December 14, 2001

To the Graduate School:

This thesis entitled "Design and Analysis of a Network Transfer L
File Systems" and written by Philip H. Carns is presented to the G
of Clemson University. I recommend that it be accepted in parti
the requirements for the degree of Master of Science with a maj
Engineering.

Walter B. Ligon III, Advis

We have reviewed this thesis
and recommend its acceptance:

Ron Sass

Adam Hoover

Accepted for the Graduat

Design and Analysis of a Netwo
Transfer Layer for Parallel File S

A Thesis

Presented to

the Graduate School of

Clemson University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Computer Engineering

---

by

Philip H. Carns

December 2001

Advisor: Dr. Walter B. Ligon III

Abstract

This document describes the design and analysis of a network t
the Parallel Virtual File System (PVFS). PVFS [6] is a parallel file sy
at Clemson University for use on Linux based Beowulf [23, 2] cluster
to serve as a platform for high performance I/O research while also n
for a production parallel file system for the scientific computing co
time, computational capability has improved so rapidly that there is
large gap between I/O performance and processing power, even in th
computer systems. This has led to a situation in which file system
the primary bottleneck for a variety of applications. PVFS seeks
lution to this problem. Many components must be brought togethe
this, including network communications, data storage, application
faces, and scheduling. This document will focus on network comm
intend to demonstrate and analyze a system for improving networ
usability for the purpose of high performance I/O. This system will
of lessons learned from current file system implementations, as wel
cluster communication technology, to help achieve the specialized goa
systems.

Dedication

To all of my family and friends who have supported me through

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Walt Ligon for his support
would also like to thank Dr. Rob Ross for his invaluable help.

Table of Contents

List of Figures

Figure

Chapter 1

Introduction

1.1 Beowulf clusters

Beowulf clusters have become increasingly popular over the last few y processing tasks [23, 2]. The beowulf architecture consists of a collecti workstations connected through a dedicated local network. These sy source system software to provide the features expected of a parallel as message passing, process management, and global file storage s intended to be used almost exclusively for parallel applications, a restricted to maintaining the functionality of individual workstation cost effective approach to performing computationally intense resea of the components are either available as commodity hardware or fr software.

Beowulf clusters have also provided many new avenues for syste search. One reason for this is the diversity inherent in building cus machinery out of commodity parts. For example, this approach mak lenging to create algorithms and scheduling policies that work wel case, because almost no two clusters are exactly alike. Today it is no to just tune software for a specific vendor's hardware implementatio

made on one architecture may not be valid on another.

Systems research is also encouraged by the use of open source softw

to inspect, modify and redistribute modifications to even the most lo

components makes it relatively easy to contribute improvements to

This leads to a cycle of gradual improvement in overall Beowulf clus

1.2 Parallel file systems

One important element of Beowulf system software is the file system.

increasingly important as processor speeds continue to increase at a

data transfer speeds. In the past decade, CPU speed has increased

1000MHz (a 100 fold increase) while average disk transfer speeds hav

3 Mbytes/sec to 12 Mbytes/sec (only a four fold increase) [13].

applications that require large amounts of file I/O are finding disk

increasingly larger bottleneck compared to computation time. In o

problem, we must make more efficient use of existing storage technol

do this is through the use of parallel file systems.

Parallel file systems are used to distribute data over several in

age devices. This data is then presented to all of the application

consistent name space. This combination of parallelism on both the

storage side allows the I/O load to be distributed across an entire cl

a subset of the cluster) so that there is no single bottleneck point fo

bandwidth and storage capacity of this arrangement is typically m

what can be obtained by using a single shared file server.

One example of such a file system is the Parallel Virtual File

[6] which was developed at Clemson University. PVFS was design

platform for high performance I/O research on Linux clusters. It has a stable production file system for use in the cluster computing commu PVFS makes use of common Linux features and brings them toget homogeneous parallel file system. In its current form it utilizes the file system available at each node for data storage and the TCP/IP p communication between nodes.

1.3 New technologies

Since the initial PVFS design, many new technologies have become could potentially impact parallel file system design for Linux clusters developments have gradually reached near-commodity status, and reliable open source system software has grown tremendously.

1.3.1 Networking

Networking infrastructure for clusters has experienced rapid develo first Beowulf clusters were built, both in terms of hardware and soft common hardware available for the earliest Linux clusters was 100M Since that time, Gigabit Ethernet has also become commodity har Ethernet provides much higher network bandwidth without introdu in application software. Several vendors have also introduced special working hardware, some of which can take advantage of customized as well. While this hardware has not reached consumer level comm is much more affordable than the custom networking hardware trad commercial supercomputers. Hardware of this category includes (b to): Myrinet hardware from Myricom, Inc. [3], SCI hardware from D nect LLC [1], and Giganet hardware from Emulex Corporation [10].

There are also more choices in network software and protocols. T been the standard for internet communications and was easily ad cluster use. However, TCP/IP has disadvantages in some environ the overhead introduced in TCP/IP to handle geographically larg unreliable hardware simply is not necessary in a typical cluster e Beowulf clusters, by definition, possess dedicated local networks. This allows the use of much lighter weight protocols.

Alternatives for network software and protocols include the Virt chitecture [28], Score/PM [26], GAMMA [8], Active Messages [29], an software such as GM [19]. Some of the features that may be provide networking systems such as these are:

- User level operation that bypasses the overhead of interacting w system kernel during communication
- Efficient abstraction of the underlying hardware
- Lightweight transmission protocols
- Low level programming interfaces that allow developers to avoi overhead software features

Shared memory is another technology that has been around for is perhaps now easier to use on Linux clusters. Multiprocessor syste available at commodity prices. They allow faster interprocessor co some cases by using local shared memory rather than external netwo At the same time, some vendors have produced products which e shared memory across a collection of nodes that would normally on through traditional message passing.

1.3.2 Data storage

The most interesting technological advancements in commodity data
have come from software developments. The Linux platform now ha
brary support for Posix asynchronous I/O [5], while the Linux kernel
for raw I/O. Asynchronous I/O allows multiple non blocking file I/
be initiated and later checked for completion. This potentially allo
cient handling of multiple requests and the ability to overlap other
with file I/O. Another new development is the Raw I/O interface. It
for directly accessing disk devices at the block level without using th
cache and abstraction path. This opens up the possibility of writing
handle their own caching and device I/O independent of kernel algor
potentially boost performance of applications that have very specific I
needs that contradict generic operating system policies.

1.4 New research findings

In addition to advances in commonly available software and hard
recent research and implementation has increased the knowledge ba
parallel I/O. There are many new ideas and implementation lessons
used in parallel file system design.

1.4.1 Scheduling

One of the most important research topics explored in the first g
design is Reactive Scheduling [24]. Reactive Scheduling is a new app
server side scheduling decisions for parallel file systems. The main g
ically choose appropriate scheduling policies depending on the stat

This is in contrast to the traditional approach of trying to optimize a s strategy to meet all of the needs of the file system.

The state of the system can be determined by system paramete work or disk utilization, and also by the workload produced by the ap parameters can be used as input to a system model. This system n cates what scheduling policy should be used to obtain the best pe dynamically switches to this policy. The scheduling policies are chos research, and could include ideas such as disk directed I/O, netwo or two phase I/O. The most important concept is the ability to cor which policy is best suited to the current state.

This research has shown that scheduling decisions have an impo parallel I/O performance. We must be able to support efficient, mo in future work. Further work can also be done to explore policy de levels of the file system abstraction.

1.4.2 MPI-IO

MPI-IO is a standard application interface for performing parallel I/ was released as part of the MPI-2 specification in 1997 [12, 18]. It pro interface for both C and Fortran applications. MPI-IO provides o discontiguous and parallel file access (through features such as de and collective I/O).

MPI-IO has been widely adopted. Several implementations, such are available. This has encouraged the creation of portable applic advantage of high performance I/O. The traditional portable Unix I/ not provide many of the features necessary to achieve efficient para and the vendor specific I/O interface implementations do not work

native hardware. Thus it has become important to support MPI-IC
features that make it's implementation easier and more efficient.

To obtain the best performance in an MPI-IO implementation, i
given file system to provide two features. The first is the ability to des
discontiguous patterns similar to those common in MPI-IO. This red
of data packing and translation that must occur outside of the file sy
the file system needs to provide an efficient, high throughput interfa
a layer of abstraction which can potentially be detrimental to perf
important to lower the overhead in the I/O path as much as possible
penalty.

1.4.3 Software engineering

PVFS has gradually become an accepted tool for use in production en
the past few years. This has lead to its use in many diverse situations
use of PVFS has made it important to locate and correct software e
as possible. It has also forced the developers to continually update P
track changes in technology.

From this we have learned the importance of thorough software e
tice. This includes modular design, well defined interfaces, and com
mentation. The use of these practices makes it much easier to support
project such as a parallel file system. New design decisions must ac
lessons in order to be successful over the life of the project.

1.5 A new file system design

All of these changes in technology, as well as new information gaine
and implementation, have prompted the design of a new parallel file

PVFS. This new file system is currently transitioning from design to

It will build upon new ideas and knowledge in order to provide a more

file system for Linux clusters.

The next generation Parallel Virtual File System will be made up

ponents. Some of the most important components include the networ

anism, the storage transfer mechanism, the application interfaces, the

and the scheduling mechanisms. Each of these components (and sev

be necessary in order to build a successful implementation.

1.6 Network layer requirements

This document will focus on just one component of a parallel file sy

computers: the network transfer layer. The network transfer layer i

moving data between processes on a parallel computer. There ar

provide this functionality, but parallel file systems impose many re

the design of such a component. The special needs of parallel I/O an

learned from current designs have prompted the following list of req

- *Simple application interface*: The interface to the network tran be concise and efficient. It should well suited to describing th munication most often needed to perform parallel I/O, with additional complexity in the design of other file system compo

- *Overlap of network I/O with other system tasks*: The netwo should be designed to allow other application activity to cont I/O tasks are performed. This is of particular importance to s tation, where data storage I/O can be performed simultaneou I/O in many cases to improve efficiency. This will become eve multiprocessor systems become more common in the commo thus more common in cluster applications. Multiprocessor nod efit significantly from the ability to overlap network communic tasks.

- *Support for both user level and kernel level network API's*: A section 1.3.1, there are now a variety of approaches to network c Some of these approaches include operating system interacti perform I/O directly from the user level. It is important for i be able to utilize both types of access efficiently.

- *Abstraction and modularity*: Access to the underlying network s should be abstracted from the user level system components. T the core design and algorithms of the parallel file system from b bound to a specific networking technology. It should be possi completely replace the underlying network technology withou implementation of other system components.

- *Efficiency*: Almost any software abstraction layer induces a perf to the application. The network transfer layer should seek t penalty as much as possible. Network I/O is a performance bot common situations, and we cannot afford to constrain it furt for additional features. The file system will be sensitive to both latency overhead.

- *Ability to interact with multiple networks simultaneously*: If the mechanism is not bound to a single network device, then it o sibility of more exotic cluster topologies. Hosts may interact w exist on dissimilar networks in order to take advantage of th communication route to each.

1.7 Approach

Advances in technology and software engineering have suggested th abstraction that can support a variety of network protocols would be development of parallel file systems. We believe that it is possible to abstraction that supports multiple protocols in this manner while s high performance. The *Buffered Method Interface* (or *BMI*) has be as a platform for testing the feasibility of such an interface. This im intended to meet the requirements listed in section 1.6.

The remainder of this document is organized as follows. First, r be discussed in order to provide background for the BMI design. outline the actual architecture of the Buffered Message Interface. BMI implementation on top of various network protocols will be used its feasibility.

The results section will provide an analysis of the performance Message Interface. We will compare BMI performance to standar network data transfer to verify its effectiveness and determine if it m ments. Finally, we will present the conclusions based on the findings and propose future work.

Chapter 2

Background and related wor

2.1 The Parallel Virtual File System

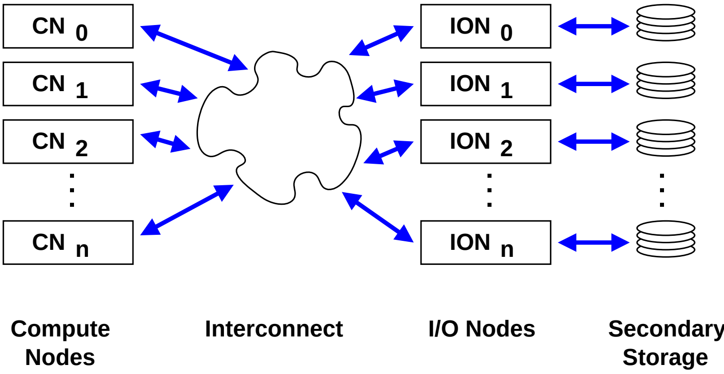
2.1.1 Motivation and goals

PVFS is a parallel file system for Linux clusters that was develo
University. It was originally designed to serve two main purposes.
intended to be a platform for parallel I/O research. Secondly, it is i
the high performance community's need for a parallel file system fo
It has been successful in both of these goals, prompting several rese
gaining acceptance as a high performance file system for use on pro

The following is a list of some of the key goals of PVFS:

- High bandwidth for parallel read and write operations to a sin
- Flexible application interfaces, including support from the RO
  MPI-IO [27]
- Compatability with existing applications that use the native U
  [14]
- Ability to tune file system parameters from the application le

Figure 2.1: PVFS System Overview



- Scalability
- Robustness
- Ease of installation

PVFS emphasizes performance and research viability over high tures. It therefore does not provide software level redundancy. It also any locking mechanism within the file system itself, nor any advan curity features such as encryption. These features are best met by projects.

2.1.2 Architecture and implementation

The Parallel Virtual File System is implemented almost entirely i does not require any kernel level support for its default mode of oper optional kernel level support is required to obtain compatability w that interact with the Unix I/O API [14]. PVFS makes use of exis for its most low level operations, including TCP/IP for networking a Linux file system (such as EXT2 or ReiserFS) for file data storage across 32 bit and 64 bit Linux architectures.

The basic layout of a PVFS system is shown in Figure 2.1. File across multiple cluster nodes that are connected by a local area netw hardware is required since PVFS makes use of existing operating s Each node that possesses a portion of the file system data must run o PVFS daemons that make the resources of that node available to Client applications may run on these server nodes, or they may run or that are connected by the local area network. Client applications ma system through either a user level library, the ROMIO MPI-IO im the Linux kernel interface (outlined in section 2.1.4).

Manager

PVFS is made up of three primary components. The first is the r is exactly one manager per file system, regardless of its size. It is maintaining metadata. In PVFS, metadata refers to the collection o characteristics of files stored on the file system. This includes infor ownership and permissions that are not actually part of the file data

In addition to the standard Unix file properties such as those l manager also maintains metadata that is unique to PVFS. This inclu distribution information for each file. This is used to determine whic the file system possess file data, and how the file data is distributed

The PVFS manager serializes all metadata operations from clie metadata consistency. This is made easier by the fact that there is on eliminating the need to maintain synchronization with another nod metadata information. It may seem like a performance bottleneck to manager, but this is eased by the fact that the manager does not p I/O operations. Once a client has verified permissions and metadata

does not communicate with the manager when reading or writing fi
handled by the PVFS I/O daemons, which will be outlined shortly.

I/O daemon

The I/O daemon is responsible for servicing I/O requests and storing
may be any number of I/O daemons, from as few as one to as many a
of the system will permit. Each I/O daemon stores data on the loc
the node that it is running on. When multiple I/O daemons are bein
is striped across them in round robin fashion. The stripe size, offset
I/O daemons to use can be specified by the user on a per file basis.

The use of multiple I/O daemons introduces parallelism on the s
file system. A client may have parts of its request serviced from
servers, thus leading to utilization of several separate disks and netw
simultaneously, rather than waiting for service at one particular b
This style of access is tailored to improving throughput for paral
especially those which demand large amounts of I/O. It may not be
parallel applications, because the client becomes a bottleneck for t
thus negating the advantage gained by having servers operate in par

Each I/O daemon operates independently of other I/O daemons i
is only aware of the portions of a file that it is in control of at any
distribution remains static over the lifetime of a file in PVFS.

Client library

The PVFS client library enables applications to interact with the
is a C library for use in user level programs, and does not require
with the Linux kernel for communication. It provides a native PVF
derived from the standard Unix I/O API. Among other things, it in

for opening, closing, reading, and writing to PVFS files. It also ad specify PVFS specific parameters for files, such as physical stripe siz I/O daemons to use.

The native PVFS library also provides the ability to make di requests. This is an important feature for parallel applications that in standard Unix interfaces. This is done by using a *partitioned fil* partitioning allows a process to alter its view of a file logical file so t discontiguous regions with single read or write requests. It is similar t file partitioning [9] and to file views provided by MPI-IO [12, 18].

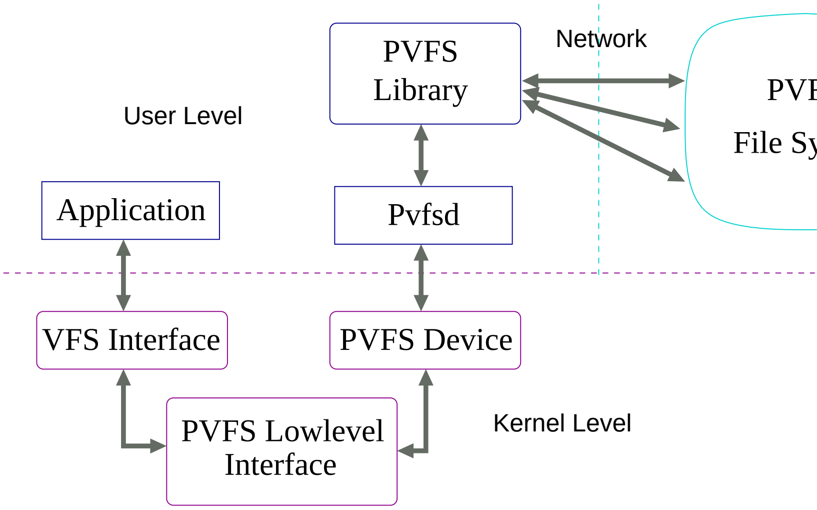
The PVFS library is responsible for orchestrating communicatio ager and any I/O daemons as necessary. For large systems, this may nicating with hundreds of servers. All of this is hidden from the app

2.1.3 Low level I/O

All PVFS data is stored on standard local file systems. Each I/ its portion of the data files on a local file system, and the manager information on its local file system as well. This is in contrast to implementations which write raw data to disks. By avoiding the use of complexity is reduced, and PVFS benefits from features of the local as caching or journaling.

All inter-node communication is carried out using the standard T This was the primary communication mechanism available for Linu PVFS was first designed. It provides reliable, ordered delivery and data communications. It is also available on almost every Linux clu

Figure 2.2: PVFS kernel architecture



2.1.4 Unix I/O compatability

The PVFS client library provides an optimized path for applications PVFS. It also provides access to PVFS specific parameters. Howeve patible with existing applications. This prompted the design of tl package as an alternative. It provides a kernel path for PVFS so t can interact with it just as they would any other file system. When age, PVFS is mostly indistinguishable from a more traditional file s user's point of view.

The architecture of the PVFS kernel implementation is shown i consists of both a user level and kernel level component. The kernel l are implemented as a module, while the user level components are in client side daemon known as the pvfsd. The pvfsd is responsible for a ing file requests to native PVFS requests and communicating them o to the PVFS file system. This is done at user level because it prov bility than a full kernel implementation. It allows the client to util mechanism that is required (some of which may not be available with system kernel). It also allows the use of the standard PVFS libra requests, rather than maintaining an independent interface within th

Communication between the user level pvfsd process and the k carried out through use of a special device file. Requests are read out by the pvfsd, and responses are written back into it. Several method bulk data through this interface are provided, but their operation is b of this document.

This implementation attempts to be as modular and portable as it can survive multiple generations of file system design. All code spec file system implementation is maintained within a strict interface.

2.2 Virtual Interface Architecture

2.2.1 User level networking background

Communications hardware has advanced rapidly in recent years, crea alternatives for high performance communications. Most Linux clu nally constructed with 10 Mbit or 100 Mbit Ethernet, but such netwo been augmented with more advanced commodity options, such as G as well as specialized system area networks, such as Myrinet [3].

These advances in hardware have prompted research into the so communication as well. Cluster specific system area networks, in benefit from software interfaces and protocols which take advantage that do not apply to general case local area networks. There are se this level of communication that may be targeted for improvemen weaknesses in the use of general purpose communications in clusters of protocol complexity and the amount of operating system intera place during data transfer [4]. Protocol complexity arises from mul

implementations that provide a wide abstraction from hardware, as
inclusion of a large suite of features which may or may not be util
system overhead arises from relying on the system kernel to provide
tion and resource allocation. These issues can be addressed by desi
systems that more closely match the ability of the underlying hardw
removing the kernel as much as possible from the critical path of data

This broad approach to optimizing application interaction with
been termed *user level networking.* This approach was largely pioneer
Messages project [29], which also encompassed several other topics
tion and network interaction. This work continues today and has pr
research to include specific vendor offerings, such as GM [19], as wel
industry initiatives such as VIA [28].

2.2.2 VIA specification

The Virtual Interface Architecture is an attempt at standardizing bo
and interface to user level networking across a variety of hardware a
dors and implementors. It officially came into being in 1997 with t
Virtual Interface Architecture Specification, which was the result o
being several leading vendors, including Microsoft, Compaq, and Inte
ument outlines the architecture of VIA from both an implementation
of view. It draws heavily from well known research efforts such as
but its aim is to provide a usable, production level standard for user
on system area networks.

2.2.3 Architectural overview

The Virtual Interface Architecture model consists of several compo
the *VI Provider*, *VI Consumer*, *Virtual Interfaces*, and *Completion*

The VI Provider is made up of the hardware and software compo
act to provide the resources necessary for a virtual interface. These r
memory protection, connection setup and teardown, and error ma
hardware utilized is generally a network interface card. The hardwar
ically designed to support VIA (and is thus considered “VIA aware”
more traditional design, for which certain features of the VIA archi
emulated. The software component of the VI Provider is usually a ke
Note that this kernel driver is only responsible for a limited number
is typically only invoked during initial setup of communication betwe
is not directly involved in the data path for communication.

The VI Consumer can roughly be thought of as the user of th
This is of course includes the application, but also encompasses th
the application uses to interact with the other VI components. T
interface is normally implemented as a user level library, and it has a
set of functions that are used to invoke communication mechanisms.
trigger the necessary kernel level setup mechanisms and abstract th
the other components.

The Virtual Interface itself is the mechanism that allows the cons
interact with with the provider in order to actually transfer data
Virtual Interface per peer that the host wishes to communicate w
work queues for both send and receive operations. These work q
concept in understanding how VIA operates. The queues do not actu
data to be transfered in communication operations. Instead, they c
descriptors that describe the data to transfered, as well as other
status information) that are necessary to describe the communicat
VI posts descriptors to the work queues, it uses *doorbells* to indicate
adapter that new work is available. Doorbells are very small, simple

intended to be implemented directly from the hardware so that no o

intervention is required to notify the hardware. The doorbells may needed, however. The goal is to provide information to the netwo efficiently as possible so that it can use hardware mechanisms such transfers to move application data, rather than relying on additio buffers for this purpose.

Completion Queues are provided as a mechanism for the VI to noti that messages have been completed. Once descriptors have been pro work queues, their status is filled in to indicate success or failure. be transfered to completion queues that were specified by the Consu application can be made aware of the completion. Again, the goa status changes and to transfer information about communication w kernel context switches or unnecessary interrupts.

2.2.4 Usage

The VIA model for communication implies that the application has bilities in the communication process than would be expected of trad interfaces. First of all, the application must be able to create and scriptor and doorbell structures that are necessary for initiating da must explicitly post these structures and inspect them upon complet

Additional constraints are also placed upon the memory regions cation may use for communication. Networking hardware normally r data to be locked into physical memory and specified in terms of phys virtual addresses before being sent across the network. Traditional n implementations allow the operating system driver to handle this req than expose it to the user. VIA, however, requires that the user ex this memory registration before submitting a descriptor that refere

Only memory that has been registered with the VI Provider may be

advance registration allows the Provider to directly access the region

an intermediate buffer. These regions may also be reused, so that

registration need not be incurred for every message.

The communication semantics of VIA are very lightweight. For e

sage buffering is provided for receive operations. This means that a

must be posted before the message data arrives, or else it may be

introduces more management responsibilities on the part of the a

sure that buffers are provided in a timely manner. In addition, the

outlines three levels of reliability which may optionally be provided

reliability differ in the amount of assurance that the application is

the successful arrival of messages. If a VIA implementation does no

enough level or reliability for the needs of an application, it may requ

of software in order to provide this functionality.

Due to the above application implementation requirements, VI

used as a foundation for a higher level API, such as MPI [18], or for

tation of system software. These environments can take advantage

of the Virtual Interface Architecture without requiring a new learni

applications programmer.

2.2.5 Implementations

There have been several adopters of the VIA specification thus far

ones are listed below with a brief outline of the approach taken to p

features. This is a sampling of the level of acceptance within the h

computing community.

Berkeley VIA

The Berkeley VIA project (based at UC Berkeley) is a research orien
has provided cross platform VIA implementations for Myricom's M
[3]. They seek to provide a high quality VIA implementation, explore
characteristics, and investigate possible improvements to the architec
full implementation of the specification, but it provides enough funct
uate performance and support most VIA applications. The Myrine
it utilizes, though not designed explicitly for use with VIA, is highl
and provides most of the hardware features suggested for VIA imple

M-VIA

M-VIA is a research prototype VIA implementation from the Natio
search Scientific Computing Center [16]. Its primary features inc
design which should ease the work of porting to new hardware. It
implementation of the VIA standard and also allows coexistence with
which may be supported by target hardware devices. M-VIA curre
variety of Ethernet hardware by emulating in software some of the
features, though it can also take advantage of VIA accelerated ha
releases target a more ambitious range of networking hardware.

Giganet

Giganet is a vendor hardware offering from Emulex Corporation [1
full VIA software implementation, and network interface cards that a
VIA support in mind.

2.3 Virtual Machine Interface

The Virtual Machine Interface [21] is a high performance messaging A
The University of Illinois, Urbana-Champaign and the National Cente
puting Applications. It provides a uniform interface for interacting
networking systems. This interface may be used directly by an appl
as the foundation for implementing a higher level interface such as
provides connectionless, reliable, ordered delivery. It also requires
buffers to be registered, much like VIA [28].

VMI also provides several other advanced features that make it s
other network abstraction implementations:

- Different network types are supported through the use of dyna
  modules. This means that applications do not need to be recom
  take advantage of new devices, nor to acomidate changes in clus
- VMI provides collective notification of process failure. If a si
  computation crashes, then all peer processes are notified. Thi
  graceful termination of the computation as a whole.
- VMI is intended to be portable across dissimilar platforms. It
  with both Linux and Microsoft Windows cluster environments,
  to interact on the same computation.
- A single host is allowed to communicate over multiple devices
  This allows VMI to be used in heterogeneous network environ
  more, extensions to VMI allow it to serve as a bridge across diss
  thus allowing full interprocess communication when only a li
  nodes share connections to both networking systems.

As of this writing, VMI supports shared memory, VIA, and TCP munications. Support is planned for Myrinet and SCI networks as w

2.3.1 Shared memory

The VMI shared memory device is used between processes that ar same physical node within a cluster. It operates by providing a uniq and synchronization structure between each pair of communicating shared buffer consists of a contiguous 1 Mbyte region that is writabl and read only for another. It may be allocated in 1 Kbyte pages. Sy handled through a bounded circular queue that indicates the send a pair of nodes. Lock based synchronization is avoided because th writable by one process at any given time.

2.3.2 VIA

The VIA device has been implemented using the Giganet device driv provides many of the primitives required for VMI module implement it does not provide flow control. VMI therefore implements a credit b mechanism on top of VIA within the module.

2.3.3 TCP/IP sockets

The TCP/IP socket interface is not as close of a match to the VMI ments as VIA or shared memory communications are. Thus implem protocol is more difficult. VMI currently supports a proof of concept that has not yet been optimized for performance.

2.4 Message Passing Interface

The Message Passing Interface is a specification for application level r

It was defined by the MPI Forum in an attempt to provide a standard

bility between parallel computers from a variety of vendors and resear

to the drafting of the MPI specification, many vendors provided thei

braries for message passing which made it difficult to create portal

Many of these libraries shared the same fundamental features but d

terms of interfaces and syntax.

The initial MPI Standard (Version 1.0) was completed in May of

this standard was later continued, resulting in the 1.1, 1.2, and 2.0

[17].

The MPI Standard encompasses both the application interfaces a

the message passing system. This system provides many features,

to point as well as collective communication. It also provides oth

as consistent process naming, virtual topologies, heterogeneous data

formance monitoring tools, and profiling interfaces. The MPI-2 St

enhancements such as parallel I/O, remote memory operations, and

management.

MPI has quickly become the default message passing library for

tions. There are implementations available for every major modern

2.4.1 MPICH

MPICH (or MPI Chameleon) is a portable implementation of the

supported by Argonne National Laboratory [11]. MPICH is unique i

opment began while the initial standard was still being drafted. Th

MPI Forum with immediate feedback from a design that tracked th was developed. This also resulted in the availability of a working MPI as soon as the standard was finalized, which aided in its acceptance.

MPICH was originally constructed by taking advantage of exis preceding systems. One of these was parallel programming librar which provided portable shared memory and message passing compo was Chameleon, a package which focused on portability over a va passing architectures. The final precursor was zipcode, which provi scalable libraries, such as contexts and groups.

The two most important design goals of MPICH were portabilit MPICH runs on a variety of systems and provides low level interfaces f to quickly port it to other environments. However, it strives to sacrificing overall performance.

MPICH architecture

The MPICH architecture was carefully designed to meet the goals portability. It can best be described in terms of three major compo

- *High level code*: The highest level MPICH code includes man as groups, communicators, and opaque objects) which are ind communications mechanism. Therefore, this portion of the de as a portable implementation that can be expressed in term abstractions.

- *Abstract Device Interface*: All high level MPICH code is writt Abstract Device Interface (ADI). There are many separate im the ADI for different architectures. It provides an interface f to quickly integrate new architectures without having to rewr library from scratch.

- *Channel Interface*: One implementation of the ADI uses the ch The channel interface is a very small abstraction of the commu anism which can be implemented with as few as five functions the quickest path to implement a new device, but at the cost o

The idea is for implementors to rapidly prototype a new device support for it at the channel interface level. As the implementation pr levels of abstraction can be replaced by device specific code, so that tually has its own ADI implementation and perhaps even optimization functionality. Implementors therefore have the advantage of a porta tion for rapid prototyping but are not constrained by it in the long t

2.5 Bringing together related work

Three distinct messaging abstractions have been discussed in the p the Virtual Interface Architecture, the Virtual Machine Interface, a Passing Interface. These tools are all currently available and perform tasks quite well. However, the realm of parallel I/O on Linux clus needs which are not yet met by any single message passing impleme

In order to be successful within the field of parallel I/O, a netw needs to bring together a hybrid of several features. It must be robu small network failures so that errors on individual hosts do not imp activity. It furthermore must be capable of sustained client/server Unexpected hosts may connect and disconnect from a file system m the system is running.

A network abstraction for parallel I/O must also be efficient for main that it is solving. There are many features critical to general p ing tools that simply are not applicable in a client/server based syst

Supporting unnecessary features is almost undoubtably a source c

maintenance difficulties.

Finally, the network abstraction must share the file system's abi

ciently and reliably on a variety of existing cluster architectures. M

clusters can not afford to experiment with sweeping system level ch

of disrupting ongoing computational work. We want to support t

systems while also leaving the door open for work with more exoti

possible.

Chapter 3

Design of the network transf

layer

The Buffered Message Interface (BMI) has been designed to serve
transfer layer for a next generation parallel file system. It is impleme
that provides a standard interface for communication between syste
ponents. Although designed for use within the Parallel Virtual File
an independent entity which may be useful in other environments as

3.1 Communications model

BMI is a message passing system that provides reliability, ordering,
If a particular underlying network protocol does not provide one o
then BMI is responsible for implementing it.

All communications operations in BMI are nonblocking. In orde
sage, the user must first *post* the message to the interface, then *test* i
The same holds for receiving messages. Once testing indicates tha
completed, the user must check the status of the message in order t
completed successfully or not. Partial completion is not allowed.

In fact, every function defined as part of the BMI interface is no
function may perform work before completing, but this work is guaran
within a bounded amount of time. This restriction implies that it m
to test for completion of a message several times before it actually c
is no mechanism that allows the interface to “wait” indefinitely fo
a particular operation. This design decision was made because b
calls (especially in large parallel systems) are prone to problems
and scalability. They may cause an application to hang in the even
programming errors. This is not acceptable within low level system

When posting receive operations, the user must specify the addres
host and the size of the message to accept. The user cannot post rece
wildcard addresses. The only exceptions to this rule are unexpect
defined in section 3.3.

BMI is a connectionless interface; the user does not have to est
tain any link between hosts before sending messages. The BMI impl
maintain connections internally if needed for a particular network c
details are not exposed to the user.

3.2 Memory buffers

The user must specify a memory buffer to use when posting send a
ations. This buffer may be a normal memory region, or it may be a
allocated using BMI memory management functions. If the user e
the memory using the BMI facilities, then BMI has the opportunity
buffer for the type of network being used. This mode of operation
achieving optimal performance. However, normal memory buffers ar
order to better support certain scenarios common to file system op

file system operations act upon existing memory regions (for exampl

Unix read() system call). In these situations, we would like to avoid i

copy, and instead give the BMI layer the flexibility to handle the b

level if possible.

If a memory buffer is allocated using BMI function calls, then it m

located using BMI. These buffers are not guaranteed to be managea

operating system libraries.

3.3 Tags and unexpected messages

The BMI interface allows the user to specify a *tag* for each message.

with a specific tag may only be accepted by a receive operation that s

ing tag. This therefore provides for the user a mechanism to diffe

distinct classes of messages. However, one particular tag is reserved

to have special meaning. This tag marks a message as *unexpected*.

sages are messages that are sent without the receiving host explicit

communication. In other words, the receiving host does not post a

for this type of message. Instead, it must periodically check to see if

messages have arrived in order to receive them successfully. This

of "listening" for new requests in a more traditional networking syst

messages may come from any host on the network. Communicat

hosts is typically initiated by one of the hosts sending an unexpecte

other.

3.4 Client/Server paradigm

The BMI system is better suited for client/server application mod

peer models. This is made evident by the concept of unexpected me

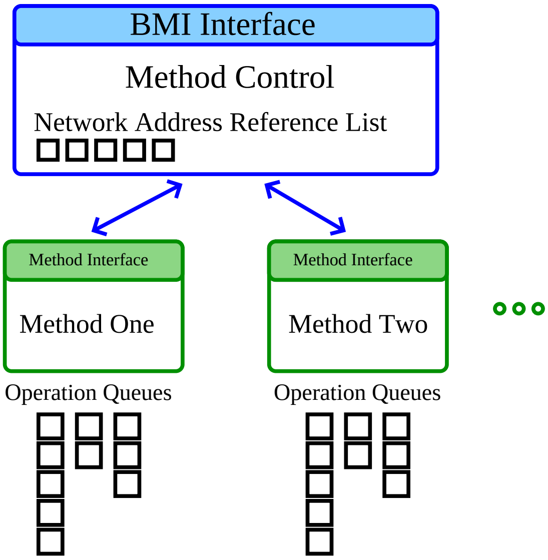


Figure 3.1: BMI Architecture

above. Consider the simple example of communication between two only one of the hosts will look for unexpected messages. This is th other host acts as a “client” by sending unexpected messages to the ser it to perform some service. This service may involve the exchange of f between the two hosts.

3.5 Architecture

The overall architecture of BMI is shown in Figure 3.1. Support for inc protocols is provided by BMI *methods*. There may be any number of at a given time. This collection of methods is managed by the *meth* The method control layer is also responsible for presenting the top lev to the application.

3.6 Method control

From a high level, the method control layer is responsible for orches

operations and managing the network methods. This includes several

including address resolution, method multiplexing, and providing a

interface. It also provides a library of support functions that may be u

implementors.

One of the most important tasks of the method control layer is t

of network methods. When an operation is posted by the user, it is u

control to decide which method will service the operation. Likewise

tests for completion, the method control must test the appropriate

operations of interest.

The method control layer provides the BMI user interface. This is

applications that communicate using BMI. The BMI interface functio

into the appropriate low level method requests that are needed to com

Address resolution is the final major responsibility of the meth

method control manages the BMI level addresses and makes sure

space is consistent to the user, regardless of which methods are in

by maintaining an internal *reference list* for addresses. Each networ

unique reference that provides mappings between BMI user level add

representation of addresses, and the method specific representation o

BMI user level addresses are handles for network hosts that the applic

calling BMI functions. The string representation is the ASCII host n

before they are resolved by BMI (as read from a "hosts" file, for ex

the method address is the representation that that methods use for i

which may contain information specific to that particular protocol. N

addresses are never, under any circumstances, exposed to the applic

reserved for internal BMI use only.

3.7 Methods

Each method is implemented as a dynamically loadable module. Th provide (and strictly adhere to) a predefined *method interface*. It su ordered delivery and flow control for the protocol that it controls. Asi these semantics and adhering to the method interface, there are no o on how the method should be implemented. Support libraries are pro features that are common to many methods, but their use is optiona

Each method is responsible for maintaining the collection of ope working on, usually through operation queues. These collections o private to each method.

3.8 BMI user interface

The BMI interface can be separated into four small categories of fu initiation, message testing, memory management, and utilities.

The message initiation functions are used by an application to req or receiving of network buffers:

- **BMI_post_send(id, destination, buffer, size, buffer_flag**

  Posts a send operation from the specified buffer. The id is w function and serves as a unique handle for the operation to be us for completion. The buffer_flags are used to indicate whethe allocated by the application or by BMI.

- **BMI_post_recv(id, source, buffer, size, buffer_flags,**

  Posts a receive operation. The argument semantics are the sa in BMI_post_send().

- **BMI_unpost(id)**: Forcefully aborts a previously posted opera of the target operation should still be retrieved through the us function(), however. It may have completed successfully before t was processed.

- **BMI_addr_lookup(new_addr, id_string)**: Performs a loo tual representation of the host address specified by id_string BMI specific address handle is filled into the new_addr param used for subsequent network initiation functions.

The message testing functions are used to check for completion o ations:

- **BMI_test(id, outcount, state)**: Tests for completion of a operation, as specified by the id argument. Outcount indicate erations completed (which will either be zero or one in this c parameter is filled in with the state of the operation in questi pletes.

- **BMI_testsome(incount, id_array, outcount, index_arra** Tests for completion of any of a specified set of operations. The to look for is specified by an array of id's of size incount. Outco indicate how many of the target operations completed, while i state_array indicate exactly which operations completed and state was. BMI_testsome() ignores any id's within the id_arra set to the null value of zero.

- **BMI_testglobal(incount, id_array, outcount, state_ar** completion of *any* operations that are currently in progress. fies how many operations the caller is willing to accept with o

BMI_testglobal. Id_array, outcount, and state_array are fille which operations completed and what their final state was.

- **BMI_testunexpected(incount, outcount, info_array)**: T tion of any newly arrived unexpected messages. The incoun many operations the caller is willing to accept, while the ou how many actually completed. The info_array is filled in wi of each completed operation, including the source address, buf size. These parameters are not known in advance by the c *unexpected* nomenclature).

The BMI memory management functions are used to control mem are optimized for use with BMI:

- **BMI_memalloc(address, size, send_recv_flag)**: Allocat memory buffer of the requested size. The address parameter i mote host that will participate in the transmission of the buffer flag indicates whether the buffer will be sent or received from t
- **BMI_memfree(address, buffer, send_recv)**: Frees a mem was allocated with BMI_memalloc(). The address and send_ possess the same semantics as those used in BMI_memalloc().

The final collection of functions perform various utility tasks that involved in network I/O:

- **BMI_initialize(module_string, listen_addr, flags)**: Init system. This function must be called before any other BMI int The module string is a comma separated list of dynamic met use. The listen addr is a comma separated list of parameters t methods use for receiving messages (if needed).

- **BMI_finalize()**: Shuts down the BMI library. This function called once all network communication is completed. It will forc any outstanding operations.

- **BMI_set_info(address, option, parameter)**: Sets option ters. If the address is specified, the function will on affect the responsible for that address. Otherwise, the function has a g the BMI methods.

- **BMI_get_info(address, option, parameter)**: Queries B parameters.

3.9 Immediate completion

The default model for each network operation is to first post it a completion. However, there are often instances in which operatio immediately (during the post procedure) and thus do not require the Examples of this occur when TCP sockets buffers are large enough to to be sent in one step without blocking. This may also occur on th communications if the required data has already been buffered by when the receive operation is posted.

In these situations, it would be good to avoid the overhead of n the test function. We therefore allow *immediate completion* from an Immediate completion is indicated from post functions by a retur BMI library users should always check this return value so that th opportunities to skip the test phase of communication.

3.10 Method interface

The method interface is very similar to the BMI user interface. It imp
the same functions. However, it includes minor variations that take i
fact that operations at this level are targeted for a single specific me

The following listing describes the BMI method interface. Note t
arguments in this interface (*source*, *destination*, and *new_addr*) ar
method address structure type. Each method address contains bina
mation that can only be understood by the specific method that cre

- **BMI_method_post_send(id, destination, buffer, size, bu
sage_tag)**: Posts a send operation from the specified buffer. T
in by the function and serves as a unique handle for the opera
when testing for completion. The buffer_flags are used to indic
buffer was allocated by the application or by BMI.

- **BMI_method_post_recv(id, source, buffer, size, buff
sage_tag)**: Posts a receive operation. The argument semanti
as those used in BMI_method_post_send().

- **BMI_method_unpost(id)**: Forcefully aborts a previously p
The status of the target operation should still be retrieved thr
a BMI_method_test() function, however. It may have comple
before the BMI_method_unpost() was processed.

- **BMI_method_addr_lookup(id_string)**: Performs a looku
representation of the host address specified by id_string. The
the resulting method address structure as generated by the met
the address.

- **BMI_method_test(id, outcount, state)**: Tests for comple network operation, as specified by the id argument. Outcoun many operations completed (which will either be zero or one in state parameter is filled in with the state of the operation in completes.

- **BMI_method_testsome(incount, id_array, outcount, in** Tests for completion of any of a specified set of operations. The to look for is specified by an array of id's of size incount. Outco indicate how many of the target operations completed, while i state_array indicate exactly which operations completed and state was.

- **BMI_method_testglobal(incount, id_array, outcount,** Tests for completion of *any* operations that are currently in a single method. Incount specifies how many operations the to accept with one invocation of BMI_method_testglobal. Id_ and state_array are filled in to indicate which operations com their final state was.

- **BMI_method_testunexpected(incount, outcount, meth** Tests for completion of any newly arrived unexpected messag indicates how many operations the caller is willing to accept, wh indicates how many actually completed. The method_unexpect filled in with a description of each completed operation. Note th is different from the info_array argument to the top level BMI_t function. This is because it contains information that is privat brary and therefore should not be visible to a BMI user.

- **BMI_method_memalloc(size, send_recv_flag)**: Allocat memory buffer of the requested size. The send/recv flag indica buffer will be sent or received from the local host. No address a because the top level interface has already used that data to method to use. It is not needed at this level.

- **BMI_method_memfree(buffer, send_recv)**: Frees a me was allocated with BMI_memalloc(). The send_recv paramet semantics as in BMI_method_memalloc().

- **BMI_method_initialize(listen_addr, method_id, flags)** method. This function must be called before any operations are method. The listen_addr is a method address that contains in how the method should listen for new messages. The method_ used to inform the method of the id handle that will be used method and it's address structures. This is assigned by the met to prevent collisions between method identifiers.

- **BMI_method_finalize()**: Shuts down the method. This only be called once all network communication is completed. terminate any outstanding operations. Each individual meth down independently without disrupting any other operations.

- **BMI_method_set_info(address, option, parameter)**: Se specific parameters.

- **BMI_method_get_info(address, option, parameter)**: Q for optional parameters.

3.11 Support libraries

The BMI library provides several support functions which may aid m
mers when implementing support for new protocols. Each method c
functions to be visible to it once it has been dynamically loaded i
These functions are intended to be as generic as possible so that th
by a variety of different methods.

3.11.1 Operation queues

Every prototype method implemented so far makes use of FIFO queu
of pending operations. Operations are described by generic operation
include common parameters (such as buffer size and location). Thi
includes abstract storage space for private method specific paramete
control or device management information). The operation queue me
is based heavily on the doubly linked list implementation found in
series Linux kernels. This implementation is used throughout the
such as CPU scheduling and the TCP/IP stack which require data
optimized for speed.

- **op_queue_new()**: Creates a new operation queue.
- **op_queue_cleanup(old_op_queue)**: Destroys an existing
  as well as any operations contained within it.
- **op_queue_add(target_op_queue, method_op)**: Adds a
  tion onto the tail of a queue.
- **op_queue_remove(method_op)**: Removes a specific ope
  queue in which it resides.

- **op_queue_search(target_op_queue, key)**: Searches for an matches the characteristics specified by the key. All searches be of the target operation queue.

- **op_queue_empty(target_op_queue)**: Determines whether or not.

- **op_queue_count(target_op_queue)**: Counts the number o an operation queue. This function requires iteration through of the queue. It is therefore only suitable for debugging pu performance is not critical.

- **op_queue_dump(target_op_queue)**: Prints out informat operation in the queue. Only used for debugging and prototyp

Two related functions are also provided for managing the creati structures:

- **alloc_method_op(payload_size)**: Allocates a new operatio cluding enough room for the private data payload that a pa may wish to store within it. Note that this private data is prov contiguous to the generic structure for efficiency.

- **dealloc_method_op(target_op)**: Deallocates an existing me

3.11.2 Method address support

Method address structures are used by methods to identify netwo operation structures, they contain private storage for internal meth functions are provided to aid in managing these structures:

- **alloc_method_addr(method_id, payload_size)**: Creates structure. The method_id field is used to tag the structure as method that created it. The payload_size indicates how much set aside within the structure for private use by the method.

- **dealloc_method_addr(old_method_addr)**: Destroys an address structure.

- **bmi_method_addr_reg_callback(target_method_addr)** by a method to inform the method control layer that it shoul method address structure. The function is typically invoked pected message arrives and the method must autonomously c dress structure to represent the source host. The new method a must be registered with the method control layer so that it is a structure for bookkeeping purposes.

3.11.3 Logging and debugging

The BMI library includes a set of functions known as the *gossip* lib be used for reporting errors, logging messages, or providing debugg The gossip library was created to provide a consistent interface for p tasks. It also can be implemented with various backends that can runtime to control where the log messages are actually recorded. A it supports stderr, syslog, and file based logging. In the future it wi mechanisms such as in core ring buffers.

Gossip also supports setting debugging masks, which can control messages are actually recorded. This is useful in BMI for selecting actually display debugging output.

- **gossip_set_debug_mask(debug_on, mask)**: Controls wh messages are on or off. The mask parameter specifies what cl messages will be displayed if debugging is turned on. Note th messages *cannot* be disabled.

- **gossip_enable_syslog(priority)**: Enables the syslog logging the syslog priority that will be assigned to each message.

- **gossip_enable_stderr()**: Enables the printing of error messa

- **gossip_enable_file(filename, mode)**: Enables the logging o to a specific file. The mode parameter is useful for specifying be truncated or appended.

- **gossip_disable()**: Turns off the gossip library.

- **gossip_debug(level, format, ...)**: Logs a printf() style m specified debugging level.

- **gossip_err(format, ...)**: Logs a printf() style error message will be recorded regardless of the current debugging mask.

- **gossip_ldebug(level, format, ...)**: Same as the gossip_debu cept that it also displays the file name and line number from wh originated on systems with preprocessors that support this fea

- **gossip_lerr(format, ...)**: Same as the gossip_err() function also displays the file name and line number from which the me on systems with preprocessors that support this feature.

3.11.4 Operation id's

Each method is responsible for creating opaque id's that can be
operations that are currently in progress. Typically these id's will
user requests to specific operation structures. The *id_generator* lib
to aid methods in performing this mapping operation. This function
within a discrete interface to allow for multiple implementations whi
tables or other data structures to store mapping information. It also
id space is consistent across all methods.

- **id_gen_fast_register(new_id, void* item)**: Registers a ne
  the interface and creates a new id that may be used to referen
- **id_gen_fast_lookup(id)**: Returns a pointer to the origina
  that was associated with the given id.

All of the interfaces listed in the preceding section come toget
method support libraries that method implementors should take ad
creating new methods.

3.12 Method control implementation

The method control layer (as introduced in section 3.6) is responsibl
conversions between the BMI user interface and the method interfac
method multiplexing that this implies. This is a relatively thin layer
it plays a critical role in providing core BMI features. Some of the
tasks that it performs are outlined in the following sections.

3.12.1 Method initialization

The BMI library must perform several steps when it is first initiali
point at which it must enable all of the active modules and initial
bookkeeping information. The steps that it performs at this time
outlined as follows:

1. Parse the list of modules that will be used by the library.

2. Create a table to track each method and its instantiation of th
face.

3. Load each method module one at a time and verify that it pro
symbols required by the method interface.

4. Create a reference list to use for mapping host addresses betwe
faces.

5. Initialize each method individually.

If any of these steps fails, then the initializer cleans up any work
to that point and exits.

3.12.2 Posting and testing single operations

Posting and testing individual operations is relatively simple becau
teracting with only one method at a time. These functions are carrie

1. Verify any arguments passed in by the user so that the metho
assume that they are safe.

2. Search through the reference list for an entry that matches the
by the caller.

3. If such an entry is not found, return an error because the user s
is invalid.

4. Find the method interface that matches the address (as indic
erence structure) and call the appropriate method to carry
operation.

3.12.3 Aggregate operation tests

Testing operations from multiple methods within one library call is
plicated version of the scenario outlined in section 3.12.2. This sit
handled by the BMI_testsome() and BMI_testglobal() function calls
ability to interact with multiple operations at once.

In the BMI_testsome() case, the array of operations that the
segregated based on which methods control which operations. Each
tested in a round robin manner to determine if any of those operations
The results of the tests are aggregated back into a single array and
user.

In the BMI_testglobal() function is similar except that the set of
known in advance; therefore the initial sorting phase can be skipped

The order in which round robin testing occurs is determined by th
the user previously specified the list of method modules at initializa
high throughput interfaces can be given a somewhat higher priority
them first in the set of available method modules.

3.12.4 Address resolution

Address resolution is the final critical component of the method co following steps are performed in order to lookup an address based o tion:

1. Search the reference list to determine if this lookup has already If so, immediately return the correct BMI address.
2. Contact each active method until one of them indicates tha successfully parse the host name.
3. Create a new reference structure to track the address.
4. Fill in the reference structure with information about which m sible for it and which address handles may be used to refer to
5. Store the reference structure and return a valid BMI address t

Note that since BMI is a connectionless interface, there is no w that an address is truly valid until a user attempts to communicate

3.13 Bringing together the BMI library

The preceding sections outline all of the components that make agnostic portion of the BMI architecture. All of these compone constructed to limit each interface to only those components that n A user has no mechanism for directly interacting with method da functionality, nor does any method have the ability to interfere wi of other methods. Modularity is preserved by only providing the ab of each interface to a given component. This helps to provide a s for both the BMI user and the method implementer. It also insure portions of the BMI architecture may be modified or optimized wit with core functionality. This is important in supporting future resea

The method control layer, support libraries, and user interface framework for the use of various network protocols, however. The real out communications and emulating necessary network features is ca the BMI methods. The design of a set of prototype methods is following chapters.

Chapter 4

BMI method case studies

For BMI to be successful, we must be able to efficiently impleme
highly diverse networking environments. Thus we have chosen two dis
systems as examples of the potential of the Buffered Message Interf
TCP/IP. TCP/IP is a well known and widely adopted network p
stream based and connection oriented, and provides full flow contro
The second is GM. GM is a user level protocol that is supported on My
series of network adapters. GM is connectionless and provides no flo

These two protocols were chosen for two reasons. The first is thei
ity. If BMI can be shown to work efficiently over both protocols, the
at least some indication that the design is flexible. The implementat
will outline the challenges that were overcome for each method in m
ondly, both protocols are mature and readily available on productio
Their behavior is well known and does not introduce any unexpected
studying network implementations.

Note that both implementations are essentially reference designs.
tions are available then have been implemented. Some of these have

for future work and will be noted where appropriate. The primary e
on obtaining dependable behavior and evaluating the Buffered Mess

4.1 Method support libraries

Several support functions are provided for use by the BMI methods
functions were previously outlined beginning in section 3.11. The mo
of functions provide queuing capability. Almost every network me
queues to keep track of network operations. Other support functions
manage two important data structures: the method addresses and me
Method addresses are used as handles for hosts on the network. Ope
are used to represent individual operations that are posted by the a
structures contain generic fields that apply to all methods while als
for each method to store its own private information.

4.2 TCP/IP

4.2.1 Challenges

- **Connectionless emulation**: The TCP/IP socket interface is
  ented. This means that a connection must be established be
  before communication can occur between them. It should als
  after all communication is complete. BMI, however, is a con
  face. The act of setting up and tearing down connections m
  below the application interface, transparent to the application.

- **Data streams**: TCP/IP sockets operate as data streams. T
  no explicit concept of message boundaries. Any amount of d
  or received in a single operation, but it may not necessarily c

Partial sends and receives are legal when using nonblocking T This behavior does not match the BMI model of communicati each message as an atomic unit that is either sent correctly or

- **Error recovery**: Recovery from communication failure using T what complicated within the BMI environment. TCP/IP socket a bit of state because of the presence of a connection and a d single operation fails, it is important to properly tear down a and safely handle any further pending operations that intended socket.

4.2.2 Approach

Building blocks

The TCP/IP method implementation is based on three primary buil first is the sockio library, which provides a very small abstraction of interface. It implements the basic operations, such as creating soc sockets, setting TCP options, and performing read and write operati library has been used extensively in the current PVFS implementati to perform reliably.

The second building block is the socket collection library. The library provides a mechanism for managing groups of active socke added to it as they are created, and removed from it as they are socket collection tracks all of the sockets and determines which ones send or receive data. When the TCP/IP method is prompted to do collection is tested to determine which sockets are ready to handle internal polling set is dynamically resized as necessary to accommo large collections of sockets. This interface also allows the possibilit

ing new polling strategies without compromising code that depend
collections.

The final building block for the TCP/IP method is the operation
provided by the method control layer. The queues are necessary
ordering of messages and to keep track of operations that are curre
There are a fixed number of queues available that are used for com
all hosts. This approach is chosen over maintaining separate queues
reasons of simplicity and maintainability. We intend to show that t
method queues is not detrimental to performance as long as the que
and searching is performed in an efficient manner.

Message modes

The TCP/IP method offers three modes of operation depending on
of the message to be sent. The first, unexpected mode, is specif
when sending unexpected messages as outlined in section 3.3. The
and rendezvous mode, are transparent to the user and are internall
TCP/IP method based on the size of the data region to be transfere

The semantics of each mode of operation are defined as follows:

- **rendezvous**: This is the simplest messaging mode supported
  method. It is chosen for larger messages (the default thresho
  rendezvous mode will be used for any message over 16 Kbyte
  situation, the sender will first send a header describing the m
  immediately follow it with message data. The receiving metho
  header at any time. However, it will not begin to read in th
  until the receiving user has posted a matching receive operati
  this mechanism is termed "rendezvous" mode; the bulk data

allowed to occur until both the sending and receiving user ha
the operation.

- **eager**: This messaging mode is chosen by the TCP/IP met
message sizes (by default, less than 16 Kbyte in size). The se
rendezvous case) first transmits a header describing the messa
lows it with the actual message data. The receiver will accept
then make a decision about how the message data should be
matching receive operation has already been posted by the use
sage data will be read into the receive buffer for that operatio
receive has not yet been posted, then the receiving method
allocate a temporary buffer for the data to be stored in. This
ation to make progress even if the receiving user is not yet rea
Once the matching receive operation is posted, the data can be
final buffer and the temporary buffer is destroyed.

- **unexpected**: Unexpected messages are handled in an almost i
to eager messages. The only variation is that there is no final bu
the sender transmits data before the receiver is ready. Instead
buffer is passed to the user when the user checks to see if
messages have been received. The semantics of unexpected m
the the receiver does not get a chance to specify the destinatio
message. It is created by the method.

It is interesting to note that in all three cases the sending meth
same. This seems counter-intuitive at first because it would appea
in the rendezvous case) that the sender should wait until the rece
before transmitting the actual message data. However, we can rely
behavior of the TCP/IP sockets interface in this context. TCP/IP

sender to start sending data before a receiver is ready. The data is si the operating system level until it can be transferred. This buffering overall latency and throughput because the operating system does on user intervention to begin moving data once the opportunity aris

Queuing model

As mentioned earlier, the TCP/IP method uses queues to keep t operations that are either in progress, awaiting resources, or complet queues are used:

- **Send queue**: Contains all send operations (for any mode cannot be initiated yet. Operations are typically queued here be operation to the same address has not yet completed. This queu message ordering when multiple send operations are posted.

- **Completed queue**: Contains all operations that have complet whether successful or not. Operations are removed from this user has queried with the appropriate BMI_test function.

- **In-flight receive queue**: Contains a list of receive operations begun but are not yet completed. It provides a fast mechanis where incoming data should be placed once a socket has data

- **Eager receive queue**: Contains receive operations that are be accepted in eager mode. These operations have not yet re Once data begins to arrive for a particular operation, it will in-flight receive queue.

- **Rendezvous receive queue**: Contains receive operations that using rendezvous mode. These operations have not yet received

data begins to arrive for a particular operation, it will be move
receive queue.

- **Buffering receive queue**: Contains eager or unexpected re
which have begun buffering data before the user posted a match
queue is searched when an eager message is posted to determine
has already been completed.

Notice that there are more receive queues available than send qu
two reasons for this design. First of all, as noted earlier, all send
treated the same from the method's point of view. There is no dist
TCP/IP sends that occur for different message modes. Secondly,
receive operations requires more queue searching than in the send ca
gain a measurable boost in performance by simply splitting up the que
search time.

Scenarios

The internal workings of the TCP/IP method can best be summa
simple examples. The first example is an eager mode send operati
Figure 4.1. The operation begins with the user calling BMI_post_sen
will first check to see if there are any sends already scheduled for th
If it finds one, the message is immediately queued to preserve orde
the method makes an attempt to send the message envelope and as
can (without blocking) before queuing. If the message is completed a
never queued, and the return value of BMI_post_send() indicates its

If the send does not complete on the first try, it will remain in
until the internal socket collection indicates that work may continue o
Since the TCP/IP method does not possess its own thread of cont

Figure 4.1: TCP method (typical send scenario)

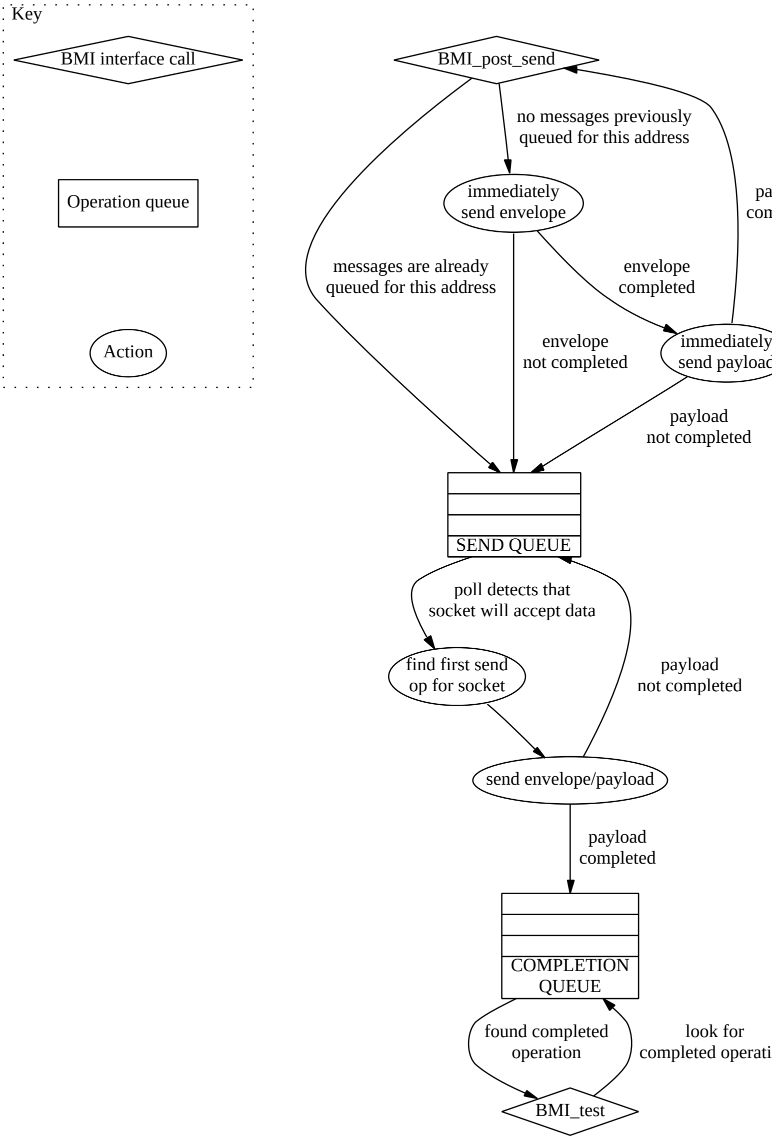
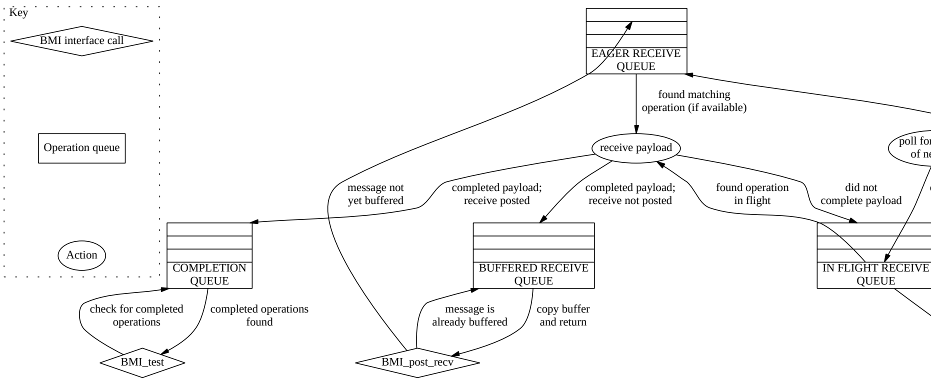


Figure 4.2: TCP method (typical receive scenario)



work until the user calls BMI_test(). Once an operation is finishe
the completion queue, where it will be recovered by the user in sub
BMI_test().

The common receive scenario is slightly more complicated than th
ure 4.2 outlines the flow of events for an eager mode receive. When B
is called by the user, it first checks to see if the receive has already
If so, it copies the payload out of the temporary buffer and retur
Otherwise, the receive operation is queued.

When the socket collection indicates that data is available, the me
the in flight receive queue to see if the data belongs to an operation
in progress. If the method finds a match, it continues receiving d
no matching operation in flight, then the method reads the envelop
socket to determine the parameters of the message.

If an operation completes before the matching receive has been pos
it will be stored in the buffered receive queue. Otherwise, the opera
the completion queue where it remains until the user tests to see if it

4.2.3 Possible optimizations

The approach outlined thus far for implementing a TCP/IP BMI met to provide the correct messaging semantics. However, there are sever which may be implemented to improve performance. Some of these presented below. A few of them will be explored more fully in sectio

- **Nagle's algorithm**: Nagle's algorithm [20] attempts to impr distributed networks by limiting the number of small packets given time. If TCP/IP messages have been sent for which no has been received, then Nagle's algorithm prevents packets bel threshold from being transmitted until all acknowledgments hav This may cause excessive delay for small messages if the netwo does not need such a conservative approach to small messages. modern TCP/IP implementations have a mechanism for disabli from the application level.

- **Eager receive opportunities**: Immediate completion of any tion is beneficial to performance because it avoids the overhe completion later. The TCP/IP method does not yet take ad such opportunity in the receive case. One example occurs if a when no other receive is queued for that address. In this situat the optimistic approach and immediately check the matching If the sender has already begun transmitting, then the recei make progress before it is even queued.

- **Tuning socket buffers**: The amount of data that can be b send or receive is determined by the operating system's TCP/ size. This parameter may be specified globally for the entire s be specified on a per socket basis. If it is set on a per socket method has the ability to adjust the buffer size dynamically the type of operations being handled. It may be lowered to resources, or increased in order to allow the kernel to do more call.

- **Reducing memory allocation**: Dynamic allocation of mem an expensive operation. At this time, the TCP/IP method internal bookkeeping structures as needed, rather than reusing structures. This may be hindering small message latency in In particular, the method operation structure (which tracks network messages) is allocated for almost every message that This could be alleviated by simply replacing the method ope with an implementation that maintains a pool of structures initialized and ready for use. Similar techniques are used in system kernels to limit the overhead of acquiring common data

- **Queue separation**: The range of available operation queues compromise between method complexity and queue search tim may not be optimal for all work loads. The queues could be provide more information about search times. This may indica that could be targeted for improvement in queue layout.

- **Alternative polling algorithms**: The current implementat same order through the socket collection on each iteration. Th be replaced with a round robin or other simple polling mech evenly distributes work among active sockets. We may also be situations in which more than one message can be placed in a iteration. Right now the test functions try to send or receive o per socket for each function call.

4.3 GM

4.3.1 Challenges

- **Memory management**: GM requires all data buffers to be p
ical memory before transmission. We can take advantage of thi
memory management functions. However, we must also suppor
trary user buffers at the BMI interface level. This requires eithe
copies or memory registration to maintain interface semantics.

- **Flow control**: GM does not provide any form of message level
fact, it requires that all receive buffers be posted before messa
begins. Therefore, the BMI method must implement flow co
relax the message ordering semantics. BMI does not guarantee
between sender and receiver.

- **Matching posted buffers**: There is no way to specify which in
will match a given receive buffer posted by the GM user. The use
that a buffer is only to be used for messages from a particular h
Therefore, once a message is received, the method must analyz
its origin and perform a memory copy if necessary to put the da
user specified location.

- **Limited token resources**: The GM interface forces the use
posting sends or receives unless it possesses an appropriate t
a finite number of tokens. They are consumed when a send
receive buffer is posted. The tokens are then returned when a
completes.

4.3.2 Approach

The BMI GM method implementation relies heavily on the queuing vided by BMI to preserve message ordering and maintain the state It also takes advantage of several features specific to the GM librar assist in managing message completion.

Message modes

The GM method supports two messaging modes. The decision to other is based solely on the message size. The most complex messag *eager handshake mode*. It is used by default for any message larger th this parameter is tunable. This mode is considered eager because it to be buffered at the receiving side before the user posts a matchin tion. However, the sending and receiving hosts must negotiate at a transmitting the message.

Smaller messages are sent using *immediate mode*. Immediate m handshaking at all. The sender assumes that the receiver is always pre messages of this size.

Methods are capable of sending and receiving control messages actual message data. These control messages may contain flow cont handshaking information, or actual message data in the case of i messages. When the GM method is initialized, its first task is to number of receive buffers for accepting control messages. These bu after processing control messages and are replaced as quickly as po that there are always buffers available to handle new control messag

When a sender initiates an eager handshake communication, it firs message to the receiver to announce that it wishes to transmit. Th prepares a buffer of the appropriate size for receiving the message.

is ready, it sends a control message back to the receiver in response t

the buffer is ready. The actual message payload is then transfered.

When a sender initiates an immediate communication, it simply

as payload on a control message. The receiver will accept these m

negotiating in advance.

Flow control

The GM method implements very basic flow control mechanisms.

munications mechanism on Myrinet networks are extremely reliabl

flow control is not to avoid congestion or lost packets, but rather to c

resources. Only a finite number of receive buffers may be posted at a

one must be careful not to exhaust the memory resources of a host

There are two types of buffers which may be posted by a recei

first is the large data payload buffer used during eager handshake

The use of these buffers can be easily controlled by the receiver sin

used in this specific mode. Once the receiver processes a control me

a buffer of this type, the receiver has the option of waiting as lon

before posting the buffer. If it runs out of memory resources, it simpl

a control response until the resources are available. This prevent

transmitting the payload too quickly.

Control message buffers are the second resource that must be con

method attempts to keep as many of these available as possible, but i

for a client to overrun the available buffers by sending small messages

can be processed. In order to prevent this situation, a limit is place

of send messages that may be in flight between hosts at any given t

is considered to no longer be in flight once the sender is sure that

processed it. The number of messages allowed per host is tunable.

parameter improves performance because it allows deeper pipelining i are allowed to be transmitted back to back. However, this parameter in larger networks to ensure that each host can accept messages from simultaneously without exhausting memory resources.

Message pipelining

Note that *pipelining* is used extensively in eager handshake mode. T steps to carrying out eager handshake messages. One message may while another message is in step two, and so on. The available reso how many messages may carry out the same step at the same time. I exhausted, other steps are allowed to continue up until stalling on tha approach is very similar to instruction pipelining in modern micropr

Retransmission

Even with the above flow control scheme, the scalability of the GM m With enough hosts on the network, any finite number of available consumed in a degenerate case, such as a many to one communica the method must be able to recover from packet loss that occurs wh sent before buffers are ready.

The GM library provides extensions to detect and recover fron However, the documentation for these extensions is incomplete at writing because these features have undergone modifications durin cycles. Implementation of a retransmission policy for the BMI GM me being postponed for future work.

Scenarios

The GM method can be understood more fully by observing a few
immediate mode messages are just a simplified version of the eager
sages, so we will focus on eager handshake mode. Keep in mind tha
not implemented using threads. Therefore, the state machine is only
caller invokes BMI function calls.

A state diagram of the send case is shown in Figure 4.3. When
posts a send buffer, the method first checks to see if the data nee
into a suitable buffer. Once the buffer is ready, it must check three
continuing. First it makes sure that a send token is available. Then i
any messages are queued ahead of it. Finally, it checks to see how ma
already in flight to the target host. If any requirement is not met, t
is queued. Otherwise, it continues by sending a control request to th

If the target host responds and grants permission to send the dat
the method must again either obtain a send token or queue the
token is available. Finally the data payload is sent and the operatio
completion queue to be recovered when the application calls BMI_te

The matching receive example is shown in Figure 4.4. When a c
received from a sending host, the method must obtain a token to
payload. If no token is available, the operation is temporarily queu
then posted to accept the payload, and a control response is sent to i
that it may proceed. The operation is then queued until the data ar

Once the data arrives, the method checks to see if a matching r
has been posted in the control match queue. If so, the operation
moved to the completion queue to be recovered during a BMI_test()
operation is placed in the receive post queue to wait until BMI_post_

Figure 4.3: GM method (typical send scenario)

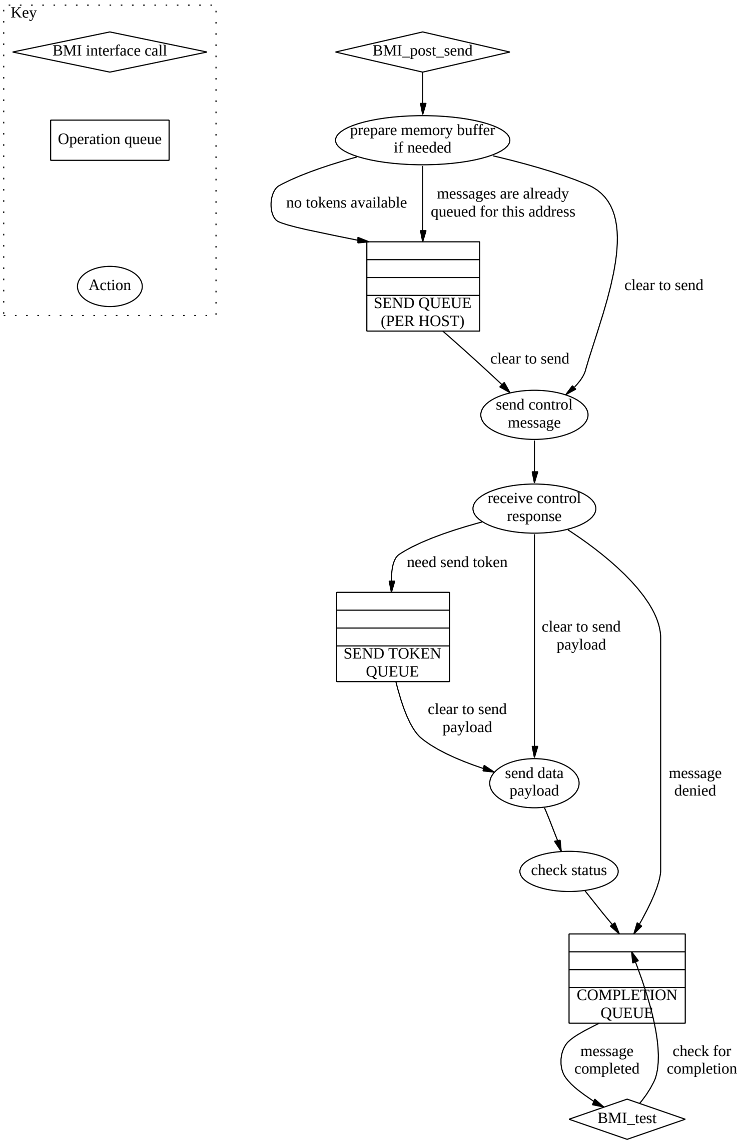
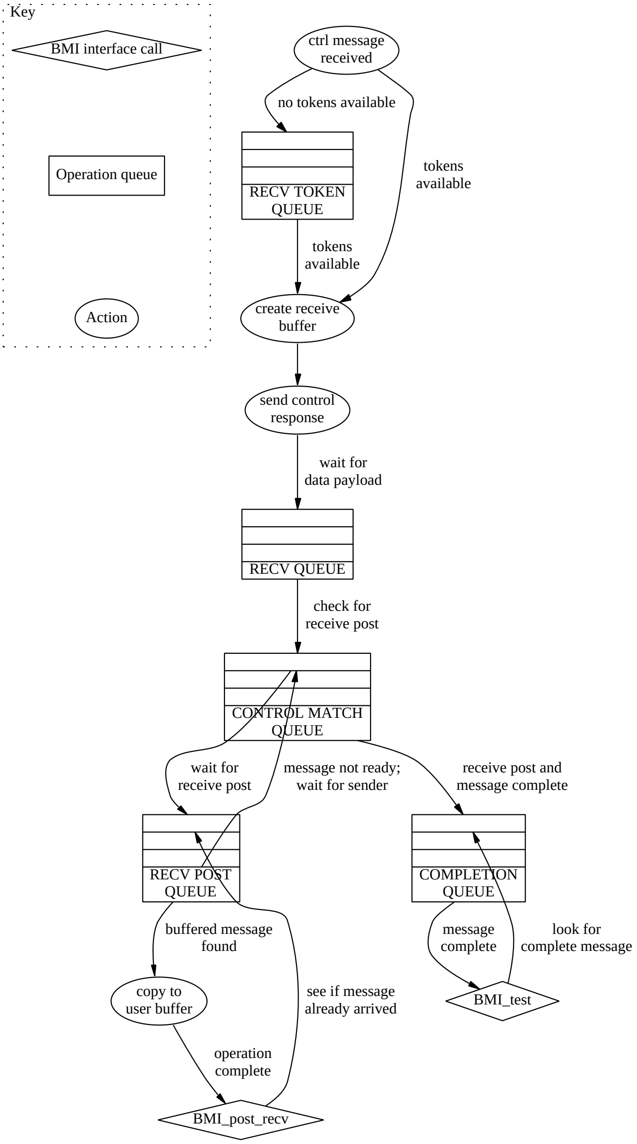


Figure 4.4: GM method (typical receive scenario)



BMI_post_recv() will complete immediately in this case because the
be available before the function is invoked.

4.3.3 Possible optimizations

The GM messaging library provides tools for implementing many
optimizations. The original GM method implementation is focuse
rather than speed, so several of these possibilities have been left for

- **Registering memory**: All memory buffers transmitted using
be allocated by the GM library or registered by the GM lib
if a BMI user passes in a buffer that was not preallocated,
created and the data is copied into it. This is done so that
be provided in the buffer for storing header information. H
probably be much faster to register the existing buffer. Alloca
header could be avoided by transmitting the data using contro

- **Fast GM messages**: GM optimizes for very small messages
them within the low level GM control packet. If the application
this type of message, it can operate directly on the data buff
control packet. If it ignores this type of message, then the data
copied into a normal GM buffer and handled just like any othe
buffer copy could be avoided if the method processed this special
(called "fast" messages by GM).

- **Avoiding memory copy on receive**: Since the GM metho
the caller to specify which incoming messages are placed in whi
ing messages are always copied to ensure that they end up in
buffer. In some cases, this is unnecessary because the caller
care where the data is placed. We could take advantage of th

special type of receive operation that provides the buffer to th operation completes, rather than specifying it in advance. Th a modification to the BMI interface.

- **Advanced flow control**: The BMI GM method implements tary flow control scheme. A more advanced approach could po speedup, but is beyond the scope of this document.
- **Pooling control buffers**: As mentioned earlier, control rec pooled and reused once they are processed to avoid the overhe new memory regions. This is not done in the send case, how is more difficult to locate available buffers when they are ne lookaside list implementation could resolve this issue, however, memory allocation time.
- **Reducing memory allocation for bookkeeping**: The BM cate several structures to track pending operations and netwo these structures were allocated in advanced and reused as need to cut down on message latency.

Chapter 5

Results

We must measure the performance of the BMI implementation in s
to evaluate its efficiency. MPICH will be used for comparison purpos
is a point of reference for observing the results. MPICH was chos
capable of providing almost all of the messaging ability that BM
there is an MPICH device available for both TCP/IP and GM (the tw
implementations).

In all cases, the benchmarks were implemented using MPI func
closely match the capabilities of BMI. However, since MPI is a muc
mentation, there are often MPI functions available that would be more
functions are ignored for this comparison, however, because we are
in general purpose baseline performance.

It must be emphasized that this is not a direct comparison of th
In general they solve very different problems and thus are subject to
constraints. BMI has an advantage in these tests because it is a muc
interface and the test applications do not always take advantage of t
MPI application approach.

The first analysis of BMI performance will focus on point to p round trip latency, and many to one and one to many communica two classes of tests will hopefully point out fundamental strengths a the interfaces, while the latter tests will attempt to evaluate BMI i similar to what would occur in a real life file server implementation.

5.1 Test environment

These tests were all carried out on the Chiba City scalable cluster at A Laboratory [7]. The cluster was configured as follows at the time of There were 256 nodes, each with two 500-MHz Pentium III processor RAM, a 100 Mbits/sec Intel EtherExpress Pro Fast Ethernet networ in full-duplex mode, and a 64-bit Myrinet card (Revision 3). The no Linux 2.4.2. There were two MPI implementations: MPICH 1.2.1 f and MPICH-GM 1.2.0 for Myrinet. None of the tests were perfor than 65 nodes at a time.

5.2 Initial TCP/IP results

All TCP/IP tests were performed using the Ethernet network on MPICH 1.2.1. In addition, baseline bandwidth measurements were the ttcp test utility, version 1.12 [25]. The ttcp utility operates dir sockets with no abstraction layer. It should therefore give a good maximum obtainable TCP/IP bandwidth.

5.2.1 Bandwidth

Bandwidth was measured by transmitting a predetermined amount
two hosts using a variety of message sizes. The data was marked with
timing began so that its correctness could be verified after receipt. A
posted in order before testing for completion. Timing for both sen
hosts includes the time required to post and test for completion of al
tests were performed using the MPI_Isend(), MPI_Irecv() and MPI_
BMI tests were performed using BMI_post_send(), BMI_post_recv()

Memory buffers were not allocated using the BMI interface in
This sort of allocation has no impact on performance in the TCP/IP
TCP/IP has no mechanism for optimizing message buffers.

All figures shown are the result of averaging five measurements
MPI tests were run within seconds of each other in each case in orde
the system was in a consistent state for each test.

Figure 5.1 shows the TCP/IP bandwidth as measured from the
very small message sizes, ranging from 100 bytes to 1000 bytes. The
data transfered in every case was 1,000,000 bytes, or nearly one Mb
that the total number of message sent for each data point ranged from
messages.

The raw TCP performance (as measured by ttcp) shows negligi
choice of message size. However, both MPI and BMI demonstrate the
from an extra layer of abstraction over the sockets interface. BMI d
peak bandwidth capability until the message size is 500 bytes or greate
also show that the BMI interface imposes an overhead of about 5% co
socket communications for message sizes larger than 500 bytes. MP
its peak capacity within this message range.

Figure 5.1: Small message TCP/IP bandwidth (send)

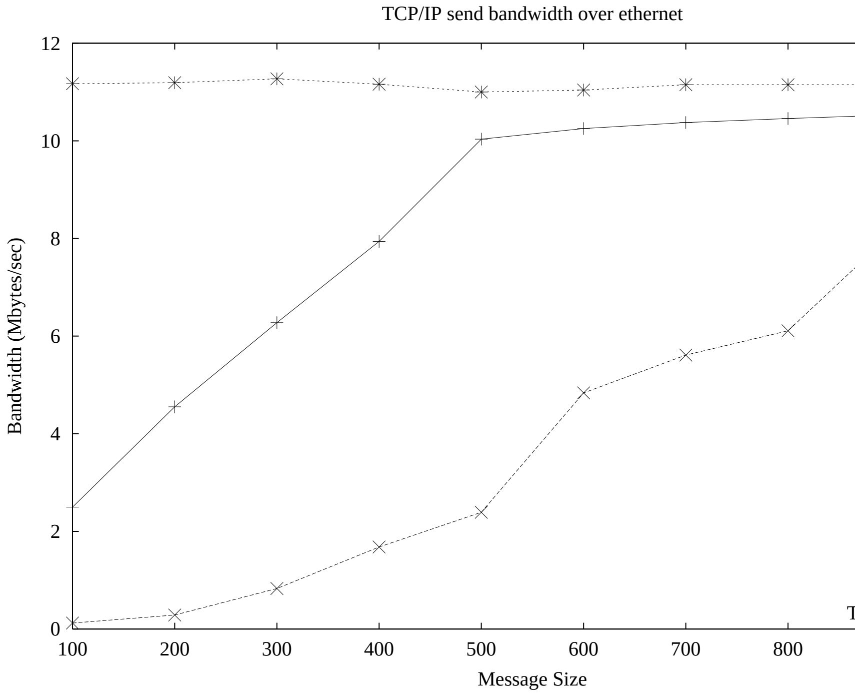


Figure 5.2: Small message TCP/IP bandwidth (receiv

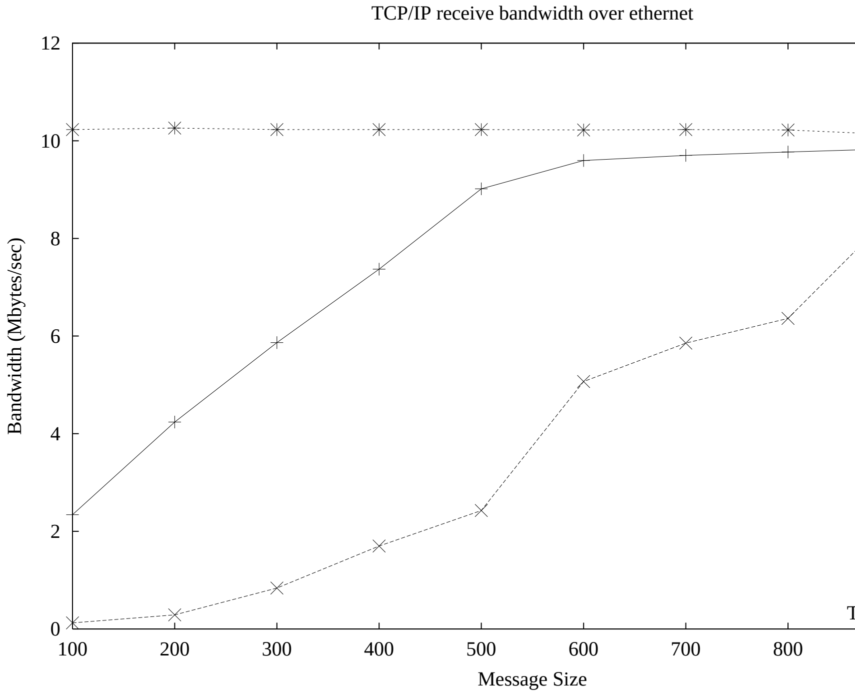


Figure 5.3: Larger message TCP/IP bandwidth (send

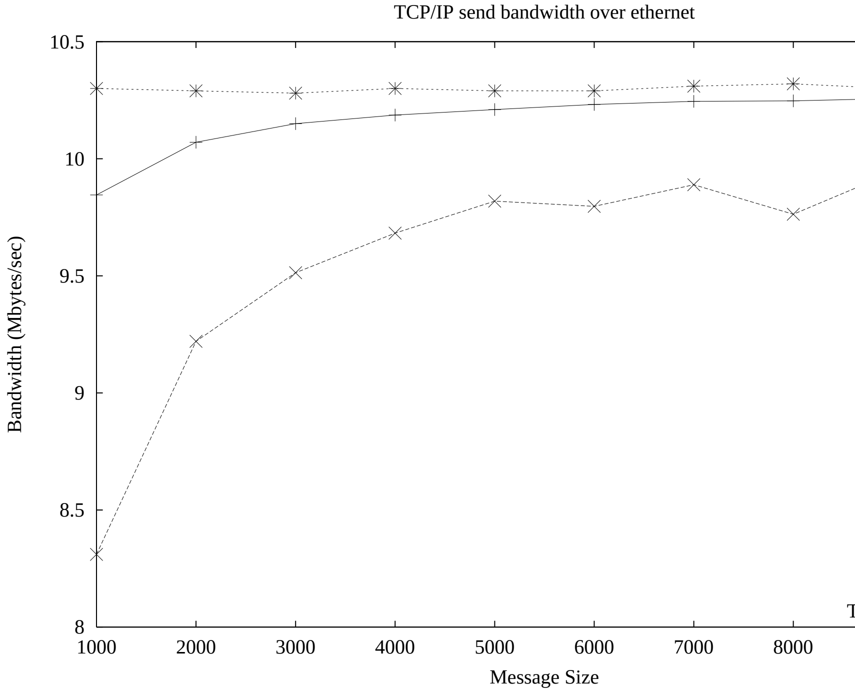
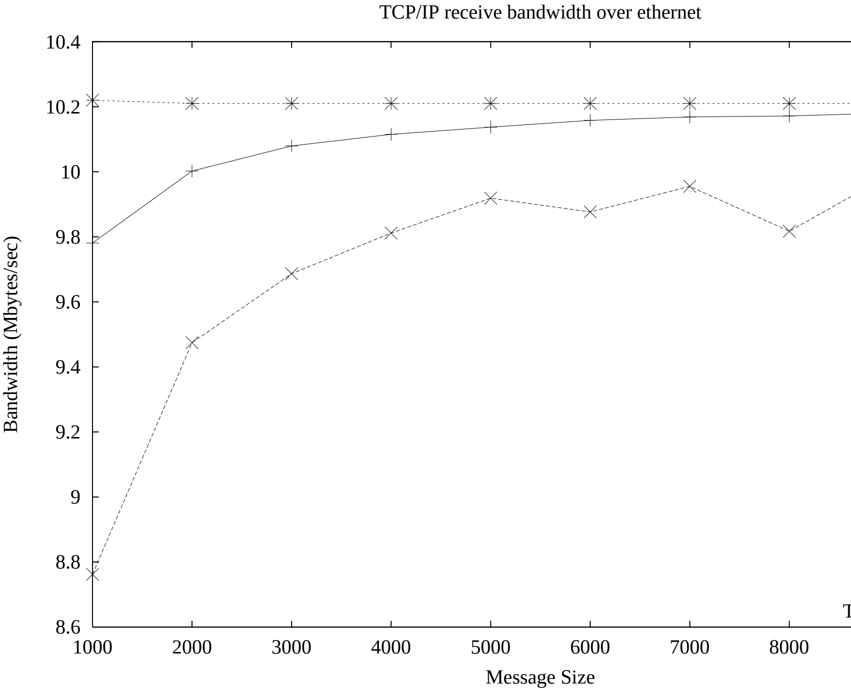


Figure 5.2 shows very similar results for bandwidth measured fro
host.

Figures 5.3 and 5.4 show bandwidth results for the transfer of 1
(nearly 10 Mbytes) of data, using message sizes ranging from 1,000
bytes. This means that the number of message needed to complete th
between 10,000 and 1000, just as in the previous case.

Notice that with 10,000 byte messages, the BMI method only ol
Mbytes/sec for a 10 Mbyte transfer, as opposed to 10.75 Mbytes/sec
transfer shown in the previous graphs. This shows that both message
of messages sent have an impact on performance. Again, the interfa
noticeable overhead when a large number of messages are queued up

Figure 5.4: Larger message TCP/IP bandwidth (recei



As the message size gets larger and the total number of mess

decreases, the BMI performance begins to match the raw TCP/IP p

large messages the overhead imposed by the interface is less than 1%

5.2.2 Latency

Latency results were obtained by measuring the round trip transm

message between two machines. All communications were done us

function calls. Timing started just before posting the first send oper

just after the receipt of the response message completed successfull

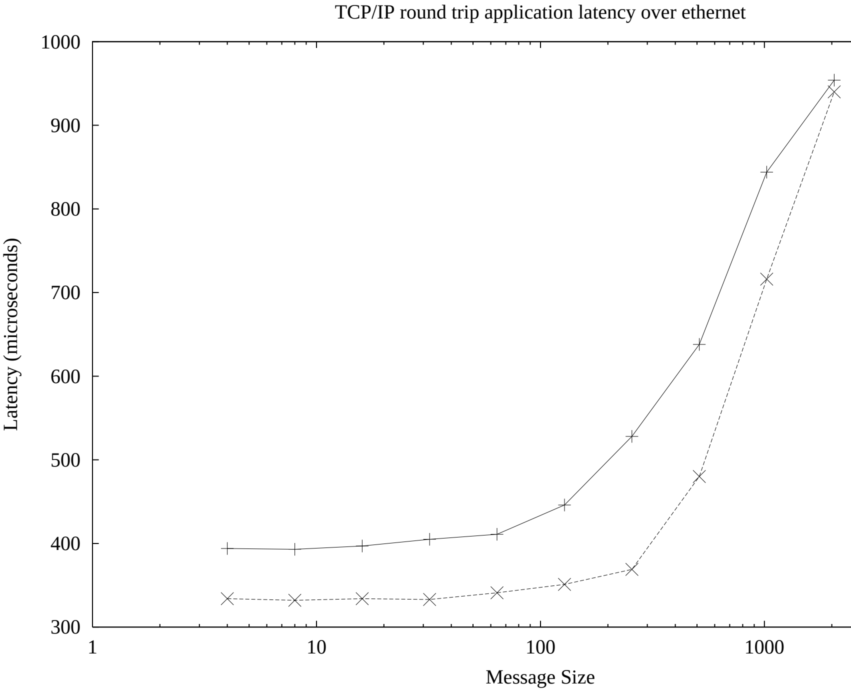
ments were carried out from the application level.

Figure 5.5 shows the round trip latency as measured using both

The message sizes ranged from 4 bytes to 4 Kbytes (on a logarith

MPI latency is much lower than the BMI latency in this test, thoug

Figure 5.5: TCP/IP round trip latency



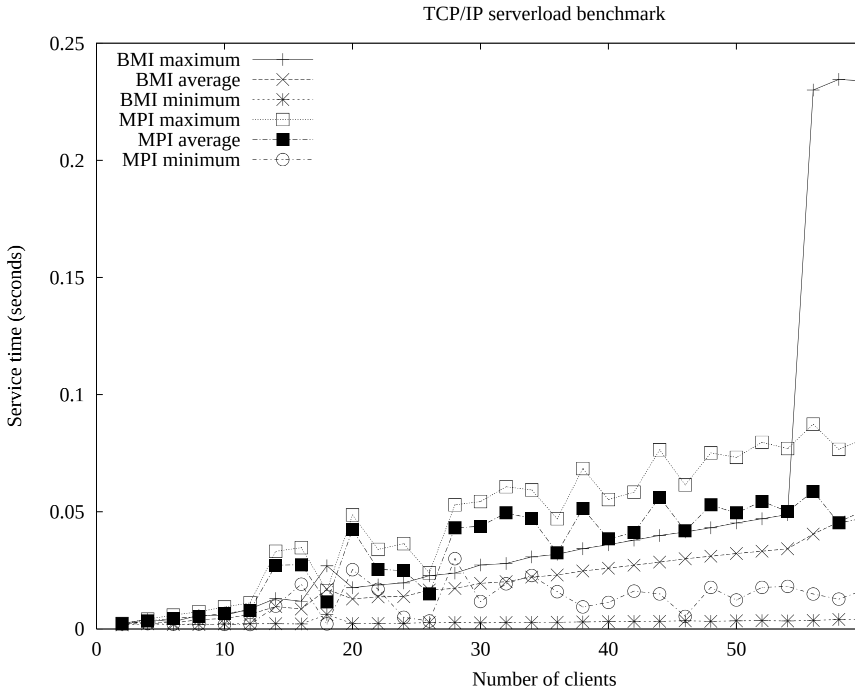
converge for larger message sizes. Section 5.4.1 explores techniques
latency found in the BMI TCP/IP method.

5.2.3 Simulated server load

This test is structured as follows. A single host is setup to listen f
messages. When it receives a message (formatted as a small reques
sending another message of the requested size back to the sender.
communicate with it are synchronized using MPI so that they all att
the server simultaneously. This is intended to measure BMI perfo
load that resembles what would happen if many clients were to
server simultaneously for a small data read operation.

In addition, the server is given no advance knowledge of which cli
it, nor in what order they will communicate. The BMI portion of

Figure 5.6: TCP/IP many to one performance (10K mess



with this by using the BMI unexpected message facility. The MPI

server handles it by posted a collection of receives that will match a

sender.

Timing information (round trip application latency) is measured

side. The average, maximum, and minimum times among all clien

across five test runs. Figure 5.6 shows the results of this test using 10

for anywhere from 2 to 64 clients.

The BMI method performs well for this message size and exhibi

performance curve.

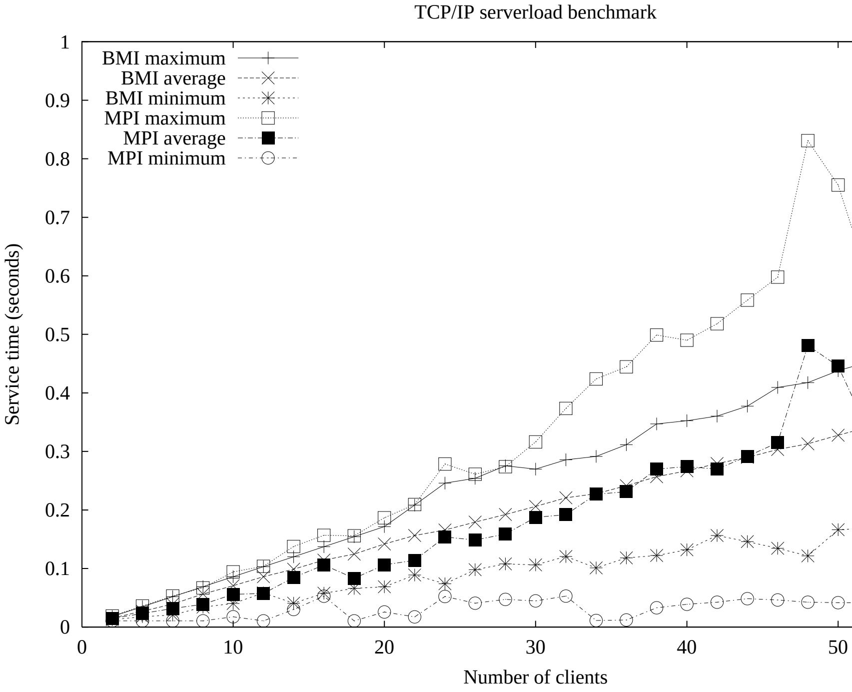
Figure 5.7 shows results from the same test scenario, but with a

100 Kbytes rather than 10 Kbytes. The interesting behavior from this

the BMI and MPI tests exhibit very similar average performance acr

clients. However, the MPI interface obtains this average through a m

Figure 5.7: TCP/IP many to one performance (100K mes



of individual client measurements. MPI obtains much lower latency

while also obtaining much higher latency on other clients when comp

5.3 Initial GM results

All GM tests were performed using the Myrinet network on Chiba Ci
GM 1.2.0. All tests were setup in an identical manner to those pres
5.2, but using the GM method for BMI and Myricom's MPICH-GM
for MPI.

5.3.1 Bandwidth

The bandwidth was measured again by transmitting a fixed amount
variety of message sizes. The BMI GM method has the ability to ta

optimized buffers, so the performance was also measured with the bu

cated in advance using BMI_memalloc(). In test cases with this BM

the memory allocation time was not included in the timing. This is

the normal test cases in which the time needed to malloc() the data

included in communication timing. All data points shown are the res

five test runs.

The original intent was to measure performance over the same r

sizes used in the TCP/IP tests. However, it was discovered that t

implementation was incapable of completing the 1 Mbyte bandwidt

Byte message sizes. The test generally failed with memory allocatio

As a result, performance measurements had to be taken from a

size range to provide data points from both interfaces.

Figure 5.8 shows bandwidth as measured for the transfer of a 1,00

using message sizes ranging from 1,000 bytes to 10,000 bytes. This re

number of messages ranging from 1,000 to 10,000.

Both MPICH-GM and BMI were configured by default to switch

handshake mode at the 8 Kbyte message size. This can clearly be se

from the jumps in the curve. BMI performance (both for normal a

memory) dropped off at 8 Kbytes, while the MPICH-GM performanc

Kbytes. The MPI performance below 8 Kbytes was surprisingly poo

Figure 5.9 results from the same test run as measured from th

The results are very similar, except that MPICH-GM performance

quite as much on the 8 Kbyte boundary as it did in the send case.

Figures 5.10 and 5.11 show the bandwidth as measured for the tran

bytes of data using messages sizes ranging from 10,000 bytes to 100

BMI method plateaus at about 27 Mbytes/sec if the memory buff

cated in advance. The steps carried out for this transfer are identic

Figure 5.8: Small message GM bandwidth (send)

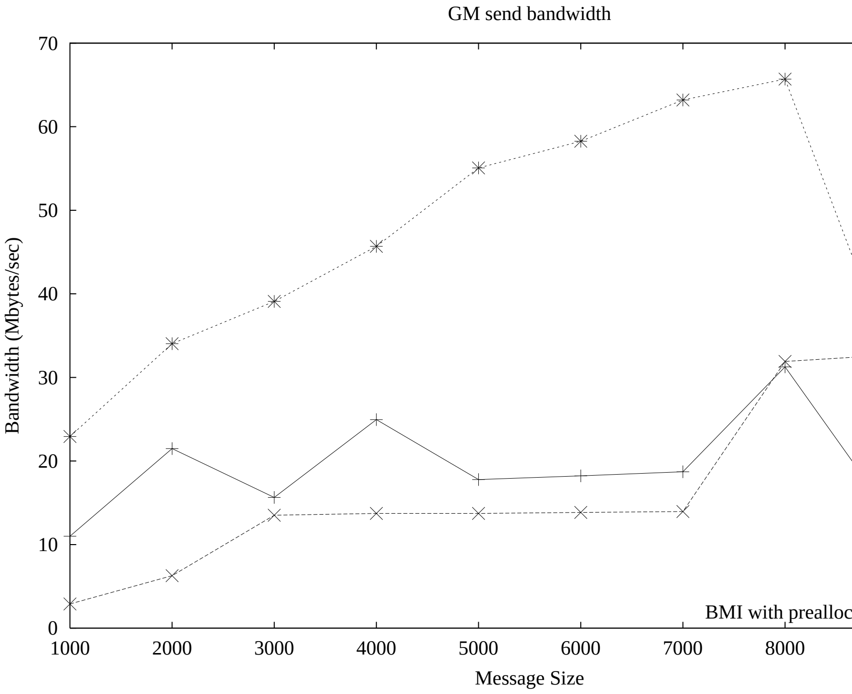


Figure 5.9: Small message GM bandwidth (receive)

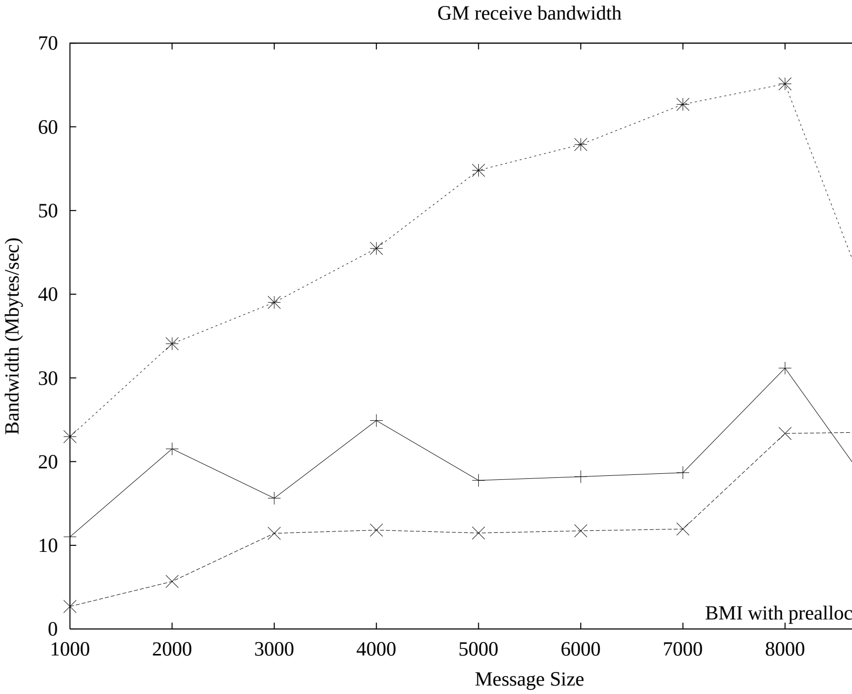


Figure 5.10: Large message GM bandwidth (send)

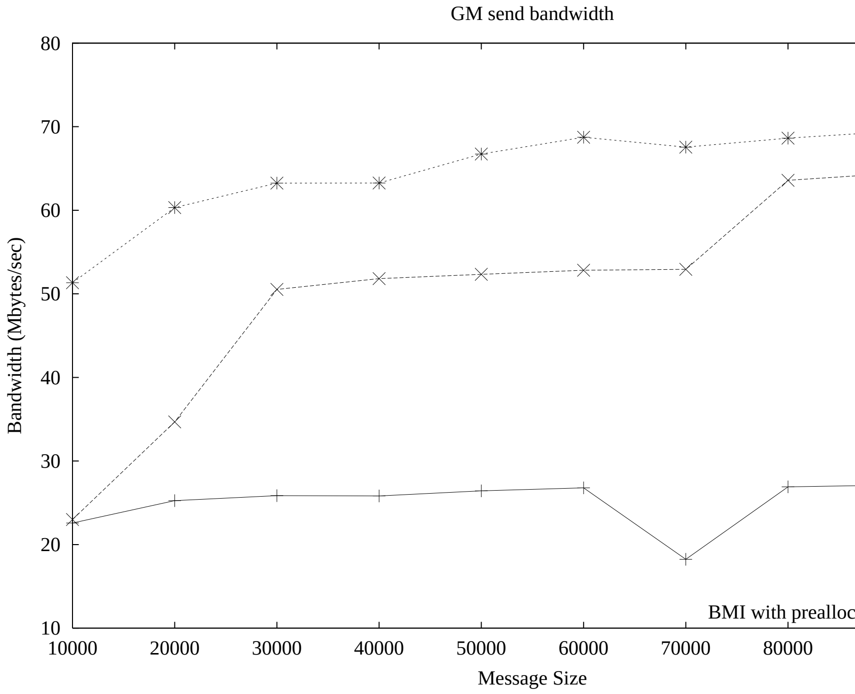


Figure 5.11: Large message GM bandwidth (receive)

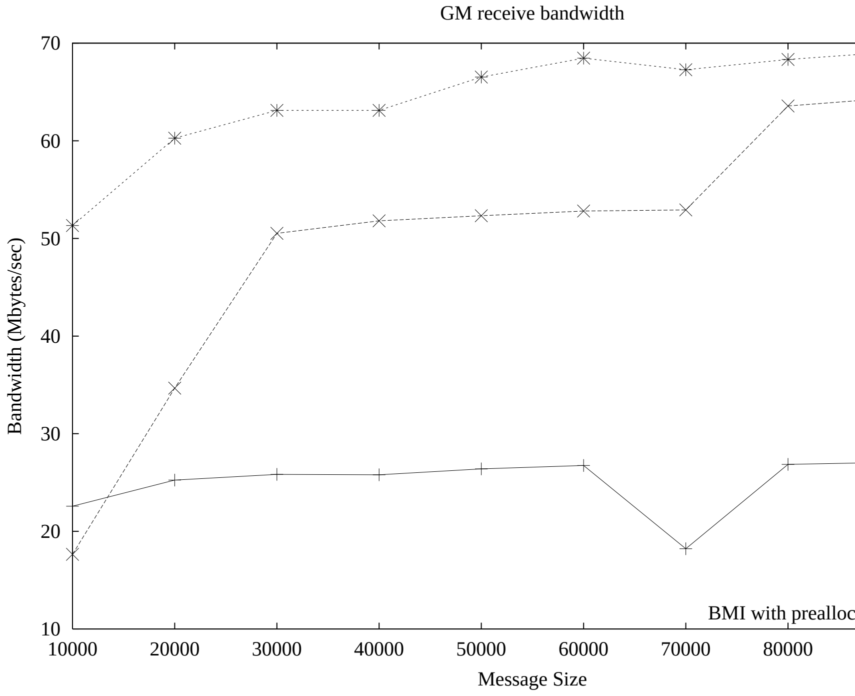
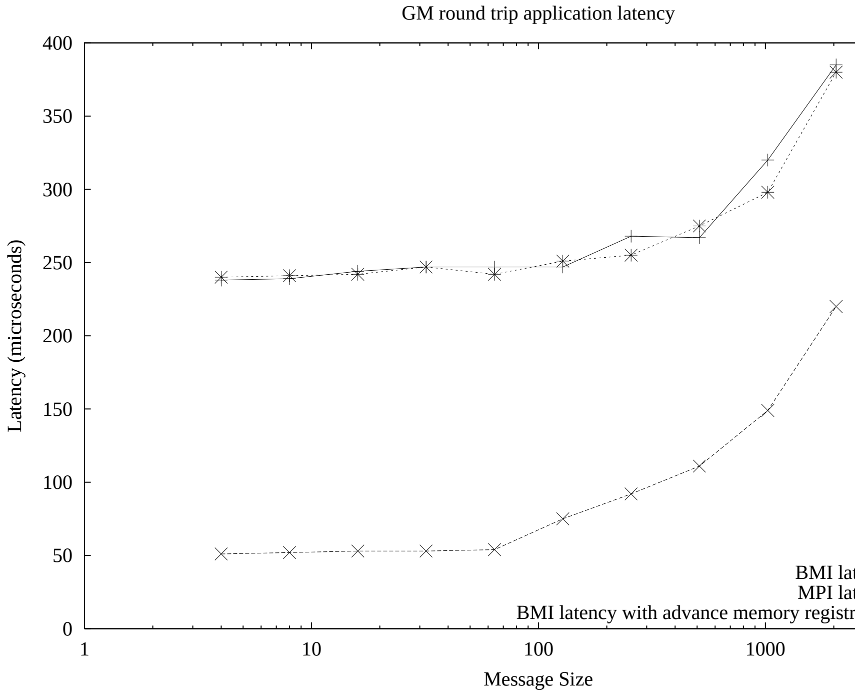


Figure 5.12: GM round trip latency



with preallocation except that an extra memory copy is incurred fo
The penalty introduced by this approach is quite evident here. S
plores methods of reducing this overhead for memory buffers that a
in advance.

5.3.2 Latency

Round trip application was measured for GM in the same manne
preceeding TCP/IP tests. The results are shown on a logarithmic
5.12.

Preallocation of memory buffers had negligible impact on overa
which demonstrates that the additional memory copy is not terrib
buffers in this size range.

Figure 5.13: GM many to one performance (10K messag

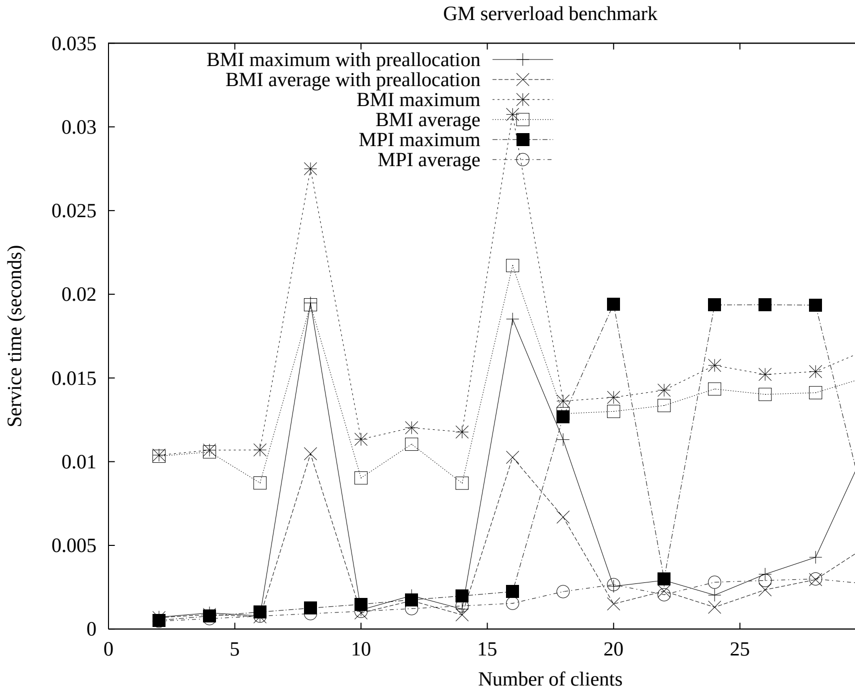


Figure 5.12 also shows that the BMI method was unable to ob
tency figures demonstrated by MPI at any message size. The disc
large, with the BMI approach taking at least four times as long as t
implementation.

5.3.3 Simulating server load

The serverload benchmark was executed over GM following the sam
in the TCP/IP tests in section 5.2.3. Figure 5.13 shows the results of
for 10,000 byte message sizes.

Performance was erratic for all three cases (MPI, BMI, and BMI
allocation). The MPI performance was very similar to the BMI pe
preallocation. The BMI performance without preallocation was foun
tially slower.

Figure 5.14: GM many to one performance (100K messa

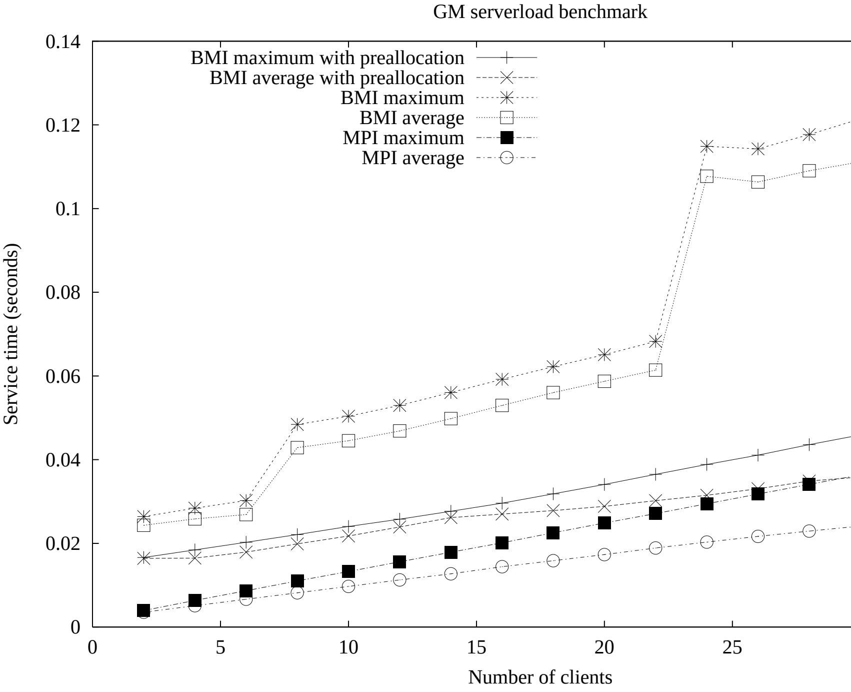
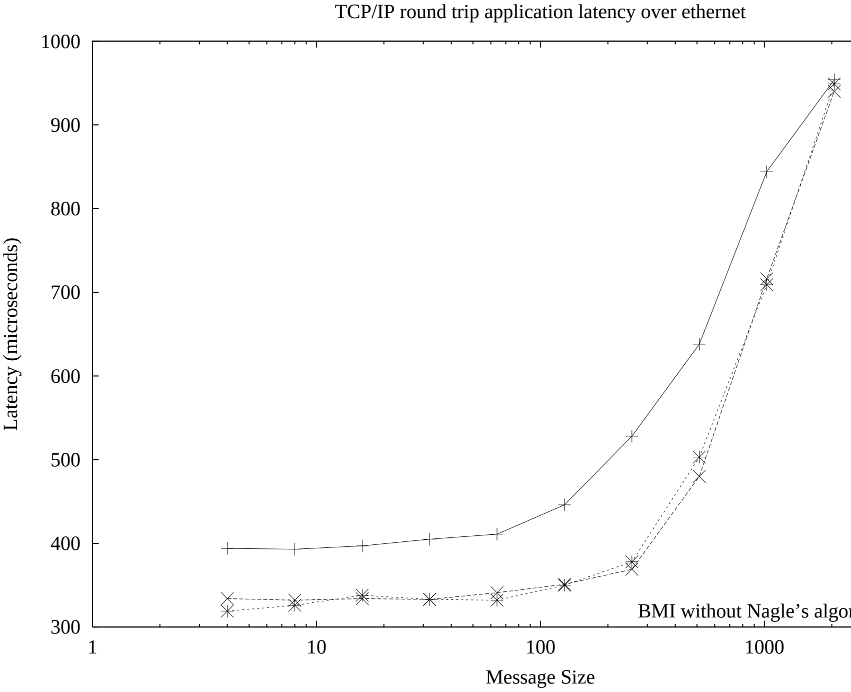


Figure 5.14 shows the results of the same tests when using 100,0
sizes. The performance is much more predictable in this case.
performance exceeded that of BMI. The margin was especially larg
buffers were not preallocated.

5.4 Evaluating problem areas

The initial BMI performance results indicate that some aspects of BM
ing the potential offered by the underlying communications systems
we will attempt to analyze and address these issues.

Figure 5.15: TCP/IP round trip latency



5.4.1 TCP/IP method latency

Figure 5.5 indicates that the BMI method is exhibiting relatively poo
terms of latency. The MPICH TCP/IP implementation is as much as
faster in round trip application measurements.

In order to find the source of this problem, the MPICH impleme
alyzed first. It was discovered that MPICH disables Nagle's algor
on BSD based systems that support this option. This option is co
TCP_NODELAY flag in the setsockopt() function. See section 4.2.3
mation about Nagle's algorithm.

Figure 5.15 shows the results of the same latency test with the T
option set for all sockets controlled by the BMI method. MPI and BM
identical behavior with this approach.

Figure 5.16: small message TCP/IP bandwidth (send

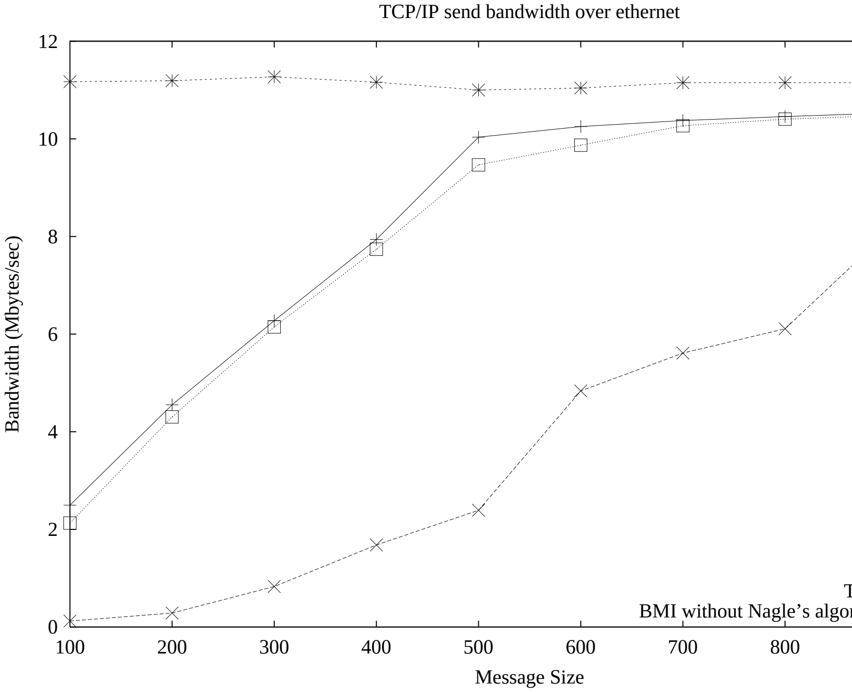
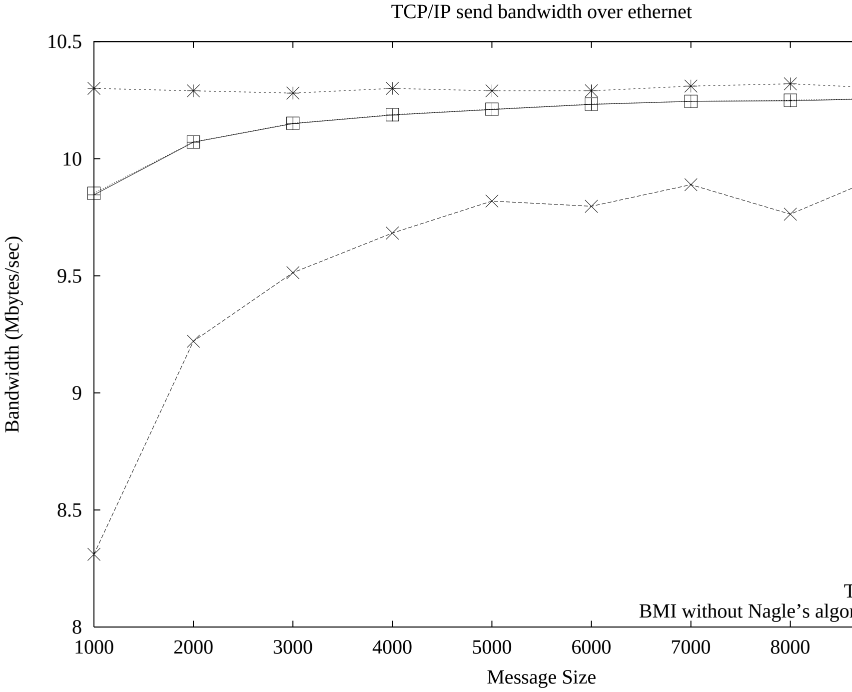


Figure 5.17: larger message TCP/IP bandwidth (send



The raw bandwidth tests were also reevaluated to see if this app

BMI bandwidth results. Figures 5.16 and 5.17 show that this opti

effect on the overall TCP/IP bandwidth. The only measurable imp

messages smaller than 700 bytes. In order to reduce even this im

possible to implement an adaptive delay policy. Such a policy could

or enable Nagle's algorithm on a per socket basis depending on the c

the messages that are queued up to be sent for that socket.

5.4.2 GM bandwidth for large messages

Initial bandwidth measurements for the GM method indicate that p

quite good *if user buffers were allocated in advance using the BMI li*

if existing user buffers were utilized, then the performance was limit

Mbytes/sec (see Figure 5.10). This is not acceptable. The MPI perf

indicate that a much higher performance can be obtained without usi

As explained in section 4.3.1, the GM library requires that all buf

mitted be located in regions of memory that have been prepared in a

purposes. The initial BMI implementation met this requirement by

DMA-able buffers for each message and copying the user buffer int

operation clearly was consuming a relatively large amount of time f

sizes.

The alternative is to actually register the existing user buffers usin

memory function call. Buffers registered in this manner must later b

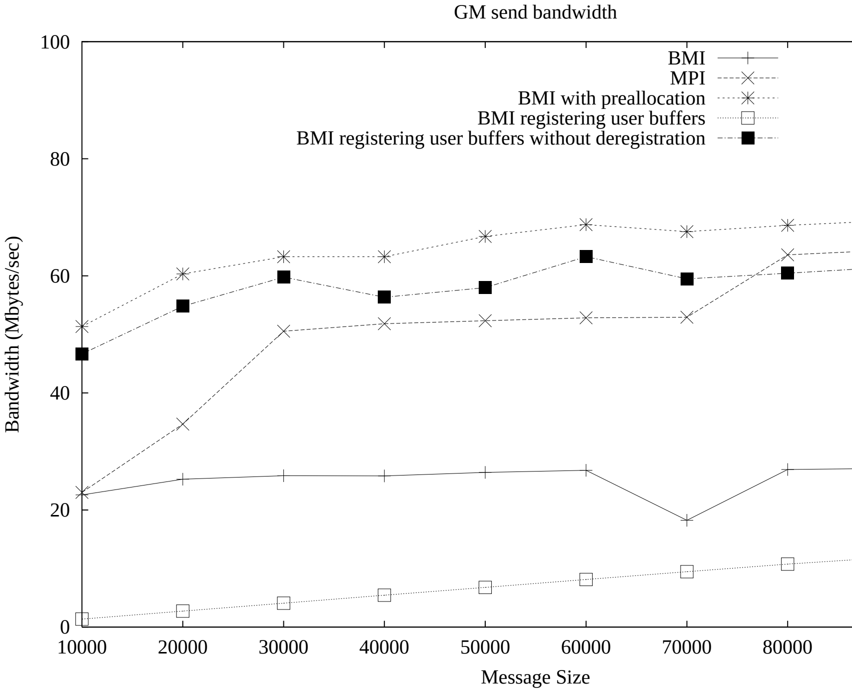
the gm_deregister_memory function. One drawback to this approach

BMI header information can no longer be piggy-backed onto the m

cause there is no way to append contiguous memory to the message

header information is not necessary for BMI messages that are tra

Figure 5.18: Large message GM bandwidth (send)



handshaking protocol, because the control messages contain a comp for the message.

To explore this possibility, the BMI GM implementation was mo user buffers that had not been allocated using BMI. Each buffer was posted, and then deregistered on completion. This would avoid th copy step. However, Figure 5.18 shows that the performance for th absolutely terrible (see the “BMI bandwidth registering user buffers”

The MPICH-GM implementation was then inspected to discove lution to this problem. As it turns out, the MPICH device uses a management system that registers user buffers as needed, but does no lease them upon message completion. The buffers remained registere to the user. This optimization is intended to be helpful if buffers are

The MPICH device keeps up with which memory regions have been only deregisters regions if system memory resources run low.

The BMI bandwidth benchmark used for the preceding experime Mbyte of system memory per host. Therefore, for experimentatio possible to disable deregistration entirely to observe the impact. Fi bandwidth registering without unregistering user buffers" data po result. The performance was *much* higher, and in fact exceeded MP most message sizes shown. Note that each buffer was used only o buffers was not an issue.

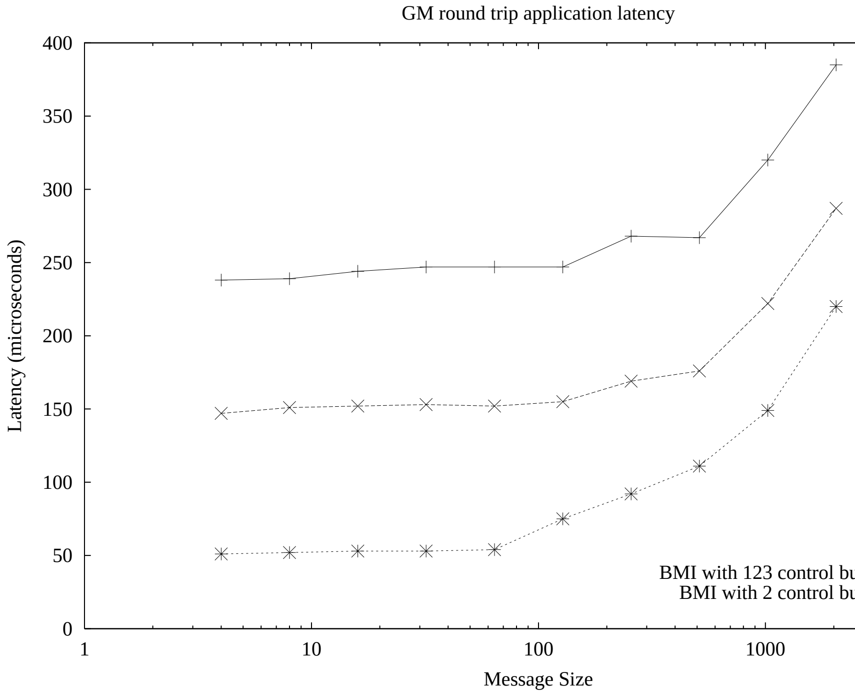
This experiment revealed that the act of deregistering GM mem tremely high overhead, which was an unexpected result. In order for with existing buffers to achieve the same level of MPICH, the B implement a similar memory management library which takes a l deregistering system memory.

Note that receive buffers in the BMI GM implementation are alwa is no easy way to relax this constraint, due the messaging system's in the role of each receive buffer posted. Each time a message is rece analyzed to determine which user buffer it matches. The data is the buffer.

5.4.3 GM method latency

Earlier GM method measurements (particularly for the round trip la load applications) show that the latency of the implementation is expected potential. The MPICH-GM implementation is as much as f faster in terms of latency.

Figure 5.19: GM round trip latency



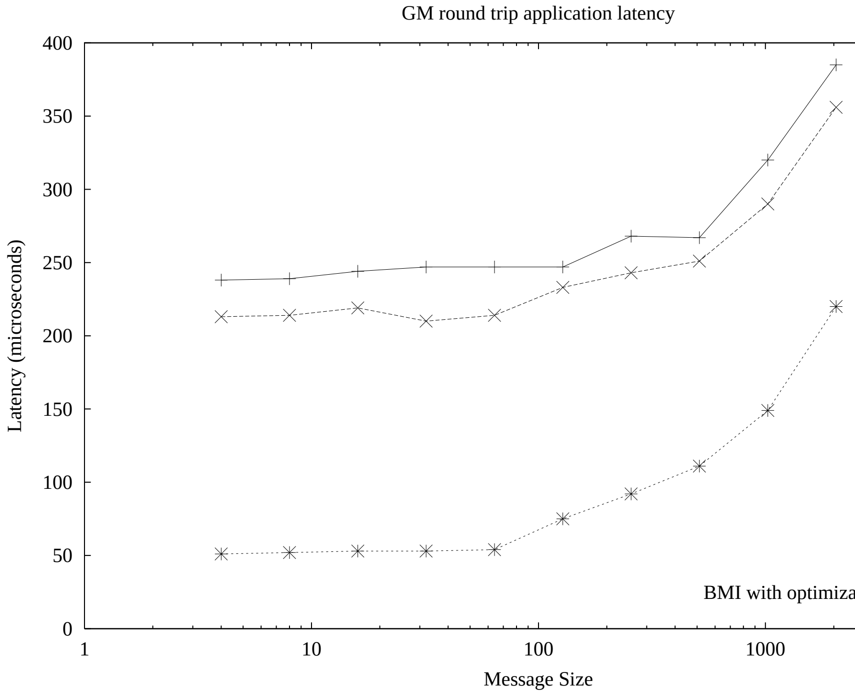
We therefore set out to investigate this discrepancy. A variety were attempted, ranging from alternative GM interface functions to cation of internal structures. A few interesting results were found.

Note that the GM method implementation (like the MPICH-GM requires that a collection of receive buffers be posted as soon as the m This ensures that it will be able to accept control messages from i discovered that tuning the number of buffers that were provided impact on latency, as shown in Figure 5.19.

The extra line in this graph indicates performance if only 2 buf in advance, rather than the typical 123. Note that the time required these buffers is not included in these measurements.

The difference in performance is nearly 100 microseconds. This *double* the total round trip time of the MPICH-GM implementati

Figure 5.20: GM round trip latency



because the MPICH-GM implementation defaults to posting 123 buf

clearly it does not suffer from the same performance penalty that

mentation does. This seems to indicate that there is a subtle di

MPICH-GM and BMI methods approach communication managem

to be discovered. This may be caused by an operation that scales

number of posted buffers that should be avoided.

Note that a production quality method implementation *must* pro

two control buffers at startup time. Otherwise the user risks the dange

the amount of available buffers when too many control messages floo

The algorithm that determines how much work should be done dur

function call was also found to have a significant impact on latency.

proach was to only perform one gm_receive operation per cycle. The

was modified so that after each call to gm_receive(), gm_receive_pe

called to determine if there was any more work that could be immedi
If so, gm_receive() was called repeatedly (up to a bounding limit)
queue was emptied. This cuts down on the overall number of func
BMI interface per communication. The results of this optimizatic
Figure 5.20. This trimmed around 20 microseconds from the total r

Despite these discoveries, the BMI method still displays inferior
tency sensitive applications. The servload application is sensitive to
and thus will not be re-evaluated until a solution is found to the lat

Chapter 6

Conclusion

We presented the Buffered Message Interface in order to meet the
work abstraction layer for implementing parallel file systems on Linu
interface provides a simple application interface for accessing all t
communications network necessary for high performance I/O.

BMI demonstrates that a modular mechanism can be built and
for communicating over various dissimilar networks. None of the perf
applications were recompiled to adapt to the protocols used. All tha
was the presence of the proper BMI module and (in this case) a c
indicating which module to use.

The BMI interface was also capable of communicating with mu
protocols simultaneously. The performance testing of this feature is b
of this document, but the semantics are implemented correctly.

We also observed that the efficiency of this implementation was on
tations in the majority of the tested scenarios. We hope that the obv
will be addressed in future work on the method modules.

BMI will be a key component of the forthcoming Parallel Virt
version 2. PVFS2 is being designed with collaboration between Cler

Argonne National Laboratory, and Goddard Space Flight Center to
step in parallel I/O technology to Linux clusters. BMI will insure tha
tation keeps pace with trends in networking technology without the
of the core file system. Once this new PVFS implementation has ar
able to evaluate the performance and usability of BMI within the co
file system implementation.

6.1 Future work

There is still much research and development to be performed wit
Message Interface. This work revolves around three critical areas:
existing protocol methods, expansion into new protocols, and brin
implementation up to production level availability.

6.1.1 Improvement of existing methods

Both the TCP/IP and GM method implementations were quite succ
both messaging systems have room for improvement. Sections 4.2
several possible optimizations, only a few of which have been realiz
latency performance of the GM module is of particular interest after
in section 5.4.3.

6.1.2 Expansion of supported methods

TCP/IP and GM methods were implemented as a proof of conce
networks. There are several other method implementations that co
however. Some possibilities include shared memory, VIA, and UDP
of these should be feasible within the previously defined Buffered M
VIA in particular should be relatively straightforward to implement

several overall concepts with the GM interface. A shared memory perhaps be most interesting in terms of proving BMI's success in abstra does not fall into the broad category of message passing communicati interesting because it would prove the feasibility of implementing within a BMI method.

6.1.3 Scheduling

BMI was designed so that it will be possible to couple it with a higher capable of making scheduling decisions. This is of particular intere design. A scheduling mechanism should be able to obtain load inform by using the get_info() function. Policy hints may be provided usin function. This approach remains untested at this time, however.

6.1.4 Production level availability

The Buffered Message Interface is relevant not only as a research as a production level component of a true parallel file system. Th emphasis on its ability to provide production level robustness. In pa be very resilient (or at least very predictable) in the face of individua A file system cannot tolerate deadlock or critical failure of the und subsystem.

Extensive stress testing will be necessary to bring BMI to product All of the tests performed in this document were done with an emph ing performance. More rigorous testing should examine degenerate network failures to ensure that the interface is robust.

Bibliography

[1] The Dolphin SCI interconnect white paper. Technical report, D
nect Solutions, Inc., February 1996.

[2] Donald J. Becker, Thomas Sterling, John E. Dorband Daniel Sav
Ranawak, and Charles V. Packer. Beowulf: A parallel workstat
computation. In *Proceedings, International Conference on Par*
1995.

[3] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seit
and W. Su. Myrinet a gigabit per second local area network. *I*
February 1995.

[4] Philip Buonadonna, Andrew Geweke, and David Culler. An imp
analysis of the Virtual Interface Architecture. In *Proceedings of*
Florida, November 1998.

[5] Bill O. Callmeister. *POSIX.4: Programming for the Real Worl*
Associates, Inc., Sebastopol, CA, 1995.

[6] Philip H. Carns, Walter B. Ligon III, Robert B. Ross, and Rajeev
A parallel file system for linux clusters. In *Proceedings of the 4t*
Showcase and Conference, pages 317–327, Atlanta, GA, October
Association.

[7] Chiba City, the Argonne scalable cluster. `http://www.mcs.anl`

[8] G. Chiola and G. Ciaccio. GAMMA: a low-cost network of wor
on active messages. In *Proceedings of PDP'97 (5th EUROMICR*
Parallel and Distributed Processing), London, UK, January 199

[9] Peter F. Corbett and Dror G. Feitelson. The Vesta parallel fil
Transactions on Computer Systems, 14(3):225–264, August 199

[10] Emulex IP storage networking. `http://wwwip.emulex.com/ip/`

[11] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skj
performance, portable implementation of the MPI message-p
standard. *Parallel Computing*, 22(6):789–828, September 1996.

[12] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2 tures of the Message-Passing Interface.* MIT Press, Cambridge,

[13] M. Halem, F. Shaffer, N. Palm, E. Salmon, S. Raghavan, and L. we avoid a data survivability crisis? *Science Information Syst* (51), 1999.

[14] IEEE/ANSI Std. 1003.1. Portable operating system interface (System application program interface (API) [C language], 1996

[15] W. B. Ligon III, S. P. McMillan, and R. B. Ross. Tuning TCP beowulf computers. Technical report, Parallel Architecture Rese Clemson University, 1999.

[16] M-VIA: A high performance modular VIA `http://www.nersc.gov/research/FTG/via/`.

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

[18] Message Passing Interface Forum. MPI-2: Extensions to the Interface, July 1997. `http://www.mpi-forum.org/docs/docs.`

[19] Myrinet software and documentation. `http://www.myri.com/s`

[20] John Nagle. Congestion control in ip/tcp internetworks (rfc- report, IETF Network Working Group, January 1984.

[21] Avneesh Pant, Sudha Krishnamurthy, Rob Pennington, Mike Qian Liu. VMI: An efficient messaging library for heterogene munication. Technical report, National Center for Supercomput (NCSA) at the University of Illinois at Urbana-Champaign, 200

[22] David A. Patterson and John L. Hennessy. *Computer Architect tive Approach (2nd Edition).* Morgan Kaufmann Publishers, Inc CA, 1996.

[23] Daniel Ridge, Donald Becker, Phillip Merkey, and Thomas St Harnessing the power of parallelism in a pile-of-pcs. In *Pr Aerospace*, 1997.

[24] Robert B. Ross. *Reactive Scheduling for Parallel I/O Syste* Electrical and Computer Engineering Dept., Clemson Universit

[25] Test TCP. `ftp://ftp.arl.mil/pub/ttcp/`.

[26] Hiroshi Tezuka, Atsushi Hori, Yutaka Ishikawa, and Mitsuhisa Operating System Coordinated High Performance Communica

Bob Hertzberger, Peter Sloot, editor, *High-Performance Com*
working, volume 1225 of *Lecture Notes in Computer Science*,
Springer-Verlag, April 1997.

[27] Rajeev Thakur, William Gropp, and Ewing Lusk. On implem
portably and with high performance. In *Proceedings of the Six*
Input/Output in Parallel and Distributed Systems, pages 23–32,

[28] VI architecture. `http://www.viarch.org`.

[29] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schaus
sages: a mechanism for integrated communication and computat
ings of the 19th Int'l Symp. on Computer Architecture, Gold C
May 1992.