UNIVERSITÀ DEGLI STUDI DI PARMA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

PRIMITIVE ASINCRONE DI VIRTUAL INTERFACE ARCHITECTURE (VIA) PER UNA LIBRERIA A SCAMBIO DI MESSAGGI IN CLUSTER DI MULTIPROCESSORI

Relatore:

Chiar.mo Prof. Ing. GIANNI CONTE

Correlatori:

Dott. Ing. MONICA REGGIANI

Dott. Ing. MARCO PANELLA

Tesi di laurea di:

MASSIMO MERLI

ANNO ACCADEMICO 2000–2001

A mamma e papà.

Sono in dovere di formulare sentiti ringraziamenti al prof. ing. Gianni Conte, per avere creduto in un progetto così interessante ed innovativo.

Ringrazio altresì l'ing. Monica Reggiani, che ha permesso con il suo vivo supporto e i preziosi consigli la realizzazione di questo lavoro di tesi.

Vorrei esprimere una grande riconoscenza all'ing. Marco Panella il quale, con pazienza infinita, mi ha permesso di migliorare quanto da lui realizzato in mesi di duro lavoro.

Ringrazio Michela, a cui devo il preziosissimo aiuto e conforto donatomi in questi lunghi anni di studio universitario.

Riconosco, infine, alla mia stupenda famiglia il grandissimo dono di avermi permesso il raggiungimento di questo straordinario obbiettivo.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco degli algoritmi

CAPITOLO 1

Introduzione

L'evoluzione e la ricerca informatica di questi anni sono caratterizzate da una notevole ramificazione in più campi applicativi, tutti volti ad accelerare od ottimizzare le elaborazioni di quantità di dati in continua crescita.

Accanto alla naturale evoluzione dei sistemi di calcolo, che si identifica nello sviluppo e nella successiva ottimizzazione di microprocessori sempre più veloci, si va affiancando un settore di ricerca che ha caratterizzato le ultime innovazioni del calcolo scientifico: lo studio di tecniche di elaborazione parallela.

Quando, infatti, la potenza di calcolo di una singola macchina non è sufficiente a sopperire a determinati carichi di lavoro, seppure si tratti di macchina "aggiornata", la più semplice via atta ad incrementare ulteriormente le prestazioni resta l'affiancamento di più macchine, secondo la strategia del "divide et impera".

Ammessa quindi la possibilità di dividere la sequenza di elaborazione in diverse parti indipendenti tra loro (codice parallelizzabile), sono innumerevoli le tecniche di parallelismo adottate, ognuna delle quali è caratterizzata da particolari pregi o difetti che hanno determinato il loro successo, la loro evoluzione e la loro storia.

Di fatto, trovandosi in un settore, quale l'informatica, in evoluzione rapidissima, ognuna delle innovazioni è stata soggetta ad una sorta di selezione naturale secondo un criterio primario piuttosto "crudo", individuabile nel rapporto costo/prestazioni. Così gioielli tecnologici del passato quale ad esempio la CM di Thinking Machines e le successive realizzazioni, non hanno avuto un'evoluzione a causa dell'eccessiva personalizzazione dell'architettura hardware, che ha portato a livelli esorbitanti i costi realizzativi di una produzione in serie.

Contemporaneamente al declino di questi colossi si è diffusa, con una capillarità inedita, la tecnologia ”x86”, la quale ha permesso un progresso tecnologico tanto veloce quanto proficuo per merito della standardizzazione di protocolli, interfacce e bus di comunicazione. Se si era abituati a sviluppi tecnologici ”orizzontali”, in cui il singolo produttore avanzava la ricerca su ogni aspetto della propria macchina (dal microprocessore ai flussi dati, alle memorie, alle comunicazioni con l'esterno), si è dunque passati, grazie alla succitata standardizzazione, ad uno sviluppo verticale di ogni componente hardware della macchina, favorendo così una importante accelerazione della ricerca con risultati sempre più sorprendenti.

Questo fenomeno ha fatto sì che ogni produttore si specializzasse in un determinato settore, ivi concentrando i propri sforzi tecnologici, per poi integrare i risultati con quelli degli altri costruttori grazie agli standard in atto.

Questo aspetto ha apportato un notevole beneficio anche allo sviluppo delle tecniche di elaborazione parallela, orientando la ricerca verso l'ottimizzazione di protocolli software basati su hardware già esistente, per altro non espressamente concepito per ”parallel processing”. Si parla, a questo proposito, di *cluster* (dall'inglese grappolo, gruppo) di personal computer, inteso come una serie di macchine, interconnesse tra loro, in grado di operare come una macchina singola tramite protocolli di interfacciamento e comunicazione espressamente studiati per lo scopo.

Saranno esaminati in questa tesi alcuni dei più significativi aspetti di uno dei protocolli di comunicazione ad alte prestazioni: *Virtual Interface Architecture* (VIA), protocollo sviluppato da un consorzio di grandi imprese (tra cui Compaq, Intel e Microsoft) appositamente costituito. Uno dei compiti a cui è destinato il cluster di personal computer costituito presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Parma riguarda il porting di tale architettura all'interno dell'ambiente di programmazione distribuita *Local Area Multicomputer / Message Passing Interface* (LAM/MPI) [9].

1.1 Il Cluster di PC

Il cluster di Personal Computer è quindi definito come un insieme coordinato di Personal Computer, sistema di interconnessione di rete e protocolli di comunicazione. Ognuno dei tre componenti deve essere scelto in funzione della tipologia di lavoro a cui il cluster viene dedicato. Inoltre, molti sistemi operativi presenti sul mercato sono legati ad un insieme di software standard che non permettono il raggiungimento delle potenziali prestazioni dell'hardware su cui lavorano. Infine, l'attraversamento dell'infrastruttura di rete rappresenta spesso il collo di bottiglia del sistema a cluster. Ciò è in molti casi dovuto alla sostanziale "obsolescenza" delle librerie di gestione della rete rispetto all'hardware e soprattutto alle *Local Area Network* (LAN) che supportano il cluster. Occorre d'altra parte specificare che i protocolli di comunicazione non sono sempre studiati per massimizzare le prestazioni, ma spesso lo scopo principale degli standard proposti è l'affidabilità delle trasmissioni.

Quando apparvero i primi software che oggi si inquadrano tra i sistemi operativi, l'hardware era così costoso che il migliore tra essi era quello che al meglio sfruttava le potenzialità dell'hardware stesso. Per contro, oggi si richiede ad un Sistema Operativo il miglior rapporto prestazioni/prezzo, che non garantisce affatto un ottimo sfruttamento delle risorse hardware a disposizione, per altro a prezzi ormai estremamente contenuti.

Lo stato dell'arte ora mostra una situazione in cui i costi del software sono di gran lunga superiori rispetto a quelli dell'hardware. Negli anni sessanta, un programmatore era spinto ad ottimizzare al limite il suo codice pur di sfruttare appieno le potenzialità degli elaboratori di allora; oggi, per contro, se un codice ha bisogno, ad esempio, del doppio della memoria RAM disponibile su di una macchina, si preferisce di gran lunga l'aggiunta di memoria fisica all'impiego di risorse uomo per l'ottimizzazione di tale codice sull'hardware a disposizione: l'efficienza nell'uso delle risorse hardware non è più prioritaria, potendo accedere a nuovi ed economici sistemi per raggiungere i richiesti livelli prestazionali.

Le LAN iniziarono a giocare un ruolo cruciale nell'operatività dei sistemi solo nel momento in cui la focalizzazione riguardante il supporto dei sistemi operativi si stava spostando dall'efficiente utilizzo di risorse hardware alla ricerca sull'ottimizzazione delle interconnessioni di componenti standard. Le comunicazioni interprocesso, dall'inglese *InterProcess Communication* (IPC), furono inizialmente concepite esclusivamente come eventi legati agli sporadici accessi alle risorse remote, mentre nessuno avrebbe pensato ad esecuzioni di applicazioni parallele su cluster. La standardizzazione per l'interoperabilità e la portabilità fu la chiave che portò allo sviluppo dei tradizionali protocolli LAN e la loro efficienza non fu inizialmente percepita come un problema rilevante.

Appurato, inoltre, che le prestazioni degli attuali microprocessori sono cresciute di più di mille volte rispetto ad una quindicina di anni fa, l'unico attuale ostacolo ad una larga diffusione dei Cluster di Personal Computer è la cronica inefficienza dei protocolli standard di comunicazione di rete, incorporati in maniera forse troppo radicata nelle architetture dei sistemi operativi durante l'ultima evoluzione delle tecnologie di comunicazione.

Di fatto, quindi, per avvantaggiarsi delle enormi potenzialità degli strumenti hardware oggi a disposizione occorre reingegnerizzare i protocolli di comunicazione per eliminare i numerosissimi colli di bottiglia formatisi per i motivi suesposti. Numerosi sono i progetti finalizzati a tale scopo, che si sono formati ed evoluti in questi ultimi anni, ognuno volto a sviluppare e migliorare uno o più dei numerosi aspetti dell'ottimizzazione dei protocolli di comunicazione.

1.2 Aspetti prestazionali

Nell'analisi delle prestazioni di un'architettura di comunicazione, riveste un'importanza fondamentale la capacità di riconoscere che il tempo trascorso per il trasferimento di un messaggio da un punto all'altro della rete (inteso come tempo intercorso tra la spedizione del messaggio da parte di un nodo e il ricevimento dello stesso da parte del nodo destinatario) può essere suddiviso in più componenti, ognuna imputabile ad un diverso tratto del percorso che il messaggio segue per arrivare a destinazione [8]. I principali stadi riconoscibili in qualsiasi sistema sono:

- il processore;
- l'interfaccia di rete;
- la rete.

1.2.1 La latenza di comunicazione

Il parametro prestazionale di gran lunga più significativo nella misura delle performance di un sistema di comunicazione applicato ai cluster è la latenza. Non vi è un accordo consolidato sull'esatta definizione di latenza e su come essa vada misurata. Lo scopo della misura di latenza è la caratterizzazione della velocità del sistema in esame per sincronizzare due processi cooperanti mediante lo scambio di messaggi. Più veloce risulta il sistema, individuato nel processo di comunicazione, più alte risultano le prestazioni per applicazioni ad alto livello di sincronizzazione [2].

Dal punto di vista di un'applicazione a scambio di messaggi, un puro scambio di informazioni riguardanti la sincronizzazione può essere operato mediante il semplice invio di un messaggio. Di fatto, il compito di un simile messaggio non richiede che esso abbia una particolare semantica, se non l'essere di dimensione più contenuta possibile, addirittura preferibilmente vuoto. La definizione più comune di latenza (unilaterale) è il tempo impiegato per spedire un messaggio di lunghezza minima da un trasmettitore ad un ricevente, dall'istante in cui il mittente inizia l'operazione di send al momento in cui al destinatario viene notificato l'arrivo del messaggio.

Normalmente, il processo mittente e quello destinatario sono applicazioni. La tecnica più comune per il calcolo della latenza è l'utilizzo di un semplice ping pong. Il processo "ping" spedisce un messaggio e lo riceve indietro dal processo "pong". "Ping", inoltre, misura il tempo necessario alla esecuzione della coppia di operazioni send/receive (*round-trip time*), mentre il processo “pong” esegue l’operazione duale. Queste istanze sono ripetute un elevato numero di volte in modo da poter effettuare il calcolo di una media significativa, possibilmente scartando i primi cicli spesso affetti da ritardi di “riscaldamento”. La latenza è infine individuata nella metà del tempo medio di “round-trip”.

1.2.2 Il modello $LogP$

È però altrettanto importante basare l’analisi su parametri che non siano eccessivamente influenzati dalla particolare tecnologia hardware utilizzata, ma che diano un risultato utilizzabile per confronti tra protocolli di trasmissione ambientati su hardware non necessariamente paragonabili.

Il modello $LogP$[1] [8] riesce a fornire una valutazione ben ponderata, essendo i suoi indici influenzati dalle prestazioni delle risorse chiave del sistema in esame, ma non dalla loro struttura.

Secondo tale modello, un sistema multiprocessore a memoria distribuita in cui i diversi processori comunicano tra loro con messaggi punto-punto è caratterizzato da quattro parametri:

- L**, Latenza**: ovvero il tempo necessario alla spedizione di un messaggio, contenente un numero di byte tendente a zero, dal processo sorgente al destinatario;

- o**, overhead**: definito come la quantità di tempo necessaria ad un processore per operare la ricezione o la trasmissione di ogni singolo messaggio, tempo in cui il processore stesso non può dedicarsi ad altri compiti;

- g**, gap**: ovvero il minimo intervallo di tempo tra la trasmissione o la ricezione di due bit consecutivi eseguite da un unico processore. Può essere anche visto come il tempo necessario ad un messaggio per attraversare il collo di bottiglia della larghezza di banda del sistema;

[1] Latency, overhead, gap, Processors

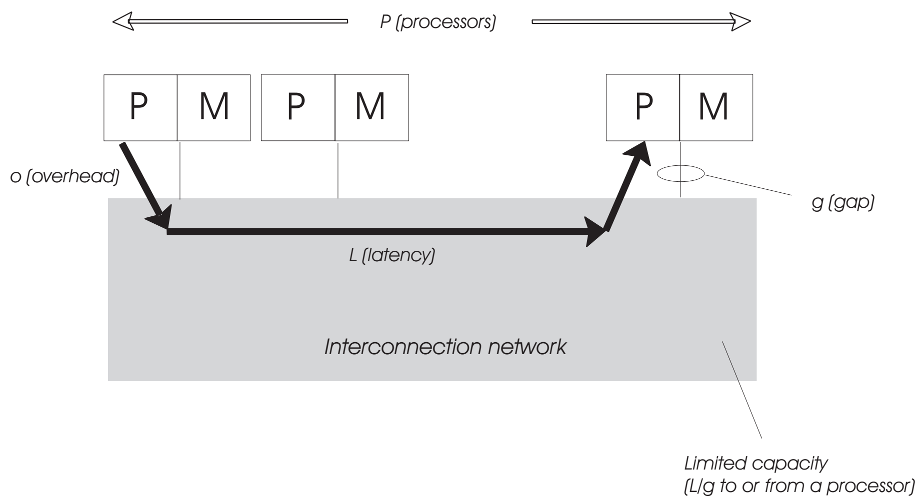

Figura 1.1: Modello di Macchina Astratta $LogP$.

- P**, Processors**: il numero di moduli processore/memoria presenti nel cluster in esame.

Le quantità L, o e g vengono solitamente espresse in unità di tempo. Si usa assumere che una rete sia caratterizzata da capacità finita, così che, al più, possa essere trasferita una quantità di messaggi pari a $\lceil L/g \rceil$ tra qualsiasi dei processori presenti nel cluster. Se un processore dovesse tentare di varcare questo limite, si porrebbe in una condizione di stallo finché la fine di altri processi in corso non permetta di processare tale trasmissione pur rimanendo nei limiti stabiliti dal rapporto suddetto.

La più semplice operazione di comunicazione, consistente nello spedire un singolo pacchetto da una macchina ad un'altra, richiede un tempo pari a $L + 2o$. La latenza include, così, sia il tempo speso all'interno dell'interfaccia fisica di rete, sia il tempo occorso per l'attraversamento della rete cablata. Questi due tempi sono indistinguibili dal processore. Di conseguenza, il tempo occorrente per una operazione con richiesta di conferma necessita di $2L + 4o$ unità di tempo. Il processore che esegue la richiesta e quello che deve inviare la risposta si trovano entrambi occupati per un tempo pari a $2o$.

Un'altra grandezza significativa è la banda disponibile per processore (o tasso di trasmissione per unità di tempo), rappresentata dalla quantità $1/g$. A seconda del tipo di macchine coinvolte, la banda disponibile può essere determinata dalla banda effettivamente offerta dalla rete fisica o da altre limitazioni della struttura. In molte macchine questo parametro è strettamente condizionato dalla velocità dell'interfaccia di rete nel processare i vari pacchetti, piuttosto che dalle prestazioni intrinseche della rete propria.

Scopo principale dei progetti di ottimizzazione della programmazione parallela è ovviamente l'aumento delle prestazioni del cluster su cui tale programmazione è applicata. Tale ottimizzazione può vedersi direttamente applicata su alcuni dei parametri del modello $LogP$. Se si considera, come sopra descritto, la più semplice operazione di comunicazione, essa coinvolge principalmente i due parametri L ed o. Il protocollo VIA è caratterizzato da una bassa latenza di comunicazione, intesa però come insieme dei due parametri L ed o, nella forma $2L+4o$ vista in precedenza.

1.3 Il Cluster ParMa^2

Il presente lavoro di tesi è stato sviluppato al Dipartimento di Ingegneria dell'Informazione dell'Università degli Studi di Parma, sul Cluster di Personal Computer ParMa^2, nato nel 1995 allo scopo di assemblare un sistema computazionale ad alte prestazioni e a basso costo, con componenti assolutamente standard e di facile e veloce reperibilità sul mercato [1].

Attualmente è in funzione la terza evoluzione del cluster iniziale, caratterizzata da componenti hardware e software di seguito descritti.

1.3.1 L'architettura Hardware

Il Cluster attualmente funzionante è formato da otto personal computer:

- Quattro PC Symmetric Multi Processing così configurati:
  - Motherboard Asustek P2B-DS, chipset Intel 440BX, 100 MHz FSB, con controller SCSI Adaptec AIC7890 UW integrato;

  - 256 Mb Synchronous Dynamic RAM in 2 DIMM da 128 Mb con specifiche PC100;
  - 2 Pentium II 450 MHz, con 32 kb di Cache L1 e 512 Kb di cache L2;
  - HD Seagate Medalist 4.5 Gb SCSI UW 5400 rpm;
  - LAN card D-LINK DFE-500TX, con chip Tulip 21140;
  - SVGA S3 Virge GX2 AGP con 4 Mb RAM;
- Un PC Symmetric Multiprocessing DELL PowerEdge 9600 così configurato:
  - Motherboard Intel con Chipset 450NX, 133 MHz FSB, con 2 controller SCSI integrati Adaptec AIC7890 ed uno AIC7860 dedicato al lettore CD-ROM SCSI;
  - 512 Mb Synchronous Dynamic RAM con specifiche PC133;
  - 4 Pentium III Xeon 550 MHz, con 64 kb di cache L1 e 512 di cache L2;
  - 2 HD IBM U2W-SCSI DMVS-09U 9.1 Gb, 10000 rpm in bay Hot-Swap;
  - LAN card Intel Ether Express 100+ Fast Ethernet
  - SVGA ATI Mach64 Rage Pro
- Infrastruttura di rete e interfaccia utente così composte:
  - due Pentium Pro 200 Mhz ad uso interfaccia utente;
  - un Pentium 100 con funzioni di Router verso l'esterno;
  - un LAN Switch Cisco Micro Switch 1548 Fast Ethernet.

1.3.2 I sistemi operativi e il software

La scelta del sistema operativo da installare su ogni personal computer del cluster è caduta sul gratuito Linux, disponibile con ampio supporto ad architetture multiprocessore e a sistemi *Beowulf* [12]. Tale sistema operativo possiede importanti

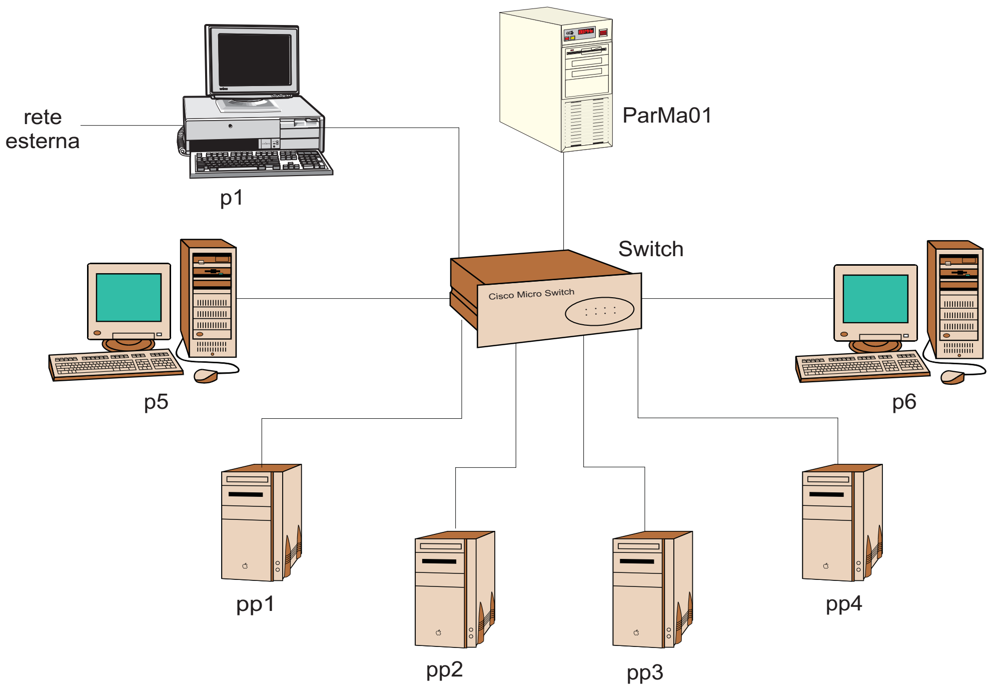

Figura 1.2: Il cluster di personal computer *ParMa*^{2}.

caratteristiche che lo rendono assolutamente idoneo per applicazioni di ricerca. Esso è coperto da licenza GNU [4], grazie alla quale è possibile accedere ai sorgenti ed apportarvi ogni tipo di modifica. È inoltre caratterizzato da alta affidabilità, robustezza e protezione dei processi. Il kernel è sviluppato da team di ricercatori a livello mondiale ed ogni nuova release è prontamente messa a disposizione dell'utenza, con sorgenti e documentazione.

Vi è, inoltre, grande disponibilità di applicativi, molti dei quali coperti dalla stessa licenza GNU, quindi anch'essi gratuiti ed "open source". Sono anche fruibili driver per un grande numero di periferiche, con la copertura totale di quelle a più larga diffusione. Il protocollo TCP/IP è pienamente supportato e la sua realizzazione risulta particolarmente efficiente.

Il sistema operativo Linux è disponibile in numerose *distribuzioni*. Ogni distribuzione comprende una determinata versione del sistema operativo, corredata di ambienti personalizzati ed user-friendly che guidano l'utente nel processo di installazione e comprendente un gran numero di pacchetti applicativi. Ogni distribuzione si distingue per una propria organizzazione delle configurazioni del sistema, tutte in ogni caso conformi allo standard UNIX.

La distribuzione installata nel cluster ParMa^{2}è RedHat [7], versione 6.4, per i quattro nodi biprocessore. È installata la RedHat 6.3 sul quadriprocessore, mentre è presente la Debian [5], v. 2.0, sul router e sulle due macchine ad uso interfaccia utente.

1.4 Sommario

Dopo avere esaminato in quete pagine i vari aspetti dell'ambiente operativo e delle strutture hardware a disposizione, segue un sommario dei contenuti dei successivi capitoli.

Nel **capitolo 2** è presentata una introduzione al funzionamento dell'architettura VIA, con particolare riferimento alla modalità di trasferimento dei dati e all'utilizzo dei descrittori. Saranno toccati anche altri aspetti quali la gestione della memoria, i modelli di connessione, i livelli di affidabilità e la gestione degli errori.

Il **capitolo 3** descrive in dettaglio aspetti e caratteristiche di due delle più diffuse realizzazioni di VIA: una hardware, CLAN, realizzata su schede LAN GigaNet [6] ed una software, *Modular VIA* (MVIA) [10], realizzata in emulazione hardware e compilata come modulo del sistema operativo Linux, la quale si appoggia, poi, sulle schede tradizionali Fast Ethernet. Saranno analizzate caratteristiche, vantaggi e svantaggi delle due soluzioni e saranno messe a confronto sia dal punto di vista realizzativo, sia dal punto di vista prestazionale. Saranno analizzati in dettaglio gli algoritmi di test forniti dai rispettivi produttori, approfondendo le modalità di funzionamento ed il significato dei risultati conseguentemente ottenuti.

Nel successivo **capitolo 4** è presentato l'ambiente di programmazione distribuita LAM / MPI, con l'analisi del paradigma Message Passing e della struttura dello standard MPI, del quale è fornita una breve descrizione delle specifiche. Si passa quindi all'analisi della struttura software di LAM e delle sue modalità di comunicazione. Il capitolo si conclude con un breve accenno all'analisi del Lower Layer e strutture dati.

Il **capitolo 5** è interamente dedicato al porting dell'architettura LAM su Virtual Interface Architecture, con particolare riferimento all'integrazione di una nuova primitiva resasi disponibile nell'emulazione MVIA dell'architettura. Sarà esaminato in dettaglio il funzionamento del codice prima delle innovazioni apportate e sarà giustificato il miglioramento e i vantaggi acquisiti dall'introduzione di questa nuova primitiva. Verrà analizzato il funzionamento della struttura base dell'interfaccia a basso livello di LAM/MVIA, con particolare riferimento alla gestione della ricezione dei messaggi da parte del peer.

Si conclude il presente lavoro con il **capitolo 6**, in cui saranno tratte le conclusioni del lavoro svolto e saranno delineate possibili vie per futuri ampliamenti dell'integrazione tra queste due architetture.

CAPITOLO 2

Virtual Interface Architecture

Un metodo utilizzato per velocizzare il percorso del messaggio tra gli applicativi attraverso la rete consiste nell'eliminare le chiamate coinvolgenti il kernel del sistema operativo dal cammino critico percorso dai messaggi medesimi. In assenza di tale intervento, le applicazioni possono scambiare tra loro messaggi con un apprezzabile miglioramento della latenza e delle prestazioni di comunicazione via rete. Come già accennato in precedenza, gli aspetti da considerare per raggiungere questi obbiettivi sono [13]:

- Rendere disponibile alle applicazioni un accesso diretto all'hardware di rete in modo da assicurare sufficiente protezione, evitando che le applicazioni stesse interferiscano l'una con l'altra;

- Mettere a disposizione una interfaccia di programmazione efficiente e versatile, in modo da rendere possibile, oltre ovviamente all'accesso alla rete, il controllo di buffer, schedulazioni di processi, indirizzamento di memoria. È inoltre importante la possibilità di poter effettuare un porting di tutto su una base relativamente ampia di piattaforme hardware.

- Utilizzare in maniera ragionevole le risorse: le applicazioni devono considerare il costo dei trasferimenti *Direct Memory Access* (DMA), che relazionano indirizzi fisici di memoria ad altri virtuali. Contemporaneamente, i dettagli realizzativi dell'interfaccia devono essere nascosti al programmatore ed il controllo ultimo dell'allocazione delle risorse deve sempre e comunque spettare al sistema operativo.

- Evitare, come si accennava in precedenza, che il cammino del messaggio verso la rete coinvolga il kernel del sistema operativo, che a sua volta gestisce un ruolo fondamentale nei protocolli di rete tradizionali.

La Virtual Interface Architecture è studiata come una infrastruttura di comunicazione per le *System Area Network* (SAN), con l'intento di realizzare una nuova forma di protocolli per cluster in grado di facilitare le comunicazioni di questo nuovo tipo di infrastruttura di rete. Il nucleo delle specifiche di VIA consiste nel fornire servizi di rete che coinvolgono il sistema operativo solamente in isolate operazioni di setup della comunicazione, soprattutto quelle che hanno direttamente a che fare con l'affidabilità e la sicurezza della comunicazione e la protezione della struttura hardware.

A tale scopo occorre progettare un modello a cui l'hardware in questione deve fare riferimento, ed un altro modello a cui deve essere consona l'interfaccia di programmazione a livello utente. Tutto ciò deve, inoltre, sfruttare i risultati della ricerca sulla programmazione distribuita, in modo da mantenere i costi di realizzazione relativamente contenuti.

Le specifiche ottenute descrivono il modello di una libreria di comunicazione chiamata *Virtual Interface Provider Library* (VIPL). Questa specifica rivela solo una parte del meccanismo di trasporto sottostante. Di rilevante importanza è il *descrittore*, consistente in una struttura che descrive la locazione e la dimensione di un buffer di dati. Questa struttura è a diretta disposizione delle applicazioni e, inoltre, è di costituzione fissa. Al contrario si hanno a disposizione strutture a coda dinamiche che raccolgono insiemi di descrittori, le quali, assieme ad opportuni meccanismi di "doorbell", sono opache agli applicativi grazie alla libreria utente VIPL.

2.1 Aspetti generali

L'obbiettivo principale di tutte le interfacce di rete a livello utente consiste nel fornire un accesso protetto alle risorse di comunicazione condivise. Nelle reti tradizionali, il processo utente esegue una chiamata di sistema per effettuare la comunicazione. Successivamente, il traffico viene multiplato da più processi richiedenti verso la rete tramite un device driver che gestisce la scheda di interfaccia rete vera e propria, chiamata *Network Interface Card* (NIC). L'operazione inversa avviene alla ricezione, demultiplando il traffico ricevuto verso ogni singolo processo destinatario.

Ora, mentre non risulterebbe difficoltoso assegnare ad ogni singolo processo una NIC, ciò non permetterebbe la condivisione di quest'ultima risorsa hardware da parte di più processi appartenenti allo stesso nodo. È quindi compito del NIC e del suo firmware farsi carico della multiplazione e demultiplazione del traffico verso la rete, sollevando il kernel del sistema operativo da tale compito.

In VIA, il sistema operativo si occupa della creazione e della distruzione di *Virtual Interface* (VI), mappando parte delle risorse del NIC nello spazio di memoria del processo utente associato. La VI rappresenta la terminazione virtuale del nodo di rete, svolgendo il ruolo delle code di descrittori DMA di una tipica interfaccia di rete.

La Virtual Interface Architecture è un protocollo di comunicazione orientato alla connessione, punto a punto, che permette al sistema operativo di intervenire nella connessione e nella disconnessione delle singole interfacce virtuali e sostituisce quest'ultimo nei successivi controlli sull'utilizzo della rete continuando a monitorare la quantità e le specifiche (ad esempio, il numero di porte) delle singole connessioni.

Nella figura 2.1 possiamo avere una superficiale idea della struttura di VIA. L'inizializzazione, la connessione e la gestione delle singole VI sono assicurate interagendo con un driver attraverso la libreria a livello utente. Questo driver ha il

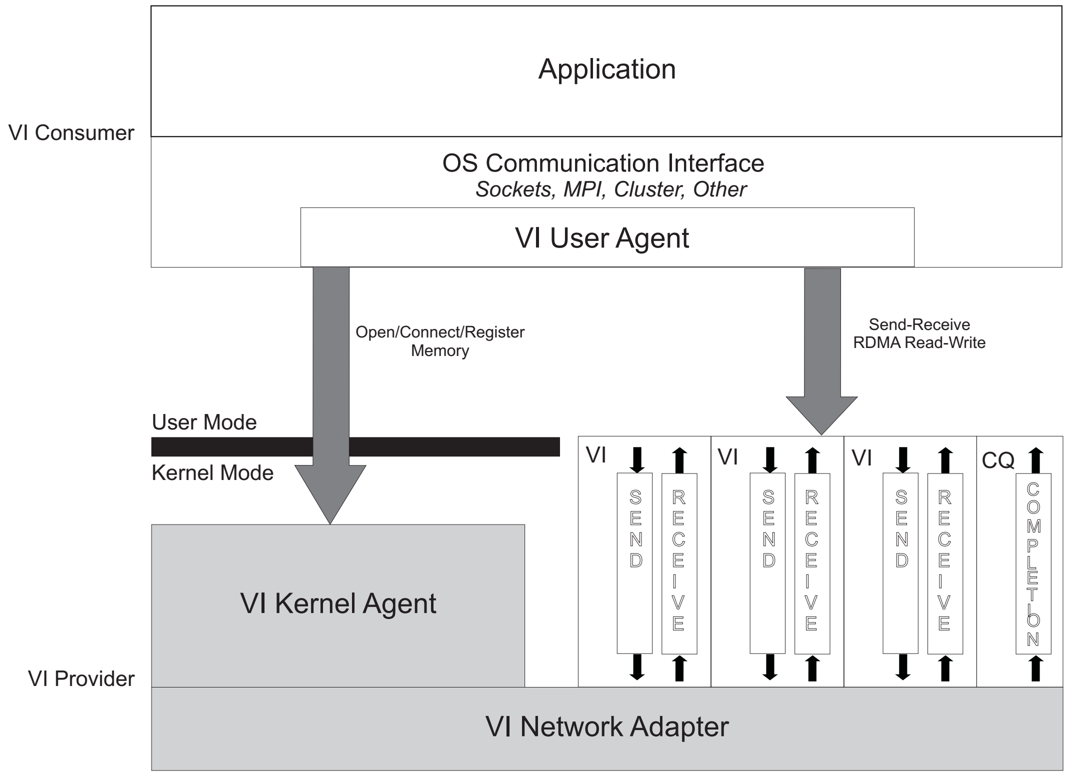

Figura 2.1: Vista d'insieme del modello architetturale di VIA.

compito di accedere ad alcune porzioni delle risorse del NIC per potergli notificare l'avvenuta creazione delle VI, la connessione ed altro, compiti tenuti nascosti all'applicazione e al livello utente.

Ogni VI è costituita da due code, chiamate *send* e *receive*, illustrate in figura 2.1 e una coppia di *doorbell*. Nessuna di queste strutture è direttamente visibile all'applicazione e l'accesso ad esse viene fatto esclusivamente per mezzo della VIPL.

Sia le code, sia i buffer dati sono allocati nella memoria di sistema in cui il NIC ha accesso diretto. Queste regioni di memoria sono classificate come memoria registrata. La necessità di un'applicazione di trasmettere o ricevere un messaggio è comunicata tramite il posting di un apposito descrittore e, mediante le *Application Program Interface* (API) VIPL, la notifica tramite la rispettiva doorbell al NIC. La mappatura degli indirizzi di memoria è garantita dal sistema operativo che, in tal modo, provvede alla necessaria protezione senza di fatto entrare nel cammino critico della comunicazione.

2.2 L'architettura VI

Sono quattro i principali componenti della struttura di comunicazione VIA: le Virtual Interface, le Completion Queue, i VI Provider e i VI Consumer. A scopo introduttivo, anticipiamo che il VI Provider è costituito dal NIC fisico e da un agente software, mentre il VI Consumer è rappresentato dal programma applicativo e dalle funzionalità di comunicazione del sistema operativo. Analizziamo in dettaglio ognuno di questi componenti:

2.2.1 La Virtual Interface

La Virtual Interface è il meccanismo che permette al VI Consumer di accedere direttamente al VI Provider, per eseguire la transazione comunicativa.

È illustrata in figura 2.2 la struttura schematica di una VI. Si distinguono due

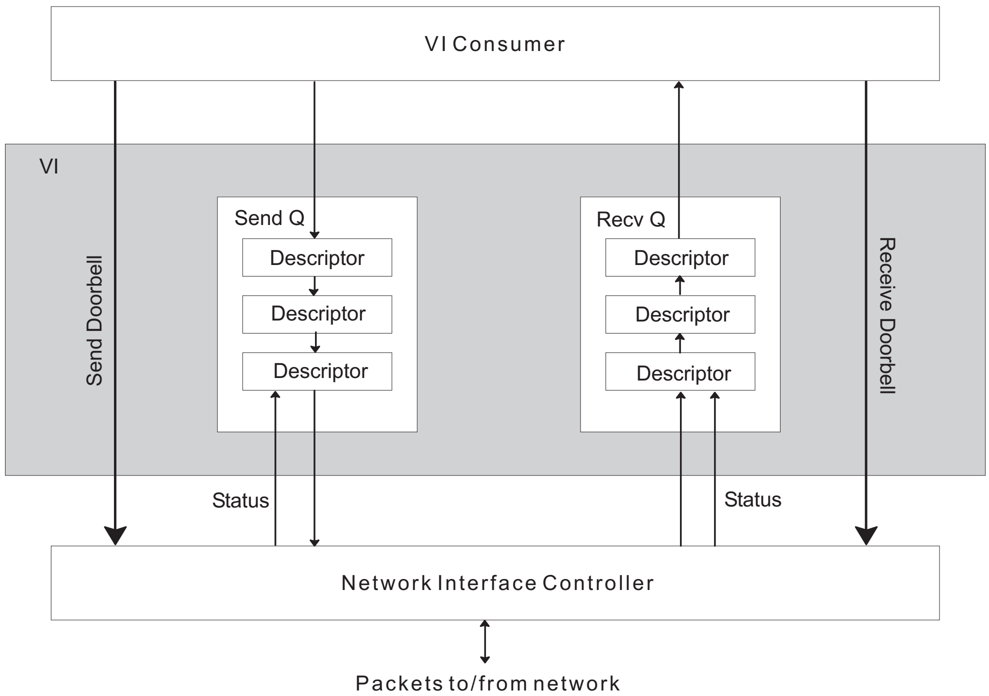

Figura 2.2: Esemplificazione schematica di una Virtual Interface.

code: una di ricezione (receive queue) ed una di trasmissione (send queue). In seguito alla necessità di spedizione o ricezione di un messaggio, il VI Consumer invia la richiesta, sotto forma di descrittore, alla relativa coda. Come accennato in precedenza, un descrittore è una struttura contenente tutte le informazioni di cui necessita il VI Provider per sopperire alla corrente richiesta, come ad esempio i puntatori ai buffer dati. Il VI Provider processa in modo asincrono il descrittore inviato, marcandolo con un flag di stato una volta completato. Il VI Consumer rimuove il descrittore completato dalla coda relativa e lo utilizza per istanziare le conseguenti richieste. Ad ogni coda è asociata una doorbell, utilizzata per notificare all'interfaccia di rete l'avvenuto invio di un descrittore sulla coda medesima. Anche la doorbell è direttamente gestita dall'interfaccia di rete e non richiede alcun intervento del sistema operativo.

2.2.2 Il VI Provider

Il VI Provider è l'insieme di componenti hardware e software responsabile dell'istanziazione della Virtual Interface. Consiste di una interfaccia di rete (NIC) e di un agente software (Kernel Agent).

Il NIC gestisce le VI e le Completion Queue (vedi in seguito) e si occupa direttamente delle operazioni di trasferimento dati.

Il Kernel Agent è solitamente rappresentato da un driver supportato dal fornitore dell'interfaccia fisica di rete, che si occupa dell'inizializzazione e della gestione delle risorse necessarie al mantenimento della VI tra il Consumer e il NIC. Queste funzioni includono la creazione e la distruzione delle VI, la creazione e l'annullamento della connessione, la gestione degli interrupt, la gestione della memoria di sistema utilizzata dal NIC ed infine il trattamento degli errori. Il VI Consumer accede al Kernel Agent facendo uso dei meccanismi standard del sistema operativo, come ad esempio le chiamate di sistema.

2.2.3 Il VI Consumer

Il VI Consumer rappresenta l'utente della Virtual Interface. Mentre il programma applicativo è l'utente finale del servizio di comunicazione, l'applicazione medesima accede a tale servizio attraverso interfacce standard di programmazione del sistema operativo, quali ad esempio Socket o Message Passing Interface. Questa caratteristica di programmazione è generalmente fornita in forma di libreria.

Nel nostro caso è proprio questa funzionalià ad effettuare le chiamate di sistema al Kernel Agent per creare la VI nel sistema locale e per connetterla, di conseguenza, alla VI del sistema remoto con cui si ha la necessità di comunicare. Una volta stabilita la connessione, tale funzionalità invia le richieste di spedizione o di ricezione generate dall'applicazione direttamente alla VI locale.

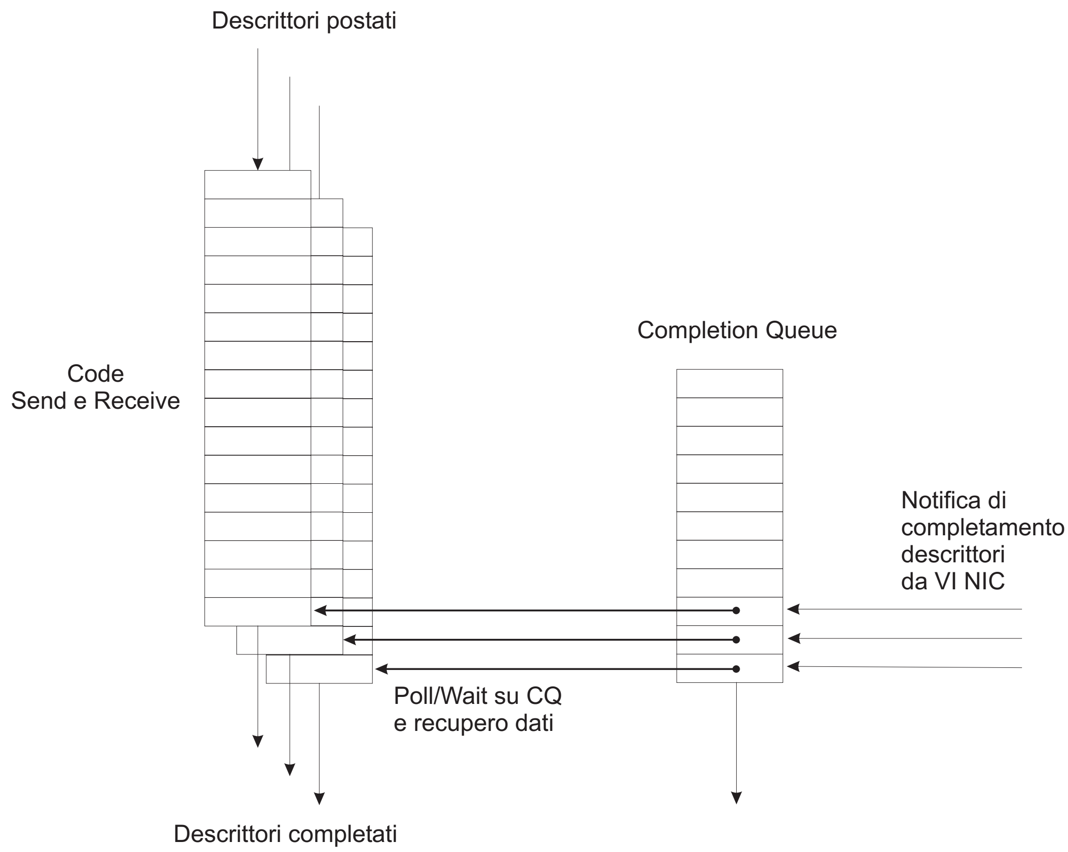

Figura 2.3: Struttura schematica della Completion Queue.

2.2.4 Le Completion Queue

Una Completion Queue permette al VI Consumer di ogni nodo di raccogliere tutte le notifiche di completamento dei descrittori sulle rispettive code di spedizione e ricezione. Ad ogni VI presente sul nodo è associata una Completion Queue. Questa corrispondenza è istanziata alla creazione delle VI. Una volta associata una VI alla relativa Completion Queue, ogni operazione di sincronizzazione deve necessariamente riferirsi alla CQ e non più alla VI. Ancora, esattamente come nelle code tradizionali, la notifica di avvenuto completamento di un descrittore è effettuata alla CQ dal NIC senza utilizzo di interrupt e, allo stesso modo, il VI Consumer può sincronizzarsi su un completamento senza l'intervento del kernel.

2.3 Modelli di trasferimento

Analizziamo ora più nel dettaglio i due tipi di trasferimento dati nominati in precedenza: il modello Send/Receive e il modello ad accesso diretto alla memoria remota (RDMA).

2.3.1 Modello Send/Receive

In questo modello di comunicazione, appoggiato a canoni classici, il VI Consumer nel nodo locale specifica sempre l'indirizzo dei dati. Dal lato di spedizione, il processo specifica le zone di memoria contenenti il dato che deve essere spedito. Dal lato ricevente, il processo specifica la regione di memoria in cui andranno scritti i dati in arrivo. Data una singola connessione, si crea una corrispondenza biunivoca tra il descrittore sulla coda di spedizione ed il descrittore sulla Receive Queue dal lato ricevente. Il VI Consumer al lato ricevente effettua un'operazione di *Pre-Posting* di un descrittore: viene inviato un descrittore vuoto nella Receive Queue locale, pronto ad essere completato dal messaggio contenuto nel descrittore della Send Queue remota.

Questo modello di comunicazione richiede che avvenga la notifica ad entrambi i VI Consumer, per motivi di sincronizzazione. I VI Consumer sono responsabili del controllo di flusso durante una connessione: il VI Consumer al lato ricevente deve postare un descrittore di dimensione sufficiente a garantire la corretta ricezione del messaggio e tale descrittore deve ovviamente essere postato prima dell'arrivo del messaggio, evitando di far trovare la Receive Queue vuota, nel qual caso verrebbe generato un errore di tipo `VIP_DESCRIPTOR_ERROR`.

2.3.2 Modello RDMA

Nel modello RDMA, colui che ha il compito di iniziare il processo di trasferimento deve specificare sia l'indirizzo del buffer sorgente, sia quello del buffer di destinazione. Abbiamo accennato a due tipi di operazioni RDMA: Read e Write.

Nella RDMA Write, il VI Consumer specifica la sorgente dei dati quale indirizzo di memoria sul nodo locale e la destinazione sotto forma di indirizzo in cui scrivere i dati nella memoria registrata del nodo remoto. La regione contenente i dati da inviare può essere definita mediante una lista di buffer comunque organizzati, mentre la regione di memoria di destinazione deve necessariamente essere singola e contigua. L'operazione di RDMA Write implica che, prima del trasferimento dei dati, il VI Consumer al nodo remoto comunichi al lato di spedizione l'indirizzo del buffer su cui scrivere, assieme alla notifica del buffer stesso pronto a ricevere dati.

Analogamente, nelle operazioni di RDMA Read, il VI Consumer ha il compito di specificare la sorgente del trasferimento dei dati al lato remoto e la regione di destinazione sotto forma di indirizzo di memoria del nodo locale. Allo stesso modo, la sorgente dell'RDMA Read deve necessariamente essere una regione singola e contigua di memoria nel nodo remoto, mentre è priva di questa restrizione la regione di destinazione nel nodo locale. L'operazione di RDMA Read implica che, prima della lettura dei dati, il VI Consumer al nodo remoto informi il VI Consumer del nodo locale sugli indirizzi del buffer sorgente e che notifichi la sua disponibilità ad essere letto.

In questo tipo di comunicazioni, nessun descrittore sulla Receive Queue del nodo remoto viene consumato dalle operazioni di RDMA e nessuna notifica è consegnata al nodo remoto riguardante il completamento della richiesta. Presenta una eccezione la comunicazione eseguita da operazioni di RDMA Write in cui è specificata l'opzione *Immediate Data*: in tal caso viene consumato un descrittore al nodo remoto a trasferimento completato, in modo da garantire una sorta di sincronizzazione. A tale scopo, il VI Consumer al nodo remoto deve postare un descrittore sulla Receive Queue del proprio nodo in modo da permetterne il suo utilizzo al completarsi dell'operazione. In caso di mancanza di tale descrittore, viene generato un errore e la connessione può essere interrotta. L'opzione Immediate Data non è disponibile nelle operazioni di RDMA Read [3].

Essendo obbligatoria, nell'architettura di VIA, solamente l'operazione di RDMA Write, il VI Provider può fornire un meccanismo atto a verificare se la contro-parte supporta o meno operazioni di RDMA Read.

2.4 I livelli di affidabilità

L'architettura VI supporta tre livelli di affidabilità nella comunicazione: *Unreliable Delivery*, *Reliable Delivery*, *Reliable Reception*. Il primo di essi deve essere garantito da tutti i NIC che gestiscono le VI, mentre risultano facoltativi il secondo e il terzo. Il livello di affidabilità costituisce un attributo proprio di ogni VI e solamente le VI con medesimo livello di affidabilità possono essere connesse tra loro. Nella tabella 2.1 sono illustrate le proprietà di ogni livello di affidabilità.

2.4.1 Unreliable Delivery

Una VI con livello di affidabilità Unreliable garantisce che una operazione di Send o RDMA Write sia trasmessa al più una volta alla VI ricevente, assicurando il consumo di al più un descrittore.

Inoltre, la VI garantisce che i trasferimenti corrotti siano sempre rilevati dal lato ricevente: in caso una Send o una RDMA Write utilizzasse un descrittore sulla Receive Queue, ma i dati trasferiti risultassero corrotti, il bit di errore associato dovrà essere modificato nell'apposito campo del descrittore di ricezione.

In caso di non ricezione della Send o della RDMA write, il VI Provider non è tenuto né alla rilevazione di questa condizione, né alla ritrasmissione del dato perduto. Le richieste, inoltre, non sono necessariamente consegnate al ricevente nello stesso ordine in cui sono state spedite.

Le operazioni di RDMA Write dovrebbero essere utilizzate esclusivamente in questo livello di affidabilità, in condizioni in cui può essere accettato un ritardo nelle spedizioni o una perdita di dati.

In caso di errore di trasmissione, la connessione non viene interrotta e la VI non viene posta in stato di errore, ma viene semplicemente notificato l'errore al VI Consumer.

In questo livello di affidabilità non sono supportate le operazioni di RDMA Read, all'invio delle quali viene notificato un errore di formato del descrittore.

Il livello Unreliable è per sua concezione adatto alle comunicazioni a bassa latenza, quali appunto gli scambi tra nodi di cluster. Sarebbe per questo motivo inaccettabile la comunicazione tra questo livello ed il livello Reliable, in cui viene speso più tempo per il controllo della correttezza delle trasmissioni.

2.4.2 Reliable Delivery

Il livello Reliable garantisce che tutti i dati trasmessi arrivino a destinazione esattamente una volta, intatti e nell'ordine di spedizione, in assenza di errori. Gli errori di trasporto sono da considerarsi eccezionali per questo livello di servizio. Il VI Provider invia un errore al VI Consumer in caso di perdita di dati, corruzione del pacchetto o mancato rispetto dell'ordine. Analogo comportamento avviene in caso di perdita di RDMA Write o Send a causa dell'assenza di descrittori sulla Receive Queue o della dimensione insufficiente del descrittore in testa alla coda. In questo caso, la VI passa in stato di errore, la connessione è interrotta e tutti i descrittori postati vengono completati con uno stato di errore.

Un descrittore associato ad una Send o ad una RDMA Write viene completato in modo corretto una volta che i dati relativi sono stati inviati con successo sulla rete; un descrittore associato ad una RDMA Read viene invece completato correttamente quando i dati richiesti sono stati correttamente scritti nei buffer di destinazione.

Gli errori intervenuti durante l'inizializzazione del sistema, riguardanti il formato del descrittore o la protezione della memoria locale, causano il completamento del descrittore in stato di insuccesso, con la variazione di uno o più dei suoi bit di errore. Gli errori hardware, di trasporto, di ricezione, di perdita pacchetti, avvenuti cioè dopo che il descrittore è stato completato, sono notificati al VI Consumer in modo asincrono dal VI Provider.

Proprietà	***Unreliable Delivery***	***Reliable Delivery***	***Reliable Reception***
Rilevazione dati corrotti	Sì	Sì	Sì
Dati inviati al più una volta	Sì	Sì	Sì
Dati inviati esattamente una volta	No	Sì	Sì
Ordine dati garantito	No	Sì	Sì
Rilevazione perdita dati	No	Sì	Sì
Stop connessione in caso di errore	No	Sì	Sì
Supporto a RDMA Read	No	Optional	Optional
Supporto a RDMA Write	Sì	Sì	Sì

Tabella 2.1: Proprietà dei livelli di affidabilità delle VI [3].

2.4.3 Reliable Reception

Come si vede dalla tabella 2.1, le proprietà di questo livello coincidono con quelle del precedente, ma si possono riscontrare le seguenti differenze: in questo livello il descrittore è completato correttamente solo quando i dati sono stati copiati nelle locazioni di memoria di destinazione, in modo da utilizzare il descrittore stesso per notificare un eventuale errore di dati corrotti, sequenza errata, ripetuta o mancata spedizione. In questo modo, il VI Provider è in grado di assicurare, in caso di errore, che nessun altro successivo descrittore, oltre a quello che ha postato l'errore medesimo, sarà completato.

2.5 Le operazioni di VIA

Analizziamo brevemente le principali operazioni che un'applicazione può eseguire sulle VI tramite la libreria VIPL.

- **Set-Up e Tear-Down**: per inizializzare un nodo VI, l'applicazione chiama la funzione `VipCreateVI`, la quale passa le informazioni attraverso il driver. A sua volta, il driver inoltra le informazioni per la creazione al NIC. È qui che il sistema operativo può inibire all'applicazione l'accesso all'interfaccia di rete o ad un particolare nodo.

- **Registrazione e Deregistrazione della memoria**: come già abbiamo accennato, tutti i buffer dati e i descrittori devono necessariamente risiedere in una regione di memoria *registrata*. Queste regioni sono ottenute mediante la funzione `VipRegisterMemory`, la quale interagisce con il kernel del sistema operativo per effettuare la registrazione specifica. Questa registrazione mette a disposizione le pagine di memoria permettendo l'accesso diretto al NIC.

- **Connessione e disconnessione**: i servizi dell'architettura VI sono orientati alla connessione. Quando una VI viene creata, non risulta connessa a nessun'altra VI e per poter iniziare la comunicazione occorre instaurare questo collegamento tra le due VI (`VipConnectRequest`) e distruggerlo al termine delle operazioni di comunicazione. Queste chiamate passano attraverso il kernel del sistema operativo, il quale gestisce gli stati interni delle VI e garantisce la loro protezione. Il NIC viene infine notificato dell'avvenuta connessione.

- **Trasmissione dei dati**: sono utilizzate, per la trasmissione, tre strutture dati. Innanzitutto il buffer dati, che l'applicazione desidera trasmettere; esso è allocato dall'applicazione stessa in una regione di memoria registrata. Usualmente una zona di memoria registrata è utilizzata per la trasmissione di più messaggi. Per ogni messaggio, l'applicazione sposta i dati nel buffer e costruisce un descrittore appropriato. Questo contiene l'indirizzo e la lunghezza del dato, oltre alle informazioni di stato e di protezione.

  L'applicazione, attraverso la VIPL, posta una doorbell ad indicare che il descrittore è pronto alla trasmissione, scrivendo l'indirizzo del descrittore medesimo in un registro del NIC. Questi mantiene una coda delle doorbell notificategli; quando una di esse raggiunge l'inizio della coda, il NIC inizia la trasmissione del dato. Alla fine, il NIC cambia lo stato del descrittore in questione, stato utilizzato dall'applicazione per determinare l'avvenuto completamento della trasmissione o un eventuale errore.

- **Ricezione dei dati**: per ricevere un messaggio, l'appicazione alloca un buffer dati vuoto nella memoria registrata e ne costruisce il relativo descrittore. Posta quindi una doorbell al proprio NIC per indicare la disponibilità dello spazio per l'arrivo del pacchetto. Entrambe queste operazioni devono essere completate prima dell'arrivo del pacchetto. In caso il NIC non abbia ricevuto la doorbell al momento dell'arrivo del pacchetto, quest'ultimo andrà irrimediabilmente perduto.

  All'arrivo di un pacchetto, il NIC scrive i dati nel buffer precedentemente allocato, quindi modifica i bit di stato del descrittore a notificare l'avvenuta ricezione di nuovi dati nel buffer medesimo.

Tutte le operazioni di Send e Receive in VIA sono non bloccanti: il flusso ritorna al chiamante non appena la doorbell è stata postata. Questo permette la gestione simultanea di più processi di comunicazione rispetto ad una semantica di tipo bloccante, mentre all'occorrenza ogni richiesta di comunicazione può essere gestita da sistemi bloccanti o a polling sul completamento dei descrittori.

Il meccanismo di doorbell in VIA è strettamente dipendente dalla particolare realizzazione dell'intera architettura: alcune versioni possono prevedere una gestione delle doorbell interamente a cura dell'hardware, mentre le release software, ad esempio quelle basate su hardware Ethernet, fruiscono di chiamate di sistema altamente ottimizzate ed interamente realizzate a livello software.

L'architettura VIA si basa su presupposti di completa affidabilità della rete hardware sottostante, la quale può essere scelta con ampi livelli di arbitrarietà: il modello a descrittori risulta di fatto compatibile con molte delle attuali realizzazioni hardware più comuni, anche se un ottimo beneficio prestazionale viene raggiunto in caso di realizzazione hardware di alcuni meccanismi, primo fra tutti appunto quello delle doorbell.

A conclusione del presente capitolo, ad esclusivo titolo documentativo, è presentato uno schema della struttura dei due dipi di descrittore presenti nelle VI: quello relativo alle operazioni di Send/Receive e quello relativo ad operazioni RDMA.

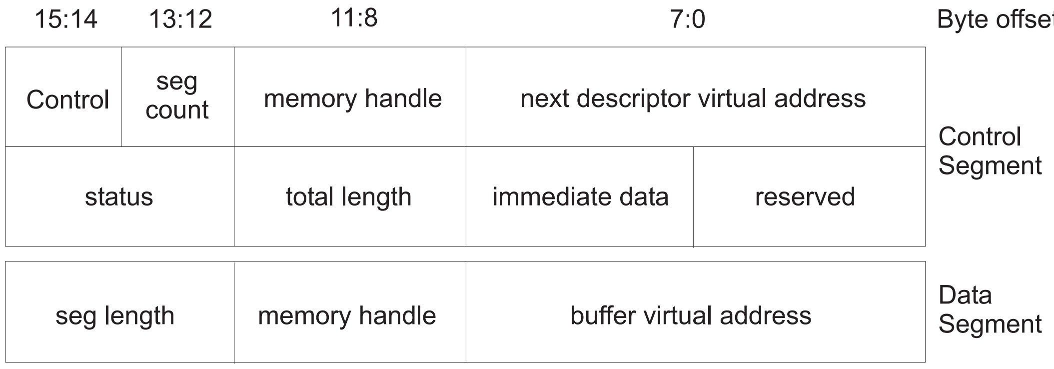

Figura 2.4: Struttura del Descrittore per operazioni di Send/Receive.

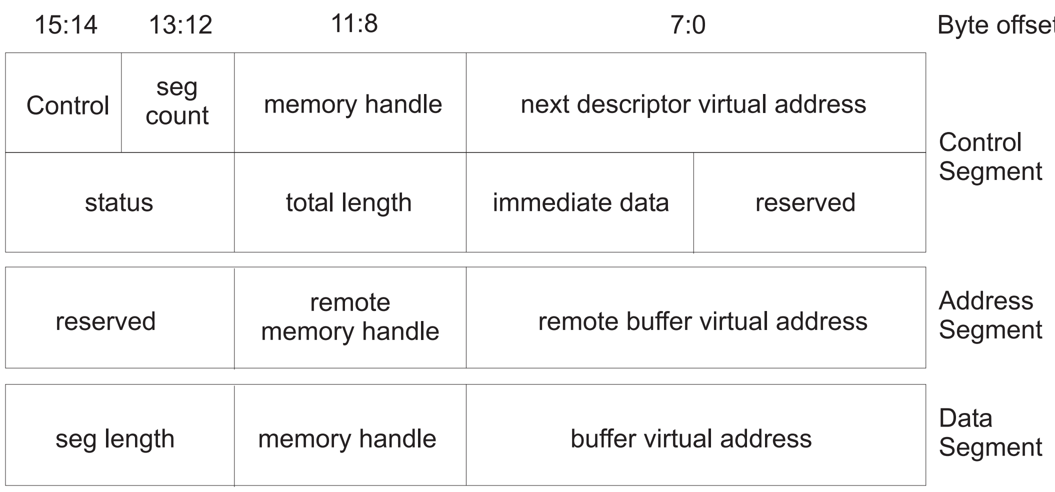

Figura 2.5: Struttura del Descrittore per operazioni di RDMA.

Seguono quattro schemi, in ognuno dei quali sono evidenziati i campi cui accedono rispettivamente le applicazioni che gestiscono le VI, le funzioni della libreria VIPL, i VI NIC durante il completamento e le funzioni VIPL al completamento.

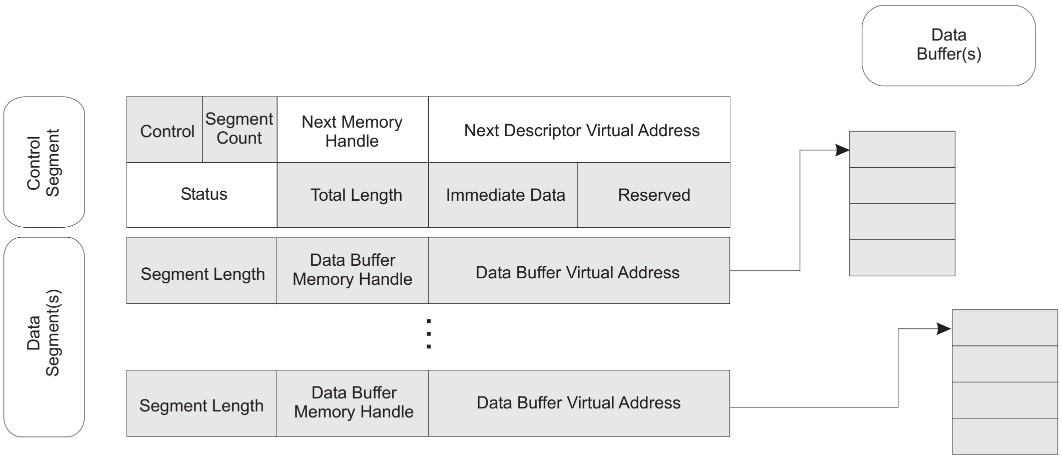

Figura 2.6: Campi del descrittore aggiornati dall'applicazione.

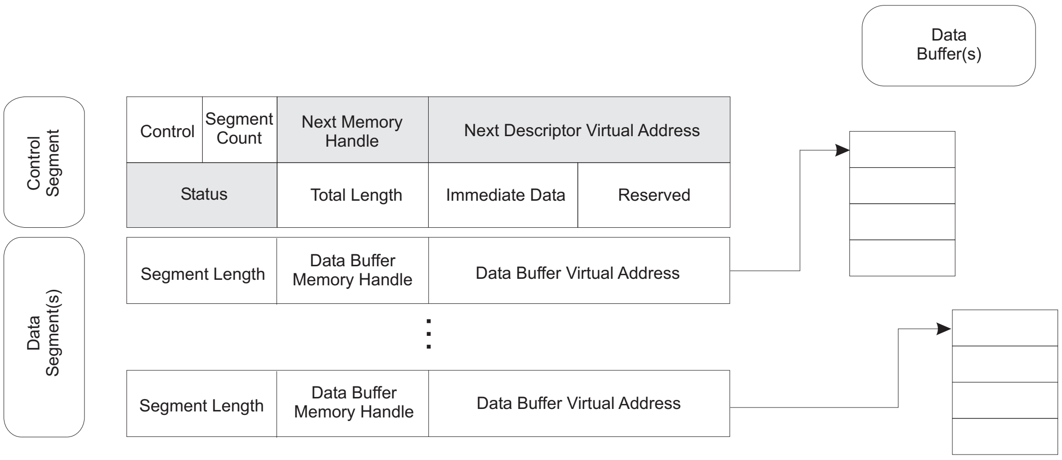

Figura 2.7: Campi del descrittore aggiornati dalle funzioni della libreria VIPL.

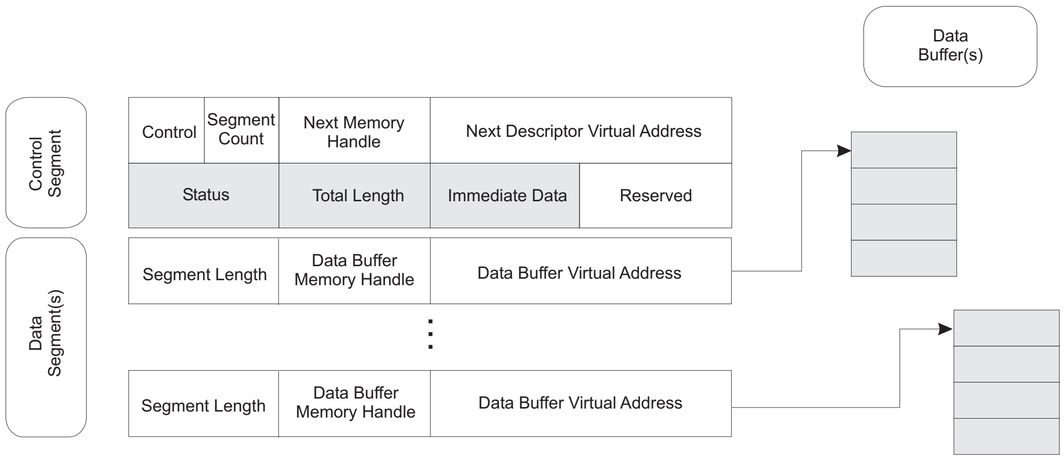

Figura 2.8: Campi del descrittore aggiornati dal VI NIC durante il completamento.

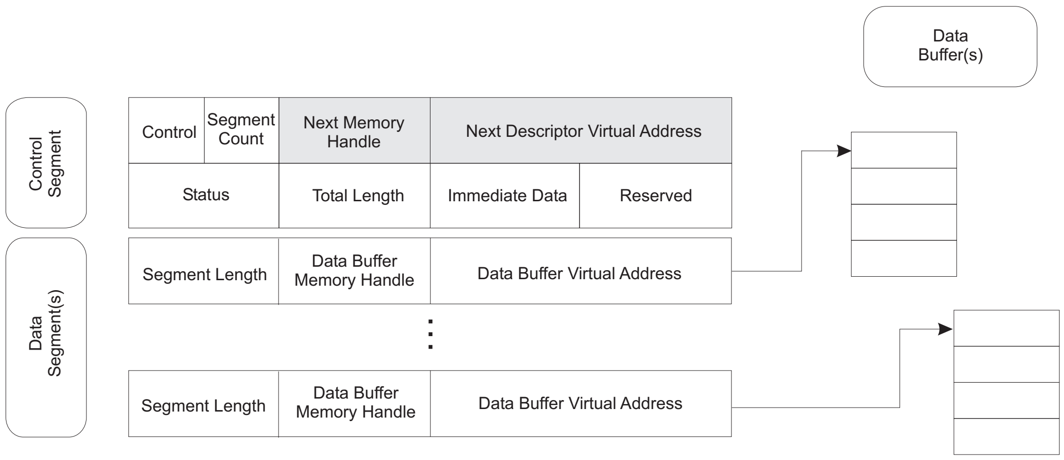

Figura 2.9: Campi del descrittore aggiornati dalle funzioni VIPL durante il completamento.

CAPITOLO 3

Realizzazioni di VIA

Sono analizzate, in questo capitolo, due delle più diffuse realizzazioni dell'architettura VIA per personal computer "x86": una realizzazione hardware fornita da GigaNet Inc., recentemente acquisita da Emulex, consistente in una scheda PCI con interfaccia a 64 bit, denominata cLan 1000, con implementazione hardware dell'architettura VI [6] ed una realizzazione software, denominata M-VIA (Modular VIA), per il sistema operativo Linux fornita dal National Energy Research Scientific Computing Center (NERSC) presso il Lawrence Berkley National Laboratory [10].

Durante l'analisi delle caratteristiche di queste due soluzioni, ne saranno presentati i vantaggi e le limitazioni, oltre ai risultati di alcuni test prestazionali ed analizzate le scelte che sono state compiute.

3.1 Modular VIA (M-VIA)

Una nota versione software dell'architettura VIA è realizzata come modulo per il sistema operativo Linux. Un modulo è una parte del kernel del sistema operativo che viene caricata all'occorrenza, senza dover risiedere permanentemente nella memoria di sistema. Il modulo, inoltre, viene autonomamente scaricato quando non più utilizzato.

M-VIA è in licenza "open source": la distribuzione del pacchetto è unita a quella dei sorgenti, i quali possono essere liberamente analizzati, modificati e redistribuiti dallo sviluppatore, anche per scopi commerciali.

Le interfacce hardware fisicamente supportate da M-VIA sono in crescita, grazie anche alla piena adozione della libreria VIPL, la quale permette l'accesso all'architettura da parte di VI NIC di più produttori contemporaneamente. Inoltre, l'architettura software è tale da permettere la coesistenza di protocolli di comunicazione tradizionali; in un nodo hardware possono avere accesso al NIC fisico sia le funzioni VIA sia le tradizionali TCP/IP, entrambe facenti capo ad una stessa scheda di rete.

Le operazioni realizzate nell'architettura sono “thread safe”, in accordo con le librerie thread native del sistema Linux. Inoltre è possibile, da parte dei produttori di interfacce hardware, la creazione di moduli binari per i loro prodotti senza richiedere all'utente finale la loro compilazione in fase di installazione delle schede stesse.

Attualmente la release a disposizione, su cui si sono basate le applicazioni qui presentate, è la 1.2b1. Sono supportate in modo nativo le interfacce di rete presenti nel cluster ParMa[2], vale a dire:

- Intel Pro/100 Fast Ethernet per il nodo ParMa01;
- D-Link con chipset Tulip DEC DC21x4x per i nodi pp1, pp2, pp3, pp4.

Il kernel richiesto dalla attuale versione di M-VIA è il 2.2.1 o successivi, con supporto al Symmetric Multi Processing.

Rispettando pienamente le specifiche di VIA, M-VIA è composto da tre componenti principali: la VI Provider Library, il VI Kernel Agent, ed i VI NIC. M-VIA identifica il Kernel Agent di VIA nell'insieme costituito dal Kernel Agent di M-VIA e da uno o più M-VIA Device Drivers (figura 3.1).

3.1.1 M-VIA Kernel Agent

Il Kernel Agent M-VIA provvede all'esecuzione delle operazioni privilegiate richieste dalla VI Provider Library ed assiste gli M-VIA Device Drivers nelle operazioni

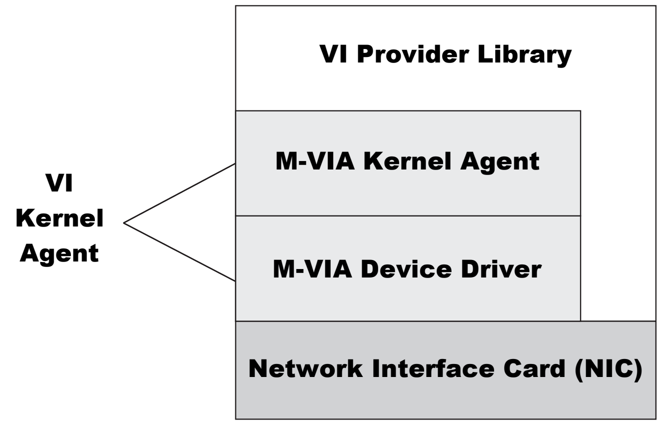

Figura 3.1: Schema d'insieme dell'architettura di M-VIA.

che richiedono l'intervento del sistema operativo. Esso è suddiviso nelle seguenti componenti:

- **Connection Manager**: stabilisce le connessioni logiche punto a punto tra le diverse VI;
- **Protection Tag Manager**: alloca, dealloca e convalida i Ptags (Tag di protezione della memoria);
- **Registered Memory Manager**: gestisce i processi di registrazione dei buffer di comunicazione e dei buffer dei descrittori;
- **Error Queue Manager**: fornisce un meccanismo per la notifica degli errori di sincronizzazione tra i devices.
- **Kernel Extensions**:forniscono tutte le necessarie funzionalità per l'efficienza della realizzazione.

3.1.2 M-VIA Device Drivers

Un M-VIA Device Driver è un'astrazione del VI NIC e si registra come tale nell'M-VIA Kernel Agent. Il device informa il Kernel Agent delle sue caratteristiche, quali la presenza di un eventuale supporto hardware di VIA, la sua dimensione MTU nativa [10], il massimo numero di descrittori che possono essere accodati per la trasmissione, ecc.

Il device, inoltre, registra le specifiche funzionalità che saranno utilizzate dal livello kernel. Lo sviluppatore del Device Driver ha infatti la possibilità di sostituire una, più di una o tutte le funzionalità standard fornite dall'M-VIA Kernel Agent. Ad esempio, se un device che fornisce un'architettura hardware di VIA, possiede una propria struttura che si occupa della registrazione della memoria, è possibile rimpiazzare completamente il Registered Memory Manager di M-VIA con la propria realizzazione hardware.

La possibilità di sostituire qualsiasi funzionalità fornita dal Kernel Agent di M-VIA permette a qualsiasi emulazione hardware o software di VIA di essere integrata e pienamente supportata da M-VIA stesso (a patto, ovviamente, che rispetti le specifiche dettate dallo standard VIA).

Molte versioni hardware di interfacce di rete possono essere raggrupate in famiglie quali Ethernet, FDDI, ATM, ecc. Per promuovere una facilità di interconnessione tra le diverse famiglie, M-VIA introduce le cosiddette "Device Classes", ognuna delle quali contiene routine di programmazione specifiche per ogni categoria di interfacce di rete. Queste routine contengono operazioni quali la creazione degli header e dei meccanismi che permettono la coesistenza di VIA con altri protocolli (ad es. TCP/IP). Queste Device Classes non sono esplicitamente supportate dall'M-VIA Device Driver, ma quest'ultimo è realizzato in modo da permetterne il pieno supporto.

3.1.3 VI Provider Library

M-VIA contiene una singola libreria VIPL, conforme alle specifiche VIA, che interopera con i vari device VIA hardware e software sviluppati all'interno della rete in oggetto.

3.1.4 Conformità a VIA

La versione attuale della distribuzione di M-VIA fornisce le funzioni di VIA a livello di Unreliable Delivery. Conformemente alle specifiche di VIA, in questo livello di affidabilità non è previsto il supporto alle operazioni di RDMA Read.

L'attributo relativo alla qualità del servizio è per ora ignorato ed occorre impostarlo a zero per evitare la generazione di errori.

M-VIA è compatibile con lo standard di programmazione "POSIX threads", con pieno supporto alle applicazioni multi-thread. Le operazioni eseguite da un'applicazione multi-thread su VI differenti sono thread-safe, ma rimane compito dell'applicazione stessa provvedere alla creazione e alla gestione dei sistemi di lock e di semafori in caso di potenziali accessi multipli alla singola VI o Completion Queue.

3.2 cLAN C1000

L'host adapter cLAN C1000 è un'interfaccia di rete PCI con realizzazione hardware di VIA. Tra le sue caratteristiche citiamo:

- Supporto a più sistemi operativi, tra cui Linux (RedHat 6.2 e 7) su piattaforma Intel, Windows NT4 e Windows 2000 (Service Pack 2) su piattaforma Intel;
- Interfaccia bus PCI a 32 e 64 bit, 33 MHz, conforme alle specifiche PCI 2.1;
- Velocità di collegamento a 1.25 Gb/secondo per ogni direzione (burst bidirezionale di 2.50 Gb/s);
- Realizzazione delle funzionalità di VIA su singolo chip;
- Architettura a ridotto carico di lavoro per la CPU di sistema;
- Latenza osservabile ridotta a 3,5 microsecondi;
- Supporto di architettura a cluster fino a 128 nodi a piena banda o 64 nodi in configurazione ridondante.

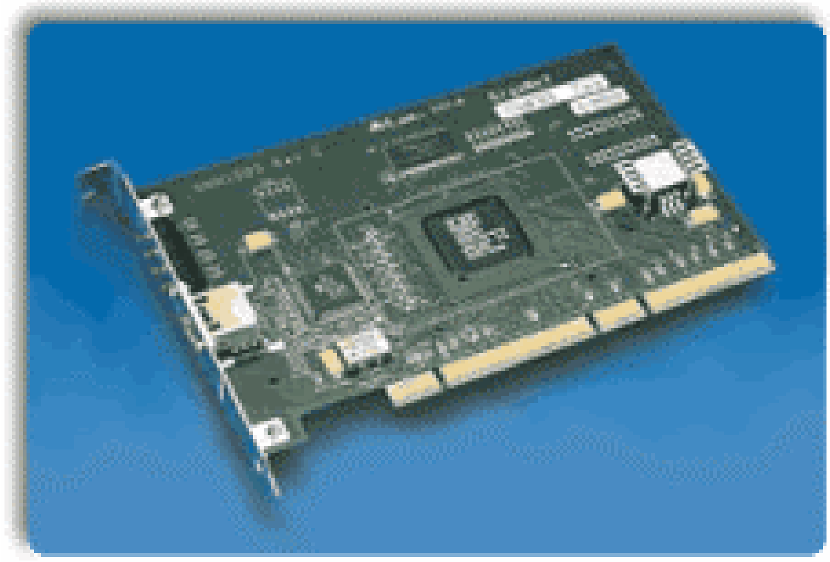

Figura 3.2: La scheda di rete Emulex cLAN C1000.

- Supporto fino a 1024 circuiti di connessione virtuali (1024 VI e 1024 Completion Queues);
- Massima memoria registrata supportabile: 1 Gb;
- Supporto contemporaneo ai seguenti protocolli (con disponibilità di drivers e software di gestione):
  - Virtual Interface Architecture v. 1.0;
  - SNMP Traps;
  - TCP/IP;
  - NDIS;
  - WinSock Direct.

3.2.1 Conformità a VIA

Per quanto riguarda la conformità alle specifiche VIA, segnaliamo il supporto dei livelli di affidabilità Unreliable e Reliable delivery, mentre non è supportato il livello Reliable Reception. Il supporto alle operazioni RDMA è garantito solo sulle

Classe di funzioni	***API***
Querying Information	`VipQuerySystemManagementInfo`
Trasferimento dati e completamento	`VipSendNotify` `VipRecvNotify` `VipCQNotify`
Protezione e registrazione della memoria	`VipCreatePtag` `VipDestroyPtag`

Tabella 3.1: API non realizzate nell'hardware Emulex cLAN C1000 [6].

operazioni di RDMA Write, in quanto la disponibilità dell'RDMA Read avrebbe solo influenzato la gestione dell'ordine delle operazioni di RDMA Read rispetto a RDMA Write. Nella tabella 3.1 sono elencate le API[1] della libreria VIPL non ancora realizzate.

3.3 Analisi delle prestazioni

Il pacchetto M-VIA è corredato da una serie di programmi che permettono la verifica della correttezza dell'installazione e il test delle prestazioni di alcune funzioni. Analizzando i sorgenti dei programmi di test, si sono estrapolati i nuclei del codice che generano lo scambio effettivo dei dati, su cui sono stati misurati i tempi di latenza e di comunicazione.

Come da specifiche di VIA, si è potuto constatare che il programma *vpingpong*, con il compito di scambiare una quantità configurabile di byte tra due nodi, compie i seguenti passi:

- `VipOpenNic`: associa un processo ad un VI NIC e fornisce un NIC Handle al VI Consumer. Il NIC Handle è utilizzato nelle successive funzioni come identificativo di quel particolare NIC.

- `VipNSInit`: inizializza il name service utilizzando il parametro specificato, per l'identificazione sulla rete del nodo locale.

[1] Application Program Interface

- `VipQueryNic`: ritorna informazioni e parametri riguardanti un particolare NIC.

- `VipNSGetHostByName`: mappa i nomi dei nodi passatigli sotto forma di stringa convertendoli in indirizzi di rete, compatibili con le funzioni di connessione successive.

- `VipCreatePtag`: la funzione crea un tag di protezione per il processo chiamante. Il tag è successivamente associato alle VI una volta create.

- `VipRegisterMem`: permette ad un processo di registrare una regione di memoria con un VI NIC. La memoria utilizzata per contenere i descrittori deve essere registrata con tale funzione.

- `VipCreateVi`: effettua la creazione della VI, associandola al NIC specificato. Una volta creata, la VI si pone in stato di *idle*.

- `VipConnectRequest`: richiede che sia stabilita una connessione tra la corrente VI ed un'altra sul nodo remoto.

- `VipConnectWait`: è effettuato dal lato remoto, che lo pone in attesa di una richiesta di connessione da uno dei nodi della rete.

- `VipConnectAccept`: una volta arrivata la richiesta di connessione, il nodo esce dal Wait ed associa la connessione alla propria VI locale.

A questo punto le VI sono create, la memoria è protetta e registrata, la conessione è stata effettuata e le due VI sono in corrispondenza biunivoca. Si passa ad effettuare il vero e proprio scambio di dati, tramite quattro fondamentali funzioni:

- `VipPostSend`: la funzione aggiunge un descrittore alla coda della Send Queue della VI, notifica al NIC che un nuovo descrittore è disponibile e ritorna immediatamente.

- `VipSendWait`: controlla il descrittore in testa alla Send Queue e ne verifica lo stato di completamento. Appena il descrittore risulta completo (il messaggio, cioè, è stato spedito) esso viene rimosso dalla coda e viene ritornato il suo indirizzo. Se il descrittore in testa alla Send Queue non è completato, la funzione si blocca finché non si realizza il completamento o non scade il timeout specificato.

- `VipPostRecv`: aggiunge un descrittore alla Receive Queue della VI specificata e notifica al NIC la presenza di tale descrittore libero, ritornando immediatamente.

- `VipRecvWait`: analogamente alla duale funzione di spedizione, essa controlla il descrittore in testa alla Receive Queue, verificandone il completamento. Alla ricezione del messaggio il descrittore si completa e viene rimosso dalla Receive Queue. In caso di completamento non ancora avvenuto, la funzione non ritorna fino al verificarsi del completamento o all'esaurimento del timeout specificato.

È possibile, anche se non ottimale per la condizione di attesa attiva che si verrebbe a creare, una realizzazione dell'algoritmo ping pong con *polling*, mediante il quale viene periodicamente verificato il completamento del descrittore, interrompendo con un ciclo attivo il flusso del programma, fino all'effettivo ricevimento del messaggio atteso. Questo tipo di verifica viene effettuato con le seguenti due funzioni:

- `VipSendDone` e `VipRecvDone`: controllano lo stato del descrittore in testa alla Send Queue o alla Receive Queue rispettivamente. Se il descrittore risulta completato, esso viene eliminato dalla coda e ne viene ritornato l'indirizzo, ma se manca lo stato di completamento le funzioni ritornano comunque puntando l'indirizzo del descrittore a *NULL*.

In figura 3.3 sono riassunti i diversi stati in cui può trovarsi una VI, mentre in

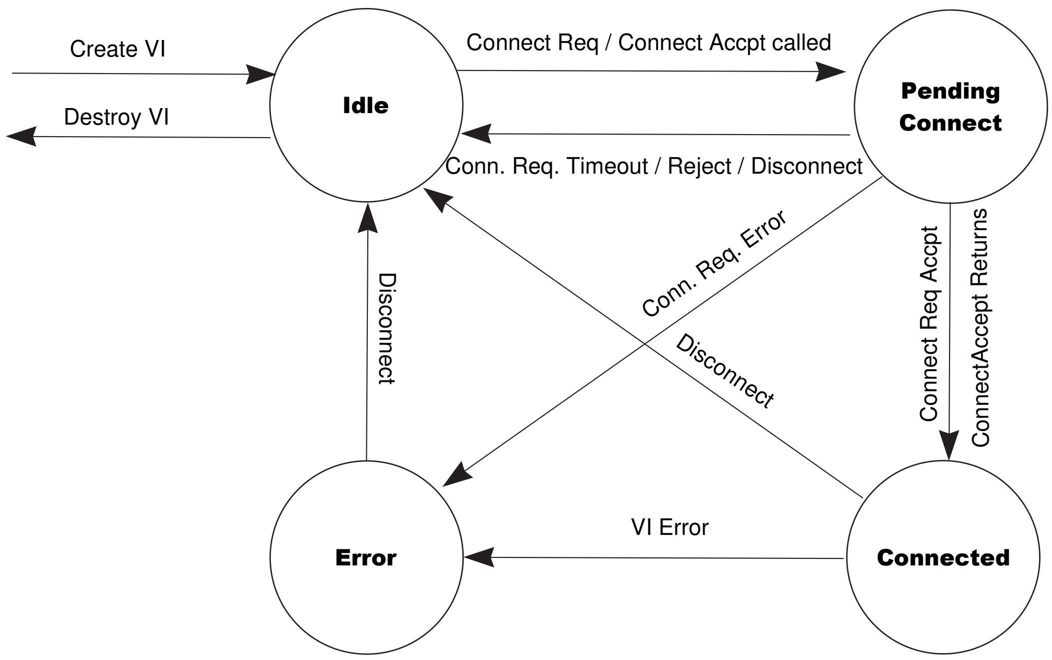

Figura 3.3: Diagramma di stato dei possibili stati di una VI (modello Client-Server).

figura 3.4 è schematizzato il susseguirsi delle fasi di connessione e comunicazione di due VI in comunicazione.

3.3.1 Primitive di inizializzazione e chiusura

Nelle tabelle 3.2 e 3.3 sono elencati i tempi medi di esecuzione delle primitive di comunicazione, utilizzando l'emulazione software M-VIA, mentre nelle 3.4 e 3.5 gli stessi tempi sono misurati sulla realizzazione hardware di Emulex. Seguono quindi i grafici che mettono a confronto le due architetture: in figura 3.5 sono indicate le funzioni del lato spedizione di durata inferiore ai 100 μs, mentre in figura 3.6 sono raccolte, per ragioni di scala, quelle di durata superiore. Analogamente, le figure 3.7 e 3.8 presentano i confronti dei tempi del lato ricevente. I valori indicati sono ottenuti da una media aritmetica di cinque prove, ognuna delle quali raccoglie i risultati temporali di 100000 scambi di pacchetti, anch'essi mediati aritmeticamente. La risoluzione delle misure è pari ad 1 μs ed il tempo richiesto dalla istruzione `GetSeconds` per rilevare il tempo di sistema è di 1.5 μs.

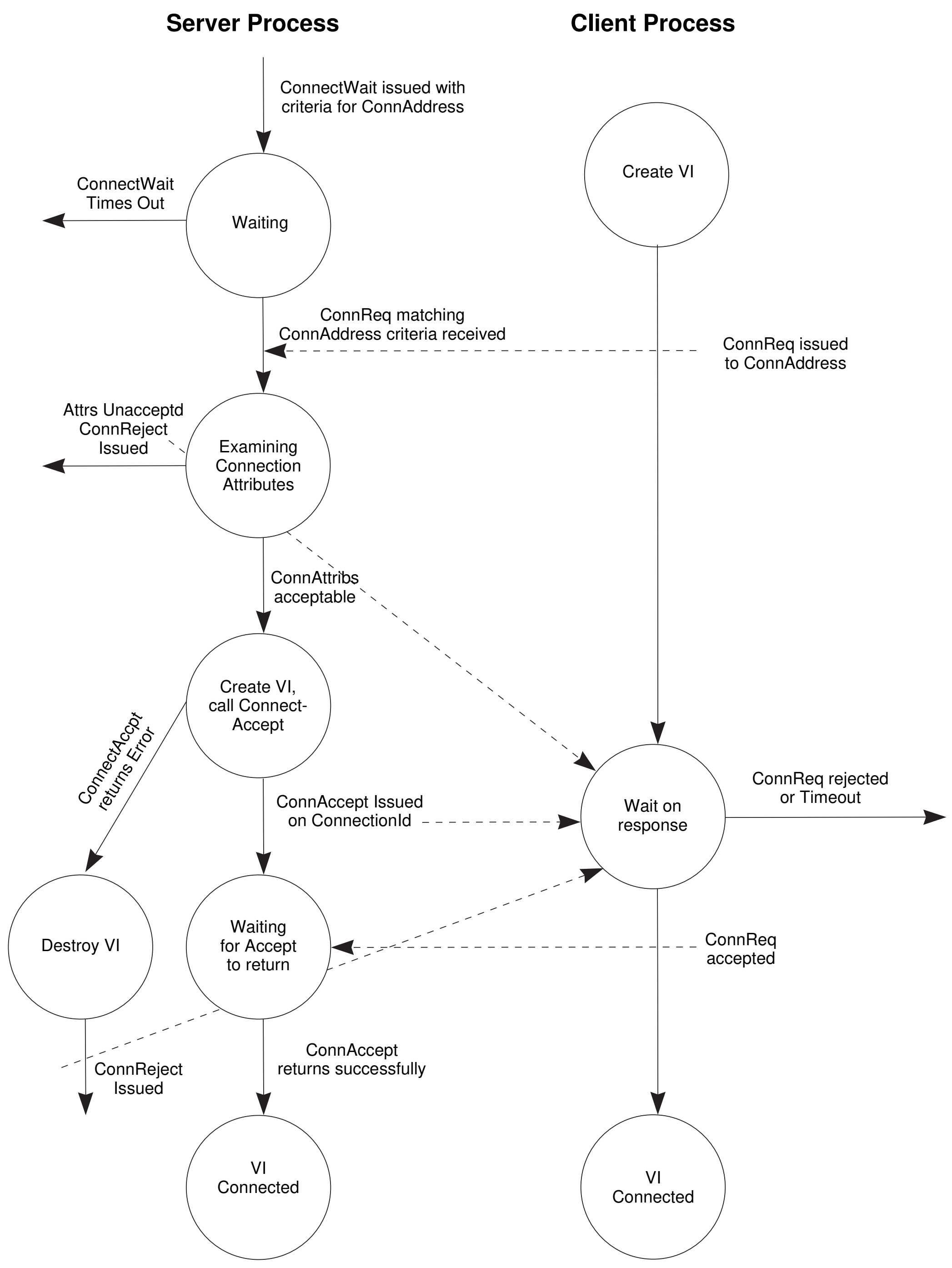

Figura 3.4: Diagramma di flusso della procedura di connessione di due VI.

Funzione	***Tempo*** (μs)
`VipOpenNic`	462
`VipNSInit`	474
`VipQueryNic`	9
`VipNSGetHostByName`	9
`VipCreatePtag`	7
`VipRegisterMem`	31
`VipCreateVi`	32
`VipConnectRequest`	540
`VipDeregisterMem`	16
`VipDisconnect`	13
`VipDestroyVi`	28

Tabella 3.2: Durata delle operazioni di inizializzazione e termine con M-VIA (lato Sender).

È da sottolineare che, nel caso della funzione `VipCreatePtag`, il tempo misurato sulla piattaforma hardware Emulex è da considerarsi privo di significato in quanto, non essendo tale funzione fisicamente realizzata, la sua esecuzione viene rimpiazzata dalla semplice impostazione del Protection Tag a *NULL* e dal ritorno della costante *VIP_SUCCESS*.

Si può notare quanto più efficienti siano le operazioni di apertura del NIC e di creazione della VI nella versione hardware, oltre all'inizializzazione del Name Service.

3.3.2 Primitive di comunicazione

I successivi dati raccolgono i risultati ottenuti misurando i tempi di esecuzione della trasmissione dei pacchetti, in modo da ottenere una ragionevole stima della latenza delle due realizzazioni. Come descritto precedentemente, in tabella 3.6 sono messi a confronto i risultati ottenuti con operazioni di ping pong utilizzando la tecnica di polling: il VI Provider continua ad eseguire operazioni di controllo sulla rispettiva coda in attesa che il descrittore in testa risulti completato (`VipRecvDone`). In caso non sia ancora avvenuto il completamento, un semplice ciclo continua ad effettuare l'operazione fino al successo. In tabella 3.7 sono raccolti i risultati di ping pong con

Funzione	***Tempo*** (μs)
`VipOpenNic`	445
`VipNSInit`	443
`VipQueryNic`	8
`VipNSGetHostByName`	9
`VipCreatePtag`	5
`VipRegisterMem`	29
`VipCreateVi`	24
`VipPostRecv`	7
`VipConnectWait`	$>$ 1 sec.
`VipConnectAccept`	532
`VipDeregisterMem`	10
`VipDisconnect`	12
`VipDestroyVi`	13

Tabella 3.3: Durata delle operazioni di inizializzazione e termine con M-VIA (lato Receiver).

Funzione	***Tempo*** (μs)
`VipOpenNic`	238
`VipNSInit`	10
`VipQueryNic`	21
`VipNSGetHostByName`	77.649
`VipCreatePtag`	10
`VipRegisterMem`	25
`VipCreateVi`	8
`VipConnectRequest`	200.037
`VipDeregisterMem`	21
`VipDisconnect`	26.198
`VipDestroyVi`	37

Tabella 3.4: Durata delle operazioni di inizializzazione e termine con Emulex cLAN (lato Sender).

Funzione	***Tempo*** (μs)
`VipOpenNic`	220
`VipNSInit`	9
`VipQueryNic`	19
`VipNSGetHostByName`	91.549
`VipCreatePtag`	9
`VipRegisterMem`	18
`VipCreateVi`	7
`VipPostRecv`	16
`VipConnectWait`	$>$ 5 sec.
`VipConnectAccept`	149.988
`VipDeregisterMem`	16
`VipDisconnect`	40.288
`VipDestroyVi`	9

Tabella 3.5: Durata delle operazioni di inizializzazione e termine con Emulex cLAN (lato Receiver).

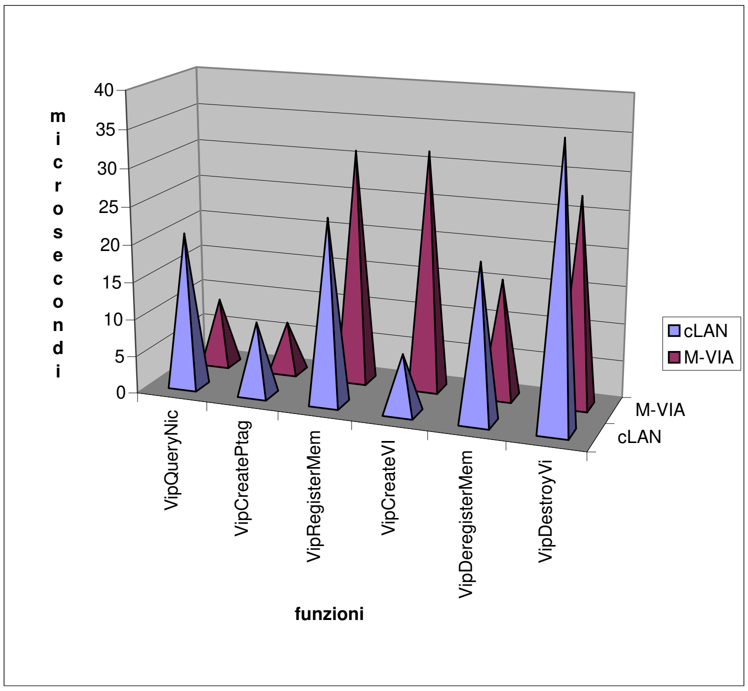

Figura 3.5: Confronto delle funzioni di durata inferiore ai 100 μs (Lato Sender).

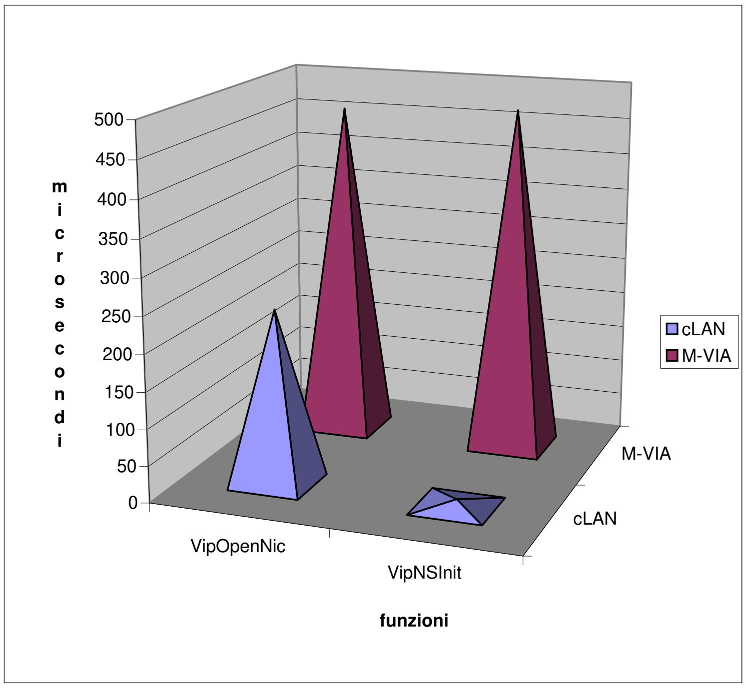

Figura 3.6: Confronto delle funzioni di durata superiore ai 100 μs (Lato Sender).

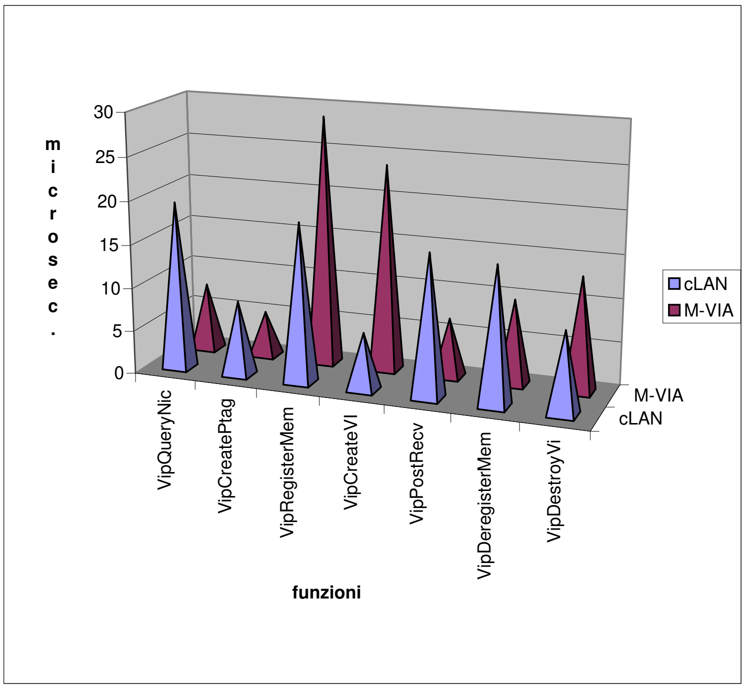

Figura 3.7: Confronto delle funzioni di durata inferiore ai 100 μs (Lato Receiver).

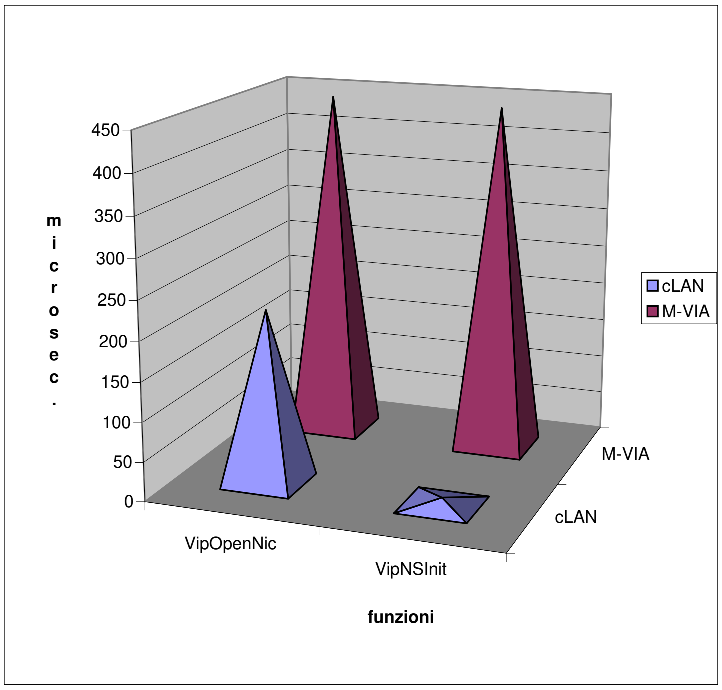

Figura 3.8: Confronto delle funzioni di durata superiore ai 100 μs (Lato Receiver).

wait bloccante: è la funzione `VipRecvWait` che, in caso di non completamento del primo descrittore della coda, si mette in attesa senza necessità di imporre cicli nel programma principale. È in quest'ultimo modo che dovrebbe funzionare la logica del ping pong: un nodo lancia il messaggio e si mette in attesa del suo ritorno dal nodo remoto.

Il produttore della piattaforma hardware cLAN, Emulex, fornisce con tale scheda un programma di test e benchmark chiamato TVIA, con il quale si vuole verificare la veridicità delle caratteristiche specificate sulla carta di tali schede, in particolare la latenza tipica di $3,5\mu s$. L'esecuzione di tale test rivela, di fatto, una latenza appartenente allo stesso ordine di grandezza delle specifiche fornite, e di fatto coincidente con i risultati presentati nella tabella 3.6, all'ultima colonna. Analizzando, però, il sorgente del programma TVIA, si vede chiaramente che tale risultato è ottenuto con il metodo *polling*, del tutto inadatto ad un utilizzo in programmazione di cluster. L'algoritmo 1 è utilizzato per l'attesa del messaggio dal ping pong di TVIA.

Se, viceversa, si sostituisce la precedente istruzione con una semplice primitiva

Dimensione del messaggio ***bytes***	***Tempo di M-VIA*** μs	***Tempo di cLAN*** μs
1	57	8
2	57	8
4	57	8
8	57	8
16	57	8
32	59	10
64	65	10
128	77	10
256	98	12
512	140	15
1024	224	20
2048	349	31
4096	518	51
8192	860	92
16384	1433	175
32768	2890	341

Tabella 3.6: Durata delle operazioni di comunicazione con metodo *polling*.

Dimensione del messaggio ***bytes***	***Tempo di M-VIA*** μs	***Tempo di cLAN*** μs
1	57	44
2	57	44
4	57	44
8	57	44
16	57	44
32	58	45
64	65	45
128	77	46
256	99	48
512	140	51
1024	224	57
2048	349	69
4096	518	89
8192	860	131
16384	1433	213
32768	2898	379

Tabella 3.7: Durata delle operazioni di comunicazione con metodo *wait* bloccante.

Algoritmo 1 Utilizzo del sistema di ricezione a *polling*.

```
for (;;) {
  status = VipRecvDone(vi, &desc);
  if (status == VIP_SUCCESS)
    break;
}
```

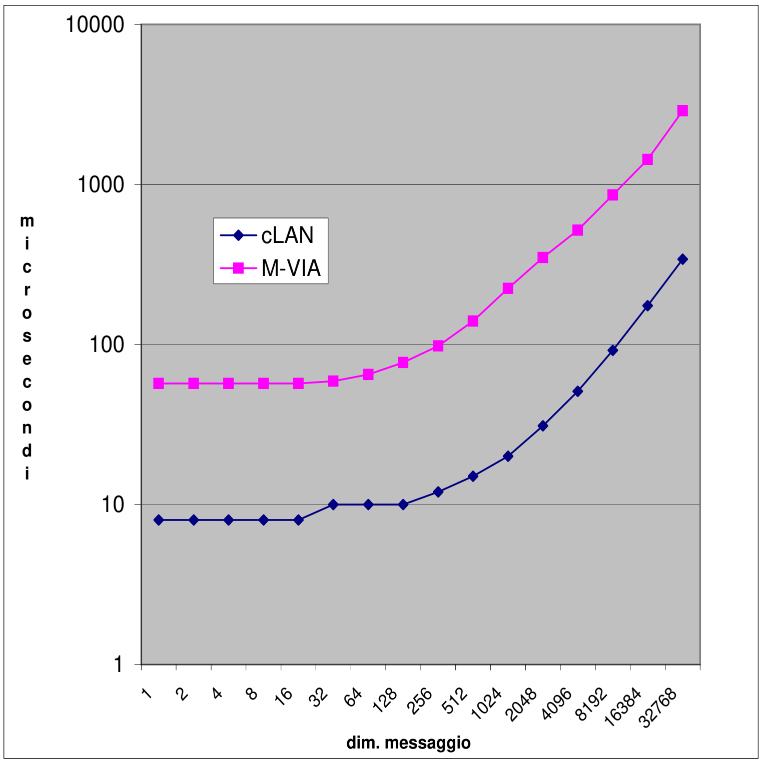

Figura 3.9: Confronto tra M-VIA e cLAN con ping-pong in modalità *poll*.

`VipRecvWait`, si ottengono tempi assolutamente paragonabili a quelli ottenuti con M-VIA, nonostante la mancanza della realizzazione hardware (tabella 3.7).

Le considerazioni suesposte vanno lette tenendo conto che la latenza di comunicazione si riferisce al tempo di trasmissione di messaggi di lunghezza tendente a zero. Quando si parla di latenza, cccorre quindi tenere in considerazione i tempi misurati con pacchetti di 1, 2, 4 e 8 byte. Le immagini 3.9 e 3.10 illustrano l'andamento dei tempi in funzione della dimensione dei pacchetti nel caso di polling e di wait, rispettivamente.

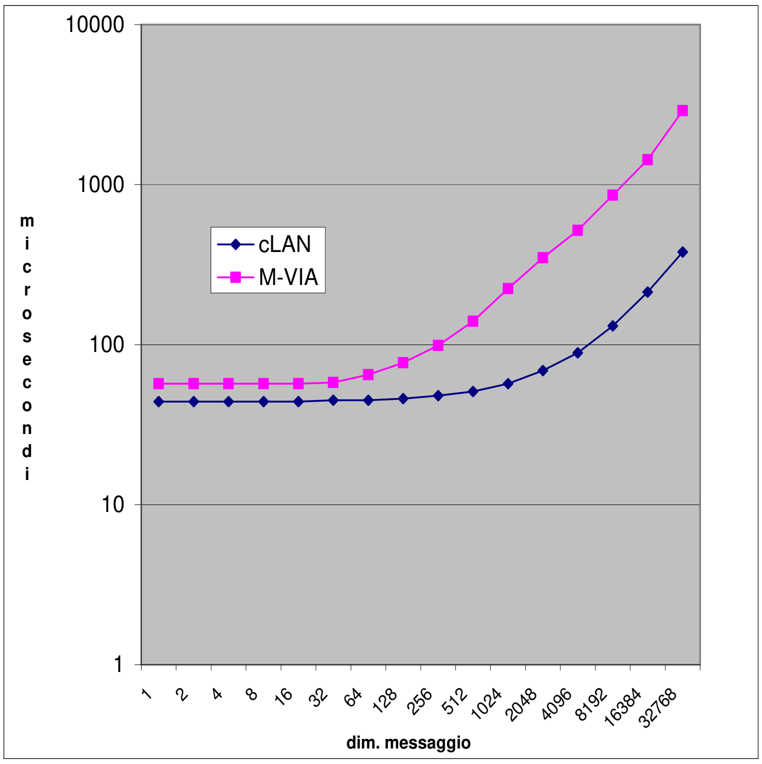

Figura 3.10: Confronto tra M-VIA e cLAN con ping-pong in modalità *wait* bloccante.

3.4 La primitiva VipRecvNotify

Accanto alle tradizionali operazioni di ricezione sopra analizzate, `VipRecvDone` e `VipRecvWait`, esiste un'altra famiglia formata dalle seguenti tre API:

- `VipRecvNotify`
- `VipSendNotify`
- `VipCQNotify`

Analizziamo ora il funzionamento della prima di esse e ne valutiamo le prestazioni, poiché è di essa che ci serviremo per il porting di LAM/MPI su M-VIA.

Occorre innanzitutto chiarire che, come già si è evidenziato nella tabella 3.1, tali primitive non sono state realizzate nella versione hardware di Emulex. Perciò, da ora in poi ci occuperemo solo dell'architettura VI M-VIA, in cui, a partire dalla versione 1.0, rilasciata il 25 settembre 1999, sono state rese disponibili le funzioni per la notifica asincrona della ricezione dei messaggi.

Questa funzione è utilizzata in sostituzione della primitiva `VipRecvDone` o della `VipRecvWait`, dal lato ricevente, con questo principio di funzionamento: la thread principale chiama la funzione `VipRecvNotify`, con la sintassi come da algoritmo 2.

Algoritmo 2 Sintassi di `VipRecvNotify`.

```
VIP_RETURN
  VipRecvNotify(
    IN  VIP_VI_HANDLE  ViHandle,
    IN  VIP_PVOID      Context,
    IN  void(*Handler) (
          IN  VIP_PVOID       Context,
          IN  VIP_NIC_HANDLE  NicHandle,
          IN  VIP_VI_HANDLE   ViHandle,
          IN  VIP_DESCRIPTOR  *DescriptorPtr
    )
  )
```

Alla funzione occorre specificare l'identificativo della VI in questione, *ViHandle* e il nome della funzione *Handler* che sarà svegliata dal completamento del descrittore. Vi è inoltre un altro campo, *Context*, utilizzabile come generico puntatore per inviare all'handler dati da utilizzare all'interno di esso. Infatti, non vi è altro modo per chiamare l'handler se non tramite la `VipRecvNotify` e qualsiasi parametro da utilizzare nell'handler medesimo, come variabili globali, strutture od altro, deve essere passato alla `VipRecvNotify` mediante il puntatore *Context* con opportune operazioni di casting.

Quando avviene il completamento del descrittore, il VI Provider lancia l'esecuzione dell'handler specificato nella `VipRecvNotify`, passandogli come parametri il puntatore *Context*, il Nic Handle dell'interfaccia in oggetto, il ViHandle della VI cui appartiene il descrittore completato ed il puntatore al descrittore medesimo.

Nel caso in cui, alla chiamata di `VipRecvNotify`, il descrittore in testa alla Receive Queue non risultasse completato, la funzione abiliterebbe gli interrupt per la Receive Queue in oggetto fino al realizzarsi del completamento stesso.

Una volta completato, il descrittore è rimosso dalla Receive Queue a cura dell'handler caricato in precedenza. L'handler, terminato il suo set di istruzioni tra cui vi sarà la gestione del messaggio appena ricevuto, cessa di esistere. È per questo motivo che si è scelto di richiamare la funzione medesima direttamente all'interno della funzione handler, in modo da essere pronti ad un nuovo completamento del descrittore successivo sulla Receive Queue.

Se un messaggio arriva prima dell'abilitazione di `VipRecvNotify`, il descrittore risulta completato ma il messaggio non risulta perduto, in quanto alla chiamata della funzione, quest'ultima invocherà l'handler immediatamente essendo già presenti sulla coda uno o più descrittori completati.

Viceversa, viene generato un errore (`VIP_DESCRIPTOR_ERROR`) nel caso in cui venga chiamata la funzione senza che vi sia nella Receive Queue un descrittore disponibile. È per questo motivo che nella funzione handler viene prima ripostato sulla coda il descrittore appena consumato, poi reinvocata la `VipRecvNotify`. Allo stesso modo, vista la ovvia necessità di invocare la prima istanza della funzione nella thread principale del programma, occorre, prima di tale operazione, aver postato almeno un descrittore sulla Receive Queue che dovrà necessariamente essere presente alla chiamata della funzione medesima.

È anche possibile registrare più handler contemporaneamente: saranno invocati in sequenza, nell'ordine di completamento dei descrittori della VI a cui sono associati. La distruzione della VI causa la cancellazione di tutte le chiamate alle funzioni pendenti in attesa di completamento dei descrittori.

La `VipRecvNotify` non può essere utilizzata per notificare il completamento di un descrittore su una VI a cui è associata una Completion Queue: in tale caso occorre fare riferimento alla funzione apposita `VipCQNotify`.

Il programma "ping pong", per sua natura realizzativa, prevede che ognuno dei due nodi subordini l'invio di un messaggio al ricevimento del precedente dal nodo remoto (a parte, ovviamente, la prima istanza). Per ottenere questa sincronizzazione si è scelto di ricorrere alla libreria `PThreads` che fornisce i sistemi di mutua esclusione e condizionali classici della teoria dei sistemi operativi per gestire l'accesso alle cosiddette zone critiche, cioè parti di codice che leggono o aggiornano variabili cui è necessario accedere in modo esclusivo e controllato. In pratica, si sono utilizzate le seguenti quattro funzioni:

- `pthread_mutex_lock(&mutex)`: pone in blocco la variabile argomento, permettendo l'esecuzione di tale operazione solo al primo di eventuali più processi che tentassero contemporaneamente di eseguirla. Gli altri verrebbero posti in blocco, in attesa che il primo rilasci il lock sulla variabile.
- `pthread_mutex_unlock(&mutex)`: è utilizzata dalla thread protetta per sbloccare la variabile di mutua esclusione, permettendo ad altre thread di accedere alla zona critica.
- `pthread_cond_wait(&cond,&mutex)`: all'interno della zona critica può verificarsi una condizione di stallo che richiederebbe l'intervento di un'altra thread eventualmente bloccata sulla condizione di lock della mutua esclu-

sione. Con questa funzione la thread attiva si pone in stato di idle e cede il controllo ad un'altra posta in lock dalla mutua esclusione.

- `pthread_cond_signal(&cond)`: quando la thread secondaria ha attuato la condizione per cui la principale può continuare il suo processo, esegue questa funzione di signal per sbloccare la thread precedente la quale, dopo aver verificato la positività della condizione, prosegue il processo o si ripone in stato di wait se la condizione non è verificata.

Lo schema in figura 3.11 esemplifica la successione temporale degli eventi nelle due thread: quella principale che lancia la prima istanza di `VipRecvNotify` e si occupa della parte di spedizione dei messaggi e la thread secondaria, rappresentata dall'Handler invocato al completamento di un descrittore a causa del ricevimento di un messaggio.

A completamento del presente capitolo, segue (tabella 3.8) la misura delle prestazioni del programma "ping pong" modificato in modo da utilizzare quest'ultima architettura di ricezione. In figura 3.12 sono messi a confronto i risultati di questa soluzione con quelli della soluzione con Wait bloccante, visto che anche in quest'ultimo caso vi è un'architettura bloccante (anche se multithread) sul completamento dei descrittori in ricezione. Infine, l'algoritmo 3 esemplifica la struttura della thread principale, mentre nell'algoritmo 4 è riportata la struttura dell'handler.

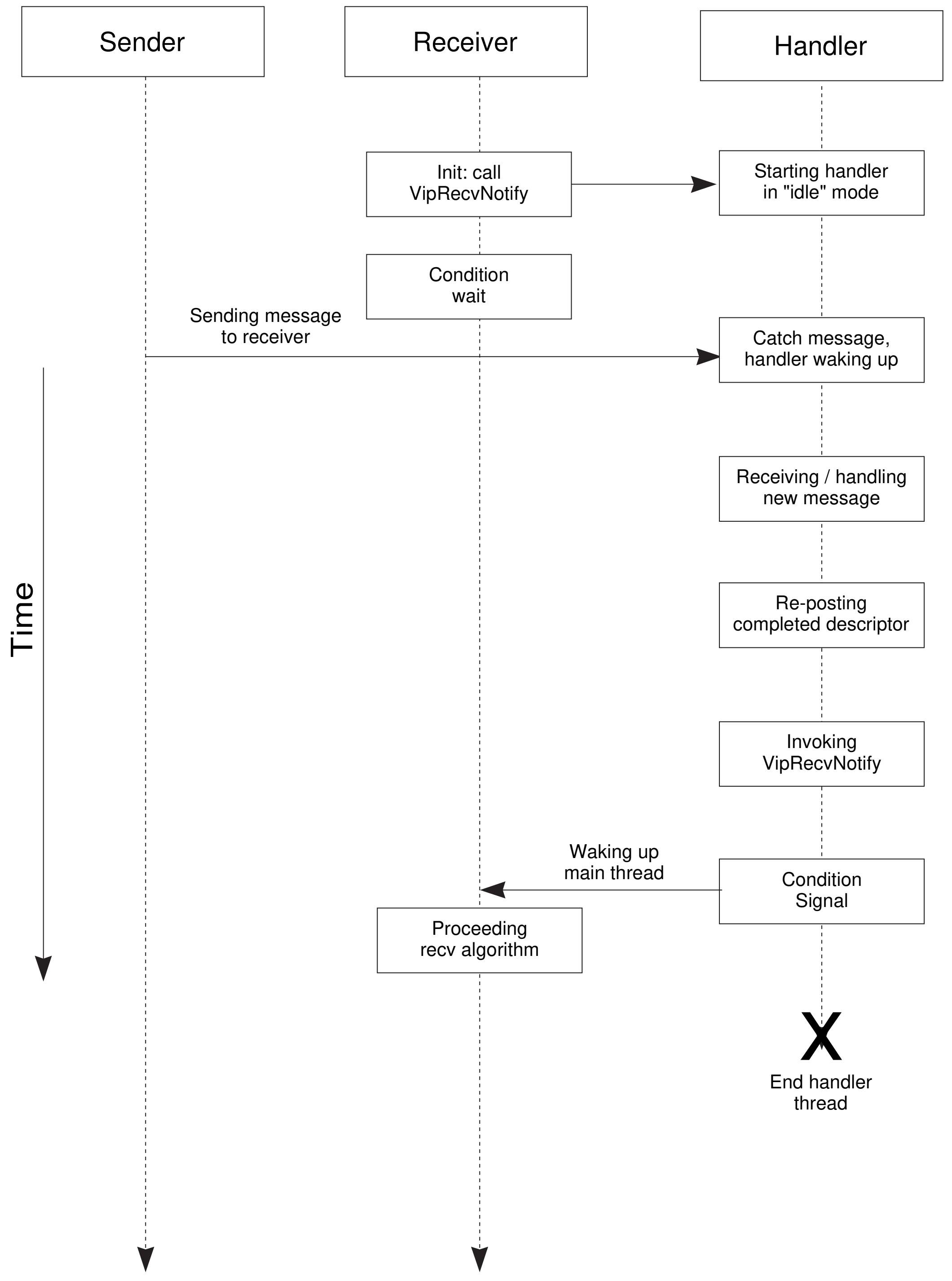

Figura 3.11: Schema di funzionamento della `VipRecvNotify` in un ping pong bloccante.

Dimensione del messaggio ***bytes***	***Tempo*** μs
1	89
2	89
4	89
8	89
16	89
32	89
64	95
128	106
256	127
512	168
1024	251
2048	114
4096	194
8192	365
16384	709
32768	1393

Tabella 3.8: Durata delle operazioni di comunicazione con metodo `VipRecvNotify`.

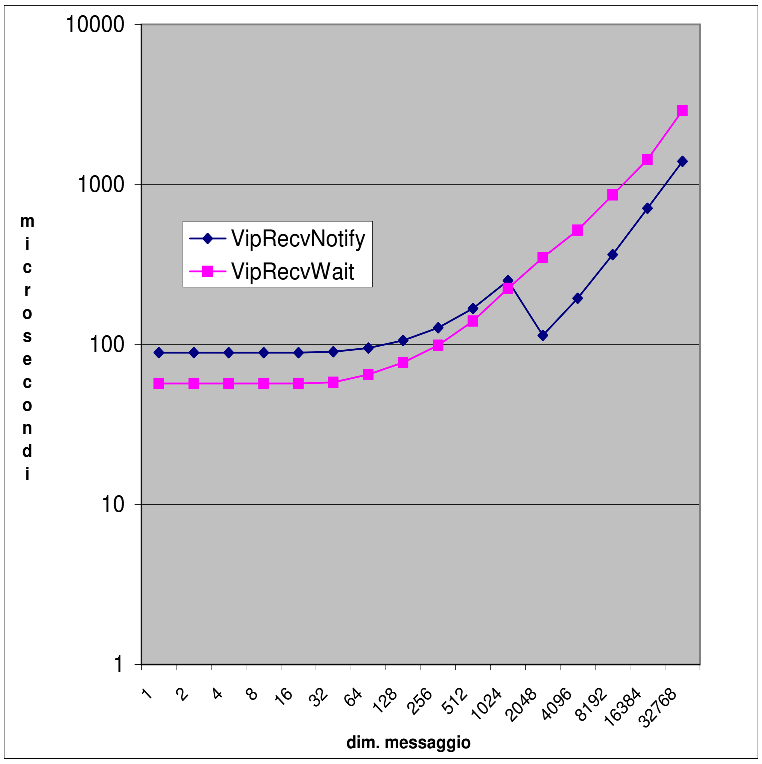

Figura 3.12: Confronto delle funzioni `VipRecvWait` `VipRecvNotify` in M-VIA.

Algoritmo 3 Schema della thread principale chiamante `VipRecvNotify`.

```
VipPostRecv(viHand, &desc, memHand);

VipConnectWait(nicHand, localAddr, VIP_INFINITE,
               remoteAddr, &remoteViAttrs, &connHand)

VipConnectAccept(connHand, viHand)

VipPostRecv(viHand, &desc[global_index], memHand);
printf("Added descriptor to tail of Recv queue\n");

VipRecvNotify(viHand, (VIP_PVOID) memHand, RecvHandler);

for(i=0; i<packets; i++) {
  pthread_mutex_lock(&mutex);

  while (queue[rcvd]==0)
    pthread_cond_wait(&cond,&mutex);

  pthread_mutex_unlock(&mutex);
  rcvd++;
  VipPostSend(viHand, descp, memHand);
  VipSendWait(viHand, VIP_INFINITE, &descp);
  }

printf("End of communication\n");
return(0);
```

Algoritmo 4 Schema dell'handler invocato da `VipRecvNotify`.

```
void
RecvHandler(VIP_PVOID contHandle,
            VIP_NIC_HANDLE nicHandle,
            VIP_VI_HANDLE viHand,
            VIP_DESCRIPTOR *descH) {

  printf("Received message n. %d\n",rcvd);

  if (i != packets-1) {
    VipPostRecv(viHand, descH, memHand);

    VipRecvNotify(viHand, (VIP_PVOID) memHand,
                  RecvHandler);
  }
  pthread_mutex_lock(&mutex);

  queue[rcvd]=1;
  pthread_cond_signal(&cond);

  pthread_mutex_unlock(&mutex);
}
```

CAPITOLO **4**

Analisi di LAM/MPI

Tra i diversi sistemi di programmazione distribuita, è utilizzato sul cluster ParMa^{2}lo standard *Message Passing Interface*, realizzato con la libreria LAM (*Local Area Multicomputer*). Vediamone nel dettaglio le caratteristiche.

4.1 Message Passing Interface

MPI è uno standard di message-passing portabile, che permette lo sviluppo di librerie di programmazione parallela. Tale standard definisce sintassi e semantica delle routine da utilizzare nelle programazioni in Fortran 77 e C. MPI è utilizzato per specificare i paradigmi di comunicazione tra un set di processi che formano un ambiente di programmazione concorrente, paradigmi di grande scalabilità e portabilità. Lo standard risulta infatti facilmente adattabile sia a cluster di computer sia a multiprocessori a memoria condivisa, sia a combinazioni delle due soluzioni. Inoltre ne è esclusa l'obsolescenza causata dall'incremento di velocità dell'hardware di rete o dal miglioramento dell'architettura degli stessi nodi.

4.1.1 Gruppi di processi

Un gruppo di processi è un insieme ordinato di processi, in cui ognuno di essi è identificato da un rank che ne rappresenta il numero d'ordine. Per un gruppo di n processi, il rank va da 0 ad $n - 1$. I gruppi di processi possono essere utilizzati in due principali modalità: innanzitutto il gruppo di processi può specificare quali di essi siano o meno coinvolti in comunicazioni collettive come il broadcast; poi, possono rappresentare diversi task in una singola macchina in cui è presente una qualsiasi forma di parallelismo.

Il modello a processi di MPI, a partire dalla versione 2, è di formato dinamico ed anche i gruppi di processi sono dinamici: possono, cioè, essere creati e distrutti ed ogni processo può appartenere a più gruppi contemporaneamente.

MPI fornisce routine per la creazione di nuovi gruppi mediante la specifica dei rank dei processi appartenenti al gruppo padre che dovranno far parte anche del nuovo, oppure mediante la specifica di criteri di partizionamento di un gruppo esistente secondo determinate chiavi.

4.1.2 I Context di comunicazione

I Context di comunicazione furono ideati inizialmente per permettere la creazione di stream di messaggi distinti e separabili, con un univoco contesto per ogni stream creato. Ora, l'utilizzo comune dei contesti è quello di assicurare che i messaggi spediti in una fase dell'applicazione non siano erroneamente intercettati da un'altra fase. Il principale vantaggio nell'utilizzo dei context si vede, ad esempio, nel caso in cui due fasi consistano in due chiamate ad altrettante differenti routine esterne: mentre il programmatore non ha modo di distinguere il traffico proveniente da una o dall'altra chiamata solo disponendo del tag, del rank o del gruppo, il Context fornisce un criterio addizionale per la selezione dei messaggi.

4.1.3 Gli oggetti Comunicatore

Lo scopo di una operazione di comunicazione è specificato dal context utilizzato e dal gruppo o dai gruppi coinvolti nel processo. In una comunicazione collettiva o in una comunicazione punto a punto tra membri dello stesso gruppo, occorre specificare solo un gruppo, mentre i processi sorgente e destinazione sono individuati direttamente dal loro rank in quel gruppo. Al contrario, in caso di comunicazione punto a punto tra processi di gruppi differenti occorre specificare entrambi i gruppi. In questo caso i processi sorgente e destinazione sono individuati dai rank nei rispettivi gruppi. In MPI sono utilizzati oggetti astratti chiamati *comunicatori*, per definire lo scopo dell'operazione di comunicazione. Essi vengono passati a tutte le comunicazioni collettive e punto a punto per specificare il context e il gruppo o i gruppi coinvolti nell'operazione di comunicazione.

4.1.4 Comunicazioni punto a punto

MPI fornisce un set di funzioni di spedizione e ricezione che permettono la comunicazione punto a punto con selettività basata sul processo sorgente, sul tag e sul *Context*. I primi due possono anche essere generici, se la selezione non si riferisce ad essi, ma il context deve sempre essere specificato. I processi sorgente e destinazione sono individuati mediante un gruppo ed un rank. In caso di comunicazioni tra il medesimo gruppo, l'identificazione di quest'ultimo è legata al context a formare un comunicatore.

Le operazioni di comunicazione possono essere sia di tipo bloccante, sia di tipo non bloccante. Nel primo caso, la chiamata alla funzione di spedizione blocca il processo finché il buffer di spedizione non può essere riutilizzato (di fatto, cioè, quando il messaggio è effettivamente inviato), mentre la funzione di ricezione blocca il processo finché il relativo buffer di ricezione non contiene effettivamente il messaggio che si sta attendendo.

Nel caso di funzioni di comunicazione non bloccanti, MPI scinde tali funzioni in due parti: la spedizione o la ricezione vera e propria e una successiva funzione di *test for completion*, che si occupa di verificare, mentre il processo continua la sua esecuzione con le istruzioni successive alla send, se quest'ultima è effettivamente stata completata con successo.

Occorre, a questo punto, sapere se, in seguito alla spedizione del pacchetto e al corretto completamento dell'operazione di spedizione, il lato ricevente abbia ricevuto correttamente quanto inviatogli, o quantomeno se abbia iniziato la ricezione del pacchetto. La risoluzione di questa problematica, intesa come l'invio di una sorta di conferma di avvenuta ricezione da parte del ricevente verso il nodo di spedizione, si pone contro uno snellimento del protocollo atto a privilegiare il lato prestazionale dell'architettura.

MPI gestisce queste complessità mediante l'introduzione dei *modi* di comunicazione punto a punto:

- Nel modo **standard** il completamento delle operazioni di send non significa necessariamente che la corrispondente funzione di receive abbia avuto inizio e, di conseguenza, nessuna assunzione relativa a ciò deve essere fatta dall'applicazione;

- Il modo **synchronous** prevede una struttura ad "appuntamento", in cui la spedizione del messaggio coincide, di fatto, con l'inizio della sua ricezione al nodo remoto;

- Il modo **ready** prevede la potenzialità di semplificare il protocollo da parte dell'utente in modo da privilegiare le performance di comunicazione. In questa modalità il lato di spedizione assume che il corrispondente nodo ricevente sia di fatto pronto alla ricezione.

Per ognuna di queste tre modalità di comunicazione, l'operazione di send può essere bloccante o non bloccante, rendendo effettivamente sei i possibili tipi di scambio messaggi. In ogni caso, una operazione di send non può terminare senza assicurare all'applicazione il possibile riutilizzo del buffer di spedizione, evitando possibili corruzioni dei dati appena inviati. Una send non bloccante non attende alcun particolare evento prima del termine, ma ritorna un handle ad un comunicatore che può essere successivamente utilizzato da routine che controllino in seguito il completamento della send stessa.

Anche l'operazione di receive può comportarsi in modo bloccante o non bloccante, portando a sei anche dal lato ricevente i possibili modi di comunicazione. Una receive bloccante non termina la sua funzione fino alla presenza dei dati ricevuti nel buffer alla locazione indicata dalla receive stessa. Un receive non bloccante ritorna un handle ad un comunicatore, utilizzabile dall'applicazione per controllare lo stato della receive o per bloccarla fino all'effettivo completamento. La receive non bloccante fornisce inoltre un secondo handle ad un "return status object", utilizzato per memorizzarvi lunghezza, sorgente e tag del messaggio stesso. Al termine della receive queste informazioni possono essere recuperate mediante la chiamata ad un'apposita routine.

4.1.5 Comunicazioni collettive

Le comunicazioni collettive trasmettono dati tra tutti i processi in comunicazione, tramite un oggetto comunicatore. La funzione *barrier* ha il compito di sincronizzare tutti i processi senza con ciò scambiare dati significativi per l'applicazione. Nelle comunicazioni collettive non vi sono forme di comunicazioni non bloccanti.

Le comunicazioni collettive possono essere suddivise in due grandi categorie: routine di movimento dati e routine computazionali.

Tra le prime individuiamo:

- Broadcast;
- Gather;
- Scatter.

Di ognuna di esse vi sono due versioni: nella versione "uno a tutti" i dati sono inviati da un processo a tutti gli altri, mentre nella versione "tutti a tutti" ogni processo invia dati a tutti gli altri. Tutti i processi devono spedire e/o ricevere buffer del medesimo tipo e della medesima lunghezza.

La funzione broadcast uno a tutti spedisce dati da un processo a tutti gli altri nel gruppo, mentre l'operazione broadcast tutti a tutti prevede che ognuno dei processi nel gruppo invii dati a tutti gli altri, ed al completamento ogni processo ha ricevuto gli stessi dati. In questo modo, ogni processo termina l'operazione con il medesimo buffer di uscita, che rappresenta la concatenazione dei buffer d'ingresso di tutti i processi posti in ordine di rank.

L'operazione di scatter uno a tutti spedisce dati distinti da un processo a tutti gli altri processi nel gruppo. È una comunicazione personalizzata di un processo verso tutti gli altri del gruppo. Nella comunicazione scatter tutti a tutti ogni processo spedisce un dato differente ad ogni altro processo nel gruppo, in modo che ogni processo del gruppo riceva dati distinti da ogni altro.

Si invertono le direzioni del percorso nella comunicazione gather uno a tutti: un processo radice riceve dati da tutti i processi del gruppo, formando nel suo buffer una concatenazione dei buffer d'ingresso di tutti i processi del gruppo in ordine di rank. La comunicazione gather tutti a tutti è identica alla scatter tutti a tutti.

Tra le routine computazionali sono due: *reduce* e *scan* che richiedono entrambe la specificazione di una funzione di input. Tale funzione di input può essere specificata tra una lista predefinita oppure mediante un puntatore alla funzione stessa. In tal modo, le operazioni computazionali assumono quattro differenti versioni.

La fugura 4.1 rappresenta alcuni dei precedenti tipi di comunicazione in maniera schematica per un gruppo di 6 processi. In ogni esempio, ciascuna riga rappresenta la posizione dei dati in ogni processo: nel primo caso, ad esempio, prima del broadcast il dato A_0 è contenuto solo nel primo processo, dopo l'operazione lo contengono tutti.

4.2 Local Area Multicomputer

La libreria di programmazione LAM, scelta per realizzare le specifiche MPI sul cluster ParMa^2, è costituita da due livelli: "upper layer" e "lower layer".

Il primo è portabile ed indipendente dal sottosistema di comunicazione e si interfaccia con il lower layer tramite la *Request Progression Interface*. La RPI è costituita da sette funzioni, destinate ad una realizzazione dedicata alle specifiche architetture, per ottenere comunicazioni dirette ad alte prestazioni tra i processi MPI. Queste funzioni possono essere riscritte per integrare LAM a nuove architetture di comunicazione, quale ad esempio VIA. Queste funzioni sono indicate con C2C-RPI. Le funzioni riguardano in particolare l'inizializzazione delle connessioni per

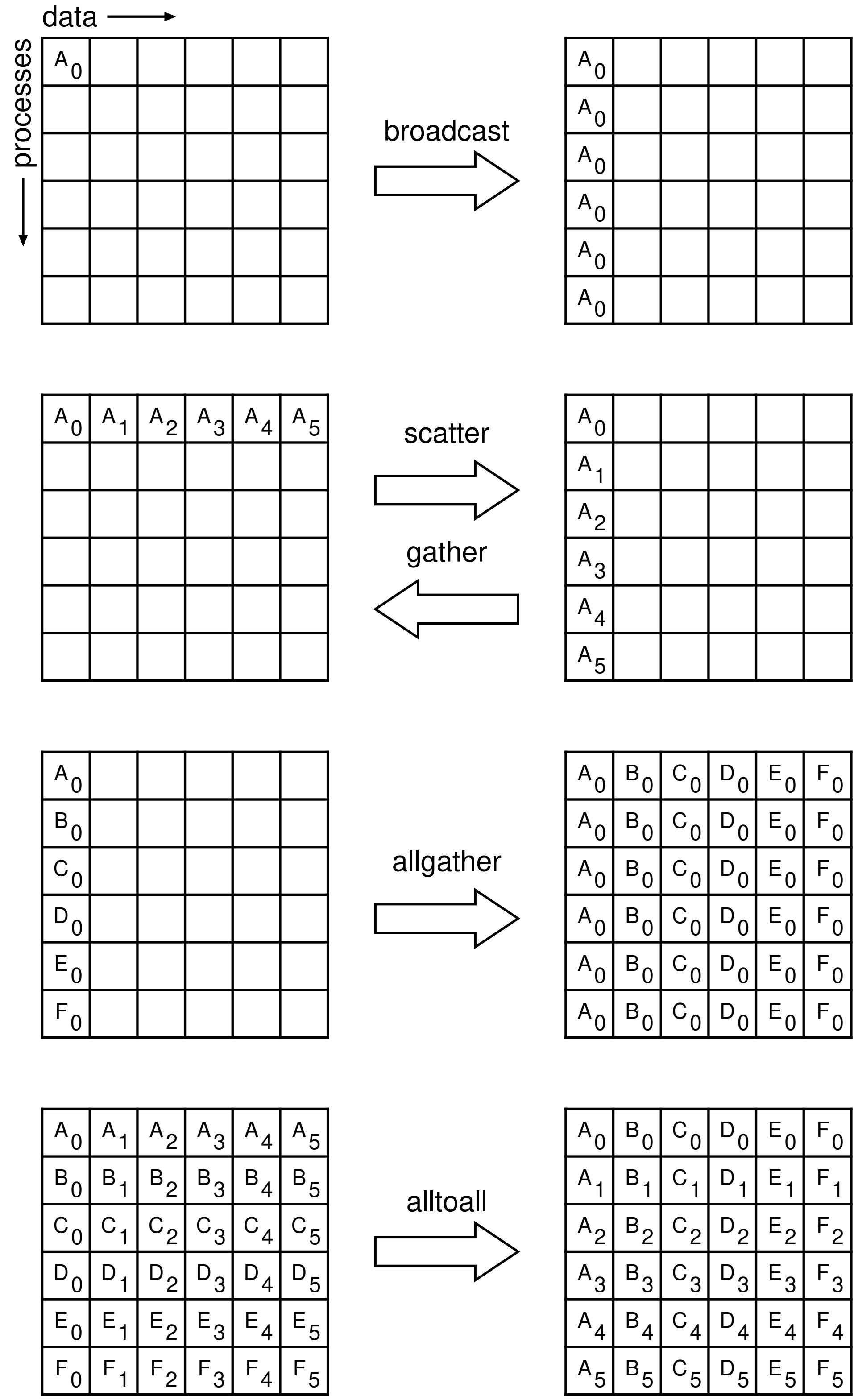

Figura 4.1: Esempi di tipi di comunicazione collettiva.

20

Figure 3: Collective move functions illustrated for a group of six processes. In each case, each row of boxes represents data locations in one process. Thus, in the broadcast, initially just the first process contains the data A_0, but after the broadcast all processes contain it.

il trasporto dei dati tra i vari processi, il trasporto dei messaggi attraverso la rete, la loro sincronizzazione e la distruzione delle comunicazioni al termine delle transazioni.

L'upper layer, per contro, si occupa di operazioni di alto livello come la costruzione e la distruzione dei buffer, la conversione dei dati, ecc.

La libreria LAM-MPI contiene diverse strutture dati. Le più importanti fra esse, per come è concepita la raccolta di funzioni RPI, sono la lista delle richieste e la lista dei processi. L'upper layer si occupa della creazione e dell'inizializzazione di tali strutture, mentre la C2C-RPI può inserire in esse ulteriori dati attraverso handler forniti dall'upper layer.

4.2.1 Strutture dati

Processi

Un processo MPI in LAM è rappresentato da una struttura del tipo `struct _proc`. Ogni processo mantiene una lista di tutti i processi attivi di cui è a conoscenza e con cui può comunicare. Anche il processo stesso compare nella lista ed è puntato dalla variabile globale `__lam_myproc__`. La lista dei processi è opaca e può essere visitata mediante le funzioni accessorie `__lam_topproc__()` e `__lam_nextproc__()`.

Request

Per ogni processo la libreria LAM mantiene una lista aggiornata di tutte le richieste che necessitano di essere processate. L'upper layer tiene aggiornata questa *progression* list e rimuove le richieste una volta completate. Una request è rappresentata da una struttura `struct _req`.

Il ciclo di vita di una richiesta segue questo schema:

Building: lo spazio in memoria viene allocato ed inizializzato con una richiesta. La richiesta è posta nello stato `LAM_RQSINIT`. Non è ancora processabile e non fa ancora parte nella lista dei request.

Starting: la richiesta è ora linkata alla lista ed è candidata ad essere processata. Non è necessario a questo punto alcun trasferimento dati da parte di C2C-RPI, ma solo che sia correttamente aggiornato il progression state. La richiesta, quindi, viene posta in uno dei tre stati `LAM_RQSSTART`, `LAM_RQSACTIVE` o `LAM_RQSDONE`.

Progression: la richiesta viene portata a termine: è spostata dallo stato iniziale allo stato attivo non appena il dato è trasferito ed è successivamente posta allo stato completo una volta che tutti i dati attesi sono stati trasferiti e che tutti gli eventuali acknowledgement sono stati ricevuti o spediti.

Completion: appena completata, la richiesta è posta nello stato iniziale se è di tipo persistente, o viene distrutta se è di tipo non persistente.

4.2.2 Le funzioni RPI

Vengono ora descritte le otto funzioni C2C-RPI. Nella descrizione sarà evidenziato da dove ognuna di esse viene chiamata ed analizzato il compito specifico del livello C2C e di cosa l'upper layer si aspetta di ricevere.

Tutte le funzioni di seguito elencate, con l'esclusione di `_rpi_c2c_advance()` e `_rpi_c2c_iprobe()`, restituiscono 0 in caso di esecuzione corretta e -1 in caso di errore.

`int _rpi_c2c_init(void)`

Effettua l'inizializzazione primaria del livello C2C-MPI ed è chiamata da `MPI_Init`. Esegue l'inizializzazione di tutti i processi rispetto alle funzioni di comunicazione. È poi necessaria solo l'esecuzione di `_rpi_c2c_addprocs()` per inizializzare il gruppo di nuovi processi. Al momento di effettuare questa chiamata, le routine LAM di scambio messaggi `nsend()` e `nrecv()` sono disponibili ed utilizzabili per scambiare messaggi tra i processi MPI.

`int _rpi_c2c_addprocs(void)`

In LAM-MPI un processo può disporre di nuovi processi con cui comunicare, ad esempio mediante la creazione di processi figli. L'upper layer aggiunge nuovi processi alla corrispondente lista e chiama la funzione `_rpi_c2c_addprocs()` per eseguirne l'inizializzazione.

`int _rpi_c2c_build(MPI_request req)`

Quando l'upper layer crea una nuova richiesta inizializza le informazioni relative e chiama questa funzione per eseguire la parte C2C-RPI della richiesta.

`int _rpi_c2c_start(MPI_Request req_top, MPI_Request req)`

L'upper layer, dopo aver aggiunto una richiesta alla progression list, chiama la funzione `_mpi_req_start()` per preparare tale richiesta alle successive esecuzioni.

Tra i rimanenti compiti, cambia lo stato della richiesta a "start" e chiama la `_rpi_c2c_start()` in modo da permettere al C2C-RPI di effettuare le inizializzazioni che la portano in stato di "ready". In caso vengano effettuate altre operazioni alla richiesta, il relativo stato deve essere necessariamente tenuto aggiornato; in particolare, i possibili stati successivi allo "start" possono essere lo stato attivo, in cui il protocollo di trasferimento dati non risulta ancora completato, ma alcuni dati sono già stati trasmessi e i relativi buffer possono essere cancellati, o lo stato "done", in cui il protocollo di trasferimento è completato e la richiesta è completamente eseguita.

`int _rpi_c2c_destroy(MPI_Request req_top, int flag_block)`

Distrugge la porzione C2C-RPI della richiesta. Essa è solitamente chiamata da `_mpi_req_destroy()`. Questa funzione dovrebbe liberare qualsiasi allocazione dinamica creata per questa richiesta dall'interfaccia RPI ed eseguire le ulteriori necessarie cancellazioni.

`int _rpi_c2c_advance(MPI_Request req_top, int flag_block)`

È la principale chiamata di comunicazione. Dato un puntatore all'inizio della progression list, lo incrementa fin dove può. Il flag specificato come secondo argomento è "true" se è possibile porsi in stato bloccante finché non risulta effettuato un avanzamento della richiesta. L'upper layer non si occupa dei protocolli di trasferimento dei messaggi e della gestione dei buffer, ma demanda tutto a C2C-RPI il quale conduce l'avanzamento della richiesta attraverso gli stati ("start", "active", "done").

Una richiesta può essere condotta dallo stato di "start" a "done" senza il canonico percorso RPI, semplicemente quando viene cancellata. Attualmente non è possibile, però, effettuare una cancellazione di una richiesta se essa si trova già nello stato attivo.

`int _rpi_c2c_iprobe(MPI_Request req)`

Chiamata da `MPI_Iprobe()`, implementa l'operazione di "probe". Genera una richiesta "probe" non bloccante, già posta in stato "start". Questa funzione verifica il risultato della probe in modo non bloccante, restituendo 0 se nulla è stato trovato, 1 se una occorrenza è presente e -1 in caso di errore.

`int _rpi_c2c_finalize(void)`

Esegue la cancellazione del sottosistema C2C-RPI, eliminando tutte le strutture dati create. La funzione è invocata da `MPI_Finalize()` dopo che hanno avuto termine tutte le operazioni pendenti di comunicazione.

CAPITOLO 5

Introduzione di M-VIA in LAM

La realizzazione delle funzioni RPI con l'utilizzo del protocollo di comunicazione VIA richiede, di fatto, la riscrittura dell'intero lower layer di MPI, in modo da sfruttare appieno le primitive che VIA mette a disposizione. MPI richiede che possano essere effettuate operazioni di tipo bloccante, non bloccante, sincrone ed asincrone con conseguenti caratteristiche piuttosto eterogenee.

5.1 Scelte progettuali

VIA mette a disposizione, come già abbiamo analizzato, due modalità di comunicazione: Send/Receive ed RDMA. Dovendo rendere disponibile la modalità di comunicazione asincrona, occorre escludere a priori il modello Send-Receive, il quale richiede la disponibilità del nodo ricevente ad accogliere il messaggio nel momento in cui questo viene spedito; ciò è ovviamente incompatibile con una operazione asincrona. Diventa perciò obbligata la scelta del modello RDMA.

Ogni processo, infatti, deve poter spedire i messaggi in qualsiasi momento, indipendentemente dalla condizione del lato ricevente. Allo stesso modo, in ricezione il processo deve potersi mettere in attesa senza conoscere l'istante di partenza del messaggio dal nodo di spedizione.

Inoltre, lo standard MPI consente di sincronizzare la comunicazione tra nodi, rendendo necessaria la realizzazione di un meccanismo basato su VIA adatto allo scopo. Occorre anche gestire la richiesta di notifiche di avvenuto completamento di operazioni di send e receive. Infine occorre assicurare in modo affidabile una corretta gestione delle risorse, con particolare riferimento al loro esaurimento che provocherebbe errori critici, fino all'interruzione delle applicazioni.

Il tipo di comunicazione con RDMA ha, però, un'importante limitazione: non è possibile avere, in ritorno da una operazione, una notifica dell'avvenuto completamento. È d'altra parte necessario, sia per le operazioni di Send, sia per quelle di Receive poter essere informati sull'avvenuta conclusione dell'operazione, per poter riutilizzare il buffer impegnato dal lato send o per leggere il buffer con i dati appena arrivati dal lato receive.

Soluzione al problema è rappresentata dalla comunicazione con RDMA Immediate Data. In questa modalità il lato ricevente consuma un descrittore sulla Receive Queue all'avvenuto completamento dell'operazione di receive. Questo evento autorizza la lettura del buffer di ricezione garantendo la presenza dei dati attesi.

Per eseguire correttamente un'operazione di RDMA occorre preventivamente scambiare gli indirizzi di memoria delle locazioni che contengono i dati da spedire e che andranno a ricevere i dati in arrivo. Tale operazione andrà ovviamente eseguita con operazioni di tipo Send/Receive sincrone.

Un'altra problematica che occorre considerare riguarda la necessità per VIA di registrare preventivamente la memoria dei buffer utilizzati. Dal lato sender si è scelta la dimensione del buffer pari a quella massima consentita da M-VIA. Dal lato receive si è registrata un'ampia regione di memoria (RDMA area) gestita con un controllo di flusso descritto in seguito nel dettaglio.

Infine occorre notare che VIA prevede esclusivamente comunicazioni punto a punto, mentre MPI prevede, come abbiamo visto nel precedente capitolo, comunicazioni collettive di vario tipo. È necessario dunque conciliare queste due caratteristiche. Inoltre MPI permette di specificare il tag "MPI_ANY_SOURCE", mediante il quale il processo può attendere un messaggio, a lui destinato, da uno qualsiasi dei nodi connessi. Questa possibilità deve quindi essere gestita dal lower layer dell'RPI per estendere le comunicazioni punto a punto di VIA.

Vi sono diverse strade percorribili per realizzare questa soluzione, ognuna caratterizzata dall'utilizzo di un differente strumento messo a disposizione da VIA. Le scelte operabili sono:

Utilizzo delle Completion Queue

Le Completion Queue, come già abbiamo accennato, sono code in cui vengono notificati i completamenti dei descrittori sulle code di Send e di Receive della relativa VI. Ricordiamo che ad ogni VI è associata una ed una sola CQ. Inoltre, in caso di utilizzo delle CQ occorre fare sempre e solo riferimento ad esse, rinunciando alle altre primitive di comunicazione.

Vi sono due soluzioni attuabili mediante le Completion Queue: l'utilizzo di una CQ o di due di esse. Queste proposte non riescono però a fornire una soluzione efficiente e completa al problema della comunicazione in cluster di nodi multiprocessore.

Con due Completion Queue, infatti, ognuna viene associata ad un ben preciso NIC: una al NIC di rete, relativo quindi a tutte le comunicazioni istanziate con nodi differenti dal proprio e l'altra al NIC di loopback, in modo da gestire le comunicazioni da e verso se stesso. Svantaggio di tale soluzione è la necessità di un continuo polling su due differenti code, che porta ad un non ottimale utilizzo delle risorse.

L'utilizzo di una sola Completion Queue ottimizza invece l'utilizzo delle risorse, ma porta lo svantaggio di dover utilizzare per ogni tipo di comunicazione il NIC di rete. Quindi, un messaggio in partenza e destinato al medesimo nodo sorgente non può sfruttare il device di loopback, ma passa dal NIC di rete.

Polling diretto sulle VI

Come si è visto nel capitolo 3, è possibile eseguire operazioni di polling continuo su ogni VI, ricercando in continuazione il messaggio che si sta aspettando mediante un ciclo che termina all'effettivo arrivo del messaggio. Questa soluzione è accantonata per l'enorme dispendio di risorse di cui necessita.

Utilizzo della `VipRecvNotify`

La soluzione infine adottata, resasi disponibile a partire dalla distribuzione M-VIA 1.0, comporta l'utilizzo della primitiva `VipRecvNotify`, mediante l'architettura dettagliatamente descritta nel capitolo 3. Purtroppo questa primitiva risulta disponibile solo con la versione software di M-VIA, mentre va a far parte delle funzioni non ancora realizzate dalla versione hardware cLAN di Emulex. Per tale motivo, quindi, il porting di LAM-MPI su VIA è stato sviluppato facendo esclusivo riferimento ad M-VIA.

Un'ultima problematica da affrontare riguarda il controllo di flusso sulle risorse disponibili: è noto che errori critici vengono generati in caso di mancanza di descrittori sulle code o in caso di incorretta gestione della memoria RDMA. La gestione del numero dei descrittori risulta molto semplificata dall'utilizzo della funzione `VipRecvNotify` in quanto, come già si è illustrato, l'handler relativo alla funzione provvede a ripostare il descrittore utilizzato. Dal lato spedizione, viceversa, viene anticipatamente postato un ridotto numero di descrittori per coprire le esigenze iniziali di scambio degli indirizzi, mentre successivamente viene prepostato di volta in volta il descrittore all'atto della spedizione vera e propria. Per il controllo della RDMA area è stato creato un meccanismo di puntatori che indicano il punto in cui la memoria è accessibile, lo spazio residuo e quello occupato.

5.2 Riscrittura della Request Progression Interface

Analizziamo ora la nuova costruzione del Lower Layer, descrivendo le caratteristiche di ognuna delle otto funzioni RPI realizzate con le primitive fornite da M-VIA [11].

`_rpi_c2c_init:`

È stata associata alla principale funzione di inizializzazione la creazione dei protection tag per la registrazione della memoria. Inizializza inoltre i NIC relativi alle code di comunicazione e il name service di M-VIA.

`_rpi_c2c_addprocs:`

Si occupa di creare la hash table in cui i processi ricercheranno i messaggi desiderati e vi inserisce la struttura dati `VIInfo`, contenente tutte le informazioni relative alla Virtual Interface corrente.

L'algoritmo 5 mostra i campi della struttura `VIInfo`, mettendo in evidenza i seguenti elementi:

- il rank del processo connesso;
- il puntatore al NIC necessario alla comunicazione;
- il protection tag;
- un identificativo della VI connessa, utile al VI Provider;
- un puntatore ai descrittori delle code di spedizione e ricezione;
- gli indirizzi delle RDMA Area coinvolte;
- le variabili per il controllo di flusso dell'area RDMA stessa.

Inoltre, la funzione invoca un handler per la gestione della ricezione dei messaggi tramite la chiamata a VipRecvNotify.

`_rpi_c2c_build:`

Oltre alle funzioni normali definite da LAM, inserisce un campo nell'envelope del messaggio, con il quale viene discriminato il messaggio che ha bisogno di un acknowledge da quello che non necessita dell'avviso e dall'acknowledge stesso. Le tre condizioni sono rappresentate, rispettivamente, da un numero positivo, dallo zero e da un numero negativo.

Algoritmo 5 La struttura `VIInfo`.

```
struct VIInfo {
  int4 rank;   /* peer process's rank */

  VIP_NIC_HANDLE nic; /* VIA nic for this VI */
  VIP_PROTECTION_HANDLE ptag;
  VIP_VI_HANDLE vi;

  /* descriptors */
  VIP_DESCRIPTOR *sendDesc;
  VIP_DESCRIPTOR *recvDesc;
  VIP_MEM_HANDLE sendDescHandle;
  VIP_MEM_HANDLE recvDescHandle;

  /* data buffers (recv send) and envelopes (send) */
  char *sendBuff;
  int  *envBuff;
  char *recvBuff;
  VIP_MEM_HANDLE sendBuffHandle;
  VIP_MEM_HANDLE envBuffHandle;
  VIP_MEM_HANDLE recvBuffHandle;

  int recvNum; /* current recv buffer and descriptor */
  int sendNum; /* current send buffer and descriptor */
  int RDMAreadpointer; /* to last RDMAArea read */
  int RDMAreadlimit;
  int RDMAreadlimitupdated;

  int RDMAnextpointer;  /* to 1st RDMA free byte*/

  int RDMAwritepointer; /* to last RDMA written byte */
  int RDMAmaxwrite; /* to last RDMA Area wrtble byte */

  VIP_MEM_HANDLE RDMAreadlimitHandle;

  int buff;
  VIP_MEM_HANDLE buffHandle;

  struct RDMAAddress addr;
  struct remoteRDMAAddress remoteAddr;
}
```

`_rpi_c2c_start`:

Non vi è nessuna particolare caratteristica oltre all'avanzamento delle richieste al processo corrente.

`_rpi_c2c_destroy`:

Non svolge alcuna funzione nella presente realizzazione: ogni routine di terminazione dei collegamenti è affidata all'ultima delle funzioni qui elencate.

`_rpi_c2c_advance`:

È la principale funzione riguardante la comunicazione. Invoca le chiamate di spedizione o ricezione, mediante le primitive di comunicazione di VIA.

`_rpi_c2c_iprobe`:

Avanza, per quanto possibile, le richieste non bloccanti di Probe, senza l'utilizzo della `_rpi_c2c_advance`.

`_rpi_c2c_finalize`:

Duale del primo, disconnette e distrugge le VI attive, deregistra la memoria permettendone il riutilizzo al sistema operativo, chiude il Name Service e il NIC di comunicazione e, eliminando le liste concatenate, rende di nuovo disponibile la memoria da esse occupata.

Descriviamo ora nel dettaglio le funzioni strettamente correlate alla spedizione e alla ricezione dei messaggi.

5.3 Le funzioni di spedizione

Tali funzioni sono previste dallo standard MPI in quattro differenti formati:

- send standard;

- send ready;
- send bufferizzata;
- send sincrona.

I primi tre casi sono gestiti tutti dal medesimo algoritmo di send standard: la send ready grazie alla possibilità permessa dallo standard MPI, mentre la send bufferizzata può essere trattata in modo standard perché è l'upper layer che si occupa della copia del messaggio in un'altra zona di memoria in caso di "MPI_Bsend".

Caso a parte, invece, è costituito dalla send sincrona. Quest'ultima viene gestita richiedendo un segnale di acknowledge dal processo mittente a quello ricevente, mediante il campo apposito nell'envelope del messaggio. Il destinatario si fa quindi mittente a sua volta del segnale di ack, spedito anch'esso tramite l'envelope come descritto nel paragrafo precedente. Possono essere effettuate richieste di acknowledgement bloccanti o non bloccanti. In caso di richiesta non bloccante, non vi è nessun particolare comportamento da parte dell'algoritmo di spedizione, se non la notifica del completamento della ricezione subordinato alla ricezione dell'acknowledgement. In caso, viceversa, di richiesta bloccante, il processo di send si arresterà in attesa dell'acknowledge, senza però instaurare una attesa attiva come sarebbe accaduto nella gestione mediante Completion Queue, ma demandando all'handler della primiiva `VipRecvNotify` il compito di monitoraggio della propria coda di ricezione. Questo compito è svolto mediante l'utilizzo di una variabile condizione, posta in stato di wait dalla trama principale della send e risvegliata mediante "signal" dalla thread "handler" alla ricezione di un messaggio. In figura 5.1 è illustrata la sequenza di operazioni di gestione dell'acknowledge bloccante.

5.4 Le funzioni di ricezione

Grazie alla disponibilità della primitiva di ricezione asincrona `VipRecvNotify`, è stato possibile realizzare le funzioni di ricezione del lower layer di LAM senza

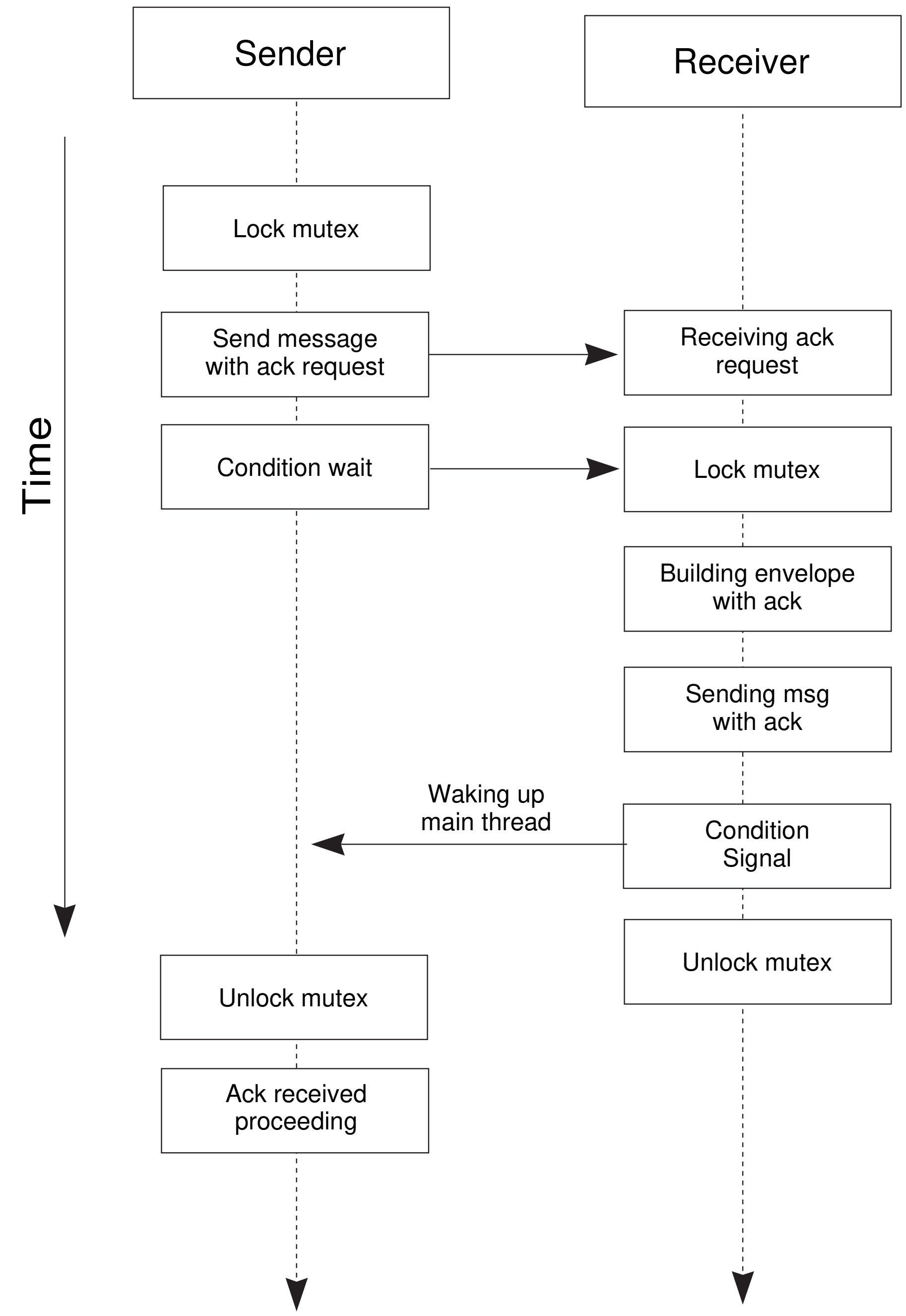

Figura 5.1: Schema di funzionamento della send sincrona bloccante.

dover ricorrere a polling attivi di code o Completion Queue. Il principio di funzionamento della primitiva menzionata è già stato ampiamente illustrato nel capitolo 3. In questo caso, come nel ping pong, un handler viene caricato durante le fasi di inizializzazione (figura 5.2) invocando per la prima volta la `VipRecvNotify`. A tale scopo è necessario il preposting di un esiguo numero di descrittori, in particolare ne viene utilizzato uno per le operazioni preliminari di scambio di indirizzi delle RDMA Area dei processi connessi ed uno per permettere il corretto caricamento del primo handler.

Successivamente, la routine di ricezione non ha più nulla a che fare con la diretta gestione delle VI, in particolare non chiama nessuna primitiva di VIA direttamente.

L'handler, risvegliato alla ricezione di un messaggio, esegue i seguenti passi (figura 5.3):

- pone in blocco la variabile mutex che difende la sezione critica di gestione delle liste concatenate;
- esegue le operazioni necessarie al controllo di flusso sulla RDMA Area, come descritto in seguito;
- legge il messaggio e lo pone in coda nella lista apposita;
- esegue una "signal" sulla variabile condizione per notificare ad eventuali routine di ricezione in attesa che è presente un nuovo messaggio nelle liste;
- riposta un descrittore in sostituzione a quello appena completato;
- carica un nuovo handler invocando la primitiva `VipRecvNotify`.

La ricezione di un messaggio può essere operata in modo bloccante o non bloccante. In quest'ultimo caso, se il messaggio non è presente nelle liste, come nel caso della send, il compito di attesa è demandato all'handler, senza sovrautilizzare le risorse di sistema per continue interrogazioni alle CQ o alle VI. Viceversa, in caso di ricezione bloccante la routine di ricezione si pone in attesa sulla variabile condizione ed attende che l'handler, sempre in modo passivo, la risvegli all'arrivo

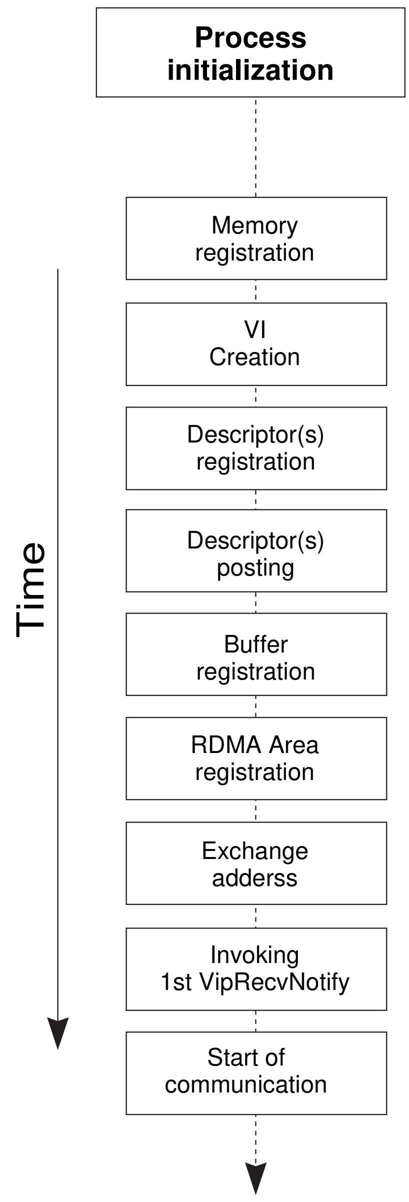

Figura 5.2: Successione degli eventi di inizializzazione della comunicazione.

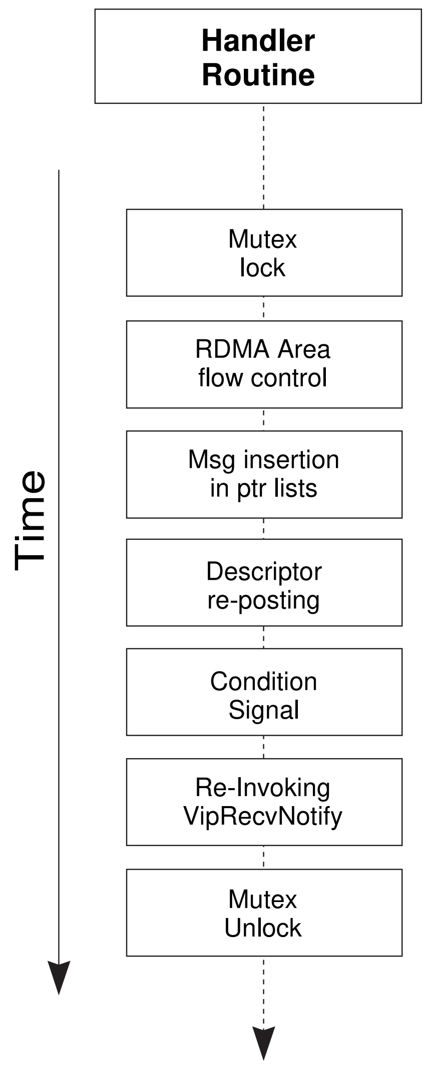

Figura 5.3: Handler gestore dell'arrivo dei messaggi.

di un nuovo messaggio mediante una signal sulla stessa variabile. Se il messaggio pervenuto è di interesse la routine ricevente prosegue la sua esecuzione, viceversa si pone nuovamente in attesa sulla medesima variabile condizione (figura 5.4).

5.5 La gestione delle Liste Concatenate

Segue una breve descrizione della modalità di gestione delle liste di gestione dei messaggi ricevuti [11]. Il meccanismo è studiato avendo cura di evitare copie dei messaggi in memoria nelle fasi intermedie, di mantenere l'ordine di arrivo dei messaggi per una corretta gestione dei messaggi provenienti da se stessi e di mantenere infine le informazioni aggiuntive riguardanti il controllo di flusso delle regioni di memoria.

Le liste sono costituite da puntatori alle envelope dei messaggi. Ogni elemento è costituito da una struttura contenente un puntatore all'inizio del messaggio e un intero indicante anch'esso l'inizio del messaggio, ma espresso come scostamento dall'inizio della RDMA Area corrispondente. Ciò è necessario per una limitazione del linguaggio di programmazione riguardante il confronto di due puntatori di tipo differente. Il numero di tali liste è ovviamente pari al numero dei processi con cui il rank in questione è connesso, che a sua volta è pari al numero delle aree RDMA da gestire.

Uno schema della composizione delle liste è illustrato in figura 5.5.

5.6 Il controllo di flusso

È stata realizzata [1] una serie di meccanismi volti a tenere sotto controllo le risorse relative alla RDMA Area, zona di memoria finita e pertanto da gestire quale preziosa risorsa di sistema.

È fornita una breve descrizione del significato delle funzioni utilizzate dai sopracitati meccanismi.

Tra le variabili riferite al processo cui fa capo la RDMA Area troviamo:

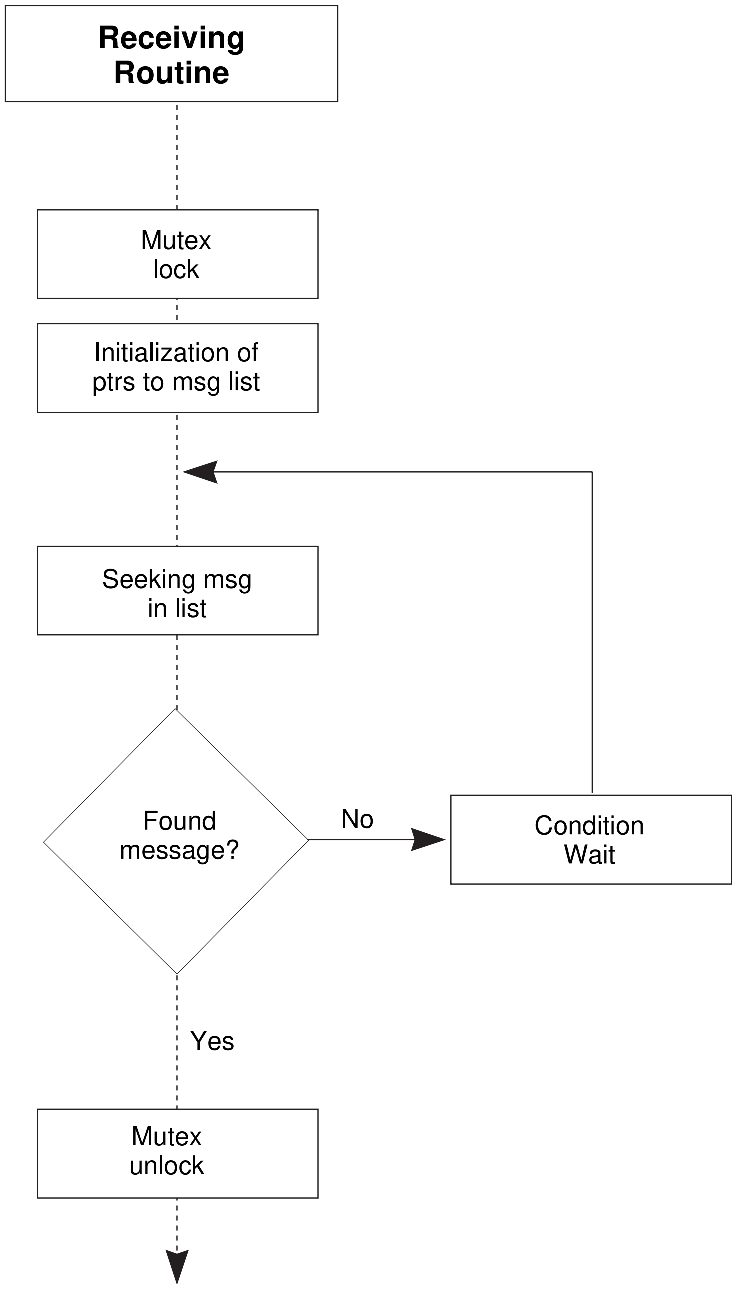

Figura 5.4: Routine di gestione bloccante della ricezione dei messaggi.

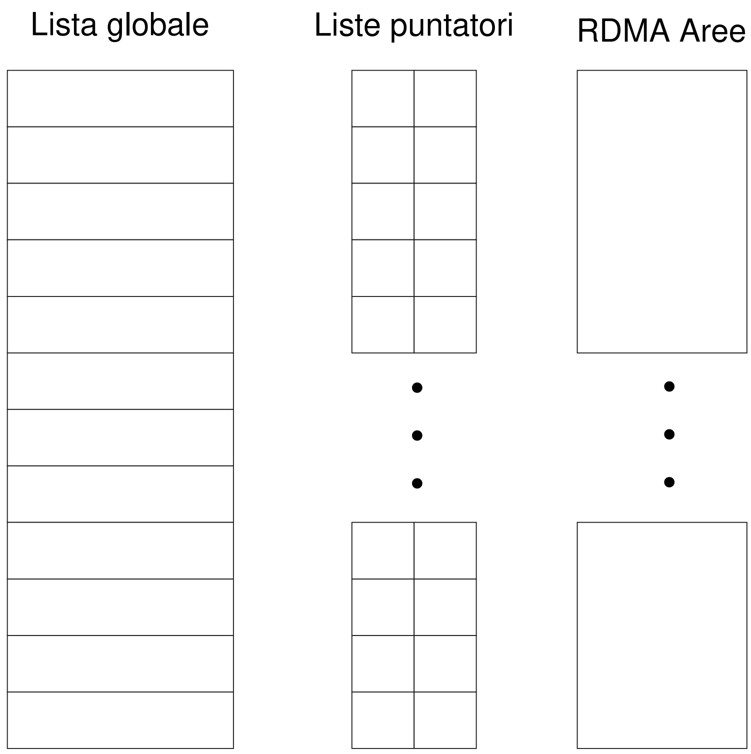

Figura 5.5: Schema della costituzione delle liste concatenate.

- `RDMAArea`: utilizzata per la lettura dei messaggi ricevuti, punta alla zona di memoria registrata per ricevere i dati dal processo remoto.
- `RDMAnextpointer`: punta al byte di inizio del prossimo messaggio.
- `RDMAreadpointer`: punta alla prima zona contigua di memoria libera per la scrittura. Prima di tale posizione tutti i messaggi eventualmente presenti sono stati letti.

Le variabili invece riferite al processo connesso sono:

- `RDMAArea`: punta alla zona di memoria in cui scrivere i messaggi.
- `RDMAwritepointer`: punta al primo byte libero, in cui avrà inizio il prossimo messaggio.
- `RDMAmaxwrite`: contiene il numero massimo di byte che possono essere scritti nell'area RDMA del processo connesso.
- `RDMAmaxread`: è scritta tramite RDMA per notificare il numero di byte della RDMA locale letti dal processo connesso e quindi riutilizzabili per nuovi invii di messaggi.

5.7 Conclusioni

La libreria LAM, funzionante con M-VIA, rispetta la conformità con lo standard MPI 1.1. Al completamento di tale conformità si è ora aggiunto, grazie al progetto delle otto funzioni RPI, il corretto sfruttamento dei cluster con nodi multiprocessore. Nonostante sia stata riscritta parte del lower layer di LAM e siano stati introdotti algoritmi aggiuntivi per la gestione delle primitive asincrone, non si è registrato alcun significativo peggioramento delle prestazioni rispetto alla precedente versione basata sulle Completion Queue. L'obbiettivo fondamentale, raggiunto nei precedenti passi del progetto, si è mantenuto anche in questa importante fase: la latenza misurata con applicativi MPI su VIA si è rivelata notevolmente inferiore a quella misurata con MPI su TCP/IP.

CAPITOLO 6

Conclusioni

Il lavoro della presente tesi ha riguardato principalmente due argomenti: il confronto tra due realizzazioni dello standard VIA, architettura per lo scambio di messaggi tra nodi di rete in modalità punto a punto, e lo sviluppo delle primitive necessarie all'integrazione di una delle realizzazioni in LAM, una libreria per il Message Passing.

Prima differenza da evidenziare nelle due realizzazioni esaminate è rappresentata dal costo, parametro importante per la realizzazione di un cluster. Come già si è sottolineato, infatti, è necessario che i componenti del cluster appartengano al mercato di massa, in modo da realizzare la rete mantenendo i costi ben inferiori a quelli da sostenere per un'architettura parallela nativa. Nel caso in esame, M-VIA è liberamente distribuito assieme ai suoi sorgenti per il sistema operativo Linux, senza alcun costo di licenza. Viceversa, le schede cLAN, essendo componenti hardware, hanno ovviamente un costo di realizzazione. Esse sono inoltre fornite ad applicazioni industriali ed hanno, oltre ai necessari driver per Linux, la possibilità di essere supportate anche su piattaforme basate su sistemi operativi Windows NT 4.0 Server versione Enterpirse e su Windows 2000 Server, Advanced Server e Datacenter server. Il costo di questa scheda è, al momento della stesura di queste pagine, superiore al costo di un microprocessore di classe "x86" di frequenza prossima al gigahertz.

Dal punto di vista delle prestazioni, è stato possibile verificare che, in un classico ping-pong con attesa sincrona dei messaggi, la latenza nelle due alternative ha dato valori differenti di pochi microsecondi a favore delle schede hardware, che non sono sufficienti per giustificare la differenza di costo. Una netta superiorità prestazionale della realizzazione hardware si è invece evidenziata nel caso di comunicazioni a polling, sistema solitamente non sfruttato nella computazione parallela, per l'eccessivo uso di risorse computazionali.

Un altro grande svantaggio della realizzazione hardware è la mancanza delle primitive asincrone "Notify", che non ha consentito lo sfruttamento di tali schede per l'integrazione dello standard VIA sulla libreria LAM/MPI in modo da permettere l'efficace sfruttamento di architetture multiprocessore.

La libreria di programmazione LAM, per la sua costruzione in due livelli, ha permesso una intuitiva integrazione del protocollo M-VIA, grazie alla definizione di otto primitive che gestiscono l'intero processo di comunicazione tra l'ambiente di programmazione e il sottostante livello hardware.

È stato così possibile ovviare alla discordanza tra le modalità di comunicazione dello standard definito da VIA - che prevede comunicazioni esclusivamente punto a punto - e MPI, che definisce anche protocolli di comunicazione collettiva ed asincrona.

La gestione della comunicazione è stata realizzata basandosi sulla primitiva `VipRecvNotify`, la quale permette di effettuare scambi di messaggi tra processi in modo asincrono. La funzione richiama infatti un handler che resta in stato "idle" fino al ricevimento di un messaggio sulla coda di ricezione e, a tale evento, lo inserisce in una lista che contiene ogni messaggio giunto al nodo in questione. Successivamente, il processo principale scandisce tale lista alla ricerca del proprio messaggio senza polling attivi sulle code di ricezione. In tale modo si è potuta proporre una soluzione alla gestione delle comunicazioni "any source", cioè con attesa del messaggio con possibilità di provenienza da un processo qualsiasi dei nodi del cluster, in particolare da un processo operante sullo stesso nodo multiprocessore. Questo meccanismo ha sostituito il precedente, basato sulla funzione delle Completion Queue (CQ), code che raccolgono le notifiche di completamento di ricezione e spedizione dei messaggi sulle code principali del processo. Tali CQ sono legate all'interfaccia fisica di rete (*Network Interface Card*, NIC) e pertanto, in macchine multiprocessore, in cui processori sullo stesso nodo e processori in remoto fanno capo a NIC diversi, non è possibile utilizzare una sola Completion Queue. Nel caso di attesa di un messaggio da un mittente qualsiasi, sarebbe necessario realizzare un algoritmo che ripetutamente verifichi lo stato delle due Completion Queue, con grave dispendio di risorse di calcolo.

La spedizione dei messaggi avviene sfruttando trasmissioni di tipo *Remote Direct Memory Access* (RDMA), mediante i quali il processo mittente invia entrambi gli indirizzi di memoria coinvolti: quello che contiene il messaggio da trasmettere e quello che dovrà riceverlo sul nodo destinatario. Esiste, nelle specifiche di VIA, la possibilità di leggere gli indirizzi di memoria del nodo remoto, che permetterebbe un notevole snellimento del protocollo di comunicazione, ma non è stato realizzato in alcuna delle due versioni del protocollo.

Nonostante l'intervento di questa tesi abbia reso la libreria compatibile alle specifiche dello standard MPI 1.1, il progetto non e' ancora terminato e numerosi sono i punti di intervento per il lavoro futuro.

In particolare, occorre rendere conforme la coda dei messaggi allo standard delle strutture dati MPI, permettendone l'accesso in lettura a strutture esterne. È infatti in corso di integrazione il supporto al debugger "Total View" per programmi in standard MPI, il quale necessita di poter accedere al numero di messaggi in coda nei diversi processi.

Inoltre, si può completare la gestione degli errori delle varie primitive di VIA: in caso di errore, la primitiva in causa deve restituire la costante `LAMERROR` all'applicazione.

È poi possibile progettare il supporto alla versione 2.0 di MPI ed in particolare alla primitiva `MPI_Spawn`, che gestisce in modo dinamico i processi, permettendone l'aggiunta o l'eliminazione in qualsiasi momento dell'esecuzione dell'applicativo.

Ancora, può essere realizzato il supporto alle comunicazioni persistenti, con le quali si effettuano ripetute transazioni di messaggi con minimi cambiamenti ad essi, limitati a solo qualche campo dell'envelope. Tale supporto può prevedere un meccanismo di spedizione dei soli campi aggiornati dell'envelope del messaggio.

Infine è possibile migliorare la gestione delle comunicazioni in caso di operazioni "veloci" (fast send e fast receive), in cui l'intera comunicazione si esaurisce con lo scambio di un solo messaggio. In questo caso si può realizzare un algoritmo che scavalchi le liste, non strettamente necessarie per scambi atomici di dati.

Bibliografia

[1] Franco Boselli. Integrazione della libreria LAM-MPI su VIA. Master's thesis, Università degli Studi di Parma, Apr 1999.

[2] Giovanni Chiola and Giuseppe Ciaccio. Lightweight Messaging Systems. *High Performance Cluster Computing vol. 1*, 1999.

[3] Compaq, Intel, and Microsoft Corporations. *Virtual Interface Architecture Specification version 1.0*, dec 1997. `http://www.viarch.org`.

[4] Free Software Foundation. Gnu licence. `http://www.gnu.org/copyleft/fdl.html`.

[5] Debian Inc. Debian linux distribution. `http://www.debian.org`.

[6] Emulex Inc. clan per windows nt e linux. `http://www.emulex.com/products/`.

[7] RedHat Inc. Redhat linux distribution. `http://www.redhat.org`.

[8] Richard P. Martin, Amin M. Vadhat, David E. Culler, and Thomas E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture, 1998.

[9] MPI-Forum. Message Passing Interface Forum. `http://www.mpi-forum.org`.

[10] National Energy Research Scientific Computer Center (NERSC). M-VIA: A high performance Modular VIA for Linux. `http://www.nersc.gov/research/FTG/via/`.

[11] Marco Panella. Realizzazione delle primitive di comunicazione su VIA per una libreria a scambio di messaggi MPI. Master's thesis, Università degli Studi di Parma, Apr 2000.

[12] Daniel Ridge, Donald Becker, and Phillip Merkey. Beowulf: Harnessing the power of parallelism in a Pile-of-PCs, 1996.

[13] Thorsten Von Eichen and Werner Vogels. Evolution of the Virtual Interface Architecture. *IEEE Micro*, nov 1998.