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

UNA LIBRERIA MPI SU VIRTUAL INTERFACE ARCHITECTURE (VIA) : ESTENSIONE DELLE FUNZIONALITA' DI COMUNICAZIONE

Relatore:
Chiar.mo Prof. Gianni Conte

Correlatore:
Dott. Ing. Monica Reggiani

Tesi di laurea di:
Gherardi Andrea

ANNO ACCADEMICO 1998-99

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

La recente disponibilità di personal computer ad elevate prestazioni ha consentito la realizzazione di cluster di PC, il cui basso costo derivante da una economia di scala ha portato ad ottenere elevati rapporti prestazioni/costo. Limitazione principale di questi sistemi è la latenza introdotta nelle comunicazioni da parte dei tradizionali protocolli, come TCP/IP, sviluppati per reti eterogenee e geografiche, che presentano una struttura composta da diversi livelli software e comportano un intervento massiccio del sistema operativo. La riconosciuta inadeguatezza di questo protocollo ha focalizzato una parte della ricerca relativa alle architetture ad alte prestazioni sulla individuazione di nuove reti e metodologie di comunicazione in grado di diminuire la latenza dei messaggi. Sulla base delle promettenti idee introdotte da alcuni protocolli limitati all'ambito della ricerca, nel 1997 Microsoft, Intel e Compaq hanno proposto VIA, un nuovo standard per le comunicazioni su reti locali, in grado di limitare la latenza associata allo scambio di dati nelle applicazioni distribuite.

Presso il Dipartimento di Ingegneria dell'Informazione dell'Università di Parma, è disponibile un cluster di PC, ParMa^2, i cui risultati raggiunti precedentemente hanno dimostrato la possibilità di ottenere elevate prestazioni, nell'ordine di quelle dei sistemi dedicati. Recentemente é iniziato tra le attività legate al sistema ParMa^2, un progetto volto alla integrazione di VIA in LAM, una libreria a scambio di messaggi conforme allo standard MPI. Il risultato finale di questo progetto sarà la disponibilità di un nuovo livello di comunicazione basato su VIA.

Lo studio delle specifiche di VIA, dalle strutture dati coinvolte nella comunicazione fino alle primitive per lo scambio di messaggi tra processi, ha evidenziato i vantaggi delle scelte compiute, dovuti principalmente ad un intervento minimo del sistema operativo e ad una più snella ed efficiente struttura del protocollo. Si è perciò scelto di utilizzare M-VIA (Modular Via), una emulazione software del protocollo su schede Ethernet, prodotta dai ricercatori del NERSC presso i National Laboratories Lawrence Berkeley.

Nell'attuale fase del progetto un'urgenza riscontrata è l'impossibilità di eseguire comunicazioni con pacchetti di dimensioni che eccedono quelle previste come massime nelle specifiche di VIA. Per superare questo limite è necessario progettare un algoritmo in grado di gestire messaggi di dimensioni maggiori. L'apporto più significativo di questa tesi è consistito nel risolvere questo problema. Per mantenere la versione di LAM su VIA conforme allo standard MPI, che non prevede limitazioni di questo genere, è stato perciò realizzato un supporto in grado di suddivedere un messaggio che eccede la dimensione massima consentita, che nel caso di M-VIA è di 32KByte, in più parti e in modo trasparente all'applicazione MPI. In ricezione il problema è più complesso, in quanto il messaggio, se composto da più parti, deve essere riassemblato correttamente. Sono perciò state introdotte informazioni aggiuntive da allegare ad ogni singola parte del messaggio che ne consentono il recupero.

Per verificare l'efficienza di VIA in rapporto ai protocolli utilizzati da LAM, sono state valutate le prestazioni delle comunicazioni sul cluster ParMa^2, confrontando inizialmente la versione di LAM su VIA con quella originale, che utilizza il TCP/IP.

Per avere un ulteriore termine di paragone oltre al TCP/IP, é stata utilizzata sul cluste ParMa^{2}anche una versione della libreria MPI basata sul protocollo GAMMA, Genoa Active Message MAchine, realizzato presso il Dipartimento di Informatica e Scienza dell'Informazione dell'Universitá di Genova.

Il **secondo capitolo** contiene un'analisi dettagliata dell'architettura e delle specifiche di VIA. Sono descritti i tipi di comunicazione permessi, i diversi modelli di connessione, le modalità di trasferimento dati e le principali strutture dati coinvolte nello scambio di messaggi.

Il **terzo capitolo** descrive le caratteristiche dell'implementazione software M-VIA per Linux. Contiene anche una analisi della struttura dei principali componenti di M-VIA. Per verificare l'efficienza del protocollo, sono anche stati eseguiti dei test per valutare il ritardo di trasmissione e l'occupazione di banda nello scambio di messaggi.

Il **quarto capitolo** descrive le caratteristiche dello standard MPI e successivamente dell'ambiente di programmazione parallela LAM. Segue poi uno studio dell'architettura interna di LAM per individuare i punti di intervento per l'intergrazione di VIA al posto del TCP/IP.

Il **quinto capitolo** descrive l'implementazione di LAM-MPI su VIA. Partendo dalle scelte progettuali, si analizzano i principali interventi realizzati fino ad ottenere un confronto con TCP/IP sulle prestazioni.

Il **sesto capitolo** contiene la descrizione dell'algoritmo che permette la trasmissione e ricezione di pacchetti di dimensioni maggiori di 32Kbytes. Vengono poi analizzati i risultati ottenuti per confrontarne le prestazioni con TCP/IP.

Il **settimo capitolo** contiene un'analisi del funzionamento di GAMMA, ed in particolare della sua integrazione all'interno della libreria MPICH. Sono poi riportate le prestazioni ottenute nelle comunicazioni per un possibile confronto con quelle dell'implementazione di LAM su VIA.

Seguono infine le **conclusioni** sul lavoro svolto, sui risultati ottenuti e sui possibili sviluppi futuri del progetto.

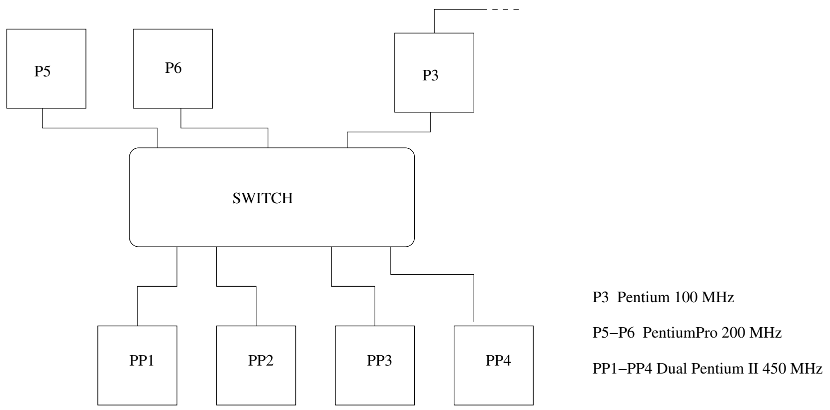

Figura 1.1: Architettura del cluster ParMa^2

1.1 Cluster ParMa^2

ParMa^{2}é un cluster di PC costruito e sviluppato presso il Dipartimento di Ingegneria dell'Informazione dell'Universitá di Parma. Il suo scopo é quello di realizzare un sistema di calcolo parallelo con un rapporto prestazioni-prezzo superione a quello delle macchine parallele dedicate. Attualmente il cluster é formato da :

- 4 nodi Dual Pentium II a 450 MHz e 256 Mbyte di RAM
- 2 nodi Pentiun Pro 200 MHz e 64 Mbyte di RAM
- 1 nodo Pentium 100 MHz e 64 Mbyte di RAM

L'architettura del cluster ParMa^{2}é riportata in figura 1.1.

Il sottocluster costituito da 2 nodi (P5 - P6) con processori Pentium Pro ha le seguenti caratteristiche:

- Processori Pentiun Pro 200 MHz
- 64 Mbyte di RAM
- Hard Disk SCSI di 2.1 GB

- Scheda di rete 3Com con chipset Digital Tulip 21140

Il sottocluster composto dai processori Dual Pentium II é invece costituito da 4 nodi omogenei (PP1 - PP4). Ogni nodo é costituito da :

- Doppio processore Pentium II a 450 MHz
- 256 Mbytes di RAM
- Scheda di rete D-LINK DFE-500TX basata su chipset DEC Tulip 21140
- Architettura SCSI con controller integrato sulla motherboard di tipo Ultra2 Wide SCSI
- Hard Disk Ultra Wide SCSI da 4.3 Gbyte

I nodi sono connessi tra loro con uno Switch Cisco Fast Ethernet.

Il Sistema Operativo che utilizzano tutti i nodi é Linux, nella distribuzione 2.0 della Debian.

Capitolo 2

VIA - Virtual Interface Architecture

VIA (Virtual Interface Architecture) [1] è uno standard per comunicazioni a bassa latenza promosso da Intel Corporation [2], Compaq Computer Corporation e Microsoft Corporation che definisce un insieme di funzioni, strutture dati e relativa semantica per trasferire dati tra processi. Il fatto che sia supportato da questi potenti promotori e che non sia solo un progetto di ricerca rende probabile che in futuro possa essere largamente utilizzato e che beneficierá delle economie di scala. VIA è progettato per permettere di ottenere vantaggi utilizzando un apposito hardware, ma può essere implementato anche interamente via software.

Il problema che VIA si propone di risolvere è quello delle prestazioni relativamente basse raggiunte dalle comunicazioni tra processi all'interno di un cluster dovute essenzialmente all'inadeguatezza dei protocolli tradizionalmente utilizzati..[3] [4]

2.1 Caratteristiche e vantaggi di VIA

Le applicazione distribuite richiedono rapidi ed affidabili scambi di informazione sia per operazioni di sincronizzazione che per trasferimenti dati. I protocolli tradizionali, come TCP/IP, sono caratterizzati da un elevato numero di livelli software che devono essere attraversati per effettuare una comunicazione, dagli interrups che devono essere generati in alcune fasi delle comunicazione, dai context switch e dalle copie di dati in memoria che avvengono, ad esempio, quando è eseguita una system call. Tutte queste caratteristiche introducono overhead, che, con il tempo di trasmissione dei dati vero e propio, costituiscono un limite alle prestazioni delle comunicazioni tra processi. Infatti il numero di livelli software da attraversare, il tempo per eseguire i context switch e gli interrupts non dipendono dal numero di bytes che devono essere trasmessi o ricevuti, mentre il tempo delle copie di dati in memoria e della trasmissione effettiva del messaggio dipendono dalle sue dimensioni. L'impatto che l'elevato overhead ha sulle trasmissioni, influisce su diversi parametri del sistema parallelo::

- Banda: mentre la larghezza di banda disponibile, negli ultimi anni, é aumentata anche di diversi ordini di grandezza, l'overhead é rimasto relativamente costante.

- Latenza e sincronizzazione: per applicazioni distribuite, l'efficiente sincronizzazione consente anche una maggiore scalabilitá. Per questo l'overhead, che contribuisce all'elevata latenza, ha effetti diretti sulle prestazioni.

- Carico del processore: l'overhead implica anche un ulteriore utilizzo della CPU, che potrebbe essere invece interamente dedicata ad altri processi.

Tutti questi problemi sono alla base della necessitá di realizzare un protocollo come VIA, che raggiunge bassa latenza ed elevata banda di comunicazione all'interno di un cluster con un utilizzo minimo della CPU. Il concetto di base è quello di offrire ad un processo utente accesso diretto all'interfacia di rete, evitando così copie intermedie di dati, context switch e l'intervento del sistema operativo e minimizzando quindi anche l'utilizzo della CPU.

2.2 Architettura di VIA

Nelle architetture tradizionali é il kernel del sistema operativo che permette la comunicazione tra due nodi all'interno di un cluster, implementando spesso anche il protocollo che rende possibile la comunicazione. Qusto modello implica peró una chiamata al kernel del sistema operativo, che puó essere relativamente dispendiosa da eseguire. VIA elimina l'overhead dovuto a questa chiamata fornendo ad ogni processo coinvolto nelle comunicazioni un interfaccia di rete virtuale (VI Virtual Interface). Ogni VI puó successivamente essere connessa ad un'altra per affettuare comunicazioni bidirezionali

Il modello architetturale di VIA é essenzialmente composto da quattro elementi (figura 2.1) :

- VI Provider
- VI Consumer
- Virtual Interface
- Completion Queue

Il **VI Provider** é costituito da due parti : il *VI NIC* e il *VI Kernel Agent*. Questi due componenti consentono la creazione di tutte le strutture dati necessarie alla comunicazione, tra cui le Virtual Interface e le Completion Queue.
Il VI NIC (*Network Interface Controller*), oltre a costruire le strutture dati, deve farsi carico del trasferimento dei dati in modo asincrono rispetto al resto del sistema, cioé senza ulteriore intervento dei componenti di VIA una volta inoltrata la richiesta di comunicazione.
Il *VI Kernel Agent* é un componente del sistema operativo utilizzato da VIA che svolge un ruolo simile al device driver ed include il sofware necessario per registrare la memoria utilizzata nella comunicazione. A differenza dei

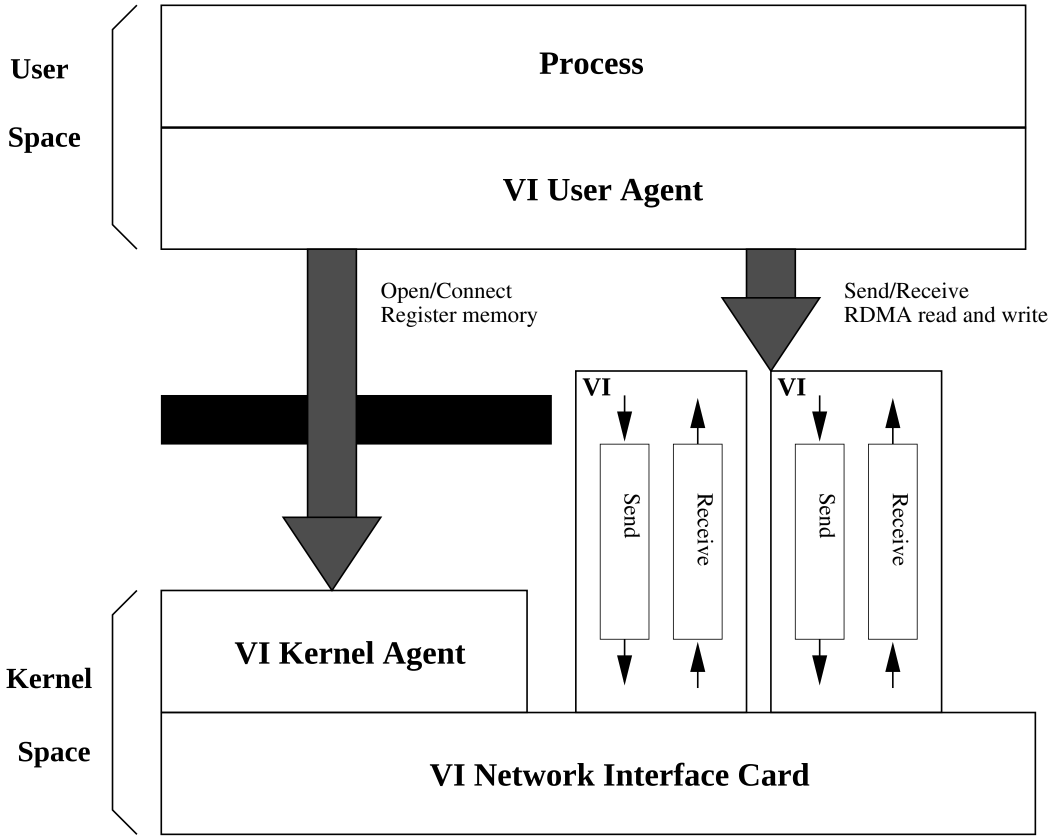

Figura 2.1: Modello architetturale di VIA.

tradizionali protocolli, l'utilizzo del kernel é quindi ridotto solo a questa fase e non alla trasmissione e ricezione dei dati vera e propria.

Il **VI Consumer** é un qualunque processo software coinvolto nella comunicazione che utilizza la Virtual Interface. Tipicamente consiste in una applicazione utente e nel VI User Agent, che é un componente software che permette le comunicazioni con il sistema operativo e che rende trasparente all'utente i dettagli del sottostante livello. E' implementato dalla VIPL (*Virtual Inteface Provided Library*), una libreria che fornisce le routines, le strutture dati e la semantica per eseguire le varie operazioni.

La **VI Virtual Interface** é un'interfaccia tra il VI NIC ed un processo utente. Permette al VI NIC di accedere direttamente alla memoria del processo creando l'illusione che questo sia proprietario dell'interfaccia di rete.

Ogni VI (figura 2.2) é costituita da due *Work Queue*: una *Send Queue* ed una *Receive Queue*. Il VI Consumer posta le richieste di trasmissione, sottoforma di *Descriptor*, su queste code. Le VI possono essere collegate tra loro tramite operazioni di connessione, che permettono ai pacchetti inviati ad una VI di essere ricevuti su un'altra. Questo rende VIA un protocollo orientato alla connessione, in quanto ogni specifica VI é connessa ad un'altra e solo da questa puó ricevere o spedire.

Tuttoció migliora la latenza perché una volta che il Kernel Agent ha stabilito la connessione, il mittente puó inviare messaggi ed il ricevente non deve verificare che la sorgente di provenienza si corretta, rimuovendo cos´í dei livelli di protezione che appesantiscono invece protocolli tradizionali.

Quando un processo posta un descrittore su una Work Queue, notifica l'operazione al VI NIC tramite la modifica di un registro chiamato *Doorbell*. Questo meccanismo di notifica deve essere protetto, cioé solo il sistema operativo puó creare una doorbell e successivamente solo il VI NIC la utilizza per risalire al proprietario della VI, senza fare intervenire il kernel del sistema operativo, effettano cos´í lo scambio di dati in modo asincrono.

VIA fornisce anche un meccanismo di sincronizzazione tra il processo

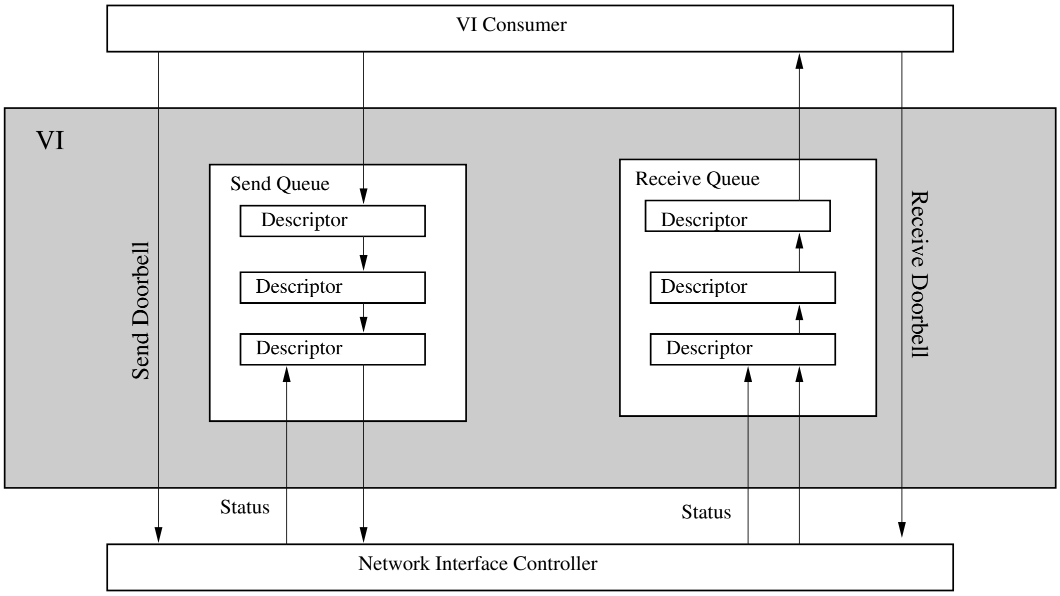

Figura 2.2: Virtual Interface.

utente e la notifica di un completamento. Quando un Descrittore é completato, il VI NIC ne setta un particolare bit e ne restituisce il controllo al processo utente. Per consentire un sincronismo tra queste fasi, si utilizza una tecnica di polling, che consiste nel testare la cima della Work Queue della VI fino a che il Descrittore presente non é stato marcato completo, o una tecnica bloccante che, quando il descrittore in cima alla coda é completato, lo rimuove e ne restituisce l'indirizzo.

La **Completion Queue** é una strutture che contiene informazioni relative ai Descrittori completati. Quando é eseguita una operazione di trasferimento dati, nella Completion Queue sono inserite informazioni che permettono di determinare in quale VI é contenuto il Descrittore completato. Sono utilizzare per avere un unico punto di notifica di completamento per diverse VI.(figura 2.3)

Se una Work Queue di una VI é associata ad una Completion Queue per notificare quando un descrittore é completato, la sincronizzazione, e quindi

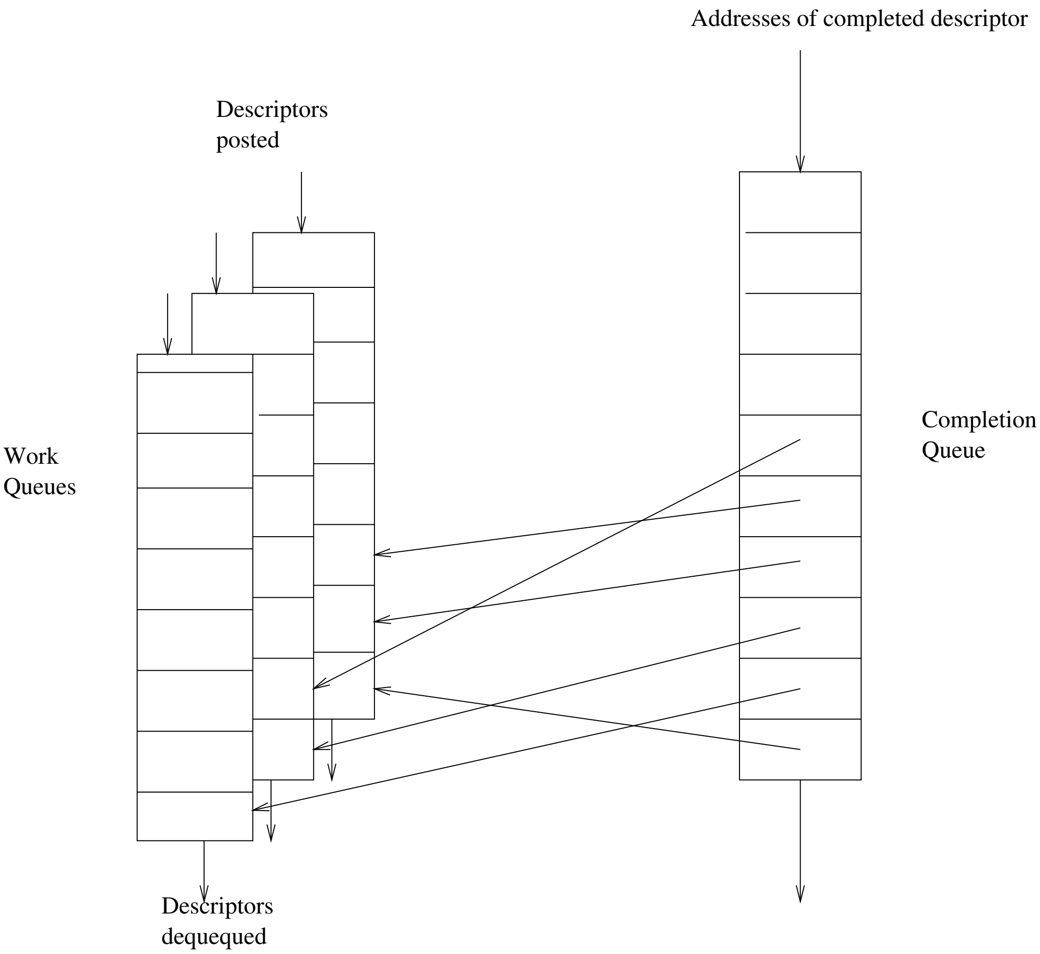

Figura 2.3: Completion queue

il testaggio della cima della coda o tramite polling o in modo bloccante, non viene piú fatto sulla VI, ma sulla Completion Queue.

2.3 Gestione della memoria

La memoria utilizzata dalle strutture dati di VIA coinvolte nelle comunicazioni, come i Descrittori, e dai buffer di comunicazione deve essere preventivamente registrata per essere bloccata nella memoria fisica. Rispetto ad un trasferimento dati tradizionale, in cui questa operazione viene fatta per ogni richesta di trasmissione e intoducendo cos´í overhead. La registrazione preventiva consente di eliminare questo overhead dalla fase di comunicazione dei dati.

Inoltre la registrazione della memoria abilita il VI Provider a trasferire i dati senza copia in buffer intermedi.

Il processo di registazione viene richiesto dal VI Consumer al VI Privider utilizzando opportune primitive. Quest'ultimo si riserva di bloccare una o più pagine nella memoria fisica, notoficarlo al Kernel Agent e restituire un riferimento (*handle)* alla regione di memoria.

VIA fornisce dei meccanismi di protezione della memoria per tutte le operazioni che riguardano le VI, per evitare che un processo utente acceda a delle zone che non gli appartengono. Questi meccanismi consistono in un **protection tag** e in una protection table (TPT), che contiene gli **attributi di protezione** della memoria.

I tag di protezione della memoria sono degli identificativi univoci, costituiti da un numero intero, che sono associati dal Kernel Agent sia ad una VI che alle regioni di memoria. Il VI Consumer, prima di creare una VI o di registare una zona di memoria, deve ottenere dal VI Provider un protection tag, che utilizzerà poi come parametro per la richiesta effettiva di creazione di una VI o di registrazione della memoria.

Nel caso di creazione di una VI, il Kernel Agent abiliterà l'accesso ai dati solo se il tag con cui questi sono stati registrati corrisponde a quello della VI. Se il confronto tra i tags da esito negativo si ha un errore di protezione della memoria e nessuna VI viene creata.

Quando invece una applicazione registra una zona di memoria, deve specificare il tag che verrà associato alla zona di memoria. Quando poi un processo tenta di accedervi, il Kernel Agent confronterà il tag con quello con cui la zona di memoria era stata registrata, in modo da permettere o meno l'accesso.

Gli attributi di protezione della memoria, come i protection tags, sono associati sia alle regioni di memoria che alle VI.

Nel primo caso servono per controllare il tipo di operazione che può essere svolto su una zona di memoria. Ad esempio se la memoria è stata registrata per permettere accessi in sola lettura, vengono rifiutate operazioni di RDMA Write da parte di un processo remoto.

Gli attributi di protezione della memoria sono utilizzati anche in fase di connessione di due VI, in modo da permettere il collegamento solo se due VI hanno gli stessi attributi.

Un altro importante compito svolto dal Kernel Agent durante la registrazione della memoria e quello di consentire al VI NIC la traduzione degli indirizzi virtuali in indirizzi fisici.

Quando un utente richiede la registrazione di una zona di memoria, il Kernel Agent esegue il controllo di proprietà, blocca le pagine nella memoria fisica ed inserisce come entry nella tabella di traduzione e protezione della memoria l'indirizzo fisico delle pagine di memoria che ha bloccato.

Quello che ritorna all'utente è il *memory handle*, una struttura dati che l'utente può utilizzare per far riferimento alla regione di memoria senza preoccuparsi della sua reale collocazione.

La registrazione della memoria può fallire se il VI Provider non è in grado di trovare una entry nella tabella di protezione o se la dimensione del buffer di memoria che si vuol registrare è troppo elevato per essere contenuto nella memoria fisica.

Quando un'applicazione non utilizza più una data zona di memoria, questa deve essere deregistrata per consentire al Kernel Agent di rimuovere l'entry corrispondente nella tabella di protezione e sbloccare così le pagine in memoria fisica.

2.4 Modelli di connessione

Il modello di comunicazione di VIA è orientato alla connessione, in quanto ogni VI deve essere collegata ad un altra per poter effettuare lo scambio di dati.

I modelli di connessione che VIA rende disponibili sono due : **client-server** e **peer-to-peer**. [1]

Questi due modelli risultano essere del tutto distinti, e due applicazioni che desiderano comunicare devono utilizzare lo stesso modello indifferentemente.

2.4.1 Modello Client-Server

Questo modello fornisce unicamente una semantica bloccante. Sia dal lato client che dal lato server, il VI Consumer deve effettuare una richiesta al VI Provider per poter connettere la propria VI con una VI remota. Il processo di connessione si svolge attraverso le seguenti fasi :

- Il server si pone in attesa di una connessione da un client.
- Arrivata la richiesta di connessione da parte di un client, questa viene analizzata per vedere se gli attributi specificati sono accettabili.

[1] Col termine peer-to-peer si intende qualunque modello in cui non si riscontrino le caratteristiche gerarchiche del modello client-server.

- Se la richiesta puó essere accettata, il server crea una VI locale che sará un estremo della connessione.

- Quando il client ha terminato la richiesta, e questa a dato esito positivo da parte del server, puó essere stabilita la connessione vera e propria tra le due VI.

La richesta é di tipo bloccante perché il client deve bloccarsi in attesa della risposta del server sul tentativo di connessione. Lo schema che riporta la successione di queste fasi é riportato in figura 2.4. La prima istruzione che il server deve eseguire é del tipo Connection Wait per porsi in attesa di connessioni provenienti da uno specificato indirizzo per un determinato periodo di tempo, trascorso il quale é inoltrato un messaggio di errore. Successivamente il client inoltra una Connect Request, che il server puó accettare o meno dopo una analisi degli attributi della richiesta.

A questo punto se il server decide di accettare la connessione, deve creare una VI con le caratteristiche opportune, in modo che corrisponda a quella del client, e chiamare una Connect Accept. Se le caratteristiche della VI locale e della VI remota non corrispondono, la Connect Accept non viene accettata ed il server rifiuta la connessione con Connect Reject. Se invece le caratteristiche delle due VI corrispondono, la Connect Accept ritorna senza produrre messaggi di errore e le due VI sono connesse.

Una volta terminato l'utilizzo delle VI, queste devono essere disconnesse. Il VI Consumer effetua una richiesta di Disconnect al proprio VI Provider, producendo cosí una caduta unilaterale della connessione.

Durante queste fasi, la VI che viene inizialmente creata ed una volta completato il suo utilizzo distrutta, passa attraverso diversi stati che ne caratterizzano il comportamento. In figura 2.5 é riportato un diagramma di stato che rappresenta il ciclo di vita di una VI.

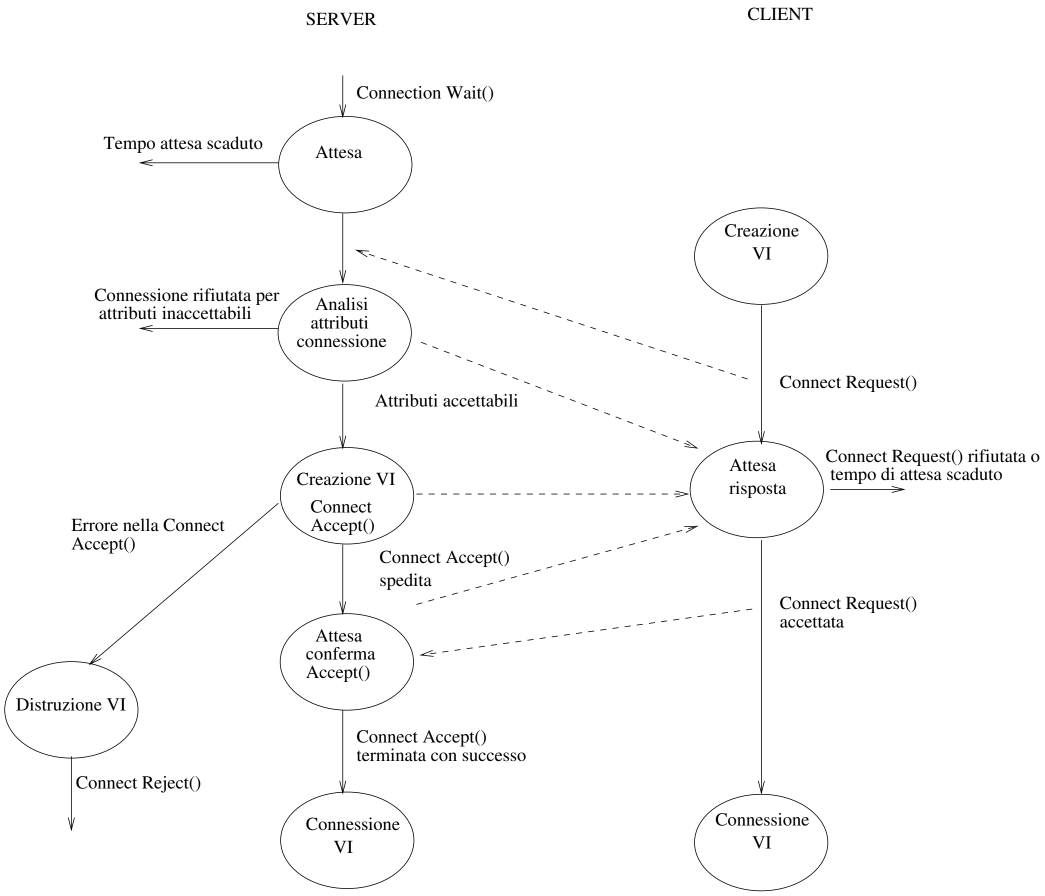

Figura 2.4: Fasi della connessione nel modello client-server

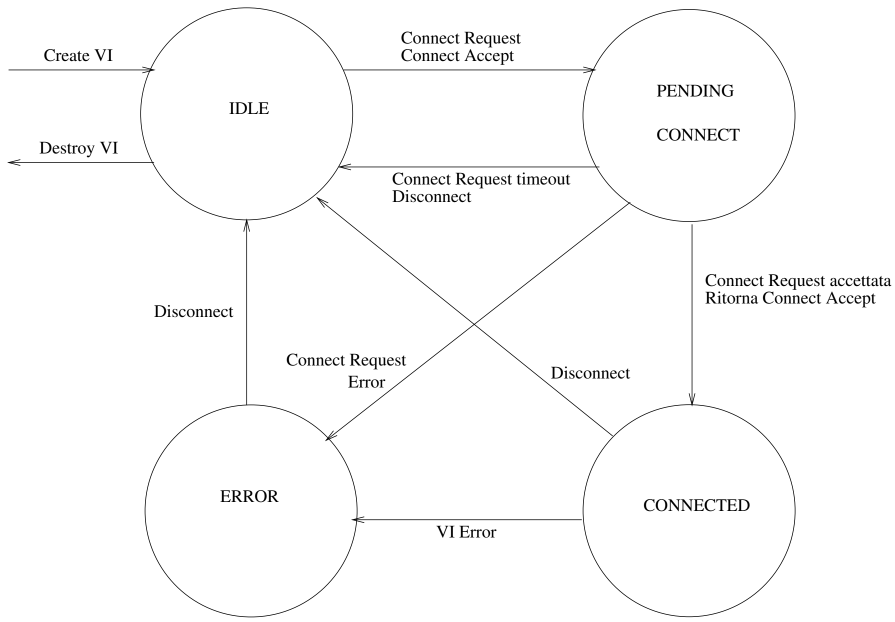

Figura 2.5: Diagramma di stato di una VI

2.4.2 Modello Peer-to-Peer

Al contrario del modello client-server, questo fornisce sia una semantica bloccante che non bloccante. Questa differenza si pone quando il processo che richiede la connessione decide se controllare lo stato della richiesta con un test non bloccante o se porsi in attesa che la fase di inizializzazione della connessione abbia termine.

Il modello di connessione peer-to-peer é riportato in figura 2.6.

Inizialmente uno dei due peer richiede la connessione tra una VI locale ed una remota con Connect Peer Request. Successivamente, prima che scada il timeout, anche l'altro peer effetua una Comnnect Peer Request, che é confrontata con quella esistente dall'altro lato della connessione per poi inviare, in caso di confronto positivo, un acknowledged. Quindi il risultato della Connect Peer Request é testato con Connection Peer Done, che non blocca il processo, o con Connect Peer Wait, che blocca invece il processo,

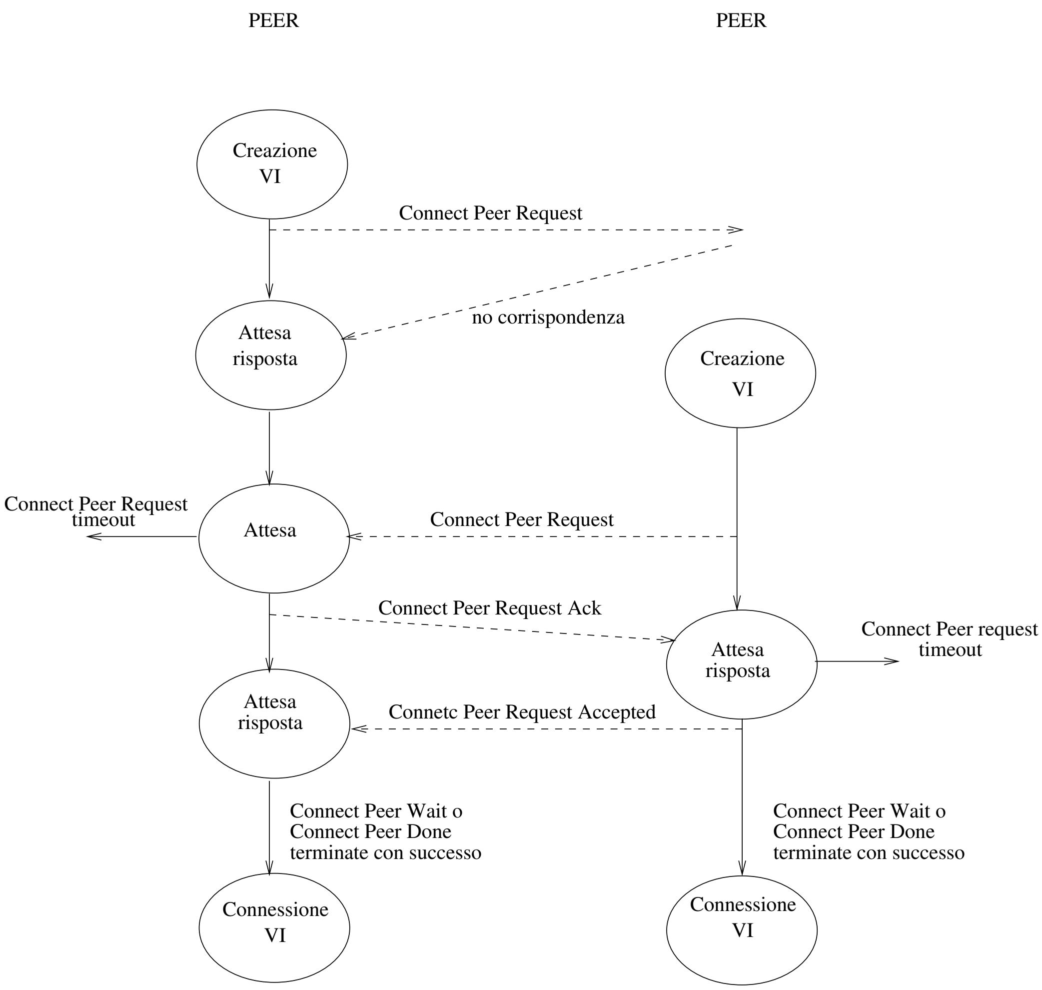

Figura 2.6: Fasi della connessine nel modello peer-to-peer

e se la connessione é stabilita con successo, le due VI transitano nello stato Connect.
La fase di disconnessione del modello peer-to-peer é identica a quella del modello client-server.

2.5 Modalità di trasferimento dati

Per eseguire una operazione di trasferimento dati, una applicazione deve postare dei Descrittori su una delle due Work Queue di una VI, che deve essere precedentemente stata connessa con una VI appartenente al processo remoto. Questi Descrittori vengono poi processati in modo asincrono e successivamente, quando l'operazione è conpletata, vengono rimossi dalle code per essere poi riutilizzati nelle richieste di trasmissione successive. Nella comunicazione può anche essere coinvolta la Completion Queue, necessaria se si utilizzano più VI e si vuole risalire al processo proprietario del Descrittore completato.

VIA consente di utilizzare due modalità per eseguire operazioni comunicazione :

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

2.5.1 Send/Receive

Ogni operazione di spedizione o di ricezione di dati avviene postando un Descrittore sulla relative Work Queue di una VI. Il VI Consumer deve specificare un solo indirizzo di memoria virtuale all'interno del Descrittore, che, nel caso di ricezione, indica la zona che conterrà i dati in arrivo, mentre nel caso di invio di dati indica la regione di memoria che contiene i dati da spedire.

Per controllare il completamento di un Descrittore, e quindi l'esito della comunicazione, VIA permette di usare una semantica sia bloccante che non.

Nel caso **bloccante** l'applicazione controlla se il Descrittore in cima alla coda è stato marcato completo. Se ciò è avvenuto, il Descrittore è rimosso dalla coda ed il suo indirizzo è ritornato al processo chiamante. Se invece non è ancora stato completato, il processo chiamante si blocca fino a completamento avvenuto o fino a che non è trascorso un tempo massimo specificato come parametro quando si chiama la funzione.

Nel caso di semantica **non bloccante**, si controlla sempre il Descrittore in cima alla coda. Se questo è marcato completo, viene rimosso ed è ritornato il suo indirizzo al processo chiamante. Se invece non è ancora stato completato, la funzione ritorna un codice di errore e la cima della coda non viene modificata.

In ogni caso prima di iniziare una comunicazine, è necessaria una accurata fase di sincronizzazione tra i due processi coinvolti. Infatti se ad esempio un processo che voglia inviare dei dati non trova nessun Descrittore postato sulla Receive Queue del processo remoto, viene generato un errore, così come se il buffer del ricevente ha dimensioni inferiori a quello del destinatario.

Nella comunicazione può essere coinvolta anche la Completion Queue. In questo caso per verificare il completamento di un Descrittore si deve testare la cima della CQ, utilizzando, come per una VI, una primitiva bloccante o non. Il comportamento è del tutto analogo, con l'eccezione che il Descrittore completato è rimosso solo dalla CQ e non dalla VI. Inoltre se un processo che ha una CQ associata ad una propria VI desidera bloccarsi in attesa del completamento di un Descrittore, non lo può fare sulla VI, ma deve bloccarsi sulla CQ.

2.5.2 Remote Direct Memory Access

La diversità principale tra questo tipo di modalità e la precedente è che il processo che intente trasmettere, oltre all'indirizzo della zona di memoria che contiene i dati, specifica anche l'indirizzo della zona di memoria di destinazione dei dati.

Inoltre in questo caso il modello di comunicazione è di tipo asincrono, in quanto il processo che effetua questa operazione non necessita di un Descrittore postato sulla coda del processo remoto perchè questo non verrebbe comunque consumato. Ciò comporta quindi che non ci sia una notifica del completamento della comunicazione.

Prima di questo tipo di operazione c'e una fase di scambio, tramite send/receive, degli indirizzi delle zone di memoria remote dei due processi, che devono sapere preventivamente dove andare a scrivere o leggere i dati.

Le operazioni RDMA possono essere di due tipi : **RDMA Write** e **RDMA Read**.

Nel caso di RDMA Write, nel Descrittore postato sulla Send Queue della VI è specificato, oltre all'indirizzo di sorgente del buffer contenente i dati, anche quello remoto destinato a contenere i dati in arrivo. Inoltre nessun Descrittore è consumato sulla Receive Queue del processo remoto, tranne nel caso di trasmissione RDMA con **Immediate Data**. Questa modalità è utilizzata per avere notifica di avvenuta ricezione anche nella modalità RDMA. Ciò avviene consumando un Descrittore che deve essere preventivamente postato sulla Receive Queue del processo remoto. In questo caso il modello di comunicazione è sincrono.

Nel caso di RDMA Read, nel Descrittore postato sulla Send Queue della VI è specificato, oltre all'indirizzo del buffer remoto che contiene i dati da leggere, anche l'indirizzo di una regione di memoria locale, che sarà destinata a contenere i dati letti sul nodo remoto. Come nel caso precedente, nessun Descrittore è consumato sul nodo remoto. Inoltre il modello di comunicazione è sempre asincrono perchè operazioni di RDMA Read con Immediate Data non sono permesse.

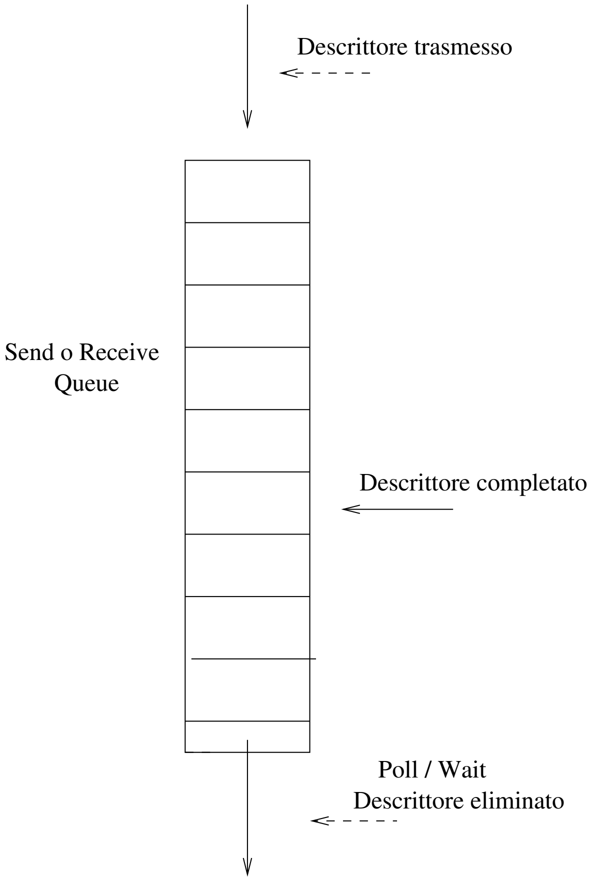

Figura 2.7: Descrittore consumato senza Completion Queue

2.6 Descrittori

Il Descrittore é quella struttura dati che viene postata nella VI e che quindi rappresenta una richiesta di trasferimento dati. Contengono tutte le informazioni necessarie al completamento di una richiesta, come il tipo di traferimento da fare, informazioni sulle code o eventali Immediate Data.

Un Descrittore puó essere processato in due diversi modi dal VI Consumer, a seconda che si utilizzo o meno la Completion Queue. Nel caso che questa non venga utilizzata, mostrato in figura 2.7, quando il Descrittore é completato, viene rimosso dalla Work Queue dal VI Consumer che era in

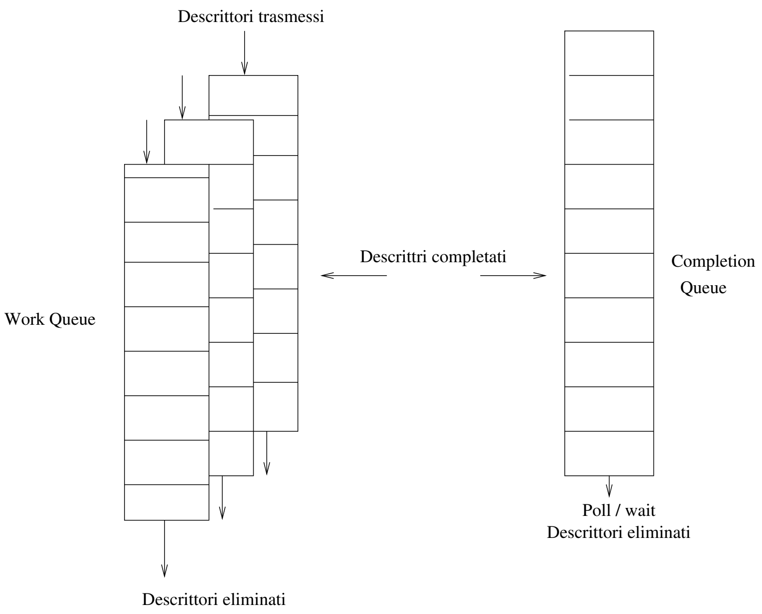

Figura 2.8: Descrittore consumato con Completion Queue

attesa del completamento o era in fase di polling della cima della coda. Nel caso invece si utilizzi la Completion Queue, mostrato in figura 2.8, quando il Descrittore é completato viene inserito nella Completion Queue e successivamente rimosso dal VI Consumer che era in attesa o in fase di polling sulla cima della stessa.

I Descrittori sono composti da tre tipi di segmenti:

- Control Segment
- Addres Segment
- Data Segment

Il **Control Sagment** é il primo componente del Descrittore e contiene informazioni riguardanti il tipo di trasmissione dati che deve essere eseguita, gli Immediate Data se presenti, il numero di segmenti che seguono, l'indirizzo

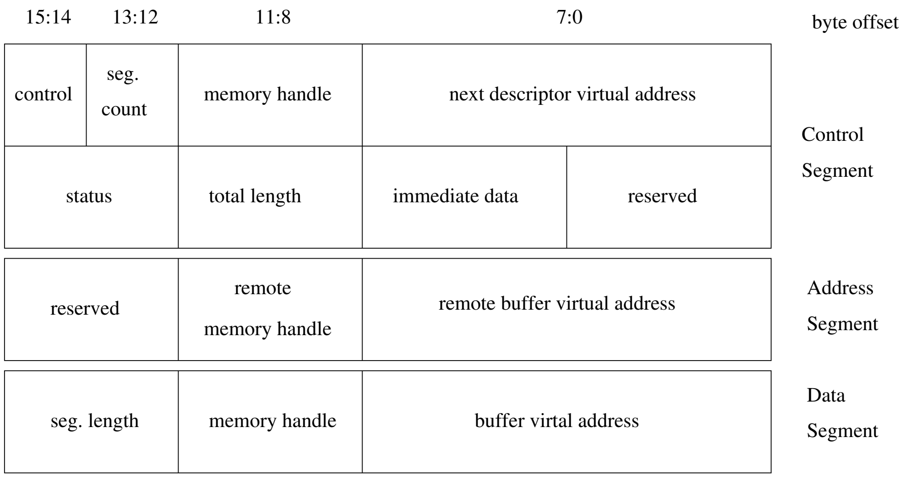

Figura 2.9: Descrittore per operazioni send e receive

virtuale del Descrittore che segue nella coda e il suo riferimento in memoria, lo stato del Descrittore e la lunghezza totale in bytes dei dati che devono essere trasmessi.
L'**Address Segment** segue il Control Segment solo per operazioni RDMA read o RDMA write. Questo contiene informazioni relative all'indirizzo virtuale della regione di memoria contenente il buffer remoto.
Il **Data Segment** contiene invece informazioni relative al buffer locale per operazioni di send e receive e per operazioni RDMA Write e RDMA Read. Un Descrittore puó contenere piú Data Segment.

In figura 2.9 é mostrata la struttura di un Descrittore per operazioni di send e receive, mentre in figura 2.10 il Descrittore usato per operazioni di RDMA.

2.6.1 Control Segment

Sono descritti i campi del Control Segment ed il loro contenuto.
Next Descriptor Virtual Address (bytes 7:0)

Questo campo contiene l'indirizzo del Descrittore successivo, in modo da

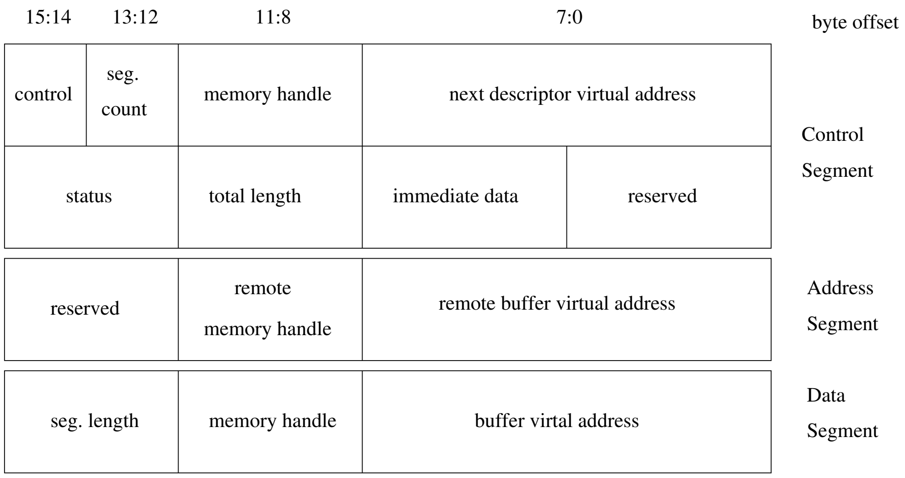

Figura 2.10: Descrittore per operazioni RDMA

collegare tra loro una serie di Descrittori e formare cosí la coda della VI. Il VI User Agent puó inserire in questo campo del Descrittore in fondo alla coda l'indirizzo di un nuovo Descrittore per inserirlo nella coda.

Next Descriptor Memory Handle (bytes 11:8)

Contiene il riferimento in memoria all'indirizzo del campo precedente. E' riempito dal VI User Agent quanto é inserito un nuovo Descrittore nel campo precedente.

Descriptor Segment Count (bytes 13:12)

Contiene il numero di segmenti che seguono il Control Segment, incluso l'Address Segment se presente. Una generica applicazione setta questo campo quando inizializza il Descrittore.

Control Field (bytes 15:14)

Questo campo contiene dei bit di controllo o delle informazioni che indicano le azioni specifiche che devono essere eseguite dalla VI quando processa il Descrittore. Come il campo precedente, é settato da una generica applicazione quando il Descrittore é inizializzato. Contiene dei sottocampi che hanno il seguente significato:

- bit 1:0 :Operatin Type

  Definiscono le operazioni da eseguire con questo Descrittore. I valori ammessi sono :
  - *00* : indica che si tratta di una operazione di Send se il Descrittore si trova sulla send queue della VI, mentre indica una operazione di Receive se il descrittore é postato sulla receive queue della VI.
  - *01* : indica che si tratta di una operazione di RDMA Write se il Descrittore é postato sulla send queue. Questo valore non é valido se si trova sulla receive queue ed il formato del Descrittore risulterá errato.
  - *10* : indica che si tratta di una operazione di RDMA Read se il Descrittore é postato sulla send queue. Questo valore é valido solo se il VI Provider supporta questo tipo di operazione. Il valore non é valido se il Descrittore é postato sulla receive queue ed il suo formato risulterá non valido.
  - *11* : questo valore non é definito dalle specifiche ed il formato del Descrittore risulterá errato.

- bit 2 :Immediate Data Indication

  Se questo bit é settato ad 1, significa che il valore nel campo Immediate Data é corretto. Quindi se il Descrittore é postato sulla send queue significa che il dato in Immediate Data é stato trasferito nel corrispondente Descrittore dall'altro lato della connessione, mentre se l'operazione da eseguire é del tipo RDMA Write il significato é lo stesso, ma comporta il completamento del Descrittore dall'altro lato della connessione, che invece non avverrebbe con normali operazioni di RDMA. Questo bit é inoltre ignorato per operazioni di RDMA Read, che non consentono trasmissioni con Immediate Data. Ció comporta che la trasmissione venga ignorata senza ritornare un errore.

- bit 3 :Queue Fence

Questo bit, se settato, fa completare tutte le precedenti operazioni di RDMA Read sulla stessa coda del Descrittore prima di completarlo.

- bit 15:4 :Reserved

  Questi bit sono riservati per utilizzi futuri. Devono essere settati a zero dalla generica applicazione perché il formato del Descrittore non sia errato.

Reserved (bytes 19:16)

Questo campo é riservato. Deve essere settato a zero perché il formato del Descrittore sia corretto.

Immediate Data (bytes 23:20)

Questo campo permette che 32 bit di dati siano trasferiti con una operazione di Send o di RDMA Write al corrispondente Descrittore postato nella receive queue. Il valore del Control Descriptor deve essere corretto per permettere questo tipo di trasmissione. Questo campo viene settato da una applicazione per operazioni di Send o RDMA Write, mentre é ritornato alla applicazione nel caso di Receive. E' ignorato nel caso di operazioni RDMA Read.

Length Field (bytes 27:24)

Questo campo contiene la lunghezza totale del buffer di dati che si vuole trasmettere con il Descrittore, esclusa la dimensione del campo Immediate Data. E' inizializzato dalla applicazione nel caso di operazioni di Send, mentre non é definito per operazioni di Receive. E' il VI NIC che, utilizzando il parametro del Data Segment, determina il valore da dare a questo campo. Se il Descrittore viene completato ritornando un errore, questo campo é indefinito.

Status (bytes 31:28)

Contiene le informazioni scritte dal VI NIC necessarie al completamento del Descrittore. Il VI User Agent si accorge dell'avvenuto completamento

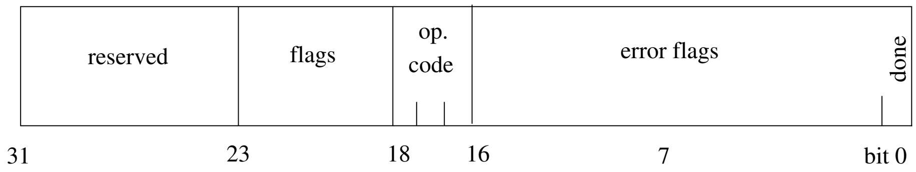

Figura 2.11: Formato del campo Status

testando questo campo, che deve quindi essere settato a zero prima che il Descrittore venga postato nella Work Queue. Il formato di questo campo é mostrato in figura 2.11.

L'utilizzo dei singoli bit é di seguito illustrato :

- bit 0 : done

  Quando questo bit é settato ad uno dal VI Provider, significa che il Descrittore é stato completato. Se i bits da 1 a 15 sono settati a zero, significa che il completamento é stato corretto, altrimenti qualche errore é stato rilevato.

- bit 1 : Local Format Error

  Questo bit é settato ad uno se il Descrittore postato nella coda locale ha un formato errato dovuto ad operazioni non valide o a settaggi errati da parte dell'applicazione di campi riservati. Non include peró errori coperti dai bit successivi.

- bit 2 : Local Protection Error

  Se questo bit é settato ad uno, significa che il Descrittore puó contenere un errato riferimento in memoria, o un indirizzo virtuale errato o che uno o piú Data Segment non sono accessibili a causa di un tentativo di accesso a zone di memoria protette.

- bit 3 : Local Length Error

Questo bit é sattato ad uno nelle seguenti condizioni:
- il campo Segment Count eccede le capacitá del VI Nic
- per Descrittori postati sulla Send Queue se il campo Total Length eccede la dimensione massima del messeggio da trasferire o non corrisponde alla somma delle lunghezze dei Data Segment
- per Descrittori postati sulla Receive Queue se il campo Total Length contiene un valore troppo basso per contenere il buffer in arrivo o la dimensione di questo eccede la dimensione massima del messaggio trasferibile.

- bit 4 :Partial Packet Error

  Per un Descrittore postato sulla Send queue, questo bit é settato ad uno quando é rilevato un errore dopo che solo una parte del trasferimento dei dati é avvenuta. Per Descrittori sulla Receive Queue indica che é stato ricevuto un pacchetto danneggiato o incompleto.

- bit 5 : Descriptor Flushed

  Questo bit indica che il Descrittore é stato tolto dalla coda quando la VI é stata disconnessa o dall'applicazione o a causa di un errore.

- bit 6 : Transport Error

  Questo bit é usato per indicare che c'é stato un errore nel trasferimento, come la perdita dei dati. Per i livelli di affidabilitá Reliable Delivery e Reliable Reception, in cui gli errori in trasmissione devono essere notificati, la VI transita per uno stato di errore, mentre per Unreliable Delivery questo bit é valido solo nel Descrittote sulla Receive Queue.

- bit 7 : RDMA Protection Error

  Questo bit é settato ad uno se la sorgente di dati per operazioni di RDMA Read o la destinazione per RDMA Write ha rilevato un errore di protezione al nodo remoto : ció puó avvenire per errati indirizzi o riferimenti in memoria. Per il livello di affidabilitá Unreliable Delivery, questo bit é ignorato, per Reliable Delivery é settato solo per Descrittori utilizzati in operazioni di RDMA Read, mentre per Reliable Reception é settato sia per operazioni RDMA Read che RDMA Write.

- bit 8 : Remote Descriptor Error

  Questo bit é settato ad uno se c'é stato un qualsiasi tipo di errore nel Descrittore postato sul nodo remoto o se non c'é nessun Descrittore sulla Receive Queue del nodo remoto dopo una operazione di send.

- bit 15:9 : Reserved Error Bits

  Questi bit sono riservati per utilizzi futuri e devono essere settati a zero dal VI Provider.

- bit 18:16 : Completed Operation Code

  Questo campo descrive il tipo di operazione completata per il Descrittore. Il bit meno significativo (LSB) indica la coda su cui l'operazione é stata completata : 0 indica Send Queue e 1 indica Receive Queue. I valori possibili sono :
  - 000 : Send completata
  - 001 : Receive completata
  - 010 : RDMA Write completata
  - 011 : RDMA Remote Write completata (indica che una RDMA Write iniziata da un nodo remoto completa e consuma questo Descrittore)
  - 100 : RDMA Read completata
  - da 101 a 111 non sono definiti

- bit 19 : Immediate Flag

  Questo bit é settato ad uno quando il campo Immediate Data é valido per un Descrittore postato sulla Receive Queue.

- bit 31:20 : Reserved Flag Bits

Questi bit sono riservati per utilizzi futuri e devono essere settati a zero dal VI Provider.

2.6.2 Address Segment

E' il secondo segmento di un Descrittore ed é incluso solo per operazioni di RDMA. Lo scopo di questo segmento é quello di notificare al VI NIC l'indirizzo di memoria del nodo remoto in cui i dati devono essere scritti o da cui devono essere letti. L'indirizzo di memoria ed il corrispondente riferimento risiedono in una zona di memoria registrata dal processo a cui é associata la VI remota e devono essere noti al processo locale prima dell'operazione di RDMA.

Remote Buffer Virtual Address (bytes 7:0)

Per una operazione di RDMA Write, questo campo contiene l'indirizzo del buffer di destinazione del nodo remoto della connessione. Per una RDMA Read specifica il buffer sorgente sempre del nodo remoto.

Remote Buffer Memory Handle (bytes 11:8)

Contiene il riferimento in memoria corrispondente al campo precedente.

Reserved (bytes 15:12)

Questi bit sono riservati per utilizzi futuri e devono essere settati a zero dal VI Provider.

2.6.3 Data Segment

All'interno di un Descrittore possono esistere da zero a piú Data Segment: ogni VI NIC deve poter far riferimento al almeno 256 Data Segment per ogni Descrittore, mentre se nessun Data Segment é contenuto, si possono eseguire solo operazioni con Immediate Data. La lunghezza totale dei buffer dei Data Segment di un singolo Descrittore non deve eccedere la dimensione del pacchetto massimo trasferibile.

Local Buffer Virtual Address (bytes 7:0)

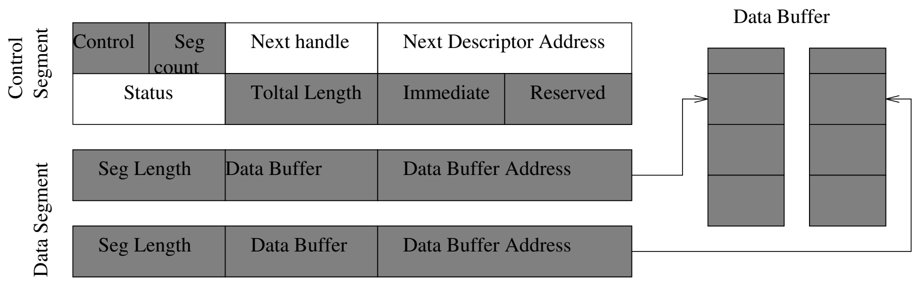

Figura 2.12: Campi del Descrittore aggiornati dalla VI Application

Contiene l'indirizzo dei dati descritti dal segmento. Deve essere inizializzato dalla generica applicazione.

Local Buffer Memory Handle (bytes 11:8)

Contiene il riferimento in memoria corrispondente al campo precedente.

Local Buffer Length (bytes 15:12)

Contine la lunghezza del buffer indirizzato dal campo Local Buffer Virtual Address.

2.6.4 Utilizzo del Descrittore all'interno dei componenti di VIA

Durante una comunicazione, la proprietà del Descrittore passa dalla generica applicazione alla VI Library (VIPL) e quindi al VI NIC, senza passare attraverso il Kernel Agent consentendo così l'eliminazione del sistema operativo dal percorso critico. Quando la comunicazione viene completata, il Descrittore ritorna all'applicazione ripassando per la VI Library. Inizialmente la **VI Application** inizializza alcuni campi del Control Segment del Descrittore, figura 2.12, ed in particolare il Control, che indica il tipo di operazione richiesta, il Segment Count, che indica il numero di parti in cui è organizzato il Descrittore, Total Length, che indica la dimensione in bytes del buffer di dati ed infine setta a zero i byte riservati. Opzionalmente può modificare anche il campo Immediate Data.

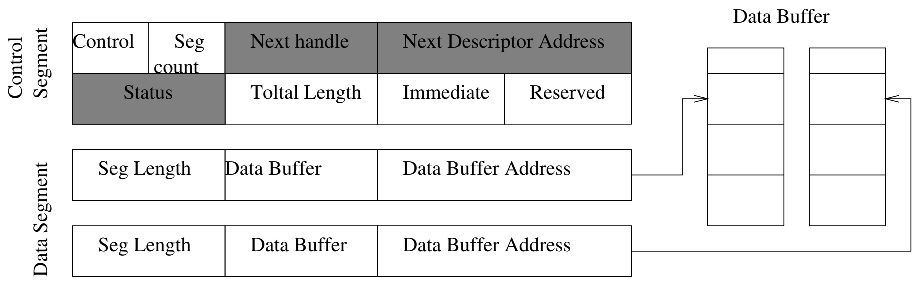

Figura 2.13: Campi del Descrittore aggiornati dalla VI Library

Oltre al Control Segment, la VI Application deve anche modificare il Data Segment, in modo da indicare gli indirizzi dei buffer da spedire o che conterranno i dati ricevuti. Quindi tutti i campi del Data Segment vengono propriamente settati. I buffer possono anche essere più di uno, per questo uno stesso Descrittore può contenere diversi Data Segments.

Successivamente, quando la VI Application posta il Descrittore su una delle code della VI, la sua proprietà passa alla **VI Library**, che si occupa di inserire il Descrittore stesso nella coda corretta modificando il campi relativi all'indirizzo del Descrittore successivo e settando a zero i bit di stato, come mostrato in figura 2.13

Per indicare che il Descrittore è pronto per essere completato, viene modificata la doorbell ed il possesso passa al **VI NIC**, che lo processa in modo asincrono (figura 2.14). Sono quindi eseguiti i trasferimenti dei dati e sono modificati i campi relativi ai buffer nel Control Segment. Infine è settato ad uno il bit di stato che notifica che il Descrittore è completato.

Dopo il completamento, il possesso del Descrittore è ritornato alla **VI Library** (figura 2.15) quando l'applicazione testa la coda della VI. Infatti in questo caso la VI Library controlla proprio il bit di stato modificato dal VI NIC all'avvenuto completamento.

Infine la **VI Application** riprende nuovamente possesso del Descrittore quando è ritornato un puntatore ad esso da una delle primitive che testano

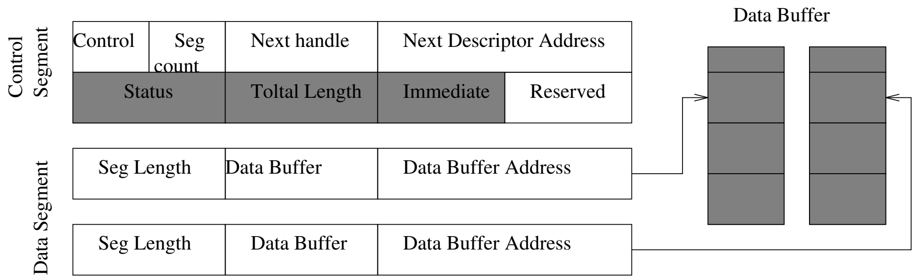

Figura 2.14: Campi del Descrittore aggiornati dal VI NIC

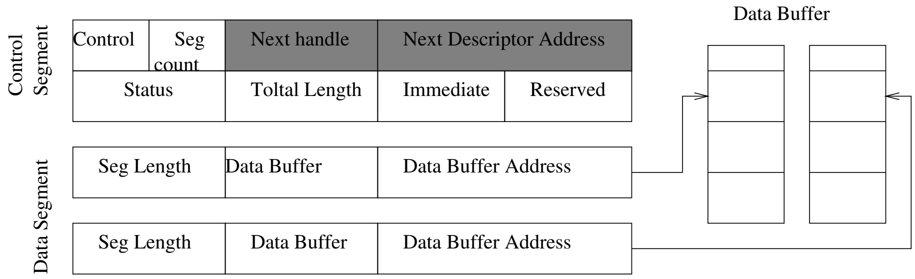

Figura 2.15: Campi del Descrittore aggiornati dalla VI Library

le code delle VI.

2.7 Livelli di affidabilitá

Lo standard VIA specifica tre livelli di affidabilitá [5]:

- Unreliable Delivery Service
- Reliable Delivery Service
- Reliable Reception Service

Per applicazioni ad alto livello, come MPI, non sempre il livello Unreiable Delivery é meglio del Reliable Delivery, in quanto l'affidabilitá puó essere realizzata ad un livello superiore di VIA, ottenendo buoni risultati nelle comunicazioni. Inoltre per certe applicazioni, come video streaming, non é necessaria l'affidabilitá.

Il livello di affidabilitá é specificato nei *VI Attributes*, che sono inizializzati quando una VI é creata. E' necessario che due VI abbiano lo stesso livello di affidabilitá perché possano connettersi tra loro.

Unreliable Delivery é il piú basso livello di affidabilitá della comunicazione. Questo livello garantisce che operazioni di invio dei dati (Send o RDMA Write) siano eseguite una sola volta e quindi che al piú un solo Descrittore é consumato sulla coda del processo remoto. Inoltre garantisce che errori nel trasferimento dei dati siano sempre rilevati dal ricevente. Se pacchetti vengono persi, questo non viene rilevato, e quindi nemmeno ritrasmessi. Non garantisce inoltre che tutti i pacchetti siano ricevuti e che l'ordine con cui questo avviene sia lo stesso con cui i pacchetti sono stati trasmessi. A questo livello di affidabilitá, se si ha un errore, la connessione tra le due VI rimane intatta, in modo che non via sia ritrasmissione e che la comunicazione possa continuare.

Reliable Delivery garantisce invece che i dati inviati siano ricevuti senza errori, una sola volta e nell'ordine in cui sono stati trasmessi. Il VI Provider deve generare un errore da notificare al VI Consumer se durante la trasmissione dei dati sono persi, corrotti o ricevuti in un ordine errato. Quando si verifica un errore, la connessione salta e tutti i Descrittori postati nelle VI sono completati e marcati come errati.

Anche **Reliable Reception** garantisce che i dati siano ricevuti senza errori, una sola volta e nello stesso ordine in cui sono stati trasmessi. Mentre nel caso di Reliable Delivery i Descrittori relativi alla comunicazione sono marcati come completi una volta avvenuta la trasmissione, con questo livello di affidabilitá i Descrittori sono marcati completi solo dopo la ricezione da parte del destinatario di tutti i dati. Come per Reliable Delivery, tutti gli errori devono essere rivelati e riportati e quando ció accade la connessione salta ed i Descittori sono marcati come errati.

2.8 Gestione degli errori

Nel Control Segment di un Descrittore, se i bit di stato non sono tutti settati a zero, significa che è stato rilevato un errore durante il completamento del Descrittore stesso. Questi errori, che rientrano nella categoria degli **errori sincroni**, dipendono dal tipo di Descrittore e dal tipo di connessione utilizzata, in accordo con le specifiche relative al livello di affidabilità della connessione.

Esistono però anche degli errori, detti **errori asincroni**, che non possono essere riportati direttamente dal Descrittore. Esempi di questo tipo di errori sono :

- Post Descriptor Error : l'indirizzo del Descrittore ed il suo riferimento in memoria non sono validi al momento in cui viene postato su una coda di una VI, oppure il campo che contiene l'idirizzo del Descrittore successivo viene modificato dopo che il Descrittore è stato postato.

- Connection Lost : la connessione è stata interrotta e la relativa VI passa in uno stato di errore.
- Receive Queue Empty : un pacchetto in arrovo viene perso perchè la Receive Queue della VI non contiene alcun Descrittore.
- VI Overrun : il VI Consumer tenta di postatre troppi Descrittori su una Work Queue di una VI.
- RDMA Write Protection Error : un'operazione di RDMA Write che fa riferimento ad una zona di memoria remota protetta. Solo nel caso si utilizzi Immediate Data, questo errore è riportato nel Descrittore stesso e non in modo asincrono.
- RDMA Write Data Error : sono rilevati dei dati corrotti durante una operazione di RDMA Write.
- RDMA Write Packet Abort : indica che solo una parte di un pacchetto di una operazione di RDMA Write è giunta a destinazione.
- RDMA Transport Error : è rilevato un errore nel trasporto dei dati durante una operazione di RDMA Write.
- RDMA Read Protection Error : una RDMA Read fa riferimento ad una zona di memoria non consentita.
- Completion Protection Error : quando una applicazione deregistra una zona di memoria che contiene un Descrittore dopo che questo è stato utilizzato, ma prima che il bit di stato che indica il completamento sia settato.
- Catastrofic Error : indica errori hardware o di configurazione.

Questo tipo di errori non vengono notificati direttamente nel Descrittore, ma sono riportati dal VI Provider al VI Consumer utilizzando un handler di default, che può però essere personalizzato dall'utente in modo da cambiare il comportamento del VI Provider per la gestione di ogni singolo errore.

Capitolo 3

Implementazione di VIA per Linux: M-VIA

Per realizzare l'integrazione di LAM-MPI su VIA, si è scelto di utilizzare **M-VIA** (Modular VIA) [6], una emulazione software della Virtual Interface Architecture, basata su schede Ethernet realizzata dai ricercatori del National Energy Research Scientific Center (NERSC) presso i National Laboratories Lawrence Berkeley [7].
Le caratteristiche principali di M-VIA sono:

- Progettazione modulare che permette portabilitá su nuovo hardware
- Elevate prestazioni
- Possibilitá di utilizzo di componenti hardware dedicate o completa implementazione software su schede di rete di diverso tipo
- Coesistenza con protocolli di comunicazione tradizionali, come TCP/IP, sullo stesso nodo

Il pacchetto M-VIA è costituito da un modulo caricabile del kernel, più una libreria che fornisce tutte le primitive utilizzabili dall'utente per effettuare la comunicazione. Una importante caratteristica é quindi quella che puó essere installato senza ricompilare il kernel del sistema operativo.

Per comunicazioni sullo stesso nodo, si deve utilizzare il device di loopback, in quanto non c'è necessità di utilizzare la scheda di rete.

M-VIA è inoltre progettato per essere facilmente portabile anche da altri device di rete, apportando solo poche modifiche a quelli già esistenti. Una importante caratteristica di M-VIA è quella di coesistere con i protocolli di rete tradizionali, così che sulla stessa rete fisica è possibile utilizzare sia VIA che TCP/IP.

Il superamento dei test di conformità Intel, dovrebbe garantire compatibilità con le specifiche di VIA. Dovrebbe inoltre assicurare che il codice prodotto per M-VIA funzioni anche per altre implementazioni di VIA. Per quanto riguarda l'utilizzo di M-VIA su macchine SMP, come il cluster ParMa[2], l'uso del device di loopback dovrebbe essere molto vantaggioso, in quanto la comunicazione tra processi all'interno dello stesso nodo richiede solo una copia di dati, direttamente dalla memoria di uno in quella dell'altro. Questo non è possibile utilizzando il mormale meccanismo di shared memory, che richiede la copia su un buffer intermedio, a meno che la memoria condivisa dai due processi si trovi sullo stesso segmento condiviso. Per quanto riguarda i livelli di affidabilità, attualmente M-VIA supporta solo Unreliable Delivery e uno degli sviluppi futuri con maggiore priorità è proprio quello di implementare il livello di Reliable Delivery.

3.1 Architettura di M-VIA

Secondo le specifiche, la Virtual Interface Architecture descrivono VIA come composto da tre principali componenti: la VI Provider Library (VIPL), il VI Kernel Agent ed il VI NIC. M-VIA rispecchia questa struttura, con la diversa scelta realizzativa che il VI Kernel Agent è piuttosto scomposto in un M-VIA Kernel Agent ed in uno o più M-VIA Device Drivers. L'architettura interna di M-VIA è mostrata in figura 3.1.

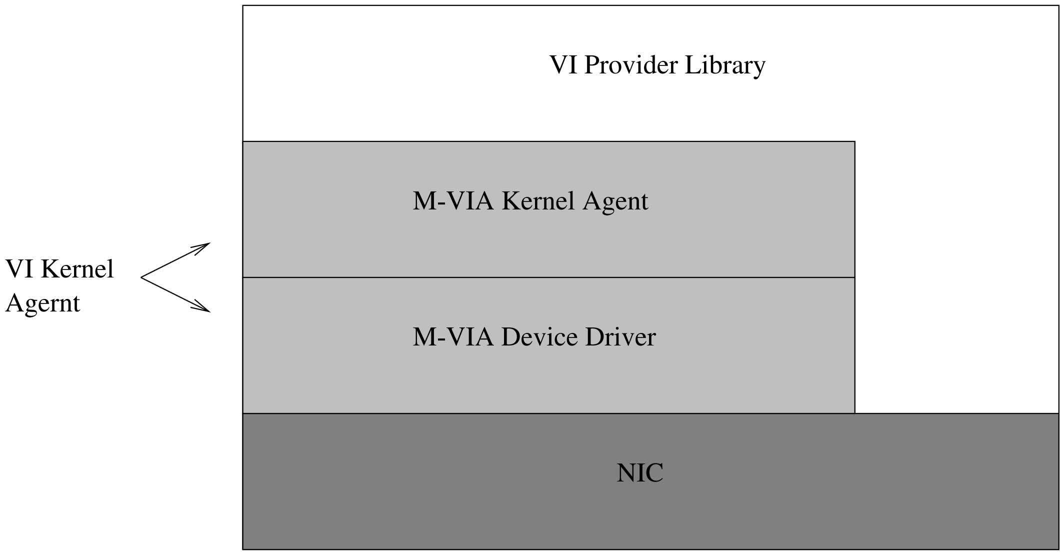

Figura 3.1: Architettura interna di M-VIA.

3.1.1 M-VIA Kernel Agent

M-VIA Kernel Agent esegue operazioni richieste dalla VI Provider Library, ed assiste l' M-VIA Device Drivers per operazioni che richiedono il supporto del sistema operativo. M-VIA Kernel Agent è suddiviso nei seguenti componenti funzionali:

- Connection Manager, che stabilisce la connessione punto-punto tra due VI
- Protection Tag Manager, che alloca, dealloca e convalida i tag di protezione della memoria
- Registered Memory Manager, che fornisce dei riferimenti per le zone di memoria registrate dall'utente e per i buffer utilizzati dai Descrittori
- Error Queue Manager, che fornisce un meccanismo di gestione degli errori asincroni

3.1.2 M-VIA Device Drivers

Un M-VIA Device Driver è una astrazione del VI NIC. Informa il Kernel Agent delle proprie capacità, come la dimensione massima del pacchetto che può essere trasferito, o il numero massimo di Descrittori che possono essere postati su una Work Queue di una VI. E' importante che l'M-VIA Device Driver fornisca tutte le funzionalità del M-VIA Kernel Agent, indipendentemente da come sono realizzate. Ciò consente che se parte di questi moduli sono implementati ad esempio via hardware, se sono conformi alle specifiche di VIA, saranno supportati anche da M-VIA.

3.2 Prestazioni di M-VIA

La versione de M-VIA utilizzata sul cluster ParMa^{2}è la 0.9.3.

Sono stati eseguiti alcuni test tra due nodi del cluster utilizzando un semplice programma che esegue un ping-pong tra i due.

Sono state valutate le prestazioni in termini di ritardo di trasmissione **D(S)**, calcolato come la metá del tempo necessario perché un messaggio di **S** bytes trasmesso ad un processo remoto sia da questo ricevuto e ritrasmesso, ed in termini di velocitá di trasmissione **T(S)**, valutata come S/D(S).

I risultati ottenuti nella comunicazione tra due nodi remoti, riportati in tabella 3.1, hanno evidenziato una **latenza** di 61usec, definita come il ritardo di trasmissione per un messaggio di dimensioni minime, D(Smin), ed una banda asintotica **B** di 11.3 MB/sec, calcolata come (S-Smin)/(D(S)-D(Smin)) quando la dimensione del messaggio S é molto elevata.

I risultati anche sono mostrati graficamente anche in figura 3.2 e in figura 3.2.

3.3 Sviluppi futuri

La versione de M-VIA utilizzata supporta le seguenti schede di rete :

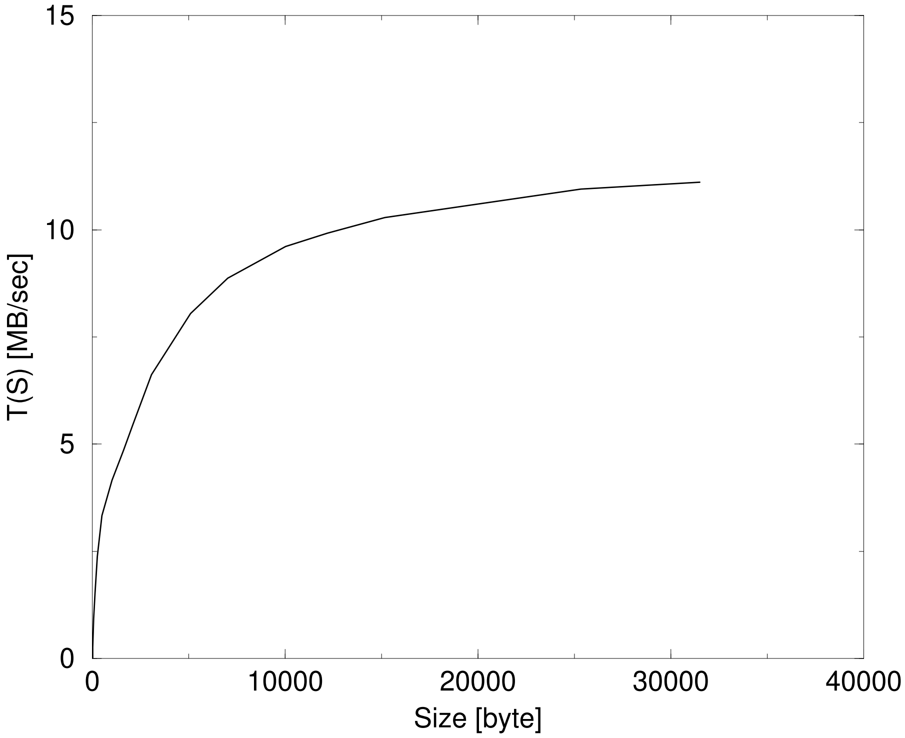

Figura 3.2: Velocitá di trasmissione con M-VIA

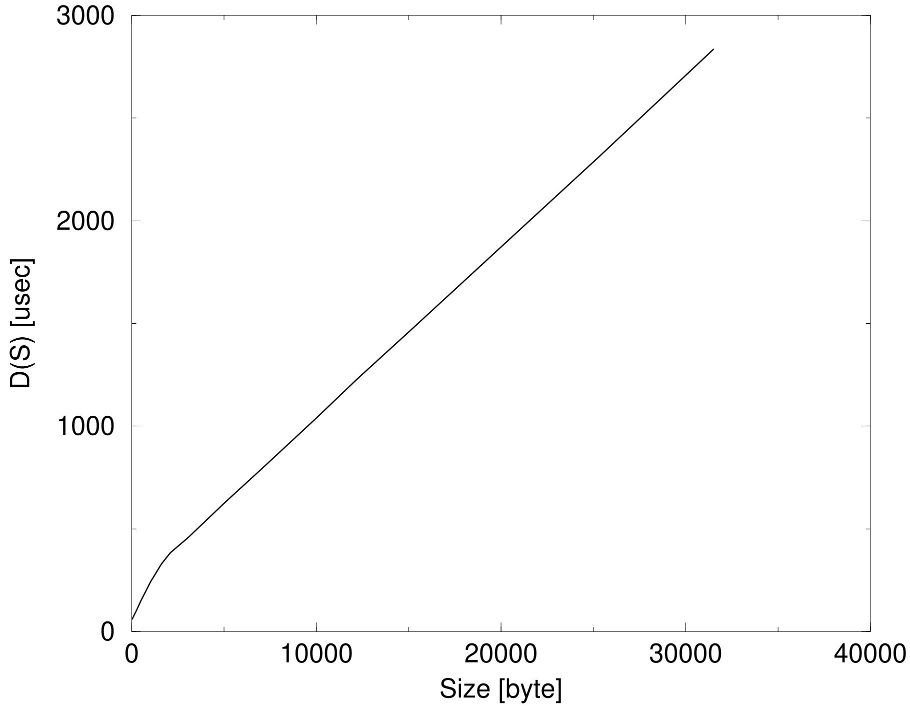

Figura 3.3: Tempi di trasmissione con M-VIA

Size(bytes)	D(S)(usec)	T(S)(MB/sec)
1	61.582506	0.016238
8	61.374009	0.130348
16	61.444998	0.260395
64	68.445027	0.935057
124	81.530511	1.520903
248	103.995025	2.384729
500	149.875462	3.336103
1008	242.412984	4.158193
1608	331.471026	4.851103
2088	383.450985	5.445285
3048	460.019529	6.625806
5088	632.763505	8.040919
7008	790.032029	8.870526
10001	1040.607989	9.610728
12187	1229.036987	9.915894
15147	1472.887993	10.283878
25328	2314.367056	10.943813
31487	2833.899498	11.110839

Tabella 3.1: Prestazioni di M-Via

- Scheda di rete Fast Ethernet con chip DEC DC21x4x (Tulip)
- Scheda di rete Gigabit Ethernet Packet Engines GNIC-I (Yellowfin)
- Scheda di rete Gigabit Ethernet Packet Engines GNIC-II (Hamachi)

Il numero di device supportati è stato lasciato intenzionalmente basso dai realizzatori, perchè la struttura interna di M-VIA evolve ancora molto rapidamente per integrare tutte le caratteristiche delle specifiche. Attualmente M-VIA potrebbe essere considerato ancora un progetto di ricerca, in quanto la sua architettura potrebbe ancora essere modificata, ma per quanto riguarda le applicazioni, queste non dovrebbero risentirne grazie alla conformitá con le specifiche di VIA. Prima di realizzare nuovi drivers per altri tipi di schede di rete, si vuole raggiungere una versione sufficientemente stabile utilizzando quelli giá esistenti. E' infatti giá prevista una nuova versione, chiamata M-VIA 2, che supporterá drivers per Servernet II, Myrinet e Alteon.

Inoltre alcune caratteristiche delle specifiche di VIA non sono ancora state realizzate all'interno di M-VIA. Parte di queste saranno presto disponibili nelle nuove release, che dovranno anche colmare i limiti della versione attuale.

Tra le principali attuali limitazioni ci sono :

- Modello di connessione peer-to-peer : infatti attualmente il solo modello disponibile è quello client-server
- Livello di affidabilità da Unreliable Delivery a Reliable Delivery
- Utilizzo dell'attributo dei Descrittori relativo alla QoS, che adesso deve essere settato a zero per evitare errori
- Limitazione della quantità di memoria che si può registrare per trasmettere o ricevere dati. Questo limite è dovuto al fatto che la memoria registrata viene effettivamente bloccata in memoria fisica. Quindi le dimensioni della RAM pongono un linite alla quantità di memoria registrabile
- La massima dimensione del pacchetto trasferibile è 32 Kbytes. Per messaggi più grandi, è il VI Consumer che si deve occupare della segmentazione e riassemblaggio
- Aggiunta di un controllo di flusso robusto

Capitolo 4

Analisi di LAM (Local Area Multicomputer)

La scelta della libreria MPI da utilizzare per l'integrazione di VIA, é ricaduta su LAM, sia per le sue caratteristiche che per la sua architettura interna. LAM (Local Area Multicomputer) è un ambiente di programmazione parallela per applicazioni distribuite comforme allo standard MPI. E' stato sviluppato presso il Laboratory for Scientific Computing dell'Università di Notre Dame. La completa implementazione dello standard MPI garantisce inoltre la portabilité di codici sorgenti LAM su altre implementazioni MPI. LAM permette anche comunicazioni all'interno di cluster con computer eterogenei tra loro e tra le sue principali caratteristiche di vi é il fornire tools per monitorare le comunicazioni e la possibilità di un controllo di flusso per ottenere informazioni sullo stato di un processo. Permette inoltre di memorizzare tutte le comunicazioni avvenute per averne un riassunto completo in fase di esecuzione o dopo il termine dell'applicazione.

4.1 Message Passing Interface (MPI)

Il Message Passing è un modello di programmazione definito per sistemi paralleli, in cui più elementi di calcolo collaborano tra loro per risolvere problemi di elevata complessità. La collaborazione tra processi avviene utilizzando delle primitive per inviare o ricevere dei messaggi. Il messaggio costituisce quindi l'informazione che i processi possono scambiarsi. Questo è composto sia dai dati veri e propri che da una serie di informazioni, dette **envelope**, necessarie per poter efficacemente eseguire lo scambio. Al momento della spedizione, l'envelope viene aggiunto ai dati da inviare e solitamente contiene :

- Il rank, un numero che identifica il processo mittente
- Il rank del processo destinatario
- Il tipo di messaggio da spedire
- La dimensione complessiva dei dati da inviare
- Il comunicatore, che è un identificatore del gruppo di processi a cui il mittente ed il destinatario appartengono

Per permettere all'utente di sagomare il modello di comunicazione al proprio uso, sono previste diverse primitive di comunicazione che si differenziano per semantica, numero di processi coinvolti o durata della comunicazione. Quindi si possono effettuare comunicazioni del tipo:

- Bloccante o non bloccante. In una semantica di tipo bloccante, il processo chiamante attende fino al completamento dell'operazione richiesta, mentre in una non bloccante, il controllo è subito restituito al processo chiamante che può, eventualmente, svolgere altri compiti. Quest'ultimo tipo di comportamento è preferibile perchè, oltre a evitare stati di attesa e quindi perdite di tempo, permette ad un processo di svolgere più comunicazioni comtemporaneamente ed é inoltre necessario per evitare situazioni di blocco nelle comunicazioni ad anello.
- Sincrona o asincrona. Comunicazioni di tipo sincrono permettono al processo mittente di venire a conoscenza dello stato del processo destinatario. Infatti di solito dopo la riuscita della trasmissione, il ricevente

invia un acknowledgement al mittente per notificare la corretta ricezione del messaggio. Se invece la comunicazione avviene in modo asincrono, i due processi non sono a conoscenza dello stato della trasmissione.

- Persistente o non persistente. Una trasmissione si dice persistente quando viene ripetuta per un elevato numero di volte. In questo caso sono possibili ottimizzazioni, come l'inviio dell'envelope una sola volta all'inizio.

- Point-to-point e collettiva. Una comunicazione point-to-point consente lo scambio di messaggi tra due processi, mentre una di tipo collettivo permette lo scambio di messaggi tra più processio all'interno dello stesso comunicatore.

Message Passing Interface (MPI) è uno standard per la programmazione basato sul paradigma a scambio di messaggi (message passing). Uno motivo per cui questo standard è stato adottato è quello di permettere alle applicazioni di essere portabili su diverse piattaforme.

Le caratteristiche principali dell'interfaccia MPI sono :

- La semantica delle operazioni è indipendente dal linguaggio di programmazione utilizzato.

- Completa portabilitá del codice su diverse piattaforme.

- Si cerca di garantire l'efficienza, consentendo cioè operazioni non bloccanti ed evitando copie di dati intermedie.

- Trasparenza del livello di trasporto al programmatore, che non si deve perciò preoccupare di come avviene la trasmissione.

- Permette comunicazioni sia point-to-point che collettive.

- Facilità d'uso.

4.1.1 Modalitá di comunicazione

La comunicazione tra i processi avviene tramite scambio di messaggi utilizzando le primitive Send e Receive. MPI fornisce quattro diverse modalità per eseguire la **trasmissione dati**, che si differenziano per il modo in cui l'istruzione viene completata :

- Send Sincrona. In questo caso il processo mittente si blocca in attesa di avvenuta ricezione da parte del destinatario. L'operazione é quindi di tipo bloccante, ma viene utilizzata anche per creare dei punti di sincronizzazione all'interno del programma.

- Send Bufferizzata. Questo tipo di send è completata subito in modo asincrono. I dati sono copiati in un buffer predefinito ed è il sistema che si occupa dell'invio, mentre il processo chiamante è libero di svolgere altre attività.

- Ready Send. E' formalmente identica a quella precedente, con la differenza che la trasmissione ha successo solo se al momento della spedizione dei dati, c'è un ricevente in attesa dall'altro lato della connessione.

- Standard Send. E' completata non appena il messaggio è stato inviato e non c'è la notifica da parte del ricevente che la trasmissione sia stata eseguita correttamente.

Tutte queste modalità di Send, che si differenziamo principalmente per il loro comportamento sincrono o asincrono, possono essere realizzate in forma bloccante o non bloccante.

Oltre al messaggio vero e proprio, l'interfaccia MPI richiede che sano trasmessi anche altre informazioni (envelope) per poter compiere correttamente l'operazione voluta. Tra queste c'è il rank del processo destinatario, il tipo di dati contenuti nel buffer, la dimensione del buffer e informazioni riguardanti il comunicatore.

Per quanto riguarda la **ricezione dei dati**, l'unica distinzione fatta è tra semantica bloccante e non bloccante. Nel primo caso il processo chiamante attende fino a che il dato ricevuto non sia disponibile, mentre nel caso non bloccante il processo inizia la ricezione, ma può proseguire svolgendo altre operazioni e controllando successivamente il completamento della ricezione. Anche in questo caso, quando è chiamata la primitiva devono essere specificati, oltre all'indirizzo del buffer che conterrà i dati in arrivo, anche il massimo numero di elementi che il buffer può contenere, il tipo di dati che il buffer deve contenere, il rank del processo sorgente ed informazioni sul comunicatore.

4.2 Architettura interna di LAM-MPI

La caratteristica determinante per la scelta dell'utilizzo di LAM come libreria MPI per l'integrazione di VIA, è la capacità di offrire comunicazioni multiprotocollo, cioè è possibile specificare in fase di compilazione del codice quale tipo di protocollo di basso livello utilizzare per il trasporto dei dati. Attualmente LAM rende disponibili diverse possibilità :

- L'uso di demoni residenti che si occupano di effettuare la comunicazione quando il relativo nodo del cluster è coinvolto nella trasmissione.
- Il protocollo TCP/IP per la comunicazione diretta tra processi residenti su diversi nodi.
- Combinazione tra TCP/IP e l'uso della memoria condivisa (shared memory) per processi che risiedono sullo stesso nodo.

Quest'ultima caratteristica si adatta molto bene all'utilizzo di LAM sul cluster ParMa^2, perchè, oltre a gestire la diversità dei vari nodi, consente di ottenere prestazioni elevate per comunicazioni tra processi che si trovano su uno stesso nodo SMP, in cui ogni processo può risiedere su un diverso processore.

Per consentire la possibilità di utilizzare più protocolli e facilitare la realizzazione del loro supporto, LAM-MPI è stato realizzato in due livelli software ben distinti: il Lower Layer e l'Upper Layer. L'Upper Layer fornisce tutte le primitive utilizzate dall'utente per effettuare la comunicazione, mentre il Lower Layer si occupa dell'implementazione di queste primitive e realizza il trasporto dei dati vero e proprio. Questi due livelli sono collegati tramite un'interfaccia costituita da otto funzioni, detta RPI (Request Progression Interface).

In pratica, per realizzare la comunicazione, l'utente invoca una delle primitive LAM dell'Upper Layer. Queste fanno tutte riferimento ad una o piú funzioni RPI, che sono infine realizzate nel Lower Layer in relazione al tipo di protocollo utilizzato per il trasporto dei dati.

Questa elevata modularità del codice ha permesso che nell'integrazione di VIA all'interno di LAM-MPI, solo le funzioni RPI e il Lower Layer per il protocollo sono stati realizzati, senza intervenire sull'Upper Layer.

In figura 4.1 è schematizzata la struttura software di LAM-MPI.

4.2.1 LAM Upper Layer

L'Upper Layer è quella parte del codice che definisce le primitive che vengono utilizzate dall'utente per effettuare la comunicazione. Per capire come queste funzioni sono organizzate, si esaminerà la primitiva Send. Questa è a sua volta espansa in quattro funzioni:

- *lam_initerr_m*, che controlla che il processo chiamante sia stati inizializzato.

- *lam_setfunc_m*, che inizializza una variabile globale che indica il tipo di operazione che si vuole effettuare.

- *lam_send*, che costruisce la struttura _req necessaria ad inviare i dati.

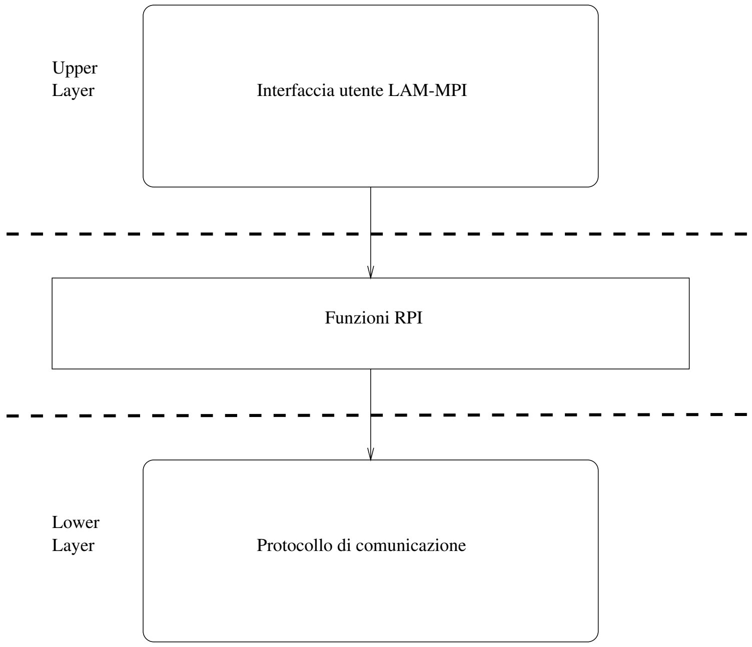

Figura 4.1: Struttura software di LAM-MPI.

- *lam_resetfunc_m*, che riporta allo stato iniziale la variabile inizializzata in precedenza che indica il tipo di operazione effettuata.

A sua volta, la funzione lam_send, è espansa in altre otto funzioni che costituiscono lo strato inferiore dell'Upper Layer e si occupano di inizializzare la struttura _req. Queste sono :

- *_mpi_req_build*, che alloca la memoria per contenere la struttura _req ed inizializza alcuni paramentri di questa.

- *_mpi_req_start*, che esegue eventuali frammentazione dei dati.

- *_mpi_req_add*, che aggiunge la richiesta corrente nella lista delle richieste.

- *_mpi_req_blkclr*, che riporta a zero il flag che indica richiesta bloccante.

- *_mpi_req_blkset*, che, contrariamente alla precedente, rende la richiesta blioccante.

- *_mpi_req_advance*, che avanza le richieste che si trovano nella lista.

- *_mpi_req_rem*, che rimuove la richiesta quando è completata dalla lista.

- *_mpi_req_destroy*, che distrugge definitivamente la richiesta, rilasciando la memoria occupata.

Anche quasi tutte le altre primitive fornite da LAM-MPI suno scomposte in funzioni più semplice che ne garantiscono la modularità, percui l'esempio fornito dall'analisi della primitiva Send risulta particolarmente significativo per comprendere il funzionamento delle altre funzioni realizzate da MPI.

4.2.2 Interfaccia RPI

L'interfaccia RPI (Request Progression Interface) é costituita da otto funzioni che, per ogni operazione di trasferimento dati realizzata dall'Upper Layer, attivano e inizializzano le connessioni e le strutture dati coinvolte nella comunicazione. Inoltre si devono anche occupare di liberare le risorse al termine della trasmissione. Le funzioni RPI sono :

- *_rpi_c2c_init*: é sempre chiamata all'inizio di una sessione MPI, ed inizializza le strutture dati coinvolte nella comunicazione. Al suo interno richiama la funzione _rpi_c2c_addprocs.

- *_rpi_c2c_addprocs*: chiamata dalla funzione precedente, completa l' inizializzazione delle strutture che saranno poi utilizzate nella comunicazione, in base al numero di processi coinvolti. E' anche possibile modificare il numero di processi richiamando delle funzioni MPI opportune che vanno a modoficare le strutture dati create all'inizio della sessione di lavoro.

- *_rpi_c2c_build*: é chiamata dalla funzione dell'Upper Layer _mpi_req_built, ed inizializza le variabili utilizzate durante la comunicazione dal Lower Layer.

- *_rpi_c2c_start*: é chiamata dalla funzione dell'Upper Layer _mpi_req_start e notifica che la richiesta inizializzata dalle funzioni precedenti é pronta per essere trasmessa.

- *_rpi_c2c_destroy*: é chiamata dalla funzione _mpi_req_destroy e serve per liberare le risorse del Lower Layer che erano state occupate durante la comunicazione.

- *_rpi_c2c_advance*: é chiamata dall'analoga funzione MPI _mpi_req_advance e si occupa di avanzare le richieste pronte per essere trasmesse.

- *_rpi_c2c_iprobe*: si occupa di costruire una richiesta di prova non bloccante e di verificare l'eventuale presenza di una risposta.

- *_rpi_c2c_finalize*: é chiamata in conclusione della sessione di lavoro, e esegue le operazioni di cancellazione di tutte le struttre utilizzate nella comunicazione.

Tutte queste funzioni restituiscono un valore che notifica la corretta esecuzione o un codice di errore nel caso contrario. Fa eccezione _rpi_c2c_advance in cui é il Lower Layer che deve tenere aggiornata la struttura _req in quanto l'Upper Layer non é a conoscenza di come i dati sono effettivamente trasferiti.

Tutte queste funzioni devono essere opportunamente riscritte se si vuole cambiare il protocollo di comunicazione e quindi il Lower Layer. E' importante peró mantenere le stesse funzionalitá descritte sopra per nascondere all'applicazione MPI l'implementazione del protocollo utilizzata.

Capitolo 5

Realizzazione di LAM-MPI su M-VIA

Lo standard MPI offre la possibilitá di effettuare il trasferimento dei dati utilizzando varie tipologie di comunicazione (bloccante o non bloccante, sincrona o asincrona, ecc.). Nell' integrazione del protocollo VIA al Lower Layer di LAM-MPI, si devono ancora poter realizzare tutte queste modalitá. Come descritto nel paragrafo 2.5, VIA offre peró solo due modelli di trasferimento dati: Send/Receive e RDMA. Il modello Send/Receive , giá discusso nel secondo capitolo, é di tipo sincrono, in quanto un processo che invia dei dati, deve assicurarsi che ci sia un altro processo in ascolto pronto a riceverli. Inoltre il processo ricevente puó utilizzare una semantica bloccante o non bloccante per il completamento della ricezione. Il modello di comunicazione offerto da RDMA é invece di tipo opposto. Nel caso di lettura o scrittura RDMA, il processo remoto non interviene nella comunicazione, non é quindi necessaria una sincronizzazione iniziale e quindi si puó assumere che il modello sia di tipo asincrono. Fa eccezione l'operazione di RDMA con Immediate Data, che richiede una sincronizzazione iniziale tra i processi con scambio degli indirizzi dei buffer contenenti i dati, e che permette di notificare l'avvenuta ricezione da parte del processo remoto. Una possibile soluzione da adottare per implementare tutte le tipologie di comunicazione che offre lo standard MPI è quella di utilizzare un diverso protocollo per ogni tipo di operazione: il modello Send/Receive per operazioni sincrone, sia bloccanti che non, ed il modello RDMA senza Immediate Data per operazioni asincrone. Questo tipo di soluzione presenta dei vantaggi nella maggiore efficienza del protocollo di comunicazione, ma ha anche lo svantaggio di una maggiore complessità e difficoltà di integrazione all'interno di LAM-MPI.

5.1 Scelte progettuali del protocollo da utilizzare

In base alle considerazioni svolte precedentemente, si è utilizzata una scelta alternativa rispetto al protocollo che implementa i diversi modelli di comunicazione. Questa consiste nell'utilizzo di un protocollo asincrono generico, capace però di adattarsi alle varie semantiche quando richiesto. Considerando le caratteristiche di VIA, per la scelta progettuale di come realizzare questo protocollo asincrono, sono state prese in esame le diverse possibilità, decidendo quindi per un protocollo basato sul pre-posting dei Descrittori [8].

Per risolvere il problema della sincronizzazione tra il processo mittente ed il destinatario, ogni processo posta inizialmente sulla propria Receive Queue un certo numero di Descrittori, evitando cosí che una eventuale comunicazione venga completata con un errore perché il processo ricevente non ha un Descrittore pronto a ricevere i dati. La decisione sul numero di Descrittori da postare, dipende dal tipo di applicazione e dalla frequenza delle comunicazioni tra i processi: infatti se uno o piú processi effetuano molte Send ad uno stesso ricevente, si rischia di consumare preventivamente tutti suoi Descrittori e di completare con un errore tutte le comunicazioni successive. Per evitare questo inconveniente, si riutilizzano i Descrittori giá consumati, realizzando una struttura dati tipo array circolare, in cui i Descrittori consumati in cima alla coda vengono reinseriti in fondo.

Per quello che riguarda invece l'implementazione di una semantica bloccante o non bloccante, si devono necessariamente utilizzare i modelli Send/ Receive o RDMA Write con Immediate Data. E' infatti necessaria la notifica al ricevente del completamento dell'operazione di invio dei dati, che non avviene utilizzando il modello RDMA Write senza Immediate Data. Un inconveniente in cui si puó incorrere é quello della inefficiente gestione della memoria, in quanto un processo che riceve dati da altri processi, non puó essere a conoscenza a priori delle dimensioni di questi dati. Utilizzando il modello Send/Receive, le dimensioni dei dati trasmessi, che devono essere contenuti nel buffer del corrispondente Descrittore sulla Receive Queue del processo che riceve, devono essere inferiori a quelle del buffer di ricezione. Quindi necessariamente questi buffer di ricezione, che devono essere allocati prima della comunicazione, devono essere a priori di dimensioni elevate, per evitare che le dimensioni del messaggio trasmesso eccedano quelle del buffer di ricezione. Utilizzando invece il modello RDMA Write con Immediate Data, é sufficiente registrare una vasta zona di memoria destinata a ricevere i dati, realizzando poi un opportuno controllo di flusso per gestirne il corretto impiego.

Riassumendo, le scelte progettuali del protocollo da utilizzare sono state:

- Protocollo basato sul pre-posting di Descrittori.
- Utizzo del modello RDMA Write con Immediate Data per trasmettere i dati.
- Registrazione di una zona di memoria per ricevere i dati.

Un processo che vuole trasmettere dei dati, deve prima verificare l'esistenza di un buffer pre-registrato di dimensioni sufficienti a contenere il messaggio. Se questo non esiste deve essere registrato, per poi scriverne l'indirizzo nel Descrittore che, dopo essere inizializzato per trasmissioni del tipo RDMA Write con Immediate Data, viene postato sulla Send Queue per essere trasmesso.

Un processo che invece voule ricevere dei dati, deve prima controllare la cima della propria Receive Queue l'eventuale presenza di un Descrittore completato. Quando ció si verifica, legge le dimensioni del buffer nel campo Immediate Data del Descrittore e poi legge i dati dalla RDMA Area. Al termine il Descrittore viene reinizializzato e ripostato sulla Receive Queue per essere poi successivamente riutilizzato.

5.2 Realizzazione del livello RPI con M-VIA

Una volta effettuata la scelta progettuale del protocollo da utilizzare, si deve poi integrarlo all'interno del Lower Layer di LAM-MPI. Il primo passo per fare ciò è quello di riscrivere le funzioni RPI, mantenendole sempre conformi allo standard MPI in modo che realizzino gli stessi compiti di prima.

- *_rpi_c2c_init*: é sempre chiamata all'inizio di una sessione MPI, ed inizializza le strutture dati coinvolte nella comunicazione. Deve perciò inizializzare il VI NIC utilizzando il device di loopback per i processi che risiedono sullo stesso nodo, e quello di rete per processi che risiedono su nodi differenti. Inoltre vengono inizializzati anche i protection tag per la registrazione della memoria e il name service, utilizzato in fase di connessione per la risoluzione dei nomi.

- *_rpi_c2c_addprocs*: chiamata dalla funzione precedente, completa l' inizializzazione delle strutture che saranno poi utilizzate nella comunicazione, in base al numero di processi coinvolti. In pratica ogno processo crea una lista (Hash Table) contenete una entry (VIInfo) per ogni altro processo coinvolto nella comunicazione. Questa contiene tutti i parametri relativi al trasferimento dati, come il numero di Descrittori da postare su ogni coda o gli indirizzi del buffer dei dati. In base alla locazione di ogni processo, si sceglie poi quale device utilizzare effettivamente, si registrano le zone di memoria destinate a contenete i buffer dei dati, con relativo protection tag, e si creano le VI. Segue poi la fase di connessione tra i processi, utilizzando il modello client-server, e lo scambio degli indirizzi delle RDMA Area. All'uscita della funzione quindi, ogni processo possiede una VI connessa con ogni altro processo coinvolto nella comunicazione, e in ogni VI sono postati un certo numero di Descrittori sulla Receive Queue per permettere una corretta sincronizzazione tra client e server.

- *_rpi_c2c_build*: é chiamata dalla funzione dell'Upper Layer _mpi_req_built, ed inizializza le variabili utilizzate durante la comunicazione dal Lower Layer. In pratica inizializza l'envelope del messaggio da inviare.

- *_rpi_c2c_start*: é chiamata dalla funzione dell'Upper Layer _mpi_req_start e completa la richiesta inizializzata dalle funzioni precedenti scrivendo tutte le informazioni necessarie al Lower Layer, come il tipo di operazione da svolgere o l'indirizzo della VI remota.

- *_rpi_c2c_destroy*: é ancora chiamata dalla funzione _mpi_req_destroy e non è stata modificata rispetto alla versione originale che utilizza TCP/IP.

- *_rpi_c2c_advance*: é chiamata dall'analoga funzione MPI _mpi_req_advance. Scandice la lista delle richieste pronte per essere eseguite e, in base al tipo di operazione, chiama le funzioni del Lower Layer che si occupano dell'effettivo trasferimento dei dati utilizzando le primitive di comunicazione di VIA.

- *_rpi_c2c_iprobe*: esegue ancora le stesse funzioni di costruzione di una richiesta di prova non bloccante e di verificare l'eventuale presenza di una risposta.

- *_rpi_c2c_finalize*: é chiamata in conclusione della sessione di lavoro, e esegue le operazioni di cancellazione di tutte le struttre utilizzate nella comunicazione. In particolare disconnette tutte le VI e le distrugge, deregistra la memoria per poterla liberare, cancella i name service e chiude i NIC aperti.

5.3 Lower Layer e controllo di flusso

Per utilizzare VIA come protocollo di comunicazione nel message passing con LAM-MPI é stato completamente riprogettato anche il Lower Layer di LAM, implementando anche le funzioni *stSend()* e *receive()* che realizzano effettivamente la comunicazione.

Quando una applicazione invia dei dati utilizzando ad esempio la primitiva MPI_Send(), a livello RPI é inizializzata una richiesta di trasmissione. questa é poi trasmessa al Lower Layer che la completa inizializzando un opportunamente un Descrittore per la comunicazione con RDMA Write con Immediate Data e postandolo poi sulla Send Queue della VI. L'RDMAArea di destinazione ha dimensione finita, e quindi é necessario un controllo di flusso per consentirne un utilizzo circolare evitando che si esaurisca dopo un certo numero di trasmissioni. Per rendere possibile questo controllo, sono stati introdotti dei puntatori alla RDMAArea che devono essere aggiornati ad ogni trasmissione e scambiati tra il processo mittente ed il ricevente. In particolare:

- RDMAmaxwrite: indica al processo mittente il limite massimo di scrittura nell'RDMAArea. Inizialmete é posto al fondo della stessa ed é modificato quando il messaggio da trasmettere ha dimensioni troppo elevate e deve perció essere scritto all'inizio della RDMAArea. In questo caso infatti il puntatote non puó piú puntare al fondo della RDMAArea perché si rischierebbe di sovrascrivere messaggi non ancora letti dal processo ricevente.

- RDMAmaxread: indica al processo mittente dove il ricevente é arrivato a leggere nella RDMAArea. Questo puntatore é necessario per evitare che dei dati siano sovrascritti prima di essere letti. E' aggiornato dal

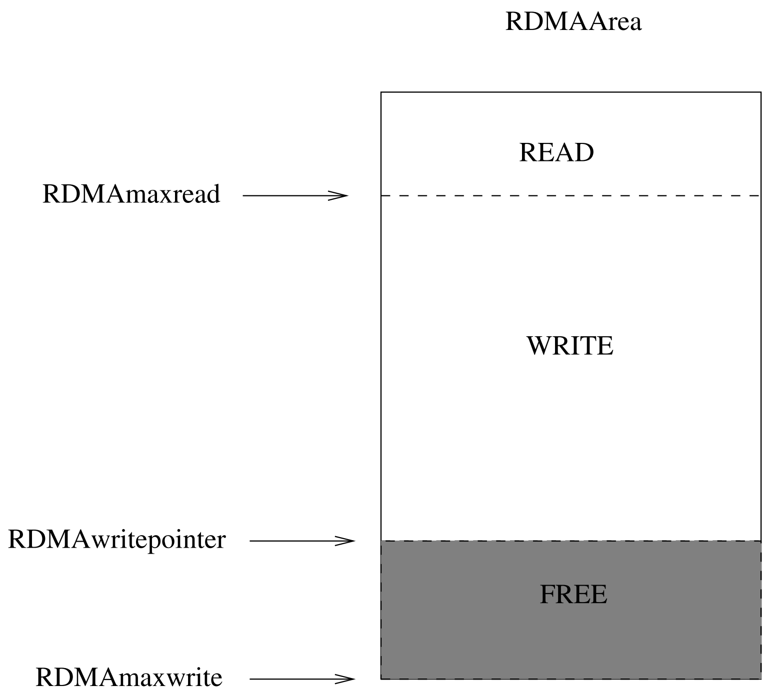

Figura 5.1: Controllo di flusso: caso 1

processo ricevente con una operazione di RDMA Write ogni volta che un messaggio viene ricevuto.

- RDMAwritepointer: é aggiornato dal processo mittente e consente di essere a conoscenza del limite dell'RDMAArea a cui é arrivato a scrivere.

I casi possibili che si possono presentare in cui il controllo di flusso é necessario per evitare che l'RDMAArea si esaurisca e che dei pacchetti vengano sovrascritti prima di essere ricevuti e letti, sono sostanzialmente di due tipi.

Nel primo caso, illustrato in 5.3, il messaggio ha dimensioni troppo elevate per essere contenuto nella parte ancora libera della RDMAArea. Si invia quindi un messaggio di RESTART al ricevente che notifica che il pacchetto successivo sará scritto all'inizio della RDMAArea. Questo ovviamente se

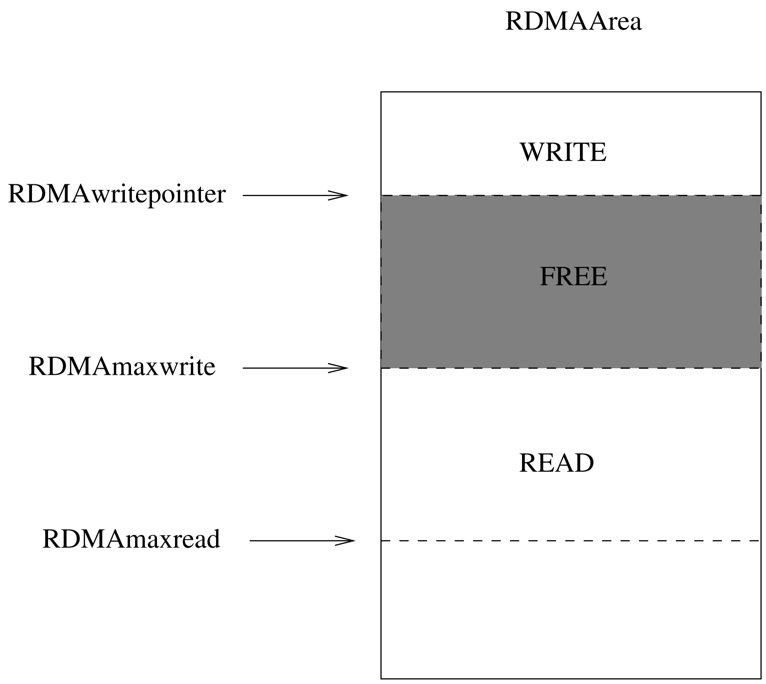

Figura 5.2: Controllo di flusso: caso 2

non implica sovrascritture di pacchetti non ancora ricevuti: infatti se l'area di memoria giá letta compreasa tra l'inizio dell'RDMAArea e il puntatore RDMAmaxread, che indica dove il ricevente é arrivato a leggere, non é sufficiente a contenere il messaggio, il processo mittente si mette in attesa che il ricevente legga altri pacchetti e che aggiorni di conseguenza il puntatore.

L'altro caso che si puó presentare é mostrato in figura 5.3. In pratica si puó trattare come il caso precedente, consentendo cioé al processo mittente di mettersi in attesa che sia liberata memoria se lo spazio libero dell'RDMAArea non é sufficiente a contenere il nouvo messaggio.

Anche il processo ricevente inizializza alcuni puntatori per gestire l'RDMAArea:

- RDMAreadpointer: indica dove il processo ricevente é arrivaro a leggere. Dopo la lettura di ogni messaggio, il ricevente aggiorna il corrispondente puntatore del processo mittente, che é RDMAmaxread.

- RDMAnextpointer: indica al ricevente il primo byte scrivibile ed é necessario per aggiornare RDMAreadpointer nel caso i messaggi da leggere non siano collocati consecutivamente nella RDMAArea.

In fase di ricezione, quindi, un processo controlla che il Descrittore completato sulla receive Queue della VI contenga un messaggio di RESTART o un RDMA Write con Immediate Data. Nel primo caso aggiorna i puntatori in modo da leggere il messaggio successivo all'inizio della RDMAArea. In caso contrario il messaggio viene letto aggiornando anche il puntatore RDMA-maxread del processo mittente. Al termine i Descrittori completati vengono ripostati sulla Receive Queue per evitare che messaggi in arrivo trovino vuota la coda.

5.4 Analisi dei risultati

Sono stati eseguiti alcuni test tra due nodi del cluster utilizzando un semplice programma MPI che esegue un ping-pong tra i due. La versione di LAM utilizzata é la 6.3.1.

I risultati ottenuti nella comunicazione tra due nodi remoti hanno evidenziato per la versione di LAM su VIA una **latenza** di 63usec ed una banda di **B** di 9.6 MB/sec per pacchetti fino a 32Kbyte di dimensioni.

In tabella 5.1 sono confrontate le prestazioni tra la versione di LAM che utilizza VIA e quella che utilizza il TCP come protocollo di trasmissione in termini di ritardo di trasmissione $\mathbf{D(S)}$, valutato come la metá del tempo necessario perché un messaggio di **S** bytes trasmesso ad un processo remoto sia da questo ricevuto e ritrasmesso. In tabella 5.2 sono invece confrontate le prestazioni in termini di velocitá di trasmissione $\mathbf{T(S)}$, valutata come S/D(S).

I risultati sono anche mostrati graficamente in figura 5.4 ed in figura 5.4.

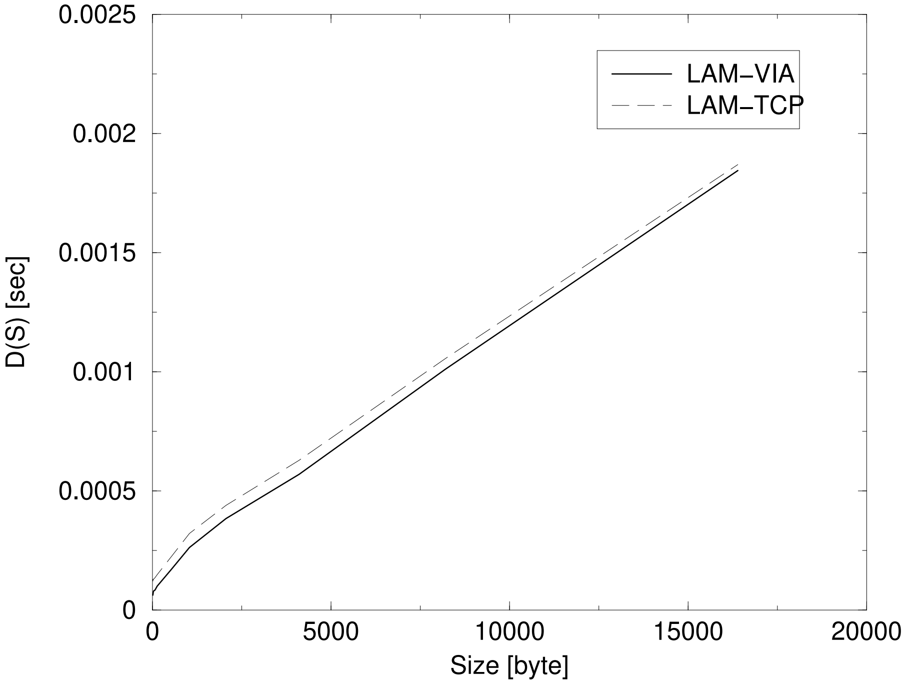

Figura 5.3: Confronto dei tempi di trasmissione tra LAM-VIA e LAM-TCP

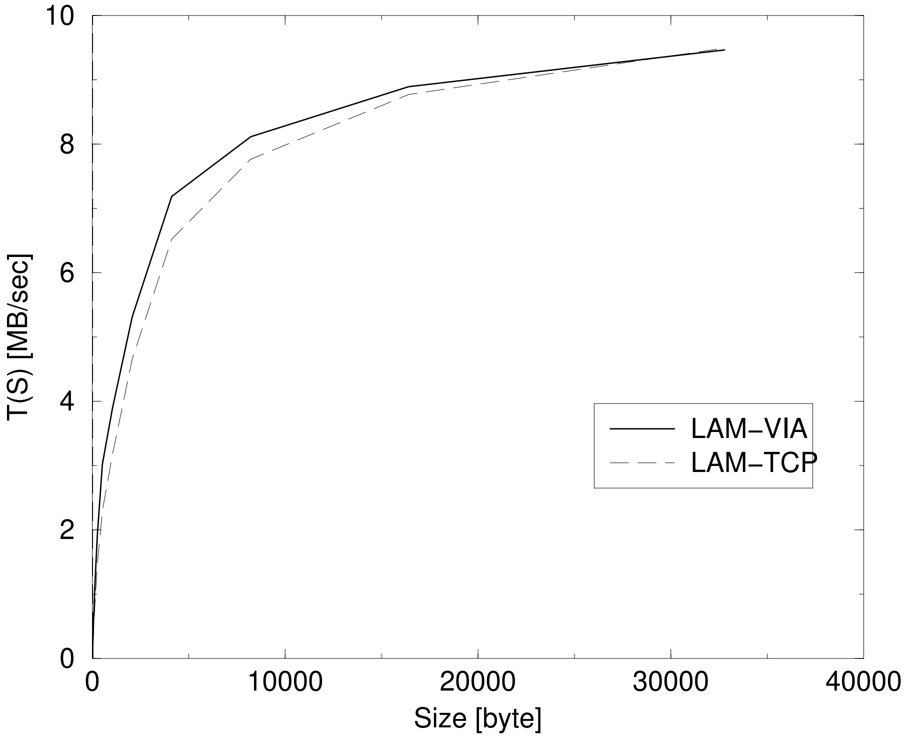

Figura 5.4: Confronto di velocitá di trasmissione tra LAM-VIA e LAM-TCP

Size(bytes)	D(S) LAM-VIA(sec)	D(S) LAM-TCP(sec)
1	0.000063	0.000124
2	0.000063	0.000122
4	0.000063	0.000129
8	0.000063	0.000129
16	0.000077	0.000127
32	0.000081	0.000131
64	0.000084	0.000137
128	0.000100	0.000149
256	0.000124	0.000174
512	0.000170	0.000224
1024	0.000263	0.000322
2048	0.000385	0.000439
4096	0.000570	0.000628
8192	0.001010	0.001055
16384	0.001844	0.001869
32768	0.003461	0.003452

Tabella 5.1: Confronto dei ritardi di trasmissione tra LAM-VIA e LAM-TCP

Sono poi state valutate le prestazioni anche in locale, cioé per due processi che risiedono sullo stesso nodo. I risultati sono mostrati in tabella 5.3.

Size(bytes)	T(S) LAM-VIA(MB/sec)	T(S) LAM-TCP(MB/sec)
1	0.015872	0.008035
2	0. 032800	0.016328
4	0.063974	0.031127
8	0.126026	0.062253
16	0.206489	0.126026
32	0.397388	0.245147
64	0.646443	0.467251
128	1.185146	0.856253
256	1.917396	1.467362
512	3.020371	2.290649
1024	3.893896	3.175577
2048	5.313088	4.660211
4096	7.185975	6.522350
8192	8.114717	7.764913
16384	8.887384	8.766078
32768	9.459685	9.493771

Tabella 5.2: Confronto delle velocitá di trasmissione tra LAM-VIA e LAM-TCP

Size(bytes)	D(S) (sec)	T(S) (MB/sec)
1	0.000023	0.043577
2	0.000023	0.085164
4	0.000023	0.173857
8	0.000023	0.347714
16	0.000023	0.681308
32	0.000024	1.335500
64	0.000024	2.664372
128	0.000025	5.113056
256	0.000027	9.316632
512	0.000031	16.268816
1024	0.000037	27.664846
2048	0.000051	39.768216
4096	0.000094	43.356137
8192	0.000172	47.639152
16384	0.000366	44.768389

Tabella 5.3: Prestazioni di LAM-VIA in locale

Capitolo 6

Frammentazione e riassemblaggio pacchetti

Nell'attuale fase del progetto un'urgenza riscontrata è l'impossibilità di eseguire comunicazioni con pacchetti di dimensioni che eccedono quelle previste come massime nelle specifiche di VIA. Per superare questo limite è necessario progettare un algoritmo in grado di gestire messaggi di dimensioni maggiori. L'apporto più significativo di questa tesi è consistito nel risolvere questo problema. Per mantenere la versione di LAM su VIA conforme allo standard MPI, che non prevede limitazioni di questo genere, è stato perciò realizzato un supporto in grado di suddividere un messaggio che eccede la dimensione massima consentita, che nel caso di M-VIA è di 32KByte, in più parti e in modo trasparente all'applicazione MPI. In ricezione il problema è più complesso, in quanto il messaggio, se composto da più parti, deve essere riassemblato correttamente. Sono perciò state introdotte informazioni aggiuntive da allegare ad ogni singola parte del messaggio che ne consentono il recupero.

Successivamente sono descritti gli algoritmi utilizzati per segmentare in trasmissione un messaggio e per poi ricomporlo correttamente in ricezione. Per rendere tutto ció trasparente all'applicazione MPI, questo meccanismo é stato implementato al Lower Layer di LAM. Come mostrato, il ritardo introdotto dai controlli e calcoli aggiuntivi ha poca influenza nelle prestazioni di comunicazione.

6.1 Frammentazione pacchetti

Ogni processo coinvolto nella comunicazione, possiede una tabella, detta *Hashtable*, contenente una entry per ogni altro attivo nella comunicazione. Queste entry, dette *VIInfo*, consistono in informazioni che un processo deve conoscere per poter trasmettere o ricevere. Ad esempio contengono il rank che identifica ogni processo, contengono anche i puntatori relativi alla RD-MA Area che il mittente ed il ricevente si devono scambiare per una corretta gestione circolare della memoria.

La struttura VIInfo é riportata in dettaglio in Appendice A insieme a tutte le altre introdotte per l'integrazione di LAM-MPI su VIA.

Quando un processo invia un messaggio ad un prefissato destinatario, cerca nella propria Hashtable l'entry relativa al processo interessato e setta nella VIInfo corrispondente i campi relativi alla trasmissione, ed in particolare inizializza l'envelope da "incollare" al messaggio.

La soluzione adottata per permettere la trasmissione di messaggi di dimensioni elevate é stata quella di frazionare in piú pacchetti da inviare consecutivamente. Per consentire poi il corretto recupero del messaggio originale, é necessaria una informazione aggiuntiva relativa al numero del pacchetto corrente. Questa, come mostrato in figura 6.1, e' stata inserita nell'envelope del messaggio.
Quando viene avanzata una richiesta di invio di un messaggio, dal campo dell'envelope che contiene la dimensione del messaggio si calcola in quanti pacchetti questo dovra' essere frazionato.

Considerando che la dimensione massima del buffer da trasmettere é di 32KB e che questi devono includere anche l'envelope, per calcolare il numero

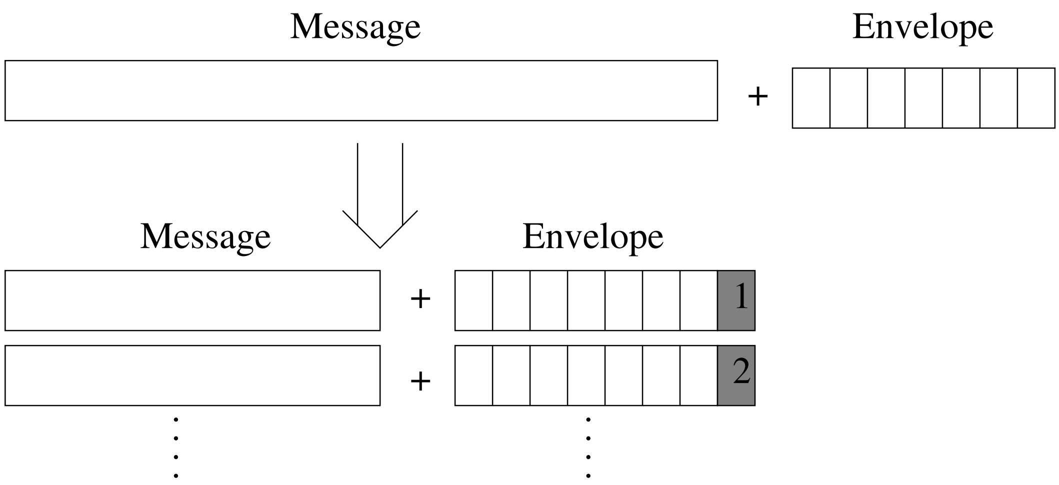

Figura 6.1: Frammentazione pacchetti

dei pacchetti da inviare si é utilizzata la seguente funzione :

```
int
via_ceil (temp)

int temp;

{
  int quot,rem;

  quot = temp / (MAX_TRANSFER_SIZE - SEND_ENV_DIM);
  rem = temp % (MAX_TRANSFER_SIZE - SEND_ENV_DIM);

  if (rem == 0)
    return quot;
  else
    return (quot + 1);
}
```

dove :

- *temp* contiene la dimensione del messaggio da trasmettere.
- *MAX_TRANSFER_SIZE* è pari a 32KB.
- *SEND_ENV_DIM* contiene le dimensioni dell'envelope da allegare al messaggio.

in pratica, se il risultato della divisione è un numero intero, allora quello sarà anche il numero di pacchetti e tutti avranno la stessa dimensione, mentre se il risultato non è intero, si deve incrementare di uno e l'ultimo pacchetto avrà dimensione inferiore agli altri. Il valore ritornato dalla funzione indica perció il numero di pacchetti in cui il messaggio originale sará frazionato.

Rispetto al caso di pacchetto singolo, in cui una richiesta di invio di un messaggio era soddisfatta sibito dopo aver eseguito la trasmissione, nel caso di pacchetti multipli la richiesta è soddisfatta solo dopo aver trasmesso tutti i pacchetti consecutivamente, utilizzando quindi più volte le funzioni del Lower Layer.

Una soluzione alternativa che si potrebbe adottare consiste nello sfruttare l'architettura della VIInfo. Questa struttura contiene infatti un array di envelope ed un corrispondente array di buffer, di cui però si utilizza un solo elemento per volta nella trasmissione. Nel caso in cui un messaggio deve essere frazionato in più pacchetti, si potrebbero utilizzare questi array per contenere i vari pacchetti. Il vantaggio che si otterrebbe da questa implementazione è che la richiesta verrebbe subito soddisfatta dopo un'unica chiamata delle funzioni del Lower Layer, con la differenza che la segmentazione dovrebbe essere poi eseguita interamente al livello superiore.

Questa soluzione è stata scartata sia per questo motivo, mantenendo così una migliore modularità del codice, sia perchè si sarebbe dovuto riscrivere l'intero controllo di flusso già esistente e funzionante.

La funzione di segmentazione dei messaggi maggiori di 32KB è stata implementata per tutti i tipi di send supportati da LAM-MPI, con la differenza che nel caso di trasmissioni sincrone, il processo mittente aspetta dal ricevente la notifica di avvenuta ricezione solo dopo l'invio dell'ultimo pacchetto.

6.2 Riassemblaggio pacchetti

I problemi da affrontare nella fase di riassemblaggio di piú pacchetti per ricostruire il messaggio originale sono più complessi, perchè un processo che è in attesa di un messaggio da un determinato mittente, può riceverne non da quello voluto, ma da altri coinvolti in quella particolare comunicazione. Inoltre, nel caso di trasmissioni di pacchetti multipli, può succedere che tra i pacchetti appartenenti ad uno stesso messaggio se ne inseriscano altri provenienti da altre comunicazioni.

Le strutture dati che sono coinvolte nella ricezione sono :

- *CQ (Completion Queue)* : in cui sono notificati i Descrittori completati
- *RANK List* : lista dei processi che hanno effettuato una trasmissione di dati
- *RDMAArea* : zona di memoria in cui sono contenuti i messaggi ricevuti
- *PTR_TO_ENV list* : lista di puntatori agli envelope dei messaggi e rispettivi offset dall'inizio della relativa RDMAArea

Quando deve essere soddisfatta una richiesta di *receive* di un processo, la prima operazione che si effettua è il controllo della CQ. Se la richiesta è bloccante, si attende che il Descrittore relativo alla send sia completato e quindi notificato nella CQ, mentre se non è bloccante, se la notifica di Descrittore completato non è ancora stata presentata nella CQ, si passa oltre.

Quando nella CQ è presente un Descrittore completato, si controlla se il mittente del messaggio è quello che corrisponde alla richiesta. In ogni caso il Descrittore è rimosso dalla CQ.

Se il mittente corrisponde, si soddisfa la richiesta, altrimenti si aggiornano la RANK list e la PTR_TO_ENV list inserendo i parametri che identificano il Descrittore rimosso. Questa operazione è ripetuta fino allo svuotamento della CQ.

Inizialmente la RANK list e la PTR_TO_ENV list sono vuote, ma è necessario controllarle ogni volta che si effettua una *receive()*.

Infatti, inizialmente, se un processo incontra queste liste non vuote, deve controllare che il messaggio corrispondente alla propria richiesta non sia già stato tolto dalla CQ da un altro processo. Si ha quindi una scansione della RANK list, confrontando il suo contenuto con la richiesta corrente. Se la ricerca ha successo, si risale al rank del processo e nella PTR_TO_ENV list all'offset nella RDMAArea in cui si trova il messaggio. Poi si effettua la *receive()*, si aggiornano i puntatori alla RDMAArea e si cancellano le entry nelle due liste relative al messaggio letto.

Se invece la ricerca non è soddisfatta, si procede testando la CQ.

Nel caso in cui il messaggio da ricevere sia frazionato in piú pacchetti multipli, si possono presentare diverse situazioni :

1. i pacchetti che formano il messaggio sono tutti notificati all'inizio della CQ consecutivamente.
2. i pacchetti che formano il messaggio sono tutti notificati nella CQ, ma intervallati da altri messaggi.
3. i pacchetti che formano il messaggio sono stati rimossi dalla CQ da un altro processo.
4. i pacchetti che formano il messaggio sono solo in parte stati rimossi dalla CQ.

Per risolvere il primo caso, quando si controlla la CQ e si vede che il messaggio soddisfa la richiesta, dai valori dell'envelope si calcolano il numero di pacchetti che formano l'intero messaggio. Si effettua poi la *receive()* correttamente e si ripete fino a che non è stato letto l'ultimo pacchetto. Infine si notifica che la richiesta è soddisfatta.

Per effettuare la receive correttamente, si utilizza il seguente algoritmo :

```
memcpy (req->rq_packbuf+(MAX_TRANSFER_SIZE-SEND_ENV_DIM)* (*(p+6)-1),
        ((char *)p) + SEND_ENV_DIM, thispacksize);
```

dove:

- *req-¿rq_packbuf* è il buffer di destinazione in cui è copiato il messaggio
- *MAX_TRANSFER_SIZE* è pari a 32KB
- *SEND_ENV_DIM* contiene le dimensioni dell'envelope da allegare al messaggio
- **(p+6)* è il puntatore al numero del pacchetto corrente
- **(p)* è il puntatore all'inizio del buffer sorgente
- *thispacksize* è la dimensione del pacchetto corrente

In pratica, quando si fa la copia dal buffer sorgente a quello di destinazione, in base al numero del pacchetto corrente si calcola l'offset della zona di memoria in cui i dati devono essere copiati. Questo meccanismo é mostrato in figura 6.2.

Questo algoritmo risolve anche il secondo caso, contollando che, quando il descrittore rimosso dalla CQ non soddisfa la richiesta, si devono settare correttamente la RANK list e la PTR_TO_ENV list in modo da non perdere il descrittore letto (fig. 6.2).

Per implementare i casi tre e quattro, si deve modificare anche la fase iniziale di testaggio della RANK list e PTR_TO_ENV list. Infatti se un

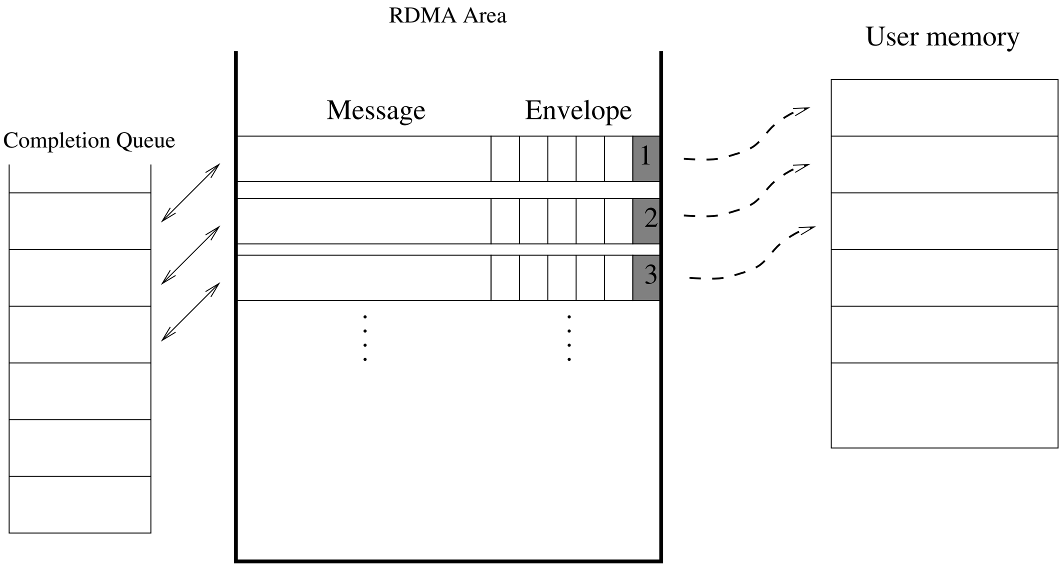

Figura 6.2: Riassemblaggio messaggio: pacchetti consecutivi

processo trova queste liste non vuote, controlla se contengono tutto o una parte del messaggio che soddisfa la richiesta.

Mentre nel caso di pacchetti singoli la scansione delle liste terminava o quando la ricerca aveva successo o quando tutte le liste erano state scandite, nel caso si incontri un messaggio formato da più pacchetti la ricerca si conclude o quando si trova l'ultimo o quando tutta la lista è stata controllata.

Se tutto il messaggio è contenuto nella lista e quindi si esce perchè è stato trovato l'ultimo allora la richiesta è soddisfatta, mentre se solo una parte del messaggio originale è contenuta nella lista e si esce perchè questa è terminata, allora la richiesta non è soddisfatta e il processo procede come nei casi uno e due testando la CQ (fig. 6.2).

In questi ultimi due casi, ogni volta che un pacchetto è letto dalla lista, si fa un controllo di flusso per aggiornare correttamente i puntatori alla RDMAArea e si cancellano le entry dalla RANK list e dalla PTR_TO_ENV list.

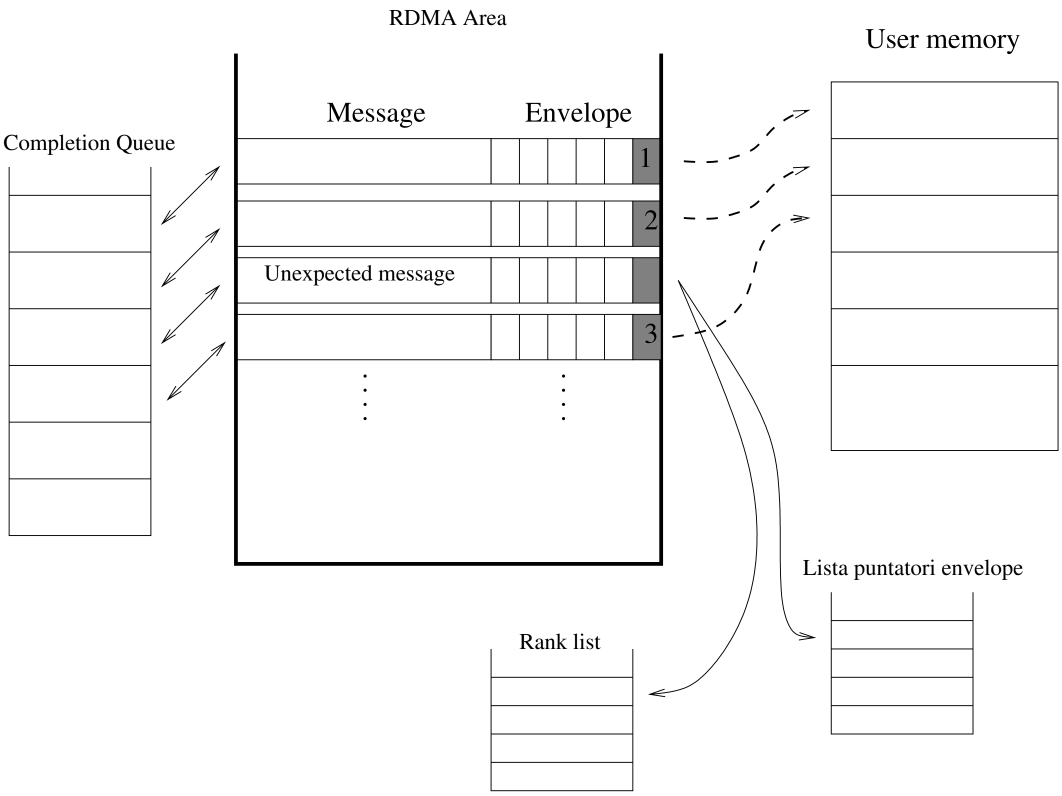

Figura 6.3: Riassemblaggio messaggio: pacchetti intervallati da altri messaggi

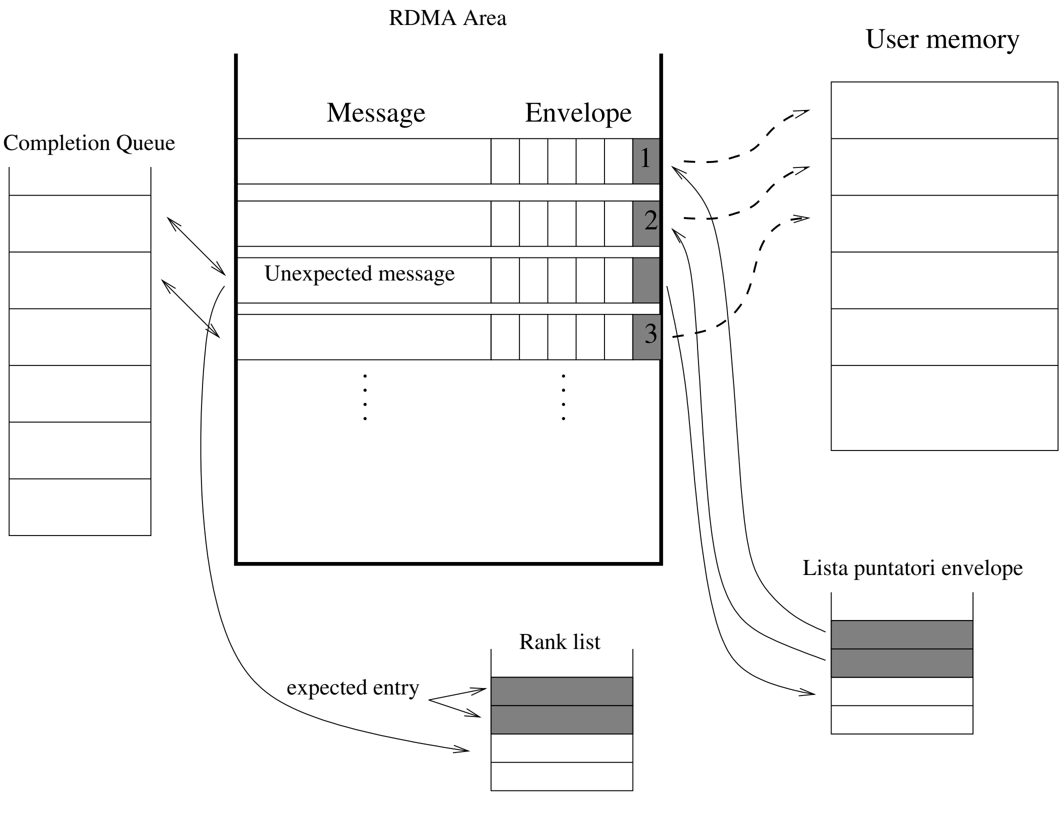

Figura 6.4: Riassemblaggio messaggio: pacchetti in parte giá rimossi dalla CQ

6.3 Analisi dei risultati

Per verificare le prestazioni della nuova funzionalitá di trasmissione di pacchetti di dimensioni maggiori di 32 KB, sono stati eseguiti test tra due nodi del cluster utilizzando un semplice programma MPI che esegue un ping-pong tra i due. La versione di LAM utilizzata é ancora la 6.3.1.

In tabella 6.1 sono riportati i risultati ottenuti in termini di ritardo di trasmissione **D(S)** e di velocitá di trasmissione **T(S)**, utilizzando il device di loopback per due processi che risiedono sullo stesso nodo.

Size(bytes)	D(S)(sec)	T(S)(sec)
1	0.000023	0.043577
2	0.000023	0.085164
4	0.000023	0.173857
8	0.000023	0.347714
16	0.000023	0.681308
32	0.000024	1.335500
64	0.000024	2.664372
128	0.000025	5.113056
256	0.000027	9.316632
512	0.000031	16.268816
1K	0.000037	27.664846
2K	0.000051	39.768216
4K	0.000094	43.356137
8K	0.000172	47.639152
16K	0.000366	44.768389
32K	0.000710	46.120454
65K	0.001465	44.735602
128K	0.003786	34.619929
256K	0.008139	32.208323
512K	0.017887	29.310345
1M	0.035845	29.253077

Tabella 6.1: Prestazioni di LAM-VIA in locale

I risultati ottenuti nella comunicazione tra due processi residenti sullo stesso nodo hanno quindi evidenziato una **latenza** di 23usec.

In tabella 6.2 sono confrontate le prestazioni nelle comunicazioni con pacchetti di dimensioni elevate tra la versione di LAM che utilizza VIA e quella che utilizza il TCP come protocollo di trasmissione in termini di ritardo di trasmissione $\mathbf{D(S)}$, ed in tabella 6.3 sono invece confrontate le prestazioni in termini di velocitá di trasmissione $\mathbf{T(S)}$, sempre nelle comunicazioni con pacchetti di demensioni elevate. I risultati ottenuti nella comunicazione tra due nodi remoti hanno evidenziato per la versione di LAM su VIA una **latenza** ancora di 63usec ed una banda asintotica **B** di 10.1 MB/sec.

I risultati anche sono mostrati graficamente anche in figura 6.3 e in figura 6.3.

Size(bytes)	D(S) LAM-VIA(sec)	D(S) LAM-TCP(sec)
50K	0.004982	0.005159
100K	0.009492	0.010419
150K	0.014428	0.015531
200K	0.019159	0.020702
250K	0.024236	0.025883
300K	0.029363	0.031008
350K	0.034623	0.036379
400K	0.039672	0.041649
450K	0.044746	0.046826
500K	0.049643	0.051974
600K	0.059494	0.062080
700K	0.069298	0.072235
800K	0.079319	0.082308
900K	0.088642	0.092334
1M	0.098351	0.102054

Tabella 6.2: Confronto dei ritardi di trasmissione tra LAM-VIA e LAM-TCP

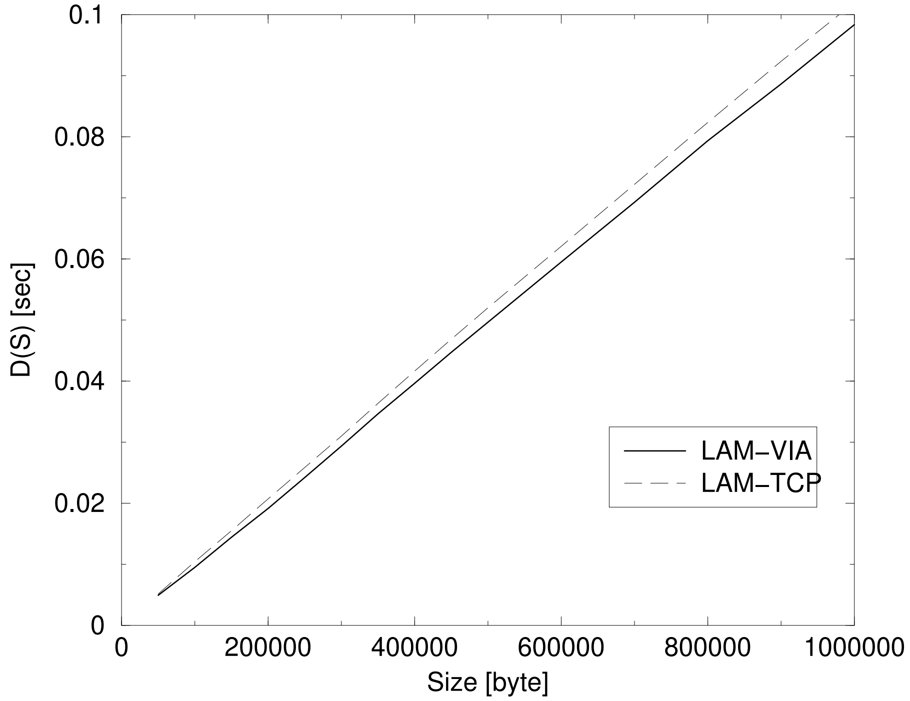

Figura 6.5: Confronto di tempi di trasmissione tra LAM-VIA e LAM-TCP

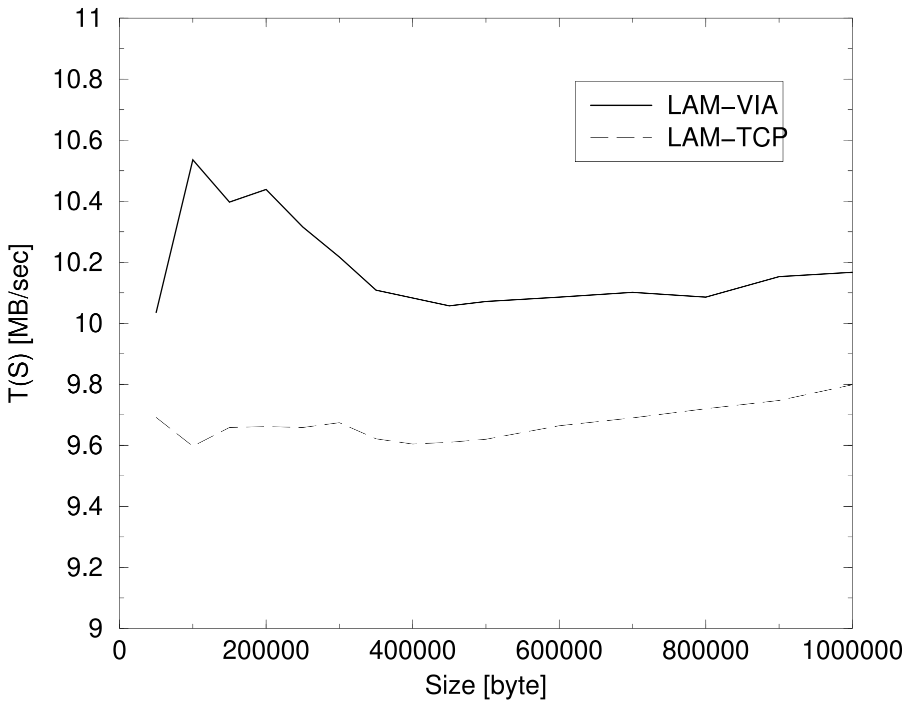

Figura 6.6: Confronto di velocitá di trasmissione tra LAM-VIA e LAM-TCP

Size(bytes)	T(S) LAM-VIA(MB/sec)	T(S) LAM-TCP(MB/sec)
50K	10.036141	9.691762
100K	10.535210	9.597895
150K	10.396480	9.658398
200K	10.438948	9.661123
250K	10.315249	9.658842
300K	10.216927	9.674921
350K	10.108879	9.621067
400K	10.082552	9.604076
450K	10.056876	9.609947
500K	10.071917	9.620107
600K	10.085047	9.664954
700k	10.101367	9.690659
800K	10.085856	9.719590
900K	10.153146	9.747219
1M	10.167665	9.798780

Tabella 6.3: Confronto di velocitá di trasmissione tra LAM-VIA e LAM-TCP

Capitolo 7

Un termine di paragone: GAMMA

Questo capitolo descrive l'architettura e le prestazioni di GAMMA, un prototipo di sistema per il trasferimento di messaggi realizzato presso il DISI dell'Universitá di Genova [9]. La necessitá di migliorare le scarse prestazioni offerte dai protocolli tradizionali per trasmissioni all'interno di cluster di PC ha motivato la realizzazione di una interfaccia di comunicazione di basso livello da utilizzare in sostituzione degli standard classici.

Inizialmente sará descritta l'architettura software di GAMMA, il modello di comunicazione che utilizza ed il protocollo di comunicazione implementato. Successivamente è analizzata una implementazione dello standard MPI, MPICH, basata su GAMMA.

Gli uguali obiettivi alla base dello sviluppo dello standard VIA e di GAMMA, unita alla loro compatibilitá hardware rendono significativo un confronto tra i due, sia in termini di prestazioni di base che di comunicazioni utilizzando la libreria MPI.

7.1 GAMMA: Genoa Active Message MAchine

GAMMA, Genoa Active Message MAchine, é un progetto di un sistema di comunicazione realizzato presso il Dipartimento di Informatica e Scienze dell'Informazione dell'Universitá di Genova.[10]

GAMMA si basa su una particolare implementazione del paradigma Active Message [11], detta *Active Ports*, che permette di realizzare la comunicazione senza memorizzazioni temporanee dei messaggi, evitando cosí dispendiose copie di dati. Il paradigma Active Message riduce notevolmente l'overhead delle comunicazioni sulle prestazioni. La sua particolare semantica, infatti, elimina la necessitá di copie intermedie dei dati, consentendo cosí maggiore velocitá nelle comunicazioni e possibilitá di sovrapposizione tra comunicazione e computazione. In un tradizionale sistema di comunicazione, un messaggio trasmesso deve essere temporaneamente memorizzato in attesa che il processo remoto lo consumi. Active Message invece utilizza delle funzioni, dette *communication handles*, che sono direttamente attivate o dal processo ricevente o da quello trasmittente in risposta a segnali di interrupts generati dal device di rete. Ad esempio, nel caso di ricezione di un messaggio, é il receiver handler, che, agendo come thread separata, consuma il messaggio, cioé si fa carico di integrare le informazioni che contiene nella thread principale, sia notificandone l'arrivo che settando le strutture dati allo scopo di prepararsi per la ricezione del messaggio successivo [12].

L'implementazione del paradigma richiede una estensione del kernel del Sistema Operativo che nel caso di GAMMA consiste nell'implementazione di un set di system call addizionali e di un device di rete che deve essere persente su ogni nodo del cluster. Il livello di comunicazione é perció in buona parte integrato nel Linux kernel, mentre la parte restante consiste in una libreria di primitive direttamente utilizzabili dall'applicazione utente.

7.1.1 Comunicazioni con GAMMA

GAMMA ottiene delle ottime prestazioni nelle comunicazioni rispetto ai protocolli tradizionali, come TCP/IP. Il motivo, come illustrato in figura 7.1.1, sta nel minor numero di livelli del kernel e del protocollo attraversati dal messaggio che si vuole spedire o ricevere..

Per i sistemi di comunicazine tradizionali, i fattori che principalmente contribuiscono all'elevata latenza e alla ridotta velocitá di comunicazione sono quindi :

- copie intermedie dei dati dalla memoria utente a quella del kernel
- protocolli con molti livelli software, come TCP/IP
- context switch che avvengono alla chiamata di system call
- scheduling e de-scheduling

Utilizzando il paradigma Active Message, la comunicazione appare invece piú snella e rapida. Un processo che desidera inviare un messaggio deve invocare una Active Message system call che ha lo scopo di copiare la parte di memoria che contiene il messaggio del processo che la ha invocata nella memoria del processo destinato a riceverlo. Quando il processo mittente invoca la system call per spedire, il messaggio è subito suddiviso in una sequenza di frame Ethernet dal driver di GAMMA, che sono immediatamente trasferiti sulla coda di output, senza copie intermedie nella memoria del kernel. Quando anche l'ultima frame è trasferita, la system call termina ed il processo mittente può riprendere la sua normale esecuzione. L'overhead introdotto è quindi molto limitato, grazie anche al leggero protocollo di trasmissione, che peró non fornisce strumenti per il recupero dei pacchetti danneggiati. Per quel che riguarda invece la ricezione, all'arrivo delle frame Ethernet é attivato un *interrupt handler* che ha il compito di copiare il contenuto della coda di input nella memoria del processo ricevente, evitando copie in buffer intermedi.

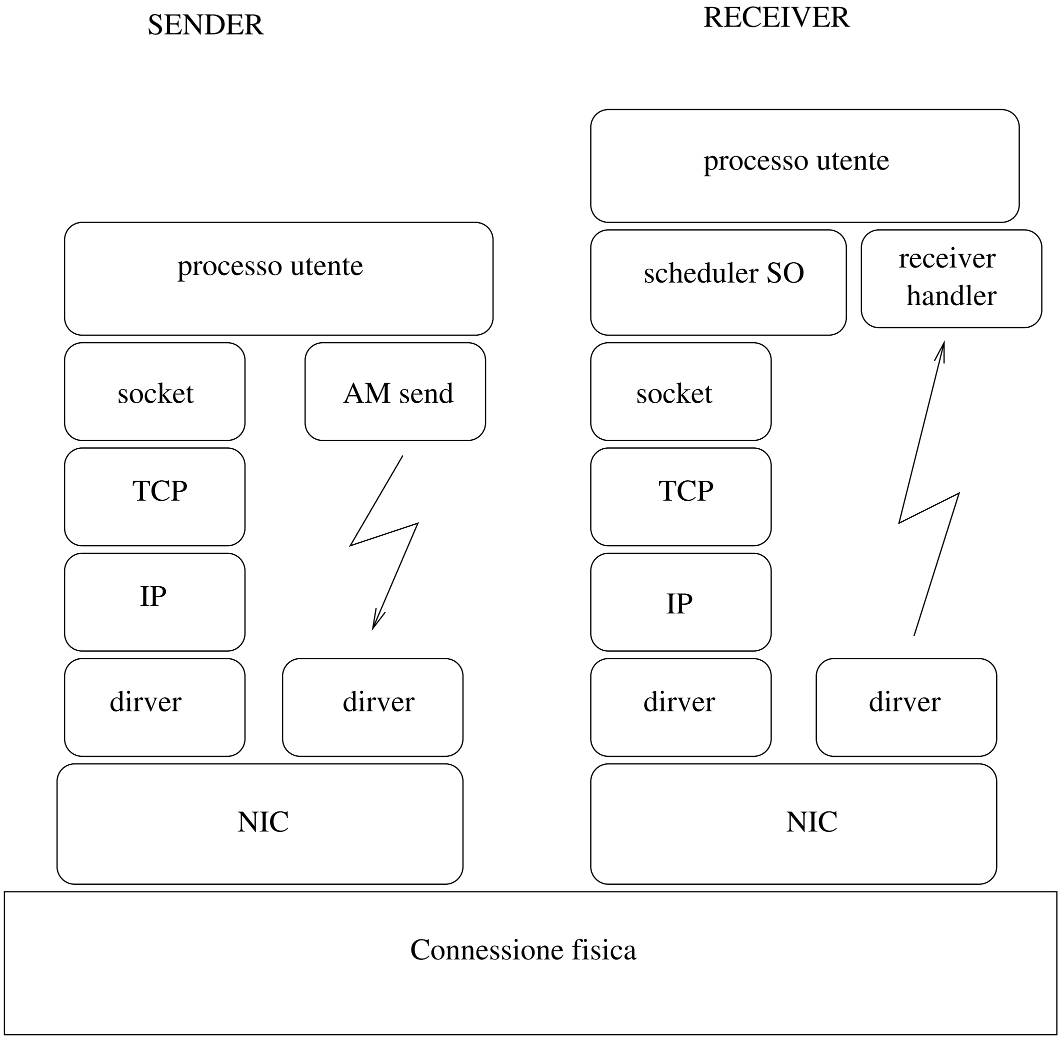

Figura 7.1: Architettura di GAMMA

Dopo il completamento della copia del messaggio dalla memoria del processo trasmittente a quella del ricevente, questo invoca il *receiver handler*, che ha il compito di integrare il messaggio arrivato nelle strutture dati del processo ricevente e di preparare il buffer di ricezione per i successivi messaggi.

Quindi una semplice implementazione del paradigma Active Message riduce sostanzialmente la latenza evitando copie in buffer di memoria intermedi e evitando lo scheduler del SO in ricezione.

7.1.2 Protocollo di comunicazione

Per eseguire la **spedizione** di un messaggio, una applicazione deve invocare la relativa funzione, specificando anche la porta di uscita che deve essere utilizzata, il buffer contenente il messaggio e la lunghezza in byte del messaggio. Questa funzione trasferisce il controllo dal processo al modo kernel attraverso una system call, ed é il driver di GAMMA implementato nel kernel stesso che divide il messaggio in frame Ethernet e che inserisce i puntatori a queste frame nella coda di output senza effettuare copie intermedie.

Quando anche l'ultima frame é trasferita, la system call ritorna ed il processo riprende le proprie attivitá.

Ad ogni frame Ethernet sono attaccati 20 byte di informazioni relative alla porta ed al processo di destinazione e che sono specifici del protocollo di comunicazione utilizzato da GAMMA :

- *byte 0-11*: contengono l'indirizzo Ethernet del ricevitore e di chi trasmette.
- *byte 12-13*: permettono di distinguere da frame di GAMMA e da frame del protocollo di comunicazione standard. Questo permette su un'unica scheda di rete possano essere usati contemporaneamente sia GAMMA che il protocollo standard, come TCP/IP.
- *byte 14-15*: identificano la porta a cui inviare i dati in base al PID ed al numero della porta.

- *byte 16-17*: indicano quanti byte di messaggio sono allegati alla frame. Questa informazione é un numero positivo compreso tra 0 e 1516, in modo da permettere che l'MSB del byte 18 viene settato ad 1 se la frame é l'ultima del massaggio.
- *byte 18-19*: contiene una sequenza di numeri di frame in cui il messaggio é stato frazionato e che permette quindi di rilevare se delle frame vanno perse.

In una prima versione del protocollo, per la **ricezione** non era prevista nessuna copia in memoria, raggiungendo così alte prestazioni, a scapito però della mancanza di un meccanismo di recupero degli errori e dell'assenza di un controllo di flusso. Successivamente, con l'introduzione del meccanismo di trasferimento dati basato su Descrittori (DBDMA) è stata creata una nuova versione del protocollo. Questa sfrutta infatti la caratteristica del DBMA che permette al device di rete di utilizzare due regioni di memoria allocate dal kernel come code di ricezione e di trasmissione. Ció implica però l'impossibilità a ricevere un messaggio senza effettuare una copia in memoria. L'impatto che ha questa copia addizionale sulla latenza e sulla velocità di trasmissione è però molto modesto. Inoltre le maggiori dimensioni della memoria allocata per la coda di ricezione permette l'implementazione di un controllo di flusso con basso overhead. Questo controllo di flusso, che garantisce bassa probabilitá di perdita di pacchetti in qualunque condizione di carico, introduce quindi solo piccoli peggioramenti delle prestazioni rispetto alla versione iniziale del protocollo, ma questo è giustificato da un sostanziale aumento di affidabilità. Quando una frame è ricevuta, vengono consumati dei Descrittori preallocati sulla receive queue, copiando i dati in arrivo nella zona di memoria che fa riferimento al Descrittore stesso. Per ricostruire il messaggio di partenza, le frame vengono riassemblate con una opportuna copia all' indirizzo di memoria corretto. Quindi la copia in memoria aggiuntiva è eseguita dalla CPU ed è necessaria per ottenere il messaggio nel buffer specificato dal processo ricevente.

7.2 Implementazione di MPICH su GAMMA

Le caratteristiche del protocollo GAMMA e le alte prestazioni ottenute nelle trasmissioni di messaggi tra processi possono essere sfruttate nell'implementazione di una interfaccia MPI, con lo scopo di minimizzare l'overhead nelle comunicazioni. L'implementazione della libreria MPI utilizzata è MPICH, prodotta presso l'Argonne National Laboratory. Questa ha come caratteristica primaria la portabilità su diverse piattaforme, grazie all'utilizzo di un intermedio livello software detto ADI, Abstract Device Interface, che contiene funzioni per realizzare le primitive MPI utilizzate dall'utente.

Nella realizzazione di MPICH su GAMMA è stato modificato proprio l'ADI, allo scopo di reimplementarlo per GAMMA [13]. Questo nuovo sistema di trasmissione è detto GADI (GAMMA ADI), e la sua architettura è mostrata in figura 7.2.

La **trasmissione** dei dati, che avviene ad esempio utilizzando la primitiva MPI_Send(), sfrutta lo stesso meccanismo utilizzato da GAMMA per evitare copie intermedie di dati e rendere cos´í la comunicazione molto efficiente.

La parte di **ricezione** invece utilizza due threads concorrenti: una invocata dalla primitiva MPI_Recv(), detta GADI receiver thread, e l'altra dal receiver handler di GAMMA, detta GADI handler thread. Queste due thread comunicano attraverso due code di memoria condivise, che sono peró mutuamente esclusive. All'arrivo di ogni messaggio, questo viene memorizzato nella memoria del kernel. La GADI handler thread permette di ispezionare ogni messaggio istantaneamente, confrontandolo con la richiesta corrente. Se la richiesta é soddisfatta allora il messaggio é copiato al corretto indirizzo nella memoria utente. Se invece la richiesta non é soddisfatta, i dati sono copiati in un buffer temporaneo appositamente allocato.

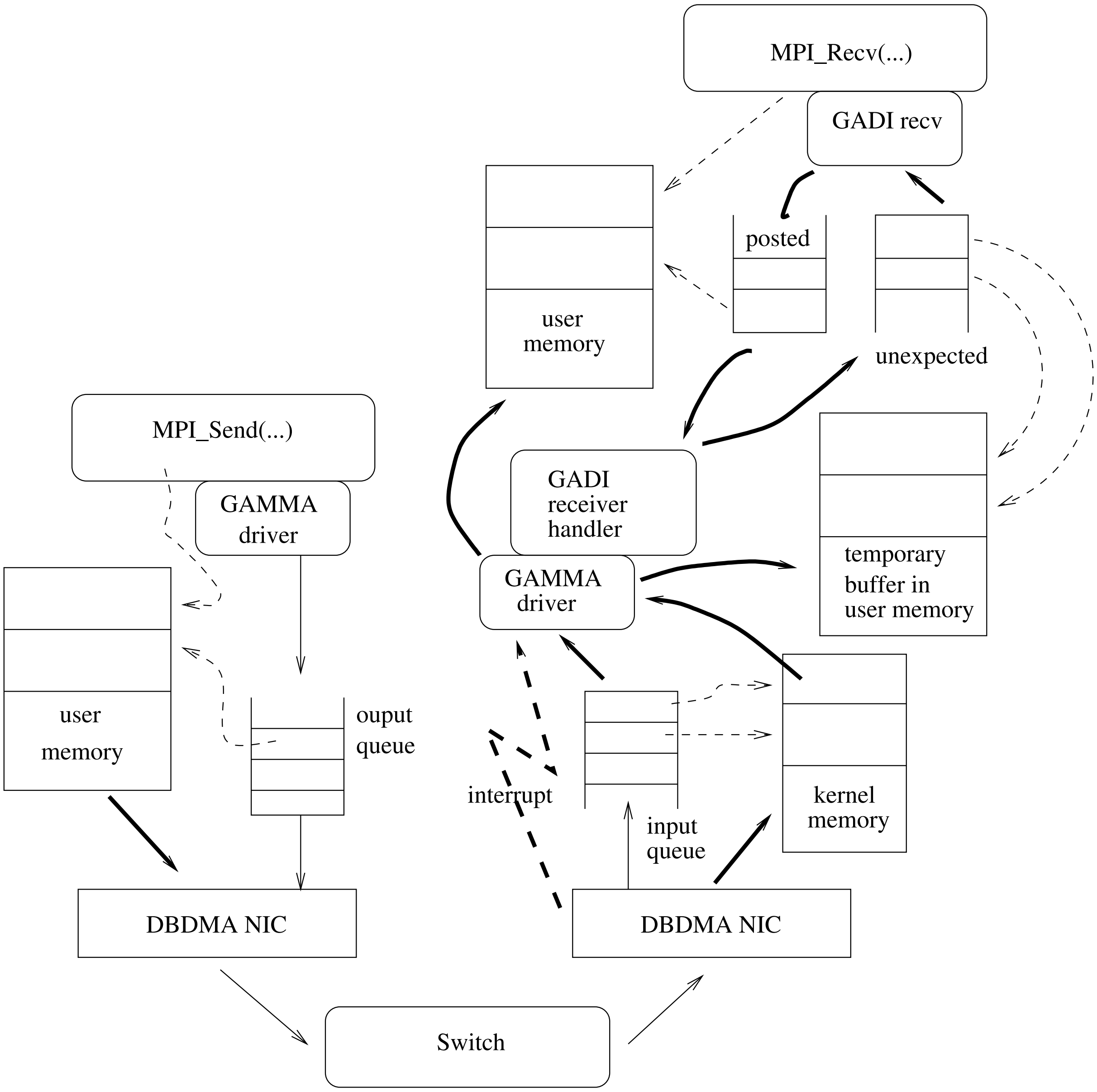

Figura 7.2: Architettura di MPICH su GAMMA

7.3 Analisi dei risultati

La versione di GAMMA per kernel 2.2.13 é stata installata su due nodi del cluster ParMa^{2}per verificarne le prestazioni e confrontarle con quelle di M-VIA.

Sono perció stati eseguiti alcuni test utilizzando un semplice programma che esegue un ping-pong tra i due.

Sono state valutate le prestazioni di GAMMA in termini di ritardo di trasmissione $\mathbf{D(S)}$ ed in termini di velocitá di trasmissione $\mathbf{T(S)}$.

I risultati ottenuti nella comunicazione tra due nodi hanno evidenziato per GAMMA una **latenza** di 21usec ed una banda asintotica **B** di 10.8 MB/sec. In tabella 7.1 ed in tabella 7.2 sono confrontete le prestazioni di M-VIA e di GAMMA. I risultati anche sono mostrati graficamente anche in figura 7.3 e in figura 7.3.

Size(bytes)	D(S) M-VIA (usec)	D(S) GAMMA (usec)
1	61.5825	22.0466
2	61.3595	22.7727
4	61.3824	22.4956
8	61.3740	22.102
16	61.4449	22.4424
32	61.7159	22.745
64	68.4450	27.143
128	81.5305	39.0477
256	103.9950	62.4302
512	149.8754	109.396
1K	242.4129	196.815
2K	383.4509	321.819
4K	549.6104	484.497
8K	894.6098	861.306
16K	1557.6205	1578.8

Tabella 7.1: Confronto di ritardi di trasmissione tra M-VIA e GAMMA

E' stata inoltre installata sui nodi del cluster ParMa^{2}la versione di MPICH-GAMMA per poterne confrontare le prestazioni con la versione di LAM-

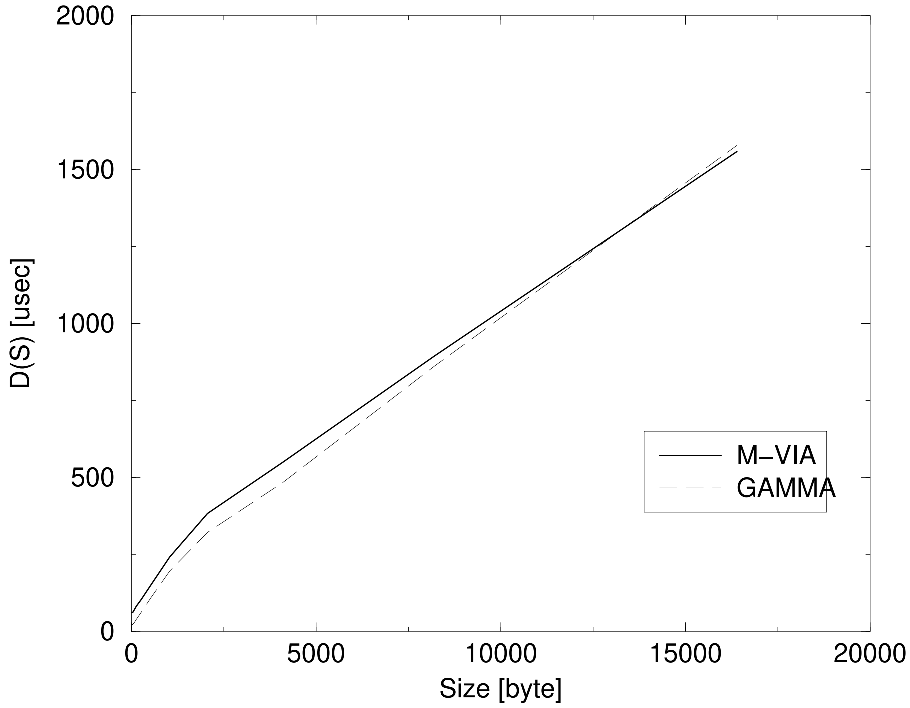

Figura 7.3: Confronto di ritardi di trasmissione tra M-VIA e GAMMA

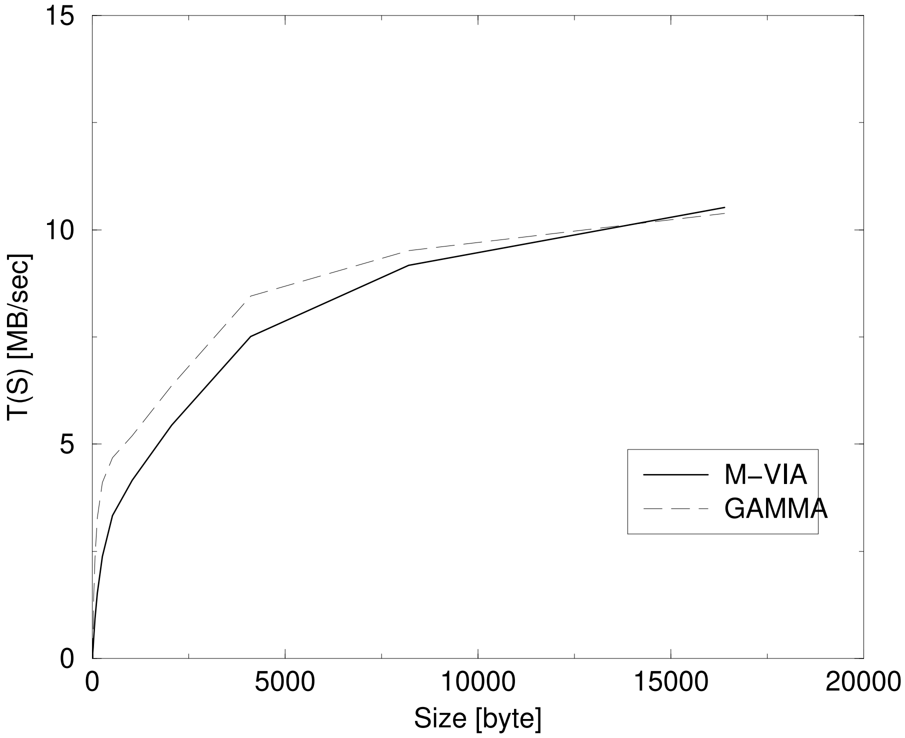

Figura 7.4: Confronto di velocitá di trasmissione tra M-VIA e GAMMA

Size(bytes)	T(S) M-VIA (MB/sec)	T(S) GAMMA (MB/sec)
1	0.0162	0.0453
2	0.0325	0.0878
4	0.0651	0.1778
8	0.1303	0.3619
16	0.2603	0.7129
32	0.4611	1.4069
64	0.9350	2.3578
128	1.5209	3.2780
256	2.3847	4.1005
512	3.3361	4.6802
1K	4.1581	5.2028
2K	5.4452	6.3638
4k	7.5107	8.4541
8K	9.1749	9.5111
16K	10.524	10.3775

Tabella 7.2: Confronto di velocitá di trasmissione tra M-VIA e GAMMA

VIA. Per completezza nel confronto sono anche state inserite le prestazioni di LAM-TCP.

I risultati sono mostrati in tabella 7.3 ed in tabella 7.4 e graficamente in figura 7.3 ed in figura 7.3.

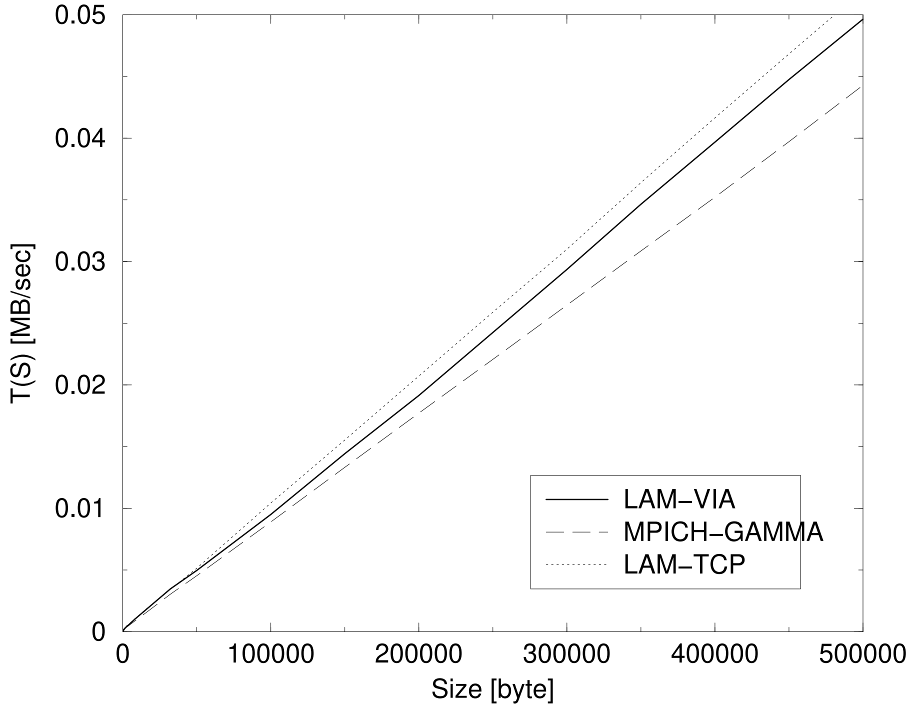

Figura 7.5: Confronto di ritardi di trasmissione tra LAM-VIA, MPICH-GAMMA e LAM-TCP

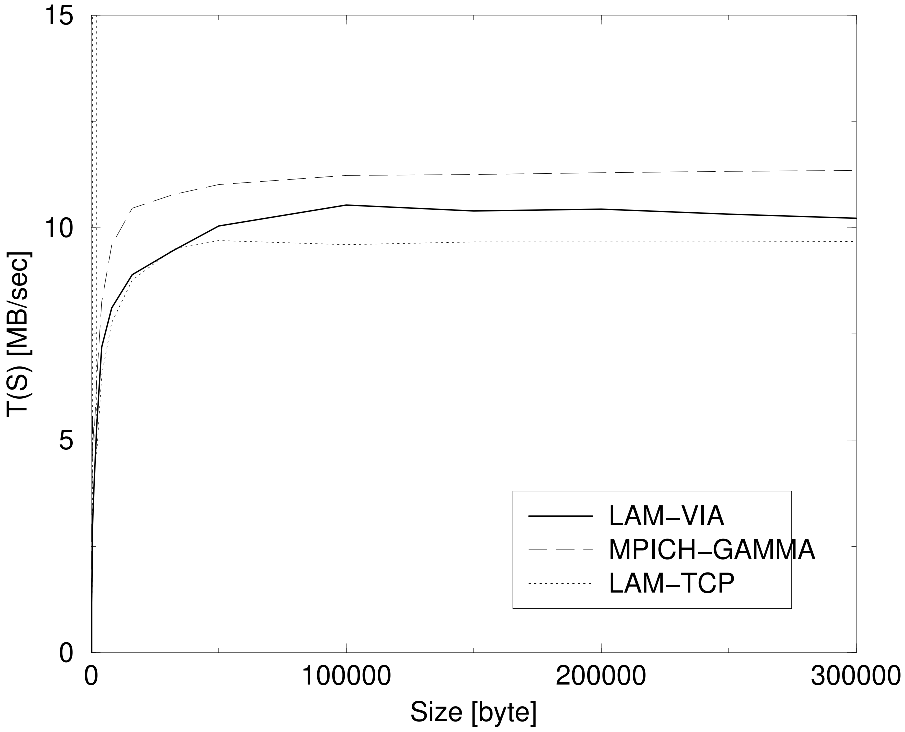

Figura 7.6: Confronto di velocitá di trasmissione tra LAM-VIA, MPICH-GAMMA e LAM-TCP

Size(bytes)	LAM-VIA(sec)	MPICH-GAMMA(sec)	LAM-TCP(sec)
1	0.000063	0.000025	0.000124
2	0.000063	0.000025	0.000122
4	0.000063	0.000025	0.000129
8	0.000063	0.000025	0.000129
16	0.000077	0.000025	0.000127
32	0.000081	0.000027	0.000131
64	0.000084	0.000033	0.000137
128	0.000100	0.000044	0.000149
256	0.000124	0.000065	0.000174
512	0.000170	0.000092	0.000224
1K	0.000263	0.000178	0.000322
2K	0.000385	0.000319	0.000439
4K	0.000570	0.000497	0.000628
8K	0.001010	0.000855	0.001055
16K	0.001844	0.001568	0.001869
32K	0.003461	0.003040	0.003452
50K	0.004982	0.004539	0.005159
100K	0.009492	0.008906	0.010419
150K	0.014428	0.013339	0.015531
200K	0.019159	0.017709	0.020702
250K	0.024236	0.022082	0.025883
300K	0.029363	0.026454	0.031008
350K	0.034623	0.030837	0.036379
400K	0.039672	0.035215	0.041649
450K	0.044746	0.039692	0.046826
500K	0.049643	0.044293	0.051974

Tabella 7.3: Confronto di ritardi di trasmissione tra LAM-VIA, MPICH-GAMMA e LAM-TCP

Size(bytes)	LAM-VIA(MB/sec)	MPICH-GAMMA(MB/sec)	LAM-TCP(MB/sec)
1	0.015872	0.039871	0.008035
2	0. 032800	0.079022	0.016328
4	0.063974	0.158468	0.031127
8	0.126026	0.316380	0.062253
16	0.206489	0.633566	0.126026
32	0.397388	1.205580	0.245147
64	0.646443	1.937897	0.467251
128	1.185146	2.904700	0.856253
256	1.917396	3.924338	1.467362
512	3.020371	5.563809	2.290649
1K	3.893896	5,759194	3.175577
2K	5.313088	6.421357	4.660211
4K	7.185975	8.251517	6.522350
8K	8.114717	9.587751	7.764913
16K	8.887384	10.457645	8.766078
32K	9.459685	10.780536	9.493771
50K	10.036141	11.015475	9.691762
100K	10.535210	11.227889	9.597895
150K	10.396480	11.245020	9.658398
200K	10.438948	11.293503	9.661123
250K	10.315249	11.321422	9.658842
300K	10.216927	11.340446	9.674921
350K	10.108879	11.350004	9.621067
400K	10.082552	11.358921	9.604076
450K	10.056876	11.337173	9.609947
500K	10.071917	11.288486	9.620107

Tabella 7.4: Confronto di velocitá di trasmissione tra LAM-VIA, MPICH-GAMMA e LAM-TCP

Capitolo 8

Conclusioni

Lo studio presentato in questa tesi riguarda l'estensione di funzionalitá di comunicazione del progetto di integrazione di VIA, Virtual Interface Architecture, come protocollo per il trasferimento dati all'interno della libreria di message passing LAM-MPI.

In particolare si risolve una delle principali limitazioni presentate da VIA, consentendo comunicazioni con messaggi di dimensioni maggiori di 32KB. E' stato quindi implementato un algoritmo di frammentazione e riassemblaggio dei pacchetti che permette trasmissioni di messaggi di dimensioni elevate, aggirando cosí il limite di VIA in modo del tutto trasparente all'applicazione, per mantenere anche conformitá con lo standard MPI.

Per verificare l'efficienza di VIA in sostituzione dei protocolli tradizionali, sono state valutate le prestazioni delle comunicazioni sul cluster ParMa^2, confrontando inizialmente la versione di LAM su VIA con quella originale di LAM, che utilizza il TCP.

I risultati sono valutati in termini di latenza e di banda di trasmissione. Lo scopo di valutare la latenza é quello di caratterizzare la velocitá di sincronizzazione di due processi, che é di fondamentale importanza per applicazioni distribuite. La tecnica impiegata é l'utilizzo di un ping-pong tra due processi: un processo (ping) invia un messaggio di dimensioni minime all'altro (pong), che riceve e ritrasmette. La metá del tempo calcolato per

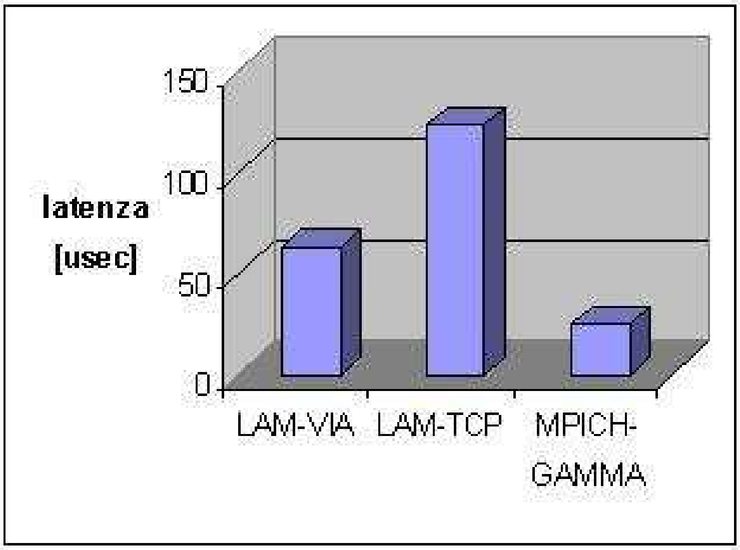

Figura 8.1: Confronto tra latenze

eseguire il ping-pong é la latenza. Il calcolo della banda asintotica, é invece una caratterizzazione di quanto velocemente avviene la trasmissione dei dati. Allo scopo di rendere il calcolo indipendente dall'overhead, ma significativo solo per il trasferimento dei dati, é valutata la velocitá di trasmissione per messaggi di dimensioni elevate.

L'algoritmo implementato, che permette la comunicazione con scambio di messaggi anche di dimensioni molto elevate, non influisce sensibilmente nelle prestazioni, ed evidenzia sia in termini di latenza che di banda asintotica, un miglioramento della versione di LAM su VIA rispetto a quella basata su TCP. Le diverse latenze sono infatti di 63usec per LAM-VIA contro i 124usec di LAM-TCP. Per quel che riguarda la banda si ottengono invece 10.2 MB/sec per la versione con VIA rispetto ai 9.8 MB/sec per quella con TCP.

Successivamente sono state analizzate l'architettura e le prestazioni di GAMMA, un prototipo di sistema per il trasferimento di messaggi realizzato presso il DISI dell'Universitá di Genova. Gli uguali obiettivi alla base dello sviluppo dello standard VIA e di GAMMA, uniti alla loro compatibilitá hardware rendono significativo un confronto tra i due, sia in termini di prestazioni di base che di comunicazioni utilizzando la libreria MPI. Il confronto delle prestazioni ha evidenziato, come previsto, migliori risultati ottenuti con MPICH-GAMMA. In termini di latenza si registrano infatti 25usec, che confrontati con i 63usec di LAM-VIA, mostrano la diversitá tra le due implementazioni. In termini di banda si registrano invece 11.3 MB/sec di MPICH-GAMMA contro i 10.2 MB/sec di LAM-VIA.

Dallo studio dell'architettura di GAMMA sono emerse sostanziali differenze con VIA che spiegano i risultati ottenuti e ne giustificano l'impiego nella comunicazione all'interno di una libreria MPI. VIA mostra infatti una riduzione apprezzabile della latenza, ma la caratteristica che ne ha fortemente motivato l'utilizzo è che questo si presenta come uno standard per le comunicazioni, permettendo compatibilità tra applicazioni sviluppate anche per diverse piattaforme hardware e software. Molti altri protocolli di comunicazione sono infatti stati sviluppati come argomento di ricerca o progetti sperimentali. VIA invece è il prodotto di tre industrie leader nel settore e ne riflette quindi gli interessi comuni e le prospettive commerciali. Inoltre i numerosi consensi che lo standard ha riscontrato lasciano supporre un suo largo impiego in futuro.

Il passo successivo per l'evoluzione del progetto é quello di eseguire dei programmi specifici, come i benchmaks NAS, per valutare il comportamento della libreria MPI su VIA con diverse tipologie di applicazioni: alcune rivolte principalmente alla comunicazione, altre alla computazione. I risultati che si otterranno sono importanti se confrontati con quelli ottenuti da altre implementazioni, allo scopo di meglio caratterizzare LAM su VIA di quanto non si possa fare con un semplice ping-pong, ed anche per meglio individuare dove intervenire per il miglioramento delle prestazioni.

Inoltre sará interessante attendere la nouva versione di M-VIA, chiamata M-VIA 2, che si propone di risolvere numerosi bugs rilevati nella versione attuale e di estendere le funzionalitá non ancora inplementate. Stanno anche facendo la comparsa sul mercato le prime schede hardware che implementano VIA a cui si potrebbe far riferimento per riprogettare il cluster ParMa^{2}per ottenere alte prestazioni.

Appendice A

Strutture dati

```
/* VIA-based protocol specific constants */

#define RECV_DESC 100
#define SEND_DESC 10
#define RECV_THRESHOLD 20
#define RECV_BUFF_DIM 32
#define SEND_BUFF_DIM 32*1024
#define SEND_ENV_DIM 28
#define SIZE_ENV 7
#define MAX_TRANSFER_SIZE 32*1024
#define RDMAAREA_DIM 1024*1024
#define RDMAAREA_FRACTION 256*1024
#define STANDARD_ALIGNMENT 64
#define RESTART -1

/* VIA-based protocol specific data structures */

/*  Process RDMA Area  */
struct RDMAAddress {
   char *RDMAArea; /* RDMA Area to store the incoming messages */
   VIP_MEM_HANDLE RDMAAreaHandle; /* RDMA Memory handle */
   int RDMAmaxread; /* bytes read by the process in its RDMA Area */
   VIP_MEM_HANDLE RDMAmaxreadHandle;
   int peerDescRead;
   VIP_MEM_HANDLE peerDescReadHandle;
   int needDesc;
   VIP_MEM_HANDLE needDescHandle;
```

```
};

/* Peer process RDMA Area */
struct remoteRDMAAddress {
  char *RDMAArea; /* peer RDMA Area address to store messages */
  VIP_MEM_HANDLE RDMAAreaHandle;
  VIP_PVOID RDMAmaxread; /* bytes read by the peer*/
  VIP_MEM_HANDLE RDMAmaxreadHandle;
  VIP_PVOID peerDescRead;
  VIP_MEM_HANDLE peerDescReadHandle;
  VIP_PVOID needDesc;
  VIP_MEM_HANDLE needDescHandle;
};

/* Virtual Interface data; one instance for each connected process */
struct VIInfo {
  int4 rank; /* peer process's rank */
  VIP_NIC_HANDLE nic; /* VIA nic for this VI (via_lo or via_ethx) */
  VIP_PROTECTION_HANDLE ptag;
  VIP_VI_HANDLE vi;
   /* descriptors */
  VIP_DESCRIPTOR *sendDesc;
  VIP_DESCRIPTOR *recvDesc;
  VIP_MEM_HANDLE sendDescHandle;
  VIP_MEM_HANDLE recvDescHandle;
      /* buffers for data to receive and to send and for envelopes to send */
  char *sendBuff;
  int *envBuff;
  char *recvBuff;
  VIP_MEM_HANDLE sendBuffHandle;
  VIP_MEM_HANDLE envBuffHandle;
  VIP_MEM_HANDLE recvBuffHandle;
  int recvNum; /* current receive buffer and descriptor */
  int sendNum; /* current send buffer and descriptor */
  int RDMAreadpointer; /* pointer to RDMA Area last read byte*/
  int RDMAreadlimit;
  int RDMAnextpointer; /* pointer to RMDA Area first free byte;*/
  int RDMAwritepointer; /* pointer to peer RDMA Area last write byte*/
  int RDMAmaxwrite; /* pointer to peer RDMA Area last writable byte*/
```

```
  VIP_MEM_HANDLE RDMAreadlimitHandle;
  int descRead;
  int descLeft;
  int buff;
  VIP_MEM_HANDLE buffHandle;
  struct RDMAAddress addr; /* process RDMA Area */
  struct remoteRDMAAddress remoteAddr; /* remote RDMA Area */
};

struct via_proc {
  VIP_NIC_HANDLE nic;
  VIP_NIC_HANDLE lonic;
  VIP_PROTECTION_HANDLE ptag;
  VIP_PROTECTION_HANDLE loptag;
  VIP_NIC_ATTRIBUTES nicAttrs;
  VIP_NIC_ATTRIBUTES nicLoAttrs;
};

#define HASH_ELEM_SIZE sizeof(struct VIInfo)
#define HASH_OP_MODE 0
#define HASH_NULL_KEY -1 /* First element (NULL) of the hash table */
#define ENTRYCOUNT 1024 /* Max entries in the Completion Queue */
#define ADVANCED 1
#define MESSAGE 0

/* Struct rq_rpi.c2c */
struct via_req {
  int nack;
  int issend;
  VIP_MEM_HANDLE buffHandle;
};

/* Struct in which memorize msg to self */
struct buffer {
  int rank;
  int tag;
  int len;
  int cid;
  int seq;
```

```
  MPI_Request requ;
  void *buf;
};

struct ptr_env {
  int *pointer;
  int offset;
};

#define LIST_PTR_SIZE sizeof(struct ptr_env)
#define LIST_BUFF_SIZE sizeof(struct buffer)
#define LIST_RANK_SIZE sizeof(int *)
```

Bibliografia

[1] VIA Org. Specification for the Virtual Interface Architecture. http://www.viarch.org/default.htm .

[2] Intel. Intel and VI architecture. http://developer.intel.com/design/servers/vi/developer/ia_imp_guide.htm http://developer.intel.com/design/servers/vi/.

[3] Compaq, Intel, Microsoft. *Virtual Interface Architecture Specification*, December 1997.

[4] Thorsten Von Eicken and Werner Vogels. Evolution of the Virtual Interface Architecture. *IEEE Computer*, November 1998.

[5] Intel. *Intel Virtual Interface (VI) Architecture Developer's Guide*, September 1998.

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

[7] Philip Buonadonna, Andrew Geweke, and David Culler. Implementation and analysis of the Virtual Interface Architecture. November 1998.

[8] Franco Boselli. Integrazione della Virtual Interface Architecture (VIA) all'interno della libreria di Message Passing LAM/MPI. Master's thesis, Università degli Studi di Parma - Facoltà di Ingegneria, 1999.

[9] Giuseppe Ciaccio Giovanni Chiola. Gamma: Architecture, programming interface and preliminary benchmarking. November 1996.

[10] Giovanni Chiola. Gamma project: Genoa active message machine. http://www.disi.unige.it/project/gamma .

[11] Culler Von Eicken and Schauser Goldstein. Active message: a mechanism for integrated communication and computation. *Computer Architecture News*, May 1992.

[12] Raykumar Buyya. *Hight Performance Cluster Computing*. PH PTR, 1999.

[13] Giuseppe Ciaccio Giovanni Chiola. Porting mpich adi on gamma with flow control.