Proceedings of the IASTED International Conference
PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS
August 21-24, 2001, Anaheim, California, USA

QMP : A Fast Communication Protocol for COW's

ATUL SINGH, SHUVABRATA GANGULY and RAMAMURTHY BADRINATH

Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur-721302, India
{atul,sganguly,badri}@cse.iitkgp.ernet.in

ABSTRACT

Common cluster computing applications (like distributed databases and PVM applications) use the standard TCP/IP as their communication protocol. For a relatively error-free cluster environment, such general purpose protocols lead to an underutilization of the network bandwidth and consequently, a degradation in performance.The primary objective of the QMP (Quick Message Protocol) is to provide network support tailor-made for the cluster environment. There are many protocols [1, 2] for cluster applications and there are efficient memory management schemes [3] but there has been little work towards integration of these. In QMP, we have integrated existing protocols and schemes and built a system which outperforms TCP/IP in a LAN. We have implemented strip-down version of XTP [4] for LAN environment with zero-copy send [3]. We have also provided **BSD Socket** interface for easy user portability.We evaluated QMP's performance by running distributed benchmarks using TCP and then using the QMP protocol suite.The results were obtained on 233 MHz Intel Pentium IIs running Red Hat Linux 6.2 with Compex 100 Mbps compatible network adapters.

KEYWORDS

XTP, Zero-Copy, Cluster Computing, QMP, QBUF

1. INTRODUCTION

The increased availability of high-speed local area networks has shifted the bottleneck in local-area communication from the limited bandwidth of network fabrics to the software path traversed by messages at the sending and receiving ends. In particular, in a traditional UNIX networking architecture, the path taken by messages through the kernel involves several copies and crosses multiple levels of abstraction between the device driver and the user application[5]. The resulting processing overheads limit the peak communication bandwidth and cause high end-to-end message latencies. The effect is that users who upgrade from ethernet to a faster network fail to observe an application speed-up commensurate with the improvement in raw network performance. A solution to this situation seems to elude such systems to a large degree because many of them fail to recognize the importance of per-message overhead and concentrate on peak bandwidths of long data streams instead. The increased use of techniques such as distributed shared memory, remote procedure calls, remote object-oriented method invocations, and distributed cooperative file caches will further increase the importance of low round-trip latencies and of high bandwidth and low latency. For a more detailed introduction to these and other issues in cluster computing see [1].

[*]Contact author: Ramamurthy Badrinath

2. PRELIMINARIES

Computing and storage over distributed environments such as clusters of workstations [6] and personal computers that are connected by local and wide area networks has very high potential, since it leverages existing hardware and software and enables affordable parallel and distributed applications.Fast communication is one of the key areas of research in this fast growing environment of distributed computing. It is also the focus of our project with the ultimate goal of creating reliable cluster environments that are based on fast communication.

The basic assumptions in a **cluster environment** are

- *Very low error rate :* The proximity of nodes in the cluster environment immensely reduces the probability of data corruption. Moreover, sophisticated digital transfer media keep the error rate as low as possible.

- *Packet receive sequence :* It is also assumed that in a cluster environment data packets are received in the same order as they are sent. Due to the lack of alternate paths of traversal in such a close-knit network (or cluster), out-of-order packets are not generally received.

- *Packet Loss :* The only way a data packet can be lost (and therefore, an out-of-order packet received) is when the receive side buffer is full and the received packet is dropped.

- *Checksumming requirement :* Since the probability of data corruption is extremely low, the hardware (CRC) checksum is assumed to be a sufficient checksum. No software checksum (like tcp checksum) is required.

3. QMP - QUICK MESSAGE PROTOCOL

QMP defines the mechanisms necessary for delivering user data from one end-system to one or more other end-systems. Well-defined packet structures, containing user data or control information, are exchanged in order to effect the user data transfer. The control information is used to provide the requested level of correctness and to assist in making the transfer efficient. Assurance of correctness is done via error control algorithms and maintenance of of a connection state machine.It is a strip-down version of **XTP** [4].The objectives of the QMP are as follows:

- to co-exist with the TCP/IP protocol suite.
- to reduce protocol processing overheads.
- to provide BSD socket interface in order to allow easy porting of user programs

3.1 Design of QMP

The collection of information comprising the **QMP** state at an end-system is called a **context**. This information represents one instance of an active communication between two or more **QMP** endpoints. A context must be created, or instantiated, before sending or receiving QMP packets. There may be many active contexts at an end-system, one for each active conversation. Each end point (consisting of sock and context) manages both outgoing data stream and incoming data stream. A data stream is an arbitrary length string of sequenced bytes, where each byte is represented by a sequence number.The aggregate of active contexts and the data streams between them is called and *association*.

A context at an end-system is initially in a quiescent state. A user awaiting the start of an association requests that the context be placed into the listening state. The context now listens for an appropriate FIRST packet. The FIRST packet is the initial packet of an association. It contains explicit addressing information. The user must provide all of the necessary information for QMP to match an incoming FIRST packet with the listening context.

At another end-system a user requests the establishment of an association. The context handling this user moves from a quiescent state to an active state, where it constructs a FIRST packet with explicit addressing and service information obtained from the user. The FIRST packet is sent via the underlying data delivery service.

When the FIRST packet is received by the destination end-system, the address and service information in the FIRST packet is compared against all listening contexts. If a match is found, the listening context moves to the active state. From this point forward an association is established, and communication can be completely symmetric since there are two data streams, one in each direction,in an association.Also, no other packet during the lifetime of the association will carry explicit addressing information. Rather, a unique “key” is carried in each packet, which allows the packet to be mapped to the appropriate context.

Once all the data from one user have been sent, that data stream from that user’s context can be closed. Sentinels in the form of options bits in a packet are exchanged to gracefully close the connection. When both users are done, and both data streams closed, the contexts move into the inactive state. One of the contexts will send a sentinel that causes the association to dissolve. At this point, both contexts return to the quiescent state. Fig. 1 shows the state of context during the lifetime of a connection.

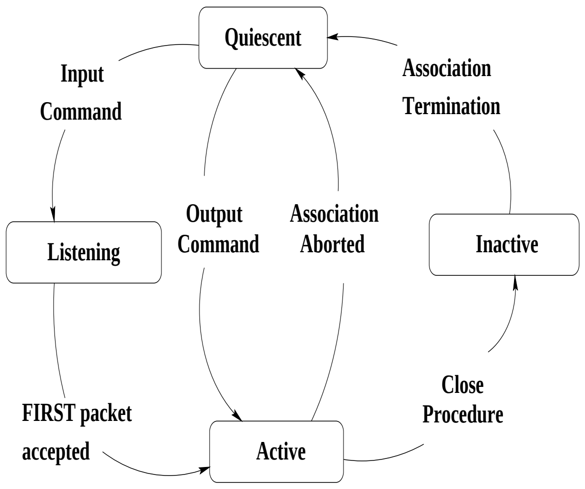

Figure 1. Context State Diagram

3.1.1 Connection Establishment

An association is established when a listening context receives a **FIRST** packet, and both this and the initiating context have moved into the active state. A received **FIRST** packet is matched against all listening contexts to find one that will accept the incoming data stream. The **full context lookup** procedure maps an incoming packet to the appropriate active context if the key value in the incoming packet is not a return key (that is, the **RTN** bit is not set in the key field of the packet). The **abbreviated context lookup** procedure is an optimized method for mapping an incoming packet to the appropriate context without using the translation map. By definition, a key value is generated by a host to be unique within that host. Then the key value is placed into the key field of the **FIRST** packet. When the **FIRST** packet is received and given to the matching listening context, that context notes the **FIRST** packet’s key value, sets its **RTN** bit, and uses this value as the return key. The return key value is placed in any packets sent in the return direction to the host that sent the **FIRST** packet.

3.1.2 Flow Control

QMP employs an optimistic flow control mechanism. **QMP's** flow control is based on a sliding window of sequence numbers. A sequence number is assigned to each output byte of the data stream, starting with the initialized sequence value.

Two fields in control packets are used in the flow control procedures. The value in the **alloc** field in a control packet sent to the transmitter indicates the sequence number not to be exceeded by the transmitter. This value represents the upper edge of the flow control window. The value in the **rseq** field in a control packet sent to the transmitter is one greater than the last byte contiguously received. This value serves as the lower edge of the flow control window.

QMP provides user to set the frequency at which control packets would be sent to identify the status of receiver. For ex, if user sets this frequency to 1, then with every packet we set the **SREQ**.

3.1.3 Error Control

Error control in QMP is based on the exchange of information regarding lost or damaged data and the retransmission of these data. Each packet is examined for damage by performing a checksum, which is performed at hardware level, i.e., CRC. Lost data are detected and recovered using an acknowledgement and retransmission procedure. The loss of a status request is detected by a timer; recovery in this case starts an exchange of packets designed to synchronize the endpoints of the association.

4. QBUF : A ZERO-COPY I/O FRAMEWORK

Traditional UNIX I/O interfaces are based on copy semantics, where read and write calls transfer data between the kernel and user-defined buffers. Although simple, copy semantics limit the abilities of the operating system to efficiently implement data transfer operations. Copy semantics suffer mainly from two deficiencies :-

- Overheads :- Excessive copying of data leads to lower bandwidth and higher CPU utilization. Data copying is a per-byte operation, which results in an increase in copying costs as the size of transfer increases. Some fast protocols [7],[8] employ checksum-offloading to prevent such operations.

- Cache pollution :- Copying being a data touching operation leads to the pollution of cache. This essentially means that useful entries are flushed from the cache due to data copying.

We have adopted an entirely different approach in order to do zero copy I/O [8]. Instead of **copying** buffer contents we **transfer** buffer ownership between the user process and the kernel. In this project we designed and implemented a **Zero-Copy** IO framework for the Linux kernel.

4.1 Design of QBUF Memory Management Scheme

In QBUF we have changed the semantics which define the transfer of data between user and kernel buffers. Instead of copying buffer contents, buffer ownership is transferred [3, 11, 10]. A **"qbuf"** can be owned either by the user or the kernel or it can be in the free pool. A **"qbuf"** allocated by a user process is transferred to the kernel when the process executes the "write" system call. After I/O is completed on that buffer the kernel returns the buffer to the free pool. Similarly for the "read" system call a **"qbuf"** is allocated by the kernel and transferred to the user process. Once the user process frees the **"qbuf"** it is returned to the free pool.Clearly with this scheme a buffer that has been transferred to the kernel cannot be reused after the system call. That buffer will be returned to the free pool after I/O is completed.

4.2 The QBUF Memory Model

Every process that intends to use the **QBUF** system must acquire an instance of the QBUF free pool. The QBUF free pool consists of **"zones"**. A zone is a contiguous region of the virtual memory (VM) of the process. Each "zone" consists of 32 **blocks**. Currently the size of a block is fixed and is equal to the size of a page (PAGE_SIZE). Each zone maintains a free list of blocks in that zone. There is also a lock associated with each zone.

Allocations take place as a multiple of blocks. One restriction of our scheme is that allocations cannot span multiple zones. Hence, currently, the maximum allocation size is 32 * 4K = 128K. The requested allocation size is rounded up to the next higher multiple of block size. The free-list for a zone is kept as a bitmap. This has been shown in Fig 2. For example bit 0 of the bitmap is 1 if block 0 is free otherwise it is 0. Allocation is first fit. If there are not enough blocks to satisfy the requested size in a zone then the next zone is tried. Hence while allocating the bitmap is searched for a series of requested number of 1's. Once that is found all 1's are converted to 0's to mark those blocks as "used".

Since allocation and deallocation of buffers from the QBUF pool can be done concurrently by the kernel and the user process there must be some mechanism for protecting the data structures. For example suppose the user process tries to allocate two blocks. It finds two consecutive 1's in the bitmap, but before it can change them to 0's, the process is preempted by an interrupt. In the interrupt handler the kernel tries to allocate from the same bitmap and resets the same 2 bits. Hence this leads to a race condition. To prevent this we have kept a atomic lock variable per zone. Before allocating or deallocating from a zone the lock must be acquired. While allocating it is first checked whether the zone is locked. If it is, then move on to the next zone. In deallocation there is a minor difference. While allocation can be done from any zone, de-allocation must be done on the zone from which the buffer was first allocated. What

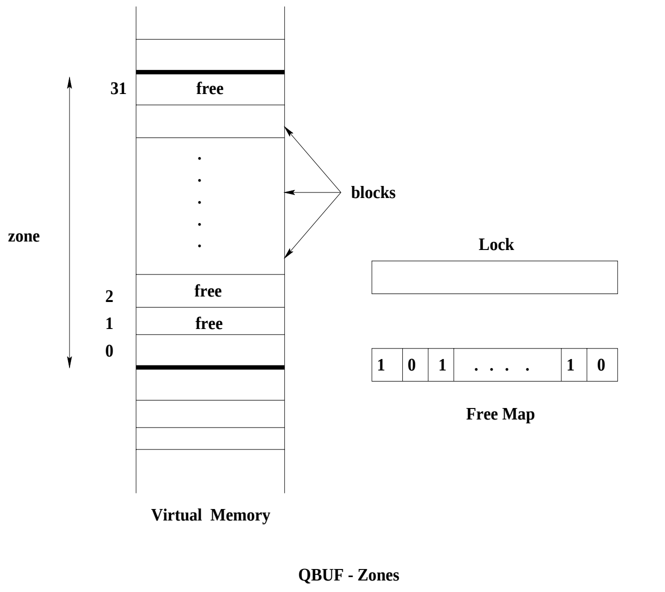

Figure 2. Qbuf Zone Model

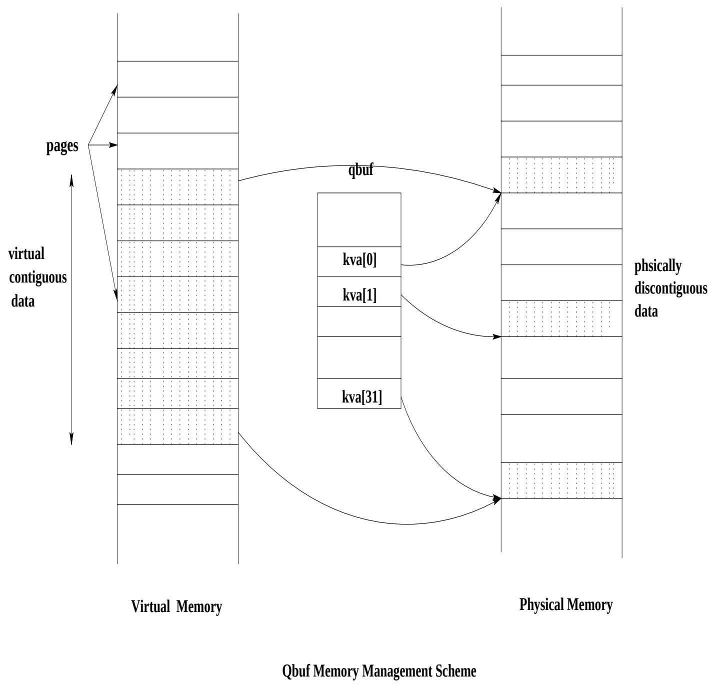

Figure 3. QBUF Architecture

should be done when that zone if locked ? In our scheme we maintain a deferred deallocation work queue to handle such cases. If, while de-allocating, the zone is found to be locked then the deallocation is queued up in the work queue. The work queue is drained whenever the kernel tries to allocate/deallocate from the QBUF pool for the process.

4.2.1 Transferring Buffer Ownership

When the user allocated a ”qbuf” it is owned by the user process. There are two new system calls to support transfer of qbufs between the user process and the kernel. The **”qb_write”** system call transfers ownership from the user process to the kernel. The **”qb_read”** system call transfers ownership from the kernel to the user process.

When a user process executes the ”qb_write” system call the ownership of a qbuf is transferred to the kernel. The underlying physical pages of the ”qbuf” are wired in order to prevent them from being paged out while I/O is taking place. After I/O is complete the pages are unwired and ”qbuf” is returned to the QBUF free pool.

In the ”qb_read” system call the kernel allocates a ”qbuf” and wires the underlying physical pages. Then I/O is done on those pages. After I/O is over the pages are unwired and the ”qbuf” is transferred to the user process. There is a minor difference between ”qb_read” and traditional ”read” calls. In ”qb_read” the user process obtains a qbuf from the kernel. In traditional ”read” the user allocates a buffer and passes its address to the kernel. On the other hand in ”qb_read” the kernel passes the address of the qbuf to the user process. Hence the user process passes the address of a pointer to ”qb_read” to obtain the address where the qbuf is mapped. Hence the interface is qb_read(fd, &p, &len).

4.2.2 Implementation

The **QBUF** system has been implemented as a character device ”/dev/qbuf”. This was done so that we do not have to make a lot of changes to the core kernel. Any process which wants to use the QBUF scheme opens ”/dev/qbuf”. Each QBUF zone is obtained using anonymous ”mmap”. There is a control structure **”struct qb_ctl”** which contains metadata. This structure is locked and is shared by the user process and the kernel. It contains information like the number of zones, block size, bitmaps and locks for each zone. Information about each zone is encapsulated in a ”zone descriptor” structure which is a part of the control structure.

Since the user process and the kernel needs to access the control structure, it is locked i.e., it cannot be paged out. Memory for the control structure is obtained by mmap’ing a page-sized buffer and then calling mlock to wire the page. A special ioctl serves to inform the kernel where the control structure resides.

4.2.3 Kernel Data-Structures

Each process which registers itself to the QBUF system obtains a **qbuf handle** (qbuf_handle_t). The handle serves to identify that instance. Every allocation/deallocation is done using the handle. The I/O subsystem must also register with the QBUF system using the qbuf_register() call which returns a qbuf handle.

Internally a qbuf is represented by a ”struct qbuf”. The **qbuf** architecture is shown in Fig 3. It contains various information like which zone it belongs to, the user virtual address of the buffer and number of blocks in the qbuf. In addition it contains the physical addresses of each block of the qbuf. This is maintained in the qbuf $\rightarrow$ kva[] vector. In order to encapsulate a user virtual address in a "struct qbuf" qbuf_wrap() must be called which wires all the underlying physical pages and returns a "struct qbuf *". Then I/O is started on the qbuf. In our case we allocate skbuffs (linux network buffers) which point to qbuf memory and transmit them. When the acknowledgement for the last skbuff which uses this qbuf comes we return the qbuf to the free pool. To do this qbuf_vfree() is called. This function unwires the underlying physical pages and frees the corresponding bits in the free-list bitmap.

5. PERFORMANCE

We evaluate the performance of our protocol and memory scheme by measuring its bandwidth [9] and latency.For measuring the bandwidth and latency provided by TCP/IP on our test-bed, we used **netperf** tool.

5.1 Bandwidth

For measuring bandwidth provided by our **QMP** protocol, we transferred different sized buffers from one node to another. The Fig 4 compares the bandwidth provided by QMP with and without QBUF memory scheme and TCP/IP. The y-scale shows the bandwidth provided on 100Mbps line. When using our protocol, we get around 92.2 Mbps while we get around 70.0 Mbps when using TCP/IP. Also, when we use QBUF, for large size messages the bandwidth is higher than when we dont use the QBUF. This is because for small messages , we would be locking pages and that is costly when message size is small. But for large size messages, since copying would take time, our transfer of ownership outperforms traditional copy operations.

The Fig 5 shows that our protocol **QMP** is not meant for small sized messages. Here we have varied the message size from 100 Bytes to 1000 Bytes and we find that performance of QMP is considerably bad for low sized messages when comparing with TCP/IP. But as message size is increased the performance of QMP outperforms that of TCP/IP.

Next Fig 6 shows the performance of QMP over message size range of 100 Bytes to 100,000 Bytes. The graph shows the performance in logarithmic scale. As we can see the performance of QMP initially for small sized messages is low, but it improves as the message size increases and finally settles down at around 91.1 Mbps. This is considerable improvement over TCP/IP.

5.2 Latency

We measured the latency provided by our protocol QMP as well as TCP/IP. For measuring latency of QMP , we

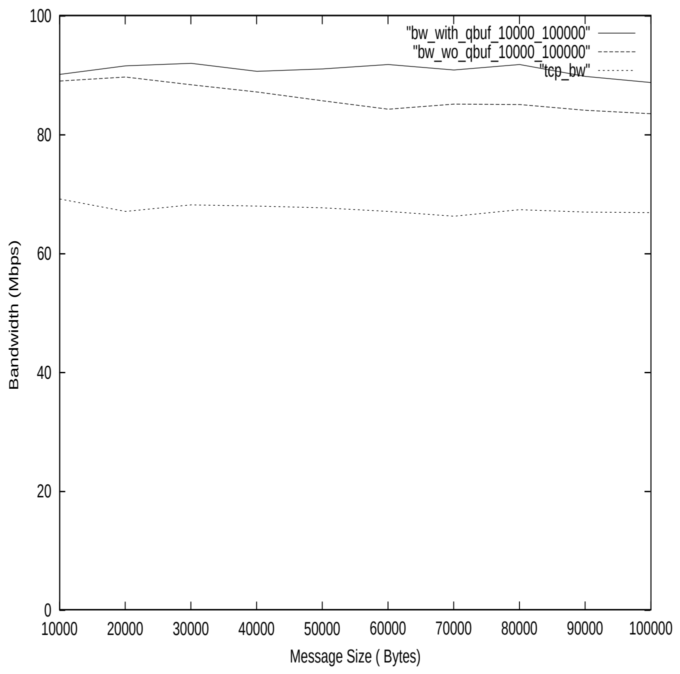

Figure 4. Comparing Bandwidth provided by QMP with and without Qbuf and TCP/IP

send very small sized packets to another node and then receive same sized messages. We repeat this many times and then take the average. We found the average latency to be around 160 usecs.For TCP/IP, we used **netperf** tool for finding its latency. It showed the latency to be around 150 usecs. So, the latency provided by our protocol is comparable with TCP/IP. Again the reason for low performance than TCP/IP is that here we are sending very small sized messages and our protocol is not meant for small sized messages.

6. CONCLUSION

In conclusion, the QMP protocol proves to be an efficient protocol for low level communication. It coexists with the TCP/IP stack of communication giving the user an alternative to choose from. In applications where error constraints are not so high and where the cluster environment assumptions hold firm[1], the QMP proves a much better alternative to TCP. Though this paper focusses on the communication, we intend to continue to work towards building a system (like [12]) that integrates it into a cluster computing environemnt.

ACKNOWLEDGEMENTS

We would like to thank the lab staff of Network Computing Lab, at CSE Department, IIT Khargapur for their sup-

[1]see Preliminaries section

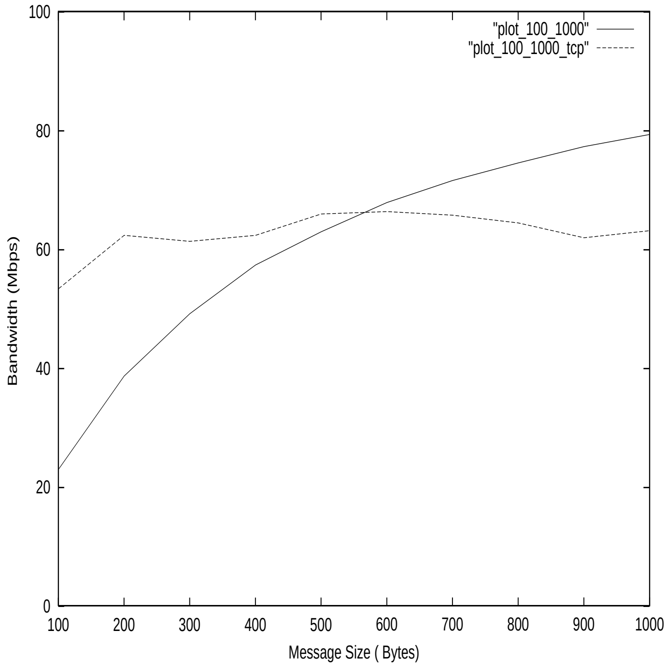

Figure 5. Comparing performance of QMP and TCP/IP for Message size from 100Byte to 1000 Bytes

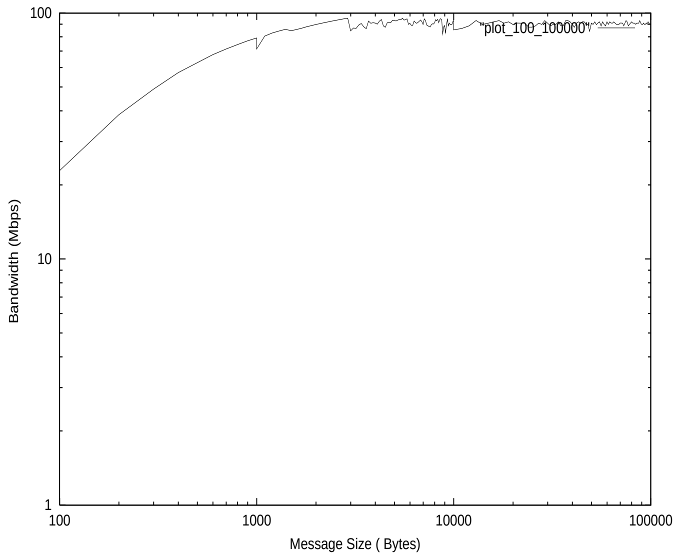

Figure 6. The performance of QMP in log scale

port. We would also like to thank D. Naidu and Venkat Ramisetty for their help in design and implementation of our protocol. Funding from the Ministry of Human Resouces Developement, Govt. of India, is also acknowledged.

REFERENCES

[1] Rajkumar Buyya, *High Performance Cluster Computing: Vol. 1*, (New Jersey: Prentice Hall, 1999).

[2] Lok Tin Liu, Alan Manwaring and Chad Yashikawa, Building TCP/IP Active Messages, Technical Report, Department of Computer Science, University of California, Berkeley. November, 1994.

[3] Moti N. Thadani and Yousef A. Khalidi An Efficient Zero-Copy I/O Framework for UNIX, Technical Report, Sun Microsystems, May 1995. Available at **http://www.sunlabs.com/technical-reports/1995/smli_tr-95-39.ps**

[4] Xpress Transfer Protocol Specification, Version 4.0, March 1995. Available at **http://www.sandia.gov/xtp/**

[5] Thorsten Eicken, Anindya Basu, Vineet Buch and Werner Vogels, U-Net: A User level Network Interface for Parallel and Distributed Computing, *Proc. of 5th ACM Symposium on Operating System Priniciples(SOSP)*,Copper Mountain, USA, Dec. 1995, 40–53.

[6] Thomas E. Anderson, David E. Culler and David A. Patterson, A Case for Network of Workstations:NOW, *IEEE Micro*, 15(1), 1995, 54–64.

[7] G. Ciaccio, Optimal Communication Performance on Fast Ethernet with GAMMA, *Proc. of PC-NOW'98*, Orlando, USA, March 1998, LNCS 1388, Springer-Verlag, 534–547.

[8] Andrew Gallatin, Jeff Chase and Ken Yocum, Trapeze/IP:TCP/IP at near Gigabit speeds, *USENIX Technical Conference*, June 1999.

[9] Richard P Martin, David E. Culler and Thomas E. Anderson, Effects of Communication Latency,Overhead and, Bandwidth in Cluster Architecture, *Proceeding of 24th International Symposium on Computer Architecture*, Denver, USA, June 1997, 85–97.

[10] N. C. Hutchinson and L. L. Peterson. The x-Kernel: An architecture for implementing network protocols,*IEEE Transactions on Software Engineering*, 17(1), 1991, 64–76.

[11] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility, *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles(SOSP)*, Dec. 1993. 189-202.

[12] K. Birman and R. van Renesse, *Reliable Distributed Computing with the Isis Toolkit*, (Los Alamitos, CA : IEEE Computer Society Press, 1994).