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

REALIZZAZIONE DELLE PRIMITIVE DI COMUNICAZIONE SU VIRTUAL INTERFACE ARCHITECTURE (VIA) PER UNA LIBRERIA A SCAMBIO DI MESSAGGI MPI.

Relatore:
Chiar.mo Prof. Ing. GIANNI CONTE

Correlatore:
Dott. Ing. MONICA REGGIANI

Tesi di laurea di:
MARCO PANELLA

ANNO ACCADEMICO 1998–1999

Alla mia famiglia,
cui va tutto il mio affetto.

Desidero rivolgere un sentito ringraziamento al Prof. Ing. Gianni Conte, di cui ho imparato ad apprezzare le doti umane e scientifiche, per gli insegnamenti preziosi e la disponibilità dimostrata.

Ringrazio l'Ing. Franco Boselli per l'aiuto e le spiegazioni fornite all'inizio di questo lavoro di tesi.

Vorrei ringraziare sia l'Ing. Monica Reggiani che, nonostante il molto lavoro da fare e il poco tempo che aveva a disposizione, mi ha fornito consigli preziosi sia l'Ing. Massimo Bertozzi per la pazienza con cui ha risposto alle mie frequenti domande.

Ringrazio i colleghi Andrea Gherardi e Roberto Benatti che hanno diviso con me il piccolo laboratorio di ParMa^2 rendendo meno noiose le ore di duro lavoro.

Un pensiero va a tutti gli amici, cui vado bene così come sono.

Infine, last but not least, il mio ringraziamento più grande va alla mia famiglia che mi ha aiutato a raggiungere questo traguardo.

Mia madre e i miei nonni avrebbero voluto vedere questo giorno; spero che ovunque essi siano, sia loro possibile.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Cluster di PC

La necessità di risolvere problemi scientifici sempre più complessi, nel minor tempo possibile, ha comportato un aumento dell'uso di macchine parallele [1]. Tali architetture, costituite da più unità di elaborazione che collaborano tra loro, consentono di fornire agli utenti una potenza computazionale più elevata.

Grazie all'incremento della potenza di calcolo dei PC, sul mercato dei supercomputer paralleli hanno fatto ingresso i cluster di PC, macchine ad alte prestazioni e a basso costo, costituiti da nodi indipendenti interconnessi con una rete.

Volendo dare una definizione di cluster di PC si potrebbe dire che esso è *un insieme di elaboratori autonomi uniti da una rete, su cui gira un software appositamente progettato in modo da produrre un ambiente integrato* [4].

I tre elementi di cui un cluster si compone, già ricordati nella definizione, sono: i nodi, la rete e il protocollo di comunicazione tra i nodi.

Il singolo nodo è costituito da un completo e autonomo Personal Computer con capacità di calcolo, possibilità di svolgimento di operazioni di I/O e interfaccia verso qualche tipologia di rete. I componenti hardware di un nodo non sono affatto differenti rispetto a quelli contenuti nei Personal Computer utilizzati da milioni di utenti nel mondo. Questo consente di assemblare un cluster di PC mantenendo bassi i costi di acquisto e di manutenzione grazie all'economia di scala.

L'interconnessione tra i nodi rappresenta l'elemento più critico, poiché costituisce il collo di bottiglia di tutto il sistema. Sono ormai disponibili sul mercato, a costi accessibili, tecnologie in grado di fornire una banda passante attorno al gigabit al secondo; sfortunatamente gli attuali protocolli di comunicazione non riescono a sfruttare pienamente le risorse messe a disposizione dai canali fisici.

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

- rapporto prestazioni-prezzo superiore alle architetture parallele commerciali, grazie soprattutto all'abbattimento dei costi;
- l'uso di componenti ampiamente disponibili sul mercato, e quindi a basso costo;
- interconnesione tra i nodi costituita da una rete SAN
- possibilità di uso di un sistema operativo e applicativi gratuiti,
- capacità di aggiornare facilmente l'architettura seguendo gli sviluppi tecnologici,
- limitati costi di manutenzione hardware e software.

Si presenta ora un modello per la caratterizzazione di un cluster di PC, in cui si possono distinguere le grandezze specifiche che è necessario tenere presente nella scelta di ognuno degli elementi di un cluster:

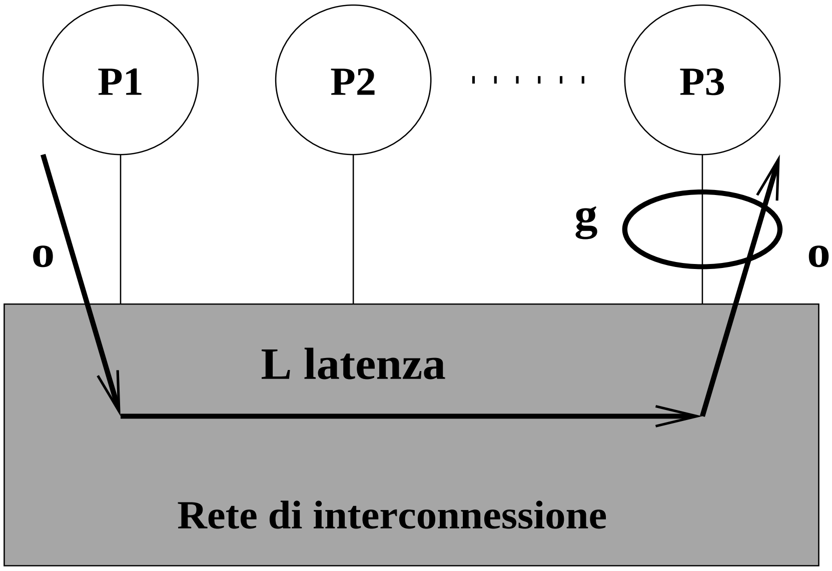

Figura 1.1: Modello LogP.

- la potenza computazionale per i nodi;
- la banda passante per la rete di interconnessione;
- la latenza per il protocollo di comunicazione.

1.2 Il modello LogP

Il modello LogP [9] (figura 1.1) definisce il tempo di trasmissione attraverso quattro parametri:

Latenza L ritardo che intercorre tra la trasmissione e la ricezione di un messaggio di dimensione tendente a zero.

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

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

Processori P numero di processori presenti nel cluster.

Seguendo questo modello il tempo di trasmissione per un messaggio di N byte è:

$$T = L + 2 \times o + g \times N$$

Uno dei pregi di questo modello è la distinzione tra la latenza dovuta al ritardo introdotto dalla trasmissione dei dati sulla rete, causata dalla scheda di rete, dalla interconessione e dal protocollo di rete, e dall'overhead causato dalle operazioni che il processore deve svolgere in conseguenza alle azioni richieste dal protocollo di comunicazione.

Le misure separate della latenza e dell'overhead risultano piuttosto complicate, per cui solitamente, si determina e si esprime solo un tipo di tempo, che indicheremo impropriamente latenza, comprensivo della latenza come definita dal modello LogP e di due volte l'overhead.

1.3 Il cluster ParMa^2

Il progetto ParMa^2 (PARallel MAchine 2) nacque nel 1995, quando vennero acquistati quattro PC con processore Intel Pentium 100, in seguito affiancati da due Pentium Pro a 200 Mhz.

L'architettura attuale è il risultato dell'evoluzione del primo cluster sperimentale. I primi quattro nodi, ora obsoleti, sono stati sostituiti, nell'ottobre del 1998, da quattro nuovi nodi basati su Pentium II Dual Processor a 450 MHz.

L'obiettivo è quello di creare un sistema parallelo ottenendo un rapporto prestazioni-prezzo superiore a quello delle macchine parallele dedicate, utilizzando solo componenti off-the-shelf.

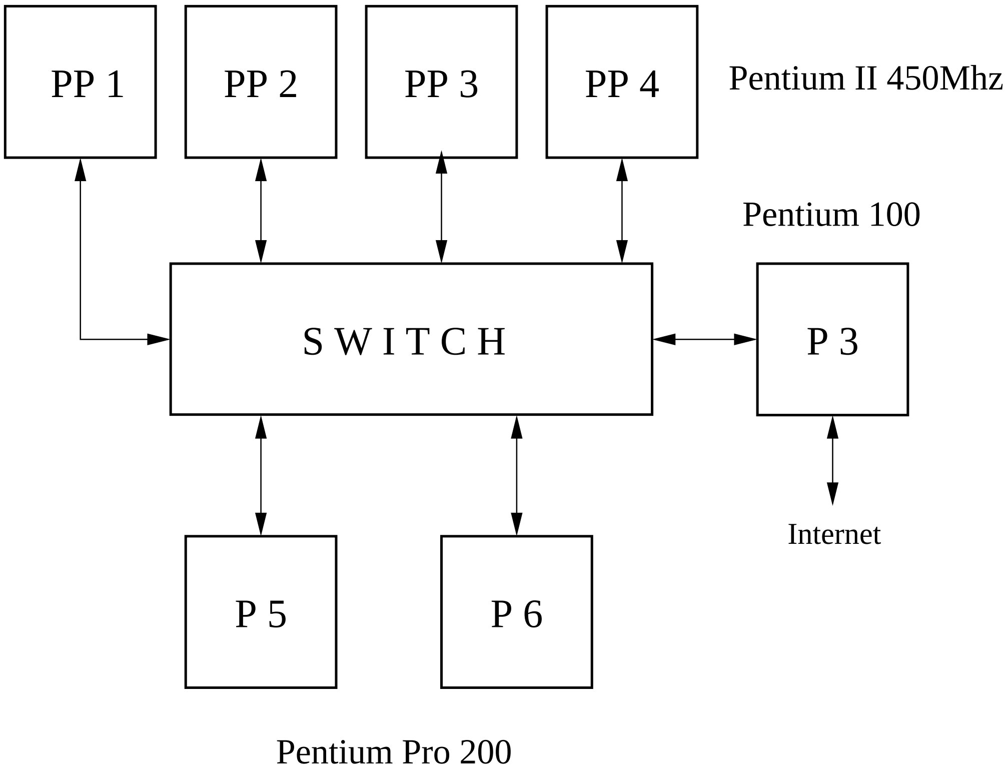

Figura 1.2: Il cluster ParMa^2.

Il sistema operativo installato sui sei nodi rimasti è Linux, distribuzione Debian 2.1, con una dotazione standard di applicazioni. Per la programmazione parallela sono disponibili due diverse realizzazioni di librerie a scambio di messaggi: MPICH 1.1.2 e LAM 6.3.1.

1.3.1 Hardware

I componenti hardware dei quattro nodi più recenti sono i seguenti:

- doppio processore Pentium II a 450 MHz con cache L1 di 32 Kbyte e L2 di 512 Kbyte all'interno del contenitore della CPU;
- motherboard ASUS P2B-DS con bus a 100 MHz;
- memoria RAM di 256 Mbyte in 2 DIMM PC100;

- scheda di rete D-LINK DFE-500TX basata su chipset DEC Tulip 21140;
- scheda video S3 Virge GX2 AGP con 4 Mbyte di RAM;
- architettura SCSI con controller integrato sulla motherboard di tipo Ultra2 Wide SCSI realizzato con il chip Adaptec AIC 7890;
- hard disk Ultra Wide SCSI da 4.3 Gbyte.

Le macchine sono connesse a stella tramite uno switch Cisco Fast Ethernet del tipo Micro Switch 1548.

Uno dei componenti del primo cluster, una macchina Pentium 100, funge da router e assicura il collegamento verso la rete esterna.

1.3.2 Software

Tutti i nodi utilizzano Linux, un sistema operativo GNU [10] appartenente alla famiglia di sistemi operativi UNIX-type, disponibile per diverse architetture tra cui Intel x86, Sparc e Alpha.

Le ragioni della scelta di Linux sono molteplici:

- è un sistema operativo completo, affidabile e robusto;
- richiede solamente una licenza di tipo GNU; è quindi un prodotto software gratuito, ridistribuibile e modificabile;
- il numero di applicazioni disponibili per tale sistema operativo è molto elevato e la maggior parte sono gratuite;
- lo sviluppo del kernel di Linux e delle applicazioni è condotta da una vasta comunità di ricercatori, che producono senza sosta nuove versioni per migliorare le prestazioni o risolvere problemi e adattare il sistema operativo ai nuovi componenti;
- è conforme allo standard POSIX [17];

- possiede una efficiente realizzazione del protocollo TCP/IP;
- i device driver offrono un supporto esteso a un gran numero di periferiche.

L'insieme del sistema operativo, dei tools di utilità e delle applicazioni costituisce una *distribuzione*. È stata scelta la distribuzione Debian 2.1 in quanto offre una utility di gestione dei pacchetti applicativi particolarmente flessibile ed efficiente e ne include solitamente le versioni più recenti.

L'attuale versione del kernel utilizzato è la 2.2.13.

1.4 Sommario

Questa tesi si propone di proseguire la progettazione e realizzazione di una possibile implementazione della libreria LAM-MPI su VIA. Scopo finale è di analizzare ed estendere il protocollo [2] affinché comprenda tutte le primitive di comunicazione contemplate nello standard di MPI. In particolare è necessario procedere all'implementazione delle primitive point to point, poiché la libreria LAM prevede che le funzioni collettive siano espanse in una opportuna serie di comunicazioni punto a punto. Per descrivere il lavoro eseguito si procederà per gradi salendo idealmente nel cluster dalla rete di interconnessione al livello utente.

Una breve analisi della *Virtual Interface Architecture* (VIA) [6], un protocollo a bassa latenza proposto da Compaq, Intel e Microsoft, è contenuta nel **capitolo 2**. Si presentano le componenti di VIA, i modelli di connessione, di trasferimento dei dati e i livelli di affidabilità previsti da VIA.

Poiché al momento attuale, le schede di rete che supportano VIA e di cui siano disponibili i driver per Linux, sono molto costose e poco diffuse, si è deciso di utilizzare una emulazione software per il sistema operativo Linux, chiamata M-VIA. Nello stesso capitolo sono quindi descritte le caratteristiche principali di M-VIA; da queste e dai risultati sperimentali sull'utilizzo e sulle prestazioni se ne deducono i pregi e i difetti.

Di pari passo con lo sviluppo di architetture a cluster di PC, è stato sentito il bisogno di formalizzare i modelli di programmazione utilizzati. Nel 1993 un organismo internazionale è stato appositamente creato al fine di redigere un documento di specifica per il Message Passing. Il **capitolo 3** contiene un'analisi delle specifiche delle funzioni di comunicazione della Message Passing Interface, oltre a una breve panoramica su questo standard in generale.

Il **capitolo 4** descrive brevemente le caratteristiche più importanti della libreria LAM, una particolare implementazione aderente allo standard MPI. LAM è una libreria suddivisa in due livelli, di cui uno portabile tra varie architetture e protocolli di comunicazione, l'altro che è modificabile in base al sistema utilizzato. I due livelli comunicano tramite un'interfaccia chiamata *Request Progression Interface* (RPI). In questo modo per introdurre un nuovo protocollo di trasporto al di sotto del livello MPI non è necessario riscrivere tutto il codice di LAM ma solo le funzioni di interfaccia e quelle del livello più basso.

Il **capitolo 5** si apre con una breve descrizione delle caratteristiche necessarie del protocollo e delle motivazioni delle scelte effettuate. Prosegue con la descrizione delle funzioni dell'intrerfaccia RPI e delle strutture dati necessarie al fine di permettere all'utente di sfruttare pienamente VIA come livello di trasporto all'interno della libreria LAM. Al termine del capitolo, dopo un esplicativo esempio di funzionamento, si presenta una breve descrizione del controllo di flusso necessario ad evitare l'esaurimento delle risorse.

Nel **capitolo 6** si riportano i dati sperimentali ottenuti, al fine di dimostrare che l'uso di VIA in un cluster di PC aumenta le prestazioni rispetto all'uso di TCP. Le prestazioni sono misurate con un semplice programma di ping-pong tra due processi. Inoltre per dimostrare la correttezza del lavoro svolto si presentano i risultati dell'esecuzione di un pacchetto software, i NAS Benchmark, ormai largamente accettato per la misura di sistemi paralleli.

Il **capitolo 7** riporta le conclusioni di questo lavoro e specifica alcune linee guida per eventuali sviluppi futuri.

Capitolo 2

Virtual Interface Architecture (VIA)

TCP/IP è indubbiamente il protocollo più diffuso per reti generiche. Alcune delle sue caratteristiche però rendono inadatto il suo uso quale protocollo di comunicazione per cluster.

I fattori limitanti per le prestazioni del protocollo TCP/IP in un cluster che si possono individuare sono:

- il numero di livelli software che devono essere attraversati per effettuare una comunicazione, a causa delle copie dei dati che sono effettuate passando da un livello al successivo
- gli interrupt che devono essere obbligatoriamente generati in alcune fasi della comunicazione
- i context switch che avvengono ogni volta che il sistema operativo debba intervenire nella comunicazione

Per ridurre l'overhead di una comunicazione la ricerca ha proposto alcune alternative al TCP, quali U-Net, Active Message (AM), Fast Message (FM) Nonostante i buoni risultati ottenuti dal punto di vista delle prestazioni, esse sono rimaste esperienze limitate all'ambito della ricerca; tuttavia VIA ha ereditato alcuni dei loro concetti fondamentali.

VIA è un protocollo nato sotto la spinta di un consorzio composto da Intel, Compaq e Microsoft; esso fornisce l'illusione, ad ogni processo utente, di avere accesso diretto all'interfaccia di rete, senza bisogno di alcun intervento da parte del sistema operativo; vengono quindi eliminati interrupt e context switch.

Le specifiche di VIA, in particolare, si basano sui seguenti concetti:

- ogni processo deve eseguire un piccolo numero di operazioni per cominciare, gestire e terminare la comunicazione;
- le operazioni di I/O non devono richiedere l'utilizzo di interrupt;
- context switch e copie dei dati in memoria devono essere evitati, quando possibile;
- l'uso concorrente della rete e della interfaccia di rete deve essere assicurato senza l'intervento del sistema operativo.

2.1 Il modello architetturale di VIA

La Virtual Interface Architecture (figura 2.1) si compone di quattro elementi essenziali:

- un *VI Provider*,
- uno o più *VI Consumer*,
- una o più *Virtual Interface* (VI),
- una o più *Completion Queue* (CQ).

Essi interagiscono al fine di assicurare la comunicazione tra più processi.

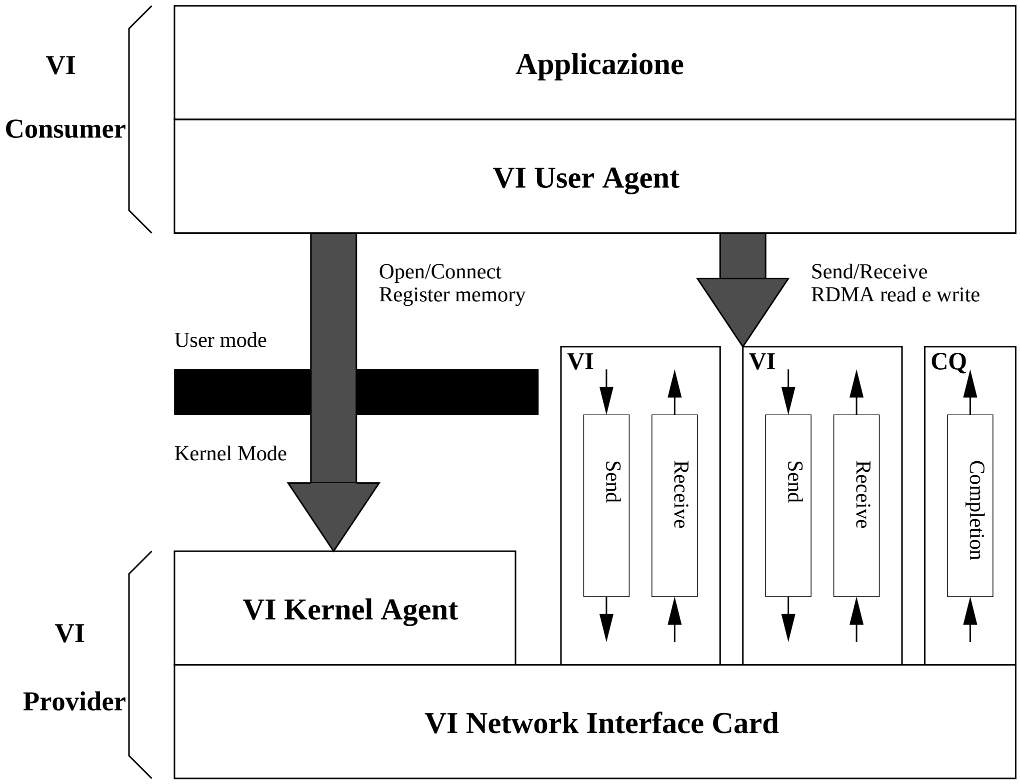

Figura 2.1: Modello architetturale di VIA.

2.1.1 VI Provider

È costituito dal *Network Interface Controller* (NIC) e dal *Kernel Agent*, e può essere definito come l'insieme dei componenti hardware e software che consentono la creazione di tutte le strutture necessarie alla comunicazione.

Durante la fase di inizializzazione, il VI NIC si occupa di creare le Virtual Interface e le Completion Queue. Una volta connessi i processi, quando uno di essi richiede uno scambio di dati, il VI NIC provvede al trasferimento dei dati operando in modo asincrono rispetto al resto dei componenti di VIA, cioè senza il loro ulteriore intervento.

Il Kernel Agent è una parte del sistema operativo che fornisce tutte le funzioni necessarie alla preparazione dello scambio dei dati, ma che non interviene nella comunicazione, a meno che il VI Consumer non richieda la registrazione di una zona di memoria. Esso è invocato al momento della creazione, connessione e distruzione delle Virtual Interface e nella gestione della memoria usata dal NIC o dal VI Consumer.

2.1.2 VI Consumer

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

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

2.1.3 Virtual Interface

Una Virtual Interface è costituita dall'insieme delle strutture dati che consentono ad un VI Consumer di accedere al VI NIC per eseguire i trasferimenti dei dati.

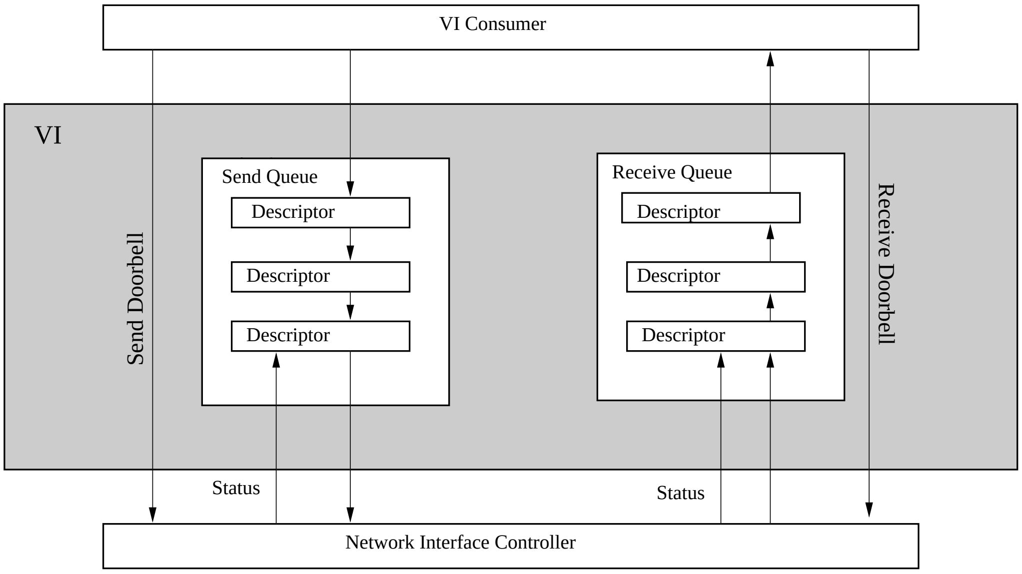

Figura 2.2: Virtual Interface.

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

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

Da questo momento è il VI NIC che si occupa di gestire la comunicazione, modificando opportunamente le informazioni di stato. Questo avviene senza alcun intervento del kernel del sistema operativo, riducendo quindi l'overhead

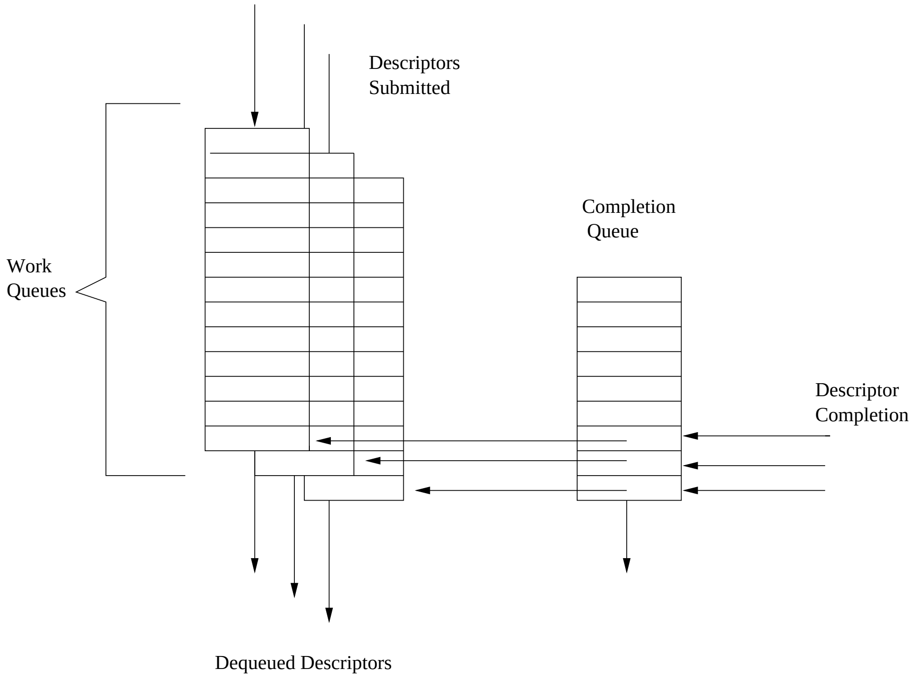

Figura 2.3: Modello di Completion Queue.

che contribuirebbe all'aumento del tempo di trasmissione.

Al VI Consumer è lasciata la possibilità di controllare lo stato del descrittore postato e di toglierlo dalla coda, una volta che l'operazione di comunicazione sia stata terminata, con successo o meno.

Il VI Consumer ha a disposizione due diverse semantiche per rilevare il completamento di un descrittore: *polling* e *operazioni bloccanti*.

Nel primo caso il processo esamina lo stato del descrittore in cima alla coda: nel caso sia stata completata la trasmissione o la ricezione del messaggio il descrittore è tolto dalla coda. Quando il VI Consumer utilizza la modalità bloccante, invece, il processo si ferma in attesa che la comunicazione associata al descrittore venga completata.

2.1.4 Completion Queue

Le Completion Queue consentono ad un processo di unire in un'unica coda le notifiche di completamento delle operazioni di comunicazione su diverse VI. Quando un descrittore postato su una coda associata ad una CQ è completato, il VI NIC accoda ad essa un puntatore alla VI su cui è avvenuta la comunicazione.

Ad una stessa CQ possono essere associate sia le code di ricezione che quelle di spedizione, oppure è possibile associarne solo un tipo; è possibile anche avere due CQ diverse, una per le Send Queue, l'altra per le Receive Queue.

L'associazione con la Completion Queue è eseguita al momento della creazione della Virtual Interface ed è poi impossibile modificare questa scelta.

Una volta associata una Completion Queue ad una coda non è più possibile verificare il completamento di un descrittore sulle singole code, ma è necessario farlo sulla CQ.

2.2 Gestione della memoria

Per evitare incongruenze tra i dati che si vogliono spedire e quelli effettivamente spediti, VIA blocca in memoria le pagine contenenti i dati da trasmettere (operazione detta *registrazione*).

La registrazione è richiesta da un VI Consumer al VI Provider, il quale riserva per il buffer da registrare una o più pagine fisiche contigue, indicate come una regione di memoria. Il Provider, inoltre, blocca le pagine in memoria fisica e inserisce un entry in una Page Table, gestita dal Kernel Agent, che contiene la traduzione tra indirizzo virtuale e fisico, assieme alla informazioni di protezione.

Quando una regione di memoria è registrata, ogni pagina di tale regione è bloccata in memoria; questo garantisce al VI NIC che le informazioni da spedire siano sempre disponibili allo stesso indirizzo.

Al VI Consumer è restituito un handle per la memoria registrata; esso consente al VI Provider di risalire all'indirizzo fisico del dato senza alcuna necessità di intervento del kernel.

Un VI Consumer deve deregistrare quelle regioni di memoria non più utilizzate in modo che il Kernel Agent possa rimuovere l'entry corrispondente nella Page Table e possa sbloccare le relative pagine riservate.

Al fine di ridurre l'overhead, questa operazione, eseguita solo una volta dal Kernel Agent per ogni buffer, dovrebbe avvenire al di fuori delle operazioni di trasmissione dei dati.

Dal momento che ogni processo può disporre di numerose Virtual Interface è bene disporre di un meccanismo per la protezione delle regioni di memoria registrate; a questo scopo si utilizzano i tag di protezione della memoria (*protection tag*) e gli attributi di protezione della memoria.

I protection tag sono degli identificativi unici, costituiti da un numero intero, che vengono associati sia a una VI che alle regioni di memoria.

Il VI Consumer richiede al VI Provider l'assegnazione di tali tag quando la memoria è registrata o quando la VI è creata. Un qualunque accesso a un dato è consentito dal VI NIC solo se il tag di protezione della memoria in cui il dato è registrato e il tag della VI che vuole accedere a tale dato coincidono. La convalida dei tag di protezione viene applicata sia alle regioni di memoria che contengono dati sia a quelle che contengono descrittori.

Gli attributi di protezione sono invece utilizzati per controllare il tipo di operazioni che possono essere svolte da una VI su una data regione di memoria. Se gli attributi coincidono l'operazione viene svolta, altrimenti l'attributo utilizzato è quello che garantisce la protezione più stringente; questo significa che l'operazione potrebbe non essere eseguita. Vengono utilizzati anche in fase di connessione: solo due Virtual Interface con gli stessi attributi di protezione della memoria possono essere connesse con successo.

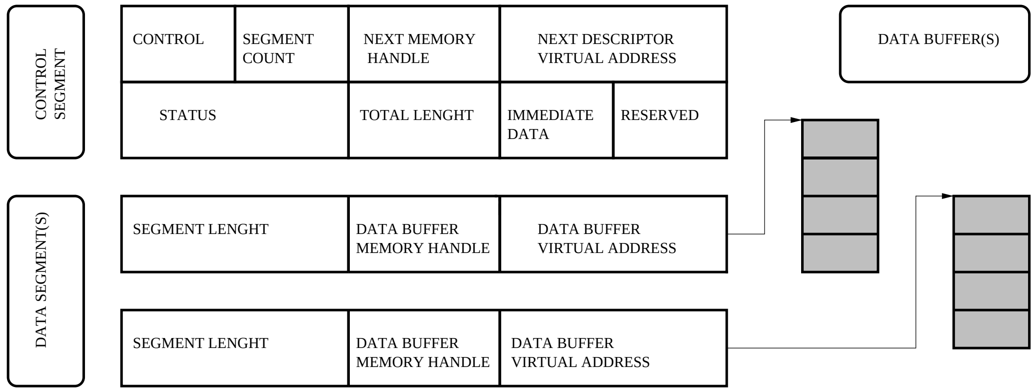

Figura 2.4: Campi di un descrittore.

2.3 Il descrittore

Lo strumento per mezzo del quale ogni comunicazione in VIA viene effettuata è il descrittore. Durante le varie fasi della comunicazione la proprietà del descrittore passa dalla VI Application alla VI Library (VIPL) e quindi al VI NIC, senza transitare attraverso il Kernel Agent, per poi compiere il percorso inverso.

Viene ora descritto il cammino che un descrittore compie all'interno dei componenti di VIA, per comprendere in quale modo VIA elimini il kernel del sistema operativo dal percorso critico delle comunicazioni.

La VI Application, cioè il processo utente che faccia uso di VIA, inizializza alcuni campi della prima parte del descrittore, il *Control Segment*, che contiene le informazioni di stato del descrittore: essa modifica il campo per il tipo di operazione richiesta (*control*), per il numero complessivo di parti di cui il descrittore si compone (*segment count*), per la dimensione totale in byte dei dati inviati o dei buffer per la ricezione, e per altre informazioni accessorie. Inoltre la VI Application deve specificare gli indirizzi dei dati da spedire o dei buffer destinati a contenere i dati in arrivo: per questa funzione un descrittore contiene anche uno o più *Data Segment* in cui la VI Application indica la dimensione totale dei buffer di dati, il loro indirizzo virtuale e l'handle creato durante la loro registrazione. Il descrittore passa dalla VI Application alla VI Library quando è invocata una delle due funzioni VipPostSend e VipPostReceive che postano un descrittore su di una coda.

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

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

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

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

2.4 Modelli di connessione

Il modello di trasferimento dei dati di VIA è orientato alla connessione; all'atto della sua creazione ogni VI non risulta associata a nessuna altra. Al fine di rendere possibile la comunicazione è necessario, per ogni processo, at-

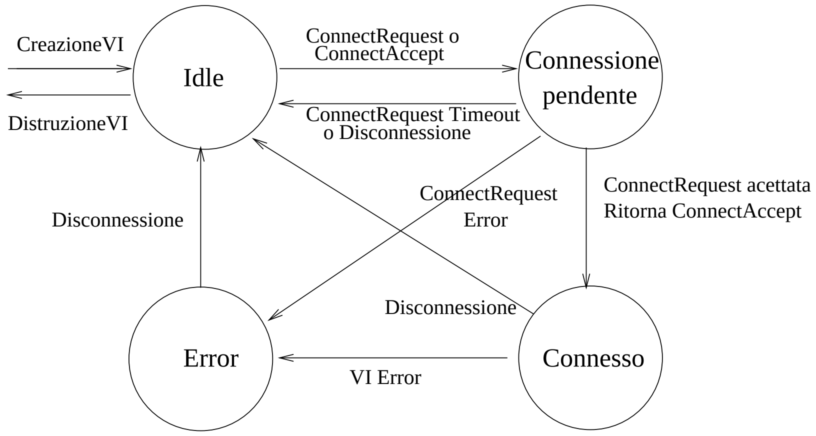

Figura 2.5: I possibili stati di una VI.

tuare un procedimento specifico per connettere una propria Virtual Interface con la VI del processo con cui vuole comunicare.

I modelli di connessione sono due: uno di tipo client-server e uno di tipo peer-to-peer. Essi risultano essere completamente distinti e le applicazioni coinvolte devono utilizzare lo stesso modello su entrambe le estremità della medesima connessione.

2.4.1 Client-server

Il modello *client-server* realizza una semantica esclusivamente bloccante. Entrambi i VI Consumer devono effettuare una richiesta al proprio VI Provider per ottenere la connessione tra una propria Virtual Interface ed una VI remota. È compito dei VI Provider realizzare la connessione tramite protocolli affidabili e robusti.

Il meccanismo di connessione, il cui schema è riportato in fig. 2.6 segue le seguenti fasi: il server si pone in attesa di una richiesta di connessione; quando questa arriva, la analizza e controllo che gli attributi specificati siano accettabili; se questo si verifica crea una VI locale (se non già non esiste una VI idle) e spedisce l'accettazione della richiesta la client; a questo punto le due VI possono essere connesse.

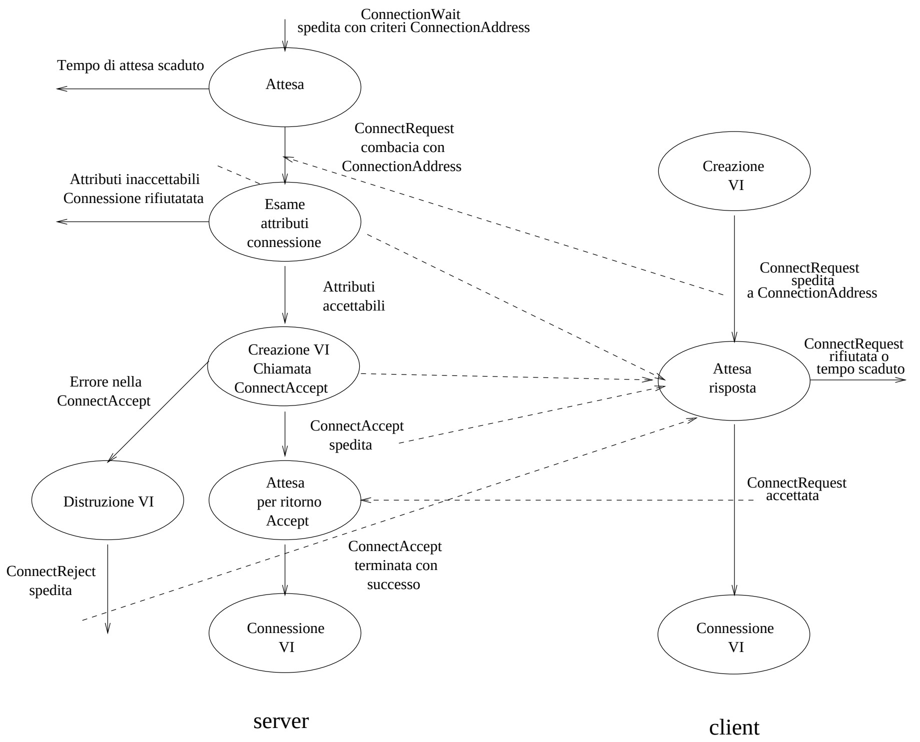

Figura 2.6: Modello client-server.

Il modello è bloccante perché il client deve attendere una risposta dal server alla sua richiesta può uscire dallo stato di attesa solo quando tale richiesta risulta soddisfatta o respinta o quando è scaduto il tempo stabilito per l'attesa della risposta.

Il VI Consumer del processo server effettua una richiesta di attesa della connessione al suo VI Provider. Questa richiesta contiene un campo che permette di individuare quali richieste possono essere accettate e quali no, in base all'indirizzo di rete del client che effettuerà la richiesta

Mentre il VI Consumer dal lato server attende una richiesta di connessione, il VI Consumer del lato client effettua una richiesta di connessione al proprio VI Provider; essa specifica quale Virtual Interface locale è in attesa di essere connessa e l'indirizzo del VI Consumer remoto.

La richiesta del client può risolversi in due modi:

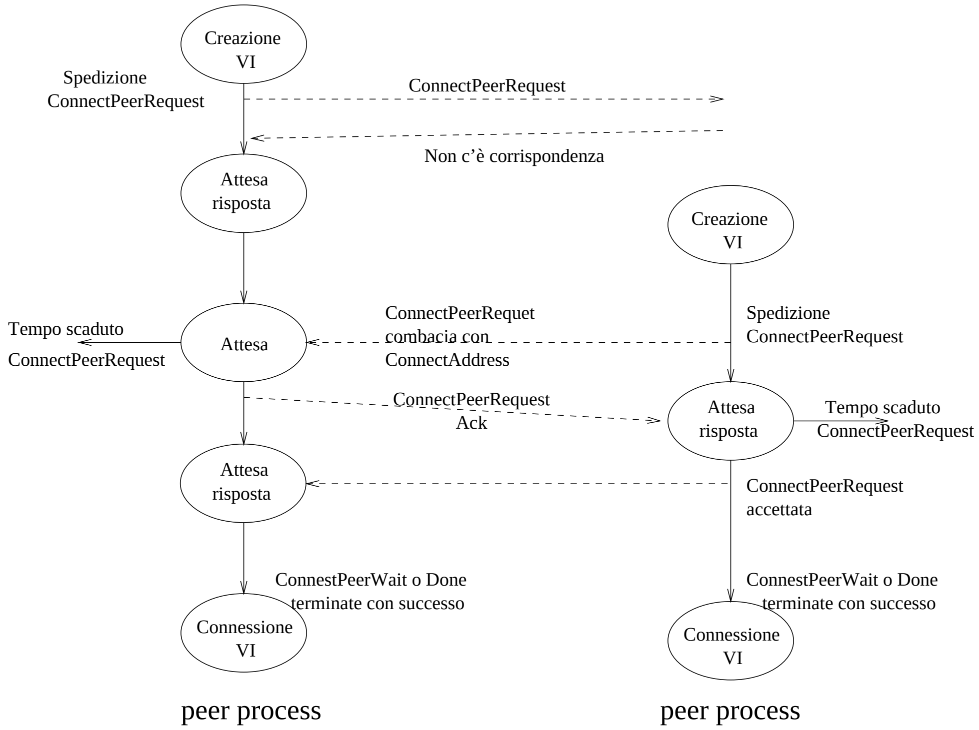

Figura 2.7: Modello peer-to-peer.

- con un errore se la VI remota non esiste, non è raggiungibile, non è in stato idle oppure se gli attributi non combaciano con quelli della richiesta;
- termina correttamente in caso contrario

Il server decide a questo punto se accettare la connessione che gli è stata inoltrata.

2.4.2 Peer to peer

Il modello di connessione *peer-to-peer* realizza una semantica sia di tipo bloccante che di tipo non bloccante.

La differenza con il modello client-server, precedentemente descritto, sta nel fatto che una volta inizializzata la richiesta di connessione, il VI Consumer può decidere se controllare lo stato della richiesta con un test bloccante o non bloccante.

Le fasi seguenti della connessione sono simili, con l'invio di un messaggio di accettazione o ricusazione della richiesta di connessione.

2.5 Modelli di trasferimento dei dati

VIA specifica due modelli di trasferimento dei dati.

Il primo, *Send/Receive*, si basa sui meccanismi solitamente usati da due entità comunicanti per scambiarsi dati: una esegue un'operazione di invio, mentre l'altra riceve in una zona di memoria di propria pertinenza i dati spediti.

Affinché una spedizione venga effettuata con successo, è necessario che sulla coda destinata a ricevere il messaggio sia postato un descrittore vuoto che specifichi un buffer di lunghezza sufficiente a ricevere il messaggio.

Se questo non avviene, la comunicazione si conclude con un errore, denominato *Asynchronous error*; le VI coinvolte entrano in uno stato di errore e non è più possibile alcuna comunicazione.

Il secondo, *Remote Direct Memory Access* (RDMA), è così chiamato perché uno dei due processi specifica sia la zona di memoria contenente i dati da spedire sia la zona di memoria in cui i dati andranno memorizzati dopo la ricezione.

Esso differisce dal precedente anche per quanto riguarda l'aspetto della sincronizzazione tra i processi comunicanti. Il processo che inizi una operazione di RDMA non ha bisogno di sapere se l'altro processo abbia postato alcun descrittore sulla Receive Queue, dato che tale tipo di operazione non ne consuma.

In questo modo, però, non è consentito ricevere notifica dell'avvenuto completamento dell'operazione; è quindi un modello asincrono.

Un aspetto importante delle operazioni di tipo RDMA è la necessità che il processo che effettui questo tipo di comunicazioni sia già informato degli indirizzi di destinazione. Prima di poter eseguire uno scambio di messaggi tramite RDMA i processi devono scambiarsi tramite Send e Receive gli indirizzi virtuali delle regioni di memoria, che dovranno essere registrate, che intendono mettere a disposizione per le comunicazioni.

Esistono due tipi di operazioni RDMA:

Write il Consumer specifica al più un indirizzo remoto, in cui andranno i dati, e una o più regioni di memoria locali, già registrate. In questo modo saranno facilmente eseguibili operazioni di *gather*.

Read il Consumer specifica la sorgente di dati da leggere sul nodo remoto e uno o più buffer di memoria registrata localmente. Saranno agevolate operazioni di *scatter*.

Mentre le operazioni di tipo RDMA Read sono sempre asincrone, quelle RDMA Write possono costituire una importante eccezione: infatti le operazioni di tipo RDMA Write con Immediate Data consumano un descrittore, se già postato, sulla Receive Queue del processo che riceve.

2.6 Livelli di affidabilità

Virtual Interface Architecture supporta tre diversi livelli di affidabilità:

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

2.6.1 Unreliable Delivery

Questo livello di affidabilità garantisce che un'operazione di Send oppure di RDMA Write sia consegnata al più una sola volta all'istanza VI ricevente;

quindi sulla Receive Queue del processo destinatario viene consumato uno o nessun descrittore. Di conseguenza una Send o un RDMA Write possono non giungere mai a destinazione risultando persi.

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

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

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

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

2.6.2 Reliable Delivery

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

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

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

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

2.6.3 Reliable Reception

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

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

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

2.7 M-VIA: una emulazione software di VIA

Attualmente VIA è disponibile in due forme:

- Con schede di rete che supportano il protocollo VIA in hardware
  - Giganet [11], di cui esistono i driver per il sistema operativo Linux
  - Finisar [8]

  - ServerNet II [7]
  - Fujitsu [5]

- Grazie a implementazioni software per schede di rete non dedicate
  - Berkeley che supporta Myrinet [3]
  - MVIA che supporta Fast e Gigabit Ethernet [16]

Al momento della costruzione del cluster ParMa^2, la scelta di M-VIA è stata obbligata in quanto non erano ancora disponibili i driver delle schede dedicate per Linux e la le schede di rete della famiglia Ethernet si avvantaggiavano della provata affidabilità e del basso costo di tale tecnologia.

M-VIA, la cui release più recente è la 1.0, è un pacchetto software che costituisce una emulazione software di VIA per Linux. È prodotta dai ricercatori del *National Energy Research Scientific Center* (NERSC) presso i National Laboratories Lawrence Berkeley.

2.8 Caratteristiche di M-VIA

Le caratteristiche di M-VIA sono:

- Il VI Provider è costituito da moduli caricabili dal sistema operativo, da cui il nome Modular VIA.
- Modularità nel disegno del software che costituisce M-VIA: moduli del kernel, driver delle schede di rete e interfaccia utente sono separati.
- Possibilità di utilizzare sia schede di rete appositamente prodotte per VIA sia schede di rete tradizionali con completa emulazione software.
- Permette l'uso contemporaneo di altri protocolli di rete più tradizionali, quali TCP, UDP e IP.

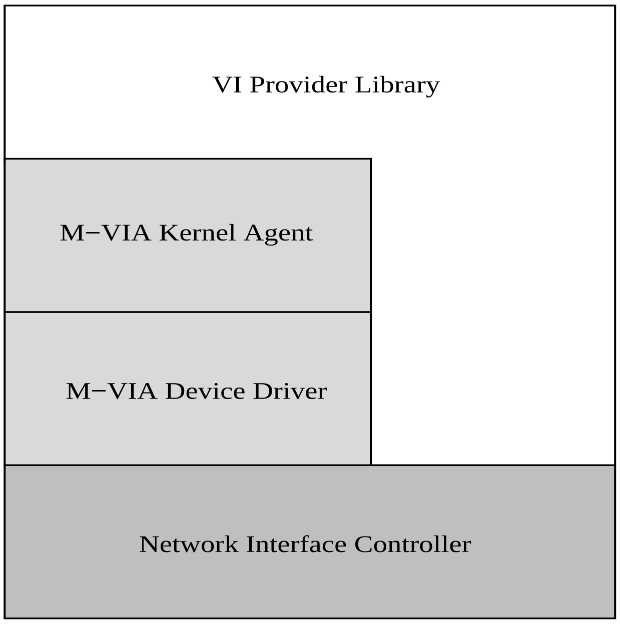

Figura 2.8: Architettura di M-VIA.

- Conformità agli standard VIA, come attesta il superamento dei test di Intel.
- È possibile ottenere e ridistribuire liberamente il codice sorgente, dato che è protetto da una licenza BSD-like [12].

Le scelte realizzative, oltre a facilitare l'estensione del codice esistente ad altre schede di rete, consentono di utilizzare M-VIA senza ricompilare il kernel del sistema operativo.

Il superamento dei test prodotti da Intel dovrebbe garantire una completa portabilità del software prodotto per M-VIA su altre realizzazioni dello standard VIA.

2.9 Prestazioni delle primitive di M-VIA

Le istruzioni dello User Agent di M-VIA possono essere suddivise, in base al compito, in tre categorie:

inizializzazione e distruzione comprende le istruzioni di gestione del NIC o delle VI, il cui tempo d'esecuzione, riportato in tabella 2.1 rappresenta un costo fisso indipendente da fattori quali dimensione dei pacchetti e device utilizzato. Si noti come queste primitive, non intervenendo nella sequenza di comunicazione vera e propria, non siano critiche.

gestione della memoria comprende solo le due operazioni necessarie alla registrazione e deregistrazione della memoria. Il tempo d'esecuzione di queste operazioni, dipendente dalla dimensione del buffer, come mostrato in tabella 2.2 per la registrazione, è più critico dei precedente perché per la spedizione di un dato con VIA esistono due possibilità:

- registrare al momento della spedizione.
- copiare il dato in un buffer, pre-registrato in fase di inizializzazione

Per verificare entrambe queste possibilità è stata testata l'istruzione C memcpy, che esegue copia da una zona di memoria ad un'altra. In particolare la copia è stata fatta da una zona di memoria qualunque ad una zona di memoria registrata dal Kernel Agent, in modo da simulare le due alternative. Si sono confrontati i tempi necessari a registrare i dati al momento della comunicazione e quelli necessari a copiare i dati in buffer preregistrati. Come si nota dai dati riportati, la copia del buffer è sempre più conveniente della registrazione con la release 0.9.3 di M-VIA.

comunicazione comprende le primitive dedicate alla comunicazione il cui tempo di esecuzione dipende sia dalla dimensione del pacchetto trasmesso sia dal device utilizzato. I dati raccolti, riportati in tabella 2.3 per il caso di due processi su due nodi distinti, mostrano che la latenza introdotta da queste primitive rende conveniente l'uso di M-VIA, rispetto al TCP/IP. I dati in locale, riportati in tabella 2.4, dimostrano che il supporto del device di loopback di M-VIA non è ancora otti-

Istruzione	Tempo (μs)
VipOpenNic	465
VipQueryNic	13
VipCloseNic	550
VipNSInit	396
VipNSGetHostByName	8
VipNSGetHostByAddr	12
VipNSShutdown	12
VipCreatePtag	9
VipDestroyPtag	10
VipCreateVi	25
VipDisconnectVI	15
VipDestroyVi	35
VipCreateCQ	73
VipDestroyCQ	23

Tabella 2.1: Prestazioni delle primitive di inizializzazione e distruzione di M-VIA.

mizzato poiché l'overhead dato da questo device per piccoli messaggi è superiore a quello dovuto all'uso della scheda di rete.

2.10 Considerazioni sulla libreria M-VIA

I risultati sperimentali effettuati sul cluster ParMa^2 dimostrano che l'emulazione software di VIA che M-VIA mette a disposizione fornisce effettivamente una riduzione della latenza rispetto al TCP.

Una attenta analisi dell'implementazione di M-VIA mostra alcune mancanze rispetto alle specifiche di VIA; in particolare:

n. Bytes (Byte)	*Copia* (μs)	*Registr.* (μs)	*n. Bytes* (Byte)	*Copia* (μs)	*Registr.* (μs)
3	1	17	7008	63	94
8	2	16	7608	69	62
52	2	17	8208	74	106
112	3	17	9060	82	101
220	4	18	10001	90	110
360	5	17	11041	100	135
500	6	16	12187	110	148
1008	10	17	13452	121	141
1608	16	25	14850	134	170
2208	21	39	16393	148	198
2808	27	31	18096	163	205
3408	32	47	19977	180	231
4008	37	43	22053	199	230
4608	43	47	24346	218	270
5208	48	69	26877	241	310
5808	53	72	29672	265	321
6408	59	77			

Tabella 2.2: Confronto tra i tempi di copia e di registrazione fino a 32kB con M-VIA 0.9.3.

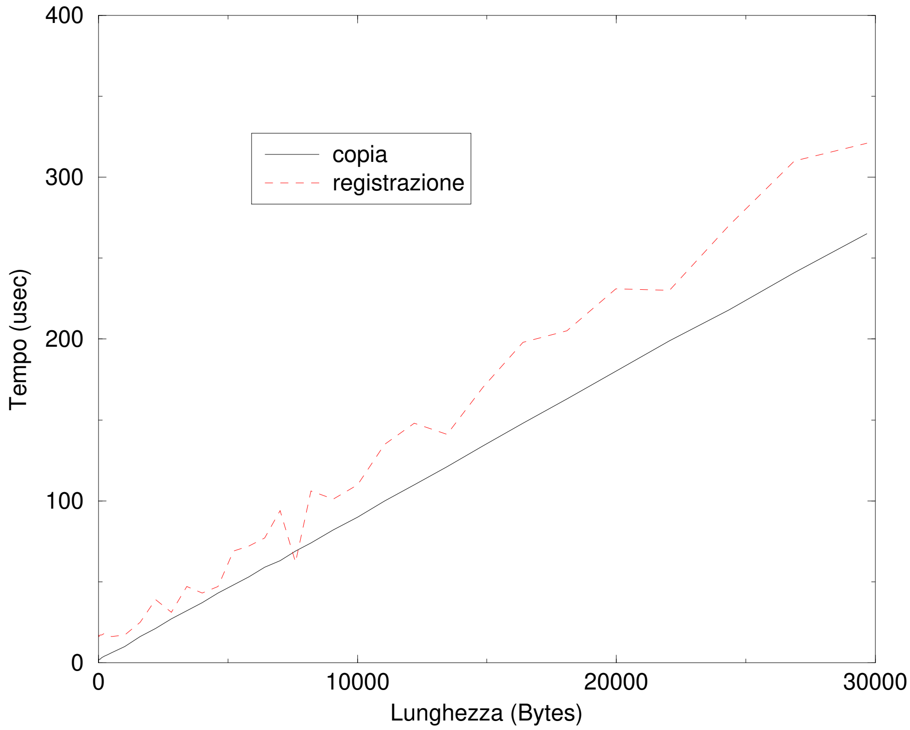

Figura 2.9: Confronto tra i tempi di copia e di registrazione con M-VIA 0.9.3.

Lunghezza (Bytes)	*VIA remoto* (μs)	*TCP remoto* (μs)
1	60	100
2	60	100
4	60	101
8	60	102
16	60	104
32	61	106
64	67	115
128	81	126
256	104	151
512	152	200
1024	246	299
2048	380	433
4096	548	597
8192	894	963
16384	1558	1645
32740	2942	3056

Tabella 2.3: Confronto tra M-VIA e TCP, in remoto.

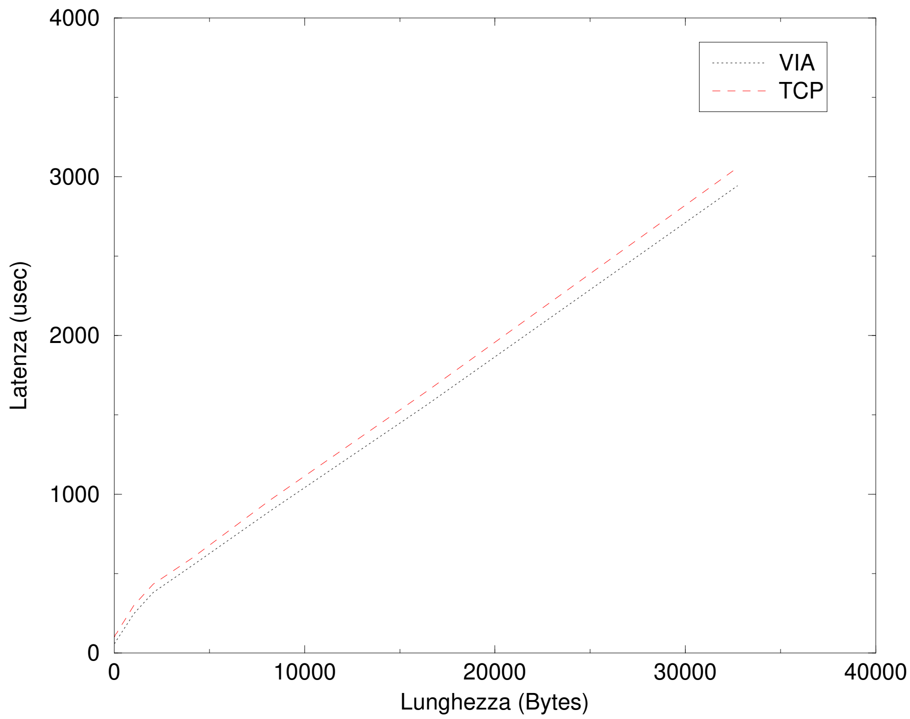

Figura 2.10: Confronto tra M-VIA e TCP in remoto.

Lunghezza (Bytes)	*TCP locale* (μs)	*VIA loopback* (μs)	*VIA rete* (μs)
1	32	5.5	4.0
4	30	5.5	4.5
16	31	6.0	4.5
64	31	6.5	4.5
256	32	6.5	5.5
1024	38	9.0	7.5
4096	72	19.0	21.5
16384	228	78.0	93.5

Tabella 2.4: Confronto in locale tra TCP e M-VIA utilizzando il device di rete e quello di loopback.

- Dei tre livelli di affidabilità previsti da VIA solo l'Unreliable delivery è presente.
- Come conseguenza del punto precedente, non è consentito l'uso delle operazioni RDMA Read, perché esse non sono utilizzabili senza un livello di affidabilità più alto.
- Manca la modalità di connessione peer to peer.
- L'attributo relativo alla qualità di servizio viene ignorato ed è necessario impostarlo a zero per evitare un errore.

Oltre a tali limitazioni M-VIA, essendo ancora in fase di sviluppo, presenta alcuni bug che hanno purtroppo impedito una completa realizzazione e verifica di un prototipo perfettamente funzionante.

La libreria dell'interfaccia utente della versione 0.9.3 contiene bug nella gestione delle Completion Queue e il driver della scheda di rete della release 1.0 impedisce di utilizzare contemporaneamente TCP e VIA.

Per questi motivi, si è stati costretti ad utilizzare la libreria dell'ultima release e il driver della scheda di rete della versione precedente, anche se probabilmente alcuni nuovi errori sono il risultato di questa unione tra programmi di versioni non omogenee.

Infine, a causa di una incompatibilità hardware con il chipset delle schede madri Intel, tutte le versioni di M-VIA mostrano difficoltà al momento della connessione: la percentuale di riuscita è del 10% circa. Per superare questo problema si è dovuto racchiudere la fase di connessione in un ciclo che termina solo a connessione avvenuta.

Purtroppo, in questo modo, la connessione può avvenire solo per una coppia di processi.

Capitolo 3

Message Passing Interface (MPI)

Il modo con cui i nodi di una architettura parallela comunicano tra loro e collaborano per la risoluzione di un problema complesso resta da definire.

Esistono tre possibili famiglie di modelli di programmazione:

Shared Address Space è generalmente utilizzato su architetture di tipo parallelo in cui ogni processo possiede un proprio spazio di indirizzi di memoria virtuale; inoltre, i processi cooperanti condividono una regione di memoria fisica mappata in indirizzi virtuali diversi, attraverso la quale essi comunicano e si scambiano dati.

Message Passing è opposto al precedente, nel senso che ogni processore può accedere solo alla memoria locale; la comunicazione avviene tramite l'utilizzo di primitive specifiche che comportano lo scambio di messaggi.

Data Parallel il parallelismo è ottenuto suddividendo i dati tra i diversi processi che svolgono operazioni simili o identiche su dati diversi.

3.1 Uno standard per lo scambio di messaggi

Il modello che meglio si adatta ad una architettura di tipo cluster come $ParMa^2$ è il Message Passing.

MPI è uno standard per la comunicazione a scambio di messaggi, proposto a partire dal 1992, anno in cui è iniziata la stesura di un documento di specifica. Un apposito organismo internazionale, l'MPI-Forum [14], ha il compito di definire e mantenere uno standard valido per tutti gli implementatori di librerie MPI.

3.1.1 Il messaggio e l'envelope

Nella programmazione parallela di tipo Message Passing l'unità di informazione che i processi possono scambiarsi è il messaggio. Al fine di trasferire efficacemente i messaggi, insieme ai dati veri e propri vanno incorporate nel messaggio anche le seguenti informazioni:

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

Alcuni di questi dati vengono esclusivamente utilizzati dalla libreria di Message Passing, la quale realizza le procedure di più basso livello che si occupano dell'effettivo trasferimento dei dati. Altri invece rappresentano delle informazioni utili anche per i processi comunicanti anche se non sono parte integrante del messaggio; l'insieme di tutti questi dati è chiamato *envelope* del messaggio.

Tra i dati che possono essere contenuti nell'envelope, i seguenti campi di informazione sono sempre presenti:

- il *rank* del processo mittente, un numero che identifica univocamente un processo.
- il rank del processo destinatario.
- il *tag*, un numero intero che identifica il tipo di messaggio.
- il *comunicatore*, un identificatore del gruppo di processi cui il mittente e il destinatario appartengono.

3.1.2 Tipi di funzione

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

Quindi una libreria di Message Passing mette a disposizione comunicazioni di tipo:

Bloccante e non bloccante una funzione bloccante attende fino al completamento dell'operazione richiesta; al contrario una funzione non bloccante inizia la comunicazione e subito dopo restituisce il controllo al processo chiamante, il quale può svolgere altri compiti mentre la comunicazione procede verso il suo completamento. Il completamento di un'operazione avviene, come si vedrà nel paragrafo 3.2, nel caso della spedizione, quando il buffer è riutilizzabile dall'utente e nel caso della ricezione quando il messaggio è stato interamente ricevuto. Le comunicazioni di tipo non bloccante portano due vantaggi: il processo non è obbligato ad aspettare in stato di attesa una operazione lenta come quelle connesse al trasferimento di dati via rete e più comunicazioni possono avere luogo contemporaneamente sovrapponendosi almeno in parte tra loro ottenendo un risparmio di tempo.

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

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

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

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

3.2 Le specifiche MPI

In questo paragrafo si fornisce una breve panoramica sulle istruzioni MPI di comunicazione e sulle specifiche che devono essere soddisfatte secondo lo standard MPI [13].

Ogni istruzione base di comunicazione prevede due modalità di funzionamento: bloccante e non bloccante. Nello standard MPI esistono quattro modalità con cui un messaggio può essere spedito: standard, ready, buffered e synchronous; mentre esiste solo un tipo di receive.

3.2.1 Send

Le istruzioni che invocano la modalità standard di trasmissione, completano quando il buffer è riutilizzabile dall'utente, vale a dire quando una receive ha completato la copia del messaggio o quando lo stesso è stato copiato in un'altra zona di memoria.

Le send che usano la modalità bufferizzata sono simili alle precedenti; la differenza consiste nel fatto che la copia intermedia del buffer è obbligatoria in quanto tali funzioni sono operazioni locali, vale a dire che il loro completamento non deve dipendere dall'eventuale esecuzione di istruzioni di receive sul nodo remoto.

Al contrario, la ready send è di tipo non locale, poiché può essere iniziata solo se è stata postata una receive corrispondente; in caso contrario il risultato è erroneo e non definito. Le specifiche MPI consentono che la ready send sia implementata come una standard send; in questo modo non vi è alcun vantaggio nell'uso di una ready send rispetto ad una standard send.

Una send sincrona, oltre ad eseguire una comunicazione punto-punto tra due nodi remoti, fornisce un meccanismo di sincronizzazione tra i due processi coinvolti. Il suo completamento, infatti, non può avvenire sino a quando la ricezione del messaggio non sia iniziata; al processo trasmittente non importa però che il messaggio sia stato ricevuto con successo, ma solo che il ricevente abbia iniziato a ricevere.

Tipicamente questo controllo è effettuato in due modi possibili: trasmettendo un messaggio di acknowledge al processo che ha effettuato la send, che può essere così completata; oppure con un protocollo di handshake: a una fase di verifica che i due processi siano pronti, uno a spedire, l'altro a ricevere, segue l'invio del messaggio vero e proprio.

3.2.2 Receive e Probe

Le funzioni di ricezione completano quando il buffer contiene integralmente il messaggio inviato, e il buffer è accessibile dall'utente.

A differenza della spedizione, è consentito che un processo non specifichi un valore per il rank del trasmettitore di un messaggio, utilizzando MPI_ANY_SOURCE. Questo vale anche per il tag, ma non per il comunicatore. Un messaggio sarà ricevuto se corrispondono le caratteristiche definite con precisione.

L'istruzione MPI_Probe completa quando un messaggio che corrisponde alle caratteristiche è stato spedito. La funzione MPI_Iprobe, non bloccante, ritorna invece uno o zero rispettivamente quando un messaggio è stato trovato o non risulta presente.

L'analisi dei messaggi prima della loro ricezione è una funzione utile in quei casi in cui la dimensione del messaggio non è nota a priori; infatti, dopo aver preso atto del numero di byte del messaggio in arrivo, l'utente può allocare un buffer in memoria sufficientemente grande da contenere il messaggio, evitando uno spreco di memoria o la mancata ricezione di una parte del messaggio.

3.2.3 Operazioni collettive

Le operazioni collettive sono definite come comunicazioni che involvono un gruppo di processi; tale gruppo può essere composto anche solo di due processi. Tutti i processi all'interno del gruppo devono invocare la stessa primitiva, la cui sintassi e semantica sono definite in modo da essere consistenti con la sintassi e la semantica delle operazioni punto a punto, con le quali però non devono interferire.

Solitamente per ottenere questo risultato si utilizzano codici differenti, per lo stesso comunicatore, secondo che si richieda un'operazione collettiva o una punto a punto.

Le operazioni collettive non sono obbligate a fornire una sincronizzazione tra i processi coinvolti e quindi il fatto che un processo ha completato una funzione collettiva non implica che anche tutti gli altri processi del gruppo abbiano completato la stessa operazione.

Alcune funzioni collettive coinvolgono un processo che si distingue dagli altri, perché è l'unico a spedire o ricevere; tale processo è chiamato *root*. Alcuni parametri di queste funzioni sono significativi solo per il processo root.

Nel caso delle operazioni collettive il numero di byte spediti e ricevuti deve essere uguale, affinché la comunicazione proceda senza errori.

barrier è utilizzato per sincronizzare più processi; non avviene alcuno scambio di dati, ma la barriera blocca ogni processo del gruppo finché tutti i processi non siano giunti a eseguire tale funzione.

broadcast realizza una comunicazione di tipo uno a molti, nella quale un processo (root) trasmette uno stesso messaggio agli altri processi del gruppo, se stesso incluso.

gather permette di raccogliere un insieme di dati; ogni processo del gruppo (root incluso) spedisce al root un dato. Il root riceve i dati e li memorizza in ordine secondo il rank del trasmettitore, creando dai dati disaggregati un insieme omogeneo.

scatter è l'opposto della precedente; il root spedisce ai processi del gruppo, secondo il loro rank, una parte dei dati originali.

reduce il processo root combina gli elementi forniti all'input buffer dai processi partecipanti usando un particolare operatore specificato nella chiamata Le principali operazioni sono predefinite, ma l'utente può definirne a piacere, posto che l'operazione sia associativa. L'ordine di valutazione delle operazioni è basata sul rank del processo. queste operazioni vengono usate nel caso che si debba eseguire un calcolo per ottenere un singolo risultato a partire da dati distribuiti su un gruppo di processi.

3.2.4 Altre istruzioni

Di seguito si presentano, per completezza, le specifiche di poche altre istruzioni.

MPI_Test controlla se una istruzione non bloccante è stata completata.

MPI_Wait attende che l'istruzione non bloccante abbia completato prima di ritornare il controllo al processo chiamante.

MPI_Cancel cancella l'esecuzione di una comunicazione non bloccante. Questo è possibile in LAM solo se la richiesta non è attiva; in caso contrario la comunicazione è completata e la funzione MPI_Cancel non completa con successo.

Esistono varie versioni delle funzioni MPI_Test e MPI_Wait (MPI_Testany, MPI_Testall, MPI_Testsome) che controllano rispettivamente il completamento di almeno una, tutte o alcune comunicazioni all'interno di un insieme di istruzioni di comunicazioni invocate in precedenza.

Capitolo 4

Local Area Multicomputer (LAM)

La libreria LAM è stata scelta tra le altre implementazioni MPI per la semplicità nell'integrare un nuovo protocollo di comunicazione.

È infatti costituita da due livelli: un *Upper Layer* che definisce le funzioni offerte all'utente in termini di operazioni più elementari, indipendenti dall'hardware e dal sistema operativo; e un *Lower Layer* che si occupa dell'effettivo trasporto dei dati.

I due livelli sono indipendenti e interagiscono tramite un'interfaccia, chiamata *Request Progression Interface* (RPI), costituita da otto funzioni.

Grazie a questa scelta nella progettazione di LAM, l'aggiunta di un nuovo protocollo di comunicazione richiede solo l'intervento sul Lower Layer, riscrivendo l'implementazione delle funzioni della RPI.

4.1 Upper layer

Il livello superiore si occupa di offrire le funzioni MPI di cui l'utente si può servire. Ogni funzione di comunicazione è espansa dall'Upper Layer, in una serie di operazioni, finalizzate alla creazione e alla gestione di una lista con-

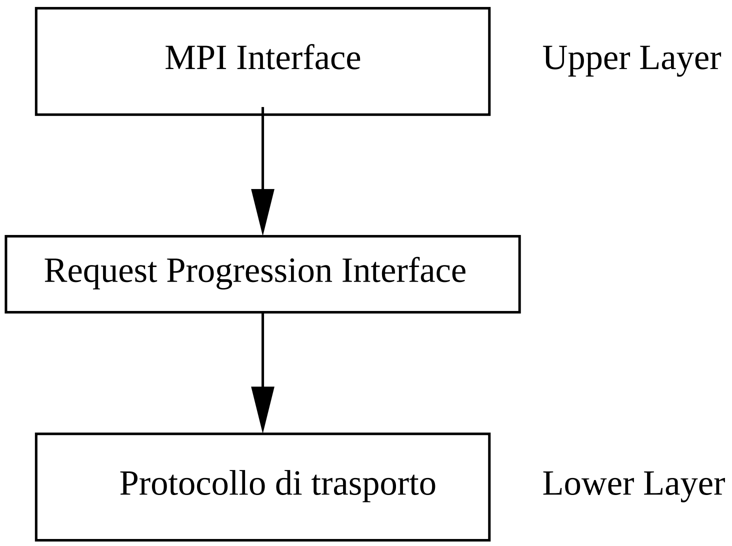

Figura 4.1: Modello di LAM.

catenata. Gli elementi di cui questa lista si compone contengono, condensati in una struttura dati che vedremo in un paragrafo successivo, i dati principali del messaggio e dell'envelope. Sarà poi compito dell'altro livello di LAM processare questa lista ed eseguire i passi necessari a completare le comunicazioni richieste.

In particolare le funzioni fondamentali, necessarie per implementare l'insieme delle primitive di comunicazione sono:

_mpi_req_build alloca la memoria per contenere un elemento della lista, inizializzando alcuni campi contenenti i parametri passati direttamente dalle funzioni MPI; inoltre stabilisce un eventuale buffer per le operazioni che lo richiedano.

_mpi_req_start reinizializza i parametri per le richieste persistenti (che non devono essere riallocate) ed esegue eventuali operazioni di frammentazione dei dati in pacchetti trasferibili dalle funzioni del Lower Layer.

_mpi_req_add aggiunge la richiesta in cima alla lista.

_mpi_req_blkclr riporta a zero il flag indicante richiesta bloccante.

_mpi_req_blkset rende la richiesta bloccante.

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

_mpi_req_rem rimuove la richiesta completata dalla lista.

_mpi_req_destroy distrugge la richiesta liberando la memoria occupata.

4.2 Request Progression Interface

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

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

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

Per effettuare le operazioni finalizzate a scambiare tra i processi i messaggi contenuti nella lista delle richieste, l'interfaccia RPI mette a disposizione dell'Upper Layer otto funzioni[1] [18] che il progettista di un nuovo Lower Layer deve necessariamente realizzare:

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

_rpi_c2c_init questa funzione è chiamata da MPI_Init ed effettua l'inizializzazione delle operazioni relative alle comunicazione che sono compiute una sola volta nel corso dell'esecuzione di un programma MPI. Al suo interno è chiamata _rpi_c2c_addprocs.

_rpi_c2c_addprocs è utilizzata per creare il complesso delle strutture che i processi dovranno usare per comunicare. È chiamata da _rpi_c2c_init affinché l'inizializzazione delle comunicazioni interprocesso abbia luogo all'atto della chiamata a MPI_Init cioè all'inizio della parte di codice che effettua chiamate alle librerie MPI; tuttavia può essere chiamata anche in altre occasioni, durante l'esecuzione di un programma, dalle funzioni MPI_Spawn e MPI_Intercomm_create per aggiungere altri processi. In entrambi i casi l'Upper Layer inserisce i processi creati dall'utente LAM nella lista dei processi poi chiama _rpi_c2c_addprocs.

_rpi_c2c_build procede all'inizializzazione delle variabili relative al Lower Layer e necessarie alla comunicazione.

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

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

_rpi_c2c_advance questa funzione svolge la quasi totalità del lavoro di comunicazione dal momento che cerca di avanzare per quanto possibile la lista delle richieste creata e aggiornata nelle fasi precedenti. Nel caso di richieste bloccanti la funzione non ritorna finché almeno una richiesta bloccante della lista non sia stata completata. È importante sottolineare che l'Upper Layer non si preoccupa delle modalità con cui i trasferimenti di dati avvengono, ciò che importa è che il Lower Layer aggiorni costantemente lo stato della richiesta. È anche possibile che una richiesta che sia già stata inserita nella lista venga rimossa dall'Upper Layer: di ciò il Lower Layer deve tenere conto, mentre non è consentito che una richiesta già in parte trasmessa venga cancellata.

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

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

Le funzioni _rpi_c2c_fastsend e _rpi_c2c_fastrecv, citate in [18] e presenti solo dalla versione 6.3, in realtà non sono funzioni RPI poiché al fine di aumentare le prestazioni sono chiamate in casi particolari aggirando la Request Progression Interface e avanzando la send o la receive bloccante senza creare la richiesta.

4.3 Lower layer

Nella distribuzione originale delle ultime versioni LAM sono compresi quattro diverse realizzazione del Lower Layer. La prima, inizialmente adottata da LAM, utilizza i demoni; prima di eseguire qualunque operazione,è necessario, attraverso una procedura di avvio, attivare su ogni nodo del cluster un demone.

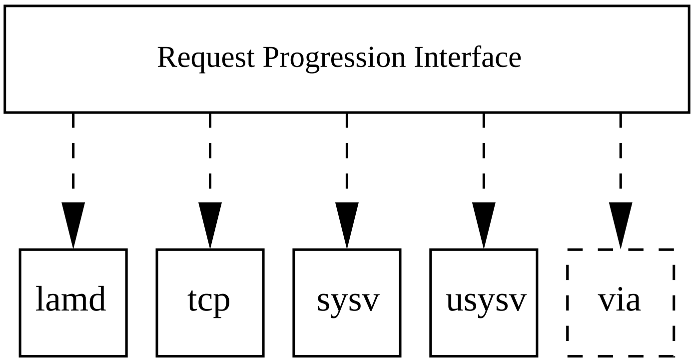

Figura 4.2: Struttura del Lower Layer.

Esso, costituito da un nano-kernel e da una serie di thread, offre vari servizi ai processi che desiderano comunicare tramite LAM: il nano-kernel fornisce semplici funzionalità di message passing ai processi locali che non comunicano mai direttamente tra loro, ma solo attraverso il demone; alcune thread si occupano delle comunicazioni tra demoni che stanno su nodi diversi tramite un protocollo basato su UDP; altre thread svolgono attività come la gestione dei processi e l'accesso al filesystem.

Gli altri tre tipi di Lower Layer sono una derivazione di un concetto di comunicazione diverso da quello precedente: la comunicazione interprocesso avviene essenzialmente tramite TCP/IP. Ciò che differenzia le tre realizzazioni sta nel modo in cui è gestita la comunicazione tra processi residenti sullo stesso nodo:

tcp socket TCP/IP.

sysv shared memory con semafori.

usysv shared memory con spin locks.

Per potere utilizzare una nuova tipologia di comunicazione all'interno di LAM non è necessario riscrivere tutto il codice, ma basta realizzare un nuovo Lower Layer che si interfaccia con l'Upper Layer con le otto funzioni che costituiscono la Request Progression Interface.

Dopo avere progettato un protocollo adatto per lo svolgimento delle comunicazioni, il passo successivo consiste nella integrazione del protocollo e del controllo di flusso all'interno del Lower Layer di LAM.

Nel capitolo successivo sono descritti i meccanismi studiati a tal fine.

4.4 Strutture dati

4.4.1 Richieste

Ogni processo attivo in LAM MPI mantiene una lista concatenata delle richieste che devono essere soddisfatte: l'Upper Layer si occupa di creare e gestire questa lista in base ai cambiamenti di stato di ogni richiesta indotti dalle operazioni del livello inferiore.

I campi della struttura _req utilizzati o modificati dal Lower Layer sono:

type tipo di comunicazione richiesta (i quattro tipi di send, receive o probe). Esiste anche una richiesta fake ma questo tipo non è mai visto dal Lower Layer.

state stato della richiesta. Il Lower Layer deve aggiornare questo campo in corrispondenza all'avanzamento della richiesta.

marks caratteristiche di una richiesta persistente.

flag caratteristiche di tutte le richieste, persistenti e non.

packbuf puntatore all'area di memoria contenente i dati da inviare o destinata ai dati da ricevere.

packsize dimensione del buffer precedente.

cid identificativo del comunicatore utilizzato.

seq numero di sequenza del messaggio.

status contiene lo stato della richiesta. Nel caso di una ricezione i campi di questa struttura vanno opportunamente settati.

next puntatore alla prossima richiesta della lista.

rpi puntatore a struttura. Consente di aggiungere alla richiesta tutti i dati utilizzati solo dal Lower Layer.

Una richiesta contiene in più i dati passati dall'utente e altri campi utilizzati esclusivamente dal livello alto di LAM.

Gli stati possibili di una richiesta sono quattro:

INIT la richiesta è appena stata creata;

START la richiesta è stata inizializzata dal Lower Layer;

ACTIVE il Lower Layer sta processando la richiesta;

DONE tutte le comunicazioni richieste sono completate.

4.4.2 Processi

LAM mantiene anche una lista di strutture che contengono le caratteristiche dei processi attivi.

Tale struttura, _proc, è così composta:

p_gps è una struttura che identifica in modo univoco il processo;

p_mode flag per memorizzare informazioni di stato. Gli otto bit meno significativi sono riservati all'Upper Layer;

p_rpi informazioni relative al processo specifiche del Lower Layer utilizzato.

Il Lower Layer può scandire la lista dei processi attivi solo attraverso le funzioni *lam_topproc ()* e *lam_nextproc ()*.

Capitolo 5

Integrazione di LAM su M-VIA

I due modelli di comunicazione consentiti da VIA non si prestano a realizzare tutte le tipologie di trasferimento dei dati previste dallo standard MPI.

Esso, infatti, offre al programmatore operazioni sincrone, asincrone, bloccanti e non bloccanti; occorre quindi utilizzare le primitive di VIA per costruire un protocollo più complesso, le cui caratteristiche principali sono così riassumibili:

- deve essere asincrono, cioè ogni processo deve poter spedire in qualunque momento, e deve potersi predisporre a ricevere senza sapere se e quando la comunicazione avverrà effettivamente;
- deve poter stabilire una sincronizzazione tra i processi, quando richiesto;
- deve fornire la notifica del completamento della comunicazione ai processi coinvolti;
- deve gestire le risorse, per evitarne l'esaurimento.

5.1 Protocollo utilizzato

Per la costruzione del protocollo [2], andranno valutate:

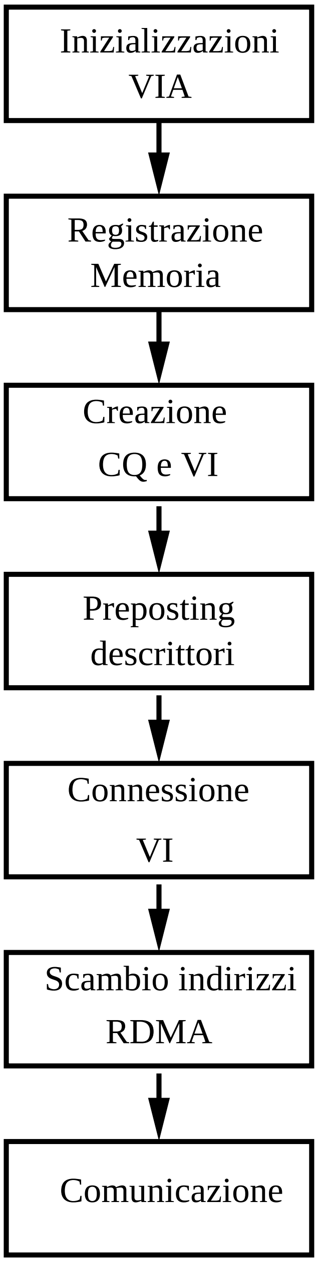

Figura 5.1: Protocollo basato sul pre-posting dei descrittori.

- la modalità di trasmissione dei dati;
- la dimensione dei buffer da registrare per spedire e per ricevere;
- la criticità delle funzioni MPI;
- un controllo di flusso asincrono rispetto alle comunicazioni.

La scelta tra l'utilizzo di Send/Receive o operazioni di tipo RDMA è obbligata. Send/Receive, infatti, non permette operazioni asincrone; se la Receive Queue è vuota o il buffer di ricezione non è sufficientemente grande a contenere il messaggio, la comunicazione restituisce un errore.

Tuttavia, RDMA Write o Read non consentono una notifica del completamento della comunicazione; il caso RDMA Write con Immediate Data costituisce una importante eccezione. Questo tipo di comunicazione consente una spedizione asincrona dei dati e consuma un descrittore della VI del ricevitore, che è così informato del completamento dell'operazione.

Occorre puntualizzare che affinché i due processi comunicanti possano utilizzare primitive di tipo RDMA, è necessario lo scambio tra di essi degli indirizzi delle locazioni di memoria che mettono a disposizione per la comunicazione. Questo può essere fatto solo attraverso operazioni di Send/Receive.

Come detto, la scelta di utilizzare RDMA Write con Immediate Data, al fine di ottenere notifica del completamento della comunicazione, comporta anche il consumo di un descrittore in ricezione. Quindi è necessario che il ricevitore preposti, in fase di inizializzazione e prima delle comunicazioni, un grande numero di descrittori vuoti sulle Receive Queue di ogni VI. Ciò è permesso da VIA, mentre un descrittore vuoto sulla Send Queue porta ad un errore.

Poiché VIA richiede che le locazioni di memoria contenenti i dati da trasferire o destinate a riceverli siano registrate, occorre decidere le dimensioni ottimali di tali buffer. Occorre tenere conto del fatto che la dimensione dei messaggi da spedire non è fissa né nota a priori.

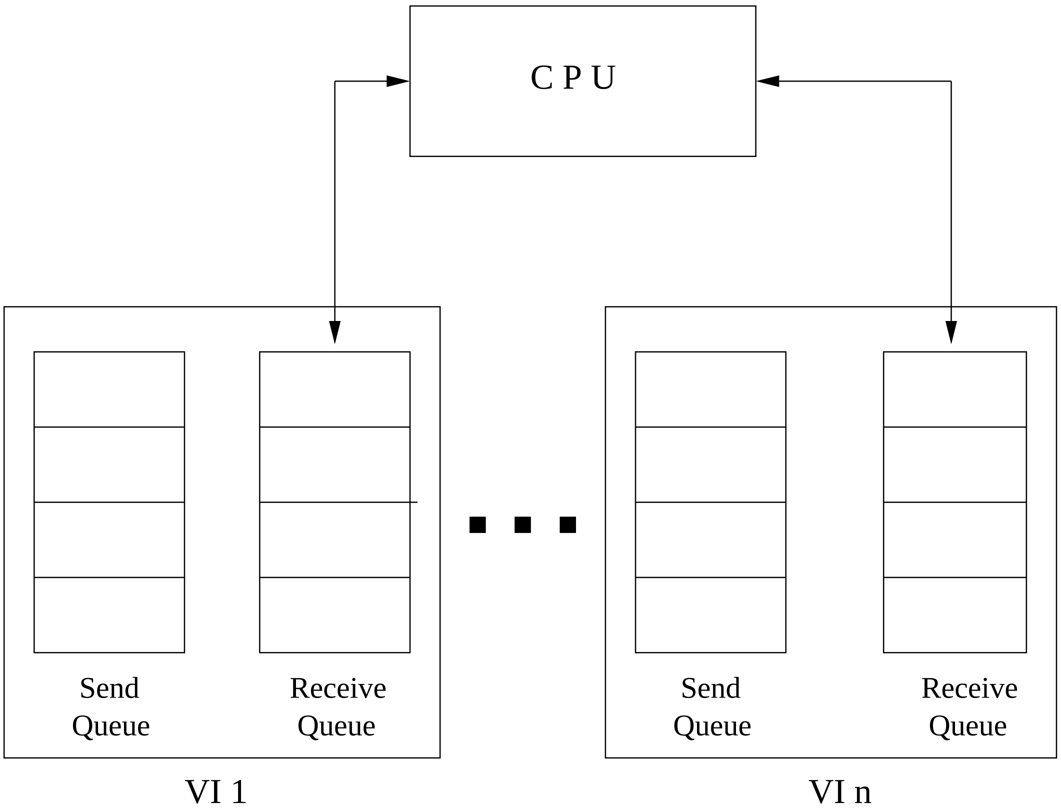

Figura 5.2: Polling sulle Receive Queue di tutte le VI.

Per questo le scelte sono state:

Trasmettitore si registrano pochi descrittori e un uguale numero di buffer di dimensione uguale a quella massima consentita da M-VIA;

Ricevitore si registrano un grande numero di descrittori, e una grande regione di memoria, detta RDMA area, in cui i messaggi verranno posti in sequenza.

Resta da analizzare la criticità delle funzioni MPI rispetto all'utilizzo di VIA. VIA è un protocollo orientato alla connessione; due processi comunicano tramite le due VI connesse tra loro, e non prevede primitive che consentano comunicazioni di tipo collettivo. Lo standard MPI, invece, prevede La receive

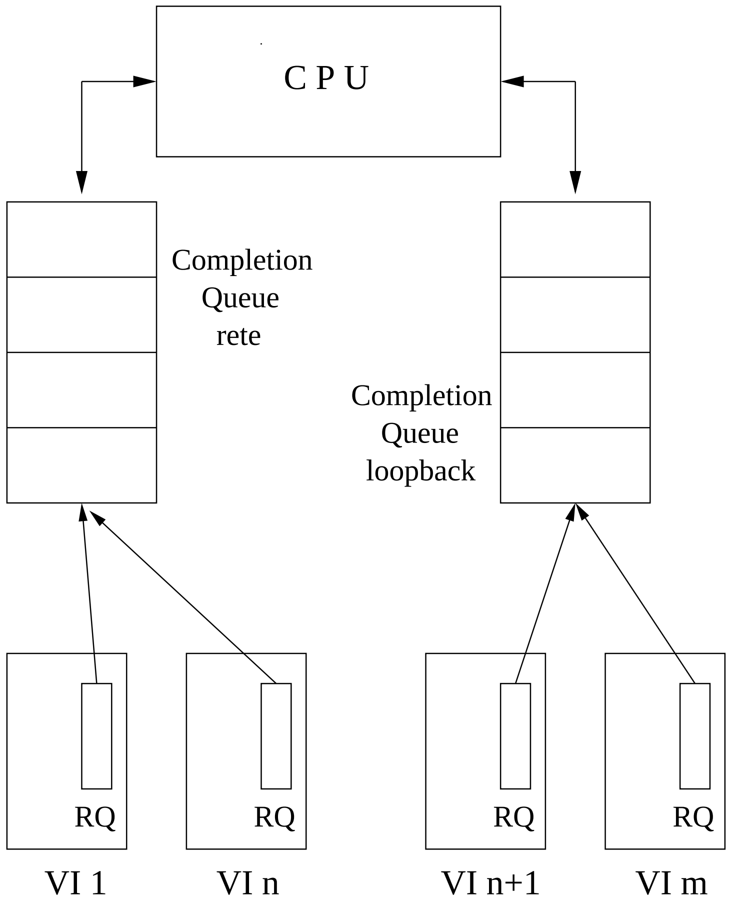

Figura 5.3: Uso di due Completion Queue.

che specifica MPI_ANY_SOURCE nel campo riservato al trasmettitore, consente di ricevere un messaggio indipendentemente da chi lo ha spedito. Date queste premesse diventa chiaro come la realizzazione di receive bloccante di questo tipo sia l'elemento più critico del protocollo.

Le possibilità studiate per la realizzazione di una tale receive in forma bloccante sono:

Polling su tutte le VI è un sistema assai dispendioso in termini di risorse di sistema perché tiene il processore in attesa su una serie di code. Vedi figura 5.2

Uso della VipRecvNotify Quando un descrittore completa su una Receive Queue, questa primitiva di VIA attiva una funzione incaricata di gestire la ricezione del messaggio, chiamata *Handler*. Purtroppo non era ancora disponibile nella versione 0.9.3 di M-VIA, al momento di iniziare questo progetto.

Uso di due Completion Queue poiché le Completion Queue sono gestite dal NIC, ogni Completion Queue è legata a un NIC; esisterà quindi la Completion Queue per i messaggi di rete e la Completion Queue per i messaggi provenienti dal NIC di loopback; questo però conduce a dover fare un polling continuo su due code, con un sotto utilizzo delle risorse. Vedi figura 5.3

Uso di una Completion Queue Questa soluzione, che è stata poi adottata, rispetto alla precedente differisce per il fatto che due processi, anche se risiedono sullo stesso nodo, comunicano attraverso la scheda di rete.

Poiché le risorse possono esaurirsi occorre un controllo di flusso che eviti per quanto possibile tale situazione e che informi i processi comunicanti quando questo avviene.

In particolare sarà necessario cautelarsi contro l'esaurimento di:

- spazio libero sulla RDMA area;
- descrittori vuoti sulla Receive Queue.

5.2 La Request Progression Interface per VIA

Una volta decise le caratteristiche del protocollo, occorre integrarlo in LAM, riscrivendo il Lower Layer. Come già detto, per potere utilizzare un nuovo livello di trasporto all'interno di LAM è sufficiente realizzare un nuovo Lower Layer che si interfaccia con l'Upper Layer con le otto funzioni che costituiscono la Request Progression Interface.

Segue una breve descrizione delle competenze di ogni funzione RPI:

_rpi_c2c_init inizializza il VI NIC coinvolto nella comunicazione, crea i protection tag per la registrazione della memoria e inizializza il name service di M-VIA.

_rpi_c2c_addprocs inserisce la struttura dati VIInfo, che contiene tutte le informazioni necessarie sulle VI e sulle RDMA area proprie e del peer connesso, in un hash table, in cui il parametro di ricerca è il rank. Infatti ogni processo è connesso con ognuno degli altri con una sola VI, perciò ogni VI può essere identificata dal rank del processo con cui essa si connette.

In particolare, tale struttura contiene

- il rank del processo connesso;
- un puntatore al NIC necessario alla comunicazione
- il protection tag;
- un identificativo della VI connessa, utilizzato dal VI Provider;

- un puntatore ai descrittori della Send Queue e della Receive Queue e il loro numero;
- gli indirizzi delle RDMA Aree coinvolte;
- le variabili per il controllo di flusso.

_rpi_c2c_build si è introdotto un nuovo campo dell'envelope per consentire la distinzione tra messaggi che necessitano di un acknowledge (identificati da un numero positivo crescente), inviati tramite una send sincrona, messaggi che non richiedono l'invio dell'ack (identificati da zero) e acknowledge stessi (identificati da un numero negativo).

_rpi_c2c_start avanza in parte o del tutto la trasmissione di richieste dirette al processo medesimo.

_rpi_c2c_destroy nel caso di questa particolare realizzazione di LAM su VIA, non occorre che tale funzione svolga alcun compito.

_rpi_c2c_advance distingue richieste di send o di receive e si occupa di invocare le opportune funzioni del Lower Layer. Nei paragrafi successivi si spiega come le funzioni di comunicazione svolgono il loro lavoro e i problemi che si sono incontrati e risolti nella loro realizzazione.

_rpi_c2c_iprobe senza chiamare la _rpi_c2c_advance, cerca di avanzare il più possibile le richieste non bloccanti di Probe.

_rpi_c2c_finalize disconnette e distrugge le VI, deregistra la memoria affinché il sistema operativo possa riutilizzarla, chiude il name service e il NIC di comunicazione. Inoltre libera la memoria occupata dalle liste create all'interno del Lower Layer.

Di seguito sono descritti i meccanismi studiati in modo che siano gestite le numerose operazioni di comunicazione definite dallo standard MPI.

5.3 Le primitive di send

Lo standard MPI prevede quattro tipi diversi di funzioni di trasmissione, come descritto nel paragrafo 3.2; che sono implementate tramite la chiamata alla funzione VipSendDone.

Le tipologie standard e bufferizzata sono identiche, dal punto di vista di VIA, in quanto il messaggio è comunque copiato dall'Upper Layer in un'altra zona di memoria nel caso di MPI_Bsend.

Per quanto riguarda la ready send, la scelta è stata quella di ricorrere ad una send standard, come permesso dallo standard MPI [13].

La scelta relativa alla send sincrona è stata quella di utilizzare un messaggio di acknowledge, una volta terminata la ricezione del dato nella RDMA area. L'equivalente istruzione non bloccante differisce da quelle bloccante nel fatto che l'attesa dell'acknowledge avviene in background.

5.4 Le primitive di receive

A causa di VIA, protocollo orientato alla connessione, si è pensato di ricorrere ad una Completion Queue, in modo che al processo ricevente venga notificato quando una qualunque VI è interessata da una comunicazione.

Le Receive Queue di tutte le VI sono state connesse ad una CQ; ogni volta che un descrittore è completato sulla RQ di una VI, nella CQ è inserita una struttura che rimanda alla VI su cui è avvenuta la comunicazione.

5.4.1 L'uso della Completion Queue

Quando un processo vuole ricevere, interroga la Completion Queue per verificare se che sia arrivato un messaggio su una qualunque delle VI. Se così è si confrontano i messaggi con la richiesta di receive; se questa e il messaggio in cima alla coda corrispondono, si interrompe l'analisi della Completion Queue e si riceve il messaggio, se non corrispondono e la receive è bloccante il processo chiama l'istruzione VipCQWait che attende l'arrivo del giusto messaggio. È da notare che solo il giusto messaggio consente al processo di uscire dal loop.

Le istruzioni per il polling della Completion Queue, però, tolgono dalla coda l'elemento interrogato; se non si prendessero adeguate contromisure il messaggio sarebbe perso. Nel prossimo paragrafo è descritto il meccanismo utilizzato per tenere traccia dei dati arrivati ma non ancora passati a livello utente.

5.4.2 Le liste

Le principali esigenze del meccanismo da implementare sono:

- la rapidità, evitando copie intermedie in memoria,
- il mantenimento dell'ordine di arrivo dei messaggi, affinché una serie di receive from ANY_SOURCE ricevano i messaggi nell'ordine in cui sono stati effettivamente ricevuti dalle VI;
- il mantenimento di informazioni aggiuntive sui messaggi, ad esempio quelle utili per il controllo di flusso.

Si è ritenuto che una lista di puntatori alle envelope dei messaggi potesse essere proficuamente utilizzata nel caso in questione. Ogni elemento di tale lista (vedi figura 5.4) è composta da una struttura costituita da un puntatore che indica l'inizio del messaggio e da un intero che dà la stessa informazione come offset rispetto all'inizio della RDMA area corrispondente.

Il motivo di questa ridondanza è dovuta all'impossibilità di ricavare un valore intero da due puntatori di tipo diverso, il puntatore all'envelope e il puntatore all'inizio della RDMA area, a causa delle specifiche del linguaggio di programmazione utilizzato. Tale intero è utilizzato per controllare il numero di byte letti di una RDMA area dal ricevitore, come specificato nel paragrafo sul controllo di flusso.

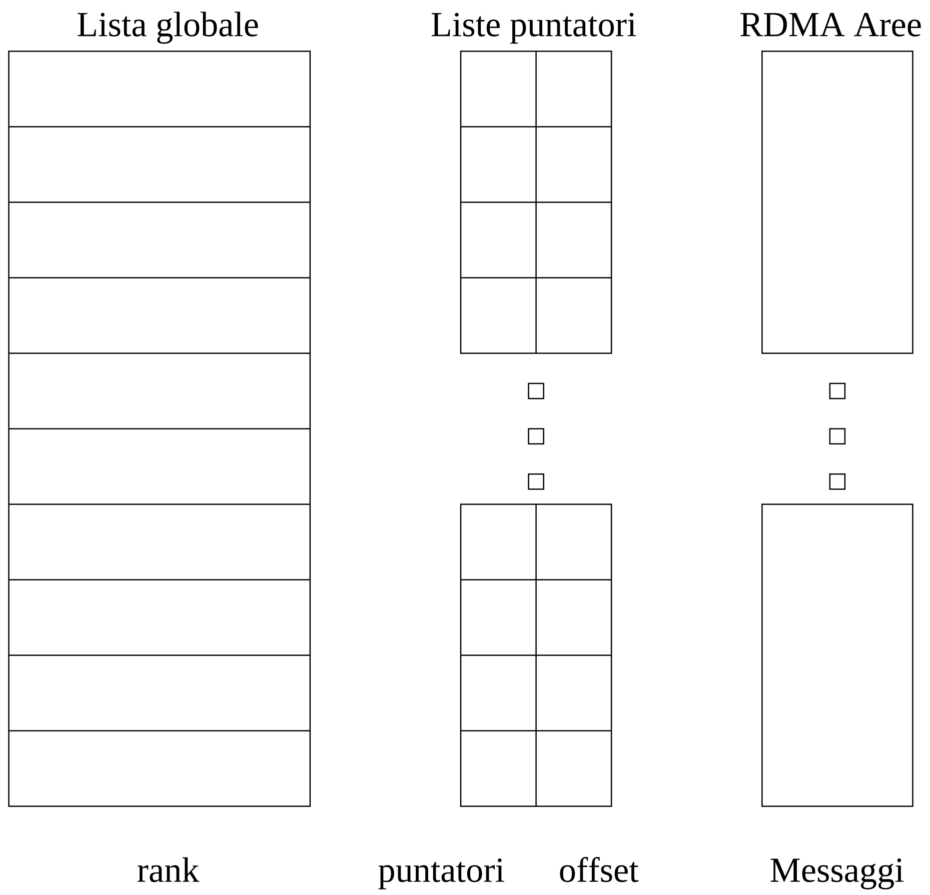

Figura 5.4: Contenuto delle liste.

Poiché le RDMA area sono tante quante sono i peer con cui il processo è connesso, anche tali liste dovranno essere in tale numero. L'array di liste concatenate, però, non permette il mantenimento di un ordine globale tra tutti i messaggi. Si utilizza, allora, una lista contenente i rank dei processi che hanno spedito un messaggio, nell'ordine di ricezione.

Quando un processo vuole ricevere, prima di interrogare la Completion Queue, controlla se la lista dei rank è vuota. Se questa contiene dei messaggi, la si scandisce alla ricerca di un elemento che indichi il rank specificato dalla richiesta di receive. Se la receive è from ANY_SOURCE, saranno scandite tutti le liste facenti parte dell'array, indicate dal rank dell'elemento della lista globale.

Consideriamo il caso che nella lista sia presente almeno un elemento; il messaggio puntato dal primo elemento della lista è comparato alla receive: se non corrisponde si continua a scandire la lista globale fino alla sua fine o fino a che non si incontri un messaggio ricevibile.

Nel caso si trovi un messaggio si elimina l'elemento dalla lista, si copia il messaggio e si torna con successo. Se arrivati in fondo alla lista dei rank non si è ancora trovato il messaggio cercato si cercherà nella Completion Queue, come descritto nel paragrafo precedente.

5.5 Controllo di flusso

5.5.1 RDMA area

Poiché la dimensione della zona di memoria di ricezione non è infinita, occorre un meccanismo che consenta il controllo, ogni volta che si effettua una comunicazione, dello spazio rimasto.

A tal fine si introducono alcune variabili e opportune procedure per la

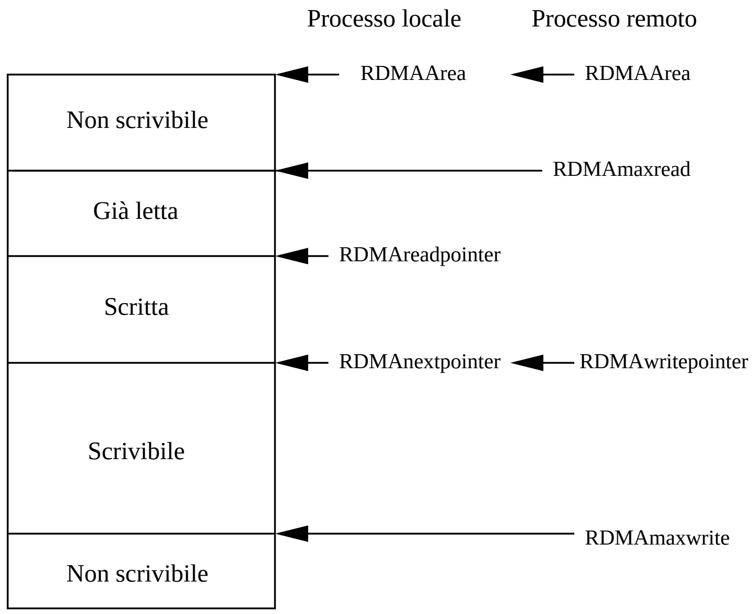

Figura 5.5: Area di memoria usata per RDMA e variabili per il controllo dello spazio disponibile.

loro gestione[1] [2]; alcune di queste si riferiscono alla RDMA area propria, altre alla RDMA area del peer con cui si comunica.

Una breve descrizione dell'uso delle variabili, riferite alla stessa RDMA Area, è qui riportato:

- variabili possedute dal processo proprietario della RDMA Area:

  RDMAArea è il puntatore alla zona di memoria registrata per ricevere dati dal peer; è utilizzato per leggere i messaggi.

  RDMAnextpointer identifica il byte da cui partirà il prossimo messaggio.

  RDMAreadpointer indica la prima zona contigua di memoria libera in cui si andrà a riscrivere; infatti, non vi sono messaggi non letti prima di essa.

[1]ogni VI possiede le sue proprie variabili che vengono gestite autonomamente

- variabili possedute dal processo connesso:

  RDMAArea è il puntatore alla zona di memoria registrata per scrivere dati; è utilizzato per scrivere i messaggi.

  RDMAwritepointer indica il primo byte libero, da cui partirà il prossimo messaggio. È quindi uguale al RDMAnextpointer.

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

  RDMAmaxread è una variabile scritta attraverso RDMA per notificare quanti byte di questa RDMA Area l'altro processo abbia letto e che quindi possono essere riutilizzati per l'invio di messaggi.

5.5.2 Descrittori sulla Receive Queue

Quando un processo cerca di spedire un messaggio su una VI, la cui Receive Queue è vuota, tale VI si porta in stato di errore, da cui può uscire solo con la disconnessione. In più tale errore non è notificato al processo ricevente, che non ha quindi modo di sapere quale sia lo stato della VI.

Per questo motivo sono necessarie alcune variabili, riferite ad ogni VI connessa, tramite le quali un processo può conoscere il numero di descrittori effettivamente consumati.

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

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

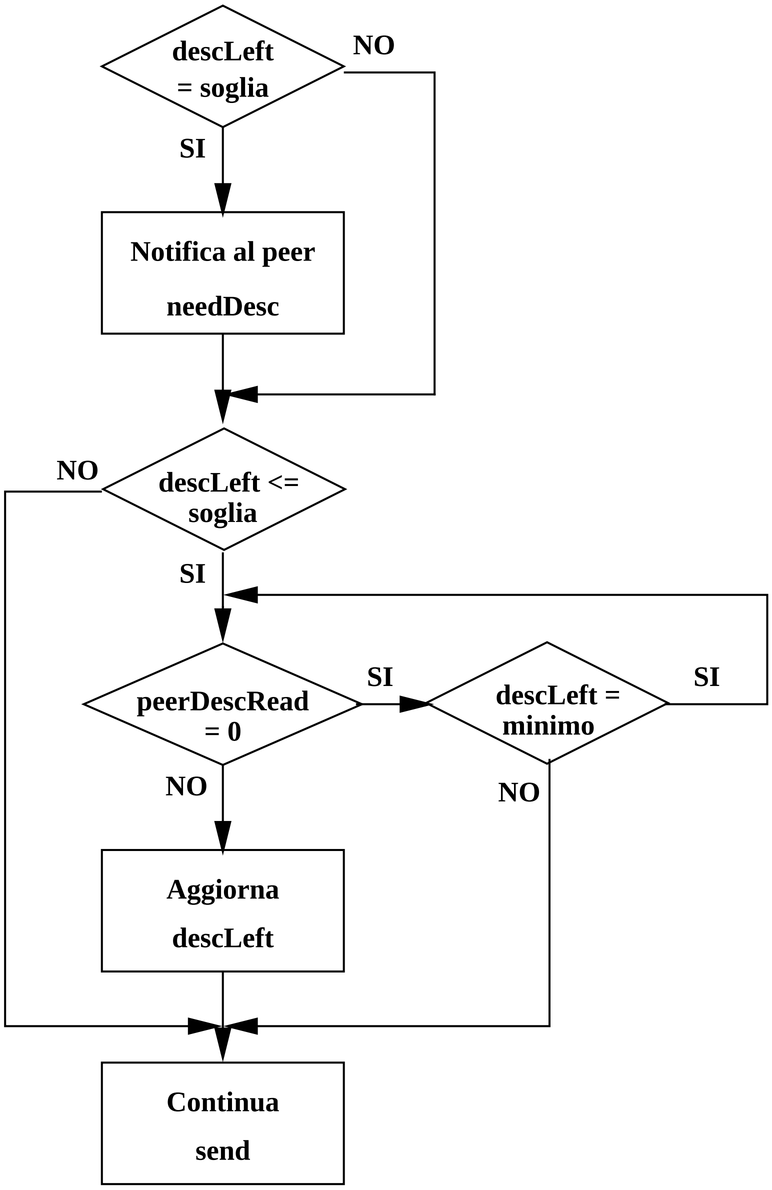

Figura 5.6: Controllo del numero di descrittori.

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

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

5.6 La comunicazione a se stessi

Non è raro che un programma MPI preveda un master che assegna i compiti da eseguire ai vari slave. Se il processo master deve solo raccogliere e presentare i dati elaborati è possibile che parte dell'elaborazione venga assegnata a se stesso.

In tal caso non è infrequente che il processo master invii e riceva dati da se stesso come dagli altri, utilizzando quindi primitive MPI.

Per questo motivo è necessario implementare una serie di funzioni per la comunicazione da e a se stessi. Per gestire queste comunicazioni non si utilizzerà VIA, ma una lista concatenata di buffer.

Per rispettare le specifiche dello standard MPI, non è possibile lasciare che una send attenda fino a quando non è stata postata la corrispondente receive; il messaggio va inevitabilmente copiato in un buffer intermedio, da dove verrà prelevato dalla receive.

5.7 Esempio di utilizzo delle liste

Per comprendere meglio il funzionamento del meccanismo della ricezione, in questo paragrafo si presenta un esempio di comunicazione tra tre processi. Si immagini che l'utente abbia create tre processi, con i rank 0,1 e 2. Fissiamo l'attenzione sul processo di rank 0 e immaginiamo che con cinque send successive degli altri due abbia ricevuto i messaggi nell'ordine riportato dalla tabella 5.1.

Source	*Tag*
1	0
2	0
1	1
1	2
1	3

Tabella 5.1: Messaggi giunti al rank 0.

Se il processo di rank 0 cerca il messaggio avente il tag 3, pone questi messaggi in lista, tranne l'ultimo che è ricevuto.

A questo punto la situazione delle liste e delle RDMA aree del processo 0 è mostrata in figura 5.7.

Si immagini ora che il processo di rank 0 esegua le receive nell'ordine riportato dalla tabella 5.2.

La prima receive, come detto, consuma l'ultimo messaggio arrivato. La seconda receive consuma il messaggio che sta al centro della lista corrispondente; questa operazione non modifica il RDMAreadpointer, poiché la scrittura della RDMA area avviene in sequenza.

La nuova situazione è quindi rappresentata in figura 5.8.

Source	*Tag*
1	3
1	1
2	0
1	0
1	2

Tabella 5.2: Ordine delle receive del rank 0.

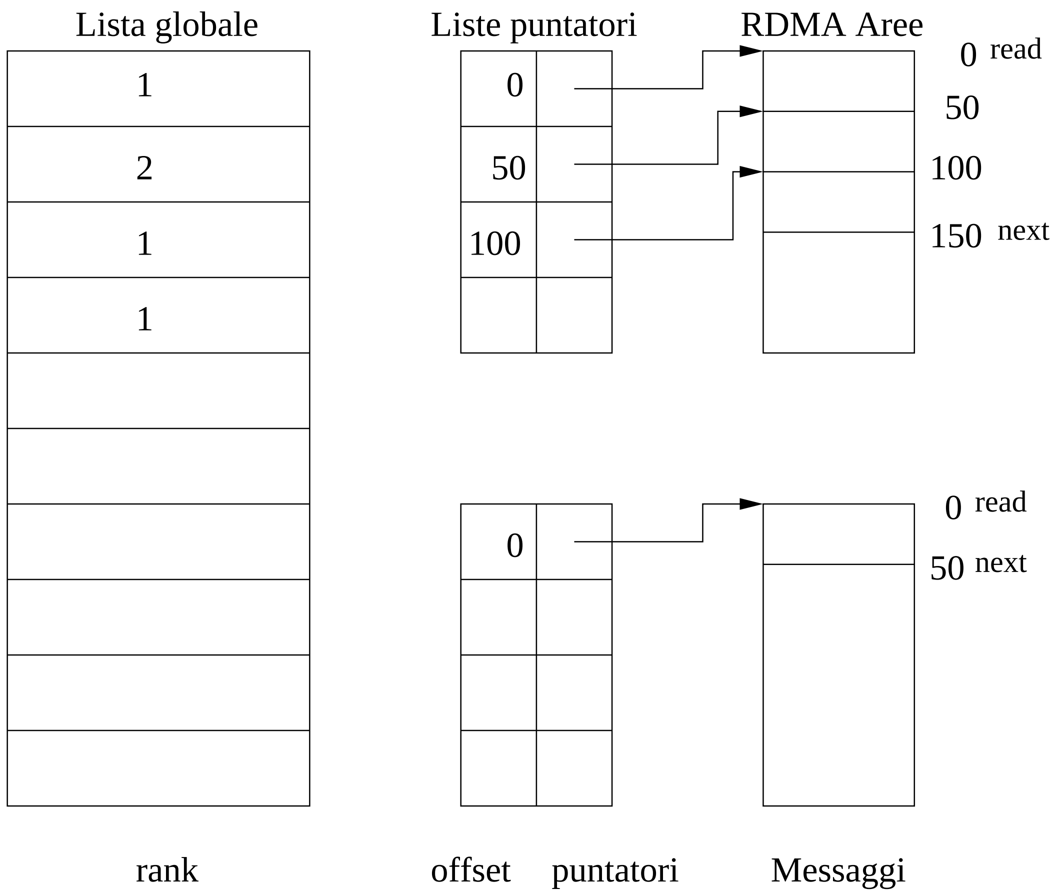

Figura 5.7: Situazione iniziale.

La terza receive riceve l'unico messaggio sulla lista 2. In questo caso RDMAreadpointer viene posto uguale a RDMAnextpointer poiché non vi sono più messaggi da consumare su questa lista.

I messaggi ancora da ricevere sono rappresentati in figura 5.9.

Con la quarta receive il processo di rank 0 consuma il primo messaggio; poiché esso è il primo della lista corrispondente, ma essa non è vuota, RDMAreadpointer viene posto uguale a all'offset dell'unico messaggio rimasto.

In figura 5.10 si mostra la situazione delle liste del processo 0, quando rimane da consumare un solo messaggio.

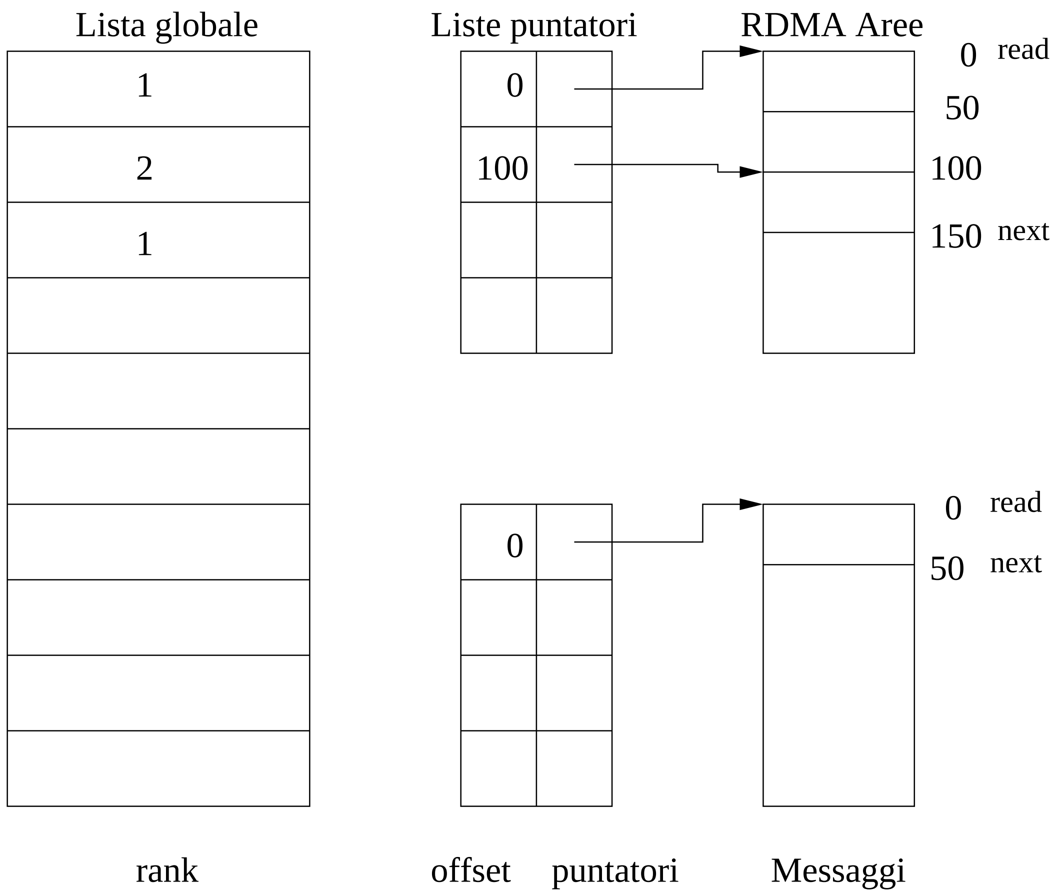

Figura 5.8: Dopo la ricezione del terzo messaggio.

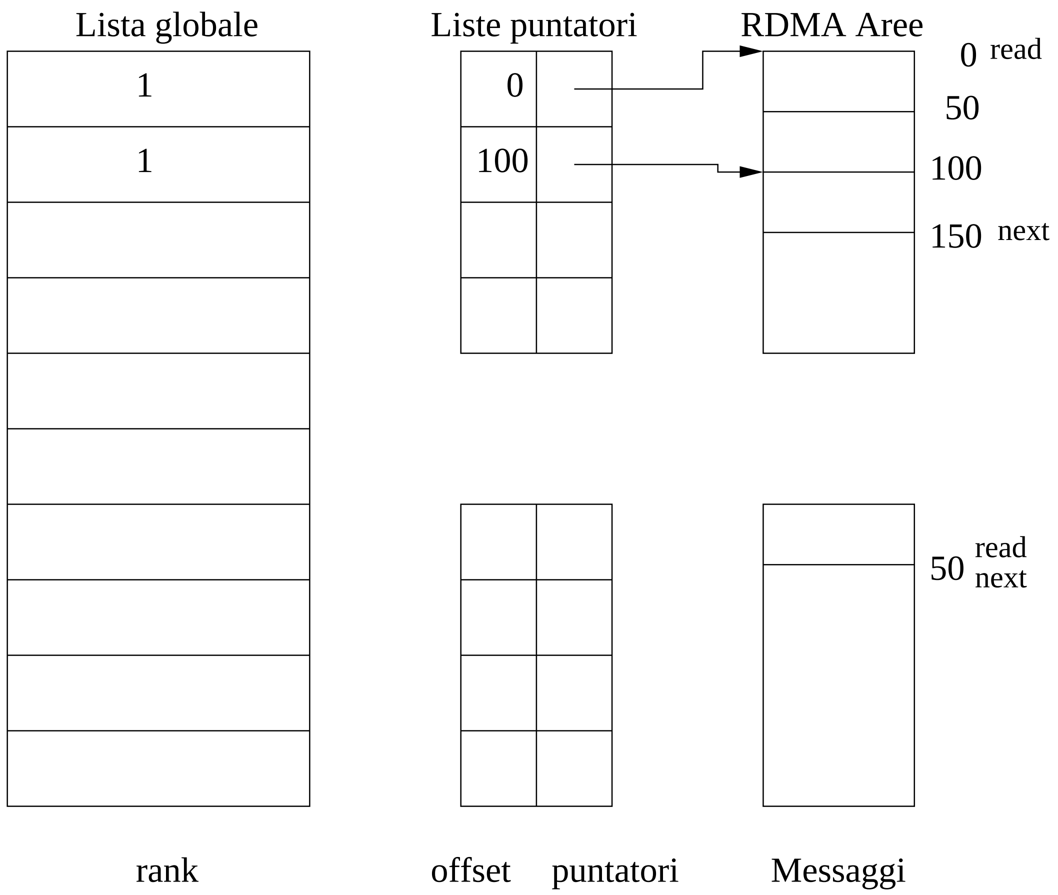

Figura 5.9: La lista 2 è vuota.

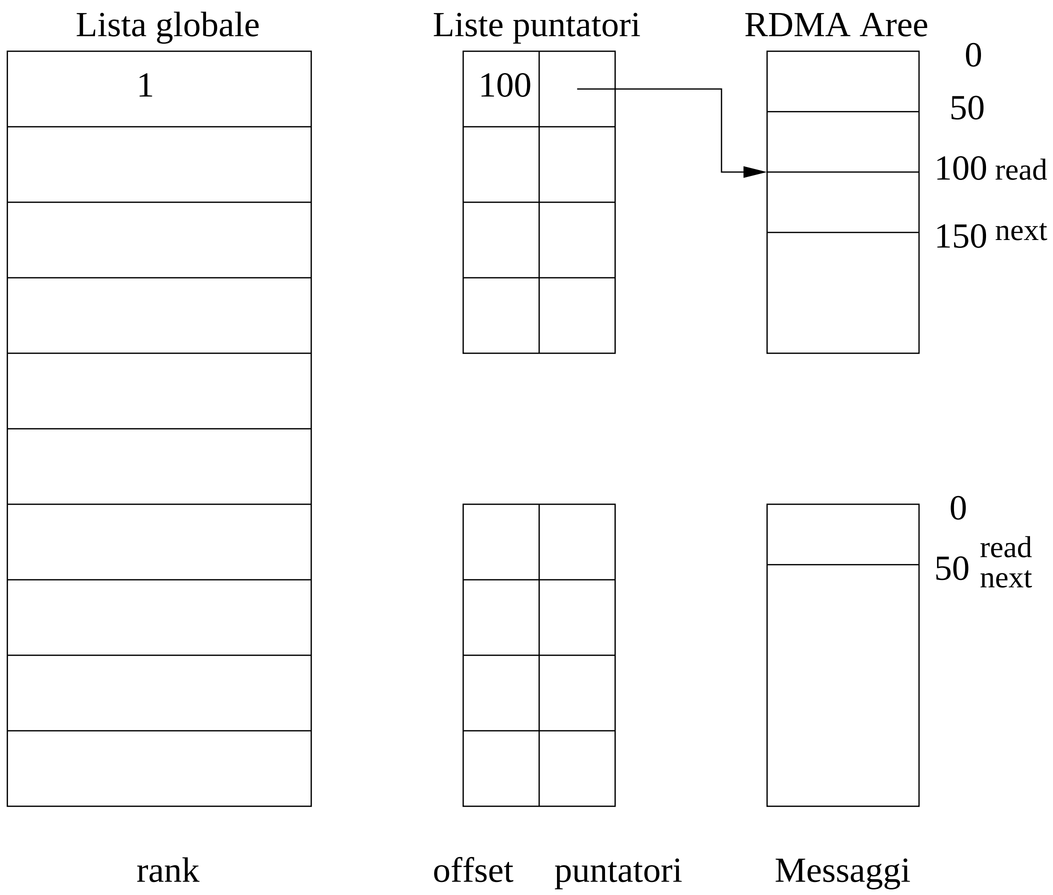

Figura 5.10: Rimane solo un messaggio.

Quando anche tale messaggio è ricevuto, le tre liste sono vuote e le RDMA aree non contengono più messaggi da leggere.

Capitolo 6

Misure e risultati sperimentali

Per mostrare le migliori prestazioni di VIA rispetto al TCP in un cluster di PC come ParMa^2, si presentano i risultati ottenuti con un semplice programma di ping-pong tra due processi.

Invece, al fine di verificare il lavoro svolto si sono utilizzati gli otto programmi contenuti nel pacchetto software dei NAS Benchmark [15] nella versione 2.3b2.

Poiché per i problemi di connessione di M-VIA (paragrafo 2.10) non è possibile eseguire più di due processi, non è stato possibile effettuare tutti i benchmark. Comunque, si ritiene che quelli svolti dimostrino sufficientemente la correttezza del lavoro eseguito.

6.1 Ping Pong

Il ping pong è un programma che coinvolge due processi la cui esecuzione si può così riassumere: il primo spedisce un messaggio, la lunghezza del quale viene variata ad ogni iterazione, e si mette in attesa di ricevere; il secondo processo, ricevuto tale messaggio, lo rimanda indietro al primo.

Misurando il tempo di attesa del primo si trova il round trip time, la cui metà è il tempo medio impiegato da un pacchetto ad essere trasmesso.

Lunghezza (Bytes)	*LAM-VIA (μs)*	*LAM-TCP (μs)*
1	57	116
2	58	117
4	58	120
8	60	120
16	59	115
32	60	131
64	72	140
128	84	151
256	108	191
512	165	243
1024	264	332
2048	393	448
4096	580	632
8192	964	1015
16384	1724	1783
32740	3243	3255

Tabella 6.1: Ping pong con send standard bloccante al variare della lunghezza del messaggio.

Se misuriamo questo dato per un pacchetto corto, otteniamo quella che, nel paragrafo 1.2, è stata definita impropriamente latenza e che consiste della latenza vera e propria e di due volte l'overhead. Misurando questo tempo per pacchetti lunghi, in modo che la latenza sia trascurabile rispetto al ritardo totale, e dividendo per la lunghezza del pacchetto otteniamo l'inverso della banda passante.

In tabella 6.1 e in figura 6.1 si presentano i dati relativi a tale prova, confrontati con i medesimi risultati ottenuti utilizzando TCP come Lower Layer di LAM.

Come si nota dai dati riportati in tabella 6.1, LAM-VIA rappresenta,

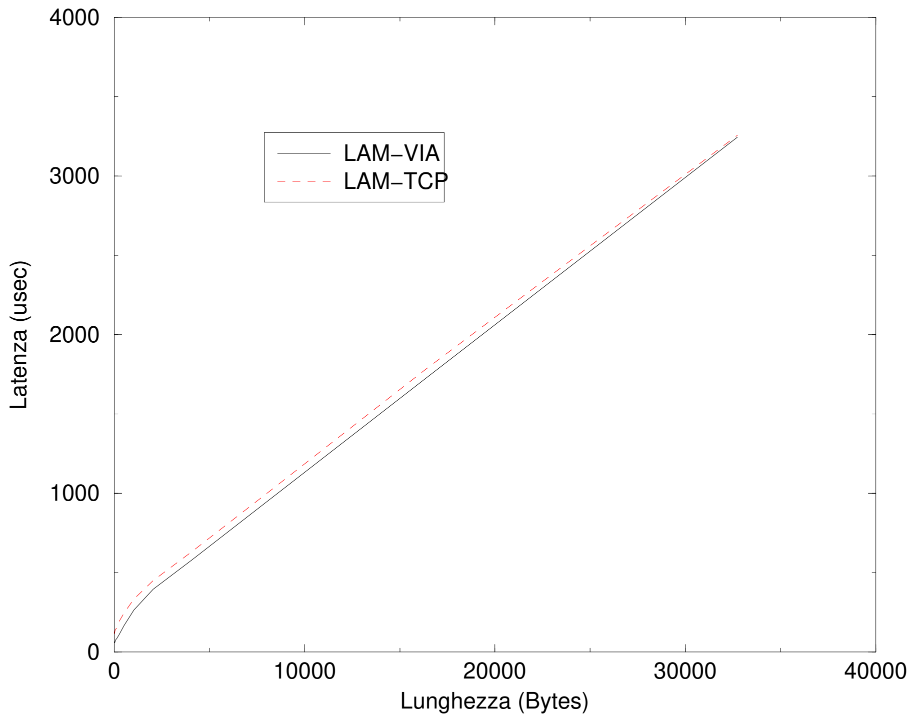

Figura 6.1: Ping pong con send standard bloccante.

soprattutto per messaggi brevi, un valido protocollo di comunicazione per librerie MPI. La latenza ottenuta sfruttando M-VIA è infatti inferiore del 51% a quella ottenuta con LAM-TCP.

6.2 NAS Benchmark

6.2.1 Descrizione

Il *Numerical Aerospace Simulation* (NAS) System Division è un gruppo specializzato nello sviluppo di tecnologie per supercomputer e di tecniche per il progetto di veicoli aerospaziali. Uno degli Obiettivi principali del NAS è sviluppare un sistema ad alte prestazioni capace di consentire progetti e missioni aerospaziali dell'ente governativo statunitense NASA.

Per questo motivo sono stati predisposti una serie di benchmark con l'obiettivo di caratterizzare un sistema parallelo. Questi programmi, derivati da modelli di fluidodinamica, sono stati progettati appositamente per valutare le prestazioni dei supercomputer paralleli; in questo caso sono stati utilizzati per valutare, oltre alle prestazioni, anche eventuali errori nella realizzazione delle funzioni di comunicazione. Infatti l'insieme di questi programmi prevede un controllo dei risultati, che non deve quindi essere fatto dall'utente, e copre quasi tutto lo spettro delle tipologie di comunicazioni previste dallo standard MPI.

Le considerazioni di cui si è tenuto conto per lo sviluppo dei benchmark sono le seguenti:

- il software deve essere sufficientemente generico in modo da non favorire alcun tipo di architettura parallela.
- La correttezza dei risultati e la rappresentazione delle prestazioni devono essere facilmente verificabili. Questo implica che l'insieme dei dati sia di ingresso sia d'uscita deve essere piccolo; inoltre, la natura dei calcoli e i risultati aspettati devono essere specificati in grande dettaglio.
- La dimensione del problema deve essere facilmente regolabile, in modo da adattarsi a nuovi sistemi aventi prestazioni maggiori.
- I benchmark devono essere scritti in un linguaggio ad alto livello, Fortran o C, per favorire la portabilità dei programmi.

Proprio per favorire al massimo la portabilità i NPB sono stati sviluppati utilizzando le primitive MPI; librerie MPI sono infatti disponibili per molte architetture.

Dopo aver valutato un certo numero di applicazioni per calcoli di fluidodinamica, sono stati scelti questi benchmark:

CG (Conjugate Gradient) calcola un'approssimazione al più piccolo determinante di una matrice definita positiva, larga, e sparsa. Verifica la comunicazione irregolare a lunga distanza.

EP (Embarassingly Parallel) misura una prestazione dei calcoli in virgola mobile (non include una significativa comunicazione inter-processo).

FT (fast Fourier Transform) soluzione parziale di equazione differenziale in tre variabili. Misura in modo rigoroso la comunicazione a lunga distanza.

IS (Integer Sort) ordina un insieme di interi. Misura sia la velocità nei calcoli su interi sia le prestazioni della comunicazione.

LU (LU solver) misura la latenza in un sistema parallelo, poiché spedisce un gran numero di messaggi brevi.

MG (Multigrid Gradient) verifica la comunicazione a lunga e a corta distanza.

BT, SP (Block Tridiagonal solver, Pentadiagonal Solver) ognuno risolve tre insiemi di sistemi disaccoppiati di equazioni. Questi sistemi sono scalari pentadiagonali nel codice SP, e blocchi tridiagonali con blocchi di 5×5 nel codice BT.

Ognuno di questi benchmark può essere eseguito con una dimensione del problema differente; dagli sviluppatori ne sono state definite cinque, la cui caratterizzazione può essere trovata nel sito web dei NPB [15]:

- S (Sample code)
- W (Workstation)
- A
- B
- C

Benchmark	*Num.Proc.*	*Classi*
BT	1	W
CG	1,2	A,W
EP	1,2	A,W
IS	1,2	W
IS	1	A
LU	1	A,W
MG	1	A,W
SP	1	A,W

Tabella 6.2: Benchmark effettuati, distinti per tipo e numero dei processi attivi.

6.2.2 Esecuzione dei benchmark e risultati

La tabella 6.2 riporta i benchmark effettuati. La tabella 6.3 riporta i benchmark che non è possibile eseguire neppure utilizzando il TCP; la tabella 6.4 quelli non eseguiti per VIA e che funzionano con il TCP e i motivi per cui ciò non è stato possibile.

Di quelli che non funzionano solo utilizzando LAM-VIA segue una descrizione più dettagliata del problema:

IS (in classe A) richiede una RDMA area più grande poiché spedisce un messaggio molto grande che riempe la RMDA area del processo cui sta spedendo prima che questo incominci a leggere.

LU, MG L'attuale stato di studio di LAM-VIA non ha ancora permesso di individuare quale sia il problema che non consente di terminare la loro esecuzione.

Non è possibile utilizzare un numero di processori superiore a due a causa dei riportati problemi di M-VIA per la connessione (paragrafo 2.10).

Nella tabella 6.5 e tabella 6.6 si riportano le misure lanciando un solo processo.

Benchmark	*Num.Proc.*	*Classi*	*Motivazioni*
BT	1	A	richiede troppo tempo
BT	2	A,W	richiedono un numero di processi quadrato (1,4,9)
FT	1,2	A,W	non compilano
SP	2	A,W	richiedono un numero di processi quadrato (1,4,9)

Tabella 6.3: Benchmark non effettuati per VIA e per il TCP e motivazioni, distinti per tipo e numero dei processi attivi.

Benchmark	*Num.Proc.*	*Classi*	*Motivazioni*
IS	2	A	si blocca perché occorrerebbe RDMA area più grande
LU MG	2 2	A,W A,W	danno segmentation fault

Tabella 6.4: Benchmark non effettuati solo per VIA e motivazioni, distinti per tipo e numero dei processi attivi.

Benchmark	*Classe W*	*Mop/s*
BT	150.05	50.98
CG	12.79	32.90
EP	37.69	1.78
IS	1.79	5.87
LU	297.15	60.78
MG	12.40	49.06
SP	289.29	49.00

Tabella 6.5: Benchmark con 1 processo.

Benchmark	*LAM-VIA*	*Mop/s*
CG	45.84	32.65
EP	301.63	1.78
IS	16.68	5.03
LU	2076.43	57.45
MG	1754.60	2.22
SP	1770.70	48.01

Tabella 6.6: Benchmark in classe A con 1 processo.

Benchmark	*Speed-up*	*LAM-VIA*	*Mop/s*
CG	1.61	7.94	26.49
EP	1.97	19.12	1.76
IS	1.33	1.35	3.90

Tabella 6.7: Benchmark in classe W con 2 processi.

Benchmark	*Speed-up*	*LAM-VIA*	*Mop/s*
CG	1.62	28.26	26.48
EP	1.97	152.91	1.76

Tabella 6.8: Benchmark in classe A con 2 processi.

In tabella 6.7 e tabella 6.8 sono riportati i dati ottenuti con due processi, per quei benchmark che lo consentono. Di essi è riportato il tempo impiegato, il numero di operazioni al secondo e lo *speed-up* rispetto ai tempi di esecuzione di un solo processo.

Capitolo 7

Conclusioni

In questo capitolo si presentano le conclusioni di questo lavoro di tesi, riguardante la realizzazione delle primitive di comunicazione per la libreria di Message Passing LAM, basate su Virtual Interface Architecture (VIA).

VIA, progettato appositamente per essere usato in reti locali, consente di ottenere prestazioni più elevate rispetto a protocolli tradizionali come il TCP, essendo capace di superare i limiti che quest'ultimo presenta quando è utilizzato nello stesso ambito. In particolare VIA evita l'intervento del kernel del sistema operativo nelle comunicazioni, consentendo l'accesso diretto alla rete da parte dei processi utente, e riduce il numero di copie in memoria essendo composto da un minor numero di livelli software. A differenza di altri protocolli che sfruttano gli stessi concetti fondamentali, rimasti limitati all'ambito della ricerca, VIA è uno standard proposto da un consorzio di decine di aziende leader nel settore tra cui Compaq, Intel e Microsoft.

Al momento della costruzione del cluster ParMa^2, disponibile presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Parma, non erano disponibili realizzazioni hardware di VIA per Linux, così si è dovuti ricorrere ad una sua emulazione software. Tra di esse è stata scelta M-VIA, il cui utilizzo ha consentito di ottenere tutti i risultati sperimentali, poiché supporta le diffuse schede Fast Ethernet. Purtroppo un problema software di M-VIA, causato da una incompatibilità hardware con il chipset di alcune schede madri Intel, rende la connessione difficoltosa. I risultati dei test hanno comunque dimostrato le ottime prestazioni che si possono ottenere in termini di latenza e ritardo di trasmissione. Al fine di migliorare tali prestazioni e risolvere i problemi di M-VIA si potrebbe ricorrere a schede di rete che realizzano VIA in hardware, che sono ormai disponibili; per una di esse, prodotta da Giganet Inc., esistono i driver per Linux.

L'analisi delle specifiche delle funzioni di comunicazione MPI si è dimostrata una fase necessaria al fine sia di definire le caratteristiche di tali funzioni sia di stabilirne la criticità rispetto all'uso di VIA come protocollo a basso livello. In particolare la tipologia di primitiva che ha richiesto la maggiore attenzione al momento della realizzazione è la receive bloccante "from any source", cioè quella che consente al processo che ne fa uso di ricevere un messaggio indipendentemente da chi lo ha spedito. VIA è un protocollo orientato alla connessione e prevede che la comunicazione si svolga tra due soli processi. Al contrario tale receive presuppone la possibilità di ricevere da chiunque senza sapere, prima che il messaggio sia arrivato, quale sia il processo che lo ha spedito; perciò la fase di realizzazione della ricezione è stata quindi guidata da questo tipo di receive.

VIA prevede le Completion Queue (CQ) come punto di raccolta delle informazioni sulle comunicazioni che riguardano un processo. L'utilizzo di tale struttura dati consente quindi di essere informati quando un descrittore su una Receive Queue di una qualunque Virtual Interface completa. Uno svantaggio delle CQ è che sono legate ad un particolare Network Interface Controller (NIC), quindi se un processo comunica con un altro sullo stesso nodo e con un terzo su un nodo diverso dovrebbe utilizzare due CQ. Questo porta ad un utilizzo pessimo del tempo della CPU, al momento di eseguire una receive bloccante "from any source", poiché il processore deve controllare alternativamente le due CQ fino a che il messaggio non è arrivato.

Poiché la CQ distrugge le informazioni relative ad un elemento quando esso è interrogato, è stato predisposto un meccanismo per memorizzare i messaggi arrivati ma non ancora passati a livello superiore. Per questo è previsto l'uso di un sistema di liste concatenate; una, i cui elementi contengono il rank del processo trasmittente, permette di mantenere un ordine globale tra i messaggi mentre un array di liste, tante quante sono i processi con cui si comunica, mantengono i puntatori ai messaggi ancora da leggere.

La spedizione di un messaggio si basa su operazioni di tipo *Remote Direct Memory Access* (RDMA) con cui il processo trasmittente specifica sia la regione di memoria che contiene il messaggio da spedire sia quella destinata a riceverlo.

A causa del problema di connessione causato da M-VIA, per misurare le prestazioni è stato utilizzato un programma di ping pong tra due processi. Il primo spedisce un messaggio, la lunghezza del quale viene variata ad ogni iterazione, e si mette in attesa di ricevere; il secondo processo, ricevuto tale messaggio, lo rimanda indietro al primo. Il ritardo di trasmissione è misurato valutando il tempo di attesa del primo processo sulla ricezione. Lo scopo è valutare latenza e banda di trasmissione; la prima si misura tramite spedizione di un messaggio di lunghezza minima, la seconda con la spedizione di un pacchetto più lungo possibile, in modo da ridurre l'incidenza della latenza sul ritardo totale. Confrontando i dati ottenuti utilizzando M-VIA e il TCP la latenza è diminuita di circa il 51%.

La correttezza del lavoro effettuato è stata verificata attraverso l'uso dei NAS Benchmark, un gruppo di programmi ormai accettato per la misura delle prestazioni di un sistema parallelo.

Bibliografia

[1] Thomas E. Anderson, David Culler, David Patterson, and the NOW team. A case for NOW (Networks of Workstations). *IEEE Micro*, feb 1995.

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

[3] Philip Buonadonna and Andrew Geweke. Berkeley via project. `http://www.cs.berkeley.edu/~philipb/via/`.

[4] George Colouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Addison-Wesley, second edition, 1994.

[5] Fujitsu Company. Synfinity cluster. `http://www.fjst.com/products/synfinitycluster/`.

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

[7] Compaq Computer Corporation. Servernet. `http://www.servernet.com`.

[8] Finisar Corporation. Synfinity cluster. `http://www.finisar.com/products/prodframe.htm`.

[9] David Culler, Richard Karp, David Patterson, Abijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eichen. LogP: towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Pratice of Parallel Programming*, San Diego, CA, may 1993.

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

[11] Giganet Inc. clan per windows nt e linux. `http://www.giganet.com/products/`.

[12] Lawrence Berkeley National Lab. Berkeley lab. via license. `http://www.nersc.gov/research/FTG/via/license.html`.

[13] Message-Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, MPI-Forum, jun 1995. `http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html`.

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

[15] NASA. Nasa parallel benchmark (NPB). `www.nas.nasa.gov/Software/NPB`.

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

[17] Portable Applications Standards Committee (PASC) sponsor of the Computer Society of the IEEE. `http://www.pasc.org/abstracts/posix.htm`.

[18] LAM Team. *Porting the LAM MPI 6.3 Communication Layer*, aug 1999.