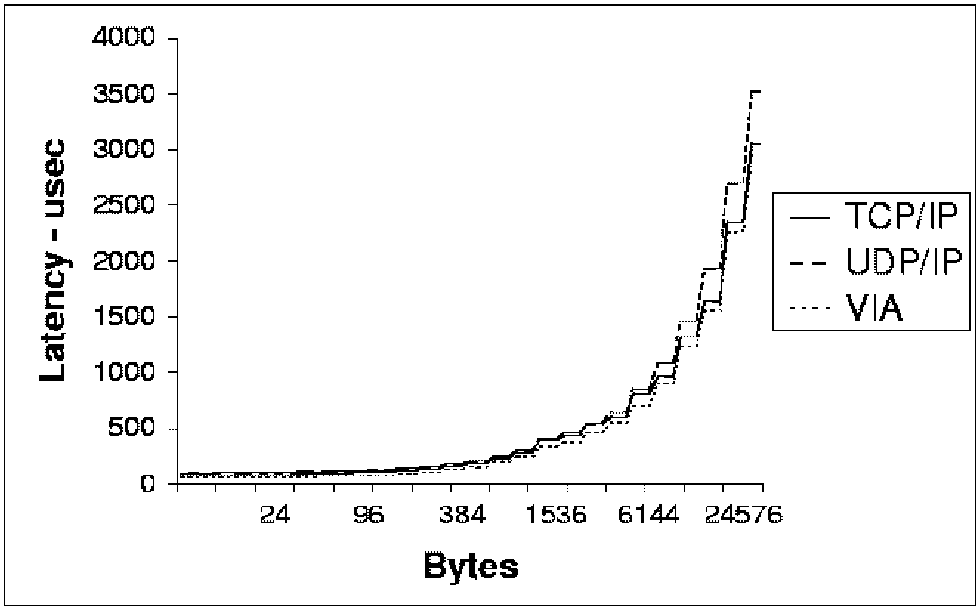

Figura 4.1: Tempi di trasmissione con TCP, UDP e M-VIA con switch fino 32KByte.

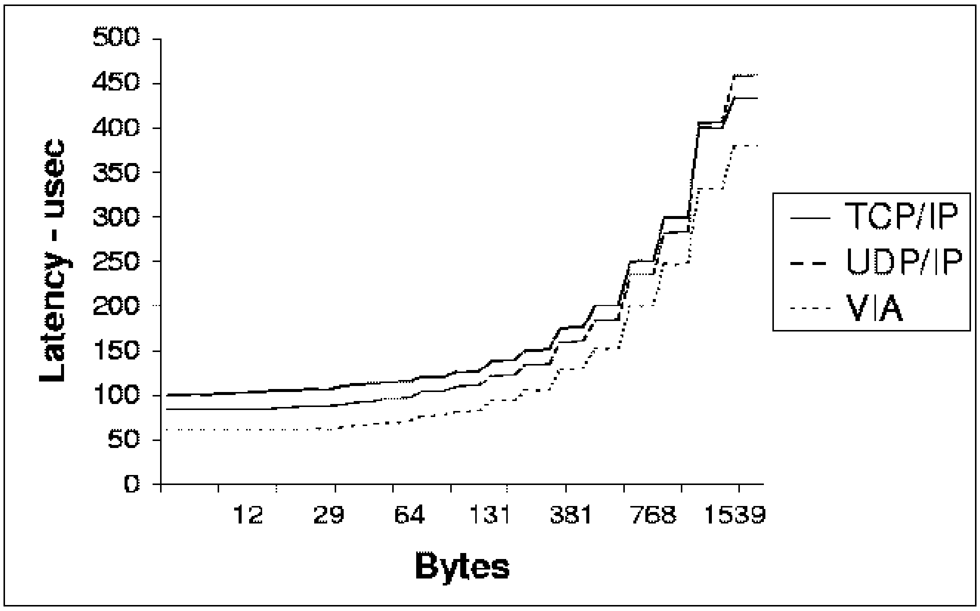

Figura 4.2: Tempi di trasmissione con TCP, UDP e M-VIA con switch fino 2KByte.

Bytes	TCP/IP (μs)	UDP/IP (μs)	VIA (μs)	Bytes	TCP/IP (μs)	UDP/IP (μs)	M-VIA (μs)
1	86	71	48	512	133	121	91
2	86	71	48	515	133	121	91
3	86	71	48	765	157	145	112
4	86	71	48	768	157	145	113
6	86	71	48	771	158	145	113
8	86	71	48	1021	181	168	135
12	86	71	48	1024	181	168	134
13	86	71	48	1027	181	169	136
16	87	71	48	1533	234	244	174
19	87	72	49	1536	234	244	173
21	88	72	49	1539	234	246	173
24	88	73	49	2045	265	297	220
27	88	73	49	2048	265	297	220
29	88	73	49	2051	265	298	220
32	88	73	49	3069	377	385	300
35	89	74	50	3072	377	385	299
45	91	75	51	3075	377	385	299
48	91	76	51	4093	451	483	390
51	92	76	52	4096	451	483	389
61	93	79	53	4099	452	483	390
64	93	79	53	6141	660	690	546
67	93	79	53	6144	660	691	544
93	95	83	56	6147	660	690	545
96	95	83	56	8189	824	920	734
121	96	84	58	8192	824	920	734
125	97	84	59	8195	824	920	734
128	97	85	58	12285	1210	1314	1071
131	98	85	59	12288	1209	1314	1071
189	103	91	64	12291	1208	1314	1071
192	104	91	64	16381	1626	1750	1396
195	104	91	64	16384	1620	1750	1396
253	109	97	70	16387	1622	1750	1395
256	109	97	70	24573	2456	2555	2093
259	110	98	70	24576	2448	2555	2093
381	121	109	80	24579	2458	2556	2093
384	121	109	80	32765	3351	3366	2783
387	121	109	81	32768	3350	3365	2782
509	133	121	91				

Tabella 4.2: Tempi di trasmissione con TCP, UDP e M-VIA con hub.

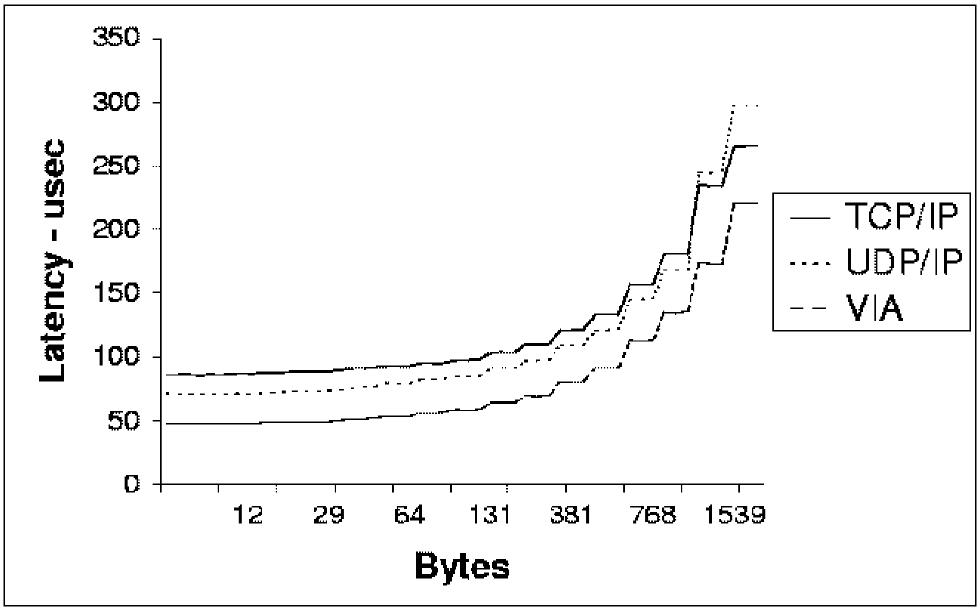

Figura 4.3: Tempi di trasmissione con TCP, UDP e M-VIA con hub fino a 2KByte.

inoltre rilevare che queste misure sono state effettuate usando il device di M-VIA adibito alla comunicazione su rete, in quanto il device di loopback presenta per ora dei bug che ne pregiudicano un funzionamento efficiente.

Analizzando il modello di testing si è detto che si sarebbe ottenuto un valore approssimabile alla banda solo per pacchetti di elevate dimensioni; tuttavia il massimo valore di dimensione supportato dai tre protocolli è diverso tra loro: non è definito per TCP, e si è quindi scelto 1MegaByte, mentre risulta di 64KByte il limite di UDP e di 32KByte quello di M-VIA; a questi valori sono riferiti i dati riportati nella tabella. La banda maggiore è assicurata sempre da M-VIA, sia che i nodi siano connessi con uno switch che con un hub, in locale e in remoto; i valori sono migliori con l'hub perché lo switch introduce un ritardo di trasmissione proporzionale alla dimensione del pacchetto inviato. Per TCP invece il ritardo introdotto dallo switch è compensato dai collegamenti full-duplex che lo switch stesso assicura: TCP richiede che i messaggi ricevano una notifica di ricezione che può essere inviata dal processo ricevente contemporaneamente a un successivo messaggio del primo processo.

4.3 Prestazioni delle primitive di M-VIA

In questo paragrafo vengono prese in considerazione le istruzioni di base di M-VIA valutando il tempo impiegato per la loro esecuzione, al fine di definire alcuni suggerimenti per l'uso di queste istruzioni per una programmazione efficiente.

La scelta di VIA è quella di ottimizzare le operazioni di trasferimento dei dati, critiche dal punto di vista della latenza, mentre per inizializzare e gestire le strutture necessarie alle comunicazioni sono richieste altre operazioni, alcune delle quali relativamente dispendiose in termini di tempo:

- La apertura di un VI NIC richiede che il Kernel Agent provveda a inizializzare tutte le strutture dati, a creare thread delegate a specifiche operazioni come la notifica degli errori, a restituire al processo chiamante un handle da usare per ogni successiva operazione; tutte queste operazioni richiedono circa un secondo[6].

[6] Il tempo è comunque talmente elevato che si sospetta la presenza di un bug.

Bytes	TCP/IP (μs)	UDP/IP (μs)	VIA (μs)	Bytes	TCP/IP (μs)	UDP/IP (μs)	M-VIA (μs)
1	32	18	4	512	34	20	6
2	30	18	4	515	34	20	7
3	30	18	4	765	36	21	8
4	30	18	4	768	36	21	7
6	30	18	4	771	36	21	8
8	31	18	4	1021	37	23	9
12	31	18	4	1024	38	23	8
13	31	18	4	1027	38	23	9
16	31	18	4	1533	40	25	12
19	31	18	4	1536	40	25	10
21	31	18	4	1539	40	25	12
24	31	18	4	2045	43	29	12
27	31	18	4	2048	43	29	12
29	31	18	4	2051	43	28	15
32	31	18	4	3069	49	33	20
35	31	18	4	3072	49	33	17
45	31	18	4	3075	50	33	21
48	31	18	4	4093	72	80	30
51	31	18	4	4096	72	80	22
61	31	18	4	4099	72	80	28
64	31	18	4	6141	89	109	39
67	31	18	4	6144	89	109	32
93	31	18	4	6147	89	109	40
96	31	18	4	8189	128	147	55
121	31	18	4	8192	128	147	43
125	31	18	4	8195	128	147	52
128	31	18	4	12285	177	212	86
131	31	18	4	12288	176	211	70
189	32	19	5	12291	177	211	86
192	32	19	5	16381	230	278	115
195	32	19	5	16384	228	278	93
253	33	20	5	16387	229	278	113
256	32	19	5	24573	373	408	171
259	32	19	5	24576	362	409	140
381	33	20	6	24579	367	409	168
384	33	20	6	32765	438	538	227
387	33	20	6	32768	439	537	186
509	34	20	6				

Tabella 4.3: Tempi di trasmissione con TCP, UDP e M-VIA in locale.

Connessione	Protocollo	Banda (Mbit/s)
Hub	TCP/IP	74,33
Hub	UDP/IP	74,19
Hub	VIA	94,24
Switch	TCP/IP	86,02
Switch	UDP/IP	71,19
Switch	VIA	88,96
Local	TCP/IP	540
Local	UDP/IP	519,6
Local	VIA	1405,92

Tabella 4.4: Banda passante di TCP, UDP, M-VIA.

- La creazione di una VI richiede una transizione attraverso il kernel, e una interazione tra il kernel e il VI NIC: il kernel deve fornire lo spazio in memoria per contenere una Receive Queue e una Send Queue, due Doorbell; il VI NIC deve a sua volta tenere traccia degli indirizzi di memoria fisica in cui queste strutture dati sono contenute.

- La connessione di due VI comporta l'esecuzione di un protocollo tra due processi che implica diverse operazioni.

- La registrazione di un dato richiede una transizione attraverso il kernel; è bene quindi non inserire questa operazione tra le comunicazioni.

Registrazione della memoria

La registrazione di un dato richiede l'intervento del kernel del sistema operativo che deve bloccare una pagina di memoria fisica impedendone lo swapping; l'indirizzo virtuale di questa pagina deve essere poi inserito in una tabella, la Page Table, gestita dal VI NIC che permetta di risalire all'indirizzo fisico. Il tempo necessario per la registrazione quindi è relativamente elevato, tanto che la registrazione di un dato al momento del suo invio aumenta la latenza della comunicazione in misura da annullare i vantaggi di VIA.

Si raccomanda quindi di non inserire la registrazione all'interno dei path critici di comunicazione, di non registrare e deregistrare con frequenza delle zone di memoria, dal momento che anche la deregistrazione è dispendiosa

Istruzione	Tempo(μs)
VipOpenNic	1.009.943,46
VipNSInit	109,52
VipQueryNic	2,49
VipNSGetHostByName	1,14
VipCreatePtag	2,18
VipCreateVi	101,8
VipDisconnect	1,74
VipDestroyVi	3,07
VipCloseNic	5,24
VipDestroyPtag	1,9
Connessione con switch	420
Connessione con hub	333
Connessione locale (via_eth)	49
Connessione locale (via_lo)	45

Tabella 4.5: Prestazioni delle primitive di M-VIA.

in termini di tempo e che si potrebbero verificare dei side-effect come la frammentazione della tabella del VI NIC.

Bytes	Tempo (μs)	Bytes	Tempo (μs)	Bytes	Tempo (μs)	Bytes	Tempo (μs)
1	52,3	61	49,6	512	51,3	4099	69,3
2	51,3	64	51,3	515	53,6	6141	82,6
3	52	67	50	765	53,3	6144	77,6
4	52	93	50	768	54	6147	81,6
6	53	96	50,6	771	54	8189	99,3
8	53,6	99	51,6	1021	55	8192	87,6
12	53,6	125	51	1024	54	8195	102,6
13	53,6	128	51,3	1027	54,3	12285	116,6
16	53	131	49,6	1533	55,3	12288	111,6
19	49,3	189	50,3	1536	56,6	12291	121,6
21	49	192	49	1539	58,6	16381	142
24	50	195	51	2045	59,3	16384	134,6
27	48,3	253	50	2048	56,3	16387	136,6
29	50	256	50,6	2051	60,3	24573	175
32	50	259	50,6	3069	67	24576	175,3
35	51,3	381	51	3072	62,6	24579	176,3
45	52,6	384	51,6	3075	66	32765	210,6
48	50	387	50,6	4093	70	32768	207,6
51	50	509	52,6	4096	63,3		

Tabella 4.6: Prestazioni dell'istruzione VipRegisterMem.

Capitolo 5

Message Passing Interface (MPI)

5.1 Paradigma Message Passing

Nel primo capitolo si è definito un sistema parallelo come un insieme di elementi di calcolo che collaborano e comunicano per risolvere in modo più efficiente problemi di notevole complessità. Tuttavia il modo con cui tali elementi cooperano può conformarsi a diversi modelli di programmazione che specificano cosa, come e quanto comunicano tra loro le entità che lavorano in parallelo al fine di risolvere il problema globale e le modalità con le quali esse si possano sincronizzare.

Si possono identificare tre famiglie di modelli di programmazione:

Shared Address Space. Questo modello sfrutta architetture i cui nodi di calcolo sono costituiti da processi che risiedono generalmente su macchine parallele di tipo *shared memory multiprocessors*, tra cui le più comuni sono quelle di tipo *Symmetric MultiProcessor*: ogni processo possiede un proprio spazio di indirizzi di memoria virtuale visibile a ogni sua thread; inoltre, i processi cooperanti hanno in comune una regione di memoria fisica mappata in indirizzi virtuali diversi e propri di ogni processo: attraverso questa zona di memoria essi possono comunicare e interagire tra loro. Queste comunicazioni hanno luogo con convenzionali operazioni di lettura e scrittura in memoria, ed è l'architettura di comunicazione tra memoria e processore che ha il compito di gestire le primitive di accesso alla memoria condivisa e le operazioni di sincronizzazione.

Questo modello di programmazione richiede, come si diceva, macchine di tipo shared memory multiprocessors; tuttavia anche in sistemi di tipo NOW sono state sperimentati meccanismi di Distributed Shared Memory per realizzare un paradigma di progrmmazione di tipo Shared Address Space: occorre realizzare delle funzioni che simulino uno spazio di indirizzamento globale garantendo un accesso alla memoria condivisa indipendemente dalla sua collocazione fisica, incapsulando le reali modalità di comunicazione tra i processi che usino la memoria condivisa ma fisicamente separata.

Message Passing. Questo modello di programmazione è opposto al precedente in quanto non esiste alcuna area di memoria accessibile a più di un processo; la comunicazione interprocesso avviene con l'utilizzo di primitive specifiche che hanno il compito di trasmettere o ricevere messaggi con una sintassi ben definita; ogni forma di cooperazione tra processi quindi è determinata da un evento relativo allo scambio di un messaggio. È un tipo di programmazione che si adatta bene a un sistema parallelo di tipo cluster di PC: utilizzando le modalità di comunicazione di basso livello offerte dall'hardware e dal sistema operativo è possibile realizzare una libreria di primitive per lo scambio di messaggi tra processi, ognuno dei quali possiede una propria zona di memoria che nel caso di processi residenti su nodi diversi è fisicamente separata dalla memoria degli altri processi e quindi difficilmente accessibile.

Data Parallel. Il parallelismo è ottenuto suddividendo i dati tra i diversi processi che svolgono sui dati stessi operazioni simili o identiche; terminata una operazione avviene una fase di scambio dei risultati tra i processi. Linguaggi di tipo data parallel possono essere realizzati a partire dalle istruzioni macchina o possono anche sfruttare per le comunicazione uno dei paradigmi di programmazione in precedenza illustrato.

Il modello di programmazione che meglio si adatta a una architettura di tipo cluster di PC come ParMa^2 è, come prima accennato, il Message Passing, in quanto lo Shared Address Space richiederebbe la realizzazione di primitive per la creazione e la gestione di uno spazio di indirizzi globale per una memoria fisicamente distribuita su nodi diversi; il paradigma Data Parallel invece non può essere realizzato direttamente poichè mancano delle operazioni di basso livello per eseguire le stesse istruzioni su nodi diversi in quanto questi sono costituiti da PC separati che possono interagire tra loro solo comunicando via rete. Il Message Passing è quindi uno dei metodi più usati per la programmazione parallela in sistemi classificati, secondo una delle tassonomie più comuni, Multiple Instruction Multiple Data (MIMD) con memoria distribuita.

Nella programmazione parallela di tipo Message Passing il messaggio costituisce l'unità di informazione che i processi possono scambiarsi; il messaggio vero e proprio è costituito dai dati che i processi desiderano scambiarsi, tuttavia per potere efficacemente trasferire messaggi ogni processo deve fornire al sistema di Message Passing anche le seguenti informazioni:

- Il processo mittente.
- L'area di memoria contenente il dato da inviare.
- Il tipo del dato da spedire.
- La dimensione complessiva dei dati.
- Il processo o i processi riceventi.
- La zona di memoria destinata a ricevere i dati.
- La quantità di dati che il processo destinatario è disposto a ricevere.

Alcuni di questi dati vengono esclusivamante utilizzati dalla libreria di Message Passing, la quale realizza le procedure di più basso livello che si occupano dell'effettivo trasferimento dei dati. Altri invece rappresentano delle informazioni utili anche per i processi comunicanti anche se non sono parte integrante del messaggio propriamente detto; essi vanno a costituire un insieme di dati chiamato *envelope* il quale viene inviato assieme al messaggio; un envelope contiene almeno i seguenti campi di informazione:

- Il processo mittente identificato univocamente da un numero chiamato rank.
- Il processo destinatario indicato dal suo rank.

- Un numero intero che identifica il tipo di messaggio, chiamato *tag*.
- Un identificatore del gruppo di processi cui il mittente e il destinatario appartengono, detto *comunicatore*.

Evidentemente le funzioni principali di una libreria di Message Passing hanno l'obiettivo primario di gestire la trasmissione e la ricezione di messaggi: molte delle comunicazioni ruotano quindi attorno alla coppia Send/Receive che permettono rispettivamente di mandare e ricevere un messaggio. Per fornire al programmatore la possibilità di costruire un'ampia gamma di tipologie di comunicazioni per la risoluzione di diverse classi di problemi le librerie di Message Passing forniscono primitive di comunicazioni differenziate per semantica, per numero di processi coinvolti e per durata della comunicazione. Quindi una libreria di Message Passing mette a disposizione comunicazioni di tipo:

Bloccante e non bloccante. Una funzione bloccante attende fino al completamento dell'operazione richiesta; al contrario una funzione non bloccante pone inizio alla comunicazione e subito dopo restituisce il controllo al processo chiamante, il quale può svolgere altri compiti mentre la comunicazione procede verso il suo completamento. Le comunicazioni di tipo non bloccante quindi portano due vantaggi: il processo non è obbligato ad aspettare in stato di attesa il completamento di una lenta operazione di I/O come quelle connesse al trasferimento di dati via rete; più comunicazioni possono avere luogo contemporaneamente sovrapponendosi almeno in parte tra loro ottenendo un risparmio di tempo.

Sincrono e asincrono. Una comunicazione di tipo sincrono richiede che il processo che effettua una operazione sia in precedenza o venga in seguito a conoscenza dello stato relativo alla comunicazione del processo destinatario: la comunicazione può avvenire con successo solo se i processi coinvolti si trovano in una situazione adatta oppure un processo attende una notifica che gli comunichi lo stato dell'altro processo. Una comunicazione di tipo sincrono solitamente comporta un *handshake* tra i processi coinvolti: il primo inizia la comunicazione e il secondo, dopo la corretta ricezione del messaggio, invia un acknowledgement con il quale notifica l'avvenuto completamento dell'operazione.

Point-to-point e collettivo. Le comunicazioni point to point coinvolgono una coppia di processi; comunicazioni di tipo collettivo permettono lo scambio di messaggi tra un numero arbitrario di processi all'interno di uno stesso comunicatore. Le comunicazioni di tipo collettivo non interferiscono con quelle di tipo point-to-point e viceversa.

Le funzioni di comunicazione collettiva di uso più frequente sono:

1. Barriera: è utilizzato per sincronizzare più processi; non avviene alcuno scambio di dati, ma la barriera blocca ogni processo appartenente a un comunicatore finchè tutti i processi non siano giunti a eseguire la funzione di barriera.
2. Broadcast: realizza una comunicazione di tipo uno a molti, nella quale un processo trasmette uno stesso messaggio agli altri processi del comunicatore.
3. Scatter e gather: sono funzioni che consentono rispettivamente di distribuire o di collezionare un insieme di dati ciascuno dei quali proviene o è destinato a processi diversi.
4. Riduzione: queste operazioni vengono usate nel caso che si debba eseguire un calcolo per ottenere un singolo risultato a partire da dati distribuiti su un gruppo di processi.

Persistente e non persistente. Nel caso in cui un processo si trovi nella situazione di dovere ripetere la medesima funzione di comunicazione con gli stessi argomenti per un numero elevato di volte è possibile migliorare le prestazioni introducendo un tipo di comunicazione detta *persistente* per la quale sono possibili alcune semplificazioni come quella ad esempio di spedire l'envelope una sola volta per tutte.

Queste caratteristiche sono tra loro combinate per ottenere operazioni di scambio di messaggi dalla diversa semantica; ogni libreria di Message Passing sceglie di fornire come primitive alcune di queste possibili combinazioni.

5.2 Uno standard per il Message Passing: MPI

I concetti di base relativi al Message Passing ebbero i loro primi sviluppi parecchi anni fa, in concomitanza con l'introduzione di sistemi MIMD e sono stati ampiamente applicati soprattutto per la programmazione di sistemi paralleli con memoria distribuita; è invece una iniziativa piuttosto recente l'introduzione di uno standard per la programmazione basata sul Message Passing, il Message Passing Interface (MPI) [26]. Lo sviluppo dello standard MPI ebbe inizio nel 1992 con il "Workshop on Standards for Message Passing in Distributed Memory Environment" in cui vennero enunciate le caratteristiche fondamentali cui lo standard avrebbe dovuto obbedire. Seguì un documento preliminare sullo standard noto come MPI1, dopo il quale si decise di formalizzare il cammino per la produzione dello standard con una serie di incontri e l'attivazione di gruppi di discussione e mailing list, similmente a quanto sucesso per High Performance Fortran (HPF) Forum. Nacque in questo modo l'*MPI Forum* aperto agli esperti di programmazione di architetture parallele di tutto il mondo.

Nel maggio 1994 venne prodotto il documento di definizione dello standard MPI, nel quale si descrivevano le caratteristiche dell'interfaccia per la programmazione di tipo Message Passing e gli scopi per l'adozione dello standard.

Il principale obiettivo che ha stimolato l'introduzione dello standard è stato quello di produrre un ambiente di programmazione parallela in cui le applicazioni, le librerie software, e i vari tools di sviluppo potessero essere portabili su differenti piattaforme hardware. Questo per valorizzare gli investimenti in software parallelo, per permettere lo sviluppo del codice su una macchina per poi eseguirlo su supercomputer paralleli ma anche per fornire ai produttori di software una interfaccia chiara e precisa che essi potessero realizzare con efficienza. Uno scopo non secondario di MPI è quello di facilitare il programmatore rendendo il suo lavoro e la sua formazione professionale più semplice e soprattutto indipendente dalla particolare libreria per la programmazione parallela utilizzata.

Le seguenti caratteristiche definiscono gli aspetti principali della interfaccia MPI:

- MPI è pensata come interfaccia per il Message Passing sia per applicazioni che per librerie software o compilatori di codice parallelo.

- Lo standard MPI è disegnato per la programmazione sia di multicomputers che di network di computer eterogenei.

- La semantica delle operazioni offerte dall'interfaccia MPI è stata pensata con lo scopo di essere indipendente dal linguaggio di programmazione utilizzato. MPI fornisce la sintassi specifica per i linguaggi C e Fortran 77.

- L'interfaccia MPI definita dall'MPI Forum è stata disegnata cercando di consentire che le sue realizzazioni garantiscano l'efficienza, in particolare fornendo operazioni dalla semantica non bloccante ed evitando le copie dei dati su buffer intermedi.

- La facilità di uso è uno degli obiettivi di progetto di MPI.

- Per quanto riguarda l'affidabilità delle comunicazioni MPI assume che il sottostante livello di trasporto sia affidabile di modo che il programmatore che usi MPI non si debba preoccupare di eventuali errori di trasporto; tuttavia la gran parte delle funzioni dell'interfaccia ritornano eventualmente un messaggio di errore.

- L'interfaccia MPI non è completamente diversa dalle più comuni librerie di Message Passing esistenti, ma anzi cerca di raggruppare tutte le loro caratteristiche fondendole assieme.

Per quanto riguarda più specificamente gli aspetti relativi alle funzionalità per la trasmissione e gestione dei messaggi, lo standard MPI include:

- Comunicazioni point-to-point.

- Funzioni di comunicazione collettiva.

- Gestione dei gruppi e delle topologie di processi.

- Definizione del contesto delle comunicazioni.

- Interfaccia per l'analisi delle applicazioni prodotte.

Al di fuori degli scopi di MPI1 sono invece:

- La definizione di tutto l'ambiente completo e integrato di programmazione parallela: MPI definisce solo delle funzioni per lo scambio e la gestione di messaggi.

- La gestione di I/O parallelo.

- Il supporto a strumenti di debugging.

- La possibilità di generare nuovi processi in fase di esecuzione.
- Il supporto esplicito per i threads.
- La gestione di operazioni che comportino degli scambi tramite shared memory.

Il lavoro di standardizzazione e studio compiuto dall'MPI Forum non si è interrotto con la stesura delle specifiche di MPI1, bensì è progredito costantemente, con la produzione di una versione MPI1.1 e ultimamente con lo studio di una estensione dello standard MPI, chiamato MPI2 che introduce in MPI ambiti che nella prima versione erano volutamente stati tralasciati:

- la gestione dei processi, in particolare la creazione e terminazione di processi in fase di esecuzione;
- le comunicazioni one-sided, quelle cioé in cui il processo comunicante specifica entrambi gli indirizzi dei dati, quello locale e quello remoto, scambiandosi quindi messaggi secondo una modalità di tipo Remote Memory Access (RMA);
- operazioni di I/O parallelo;
- la sintassi delle funzioni in linguaggio C++.

Si ponga ora l'attenzione sulle primitive fondamentali di Send e Receive che permettono di scambiare messaggi tra coppie di processi; MPI mette a disposizione quattro diverse modalità di comunicazione in riferimento all'operazione di Send, distinte in relazione al momento in cui la funzione viene completata, cioé, per definizione, quando il buffer usato da una Send per contenere i dati da inviare può essere riutilizzato correttamente:

Synchronuos send. Il processo che invia il messaggio attende finchè l'operazione non abbia raggiunto il suo completamento, ovvero il processo destinatario non abbia ricevuto il messaggio. Send di questo tipo possono bloccare il processo mittente, tuttavia sono usate per stabilire dei punti di sincronizzazione tra processi.

Buffered send. Una send bufferizzata è una send asincrona che viene immediatamente completata in quanto i dati da inviare sono copiati in un buffer fornito dal sistema che si occupa di effettuare il trasferimento fisico dei dati mentre il processo mittente è libero di svolgere altri compiti.

Ready send. Come nel caso della precedente operazione la ready send viene completata immediatamente, ma a differenza dalla buffered send la comunicazione ha successo solo nel caso che il processo destinatario sia pronto a ricevere nel momento in cui la ready send viene effettuata.

Standard send. Una standard send viene completata non appena il messaggio è stato inviato; ciò però non implica che il messaggio sia effettivamente arrivato a destinazione. Una standard send può essere realizzata come una send bufferizzata o come una send sincrona: per questa ragione il programmatore non deve assumere che una standard send sia equivalente a nessuna delle due send precedenti.

La precedente distinzione riguarda essenzialmente la differenziazione tra comunicazioni sincrone e asincrone; le funzioni di comunicazione di tipo Send infatti sono disponibili secondo i quattro modi di comunicazione oltre che in forma persistente o non persistente anche in versione bloccante o non bloccante come indicato nella tabella 5.1.

Le informazioni che bisogna inviare in una operazione di send si possono desumere dalla sua sintassi [27]:

int **MPI_Send** (
    void ∗buf,
    int count,
    MPI_Datatype datatype,
    int dest,
    int tag,
    MPI_Comm comm,
)

- buf è l'indirizzo dei dati da inviare;
- count indica il numero di elementi di tipo datatype che il buf contiene;
- datatype è il tipo di dato MPI contenuto in buf;

Modo di comunicazione	Forma bloccante	Forma non bloccante
Standard send	MPI_Send	MPI_ISend
Synchronous send	MPI_SSend	MPI_ISSend
Buffered send	MPI_BSend	MPI_BSend
Ready send	MPI_RSend	MPI_RSend

Tabella 5.1: MPI Send routines.

- dest è il rank del processo destinatario;
- tag viene utilizzato per distinguere tra i diversi messaggi;
- comm rappresenta il comunicatore, cioé il gruppo di processi coinvolti nelle comunicazioni.

Per quanto riguarda l'operazione di Receive l'unica distinzione che viene fatta è tra receive bloccante, che attende finchè non viene ricevuto un messaggio, e non bloccante, che inizia la ricezione di un messaggio lasciando però libero il processo di svolgere lavoro utile; in un secondo momento il processo può verificare il completamento della ricezione con semantica bloccante o non bloccante.

La sintassi delle MPI_Recv è [27]:

int **MPI_Recv** (
    void ∗buf,
    int count,
    MPI_Datatype datatype,
    int source,
    int tag,
    MPI_Comm comm,
    MPI_Status status,
)

- buf è l'indirizzo del buffer destinato a contenere i dati in arrivo; affinchè la comunicazione abbia successo esso deve avere una dimensione sufficiente a contenere il messaggio ricevuto senza troncamenti;
- count indica il massimo numero di elementi di tipo datatype che il buffer può contenere; il numero di elementi effettivamente ricevuti può essere minore;

- datatype è il tipo di dato MPI e deve corrispondere al tipo di dato indicato nella funzione di invio del messaggio;
- source è il rank del processo dal quale il messaggio può essere ricevuto; è anche possibile indicare che si desidera ricevere da uno qualsiasi dei processi del comunicatore;
- tag rappresenta un identificativo del tipo di messaggio che la receive può accettare; si può indicare anche in questo caso una ricezione non selettiva sul tipo di messaggio;
- comm rappresenta, come per la MPI_Send, il comunicatore;
- status viene aggiornato con alcune informazioni riguardanti la comunicazione, come il rank del processo mittente e il tag del messaggio ricevuto.

5.3 Prestazioni di LAM e MPICH

Lo sforzo di standardizzazione compiuto dall'MPI Forum è stato accolto con favore dagli operatori nel campo della programmazione parallela: attualmente infatti si contano più di una decina di diverse librerie per il Message Passing conformi a MPI gratuitamente fruibili e altrettante prodotte da venditori di software[1].

Ogni realizzazione di MPI presenta caratteristiche differenti in relazione alla tipologia di hardware e di sistema operativo supportato e alle modalità con cui i dati sono trasferiti. Si è scelto di analizzare il comportamento e le prestazioni di due librerire conformi a MPI: MPICH in quanto è molto comune dal momento che viene fornito nelle più comuni distribuzioni del sistema operativo Linux ed è adottato da grandi cluster come Beowulf; LAM MPI per la sua capacità di sfruttare anche la memoria condivisa per le comunicazioni tra processi residenti sullo stesso nodo in modo da migliorare sensibilmente le prestazioni di cluster con nodi di tipo Symmetric Multi Processor (SMP), come ParMa[2].

MPICH

[1]Una lista esauriente delle librerie per il Message Passing è reperibile alla pagina web *http://www.mpi.nd.edu/MPI/*.

MPICH è una realizzazione di MPI prodotta presso l'Argonne National Laboratory [26], con l'obiettivo primario di integrare la portabilità del codice prodotto su sistemi diversi e la capacità di ottenere elevate prestazioni nelle comunicazioni.

Le piattaforme hardware supportate, come previsto dagli obiettivi di progetto, sono molto numerose:

- Distributed memory parallel supercomputer come Intel Paragon, IBM SP2, Meiko CS-2, Thinking Machine CM-5, Cray T3D.
- Shared memory architecture come le diverse macchine della SGI e i sistemi SMP IBM.
- Network di workstation anche eterogenei, costituite da macchine con sistema operativo Unix dei produttori Sun, DEC, HP, SGI oppure macchine Intel con Linux o Windows 3.1.

MPICH inoltre si pone l'obiettivo di fornire un ambiente completo e portabile per la programmazione parallela, offrendo all'utente alcuni tools di sviluppo come un debugger, degli strumenti per la valutazione delle prestazioni, delle routine per la parallelizzazione di codice sequenziale, delle librerie grafiche parallele.

La portabilità di MPICH su diverse macchine è resa possibile dalla architettetura del software: gran parte del codice è indipendente dall'hardware sottostante grazie all'adozione di una Abstract Device Interface (ADI) le cui routine sono usate per costruire le funzioni MPI fruibili da parte dell'utente; vi sono diverse implementazioni di ADI per diversi sistemi hardware; a sua volta una singola ADI è realizzata tramite le funzioni di basso livello di una interfaccia semplice e leggera chiamata *channel interface* usata per facilitare la rapida costruzione di nuove ADI. Ne risulta una architettura software molto complessa[2] che riesce a supportare un notevole numero di piattaforme hardware.

Per quanto riguarda l'uso di MPICH sul cluster ParMa[2] è necessario specificare un particolare device chiamto p4 che costituisce il channel interface utilizzato per macchine dotate di sistema operativo Linux con processi che comunicano tra loro tramite socket; l'utilizzo di questo device è possibile anche

[2]Si veda la pagina web *http://www-unix.mcs.anl.gov/mpi/mpich/*.

in cluster eterogenei dal punto vista hardware, ma il suo limite è la mancanza del supporto per comunicazioni tramite shared memory per processi che risiedano sul medesimo nodo SMP.

LAM

Local Area Multicomputer (LAM) è un ambiente di programmazione parallela conforme allo standard MPI [28]; disegnato e realizzato in origine presso il Supercomputer Centre dell'Università statale dell'Ohio, oggi è sviluppato all'Università di Notre Dame.

A differenza di MPICH supporta solo cluster di computer, anche eterogenei dal punto vista dell'hardware.

Strumenti per il debugging e per il controllo delle applicazioni contribuiscono a fare di LAM un ambiente per la programmazione parallela; in particolare il controllo svolge la sua attività a due livelli: all'interno del codice possono essere inserite apposite istruzioni per ottenere informazioni riguardo lo stato di un processo e il contenuto dei messaggi, la sincronizzazione e i gruppi di processi; esistono anche strumenti per ricavare un riassunto completo delle comunicazioni visualizzabile in fase di esecuzione o dopo la terminazione dell'applicazione.

LAM mette già a disposizione dell'utente anche alcune delle caratteristiche che fanno parte delle specifiche MPI2, come la gestione dinamica dei processi che permette di variare il numero di processi in fase di esecuzione, il filesystem remoto che consente l'accesso da parte di un processo a un file esistente che risieda su uno dei nodi del cluster e le operazioni di I/O collettive, grazie alle quali i membri di un gruppo di processi possono partecipare alla stessa operazione di input o output. LAM facilita anche la gestione dei nodi partecipanti alla sessione di lavoro MPI consentendo la creazione e la soppressione di nodi, la rilevazione e la correzione di situazioni di nodo non funzionante.

La caratteristica che rende LAM particolarmente interessante è la capacità di offrire communicazioni multi protocollo: è cioé possibile specificare in parte al momento della compilazione del codice, in parte all'avvio del programma MPI che si desidera eseguire, quale tipo di protocollo di basso livello deve essere utilizzato per il trasporto dei dati sulla rete di interconnesione tra i nodi del cluster. Attualmente sono disponibili quattro diversi protocolli per lo scambio di dati: quello usato fin dalle prime versioni di LAM è basato su demoni che vengono creati all'avvio della sessione di lavoro su ogni nodo del cluster e si fanno carico di effettuare le comunicazioni che vengono richieste da ogni processo residente sul proprio nodo. Gli altri tre protocolli invece svolgono delle comunicazioni dirette tra i processi in esecuzione in parallelo sui diversi nodi, senza transitare attraverso i demoni, ottenendo delle performance migliori; tale comunicazione viene denominata *client-to-client* (c2c) e presenta tre diverse realizzazioni: una basata sull'uso del protocollo TCP/IP, due che integrano TCP/IP e l'uso della shared memory per processi residenti sullo stesso nodo[3]. Questa ultima caratteristica rende LAM particolarmente adatto all'uso nel cluster ParMa[2]: viene infatti gestita oltre all'eterogeneità dell'hardware dei nodi anche la comunicazione tramite memoria condivisa che consente di ottenere prestazioni molto elevate nel caso che più processi si trovino sullo stesso nodo SMP del cluster. Non è invece molto conveniente utilizzare più di un processo MPI alla volta su di un nodo a singolo processore in quanto i vantaggi assicurati dalla maggiore velocità di comunicazione interprocesso sono vanificati dall'overhead dovuto ai context switch provocati dallo scheduling della CPU che deve gestire in time sharing più processi; su di un nodo SMP due processi possono procedere nella loro esecuzione occupando ciascuno un processore e comunicando tra loro fruendo della memoria del nodo che è condivisa tra tutti i processi indipendentemente dalla CPU che li ospiti.

Prestazioni

Per valutare quale libreria di MPI fornisce le migliori prestazioni sono stati svolti alcuni semplici test per esaminare l'efficacia di LAM e MPICH nelle comunicazioni interprocesso sia nel caso di processi residenti su nodi diversi in cui la comunicazione avviene via rete, sia nel caso di processi sullo stesso nodo, in cui le comunicazioni si svolgono tramite shared memory, se supportato dalla libreria di MPI, oppure attraverso un device di rete di tipo loopback.

Pingpong bloccante. È un semplice test in cui una coppia di processi si scambiano per un certo numero di volte un dato[4] di dimensioni crescenti utilizzando delle primitive MPI_Send e MPI_Recv di tipo bloccante. Viene valutato il tempo richiesto affinchè un messaggio venga inviato

[3] Nel prossimo capitolo verranno forniti dettagli maggiori.
[4] La dimensione è indicata in doubles; un double occupa 8byte.

da un processo mittente a un processo destinatario il quale rispedisce lo stesso messaggio al primo processo. Questa procedura è ripetuta per un numero elevato di volte per mediare i tempi così ottenuti che rappresentano il tempo di *round-trip* impiegato da un messaggio per passare da un processo a un altro e tornare al primo; il tempo di round-trip viene diviso per due e per il numero di ripetizioni della prova e moltiplicati per il numero di byte inviati fornendo in questo modo la banda disponibile per la comunicazione. Il test viene eseguito tra due processi residenti su due nodi distinti del sottocluster B di $ParMa^2$ (tabella 5.2), e in un secondo caso con processi residenti sullo stesso nodo (tabella 5.3).

I risultati dimostrano che MPICH offre delle prestazioni migliori a LAM con comunicazioni tramite demoni; la situazione è invece invertita nel caso di LAM con comunicazione diretta tra processi via TCP/IP: la velocità di trasmissione è superiore in tutto l'intervallo di valori rispetto a MPICH.

La banda massima disponibile rilevata con LAM client to client è superiore a quella di MPICH, 11.8 contro 10.4 MegaByte al secondo, mentre simile è la dimensione del pacchetto che consente di raggiungere il massimo sfruttamento del canale fisico di trasmissione.

Anche nel caso di pingpong in locale MPICH si dimostra migliore di LAM con comunicazione tramite demoni, in modo più netto rispetto al pingpong remoto; il confronto con LAM client to client mette in rilievo la differenza di prestazioni ottenibile con l'uso della shared memory piuttosto che un device di loopback per le comunicazioni di processi sullo stesso nodo: i valori di velocità di trasmissione riscontrati sono più del doppio di quelli di MPICH per tutto l'intervallo dei valori.

Pingpong non bloccante. Questo programma è molto simile al precedente con la differenza che le funzioni di comunicazione utilizzate hanno una semantica non bloccante.

Per le primitive non bloccanti MPICH risulta ancora decisamente superiore a LAM con comunicazioni tramite demoni, ma a differenza dei casi precedenti esiste anche un intervallo di valori (tra 1 e 4 KByte) in cui MPICH è migliore anche di LAM con comunicazioni dirette client to client; questo vantaggio tuttavia si perde per dimensioni di pac-

Dimensione (doubles)	MPICH (MB/s)	LAM via demoni (MB/s)	LAM via TCP/IP(MB/s)
1	0,049236	0,012722	0,079691
2	0,098314	0,025349	0,156011
4	0,197648	0,050358	0,300633
8	0,39209	0,098525	0,566873
16	0,678755	0,193335	1,036165
32	1,324985	0,36756	1,724469
64	2,236133	0,689996	2,590205
128	3,120592	1,210505	3,445281
256	4,39957	1,978443	4,944804
512	5,979562	3,118317	6,860928
1024	7,567667	4,35633	8,691781
2048	9,290615	4,41557	10,011656
4096	10,01314	6,114004	10,017711
8192	10,51689	7,412309	10,785122
16384	10,335686	8,554498	11,416424
32768	10,321239	9,203518	11,622425
65536	10,237201	9,410347	11,672507
131072	10,278142	9,489589	11,797664
262144	10,280711	9,393991	11,829936
524288	10,41649	9,394296	11,845707
1048576	10,361154	9,418317	11,852124

Tabella 5.2: Pingpong remoto.

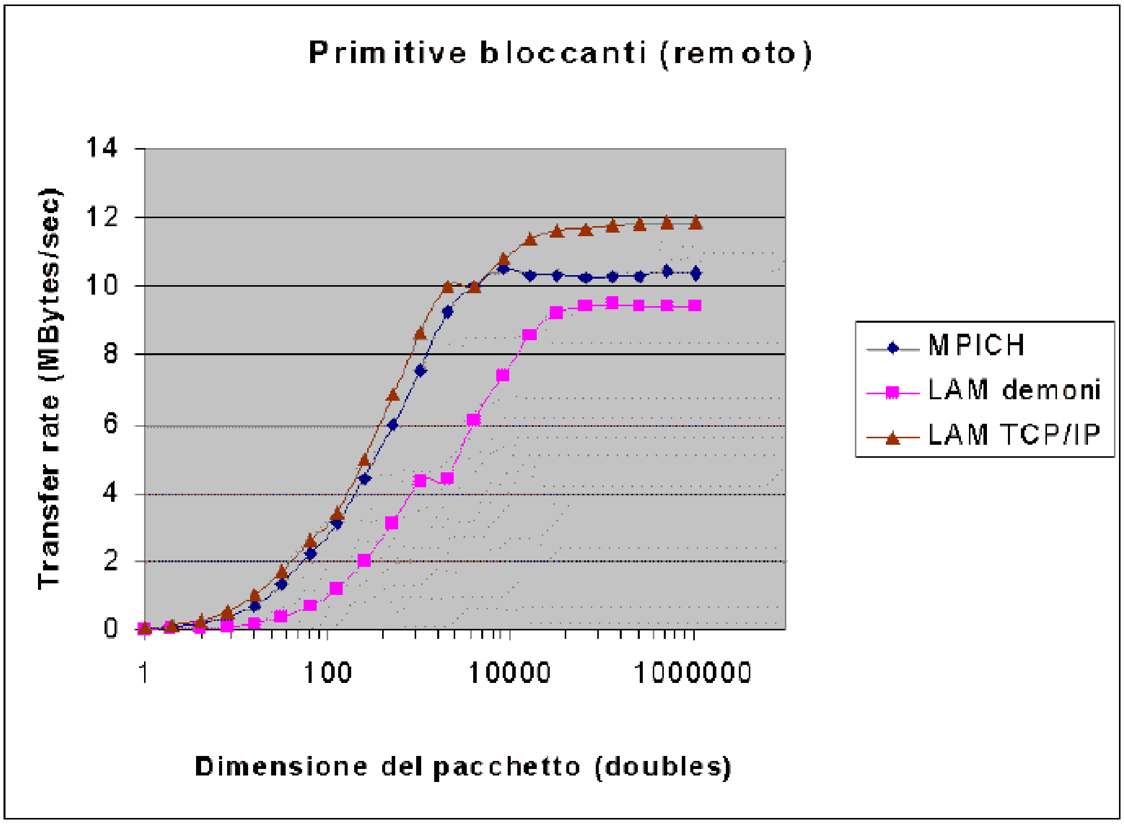

Figura 5.1: Pingpong remoto.

Dimensione (doubles)	MPICH (MB/s)	LAM via demoni (MB/s)	LAM via shared memory(MB/s)
1	0,072714	0,018878	0,433992
2	0,151905	0,036894	0,862485
4	0,302646	0,073428	1,732171
8	0,600804	0,14682	3,431963
16	1,193922	0,293741	6,806132
32	2,32898	0,587634	13,149461
64	4,466415	1,162229	25,34807
128	8,145455	2,26692	45,084224
256	13,607973	4,338951	77,303259
512	18,367713	7,999817	99,958584
1024	23,173975	13,213427	67,158917
2048	33,40265	16,031211	89,789594
4096	41,165829	17,940384	109,968163
8192	13,756507	19,062371	123,65709
16384	19,562985	19,462783	98,072993
32768	26,693549	18,790352	79,007097
65536	27,362246	18,672571	71,215532
131072	27,009144	18,444934	66,373966
262144	27,922005	17,8573	65,568842
524288	28,730471	17,456048	66,161937
1048576	28,867206	17,253233	64,948929

Tabella 5.3: Pingpong locale.

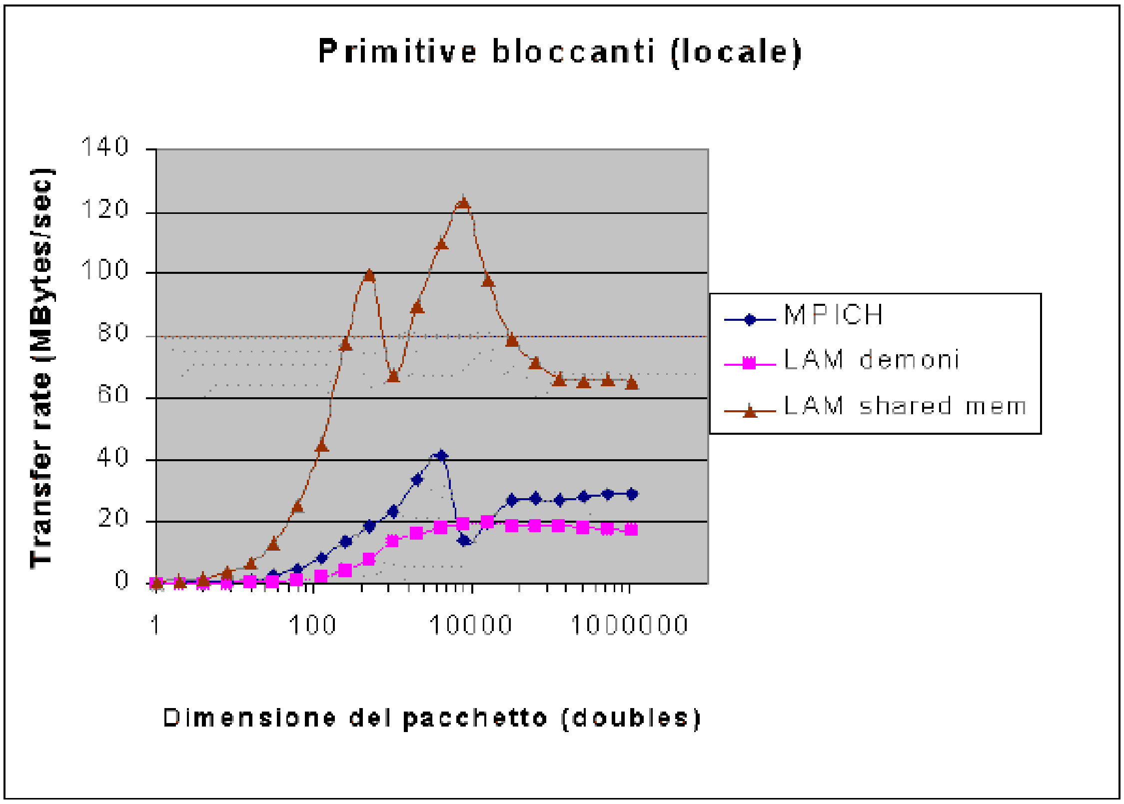

Figura 5.2: Pingpong locale.

Dimensione (doubles)	MPICH (MB/s)	LAM via demoni (MB/s)	LAM via TCP/IP(MB/s)
1	0,086494	0,008167	0,107674
2	0,164775	0,016354	0,212712
4	0,343083	0,032746	0,4141
8	0,673684	0,065628	0,79361
16	1,055951	0,130296	1,631922
32	2,027852	0,200336	2,795828
64	3,939472	0,398485	4,564652
128	6,323776	0,78644	6,131834
256	9,474171	1,584108	8,69055
512	15,142329	3,129074	11,587209
1024	15,98439	5,632156	15,545284
2048	17,857221	4,755843	18,74626
4096	14,91149	6,851646	10,160645
8192	12,061471	8,540024	9,58092
16384	11,775402	9,602336	11,465341
32768	11,232737	9,927054	11,654467
65536	11,018968	10,433694	11,444959
131072	10,925113	10,492736	11,805227
262144	10,871114	10,524648	11,833141
524288	10,85217	10,38617	11,81787
1048576	10,846775	10,347581	11,85504

Tabella 5.4: Pingpong remoto con primitive non bloccanti.

Numero di processi	MPICH (μs)	LAM (μs)
2	189	243
3	380	287
4	399	350
5	573	372
6	765	425
7	881	516
8	1431	665
9	1715	647
10	2133	707

Tabella 5.5: Tempi di esecuzione di una barriera.

chetto superiori; ancora la banda masssima ottenibile com LAM con comunicazioni client to client è superiore a quella di MPICH.

Barriera. La barriera è una funzione di comunicazione collettiva che serve a fornire un punto di sincronizzazione a tutti i processi appartenenti al medesimo comunicatore. Questo test invoca semplicemente la funzione MPI_Barrier e valuta il tempo necessario alla sincronizzazione di un numero variabile di processi.

Come si può vedere in tabella 5.5, se si eccettua il tempo per sincronizzare due soli processi, i tempi di sincronizzazione tra processi sono superiori per MPICH rispetto a LAM e il divario aumenta con il crescere del numero dei processi coinvolti nella barriera.

Le conclusioni che si possono trarre da questi semplici ma efficaci test di prestazioni indicano che la realizzazione di MPI con le prestazioni complessivamente migliori tra MPICH e LAM è LAM MPI purchè si utilizzi la comunicazioni di tipo client to client su di un cluster omogeneo dal punto di vista del sistema operativo, in questo caso infatti:

- la banda massima di LAM risulta sempre superiore di circa il 10 per cento rispetto a MPICH, avvicinandosi alla banda massima teorica permessa da Fast Ethernet (12.5 MegaByte al secondo);

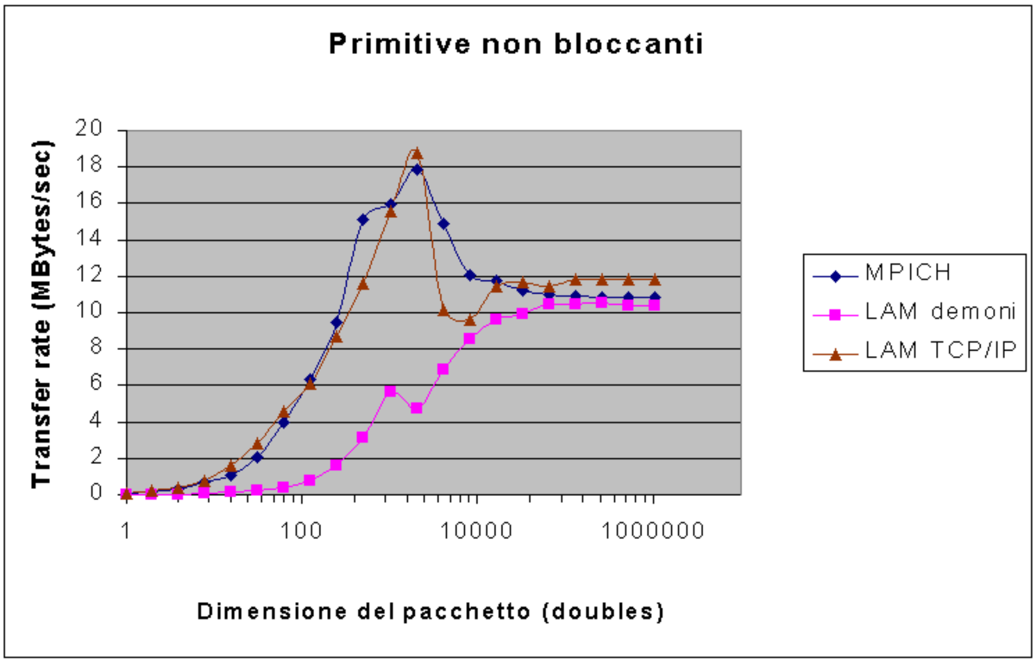

Figura 5.3: Pingpong con primitive non bloccanti.

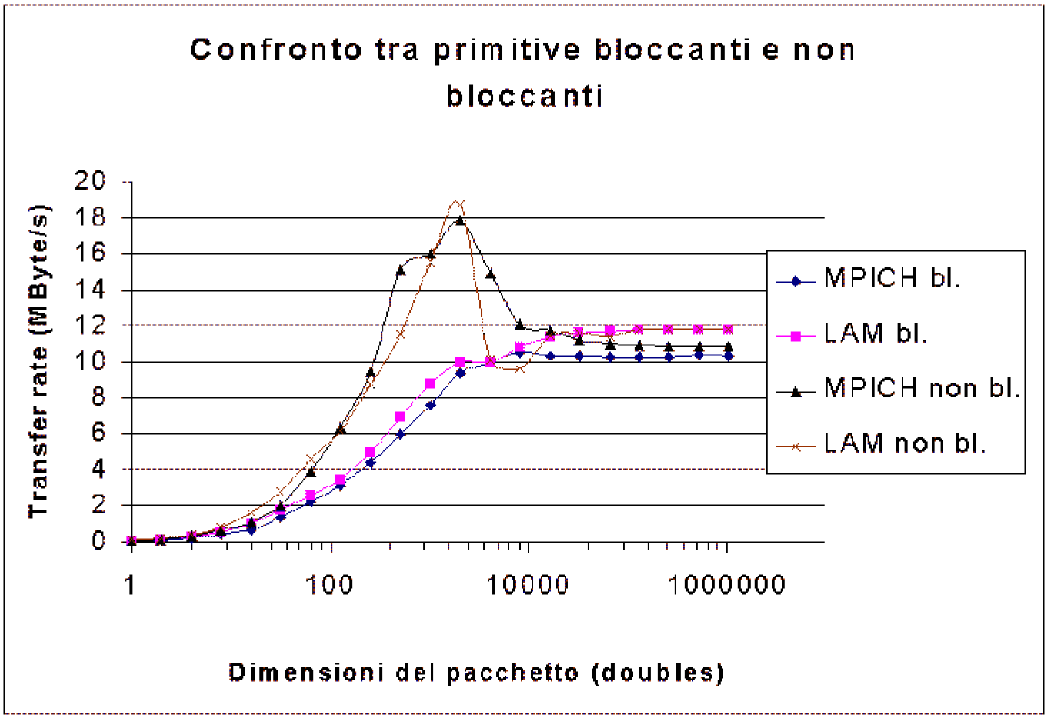

Figura 5.4: Confronto tra primitive bloccanti e non bloccanti.

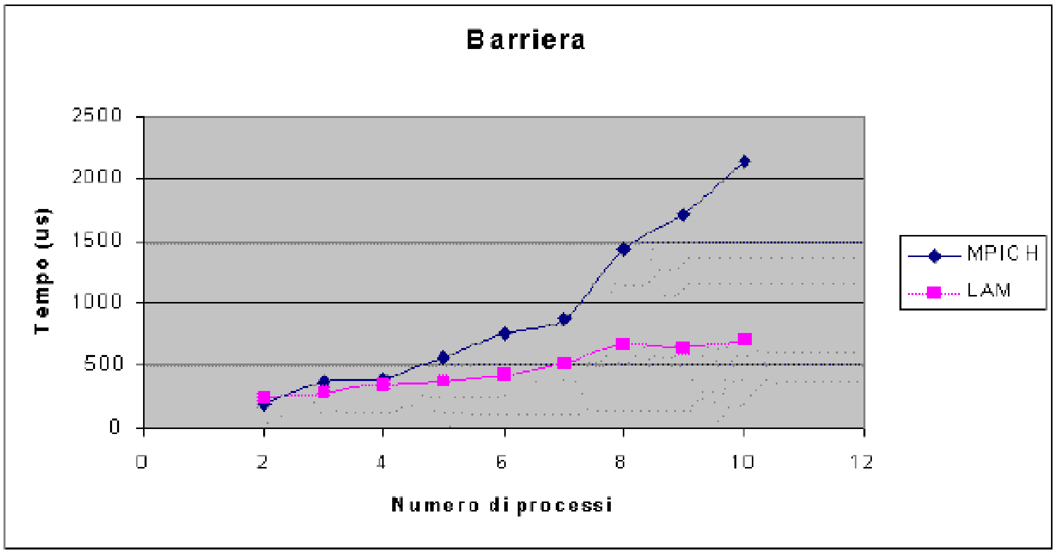

Figura 5.5: Tempi di sincronizzazione.

- la dimensione di pacchetto che consente di sfruttare la banda massima è molto simile per le due librerie;
- la velocità di trasmissione di LAM è superiore a quella ottenibile con MPICH per quasi tutto l'intervallo di dimensioni di pacchetto esaminato e nelle diverse situazioni;
- i tempi di sincronizzazione tra processi sono decisamente inferiori per LAM rispetto a MPICH, e questo è tanto più evidente quanto più grande è il numero di processi coinvolti;
- nel caso si abbia a disposizione un cluster costituito da nodi SMP, LAM attraverso l'uso delle comunicazioni tramite shared memory riesce a ottenere prestazioni dello stesso ordine di grandezza di quelle offerte da macchine parallele[5].

[5]Come ad esempio la macchina Origin Onyx2 della Silicon Graphics ospitata presso il centro di calcolo dell'Università di Parma.

Capitolo 6

Analisi di LAM MPI

6.1 Struttura software di LAM MPI

Nel capitolo precedente sono state determinate le prestazioni che LAM fornisce all'utente MPI; ora ci si sofferma sulla struttura del codice di LAM con particolare attenzione alle funzioni che svolgono le operazioni relative alle comunicazioni tra processi.

La libreria LAM MPI è costituita di due livelli: un Upper Layer che definisce le funzioni offerte dall'interfaccia MPI all'utente in termini di operazioni più elementari, comuni a tutti i tipi di hardware e sistema operativo e indipendenti dal sottostante Lower Layer che si occupa di realizzare effettivamente i trasferimenti di dati. I due livelli sono indipendenti e interagiscono tra loro tramite un'interfaccia costituita da otto funzioni che definiscono una Request Progression Interface (RPI). In tutte le operazioni che comportano uno scambio di dati di qualunque tipo l'Upper Layer invoca una delle otto funzioni.

Questa scelta nella progettazione del software LAM MPI ha permesso di utilizzare diverse modalità e protocolli per le operazioni di trasferimento dei dati, cosicchè per aggiungere una nuova tipologia di comunicazione non è necessario intervenire sull'Upper Layer, bensì è sufficiente riscrivere le otto funzioni dell'interfaccia RPI, intervenendo quindi solo sul Lower Layer.

Nello sviluppo del software di LAM infatti sono progressivamente stati introdotti nuovi protocolli di comunicazione. Fino alla versione LAM 6.0 era disponibile una sola tipologia di comunicazione e quindi anche una sola realizzazione dell'interfaccia RPI e del Lower Layer. Per poter comprendere questi meccanismi di comunicazione è necessario introdurre brevemente il

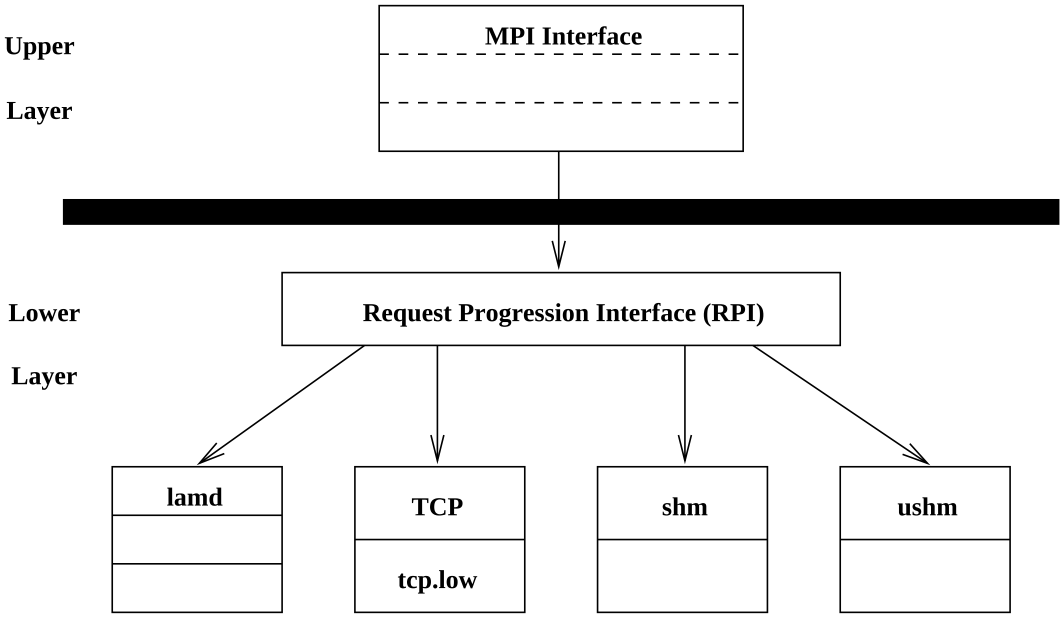

Figura 6.1: Struttura software della libreria LAM MPI.

funzionamento dei processi in LAM: prima di eseguire qualsiasi applicazione con LAM è necessario svolgere una procedura di avvio in cui su ogni nodo del cluster viene creato un demone composto da un nano-kernel e da una serie di thread che forniscono dei servizi al demone. Il nano-kernel offre delle semplici funzionalità di message passing ai processi locali che non comunicano mai direttamente tra loro; alcune delle thread di ogni demone si occupano invece delle comunicazioni tra demoni sui nodi diversi tramite un protocollo scalabile basato su UDP. Quindi la separazione dei compiti tra i vari componenti del demone è ben distinta: il nano-kernel si occupa delle comunicazioni intranodo, mentre è una thread che comunica via rete con le thread dei demoni sugli altri nodi e con il nano-kernel. Altre thread dei demoni svolgono altre attività come la gestione dei processi e l'accesso al filesystem.

Con la versione LAM 6.1 è stato introdotto un concetto di comunicazione diverso rispetto a quello utilizzato nella versione precedente: per velocizzare le comunicazioni tra i processi si è pensato di eliminare il controllo sulle comunicazioni svolto dal nano-kernel del demone adottando una comunicazione in cui ogni processo si connette a ciascun altro sfruttando TCP/IP come protocollo. Questa tipologia di comunicazione interprocesso viene chiamata client to client e indicata con la sigla *c2c*. La versione 6.1 di LAM offre due diverse realizzazioni di Lower Layer di tipo c2c: una con TCP/IP e una che integra TCP/IP con scambi di messaggi tramite memoria condivisa nel caso che i processi comunicanti si trovino sullo stesso nodo. La scelta del tipo di comunicazione da utilizzare viene eseguita dall'utente al momento dell'esecuzione del programma con l'indicazione di appositi flag nella riga di comando[1]; nell'interno del codice dell'Upper Layer l'invocazione di una funzione RPI è gestita da uno switch[2] che si preoccupa di scegliere la realizzazione della funzione chiamata corrispondente al tipo di comunicazione scelta.

Nella versione LAM 6.2 tuttora in fase di beta testing, è stata introdotta una nuova realizzazione dell'interfaccia RPI e del Lower Layer, destinata all'uso di shared memory. Sono dunque due le realizzazioni del Lower Layer per l'uso della shared memory che si differenziano tra loro per la modalità con cui si realizza la mutua esclusione alle zone di memoria condivisa: con semafori o strutture dette spin-locks. La scelta su quale tipo di comunicazione utilizzare viene effettuata in parte a tempo di compilazione tra le tipologie di comunicazione di tipo client to client, in parte a tempo di esecuzione tra l'adozione della comunicazione con demoni o client to client.

6.2 LAM Upper Layer

L'Upper Layer è la parte del codice di LAM che definisce le funzioni di interfaccia di cui l'utente programmatore di MPI può servirsi ed espande tali funzioni in una serie di operazioni finalizzate alla creazione di una lista concatenata di apposite strutture dati adatte all'invio dei messaggi di MPI[3].

Un esempio significativo è costituito dalla funzione MPI_Send[4]: essa è espansa in quattro funzioni di cui una, *lam_initerr_m*, si occupa di controllare che il processo chiamante la funzione sia stato inizializzato, la seconda, *lam_setfunc_m*, notifica attraverso una variabile globale il tipo di funzione che si sta per eseguire, la terza e più importante, *lam_send*, ha il compito di costruire la struttura *_req* da inviare, mentre la quarta, *lam_resetfunc_m*

[1] Nella versione LAM 6.1 indicando l'opzione -c2c si esclude la comunicazione tramite demoni a favore della diretta client to client che automaticamente usa la shared memory nel caso di processi residenti sul medesimo nodo.

[2] Realizzato tramite una macro e quindi facilmente modificabile.

[3] Si tratta della struttura _req che verrà in seguito descritta.

[4] MPI_Send è realizzata nel file *send.c* nella directory lam/share/mpi/.

riporta allo stato iniziale la variabile con la funzione eseguita indicata in precedenza. A sua volta la lam_send è espansa in otto funzioni[5] che costituiscono lo strato inferiore dell'Upper Layer, si occupano di costruire la lista di _req, e sfruttano le funzionalità del Lower Layer utilizzando l'interfaccia RPI:

_mpi_req_build: alloca la memoria per contenere una struttura _req, inizializzando alcuni campi contenenti i parametri passati direttamente da MPI_Send; inoltre stabilisce un eventuale buffer per le operazioni che lo richiedano.

_mpi_req_start: esegue eventuali operazioni di frammentazione dei dati in pacchetti trasferibili dalle funzioni del Lower Layer.

_mpi_req_add: aggiunge la richiesta in cima alla lista di _req.

_mpi_req_blkclr: riporta a zero il flag indicante richiesta bloccante.

_mpi_req_blkset: rende la richiesta bloccante.

_mpi_req_advance: avanza per quanto possibile le richieste che si trovano nella lista, chiamando la corrispondente funzione dell'interfaccia RPI *_rpi_c2c_advance*.

_mpi_req_rem: rimuove la richiesta completata dalla lista.

_mpi_req_destroy: distrugge la richiesta liberando la memoria occupata.

Per tutte le altre funzioni di comunicazione, la scomposizione in operazioni mirate a costruire la struttura _req risulta simile a quella effettuata dalla lam_send.

Funzioni invece che implicano il trasferimento di più messaggi tra processi sono spesso scomposte in termini di funzioni MPI più elementari: ad esempio MPI_Barrier è realizzata inviando opportunamente messaggi di lunghezza nulla con MPI_Send e ricevendo con MPI_Recv.

In altri casi manca la strutturazione in tre livelli osservata nella MPI_Send: il codice in cui viene definita una funzione richiama direttamente le funzioni che si occupano di costruire la lista di richieste. Per questa ragione nella figura 6.1 i livelli dell'Upper Layer sono stati indicati con una linea tratteggiata, per evidenziare che non sempre sono presenti.

[5]Definite in lamreqs.c.

6.3 LAM Lower Layer

L'interfaccia RPI (Request Progression Interface) rende disponibili funzioni che attraverso il Lower Layer devono eseguire i seguenti compiti:

- attivare e inizializzare le connessioni per il trasferimento dei dati tra i vari processi,
- trasferire i messaggi tra i processi,
- fornire dei meccanismi di sincronizzazione tra i processi,
- effettuare le disconnessioni una volta cessate le operazioni di trasmissione di dati.

Inoltre, per ogni singola operazione svolta da una delle funzioni del Lower Layer su di una richiesta, è necessario aggiornare e mantenere consistenti tutte le informazioni riguardanti lo stato dei dati coinvolti nei trasferimenti.

Non sono compiti di pertinenza del Lower Layer:

- la gestione dei buffer per le operazioni di send che lo richiedano;
- la frammentazione e deframmentazione dei messaggi;
- la conversione dei dati.

Queste competono all'Upper Layer.

Per effettuare le operazioni finalizzate a scambiare tra i processi i messaggi contenuti nella lista delle richieste, l'interfaccia RPI mette a disposizione dell'Upper Layer otto funzioni[6] che il progettista di un nuovo Lower Layer deve necessariamente realizzare [29]:

_rpi_c2c_init: questa funzione è chiamata da MPI_Init() e quindi effettua la inizializzazione delle operazioni relative alle comunicazione che devono essere compiute una sola volta nel corso dell'esecuzione di un programma MPI. Inoltre al suo interno chiama _rpi_c2c_addprocs.

_rpi_c2c_addprocs: è utilizzata per creare il complesso delle strutture che i processi dovranno usare per comunicare. È chiamata da _rpi_c2c_init

[6] Tutte le funzioni, con l'eccezione di _rpi_c2c_iprobe e _rpi_c2c_advance, restituiscono un intero: 0 nel caso di esecuzione corretta, LAMERROR, cioé -1 altrimenti.

affinchè l'inizializzazione delle comunicazioni interprocesso abbia luogo all'atto della chiamata a MPI_Init() cioé all'inizio della parte di codice che effettua chiamate alle librerie MPI; tuttavia può essere chiamata anche in altre occasioni dalle funzioni MPI_Spawn e MPI_Intercomm_create per aggiungere dinamicamente, cioé nel corso dell'esecuzione di un programma, altri processi. In entrambi i casi l'Upper Layer inserisce i processi creati dall'utente LAM nella lista dei processi poi chiama _rpi_c2c_addprocs.

_rpi_c2c_build: è invocata dalla funzione dell'Upper Layer _mpi_req_build che, dopo aver creato una nuova richiesta inizializzandone le variabili attinenti all'Upper Layer, chiama _rpi_c2c_build cosicchè anche il Lower Layer possa procedere a sua volta all'inizializzazione delle proprie variabili.

_rpi_c2c_start: la fase successiva all'inizializzazione di una richiesta consiste nel renderla pronta per essere trasmessa: il suo stato viene cambiato sia dall'Upper Layer che dal Lower Layer il quale deve aggiornare alcune informazioni. Inoltre questa funzione può avanzare in parte o del tutto la trasmissione di richieste particolari: ad esempio le richieste dirette al processo medesimo vengono completate.

_rpi_c2c_destroy: distrugge la parte riguardante il Lower Layer di un richiesta sia per quanto concerne variabili di stato introdotte dal Lower Layer, sia in relazione alla eventuale memoria dinamica allocata dal Lower Layer che deve ora provvedere a liberarla; è chiamata dall'analoga funzione dell'Upper Layer _mpi_req_destroy.

_rpi_c2c_advance: questa funzione svolge la quasi totalità del lavoro di comunicazione dal momento che cerca di avanzare per quanto possibile la lista delle richieste creata e aggiornata nelle fasi precedenti. Nel caso di richieste bloccanti la funzione non ritorna finchè almeno una richiesta bloccante della lista non sia stata completata.

È importante sottolineare che l'Upper Layer non si preoccupa delle modalità con cui i trasferimenti di dati avvengono, ciò che importa è che il Lower Layer aggiorni costantemente lo stato della richiesta. È anche possibile che una richiesta che sia già stata inserita nella lista venga rimossa dall'Upper Layer: di ciò il Lower Layer deve tenere conto, mentre non è consentito che una richiesta già in parte trasmessa venga cancellata.

_rpi_c2c_iprobe: è chiamata da MPI_Iprobe che si occupa di inizializzare e costruire una richiesta di prova non bloccante che deve verificare la presenza di una risposta che possa soddisfare la richiesta.

_rpi_c2c_finalize: esegue le operazioni di reset e pulizia finale di tutte le strutture utilizzate per le comunicazioni. È chiamata una sola volta all'interno di un programma dalla funzione MPI_Finalize che conclude la sessione di lavoro con la libreria MPI.

6.3.1 Strutture dati: richieste e processi

Ogni processo attivo in LAM MPI mantiene una lista concatenata delle richieste che devono essere soddisfatte: l'Upper Layer si preoccupa di costruire e mantenere questa lista ordinata in base ai cambiamenti di stato indotti dalle operazioni del Lower Layer.

Il ciclo di vita di una richiesta, rappresentata da una struttura *_req* si articola in quattro fasi:

Costruzione (Building). Viene allocata la memoria per contenere la struttura _req che è inizializzata con gli argomenti passati da una funzione MPI: processi mittente e destinatario, tipo e quantità di dati, tag. La richiesta è posta in uno stato di LAM_RQSINIT in cui non può essere avanzata in quanto non è ancora stata inserita nella lista delle richieste.

Attivazione (Starting). La richiesta viene inserita nella lista, il suo stato è aggiornato a LAM_RQSSTART e quindi da questo momento può essere eseguita.

Svolgimento (Progression). Non appena un trasferimento di dati relativo a una richiesta viene iniziato, lo stato della stessa richiesta passa a LAM_RQSACTIVE. Da questo momento il Lower Layer può eseguire le operazioni necessarie per trasmettere anche in parte la richiesta, finchè essa non sia completata; la funzione deputata a questo scopo è la _rpi_c2c_advance.

Completamento (Completion). La richiesta viene completata quando tutti i dati sono stati trasferiti ed eventuali messaggi di acknowledgment sono stati ricevuti o trasmessi: lo stato diventa LAM_RQSDONE. Una volta che è stata completata, la richiesta viene distrutta oppure, se persistente, ne vengono reinizializzate alcune variabili e lo stato è portato a LAM_RQSINIT.

Un processo può accedere a tutte le richieste della lista scandendone gli elementi con l'istruzione:

for(req=req_top; req; req=req->next)

Si analizza ora in dettaglio la struttura _req al fine di comprendere quali informazioni il Lower Layer debba aggiornare e quali dati debba trasmettere.

struct _req {

int rq_type;

Tipo di comunicazione della richiesta; sono possibili solo i tipi definiti di seguito. Questo campo non specifca se la richiesta sia bloccante o meno: è adibito a questa funzione il campo *rq_marks*. La richiesta di tipo *fake* viene usata solo dall'Upper Layer. La richiesta di tipo *probe* verifica la ricezione di un messaggio senza ritornare alcun dato.

```
#define LAM_RQISEND     0    /* standard send */
#define LAM_RQIBSEND    1    /* send bufferizzata */
#define LAM_RQISSEND    2    /* send sincrona */
#define LAM_RQIRSEND    3    /* ready send */
#define LAM_RQIRECV     4    /* generica receive */
#define LAM_RQIPROBE    5    /* probe request */
#define LAM_RQIFAKE     6    /* richiesta fake*/
```

int rq_state;

Stato della richiesta: il Lower Layer deve provvedere ad aggiornare questo campo in corrispondenza all'avanzamento della richiesta secondo quanto illustrato in precedenza:

```
#define LAM_RQSINIT     0    /* richiesta inizializzata */
```

```
#define LAM_RQSSTART    1    /* attiva non ancora iniziata */
#define LAM_RQSACTIVE   4    /* attive in completamento */
#define LAM_RQSDONE     3    /* richiesta completata */
```

int rq_marks;
Campo per le richieste persistenti; il Lower Layer non deve modificarlo, bensì limitarsi a gestire questo tipo di richieste:

```
#define LAM_RQFPERSIST  0x0001  /* richiesta persistente */
#define LAM_RQFDYNBUF   0x0004  /* buffer dinamico */
#define LAM_RQFDYNREQ   0x0008  /* richiesta dinamica */
#define LAM_RQFSOURCE   0x0200  /* sorgente */
#define LAM_RQFDEST     0x0400  /* destinazione */
#define LAM_RQFBLKTYPE  0x0800  /* richiesta persistente
                                   e bloccante */
#define LAM_RQFOSORIG   0x1000  /* origine per richieste
                                   one-sided */
#define LAM_RQFOSTARG   0x2000  /* destinazione per richieste
                                   one-sided */
```

int rq_flags;
Flag per memorizzare le caratteristiche non persistenti di una richiesta; non deve essere modificato dal Lower Layer tranne nel caso in cui la richiesta provochi un errore di tipo *MPI_ERR_TRUNCATE* cioé i dati siano ricevuti in modo parziale.

```
#define LAM_RQFCANCEL  0x0002     /* richiesta cancellata */
#define LAM_RQFBLOCK   0x0010     /* richiesta bloccante */
#define LAM_RQFTRUNC   0x0040     /* dati troncati */
#define LAM_RQFORPHAN  0x0100     /* cancella se completata */
#define LAM_RQFCHAR    0x0100000  /* raw data */
#define LAM_RQFINT     0x0200000  /* intero */
#define LAM_RQFFLOAT   0x0400000  /* float */
#define LAM_RQFDOUBLE  0x0800000  /* double */
#define LAM_RQFSHORT   0x01000000 /* short int */
```

char *rq_packbuf;

Puntatore all'area di memoria contenente i dati da inviare o destinata ai dati da ricevere. In dipendenza dal tipo di operazione di comunicazione può coincidere o meno con *rq_buf*.

int rq_packsize;

Dimensione del buffer precedente.

int rq_count;

void *rq_buf;

MPI_Datatype rq_dtype;

int rq_rank;

int rq_tag;

MPI_Comm rq_comm;

Parametri delle funzioni MPI; non sono modificati dal Lower Layer ma solo utilizzati per gestire le comunicazioni.

int rq_cid;

Identificativo del contesto di un processo; deve essere inviato nell'envelope; corrisponde al comunicatore a cui appartiene il processo.

int rq_func;

Tipo di funzione MPI che ha generato la richiesta.

int rq_seq;

Identificativo del numero di sequenza della richiesta; impostato dall'Upper Layer; deve essere trasmesso nell'envelope del messaggio per consentire l'analisi delle trasmissioni di messaggi da parte di tools di LAM.

int rq_f77handle;

Handle per Fortran 77.

MPI_Status rq_status;
Struttura contenente lo stato della richiesta. È necessario che il Lower Layer aggiorni i campi relativi alla sorgente del messaggio, alla sua effettiva dimensione e al suo tag nel caso che la richiesta sia una MPI_Recv.

struct _bsndhdr *rq_bsend;
Puntatore all'header del buffer utilizzato dall'Upper Layer per le send bufferizzate.

struct _proc *rq_proc;
Puntatore a una struttura di tipo _proc riferita al secondo processo coivolto nella comunicazione; è inizializzato dall'Upper Layer; nel caso di una operazione di Receive disposta ad accettare messaggi provenienti da qualsiasi processo viene posto a 0, e una volta che il messaggio è stato effettivamente ricevuto il Lower Layer può, ma non è necessario, aggiornarlo con il processo mittente.

struct _req *rq_next;
Puntatore alla prossima richiesta nella lista. Non deve essere modificato dal Lower Layer; è l'Upper Layer che si occupa della gestione della lista delle richieste.

void *rq_extra;
Per ora non utilizzato.

int (*rq_hdlr)();
Puntatore a funzione; per ora non utilizzato.

union rpi_req rq_rpi;
Puntatore a struttura per consentire di aggiungere alla richiesta tutti i dati relativi al Lower Layer utilizzato.
};

Il Lower Layer ha il compito, come visto con l'analisi della struttura _req, di aggiornare e intervenire su alcune variabili utilizzate anche dall'Upper Layer; esistono invece altre variabili nella struttura _req di pertinenza

esclusiva del Lower Layer, contenute in *rq_rpi*: tramite la *union* è possibile scegliere il tipo di struttura da aggiungere alla _req in base al tipo di Lower Layer e quindi di meccanismo di comunicazione usato, in particolare si può optare per una struttura tipo *lamd_req* per la comunicazione tramite demoni o di tipo *c2c* per la comunicazione diretta tra processi via TCP/IP o shared memory, come indicato:
union rpi_req {
struct lamd_req lamd;
struct c2c_req c2c;
};

Il vantaggio della scelta di questo tipo di struttura dati consiste nel fatto che nel caso si voglia aggiungere un nuovo livello di trasporto non è necessario intervenire direttamente sulla struttura _req ma si possono inserire i dati relativi al nuovo Lower Layer nella union rpi_req.

Si analizzano brevemente i dati relativi alla comunicazione diretta tra processi, attinenti prevalentemente ai buffer utilizzati dal Lower Layer per ricevere e immagazzinare i dati da scambiare:

struct c2c_req {

int cq_state;

```
#define C2CDONE        1       /* trasferiti tutti i dati */
#define C2CWRITE       2       /* scrittura su una socket */
#define C2CREAD        3       /* lettura da una socket */
#define C2CSENDSELF    4       /* invio a se stesso */
```

int4 cq_peer;
Rank del processo peer, cioé il processo con cui si desidera effettuare la comunicazione.

struct c2c_envl cq_env;
Envelope del messaggio.

struct c2c_envl cq_outenv;
Envelope da spedire in formato byte dopo byte pronto per l'invio.

int cq_nenvout;
Numero di byte di envelope rimasti da inviare.

int cq_nmsgout;
Numero di byte del messaggio rimasti da inviare.

char *cq_envbuf;
Buffer per ricevere un envelope in arrivo.

char *cq_msgbuf;
Puntatore al messaggio da spedire o ricevere.

int cq_bufoff;
Offset del messaggio in un buffer condiviso.

int cq_bufsize;
Dimensione del buffer condiviso.

int (*cq_adv)();
Funzione usata per avanzare la richiesta da parte del Lower Layer; varia a secondo dello stato della richiesta e delle comunicazioni.
};

Un processo MPI è rappresentato nel codice da una struttura di tipo **_proc**. Ogni processo mantiene una lista concatenata di tutti i processi di cui è a conoscenza e con cui può comunicare; il processo stesso è presente nella lista e la variabile globale *lam_myproc* è un puntatore al processo stesso.

La lista non è trasparente al Lower Layer che può solo scandirla utilizzando le funzioni *lam_topproc()* e *lam_nextproc()*, che restituiscono rispettivamente il primo e il prossimo processo della lista, in questo modo:

for (p=lam_topproc(); p; p=lam_nextproc())

Le informazioni necessarie per definire univocamente un processo sono indicate nella struttura:
struct _proc {

struct _gps p_gps;
Global Positioning System (GPS) per identificare in modo univoco il nodo sul quale il processo è in esecuzione.

int p_ger_nsnd;
Utilizzato solo dai demoni LAM. Non deve essere modificato o utilizzato da altri tipi di Lower Layer. Rappresenta il numero di messaggi inviati.

int p_mode;
Flag per memorizzare informazioni di stato. Il Lower Layer può usarlo ma non ha la possibilità di scrivere gli 8 bit meno significativi che sono riservati all'Upper Layer.

```
#define LAM_PFLAG       0x001       /* valore di default */
#define LAM_PDEAD       0x002       /* nodo non attivo */
#define LAM_PRPIINIT    0x004       /* RPI inizializzato */
#define LAM_PCLIENT     0x008       /* processo client */
#define LAM_PHOMOG      0x010       /* ambiente omogeneo */
```

int p_refcount;
Non ancora utilizzato.

union rpi_proc p_rpi;
Informazioni relative al processo specifiche del Lower Layer utilizzato.
};

Il GPS è il sistema per la localizzazione del processo all'interno dei diversi nodi del cluster abilitati dall'utente in fase di avvio di una sessione di lavoro con LAM, iniziata con l'esecuzione del programma *lamboot*. I dati necessari per una completa individuazione della posizione di un processo sono contenuti in una apposita struttura:
struct _gps{

int4 gps_node;
Identificativo del nodo assegnato dall'Upper Layer all'avvio.

int4 gps_pid;
PID del processo come assegnato dal sistema operativo su ogni nodo.

int4 gps_idx;
Indice del processo assegnato dall'Upper Layer.

int4 gps_grank;
Rank del processo come assegnato da LAM MPI.
};

Similmente a quanto accade per la struttura _req, anche per la struttura _proc è possibile aggiungere le informazioni specifiche del Lower Layer; per ora questo campo è utilizzato solo per la comunicazione di tipo client to client:

union rpi_proc {
union struct c2c_proc c2c;
};

struct c2c_proc {

struct c2c_envl cp_env;
Envelope in arrivo dal processo con cui si sta comunicando.

int cp_sock;
Identificativo per la socket usata per le comunicazioni.

int cp_nenvin;
Numero di byte rimasti da leggere nell'envelope in arrivo.

char *cp_envbuf;
Puntatore al buffer destinato a contenere i dati in arrivo.

int cp_nmsgin;
Numero di byte rimasti da leggere del messaggio in arrivo.

char *cp_msgbuf;
Puntatore al buffer per il messaggio.

int cp_extra;
Numero di byte letti dopo un messaggio ricevuto parzialmente.

MPI_Request cp_wreq;

MPI_Request cp_rreq;

MPI_Request cp_mreq;
Puntatori a una richiesta usata rispettivamente per inviare un messaggio al processo, per ricevere un messaggio dal processo, per corrispondere a una delle due richieste precedenti.

int (*cp_readfn)();
Puntatore alla funzione da usare per proseguire la comunicazione.

struct cbuf_msg *cp_bmsg;
Struttura utilizzata per il buffering del messaggio.

int cp_nbfde;
Numero di envelope bufferizzati.
};

Capitolo 7

Progetto e realizzazione di LAM su M-VIA

7.1 Comunicazioni asincrone con M-VIA

I due modelli di comunicazione supportati da M-VIA, cioé Send/Receive e operazioni di RDMA, non si prestano per realizzare, senza l'uso di un opportuno protocollo, tutte le tipologie di trasferimento dei dati utilizzabili in MPI.

MPI infatti offre al programmatore operazioni di trasferimento dati asincrone e sincrone, bloccanti e non bloccanti; si possono quindi progettare più tipologie di protocolli per lo scambio di dati, trattando in modo diverso ogni operazione, cioé si utilizza un protocollo per l'invio e la ricezione sincrono e bloccante di dati, uno per l'invio e la ricezione asincrono non bloccante, e così per gli altri casi. Il vantaggio di questa scelta risiede nella capacità di svolgere ogni diverso tipo di trasferimento di dati con il protocollo più adatto, quindi presumibilmente con la maggiore efficienza possibile; lo svantaggio risiede nel maggior tempo di sviluppo e nella più difficile integrazione nel codice di LAM.

Un approccio alternativo consiste nel progetto di un protocolllo asincrono flessibile, che integri la possibilità di stabilire momenti di sincronizzazione se richiesti e di fornire eventualmente notifica del completamento delle operazioni. Probabilmente la scelta migliore è quella di utilizzare un unico protocollo asincrono adattabile a tutte le tipologie di operazioni, riservandosi l'uso di protocolli specifici solo in casi particolari.

Con il termine asincrono in questo capitolo si indica un modello di comunicazione in cui ogni processo deve poter iniziare la trasmissione di dati in qualunque momento, senza sapere se il processo destinatario sia pronto a ricevere o meno; al più al processo mittente può essere eventualmente notificato il fallimento dell'operazione. Per quanto riguarda la ricezione ogni processo deve potersi predisporre a ricevere dati senza conoscere se e quando avverrà effettivamente il trasferimento di dati.

Le operazioni Send/Receive di VIA rispondono a un modello di comunicazione sincrono: un processo che effettui una Send per inviare dati deve preventivamente assicurarsi che il ricevente sia in ascolto, cioé che abbia postato un descrittore sulla Receive Queue della VI connessa con il mittente. In più il descrittore postato deve contenere l'indirizzo di un buffer di dimensione sufficiente a contenere i dati in arrivo, altrimenti la Send viene completata con un errore.

Un processo che invece desideri ricevere dati, dopo avere postato un opportuno descrittore sulla Receive Queue, può verificare il completamento della ricezione con funzioni dalla semantica bloccante o non bloccante; tuttavia anche in questo caso è necessaria una sincronizzazione in quanto un dato inviato da un processo ad un destinario che al momento della trasmissione abbia una Receive Queue vuota o con descrittori dalla dimensione insufficiente, non solo viene perso senza possibilità di un futuro recupero con una receive successiva, ma causa anche un errore (*asyncronous error*).

La comunicazione è diversa nel caso di operazioni di tipo RDMA: in questo caso il processo che effettui una scrittura o lettura RDMA specifica gli indirizzi dei dati sia locali che remoti; l'operazione RDMA viene completata senza bisogno di alcun intervento da parte del processo destinatario remoto, quindi in modo asincrono tra i processi. Tuttavia non viene notificato a nessun processo il completamento di una operazione RDMA.

Gli RDMA offrono quindi un modello di comunicazione opposto al caso Send/Receive: un trasferimento di dati di tipo RDMA non richiede una sincronizzazione iniziale ma non è in grado di fornire informazioni sul completamento dell'operazione. Scambi di dati Send/Receive invece richiedono sincronizzazione iniziale tra i processi, ma possono fornire la notifica del completamento delle operazioni.

Costituisce una importante eccezione la scrittura RDMA con Immediate Data, la quale, oltre a indicare gli indirizzi locali e remoti dei dati, permette il trasferimento di un numero intero che viene depositato in uno specifico campo di un descrittore che deve essere stato in precedenza postato dal processo ricevente: questo meccanismo da un lato introduce la possibilità di inviare la notifica del completamento di una scrittura RDMA, dall'altro però richiede una preventiva sincronizzazione come nel caso di una Send.

Comunque le operazioni RDMA richiedono un momento iniziale di sincronizzazione che deve essere svolto dai processi comunicanti una sola volta prima di potere effettuare RDMA: poichè l'istanziatore di un RDMA specifica gli indirizzi non solo dei propri dati, ma anche di quelli del processo destinatario, esso deve essere a conoscenza delle locazioni di memoria che il destinatario ha messo a disposizione per ricevere o inviare dati attraverso RDMA; questo scambio di indirizzi, utilizzando solo le operazioni di VIA, può avvenire unicamente con Send e Receive, richiedendo quindi sincronizzazione.

Con l'obiettivo di costruire un protocollo asincrono flessibile come indicato in precedenza, utilizzando in diversi modi i due tipi di operazioni di VIA per il trasferimento di dati, si sono studiate diversi protocolli che possono essere suddivisi in tre famiglie: quelli basati sul pre-posting di descrittori, quelli che combinano Send/Receive e RDMA Write, quelli che utilizzano RDMA Read.

L'obiettivo del protocollo di comunicazione, definibile come l'insieme di regole con cui due o più entità si scambiano messaggi, è quello di fornire un servizio di trasmissione di dati affidabile; per quest'ultimo compito bisogna gestire:

- la perdita di pacchetti,
- la duplicazione di pacchetti,
- la sincronizzazione tra le entità comunicanti,

Per quanto riguarda i primi due problemi le specifiche di VIA prevedono tre diversi livelli di affidabilità: unreliable, reliable delivery e reliable reception; tutti e tre i livelli evitano la duplicazione dei pacchetti, mentre solo gli ultimi due garantiscono la consegna ordinata dei pacchetti. A livello di applicazione software quindi è sufficiente gestire la sincronizzazione tra i processi comunicanti e garantire che le risorse allocate per la comunicazione in ogni istante non siano esaurite: il primo problema è risolto dal protocollo di comunicazione descritto in seguito, il secondo costituisce l'obiettivo del controllo di flusso che sarà oggetto di un successivo paragrafo.

Tuttavia attualmente M-VIA non supporta tutti i tre livelli di affidabilità, bensì solo l'unreliable che non recupera situazioni di perdita di pacchetti per problemi della rete o overflow del buffer della scheda di rete. Per risolvere tale problema è necessario procedere alla rilevazione e correzione di errore via software, in modo da costruire un protocollo affidabile al di sopra di M-VIA non affidabile: si è scelto per quanto concerne il lavoro di questa tesi di non realizzare il controllo di errore in quanto i piani di sviluppo di M-VIA prevedono l'integrazione del livello reliable. I test e le prove di funzionamento sono però state svolte usando la corrente versione di M-VIA, assumendo bassa la probabilità di perdita di un pacchetto, ipotesi verificata con discreta approssimazione sul cluster ParMa^2.

7.1.1 Registrazione dei buffer

Prima di descrivere e confrontare le caratteristiche delle tre tipologie di protocolli occorre affrontare il problema di come e quando registrare i buffer per l'invio e la ricezione dei dati.

Poichè ogni dato può essere mandato soltanto per mezzo di un descrittore in cui deve essere indicato l'indirizzo della zona di memoria contenente i dati da spedire assieme al gestore di tale memoria, è indispensabile che i buffer per i dati da inviare siano registrati grazie all'intervento del Kernel Agent. Questa operazione è dispendiosa in termini di tempo e di risorse di calcolo perché implica una transizione da user mode a kernel mode, ed è quindi sconsigliato inserirla nelle operazioni di invio dei dati, critiche dal punto di vista della latenza.

Si può in alternativa pre-registrare un numero di buffer di dimensione fissa destinati a contenere i dati da inviare: in tal caso si elimina il ricorso al Kernel Agent durante le comunicazioni, ma si introduce nuovo overhead a causa della copia che spesso è necessario fare tra il buffer contenente i dati da spedire e il buffer registrato in precedenza.

Quindi si hanno tre possibilità per i buffer di invio:

- Registrare i dati al momento della loro allocazione in memoria, senza bisogno di buffer intermedi.
- Registrare al di fuori delle operazioni di comunicazione alcuni buffer in cui all'occorrenza sono copiati i dati usati effettivamente nell'applicazione.

- Registrare i dati al momento della comunicazione.

La prima alternativa è senza dubbio la migliore in quanto evita copie e intervento del Kernel Agent durante la comunicazione, introducendo solo un tempo aggiuntivo al momento dell'allocazione della memoria. Tuttavia non sempre è di facile applicazione, come nel nostro caso di integrazione di LAM MPI su VIA, in quanto questo progetto si propone di riscrivere solo il Lower Layer di LAM il quale utilizza strutture dati create e allocate dall'Upper Layer di LAM. In generale quindi la registrazione dei buffer al momento dell'allocazione della memoria è di difficile applicazione in grandi progetti software suddivisi in diversi livelli con strutture dati comuni.

La pre-registrazione dei buffer in cui vengono copiati i dati da spedire al momento dell'invio non presenta la difficoltà di utilizzo della registrazione al momento dell'allocazione; tuttavia riduce le prestazioni proprio perché introduce una copia di dati non necessaria, e contraddice la caratteristica di VIA che si propone come protocollo di comunicazione zero-copy, cioé permette lo scambio di dati senza l'uso di buffer intermedi ma tramite la registrazione di buffer. Inoltre la pre-registrazione dei buffer presenta anche il problema delle dimensioni dei buffer stessi, indicate al momento della creazione e della registrazione e che devono essere sufficienti a contenere i dati da inviare.

Si possono utilizzare buffer:

- di dimensione fissa ed elevata in piccolo numero,
- di dimensione fissa e limitata in grande quantità,
- di diverse dimensioni.

Nel primo caso si cerca di fare in modo che ogni possibile messaggio da spedire possa essere contenuto in un singolo buffer; non è quindi necessario un gran numero di buffer di invio poichè una volta terminata l'operazione di invio il buffer può essere riutilizzato. Diverso è il discorso per i buffer di ricezione, che non possono essere riusati finchè il descrittore completato sulla Receive Queue non viene consumato dal processo ricevente.

Gli svantaggi risiedono nella grande quantità di memoria fisicamente contigua che il procedimento di registrazione riserva e blocca, nella necessità di registrare al momento della comunicazione eventuali buffer di dimensione maggiore di quella dei buffer preregistrati. Non influisce invece sulle prestazioni il fatto di inviare dati di dimensioni molto piccole in buffer dalle dimensioni elevate in quanto gli effettivi byte da trasferire sono indicati in un apposito campo del descrittore utilizzato.

Nel caso di buffer di dimensioni ridotte, si effettua una eventuale operazione di frammentazione dei messaggi in blocchi pari alla dimensione dei buffer preregistrati; si aggiunge quindi complessità al protocollo di comunicazione, però rispetto al caso precedente non si ha bisogno di grandi quantità di memoria riservate ai buffer. Particolarmente delicata è la scelta della dimensione dei buffer, che non deve essere troppo piccola da non riuscire a sfruttare l'intera banda a disposizione e troppo grande per evitare di riservare molta memoria come nel caso precedente.

In alternativa si possono usare buffer di varie dimensioni e combinare insieme le due politiche precedenti in base alla dimensione del messaggio; in tal modo si può migliorare il rapporto tra memoria riservata e overhead dovuto a computazioni aggiuntive.

La scelta migliore dipende fortemente dalla conoscenza della distribuzione dei messaggi in funzione della loro dimensione.

La registrazione del buffer al momento dell'invio dei dati è l'operazione più semplice e sempre applicabile, ma fornisce, a causa dell'intervento del Kernel Agent, le peggiori prestazioni.

Queste considerazioni si applicano ai buffer di invio nel caso di Send o di RDMA Write e ai buffer di ricezione locali nel caso di RDMA Read. Per quanto riguarda i buffer di ricezione per Send o RDMA Write o per quelli di invio per RDMA Read, poichè in generale non si conoscono le dimensioni dei dati che verranno ricevuti, è sempre necessario pre-registrare delle zone memoria destinate alla ricezione; tali operazioni di registrazione sono eseguite al di fuori dei momenti critici della ricezione e trasmissione.

Per quanto riguarda la dimensione dei buffer la situazione è diversa nel caso di RDMA o di Receive. Una operazione di RDMA richiede al mittente la conoscenza dell'indirizzo della zona di memoria remota: non è quindi conveniente utilizzare buffer remoti diversi per ogni singola comunicazione che richiederebbe una preventiva comunicazione sincrona per trasmettere l'indirizzo; è più efficiente utilizzare una o comunque poche zone di memoria destinate a RDMA i cui indirizzi siano comunicati una volta per tutte in fase di inizializzazione. Le operazioni di tipo RDMA quindi sembrano adatte a creare una ampia zona di memoria in cui due processi possano scambiarsi messaggi di dimensione variabile senza grande spreco di memoria.

Per le operazioni di receive, un descrittore postato su di una Receive

Queue contiene l'indirizzo di un buffer preregistrato per i dati in arrivo: esso deve essere sufficientemente capiente per contenere il messaggio inviato, quindi spesso può accadere che gran parte dello spazio riservato non risulti in effetti utilizzato. Inoltre non è possibile riusare lo stesso buffer per descrittori diversi a meno che essi non siano mai postati contemporaneamente sulla Receive Queue; quindi l'eventuale spreco di memoria per un messaggio deve essere moltiplicato per il massimo numero di descrittori che si intende postare allo stesso tempo sulla RQ. Volendo invece diminuire lo spazio lasciato inutilizzato si deve diminuire la dimensione dei buffer indicati nei descrittori, ma ciò può significare maggiore probabilità che qualche messaggio non sia contenuto nel descrittore sulla RQ, causando un errore che deve essere rilevato e gestito. L'uso di Receive è consigliato nel caso di messaggi di cui si conosca in anticipo la dimensione.

Dunque l'utilizzo dei buffer in ricezione dipende dal tipo di operazione effettuata: nel caso generale di non conoscenza delle dimensioni dei messaggi in arrivo gli RDMA consentono un utilizzo della memoria destinata alla ricezione meno dispendiosa in termini di dimensioni anche se più costosa dal punto di vista computazionale in quanto necessita di politiche di gestione delle zone destinate a RDMA.

7.1.2 Protocolli basati sul pre-posting di descrittori

Il problema più consistente nel trasferimento di dati tramite Send è la necessità di avere descrittori pronti alla ricezione postati sulla Receive Queue del processo destinatario: per assicurarsi che qualsiasi operazione di Send sia completata correttamente ogni processo posta sulla propria, o sulle proprie, Receive Queue un numero elevato di descrittori prima di effettuare la connessione a una VI qualsiasi. Tale operazione è permessa da VIA, mentre ogni Send postata sulla Send Queue di una VI non connessa viene completata con un errore: si ha dunque la certezza che almeno un numero di Send pari ai descrittori pre-postati dal processo destinario sulla propria Receive Queue verrà completato senza errori dovuti a code vuote.

Il parametro più delicato risulta quindi essere il numero di descrittori da pre-postare sulla Receive Queue: esso dipende fortemente, oltre che dal tipo di applicazione, dalla sequenza con cui le operazioni dei due processi comunicanti si svolgono. Infatti se un processo effettua molte operazioni di Send mentre il ricevente è bloccato o sta facendo qualcos'altro si corre il rischio di svuotare una Receive Queue; questo problema può essere affrontato scrivendo in un secondo momento un controllo di flusso aggiuntivo.

Per quanto riguarda il riuso dei descrittori, essi possono essere inseriti in una lista circolare, oppure in un array. Ogni volta che il processo destinatario di un messaggio abbia completato una operazione di Receive togliendo quindi dalla cima della coda un descrittore, esso viene reinizializzato impostando il campo contenente lo stato e la dimensione dei dati da ricevere, e inserito sul fondo della Receive Queue. In tal modo il processo può mantenere la propria Receive Queue ragionevolmente piena, anche se non devono essere escluse condizioni di coda piena di descrittori completati e non ancora consumati; tali situazioni verranno, come detto, gestite dal controllo di flusso.

La seconda importante scelta da effettuare riguarda il tipo di operazione da usare per inviare i dati: le possibilità sono due, Send o RDMA Write con Immediate Data, mentre bisogna scartare RDMA Write senza Immediate Data e RDMA Read in quanto, non consumando descrittori sulla Receive Queue, impediscono la notifica al destinatario del completamento dell'invio di dati, e questo rende problematica la realizzazione delle funzioni MPI di receive bloccanti.

L'utilizzo di Send implica che i dati trasmessi siano immagazzinati nei buffer indicati nei descrittori postati sulla Receive Queue; valgono in questo caso le considerazioni svolte per la pre-registrazione dei buffer di ricezione con Send: se non si conoscono a priori le dimensioni dei messaggi da ricevere si può incorrere in un pessimo utilizzo della memoria. Poichè in questa applicazione la dimensione dei blocchi di dati inviati non è nota a priori e varia in un ampio intervallo, la scelta più opportuna è l'uso di RDMA Write con Immediate Data; di conseguenza, come zona per ricevere i dati, è stata scelta una vasta area di memoria registrata all'atto della sua allocazione.

Quindi in questo caso il protocollo più efficiente basato sul pre-posting di descrittori sulla Receive Queue presenta le seguenti caratteristiche:

- Registrazione e preposting di un gran numero di descrittori sulla Receive Queue.
- Registrazione di un piccolo numero di descrittori da postare sulla Send Queue.
- Utilizzo di RDMA Write con Immediate Data per inviare dati.
- Preregistrazione di una vasta zona di memoria per ricevere dati.

- Preregistrazione di alcuni buffer di grandi dimensioni per trasmettere i dati.

Le fasi attraverso cui un processo comunicante secondo questo protocollo transita, indicate nello schema a blocchi 7.1, sono:

Apertura del VI NIC: è necessario per accedere a qualsiasi funzione di VIA.

Creazione della VI: il processo crea una VI per ogni processo a cui si dovrà connettere.

Pre-posting dei descrittori sulla RQ: deve essere effettuato sulla RQ di ogni VI creata prima della connessione.

Connessione della VI: per potere comunicare con i processi, dal momento che VIA è orientato alla connessione.

Scambio degli indirizzi: poichè è stato scelto di inviare messaggi tramite RDMA Write, è necessario conoscere l'indirizzo della memoria appartenente al processo remoto.

Send o receive: sono le operazioni in cui si scambiano i messaggi.

Per comprendere l'esatto funzionamento delle operazioni di send e receive è indispensabile introdurre alcune variabili che ogni processo deve avere in quantità pari al numero di processi[1] a cui è connesso (figura 7.2):

RDMAArea: è il puntatore alla zona di memoria registrata per ricevere dati dal peer. Inoltre il processo possiede l'indirizzo e il gestore della RDMAArea del peer. In questo modo i processi possono comunicare tra loro in modo bidirezionale tramite operazioni RDMA.

RDMAwritepointer: indica quanti byte di dati sono stati finora scritti nella RDMAArea del peer a partire dal suo inizio.

RDMAreadpointer: indica il numero di byte finora letti nella RDMAArea del processo[2].

L'invio di messaggi da parte di un processo connesso a un peer secondo lo schema 7.1, avviene nel modo seguente (figura 7.3):

[1] Chiameremo uno generico tra questi processi peer.

[2] Si noti che è la RDMAArea del processo stesso e non quella del peer.

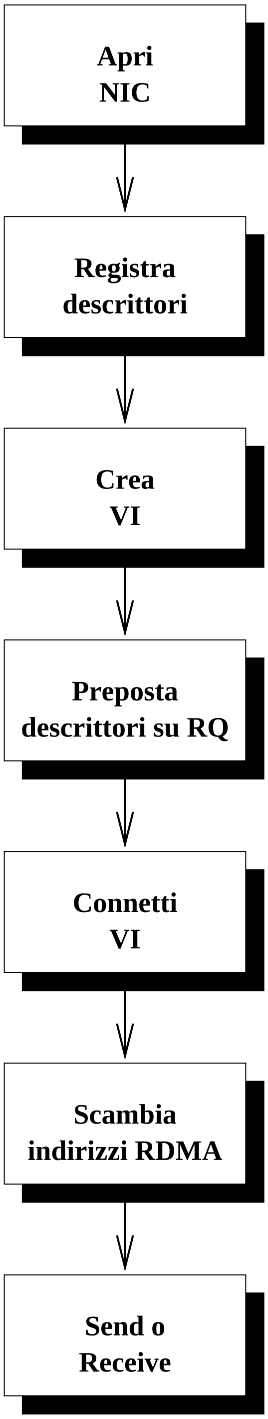

Figura 7.1: Protocollo basato sul pre-posting di descrittori sulla RQ.

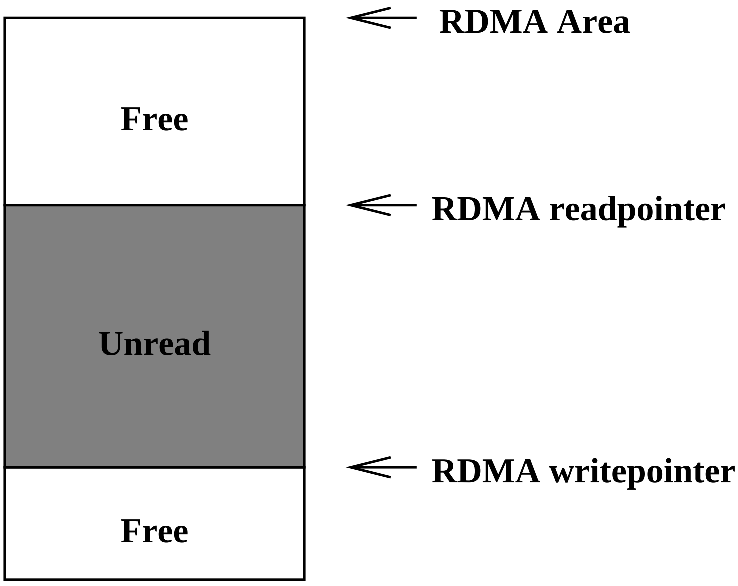

Figura 7.2: Area di memoria usata per RDMA.

1. Verifica se il messaggio da inviare si trova in un buffer già registrato.

2. In caso positivo scrive l'indirizzo del buffer nel descrittore.

3. In caso negativo verifica di avere a disposizione un buffer preregistrato di dimensioni sufficienti a contenere il messaggio da inviare; se la disponibilità esiste viene scritto nel descrittore l'indirizzo di questo buffer, altrimenti si procede a una nuova registrazione.

4. Tutti i campi necessari del descrittore vengono inizializzati per ottenere una comunicazione di tipo RDMA Write con Immediate Data, il quale rappresenta la dimensione del messaggio da scrivere nella RDMAArea del peer; l'indirizzo remoto in cui scrivere è ricavato dall'RDMAwritepointer che memorizza il primo indirizzo libero nell'RDMAArea.

5. Il descrittore viene postato sulla Send Queue.

6. RDMAwritepointer è aumentato di una quantità pari alla dimensione del messaggio spedito.

Il processo che voglia ricevere:

1. Analizza il descrittore in cima alla propria Receive Queue.

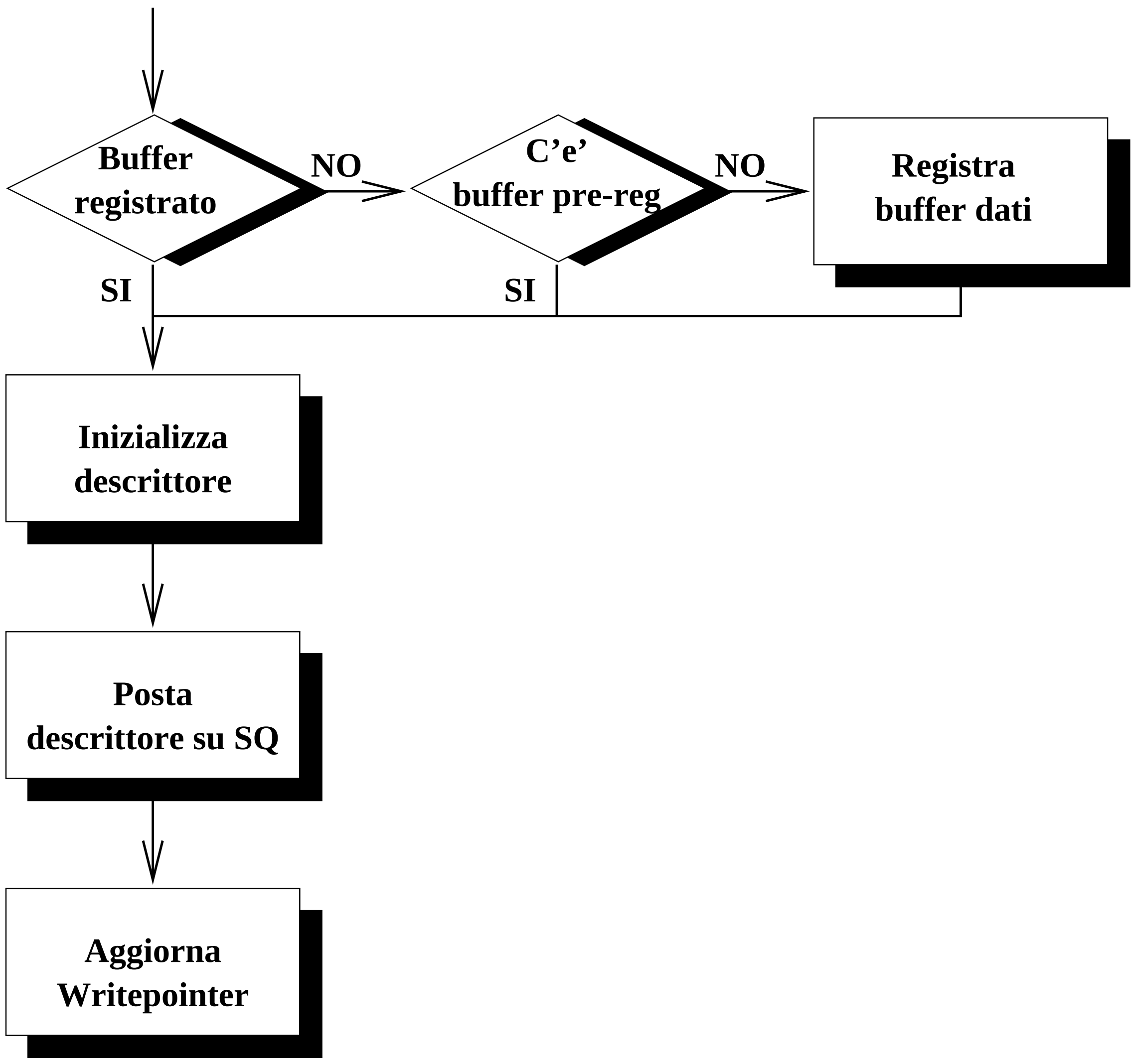

Figura 7.3: Operazioni di Send con protocollo basato sul pre-posting.

2. Se sulla cima della RQ vi è un descrittore completato il processo ha già il dato nella propria RDMAArea, altrimenti può scegliere se bloccarsi o uscire dalla procedura di receive.

3. Legge la dimensione del messaggio ricevuto nell'Immediate Data del descrittore completato.

4. Legge il dato dalla propria RDMAArea all'indirizzo indicato da RDMAreadpointer.

5. Aggiorna RDMAreadpointer con la dimensione del messaggio letta in Immediate Data del descrittore ricevuto.

6. Reinizializza il descrittore utilizzato per la receive.

7. Riposta sulla Receive Queue il descrittore in modo da mantenere la coda piena.

7.1.3 Protocolli basati su RDMA Write

Per evitare di postare sulla Receive Queue un gran numero di descrittori precedentemente registrati e inizializzati e per potere al contempo fornire delle primitive per una comunicazione di tipo asincrono si può pensare di sfruttare le operazioni di RDMA Write; il loro limite risiede nella incapacità di notificare al ricevente l'avvenuto invio di dati, se non con l'uso di Immediate Data.

L'idea alla base di questo tipo di protocolli è di utilizzare RDMA Write solo nel caso che il processo ricevente non sia pronto ad accettare un invio di dati effettuato tramite Send che necessita di un descrittore postato sulla RQ.

Le modalità per rilevare situazioni di mancanza di descrittori pronti per ricevere possono essere diverse:

- un controllo sulla correttezza della Send, che qualora fallisse indicherebbe che la RQ del processo destinatario è vuota.

- L'attesa di un segnale di acknowledge inviato dal ricevente del messaggio originario mandato tramite Send al mittente, il quale dopo un certo intervallo di tempo senza avere ricevuto nulla rispedisce il messaggio con RDMA Write.

- Un controllo preventivo effettuato sulla base di informazioni di stato che indichino la capacità di un processo di potere ricevere delle Send.

Le prime due alternative di verifica a posteriori soffrono sicuramente di una scarsità di prestazioni in quanto implicano la ritrasmissione dell'intero messaggio in caso di errore, e, nel secondo caso, un tempo di attesa dell'acknowledge; potrebbero trovare applicazione nel caso che tali situazioni di errore fossero altamente improbabili, cosicchè grazie alla loro semplicità le prestazioni sarebbero migliori. Ma nel caso delle librerie MPI la probabilità di effettuare una Send prima di una Receive può essere alta.

Si è quindi scelto di utilizzare il controllo preventivo basato su informazioni di stato che i processi devono tenere aggiornate e all'occorrenza scambiarsi.

Una possibile realizzazione del protocollo richiede a ogni processo di mantenere le seguenti variabili per ogni altro processo connesso:

RDMAArea : similmente al protocollo basato sul pre-posting dei descrittori è il puntatore a una vasta area di memoria destinata a raccogliere i dati inviati tramite operazioni di RDMA Write. Questa area di memoria è gestita in maniera analoga a quanto succede alla RDMAArea nel protocollo con pre-posting.

myRecv : rappresenta il numero di descrittori postati dal processo sulla propria Receive Queue e non ancora usati dal processo peer.

peerRecv : indica il numero di descrittori postati sulla RQ dal peer e che il processo non ha ancora utilizzato per operazioni di Send.

myRDMA : contiene il numero di scritture RDMA eseguite dal processo nella RDMAArea del peer e da questo non ancora utilizzate.

peerRDMA : immagazzina il numero di scritture RDMA effettuate dal peer e non ancora lette dal processo.

Tutte queste variabili devono potere essere accessibili dal processo peer che in certe circostanze scrive alcune di esse, senza tuttavia poterne conoscere il valore precedente alla scrittura da esso effettuata. Questo implica uno scambio iniziale tra i due processi degli indirizzi e dei gestori di tutte le zone di memoria o variabili interessate da operazioni di RDMA Write.

Quindi il protocollo si svolge nel seguente modo:

- Apertura del NIC.
- Creazione della VI.
- Connessione delle VI di due processi.
- Scambio di indirizzi e handle per RDMA.
- Send e Receive.

L'operazione di Send si svolge secondo lo schema seguente:

1. Prima di inviare dati un processo deve decidere quale operazione usare; la scelta è compiuta in base al valore del proprio peerRecv: se è diverso da zero significa che sulla Receive Queue del processo peer è a disposizione almeno un descrittore per ricevere dati che quindi il processo può inviare con la primitiva Send di VIA; in caso contrario il trasferimento di dati viene svolto con RDMA Write allo stesso modo del protocollo con preposting dei descrittori ma senza l'uso di Immediate Data. In tal caso la dimensione del messaggio inviato deve essere acclusa al messaggio stesso ad esempio nei suoi primi quattro byte.

2. Nel caso di Send occorre postare il descrittore sulla propria Send Queue e aggiornare le informazioni di stato: si decrementa il numero di descrittori postati dal peer sulla sua RQ (peerRecv) e si scrive nella variabile myRecv del peer il valore appena aggiornato di peerRecv.

3. Nel caso di RDMA Write occorre inviare i dati al giusto indirizzo, aggiornare la zona stessa di scrittura, incrementare il numero di RDMA Write effettuate dal processo (myRDMA) e inserire nella variabile peerRDMA del peer il valore appena cambiato di myRDMA.

In modo complementare la Receive si svolge secondo le seguenti modalità:

1. Il processo ricevente deve scegliere quale primitiva usare per ricevere i dati; la scelta è compiuta in base al numero di RDMA Write che il processo peer ha svolto: se diverso da zero significa che in precedenza il peer ha inviato tramite RDMA dei dati che ora sono memorizzati nella RDMA Area del processo; se invece peerRDMA è uguale a zero significa che non vi sono dati immagazzinati nell'RDMA Area e quindi è necessario attendere l'arrivo di messaggi postando un descrittore sulla Receive Queue.

2. Nel caso che il peer abbia trasmesso con RDMA Write il processo deve recuperare il dato nella sua RDMAArea, gestita similmente al caso del protocollo con pre-posting dei descrittori, decrementare la variabile peerRDMA che indica il numero di RDMA fatte dal peer e non ancora consumate, trasmettere per mezzo di RDMA Write il valore peerRDMA nella variabile del peer myRDMA per notificargli il numero di RDMA fatte dal peer e non ancora usate da una Receive.

3. Nel caso che non vi siano dati nell'RDMA Area, la ricezione viene compiuta postando un descrittore sulla Receive Queue del processo; è però necessario notificare al peer che egli, se desidera trasmettere, deve ora usare una Send e non una RDMA Write: dopo avere incrementato myRecv il processo inserisce con una RDMA Write questo valore nella variabile peerRecv del peer per informare il peer stesso del numero di descrittori pronti sulla RQ del processo. Tornando alla Send infatti si nota come la scelta del tipo di operazione da usare per inviare messaggi sia effettuata controllando proprio la variabile peerRecv che abbiamo visto essere aggiornata dal processo peer in base alla propria disponibilità di descrittori sulla RQ.

I problemi più significativi di questo protocollo sono:

La consistenza dei dati. Nella spiegazione delle operazioni di Send e di Receive si è supposto che l'invio dei dati tramite una delle due modalità permesse da VIA e l'aggiornamento delle variabili di stato sul processo e soprattutto sul processo remoto siano operazioni atomiche, cioé che avvengano in modo indivisibile senza che nessun altro evento possa interromperle; in caso contrario si potrebbe verificare che nel lasso di tempo che intercorre tra la trasmissione e la ricezione di una variabile, ad esempio peerRecv, da un processo a un peer, il peer stesso esamini il valore di peerRecv portandosi in una situazione di inconsistenza dei dati che preclude il corretto funzionamento dell'algoritmo del protocollo. La soluzione può risiedere nell'uso di semafori di tipo mutex per consentire l'accesso alle variabili peerRecv, myRecv, peerRDMA e myRDMA a un solo processo alla volta.[3]

[3] È possibile realizzare un semaforo mutex con operazioni di tipo RDMA Write svolgendo un numero elevato di comunicazioni.

Le prestazioni. Un trasferimento di dati con questo protocollo implica lo svolgimento di più di una operazione di comunicazione: l'invio di dati vero e proprio effettuato tramite Send o RDMA Write, l'aggiornamento tramite RDMA Write delle variabili opportune che devono essere protette da un semaforo mutex distribuito aumentano sensibilmente la latenza rispetto a quella di un singolo messaggio. Tale risultato è inaccettabile per messaggi di piccole dimensioni, e solo per grandi messaggi il ritardo introdotto dalle operazioni di gestione del procollo sono tracurabili.

Il primo problema, poi riconducibile al secondo, rende il protocollo inutilizzabile per applicazioni il cui scopo principale sia quello di ridurre la latenza e in cui le trasmissioni siano di tipo bursty, cioé frequenti e di dimensioni limitate. Questo protocollo è stato tuttavia descritto perché con l'uso di RDMA Read e alcune modifiche è effettivamente realizzabile.

Il vantaggio rispetto al precedente protocollo risiede nell'assenza del preposting dei descrittori ma soprattutto nella mancanza della necessità di preregistrare un gran numero di descrittori dal momento che con questo protocollo il numero di descrittori da registrare è pari al numero di Receive che il processo intende fare senza consumarne alcuna.

7.1.4 Protocolli basati su RDMA Read

L'attuale problema di protocolli basati su operazioni di RDMA Read è la mancanza della realizzazione di tale primitiva nella versione corrente di M-VIA, quindi, mentre dei precedenti protocolli sono stati costruiti vari prototipi, alcuni dei quali parzialmente, in questo caso si può solo accennare al possibile funzionamento.

Le operazioni di RDMA Read possono essere usate per rendere il precedente protocollo basato su RDMA Write fattibile e migliore dal punto di vista delle prestazioni: per i trasferimenti di dati si utilizzano ancora scritture RDMA, mentre le RDMA Read vengono sfruttate per le gestione delle variabili di stato della comunicazione tra i vari processi.

In questo scenario oltre alle variabili per la gestione della RDMA Area sono necessarie solo altre due variabili accessibili in lettura e scrittura da remoto per ogni processo:

recvPosted: indica il numero di descrittori attualmente postati da un processo sulla propria Receive Queue.

RDMAdone: registra il numero di scritture RDMA svolte dal peer remoto nella RDMA Area del processo, cioé il numero di messaggi che un processo ha ricevuto tramite RDMA Write ma non ha ancora consumato.

In questo caso per spedire un messaggio la procedura è la seguente:

1. Il processo mittente legge la variabile remota[4] contenente il numero di descrittori postati sulla RQ del processo peer, assieme alla variabile RDMAdone, usata sucessivamente, con una sola operazione di lettura, raggruppando le due variabili in una struttura.[5]

2. Nel caso questo valore sia diverso da zero si effettua una Send postando un descrittore sulla propria Send Queue dal momento che vi è un descrittore pronto per la ricezione; si aggiorna quindi con una RDMA Write la variabile recvPosted appena letta sul peer decrementandola di una unità.

3. Nel caso di assenza di descritori sulla RQ del peer il processo è costretto a trasmettere tramite RDMA Write: prima legge dal peer il numero di RDMA trasmesse dal processo al peer e da quest'ultimo non ancora consumate con receive, quindi invia il dato assieme al valore di RDMAdone appena letto incrementato di una unità, operazione che grazie alle proprietà delle RDMA Write viene svolta con una sola lettura.

Per ricevere:

1. Il processo ricevente legge la variabile locale RDMAdone ed esamina il numero di RDMA compiute dal peer.

2. Se non vi sono scritture RDMA fatte dal peer non consumate allora la receive viene effettuata postando sulla propria Receive Queue un descrittore; il processo incrementa quindi la variabile locale recvPosted.

3. Se vi sono scritture RDMA disponibili il processo legge la opportuna zona di memoria e decrementa la variabile contenente gli RDMA Write non consumati.

[4]È la variabile recvPosted del processo peer.

[5]Le operazioni di RDMA Write consentono scritture di tipo gather, cioé un numero arbitrario di buffer locali può essere inviato con un solo descrittore; viceversa le operazioni di RDMA Read consentono letture di tipo scatter, cioé un numero arbitrario di buffer locali possono contenere dati letti in un unico buffer remoto.

In questo modo si osserva come le operazioni di comunicazione pertinenti alla gestione del protocollo si sono ridotte a due: la RDMA Read compiuta prima di mandare messaggi e la RDMA Write svolta dal processo nel caso trasmetta con la primitiva Send. Tuttavia non sono state stimate le prestazioni in quanto non è ancora nota la latenza introdotta da un operazione di RDMA Read. Anche in questo caso però, come nel protocollo precedente, il problema maggiore è costituito dalla consistenza dei dati che rende obbligatorio l'uso di semafori mutex per l'accesso alle variabili recvPosted e RDMAdone; in questo caso però la realizzazione dei semafori mutex risulta più semplice.

Una diversa possibilità consiste nel realizzare un protocollo in cui la semantica delle operazioni di Send e Receive sia opposta: in questo caso una Send deposita un dato in una vasta zona di memoria accessibile da remoto tramite RDMA Read[6], mentre l'operazione che implica un effettivo transito di dati sulla rete è la Receive realizzata con una RDMA Read; l'operazione difficile da realizzare è ora la send sincrona, in quanto un processo mittente non riceve la notifica del completamento della RDMA Read.

7.2 Controllo di flusso

Visti i problemi del protocollo basato su RDMA Write e l'indisponibilità di RDMA Read la scelta del protocollo cade evidentemente su quello basato sul preposting dei descrittori. Tale protocollo deve fornire un servizio di trasmissione affidabile, per cui è necessario integrare altre funzionalità che evitino il fallimento delle comunicazioni a causa dell'esaurimento delle risorse allocate per lo svolgimento corretto del protocollo di comunicazione. Per la risoluzione di questo problema è stata aggiunta una parte di codice che svolge le funzioni di controllo sul flusso di dati che due generici processi si scambiano.

Avendo dunque adottato il protocollo basato sul pre-posting dei descrittori il controllo di flusso si deve occupare dei seguenti problemi:

1. l'esaurimento della RDMA Area, nel caso in cui non disponga di spazio sufficiente a contenere i messaggi che il mittente desidera inviare;

2. la mancanza di descrittori sulla RQ per ricevere messaggi.

[6]Questa area di memoria è l'equivalente della RDMA Area

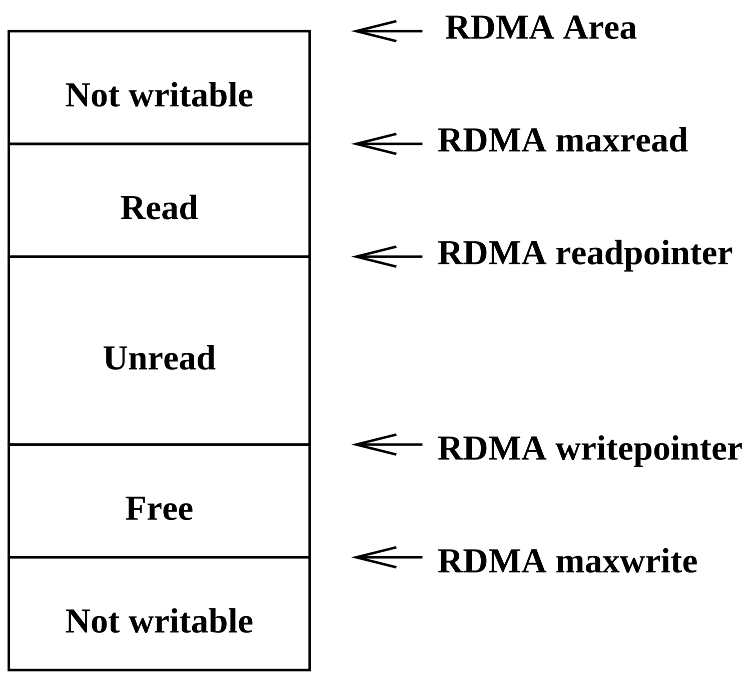

Figura 7.4: Area di memoria usata per RDMA con variabili per il controllo dello spazio disponibile.

Per risolvere il primo problema occorre aggiungere alcune altre variabili necessarie alla gestione della RDMA Area, che ora diventa come in figura 7.4:

RDMAmaxwrite: indica il numero massimo di byte che possono essere attualmente scritti nella RDMA Area del peer.

RDMAmaxread: è una variabile scritta attraverso RDMA dal peer per notificare al processo quanti byte della sua RDMA Area egli abbia letto e che quindi possano essere riutilizzati per l'invio di messaggi.

Queste variabili in aggiunta a quelle presentate in precedenza (RDMAArea, RDMAwritepointer e RDMAreadpointer) vengono utilizzate in questo modo dal mittente:

1. Il processo mittente controlla se nella RDMA Area del peer vi sia ancora spazio per il messaggio che egli intende spedire; in caso affermativo prosegue con la RDMA Write con Immediate Data.

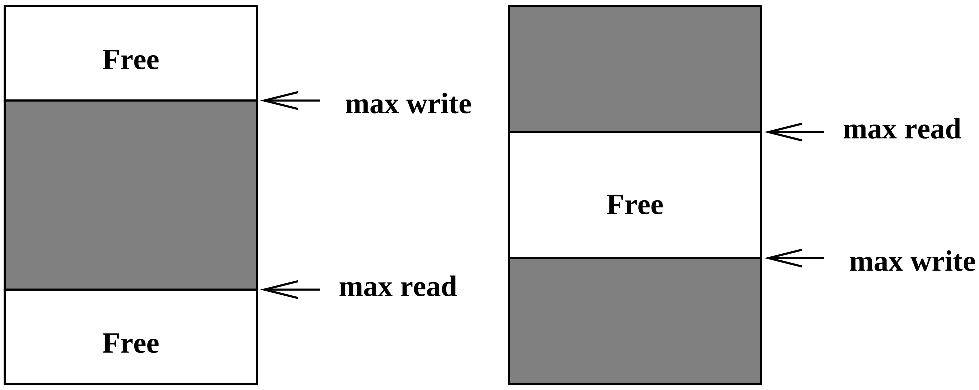

Figura 7.5: Possibili relazioni tra RDMAmaxwrite e RDMAmaxread.

2. Nel caso non vi sia spazio il processo verifica se RDMAmaxwrite sia maggiore di RDMAmaxread: questo per discriminare due possibili scenari illustrati dalla figura 7.5, cioé che la RDMA Area sia disponibile fino alla sua fine oppure che sia disponibile solo fino a RDMAmaxread.

3. Se la RDMA Area è libera sino al suo termine allora, dopo aver verificato se il messaggio da inviare vi possa essere contenuto, si riprende a scrivere dal primo byte della RDMA Area: si potrà giungere fino al valore indicato da RDMAmaxread che verrà copiato in RDMAmaxwrite; è però necessario avvertire il peer che si è ripreso a scrivere dall'inizio della RDMA Area, inviando un messaggio contentente solo Immediate Data posto per convenzione uguale a zero per segnalare l'evento.

4. Se invece la RDMA Area è libera solo fino a RDMAmaxread tutto quello che si può fare è aggiornare RDMAmaxwrite a questo valore.

5. Nel caso che ancora il messaggio non possa stare nella zona libera della RDMA Area, allora il processo è costretto ad attendere e ripetere la procedura.

Il processo ricevente deve:

1. Verificare se sono giunti messaggi con Immediate Data pari a zero, e in tal caso aggiornare il suo RDMAreadpointer a zero che corrisponde all'inizio della RDMA Area.

2. Periodicamente scrivere con una operazione di RDMA Write nella variabile RDMAmaxread del peer il valore che rappresenta l'inizio zona

della RDMA Area che non è ancora stata letta e che non può quindi essere sovrascritta.

Per quanto riguarda il numero dei descrittori postati sulla Receive Queue che in alcune circostanze può rivelarsi insufficiente, sono possibili due tipi di soluzioni:

1. Gestire l'errore una volta verificato, effettuando una nuova procedura di sincronizzazione.

2. Prevenire l'errore con l'uso di alcune variabili, similmente a ciò che si fa per la RDMA Area.

Nel primo caso occorre controllare che una operazione di scrittura RDMA con Immediate Data vada a buon fine: VIA prevede che nel caso che una Send o una RDMA Write con Immediate Data siano mandate a una Virtual Interface che presenti la sua Receive Queue vuota, sia generato sul processo ricevente un errore particolare, chiamato asyncronous error, senza possibilità di gestione dell'errore e senza notifica sul processo mittente. Sotto queste condizioni l'errore può essere recuperato attraverso un protocollo in cui ciascun messaggio inviato, singolarmente o a gruppi, attenda, con un certo ritardo, dal destinatario un notifica di avvenuta ricezione. Questa soluzione riduce drasticamente le prestazioni richiedendo qualche forma di risposta per ogni messaggio inviato e rende vano il miglioramento dovuto all'utilizzo delle code delle VI con il relativo pre-posting dei descrittori.

Si è quindi scelto di prevenire l'errore, introducendo alcune variabili e opportune procedure per la loro gestione:

descLeft: indica il numero di descrittori rimasti sulla Receive Queue del peer; è decrementato ogni volta che il processo esegue una Send o una RDMA Write con Immediate Data, operazioni che consumano un descrittore sulla Receive Queue del peer.

needDesc: è un flag che viene posto a 1 tramite RDMA dal peer nel momento in cui la sua variabile descLeft abbia raggiunto un valore di soglia minima, indicando in tal modo che il peer necessita di sapere quanti descrittori siano in realtà postati sulla Receive Queue del processo.

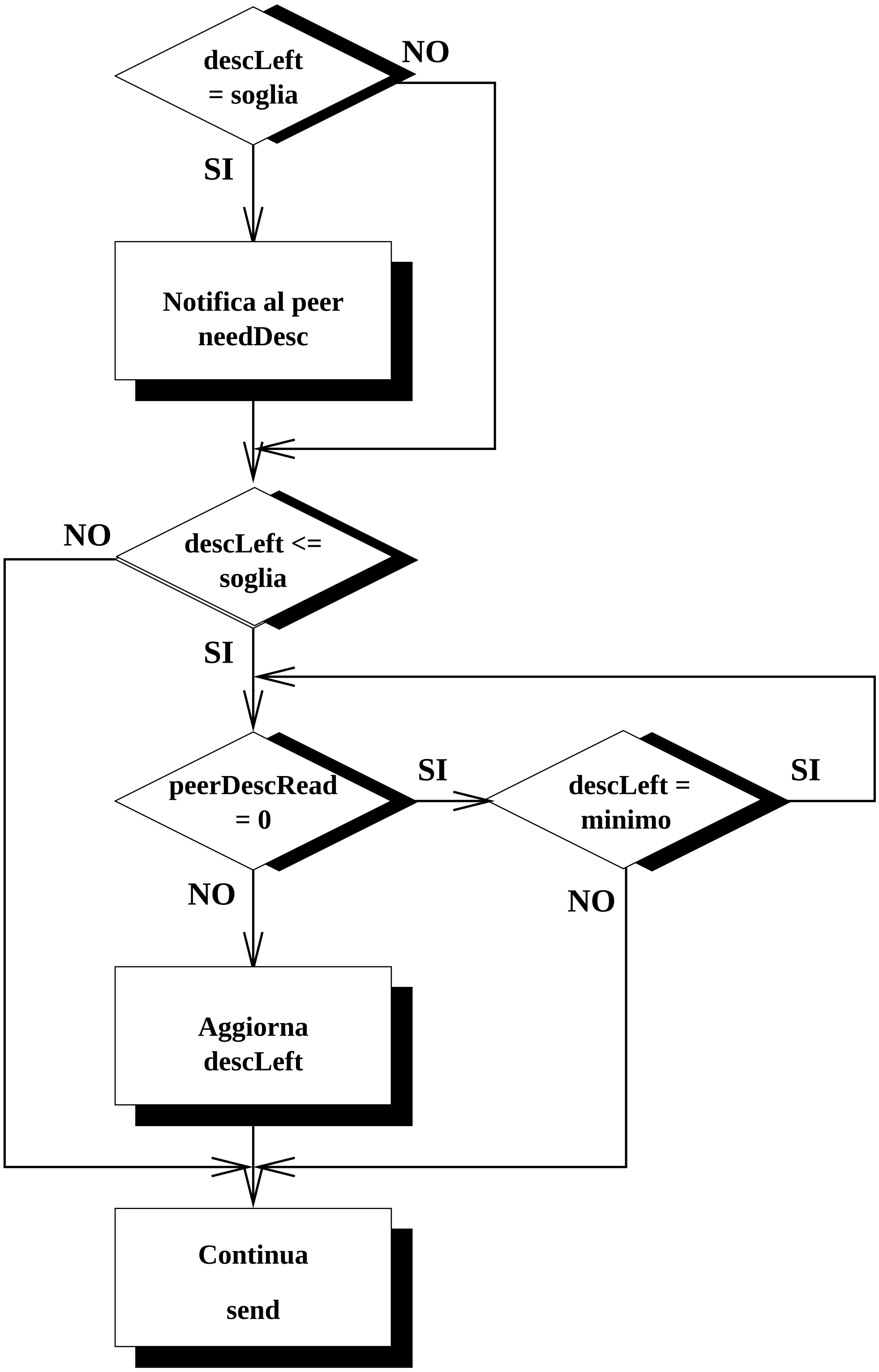

Figura 7.6: Send con controllo del numero dei descrittori.

peerDescRead: è una variabile destinata a contenere il numero di descrittori letti dal peer a partire dall'ultima volta in cui esso abbia comunicato questa informazione; viene scritta tramite RDMA dal peer.

descRead: indica il numero di descrittori letti dal processo a partire dall'ultima volta in cui il processo abbia inviato al peer il numero di descrittori letti.

Un processo che voglia inviare un messaggio (figura 7.6):

1. Controlla che il valore di descLeft sia uguale alla soglia minima; in caso negativo si prosegue, altrimenti si imposta a 1 tramite RDMA Write il flag needDesc del peer, notificandogli il bisogno di conoscere quanti descrittori siano attualmente postati sulla sua Receive Queue.

2. Verifica che descLeft sia minore o uguale alla soglia minima: in caso negativo esce dal controllo di flusso, altrimenti esamina peerDescRead che può essere stato aggiornato dal peer.

3. Se peerDescRead è diverso da zero significa che il peer ha inviato in peerDescRead il numero di descrittori da esso letti e ripostati sulla propria Receive Queue e quindi a disposizione del processo per spedire; descLeft è aumentato di peerDescRead; peerDescRead è posto a zero dal processo.

4. Se peerDescRead è rimasto uguale a zero, il peer non ha comunicato di avere proceduto all'invio dei descrittori letti; in questo caso il processo può avanzare finchè descLeft non abbia raggiunto il valore minimo di descrittori oltre cui un successivo invio di informazioni provoca un asyncronuos error.

Un processo che riceve un messaggio:

1. Prima di inviare un messaggio controlla la sua variabile needDesc: se è uguale a zero, il processo mittente non ha fatto richiesta di conoscere il numero di descrittori sulla Receive Queue del processo ricevente.

2. Se needDesc è diverso da zero occorre mandare al processo mittente il numero di descrittori letti e ripostati, valore che è contenuto in descRead, che dopo l'invio viene riportato a zero.

3. Ogni volta che un processo riceve un messaggio e riposta un descrittore sulla propria Receive Queue incrementa descRead.

7.3 Progetto delle 8 funzioni dell'interfaccia RPI con M-VIA

Dopo avere progettato un protocollo adatto per lo svolgimento delle comunicazioni richieste dal Lower Layer di LAM MPI, il passo successivo consiste nella integrazione del protocollo con controllo di flusso all'interno del Lower Layer di LAM. Come descritto nel capitolo precedente, per potere utilizzare una nuova tipologia di comunicazione all'interno di LAM non è necessario riscrivere tutto il codice, ma basta realizzare un nuovo Lower Layer che si interfaccia con l'Upper Layer con le otto funzioni che costituiscono la Request Progression Interface.

Vengono di seguito spiegate le operazioni che ciascuna delle otto funzioni della nuova interfaccia RPI svolge.

_rpi_c2c_init. Viene chiamata da MPI_Init() e si occupa della inizializzazione del livello di trasporto di LAM. Deve quindi effettuare le operazioni per avviare una sessione di comunicazione con VIA. In primo luogo viene aperto il VI NIC. Dal momento però che un processo non conosce a priori la localizzazione degli altri processi che possono risiedere anche sullo stesso nodo, vengono aperti due NIC, uno con il device di loopback, e uno con il device per le comunicazioni remote. Per sapere il nome corretto dei device si è scelto di leggere le informazioni necessarie da un file di configurazione che deve specificare il nome del nodo e, separato da uno spazio, il nome del device M-VIA per le comunicazioni internodo.

Per ogni device aperto vengono creati dei protection tag che saranno usati per le operazioni di registrazione delle zone di memoria; viene inoltre attivata la funzione di name service, che sarà usata in fase di connessione per la traduzione dei nomi dei nodi in indirizzi Ethernet.

Le informazioni create all'interno di _rpi_c2c_init vengono immagazzinate nell'apposito campo delle struttura _proc riservato ai dati relativi al livello di trasporto.

_rpi_c2c_addprocs. Questa funzione è incaricata di creare le strutture dati per la comunicazione e di connettere ciascun processo a ogni altro. In primo luogo vengono allocate le matrici di descrittori e di buffer usate per tutti i trasferimenti di dati tra i processi; ogni matrice ha come dimensioni il numero dei processi meno uno e il numero di descrittori da postare su ogni coda, scelto in fase di progetto, affinchè un processo abbia a disposizione sempre il medesimo numero di descrittori per ogni processo a cui si collegherà. Quindi ogni processo crea e inizializza una lista, chiamata *VI*, con un elemento per ogni processo eccetto se stesso, contenente tutti i parametri relativi al protocollo di comunicazione in precedenza descritto. Segue la scelta del device effettivo da usare per le comunicazioni con ogni altro processo (loopback o remoto) in base all'ubicazione del processo stesso; questa informazione viene scritta in un campo di VI. Vengono quindi registrati tutti i buffer allocati in precedenza con il corretto device e protection tag; poi sono create le Virtual Interface, una per ciascun processo con cui si desidera comunicare, aggiornando anche in questo caso un campo della lista VI. Sulla Receive Queue di ogni VI vengono postati i descrittori, prima della fase di connessione in modo da evitare problemi di sincronizzazione dovuti a una Send che non ha a disposizione un descrittore su di una RQ.

A questo punto è necessario effettuare la connessione tra le VI dei processi: si utilizza un modello client-server che presenta l'inconveniente di richiedere una forma di sincronizzazione in quanto il server deve iniziare l'attesa di un client prima che questo invii la sua richiesta di connessione. Si è quindi mantenuto lo schema di connessione utilizzato nel Lower Layer costruito su TCP/IP, in cui la sincronizzazione viene ottenuta con l'uso di funzioni di comunicazione che sfruttano i demoni creati all'avvio di LAM; i processi sono suddivisi tra client e server in base al proprio rank all'interno della sessione di lavoro con LAM per connettersi. Dopo la fase di connessione segue lo scambio degli indirizzi delle zone di memoria che sono usate, oltre dal processo proprietario, anche che da altri processi per comunicare tramite RDMA.

A questo punto ciascun processo possiede una Virtual Interface connessa con una VI di ogni altro processo; sulle Receive Queue di ogni VI è inoltre postato un certo numero di descrittori in attesa di dati in arrivo.

_rpi_c2c_build. Si occupa di inizializzare con i valori corretti l'envelope che deve essere inviato con il messaggio.

_rpi_c2c_start. È deputata alla scrittura di tutte le informazioni relative a una richiesta concernenti il Lower Layer: il tipo di operazione da svolgere, send o receive, e la Virtual Interface destinataria. Aggiorna lo stato della richiesta a LAM_RQSSTART.

_rpi_c2c_destroy. La funzione non è stata modificata rispetto alla versione del Lower Layer con TCP/IP, e si occupa di liberare le risorse relative al Lower Layer di una richiesta.

_rpi_c2c_advance. Effettua le trasmissioni o ricezioni di dati. Scandisce la lista delle richieste e, in base alle informazioni contenute in ognuna di esse, esegue la corretta operazione di trasferimento di dati invocando delle funzioni di più basso livello che si occupano della ricezione o dell'invio di un messaggio. È all'interno di queste funzioni che sono realizzati gli scambi di dati con le primitive di VIA e i meccanismi di controllo di flusso illustrati in precedenza. I parametri da fornire a queste funzioni in parte sono contenute nella struttura _req e in parte nella lista VI.

_rpi_c2c_iprobe. È l'unica funzione non ancora realizzata, ma dal momento che non viene usata se non da una sola funzione MPI, MPI_Probe e la sua versione non bloccante MPI_Iprobe, questo non pregiudica il funzionamento corretto di altre operazioni.

_rpi_c2c_finalize. Chiamata una sola volta all'interno di un programma MPI alla fine del codice parallelo, si occupa di importanti funzioni di pulizia e deallocazione della memoria: ogni Virtual Interface deve infatti essere disconnessa e quindi distrutta. È di fondamentale importanza che questa funzione svolga l'operazione di deregistrazione della memoria per evitare che essa rimanga bloccata anche dopo la fine del processo. Infine bisogna cancellare il name service e chiudere i NIC aperti.

7.4 Analisi dei risultati

Si descrivono analiticamente in questo paragrafo le prestazioni ottenute dal protocollo di comunicazione progettato a partire dalle primitive di M-VIA con il controllo di flusso prima della sua integrazione nel livello di trasporto di LAM: vengono valutati i risultati di questo protocollo dal momento che esso può essere usato anche in altre applicazioni oltre a LAM.

In seguito si valutano le prestazioni delle operazioni MPI_Send e MPI_Recv svolte con il Lower Layer costruito a partire dal protocollo con controllo di flusso; il parametro che in entrambi i casi viene valutato è il ritardo di trasmissione.

Il controllo di flusso, come in precedenza dettagliatamente descritto, svolge i compiti di gestione delle risorse di comunicazione, in particolare della RDMA Area destinata a contenere i messaggi in arrivo, e del numero dei descrittori postati sulla Receive Queue di ogni processo.

Per quanto riguarda la RDMA Area i parametri di interesse per valutare l'incidenza delle operazioni dovute al controllo di flusso sul ritardo di trasmissione di un messaggio sono:

- la dimensione della RDMA Area;
- la frazione di RDMA Area la cui lettura provoca la notifica della disponibilità di spazio nella RDMA Area.

Ogni volta che il processo mittente arriva a consumare tutto lo spazio della RDMA Area del peer, riparte dall'inizio, inviando un opportuno messaggio al peer attraverso una RDMA Write con Immediate Data: questo introduce un ritardo molto limitato, dovuto al tempo impiegato per postare un descrittore sulla Send Queue.

Ogni volta che il processo peer arriva a leggere una quantità di dati pari alla frazione di RDMA Area scelta in fase di progetto, esso deve comunicare al processo mittente di avere avanzato nella RDMA Area mettendo a disposizione per nuove scritture la zona letta: anche in questo caso viene introdotto un ritardo pari al tempo necessario per postare un descrittore.

I ritardi di trasmissione dei pacchetti, quindi, non aumentano in modo apprezzabile, dal momento che i controlli sono effettuati prima che la risorsa RDMA Area si esaurisca e non è prevista l'attesa da parte di un processo, se non nel caso in cui un processo consumi per la scrittura una quantità di risorse pari al massimo allocato inizialmente senza che il peer abbia effettuato alcuna operazione di lettura; i tempi di trasmissione dei messaggi usati per il controllo di flusso non si vanno a sommare a quelli per l'invio dei pacchetti, piuttosto si sovrappongono e sono limitati solo dalla banda della rete.

La seconda risorsa sulla quale sono svolti controlli è il numero di descrittori postati sulla Receive Queue di un processo, per evitare che un processo effettui una operazione di RDMA Write con Immediate Data senza disporre di un descrittore postato sulla Receive Queue del peer. I parametri di interesse in questo caso sono:

- il numero di descrittori postati sulla Receive Queue;
- la soglia minima oltre la quale si richiede al peer di conoscere il numero di descrittori disponibili al momento.

Quando un processo che desideri spedire un dato verifica di avere raggiunto la soglia minima, inoltra tramite RDMA Write una richiesta al peer, il quale risponde inviando il numero di descrittori che esso ha a disposizione. Similmente al caso della RDMA Area, queste operazioni non introducono ritardi apprezzabili, se non nel caso di uso di tutte le risorse allocate senza che il peer ne abbia rilasciata alcuna, poichè i processi non si bloccano in attesa di risposta.

I ritardi di trasmissione dei pacchetti, quindi, come si può vedere nella tabella 7.1, per effetto delle operazioni di trasmissione dovute al controllo di flusso, non variano sensibilmente.

Prestazioni di MPI_Send e MPI_Recv

I risultati di tabella 7.2 sono i dati sperimentali che mettono a confronto i ritardi di trasmissione ottenuti nella comunicazione tra due processi su due nodi del sottocluster B di ParMa^2 con LAM MPI con il Lower Layer realizzato rispettivamente con M-VIA e TCP/IP. Il modello di testing è lo stesso usato nella valutazione delle prestazioni di LAM e MPICH: un processo effettua una MPI_Send per inviare un messaggio e si mette in attesa di risposta con un MPI_Recv; il secondo prima riceve, poi risponde ritrasmettendo lo stesso messaggio. Il tempo misurato su di un processo equivale al tempo di trasmissione di due messaggi uguali.

I dati mettono in evidenza il sensibile miglioramento ottenuto per messaggi di dimensioni ridotte, per i quali il contributo della latenza al tempo di trasmissione è dominante: la riduzione del tempo è di circa un quarto, mentre i 40 microsecondi che M-VIA guadagna in latenza rispetto a TCP si traducono ora in un vantaggio di 34 microsecondi dal momento che si è dovuto introdurre un overhead aggiuntivo per le copie dei dati nei e dai buffer. Per dimensioni di pacchetto più elevate tale guadagno si annulla, in quanto

Pacchetto Byte	RDMA Area 256KB (μs)	RDMA Area 1M (μs)	M-VIA (μs)
1	62	62	61
2	62	62	61
4	62	62	61
8	62	62	61
16	63	62	61
32	64	63	61
64	70	70	69
128	83	83	82
256	108	108	105
512	156	156	152
1024	253	252	146
2048	383	382	380
4096	555	555	549
8192	897	896	894
16384	1574	1571	1555
32768	2956	2955	2947

Tabella 7.1: Confronto tra i ritardi di trasmissione per pacchetti di diverse dimensioni con il protocollo con controllo di flusso e con M-VIA senza alcun controllo di flusso.

Pacchetto Byte	LAM con M-VIA (μs)	LAM con TCP/IP (μs)
1	98	132
2	98	132
4	98	132
8	98	132
16	100	137
32	104	140
64	111	146
128	125	158
256	151	183
512	213	233
1024	328	335
2048	470	486
4096	675	680
8192	1070	1122
16384	1884	1931
32768	3854	3718

Tabella 7.2: Prestazioni di LAM con M-VIA.

l'overhead dovuto alla copia dei dati aumenta proporzionalmente alla dimensione del pacchetto, mentre il vantaggio sul tempo di trasmissione di M-VIA rispetto a TCP diminuisce.

Capitolo 8

Conclusioni

Questo capitolo si occupa di trarre le conclusioni del lavoro svolto, seguendo il medesimo percorso che inizia la costruzione di un cluster dal nodo, per poi proseguire con la rete, con il protocollo di comunicazione fino alla libreria di programmazione parallela.

Per quanto riguarda la costruzione di un cluster di PC si è verificato che la scelta di utilizzare nodi costituiti da macchine multiprocessore aumenta ulteriormente il rapporto prestazioni-prezzo, in quanto con un costo aggiuntivo pari a circa un quarto del prezzo di un intero nodo si è in grado quasi di raddoppiare la potenza di calcolo di una macchina, senza penalizzare altri componenti. Questo è possibile grazie all'uso di un sistema operativo in grado di offrire pieno supporto a sistemi SMP, capace quindi di ripartire in modo automatico, senza introdurre overhead eccessivi, il carico di lavoro tra i due processori. Il sistema operativo Linux, oltre a presentare altri vantaggi tra cui la gratuità, fornisce queste caratteristiche. Si è quindi verificato come alcune applicazioni quasi dimezzino i loro tempi di esecuzione se suddivise in due processi ciascuno dei quali eseguito su una delle due CPU di un nodo.

L'interfaccia di rete attualmente più usata nei cluster per il suo rapporto prestazioni-prezzo è Fast Ethernet, anche se la sua banda passante limitata teoricamente a 12,5MegaByte al secondo, costituisce il collo di bottiglia del cluster. Per migliorare questo aspetto si può optare per reti con banda vicino al Gigabit al secondo, come Myrinet o Gigabit Ethernet, aumentando il costo per la rete di un ordine di grandezza. Tuttavia le previsioni indicano che nel prossimo futuro i costi soprattutto dell'hardware per Gigabit Ethernet diminuiranno sensibilmente, rendendo più conveniente la migrazione da Fast a Gigabit Ethernet. Una valida alternativa è costituita dall'adozione di una rete di interconnessione tra i nodi basata su VIA: sono ad oggi disponibili alcune schede di rete che realizzano VIA in hardware, garantendo una banda passante simile a quella di Gigabit Ethernet e una latenza vicina a quella di Myrinet; i costi, pur dello stesso ordine di grandezza di quelli di Myrinet, sono minori e comunque destinati a diminuire con l'affermazione di VIA come standard per cluster di PC.

La topologia di interconnessione più adatta a cluster di piccole e medie dimensioni, cioé fino a 16 nodi, è rappresentata da un collegamento in cui ogni nodo sia connesso a ogni altro con la piena banda messa a disposizione dalla rete: uno switch permette di realizzare una tale topologia. Tuttavia questo dispositivo presenta anche alcuni inconvenienti: introduce una latenza aggiuntiva a ogni messaggio in transito; presenta un costo non irrilevante, anche se in continua diminuzione.

Il concetto principale di VIA, cioé di rimuovere il kernel del sistema operativo dal percorso critico della trasmissione dei dati, senza alcun dubbio determina una diminuzione consistente della latenza, come si può vedere dalla tabella 8.1, con decrementi del 40 per cento circa rispetto a TCP e del 30 per cento circa rispetto a UDP. Tuttavia il solo confronto di questi risultati è insufficiente a valutare i reali benefici di VIA rispetto agli altri due protocolli considerati, dal momento che ciascuno di essi offre servizi diversi e si colloca in livelli del modello ISO/OSI non omologhi. TCP infatti fornisce un trasporto di dati affidabile, realizzando il controllo e la correzione dei pacchetti corrotti o persi; UDP invece è inaffidabile e non esegue ritrasmissione dei pacchetti che possono quindi non arrivare. VIA invece prevede tre diversi livelli di affidabilità, per cui risulta difficile determinare quali livelli OSI esso copra. Nel caso di VIA unreliable non sono realizzati dal protocollo meccanismi di controllo di errore, di controllo di flusso, per cui offre funzionalità più limitate rispetto a TCP; però è un protocollo orientato alla connessione, a differenza di UDP, che è simile a VIA per quanto riguarda il livello di affidabilità; per queste ragioni si potrebbe dire che VIA si colloca nel livello di rete del modello OSI. Tuttavia nel caso invece di VIA reliable viene realizzato un protocollo a livello di trasporto che garantisce la consegna ordinata dei pacchetti.

Quindi protocolli di comunicazione come VIA sono difficilmente classificabili secondo il modello ISO/OSI, dal momento che cercano di migliorare le prestazioni in termini di latenza e ritardo di trasmissione eliminando quanto più possibile i livelli software intermedi.

Protocollo	Latenza con hub (μs)	Latenza con switch (μs)
TCP	86	100
UDP	71	84
M-VIA	48	61

Tabella 8.1: Latenza con TCP, UDP e M-VIA.

Bisogna anche osservare che tutti i risultati sperimentali sono stati ottenuti con l'uso di una particolare realizzazione di VIA, M-VIA, progettata per il sistema operativo Linux. M-VIA è tuttora in fase di beta testing: presenta infatti dei malfunzionamenti che sono stati riportati agli sviluppatori, in particolare nelle connessioni contemporanee tra un numero elevato di processi, nella trasmissione simultanea di dati da parte di più processi. Inoltre non sono state integrate in M-VIA alcune importanti funzionalità come il modello di connessione peer-to-peer e i livelli di affidabilità diversi dall'unreliable: ciò ha un pesante impatto sull'utilizzo di M-VIA, in quanto non può essere garantita all'utente la consegna dei pacchetti. I problemi più consistenti dal punto di vista della stabilità di M-VIA dovrebbero trovare soluzione con l'uscita della prossima beta release, prevista nel mese di aprile 1999; mentre per quanto riguarda l'introduzione di livelli di affidabilità ulteriori, gli sviluppatori ne hanno esclusa la realizzazione prima della prossima estate. Nonostante i problemi, i risultati di test svolti con M-VIA al fine di determinarne latenza e ritardo di trasmissione hanno evidenziato delle ottime prestazioni, come in precedenza si faceva notare in tabella 8.1.

La scelta di preferire LAM MPI ad altre librerie di Message Passing, come si è visto, è motivata da una serie di elementi: LAM MPI è specificamente dedicata solo ai cluster di workstation; è una libreria multiprotocollo, cioé consente la scelta tra diversi meccanismi e protocolli che si occupano del trasferimento dei messaggi tra i processi, tra cui la comunicazione tramite shared memory; la struttura software di LAM MPI è stata disegnata per permettere l'integrazione di nuovi protocolli di comunicazioni di basso livello per la trasmissione dei messaggi; le prestazioni ottenute con LAM MPI sono superiori di circa il 10 per cento a quelle di MPICH.

Il primo problema affrontato per realizzare un nuovo livello di trasporto all'interno di LAM MPI utilizzando VIA per le comunicazioni è stato rappresentato dalla creazione di un protocollo per lo scambio di dati in grado di soddisfare tutte le esigenze di MPI in termini di sincronizzazzione e semantica delle operazioni di comunicazione. Dal momento che tra le funzioni MPI si annoverano operazioni sincrone e asincrone, bloccanti e non bloccanti non può essere utilizzato un modello totalmente asincrono, come quello con le operazioni RDMA di VIA. Resta quindi il modello basato su Send e Receive che presenta il problema della necessità di un descrittore pronto a ricevere prima che una operazione di Send venga iniziata: si è adottato un meccanismo di pre-posting dei descrittori, che consiste nell'inserire su ogni Receive Queue un numero elevato di descrittori prima della fase di connessione delle VI. Tuttavia, dal momento che ogni descrittore deve contenere un buffer di dimensioni pari alla dimensione massima di un messaggio inviato, riservare un numero elevato di buffer così grandi sfrutta in modo pessimo le risorse di memoria. Si è quindi deciso di usare operazioni di RDMA con Immediate Data, che hanno un comportamento analogo a quello delle Send per quanto riguarda gli aspetti di sincronizzazione e consumo di descrittori, ma non richiedono buffer in ricezione, in quanto i dati vengono scritti in una ampia zona di memoria riusabile per ricevere nuovi messaggi. Il protocollo di comunicazione è stato progettato in questo modo per consentire la realizzazione delle restanti funzioni di base di MPI in maniera agevole, utilizzando le strutture dati allocate da ogni processo LAM MPI: questo può costituire il passo successivo del progetto finalizzato alla realizzazione completa ed efficiente di LAM con M-VIA. L'istruzione che necessita invece di strutture dati aggiuntive è l'operazione di ricezione senza l'indicazione del processo mittente: si può pensare di introdurre nuove Virtual Interface per ogni processo per distinguere tipi di messaggi diversi, usando le Completion Queue per avere un unico punto di notifica dell'arrivo dei messaggi.

Il protocollo costruito dunque risolve i problemi di sincronizzazione, ma non evita che in particolari occasioni le risorse dedicate alla comunicazione vengano consumate. A questo scopo sono stati introdotti nel protocollo di comunicazione dei meccanismi di controllo di flusso per il monitoraggio delle due risorse a rischio di esaurimento: i descrittori sulle Receive Queue e le RDMA Area, la zone destinate a ricevere i dati tramite scritture RDMA. Questo controllo di flusso non incide sensibilmente sui tempi di trasmissione dei messaggi poichè usa solo comunicazioni asincrone realizzate tramite RDMA Write, e prevede le condizioni di scarsità di risorse, correggendole in anticipo. Il suo limite risiede nel fatto che se un processo mittente usa tutte le risorse allocate inizialmente per la comunicazione senza che il ricevente ne abbia liberata alcuna, il mittente si blocca in attesa del rilascio delle risorse.

Il protocollo con controllo di flusso è stato quindi integrato all'interno del codice di LAM MPI riscrivendone il suo Lower Layer: i messaggi da inviare contenuti in una struttura dati creata dall'Upper Layer di LAM sono copiati in buffer per la spedizione, e una volta arrivati sono copiati dalla RDMA Area alla loro destinazione finale. Queste due copie introducono un overhead, seppur abbastanza limitato; per evitarle si può pensare di inviare direttamente la struttura dati e registrarla prima della spedizione: ciò è certamente sconveniente dal momento che l'operazione di registrazione della memoria, necessaria per potere trasmettere un dato con VIA, è molto dispendiosa in termini di tempo in quanto richiede l'intervento del kernel del sistema operativo. L'alternativa consiste nel procedere alla registrazione al momento della creazione della struttura dati destinata a contenere i messaggi: questa soluzione richiede la riscrittura di gran parte del codice di LAM, anche nell'Upper Layer. Comunque i risultati sperimentali ottenuti con la coppia di funzioni MPI realizzate, MPI_Send e MPI_Receive, mostrano una riduzione dei tempi di trasmissione per messaggi di piccole dimensioni di circa un terzo rispetto a quelli con la comunicazione client-to-client con TCP/IP.

L'attuale limite delle comunicazioni di LAM realizzate tramite M-VIA risiede nella affidabilità: dal momento che per ora M-VIA supporta solo il livello di affidabilità che fornisce le minori garanzie, solo la duplicazione dei pacchetti è evitata ed è supportata la rilevazione di dati corrotti, non si ha la certezza che un messaggio inviato venga ricevuto. Tuttavia il controllo e la correzione di errore per ora non sono stati realizzati dal momento che i piani di sviluppo di M-VIA inizialmente prevedevano l'integrazione dei livelli di affidabilità Reliable per l'inizio del 1999, poi rimandati. Si può comunque procedere con lo sviluppo di questo progetto con l'introduzione nel protocollo di comunicazione di un livello software che si occupi di rilevare la perdita di pacchetti e di operarne la ritrasmissione, anche se tale scelta rischia di penalizzare le prestazioni a causa dell'overhead dovuto ai controlli aggiuntivi sulla ricezione dei pacchetti, mentre sono già disponibili schede di rete che realizzano VIA e prevedono anche i livelli di ricezione e trasporto affidabili. Inoltre le caratteristiche delle attuali versioni di M-VIA, in caso di perdita di un pacchetto, determinano comportamenti anomali nelle successive operazioni di comunicazione, rendendo difficoltosa la realizzazione delle ritrasmissioni, in quanto alcune strutture di VIA pervengono a uno stato di errore.

I test svolti hanno dimostrato che per comunicazioni tra una sola coppia

Pacchetto (bytes)	Ritardo di trasmissione (μs)
1	32
2	32
4	32
8	32
16	32
32	33
64	33
128	33
256	34
512	35
1024	38

Tabella 8.2: Latenza di MPI/Pro per VIA.

di processi e messaggi fino a 16 Kbyte, la probabilità di avere un pacchetto perso, è bassa nel sottocluster B di ParMa[2], circa uno su diecimila. Quando invece i processi comunicanti simultaneamente crescono di numero, la probabilità di errore cresce notevolmente.

Per dimostrare le possibilità offerte da VIA, infine, si riportano le prestazioni ottenute da una delle prime complete realizzazioni di MPI per VIA, MPI/Pro, prodotto da una software house americana, MPI Software Tecnology[1].

Il sistema utilizzato consiste di un cluster di macchine quadriprocessore Pentium II Xeon, connessi tramite switch e con schede di rete VIA con banda massima di 1.25 Gigabit al secondo; il sistema operativo è Windows NT.

I dati riportati in tabella 8.2 e 8.3 dimostrano con efficacia i grandi benefici che una System Area Network realizzata completamente con interconnessioni di rete VIA possa ricevere in termini di latenza e di banda passante.

Questi dati possono essere considerati come l'obiettivo finale del complesso progetto di realizzazione di LAM MPI con VIA, che con questo lavoro di tesi ha visto il suo inizio.

[1]Reperibile all'indirizzo `http://www.mpi-softtech.com`.

Pacchetto (bytes)	Banda (MBytes/s)
256	4.55
512	8.52
1024	15.01
2K	24.87
4K	36.32
8K	46.34
16K	59.57
32K	76.82
64K	89.21
128K	97.51
256K	101.47
512K	103.80
1024K	105.07

Tabella 8.3: Banda di MPI/Pro per VIA.

Capitolo 9

Appendice

9.1 L'interfaccia Small Computer Systems Interface (SCSI)

Benchè l'acronimo definisca SCSI come una interfaccia per lo scambio di dati, esso può essere catalogato come un bus, con un proprio protocollo, un serie di comandi per le operazioni da effettuare, con controller dotati di intelligenza e in grado di eseguire autonomamente parte delle operazioni di scambio di dati, con delle interfacce ben definite.

SCSI supporta periferiche di vario tipo da hard disk, a CD-ROM a scanner, ed offre vantaggi notevoli rispetto ad altre interfacce ed in particolare all'IDE/EIDE/ATA per la gestione dei dischi fissi, soprattutto in termini di prestazioni, espandibilità e compatibilità.

La struttura di un sistema SCSI comprende un controller che è capace di gestire solitamente sette periferiche di tipi diversi connesse in cascata l'una all'altra; proprio per la sua caratteristica di disporre di controller che svolgono la maggior parte della operazioni legate al trasferimento dei dati insieme all'alto throughput, l'architettura SCSI è particolarmente adatta per sistemi multitasking, in cui diverse periferiche colloquiano con le altre contemporaneamente. Per tutti questi motivi in cluster con necessità di operazioni di I/O l'interfaccia SCSI viene di solito preferita all'EIDE per i dispositivi di memorizzazione di massa, per evitare di introdurre nel sistema colli di bottiglia in applicazioni che eseguano frequenti operazioni di I/O.

Lo svantaggio del bus SCSI è sicuramente il costo, molto superiore ad esempio a quello da sostenenere per hard disk EIDE: a parità di capacità di memorizzazione circa il doppio.

Il trasferimento dei dati sul bus SCSI avviene in diverse fasi.

La prima fase è quella di bus libero, in cui l'unità che richiede di effettuare una operazione di I/O, chiamata initiator, verfica se al momento il bus non è sotto il controllo di un'altra unità.

Nella successiva fase di arbitraggio l'initiator invia un segnale di controllo per impostare il bus nello stato di occupato, spedendo anche il proprio identificativo SCSI fornito dal BIOS del controller.

Segue una breve pausa durante la quale altre unità possono inviare la richiesta del possesso del bus: in tal caso il bus è assegnato all'unità con la priorità più alta, individuata dall'identificativo.

Questo meccanismo consente a una sola unità di assumere il controllo del bus, entrando nella fase di selezione, in cui l'initiator specifica una unità target, quella cioè che deve effettuare fisicamente l'operazione di I/O, ed invia a essa opportuni comandi. Le unità sono in ascolto sul bus in modo da riconoscere un'eventuale chiamata al proprio indirizzo per rispondere alla richiesta, instaurando quindi una comunicazione durante la quale il bus non è disponibile per altre unità; per comunicazione si intendono tutte le operazioni necessarie per eseguire il comando inviato dall'initiator.

Appena il target completa il comando si passa ad una fase di riselezione, in cui si ripetono le operazioni di bus libero, arbitraggio e selezione per determinare il nuovo initiator di una comunicazione.

Il bus SCSI offre due diverse ampiezze: narrow, con 8 bit, ed è l'ampiezza implicitamente assunta se non viene indicato diversamente; wide, in cui l'ampiezza è raddoppiata a 16 bit; inoltre il bus wide raddoppia, oltre ovviamente al throughput a parità di velocità, anche il numero di periferiche che si possono collegare, fino a 15 (16 considerando anche il controller stesso contro le 8 del narrow).

Vi sono tre differenti velocità supportate dal bus SCSI:

Regular: è la velocità di default, 5 MHz, è quella che fu definita nel primo standard.

Fast: raddoppia la velocità del bus a 10 MHz; fu definita con la seconda specifica SCSI.

Nome	Velocità (MHz)	Ampiezza (bits)	Throughput (MByte/s)
SCSI	5	8	5
Fast SCSI	10	8	10
Wide SCSI	5	16	10
Fast Wide SCSI	10	16	20
Ultra SCSI	20	8	20
Ultra Wide SCSI	20	16	40
Ultra2 SCSI	40	8	40
Ultra2 Wide SCSI	40	16	80

Tabella 9.1: Throughput SCSI.

Ultra: 20 MHz.

Ultra2: di recente introduzione porta la velocità a 40 MHz.

Combinando insieme le velocità con le ampiezze del bus si ottengono i possibili throughput come indicato nella tabella 9.1.

9.2 Moduli del sistema operativo Linux

Per capire più efficacemente il modo di operare di M-VIA e comprenderne la collocazione in rapporto al sistema operativo si illustrano alcuni concetti basilari sull'uso e sulle funzioni dei moduli nel sistema operativo Linux [13] [30] [31].

Linux è un kernel monolitico, cioé è un unico e singolo grande programma in cui tutti i suoi componenti hanno accesso a tutte le strutture dati e a tutte le funzioni. La possibile alternativa è quella di strutturare il kernel del sistema operativo in diversi componenti che svolgono specifiche funzioni e il cui funzionamento è orchestrato da un micro-kernel con cui tutte le parti comunicano. Questa secondo schema rende più agevole l'inserimento nel sistema operativo di nuovi elementi funzionali, mentre nel primo caso per aggiungere al kernel una funzionalità si rende necessaria la ricostruzione e quindi la ricompilazione dell'intero kernel.

Per ovviare a questo problema Linux mette a disposizione una tecnica per caricare e scaricare in maniera dinamica dei componenti del sistema operativo: i moduli.

Un modulo è una parte di codice rilocabile che può essere linkata dinamicamente al kernel del sistema operativo in qualunque momento dopo l'avvio del sistema; quando il modulo non è più utilizzato esso può essere rimosso.

Gran parte dei moduli sono costituiti da device driver o da filesystem.

L'inserimento di un modulo nel sistema operativo può essere effettuato esplicitamente da parte dell'utente con previlegi di root tramite un apposito comando, *insmod*, oppure può essere demandato al sistema operativo stesso che attraverso un demone, *kerneld* sostituito nelle ultime versioni del kernel da *kmod*, si incarica di installare il modulo quando qualche operazione richieda una funzionalità offerta dal modulo stesso.

Il vantaggio nell'uso dei moduli consiste nella possibilità di non includere nel kernel monolitico funzionalità di uso sporadico, mantenendo in questo modo le dimensioni del kernel limitate e allo stesso tempo potendo fruire delle funzioni dei moduli caricandoli a richiesta senza dovere riavviare il sistema.

Questi vantaggi sono pagati con una diminuzione delle prestazioni delle operazioni fornite dai moduli in quanto essi introducono un livello di indirettezza che rende l'accesso alle risorse del kernel più lento, e con un maggior consumo di risorse poichè un modulo caricabile dinamicamente, oltre a presentare un complessità del codice superiore, deve provvedere delle strutture dati aggiuntive che consumano memoria.

Si analizzano ora le operazioni che vengono compiute all'atto del caricamento di un modulo nel sistema operativo: l'utility insmod si occupa per prima cosa di leggere il modulo da installare nella propria memoria; una volta caricato un modulo è parte del kernel a tutti gli effetti, con gli stessi diritti e le stesse responsabilità. Innanzitutto un modulo può utilizzare le risorse del kernel; per poterlo fare è necesario permettere al modulo di localizzare le risorse di cui ha bisogno: quando un modulo viene caricato insmod deve risolvere i riferimenti fatti all'interno del modulo alle risorse del kernel per mezzo di una tabella dei simboli che il sistema operativo mantiene; insmod scrive fisicamente gli indirizzi dei simboli utilizzati nei punti del modulo in cui sono utilizzati. Quindi insmod richiede al sistema operativo una zona di memoria in cui inserire il modulo: il kernel fornisce la memoria creando una nuova struttura dati di tipo *module* per il nuovo modulo e la inserisce in coda a una lista dei moduli *module_list* come indicato in figura 9.1; ciascuna struttura *module* contiene un puntatore alla lista dei simboli esportati dal modulo in modo che il kernel venga informato dei servizi forniti dal modulo stesso.

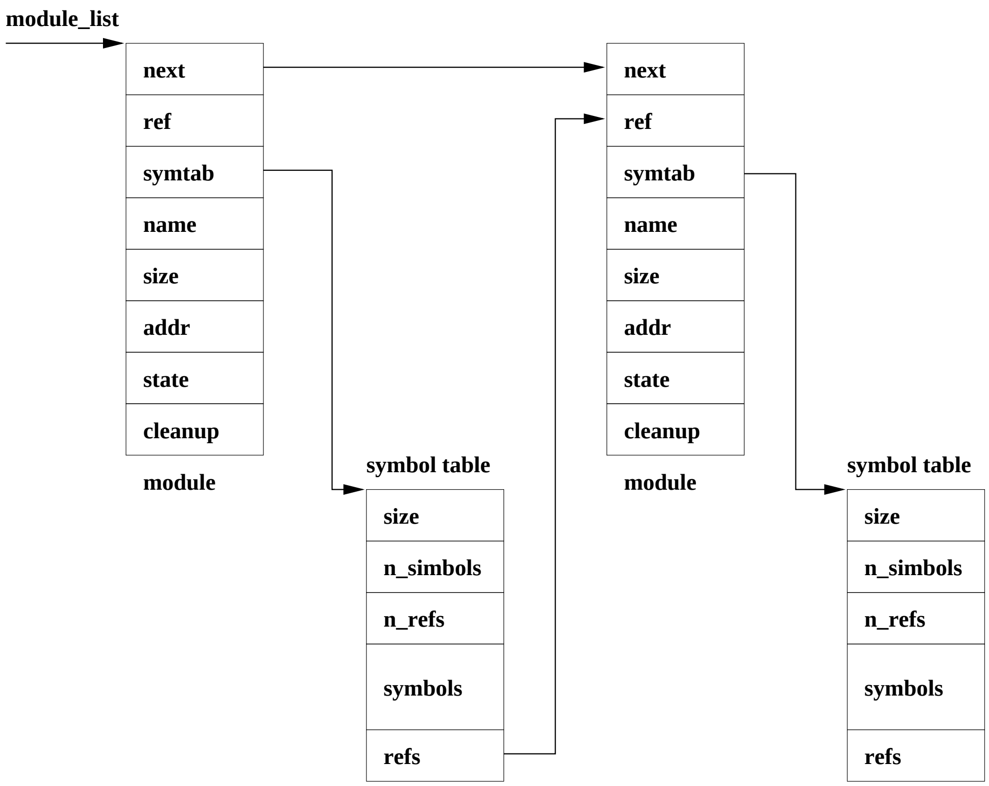

Figura 9.1: Strutture dati relative ai moduli.

La rimozione di un modulo dal sistema operativo avviene automaticamente nel caso sia stato caricato da uno degli appositi demoni oppure manualmente con il comando *rmmod*; in entrambi i casi viene verificato che nessuna entità utilizzi alcun servizio del modulo che viene in seguito depennato dalla *module_list* con la deallocazione della memoria usata.

Linux consente anche lo stacking dei moduli, cioé l'uso da parte di un modulo di servizi messi a disposizione da un altro modulo.

9.3 L'interfaccia VIPL di M-VIA

Si riportano di seguito le strutture dati utilizzate dalla VI Programming Library di M-VIA, che definisce l'insieme delle funzioni messe a disposizione all'utente di M-VIA [24].

I tipi di base sono ridefiniti:

```
typedef void *           VIP_PVOID;
typedef int              VIP_BOOLEAN;
typedef char             VIP_CHAR;
typedef unsigned char    VIP_UCHAR;
typedef unsigned short   VIP_USHORT;
typedef unsigned long    VIP_ULONG;
typedef __u8             VIP_UINT8;
typedef __u16            VIP_UINT16;
typedef __u32            VIP_UINT32;
typedef __u64            VIP_UINT64;
```

Handle opachi per i componenti di M-VIA:

```
typedef struct _VIP_NIC*   VIP_NIC_HANDLE;
typedef struct _VIP_VI*    VIP_VI_HANDLE;
typedef struct _VIP_CONN*  VIP_CONN_HANDLE;
typedef VIP_UINT32         VIP_PROTECTION_HANDLE;
typedef VIP_PVOID          VIP_MEM_HANDLE;
typedef struct _VIP_CQ*    VIP_CQ_HANDLE;
```

Altre definizioni:

```
typedef union {
    VIP_UINT64          AddressBits;
    VIP_PVOID           Address;
} VIP_PVOID64;

#define VIP_FALSE       0
#define VIP_TRUE        1
#define VIP_INFINITE    (~(VIP_ULONG) 0)
```

Allineamento dei descrittori:

```
#define VIP_DESCRIPTOR_ALIGNMENT 64
```

Codici di ritorno delle funzioni:

```
typedef enum {
    VIP_SUCCESS,
    VIP_NOT_DONE,
    VIP_INVALID_PARAMETER,
    VIP_ERROR_RESOURCE,
    VIP_TIMEOUT,
    VIP_REJECT,
    VIP_INVALID_RELIABILITY_LEVEL,
    VIP_INVALID_MTU,
    VIP_INVALID_QOS,
    VIP_INVALID_PTAG,
    VIP_INVALID_RDMAREAD,
    VIP_DESCRIPTOR_ERROR,
    VIP_INVALID_STATE,
    VIP_ERROR_NAMESERVICE,
    VIP_NO_MATCH
} VIP_RETURN;
```

Struttura dati di tipo **descrittore:**

- Segmento di controllo:

  ```
  typedef struct {
  ```

```
        VIP_PVOID64          Next;
        VIP_MEM_HANDLE       NextHandle;
        VIP_UINT16           SegCount;
        VIP_UINT16           Control;
        VIP_UINT32           Reserved;
        VIP_UINT32           ImmediateData;
        VIP_UINT32           Length;
        VIP_UINT32           Status;
    } VIP_CONTROL_SEGMENT;
```

- Segmento per gli indirizzi remoti RDMA:

```
    typedef struct {
        VIP_PVOID64          Data;
        VIP_MEM_HANDLE       Handle;
        VIP_UINT32           Reserved;
    } VIP_ADDRESS_SEGMENT;
```

- Segmento per i dati:

```
    typedef struct {
        VIP_PVOID64          Data;
        VIP_MEM_HANDLE       Handle;
        VIP_UINT32           Length;
    } VIP_DATA_SEGMENT;
```

Descrittore completo:

```
typedef union {
    VIP_ADDRESS_SEGMENT Remote;
    VIP_DATA_SEGMENT    Local;
} VIP_DESCRIPTOR_SEGMENT;

typedef struct {
    VIP_CONTROL_SEGMENT         CS;
    VIP_DESCRIPTOR_SEGMENT      DS[2];
} VIP_DESCRIPTOR;
```

Attributi di un VI NIC:

```
typedef struct {
    VIP_CHAR            Name[64];
    VIP_ULONG           HardwareVersion;
    VIP_ULONG           ProviderVersion;
    VIP_UINT16          NicAddressLen;
    const VIP_UINT8     *LocalNicAddress;
    VIP_BOOLEAN         ThreadSafe;
    VIP_UINT16          MaxDiscriminatorLen;
    VIP_ULONG           MaxRegisterBytes;
    VIP_ULONG           MaxRegisterRegions;
    VIP_ULONG           MaxRegisterBlockBytes;
    VIP_ULONG           MaxVI;
    VIP_ULONG           MaxDescriptorsPerQueue;
    VIP_ULONG           MaxSegmentsPerDesc;
    VIP_ULONG           MaxCQ;
    VIP_ULONG           MaxCQEntries;
    VIP_ULONG           MaxTransferSize;
    VIP_ULONG           NativeMTU;
    VIP_ULONG           MaxPtags;
    VIP_RELIABILITY_LEVEL MaxReliabilityLevel;
    VIP_BOOLEAN         RDMARead;
} VIP_NIC_ATTRIBUTES;
```

Attributi di una VI:

```
typedef struct {
    VIP_RELIABILITY_LEVEL  ReliabilityLevel;
    VIP_ULONG              MaxTransferSize;
    VIP_QOS                QoS;
    VIP_PROTECTION_HANDLE  Ptag;
    VIP_BOOLEAN            EnableRdmaWrite;
    VIP_BOOLEAN            EnableRdmaRead;
} VIP_VI_ATTRIBUTES;
```

Attributi di una regione di memoria:

```
typedef struct {
    VIP_PROTECTION_HANDLE  Ptag;
    VIP_BOOLEAN            EnableRdmaWrite;
    VIP_BOOLEAN            EnableRdmaRead;
} VIP_MEM_ATTRIBUTES;
```

Stati di una VI:

```
typedef enum {
    VIP_STATE_IDLE,
    VIP_STATE_CONNECTED,
    VIP_STATE_CONNECT_PENDING,
    VIP_STATE_ERROR
} VIP_VI_STATE;
```

Indirizzi di rete:

```
typedef struct {
    VIP_UINT16             HostAddressLen;
    VIP_UINT16             DiscriminatorLen;
    VIP_UINT8              HostAddress[1];
} VIP_NET_ADDRESS;
```

Bibliografia

[1] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture.* Morgan Kaufmann, 1999.

[2] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing.* Benjamin/Cummings Publishing Company, 1994.

[3] Paul Messina, David Culler, Wayne Pfeiffer, William Martin, Tinsley Oden, and Gary Smith. Architecture. *ACM Communications*, November 1998.

[4] Thomas E. Anderson, David Culler, David Patterson, and the NOW team. A case for NOW (Networks of Workstations). *IEEE Micro*, February 1995.

[5] Daniel Ridge, Donald Becker, and Phillip Merkey. Beowulf: Harnessing the power of parallelism in a Pile-of-PCs. 1996.

[6] David Culler, Richard Karp, David Patterson, Abijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eichen. LogP: towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Pratice of Parallel Programming*, San Diego, CA, May 1993.

[7] SMP. Link Linux SMP. `http://www.irisa.fr/prive/mentre/smp-faq/`
`http://www.linux.org.uk/SMP/title.html`
`http://isunix.it.ilstu.edu/ thockin/pset/`
`http://www.linuxhq.com/lnxlists/linux-smp/`
`http://www.cs.inf.ethz.ch/ rauch/procps.html` .

[8] A.S Tanenbaum. *Reti di Computer.* Prentice Hall, 1991.

[9] Luca Ferrè. Ethernet switching: il punto sulla tecnologia. *LAN e Telecom*, 1995.

[10] Gigabit Ethernet Alliance. Link Gigabit Ethernet Alliance. `http://www.gigabit-ethernet.org/`.

[11] Inc. Myricom. Link Myricom. `http://www.myri.com`.

[12] Richard P. Martin, Amin M. Vahdat, David E. Culler, and Thomas E. Anderson. Effects of communication latency, overhead, and bandwidth in a cluster architecture. *Computer Architecture News*, May 1997.

[13] David A. Rusling. *The Linux Kernel*. Linux LDP, March 1998.

[14] W. Richard Stevens. *UNIX Network Programming*. Prentice Hall, 1990.

[15] Dave Dunning, Greg Regnier, Gary McAlpine, Don Cameron, Bill Shubert, Frank Berry, Anne Marie Merritt, Ed Gronke, and Chris Dodd. The Virtual Interface Architecture. *IEEE Micro*, 18(2), March 1998.

[16] Thorsten Von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net:a User-Level network interface for parallel and distributed computing. In *In Proceedings of the 15th SOSP*, Copper Mountain, CO, December 1995.

[17] Thorsten Von Eicken and al. Active Messages: A mechanism for integrated communication and computation. In *In Proceedings of the 19th Symp. Computer Architecture*, Gold Coast, Qnd. Australia, May 1992.

[18] S. Pakin, V. Karamcheti, and A.A. Chien. Fast Messages: efficient, portable communication for workstation clusters and MPPs. *IEEE Concurrency*, April 1997.

[19] VIA Org. Specification for the Virtual Interface Architecture. `http://www.viarch.org/default.htm` .

[20] Intel. Intel and VI Architecture. `http://developer.intel.com/design/servers/vi/`.

[21] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification*, December 1997.

[22] Thorsten Von Eicken and Werner Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, November 1998.

[23] Intel. *Intel Virtual Interface (VI) Architecture Developer's Guide*, September 1998.

[24] National Energy Research Scientific Computer Center. M-VIA: A high performance Modular VIA for Linux. `http://www.nersc.gov/research/FTG/via/` .

[25] Philip Buonadonna, Andrew Geweke, and David Culler. Implementation and analysis of the Virtual Interface Architecture. November 1998.

[26] MPI. Link MPI. `http://www.epm.ornl.gov/ walker/mpi/index.html`
`http://www.mpi.nd.edu/lam/`
`http://www.mcs.anl.gov/mpi/`
`http://science.nas.nasa.gov/Software/NPB/` .

[27] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.

[28] Ohio Supercomputer Center. *MPI Primer/Developing with LAM*. The Ohio State University, November 1996.

[29] Nicholas Nevin. *Porting the LAM MPI 6.0 Communication Layer*. LAM team, May 1996.

[30] Alessandro Rubini. *LINUX Device Drivers*. O'Reilly & Associates, Inc., 1998.

[31] Beck, Bohme, Dziadzka, Kunitz, Magnus, and Verworner. *Linux kernel internals*. Addison Wesley, 1998.