Design and Implementation of Reliable, Efficient Communication Protocol for Cluster of Workstations

Thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology (Honours)

in

Computer Science and Engineering

by

Atul Singh and **Shuvabrata Ganguly**

Under the Guidance of

Dr. R. Badrinath

Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur

May 2001

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 bottleneck for faster communication are the protocol overheads namely

- *Data touching operations :* data copying and computing checksums.
- *Per packet interrupt overhead.*
- *VM (Virtual Memory) overheads :* due to allocation and deallocation of buffers.
- *Software latency :* due to the layers of protocol code the data has to undergo in the tcp protocol stack.

The primary objective of the QMP (Quick Message Protocol) is to provide network support tailor-made for the cluster environment. Each of the above overhead issues was adressed in some form or the other in the QMP protocol. We evaluated QMP's performance by running distributed benchmarks using TCP and then using the QMP protocol suite. The QMP coexists with the Tcp/Ip protocol stack. All results were obtained on 233 MHz Intel Pentium IIs running Red Hat Linux 6.2 with Compex 10/100 Mbps compatible network adapters.

CERTIFICATE

This is to certify that the thesis titled **Quick Message Protocol (QMP) for cluster computation** submitted by **Atul Singh and Shuvabrata Ganguly**, to the Department of Computer Science and Engineering, in partial fulfillment for the award of the degree of *Bachelor of Technology* is a bonafide record of work carried out by them under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Dr. R. Badrinath
Dept. Of Computer Sc. and Engg.
Indian Institute of Technology,
Kharagpur — 721 302, INDIA.

Dated : May, 2000.

ACKNOWLEDGEMENT

We are greatly indebted to our supervisor **Dr. R. Badrinath** for his invaluable guidance and encouragement throughout the course of this work. We would also like to thank **Mr. Naidu and Mr. Venkat** for being a constant help to us in the design, analysis and experimentation of our designed protocol, and all our lab staff for making the required resources available on time.

Atul Singh and **Shuvabrata Ganguly**
Dept. of Computer Sc. and Engg.
Indian Institute of Technology
Kharagpur- 721 302 , India.

Contents

List of Figures

Chapter 1

Motivation

Computing and storage over distributed environments such as clusters of workstations 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. However, currently those computing, storage and communication environments are not widely integrated to support high performance distributed applications mainly due to the relatively low performance of inter-computer communications. 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 tranfer 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.

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[4]. 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. While this may be justifiable for a few applications such as video playback, most applications use relatively small messages and rely heavily on quick round-trip requests and replies. 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.

The **latency of communication** is mainly composed of processing overhead and network latency (time-of-flight). The term processing overhead is used here to refer to the time spent by the processor in handling messages at the sending and receiving ends. This may include buffer management, message copies, checksumming, flow-control handling, interrupt overhead, as well as controlling the network interface. Separating this overhead from the network latency distinguishes the costs stemming from the network fabric technology from those due to the networking software layers. Recent advances in network fabric technology have dramatically improved network bandwidth while the processing overheads have not been affected nearly as much. The effect is that for large messages, the end-to-end latency i.e., the time from the source application executing "send" to the time the destination application receiving the message, is dominated by the transmission time and thus the new networks offer a net improvement. For small messages in local area communication, however, the processing overheads dominate and the improvement in transmission time is less significant in comparison. In wide area networks the speed of light eventually becomes the dominant latency component and while reducing the overhead does not significantly affect latency it may well improve throughput.

This project is about building kernel and user level support for using a cluster of workstations to act as a distributed supercomputer[6]. The QMP protocol is aimed as reducing most of the processing overheads which stifle fast communication. In particular it emphasises on

- *Reducing user-kernel data copy.*
- *Doing away with the software checksumming overhead.*
- *Slashing the software latency involved.*
- *Implementing an optimistic flow control mechanism.*
- *Zero Copy send.*

Because of the volume production, commercial workstations today offer much better price/performance than the individual nodes of Massively Parallel Processing Architectures (MPP's). In addition, switch-based networks such as ATM and Gigabit ethernet will provide cheap, high-bandwidth communication. This price/performance advantage is increased if the cluster of workstations can be used for both the tasks traditionally run on workstations and large distributed programs.

Chapter 2

Our Approach

After conducting a study of the various contemporary projects on cluster computing and after studying the previous work done, we felt that nothing more substantial could be done by trying to modify the existing TCP/IP. We decided that our protocol would co-exist with the existing TCP/IP protocol suite but would be completely different in the sense that it would be solely suited to cluster computing. Our main objective was to reduce the different software latencies involved at the various levels. With this aim, we set about the design of our new protocol.

Our first step (*see figure 2.1*) was the study of the linux kernel and the implementation of TCP/IP in it and then that of the previous work which aimed at tailoring the existing protocol suite. After gaining an understanding of the various intricacies involved, we began the design of our *QMP* protocol. After that we planned to integrate with PVM and name it QPVM.

The choice of Linux over other operating systems had its very clear advantages. Firstly, the availability of kernel source was a must to gain an understanding of the different network modules and their implementation and thus to model our new protocol. Linux code is an open source distribution and hence was our best choice. Linux is itself used to support the networking needs of various of organizations. Also, the increasing widespread use of Linux would help port different applications between Linux and other Unix platforms.

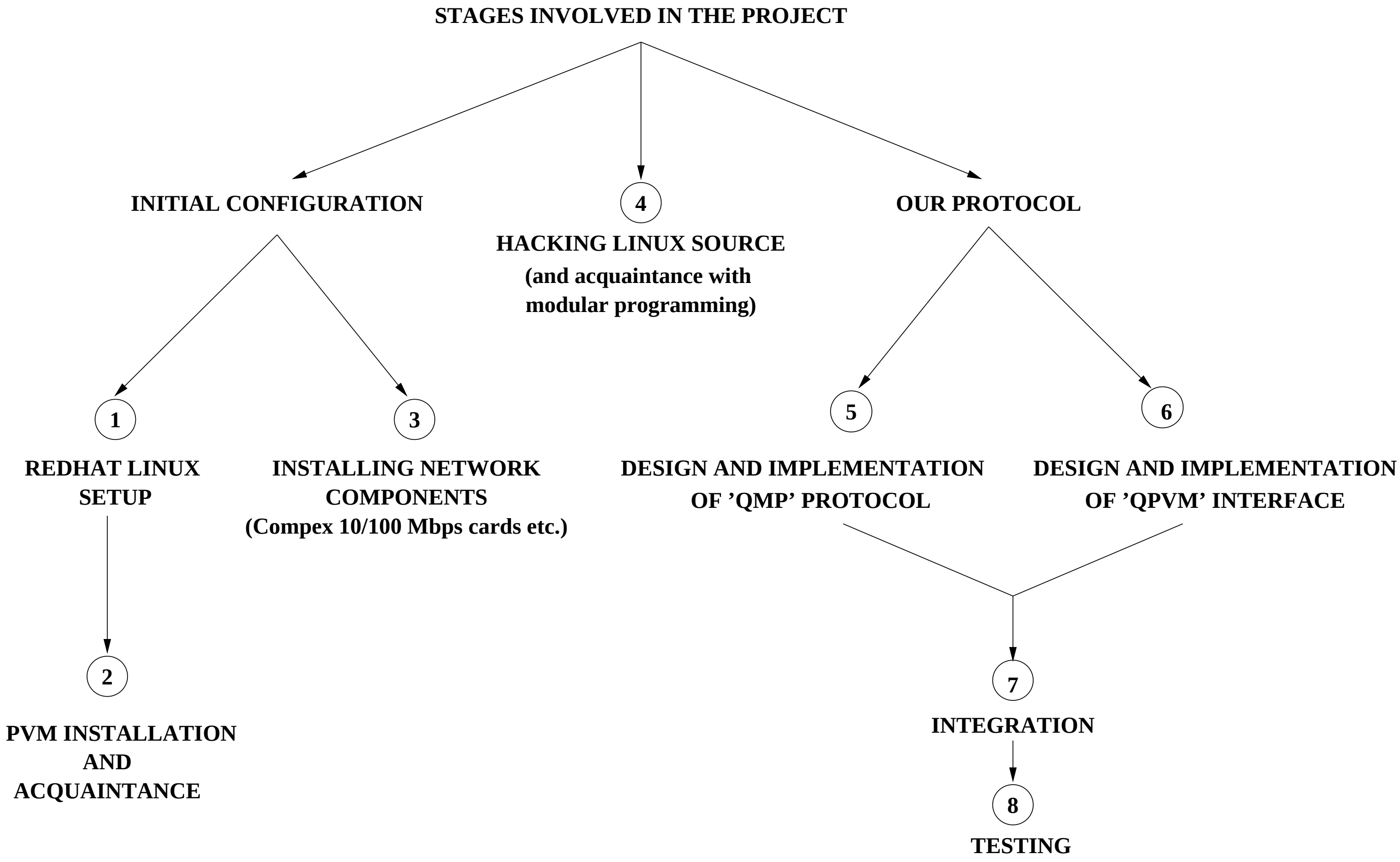

Figure 2.1: The different stages of the QMP project.

2.1 Module-based programming

Our approach to the implementation of the new protocol was different from that of our predecessors. We have used **Module-based programming** which is a very useful and efficient feature of Linux kernel programming. While a normal application performs a single task from beginning to end, a module registers itself with the kernel to serve future requests. We can load and unload the module dynamically using the *insmod* and *rmmod* command. The task of the *init_module* is to prepare for the invocation of the kernel's functions whenever it is loaded. Similarly, *cleanup_module* gets invoked when the module is unloaded.

An application can call functions that it doesn't define: the linking stage resolves external references using the appropriate library functions. A module, on the other hand, is linked only to the kernel and the only ones it can call are the ones exported by the kernel. The usual header files should never be included in the source files of the modules, since no library is linked to the modules. Thus, when the module is dynamically loaded, all references to the kernel functionalities are resolved i.e.., the module can call the kernel functions. In turn, the services of the module are available to the kernel. This resolving of the services between the module and the kernel constitutes the *registering* of the module with the kernel.

When the module is unloaded, all the references between the module and the kernel are unlinked. This constitutes the *unregistering* of the module. (*See figure 2.2*).

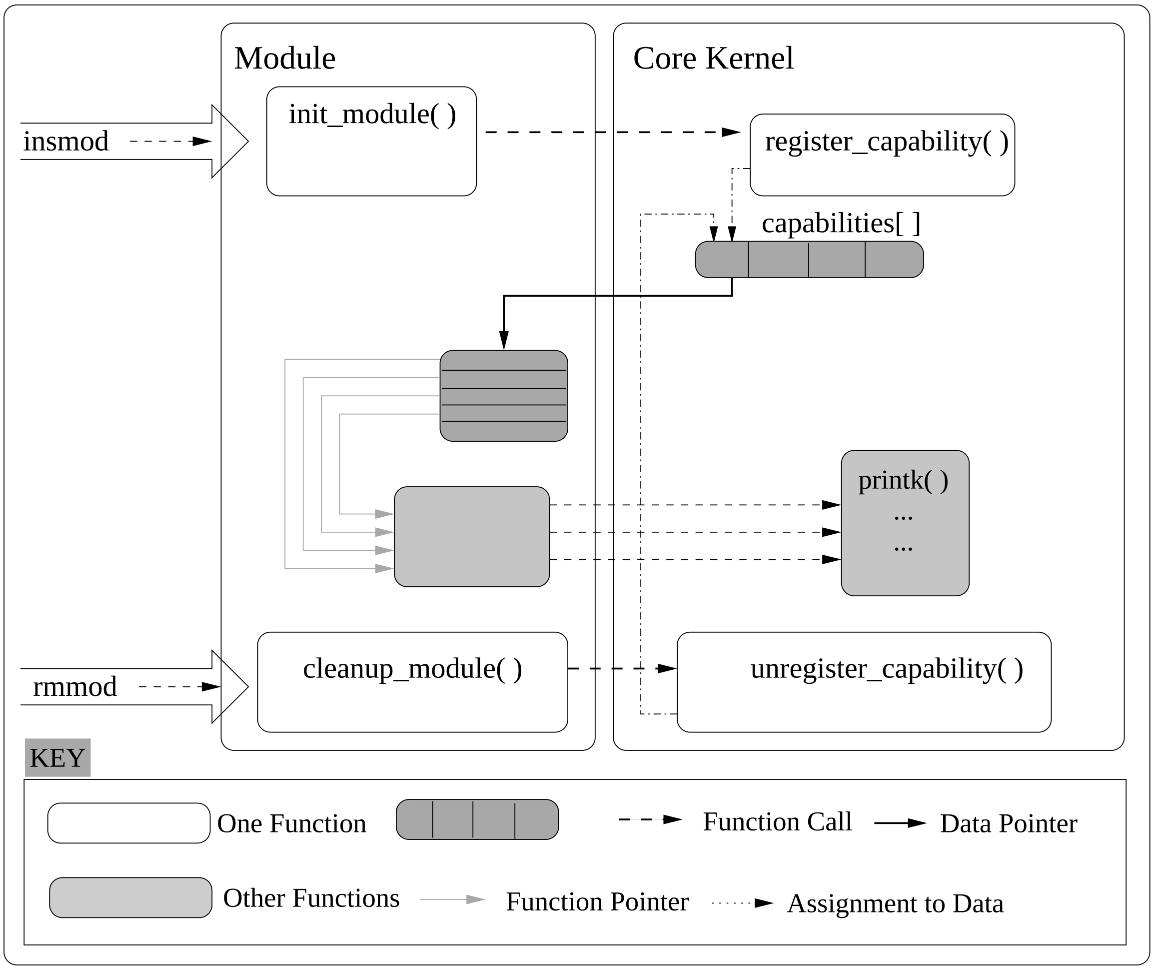

Figure 2.2: How modules are dynamically linked.

The advantages of **module-based programming** are:

- *Easier:* It is a lot easier to install and uninstall the module dynamically with just a command than the cumbersome frequent compilation and re-compilation of the whole kernel during the development stage.
- *Faster:* It is faster because it helps cutdown the development time by not going through the lengthy shutdown/reboot cycle everytime one has to test the code.
- *Safer:* Using modules enables us to write our own functionalities and leave the kernel code, atleast directly, unhampered. On the other hand, previously, a backup of the whole kernel had to be maintained since the changes were made directly to the kernel code and any serious error would mean a tampered kernel.

In the following chapters, the work done during the course of the project is presented in the order it was carried out.

Chapter 3

Linux communication layers and the QMP design

One of the main objectives of the project was that the QMP protocol stack would co-exist with the TCP/IP protocol stack. Figure 3.1 shows this coexistence of the two protocol suites. Any TCP packet sent from normal PVM applications would follow the path of the left suite at the snder end and similarly at the receiving end. Analogously, a QMP packet will follow the path of the right suite. Let us understand and analyse the detailed procedure of sending and receiving in both these suites.We have also provided the user-level socket interface which a user can use and directly open a socket using **QMP**. We provide reliable, stream oriented data delivery service. We start with understanding the TCP/IP suite and follow that with the detailed design of QMP. Much of the information in the next section is a summary of communication in the LINUX layers as given in the tlk (tour of the LINUX kernel).

3.1 Understanding Linux Networking layers

Just like the network protocols themselves, Figure 3.1 shows that Linux implements the internet protocol address family as a series of connected layers of software. BSD sockets are supported by a generic socket management software concerned only with BSD sockets. Supporting this is the INET socket layer, this manages the communication end points for the IP based protocols TCP and UDP. UDP(User Datagram Protocol) is a conectionless protocol while TCP(Tranmission Control Protocol) is a reliable end to end protocol. When UDP packets are transmitted, Linux neither knows nor cares if they arrive safely at their

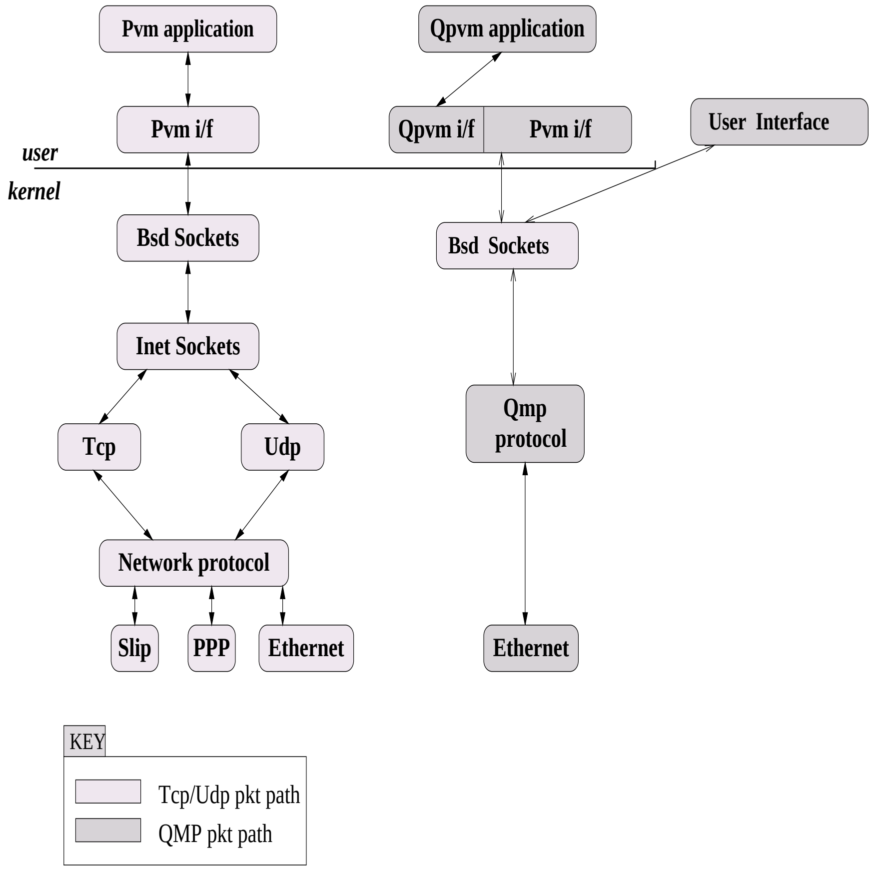

Figure 3.1: Coexisting Linux Networking and QMP Layers

destination. TCP packets are numbered and both ends of the TCP connection make sure that transmitted data is received correctly. The IP layer contains code implementing the Internet Protocol. This code appends IP headers to transmitted data and understands how to route incoming IP packets to either the TCP or UDP layers. Underneath the IP layer, supporting all of Linux's networking are the network devices, for example PPP and ethernet. Network devices do not always represent physical devices; some like the loopback device are purely software devices. Unlike standard Linux devices that are created via the **mknod** command, network devices appear only if the underlying software has found and initialised them.

3.1.1 The BSD Socket Interface

This is a general interface which not only supports various forms of networking but is also an inter-process communicaton mechanism. A socket describes one end of a communications link, two communicating processes would each have a socket describing their end of the communication link between them. Linux supports several classes of socket and they these are known as address families. This is because each class has its own method of addressing its communications. Linux supports the following socket address families or domains:

UNIX	Unix domain sockets
INET	The INET family supports communications via TCP/IP protocols
AX25	Amateur Radio X25
IPX	Novell IPX
APPLETALK	Appletalk DDP
X25	X25

There are several socket types and these represent the type of service that supports the connection. Not all address families support all types of service. Linux BSD sockets support a number of socket types:

Stream These sockets provide reliable two way sequenced data streams with a guarantee that data cannot be lost, corrupted or duplicated in transit. Stream sockets are supported by the TCP protocol of the INET family.

Datagram These sockets also provide two way data transfer but, unlike stream sockets, there is no guarantee that the messages will arrive. Even if they do arrive there is no guarantee that they will arrive in order or even not be duplicated or corrupted. This type of socket is supported by the UDP protocol of the INET address family.

Raw This allows process direct (hence "raw") access to the underlying protocols. It is, for example, possible to open a raw socket to an ethernet device and see raw IP data traffic.

Reliable Delivered Messages These are very like datagram sockets but the data is guaranteed to arrive.

Sequenced Packets These are like stream sockets except that the data packet sizes are fixed.

Packet This is not a standard BSD socket type, it is a Linux specific extension that allows processes to access packets directly at the device level.

Processes that communicate using sockets use a client server model. A server provides a service and clients make use of that service. A server using sockets, first creates a socket and then binds a name to it. The format of this name is dependent on the socket's address family and it is, in effect, the local address of the server. The socket's name and address is specified using the **sockaddr** data structure. An INET socket would have an IP address bound to it. Having bound an address to the socket, the server then listens for incoming connection requests specifying the bound address. These incoming requests must find their way up through the various protocol layers and then wait on the server's listening socket. Once the server has received the incoming request it either accepts or rejects it.

Creating a BSD Socket

The system call to create a new socket passes identifiers for its address family, socket type and protocol. Firstly the requested address family is used to search the **pops** vector for a matching address family. It may be that a particular address family is implemented as a kernel module and, in this case, the **kerneld** module must load the module before we can continue. A new socket data structure is allocated to represent the BSD socket. Actually the socket data structure is physically part of the VFS inode data structure and allocating a socket really means allocating a VFS inode. The newly created BSD socket data structure contains a pointer to the address family specific socket routines and this is set to the **proto_ops** data structure retrieved from the **pops** vector. Its type is set to the socket type requested; one of SOCK_STREAM, SOCK_DGRAM and so on. The address family specific creation routine is called using the address kept in the **proto_ops** data structure.

Binding an Address to an Inet BSD Socket

In order to be able to listen for incoming internet connection requests, each server must create an INET BSD socket and bind its address to it. The bind operation is handled within the INET socket layer with some support from the underlying TCP and UDP protocol layers. The socket having an address bound to cannot be used for any other communication. This means that the socket state must be TCP_CLOSE. The **sockaddr** passed to the bind operation contains the IP address to be bound to and optionally a port number. Normally the IP address bound to would be one that has been assigned to the network device that supports the INET address family and whose interface is up and able to be used. We can see which network interfaces are currently active in the system by using the **ifconfig** command. The IP address may also be the IP broadcast address of either all ones or all zeroes. These are special addresses that mean "send to everybody". The IP address bound to is saved in the **sock** data structure in the **recv_addr** and **saddr** fields. These are used in hash lookups as the sending IP address respectively.

As packets are being received by the underlying network devices, they must be routed to the correct INET and BSD sockets so that they can be processed. For this reason, UDP and TCP maintain hash tables which are used to lookup the addresses within incoming IP messages and direct them to the correct **socketsock** pair. TCP maintains several hash tables. However TCP does not actually add the binding sock structure into its hash tables during the bind operations , it merely checks that the port number requested is not currently being used. The sock data structure is added to TCP hash table during the listen operation.

Making a connection on an INET BSD socket

Once a socket has been created and provided it has not been used to listen for inbound connection requests, it can be used to make outbound connection requests . For connection oriented protocols like TCP, it involves building a virtual circuit between two applications. An outbound connection can only be made on an INET BSD socket that is in the right state; that is easy to say one that does not already have a connection established and one that is not being used for listening for inbound connections. This means that the BSD socket data structure must be in state SS_UNCONNECTED. For a connect operation on a TCP BSD socket, TCP must build a TCP message containing the connection information and send it to the IP destination given. The TCP message contains information about the connection, a unique starting message sequence number, the maximum sized message that can be managed by the initiating host, the transmit and receive window size and so on. Within TCP, all messages are numbered and the initial sequence number is used as the first message number. Linux chooses a reasonably random value to avoid malicious protocol attacks. Every message transmitted by one end of the TCP connection and succesfully received by the other is acknowledged to say that it arrived successfully and uncorrupted. Unacknowledged messages should be retransmitted. The transmit and receive window size is the number of outstanding messages that there can be without an acknowledgement being sent. The maximum message size is based on the network device that is being used at the initiating end of the request. If the receiving end network device supports smaller maximum message sizes, then the connection will use the minimum of the two. The application making the outbound TCP connection request must now wait for a response from the target application to accept or reject the connection request. As the TCP sock is now expecting incoming messages, it is added to the **tcp_listening_hash** so that incoming TCP messages can be directed to this **sock** data structure. TCP also starts timers so that the outbound connection requests can be timed out if the target application does not respond to the request.

Listening to an INET BSD socket

Once a socket has had an address bound to it, it may listen for incoming connection requests specifying the bound addresses. A network application can listen on a socket without first binding an address to it; in this case the INET socket binds an unused port number (for this protocol) and automatically binds it to this socket. The listen socket function moves the socket into state TCP_LISTEN and does any network specific work needed to allow incoming connections. TCP then adds the sockets **sock** data structure into two hash tables as it is now active. These are the **tcp_bound_hash** table and the **tcp_listening_hash** . Both are indexed via hash function based on the IP port number.

Whenever an incoming TCP connection request is received for an active listening socket, TCP builds a new **sock** data structure to represent it. This **sock** data structure will become the bottom half of the TCP connection when it is eventually accepted. It also clones the incoming **sk_buffs** containing the connection request and queues it onto the receive queue for the listening sock data structure, The cloned **sk_buff** contains a pointer to the newly created **sock** data structure.

Accepting Connection Requests

An accept operation on a listening socket causes a new **socket** data structure to be cloned from the original listening **socket**. The accept operation is then passed to the supporting layer, in this case INET, to accept any incoming connection request. The INET protocol layer will fail the accept operation if the underlying protocol, say UDP, does not support connections. Otherwise the accept operation is passed to the real protocol, in this case TCP. Once a connection request has been received, the **sk_buff**, containing the request is discarded and the **sock** data structure is returned to the INET socket layer where it is linked to the new socket data structure created earlier.

3.1.2 The IP Implementation in Linux

Socket Buffers

One of the problems of having many layers of network protocols, each one using the services of another, is that each protocol needs to add protocol headers and tails to data as it is transmitted and to remove them as it processes received data. This makes passing data buffers between the protocols difficult as each layer needs to find where its particular protocol headers and tails are. One solution is to copy buffers at each layer but that would be inefficient. Hence, Linux uses socket buffers or **sk_buffs** to pass data between the protocol layers and the network device drivers. **sk_buffs** contain pointer and length fields that allow each protocol layer to manipulate the application data via standard functions or "methods".

Figure 3.2 shows the **sk_buff** data structure; each **sk_buff** has a block of data associated with it. The **sk_buff** has four data pointers, which are used to manipulate and manage the socket buffer's data:

head points to the start of the data area in memory. This is fixed when the **sk_buff** and its associated data block is allocated.

data points to the current start of the protocol data. This pointer varies depending

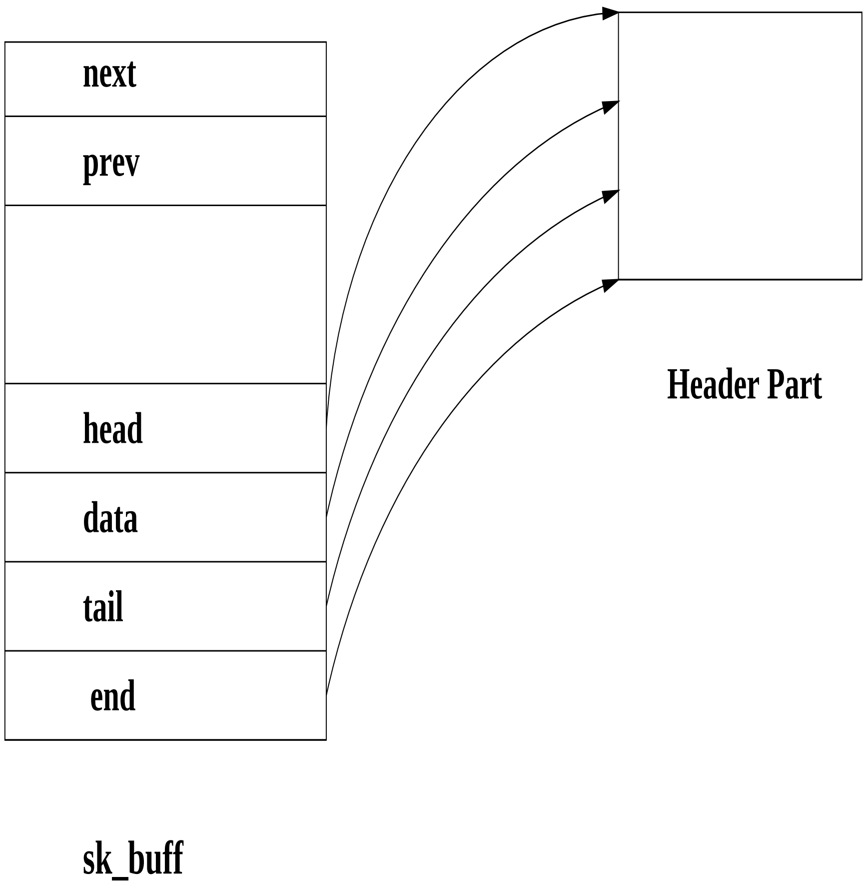

Figure 3.2: The Socket Buffer (sk_buff)

on the protocol layer that currently owns the **sk_buff**.

tail points to the current end of the protocol data. Again this pointer varies depending on the owning protocol layer.

end points to the end of the data area in memory. This is fixed when the **sk_buff** is allocated.

There are two length fields **len** and **truesize**, which describe the length of the current protocol packet and the total size of the data buffer respectively. The **sk_buff** handling code provides standard mechanisms for adding and removing protocol headers and tails to the application data. They safely manipulate data, **tail** and **len** fields in the **sk_buff** :

skb_push This moves the **data** pointer towards the start of the data area and increments the **len** field. This is used when adding data or protocol headers to the start of the data being transmitted.

skb_pull This moves the **data** pointer away from the start, towards the end of the data area and decrements the **len** field. This is used when removing data or protocol headers from the start of the data that had been received.

skb_put This moves the **tail** pointer towards the end of the data area and increments the **len** field. This is used when adding data or protocol information to the end of the data to be transmitted.

skb_trim This moves the **tail** pointer towards the start of the data area and decrements the **len** field. This is used when removing data or protocol tails from the received packet.

Receiving IP packets

There is a series of **device** data structures linked together in the **dev_base** list. Each **device** data structure describes its device and provides a set of call back routines that the network protocol layers call when they need the network driver to perform work. These functions are mostly concerned with transmitting data and with the network devices addresses. When a network device receives packets from its network, it must convert the received data into **sk_buffs** data structures. These received **sk_buffs** are added onto the **backlog** queue by the network drivers as they are received. If the **backlog** queue grows too large then the received **sk_buffs** are discarded. The network bottom half is flagged as ready to run as there is work to do.

Sending IP packets

Packets are transmitted by applications exchanging data or else they are generated by the network protocols as they support established connections or connections being established. Whichever way the data is generated, an **sk_buff** is built to contain the data and various headers are added by the protocol layers as it passes through them. The sk_buff, needs to be passed to a network device to be transmitted. First though the protocol, for example IP, needs to decide which network device to use. This depends on the best route for the packet.

Data Fragmentation

Every network device has a maximum packet size and it cannot transmit or receive data packets bigger than this. The IP protocol allows for this and will fragment data into smaller units to it into the packet size that the network device can handle. The IP protocol header includes a fragment field which contains a flag and a fragment offset.

When an IP packet is ready to be transmitted, IP finds the network device to send the IP packet out on. This device is found from the IP routing tables. Each device has a field describing its maximum transfer unit (in bytes), this is the **mtu** field. If the devices' **mtu** is smaller than the packet size of the IP packet that is waiting to be transmitted, then the IP packet must be broken down into smaller (mtu sized) fragments. Each fragment is represented by an **sk_buff**; its IP header marked to show that it is a fragment and what offset into the data this IP packet contains. The last packet is marked as being the last IP fragment. If, during the fragmentation, IP cannot allocate an **sk_buff**, the transmit will fail.

Receiving IP fragments is a little more difficult than sending them because the IP fragments can be received in any order and they must all be received before they can be reassembled. Each time an IP fragment is received, it is checked to see if it is an IP fragment. The first time that the fragment of a message is received, IP creates a new **ipq** data structure and this is linked to the **ipqueue** list of IP fragments awaiting recombination. As more IP fragments are received, the correct **ipq** data structure is found and a new **ipfrag** data structure is created to describe this fragment. Each **ipq** data structure uniquely describes a fragmented IP received frame with its source and destination IP addresses, the upper layer protocol identifier and the identifier for this IP frame. When all of the fragments have been received, they are combined into a single **sk_buff** and passed up to the next protocol level to be processed. Each **ipq** contains a timer that is restarted each time a valid fragment is received. If this timer expires, the **ipq** data structure and its **ipfrags** are dismantled and the message is presumed to have been lost in transit. It is then upto the higher level protocols to retransmit the message.

3.2 Brief Overview of QMP

Now that we have understood the works of how a TCP/IP packet is sent and received, let us discuss the Quick Message Protocol (QMP) design in detail. 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

3.2.1 Co-existing with the TCP/IP suite

We achieved this objective by doing **module-based programing** while writing **kernel-level code**. Registering the QMP protocol was simply achieved by adding a **packet_type** structure to the **p_type hash table** of the kernel. This packet type corresponds to QMP packets and therefore, has information about the handler routine (**qmp_rcv( )**) to be called when a QMP packet is encountered at the receive end. The above code was written in the **init_module** and so QMP is *registered* with the kernel as soon as the **insmod** (insert module)[1] command is executed at the prompt. Figure 3.1 describes the different paths a TCP/UDP packet and a QMP packet take.

3.2.2 Reducing protocol processing overheads

The more important objective was to enhance the protocol performance for a cluster environment. We have already discussed our assumptions for a cluster environment[2]. We have

[1]for registering details see **chapter 2 (Our Approach)**
[2]see Chapter 2. Motivation.

also extensively discussed the overheads involved in typical TCP/IP data processing[3]. The salient features of QMP to reduce processing overheads are as follows :

- **Software checksum removal :**

  Research suggests that **data touching** operations are the most costly ones. Checksum computation is one of the data touching operations. However, as discussed earlier, the error probability in a cluster environment is so low that the *hardware CRC checksum*[4] is sufficient enough. Hence, the QMP protocol does not perform any software checksum as opposed to TCP.

- **Reducing data copy :**

  Data copying is the other part of **data touch** operations. From the user level down to the physical media, the data undergoes many copying operations (eg. User-kernel data copy, copying from the kernel to the network, etc.).

  Another copy operation is done at the application level of the PVM (Parallel Virtual Machine) software. When a buffer has to be sent to another task from a source PVM task, it must be first *packed* into an *active buffer* and then sent. This leads to an extra copy at the user level itself. The QMP protocol circumvents this problem. The basic **qmp_send( )** system call takes an iovec array as parameter instead of the user buffer itself. The iovec array is simply an array of base addresses and lengths of discontiguous user buffer locations. So now the QPVM interface does not need to pack the user buffer into an **active buffer** but just its base address and length in an iovec array and pass it to **qmp_send()**.

- **Slashing software latency :**

  Another disadvantage of the TCP/IP protcol stack (for cluster applications) is that there is a lot of *software latency* encountered when the data packet percolates down the layer, and each layer adds its own header and tail (when the packet is being sent) or strips them (when the packet is being received).

  The QMP protocol collapses all the network layers into one common QMP layer thus needing to handle only one header, the *qmp_header*. The software latency for sending and receiving is therefore slashed.

[3]see Chapter 1 (**Motivation**) for details

[4]Cyclic Redundancy Check

- **Zero copy send :**

  The second important aspect of our protocol is zero copy send. We have introduced the concept of kernel-user sharable buffers ,**qbuf**. Now, when a user sends a buffer through a new send call, which we call qb_sendmsg(), we use these sharable buffers . But to use our**qbuf** memory scheme, the user program has to first register with it by calling **qbuf_init()** . For buffer allocation, it should use **qbuf_alloc()** in place of malloc(). We use a bitmap to identify which buffers are free and which are occupied now.Now, when the protocol recieves a ack, it frees the sharable buffer used by user.

  We will discuss qbuf in detail in chapter 6.

Chapter 4

QMP Design

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.

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 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 lenght 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*. The QMP communication model is shown in next figure 4.1.

A context at an end-sytem 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 teh establishment of an association. The context handling this user moves from a quiescent state to an active state, where it constructs a

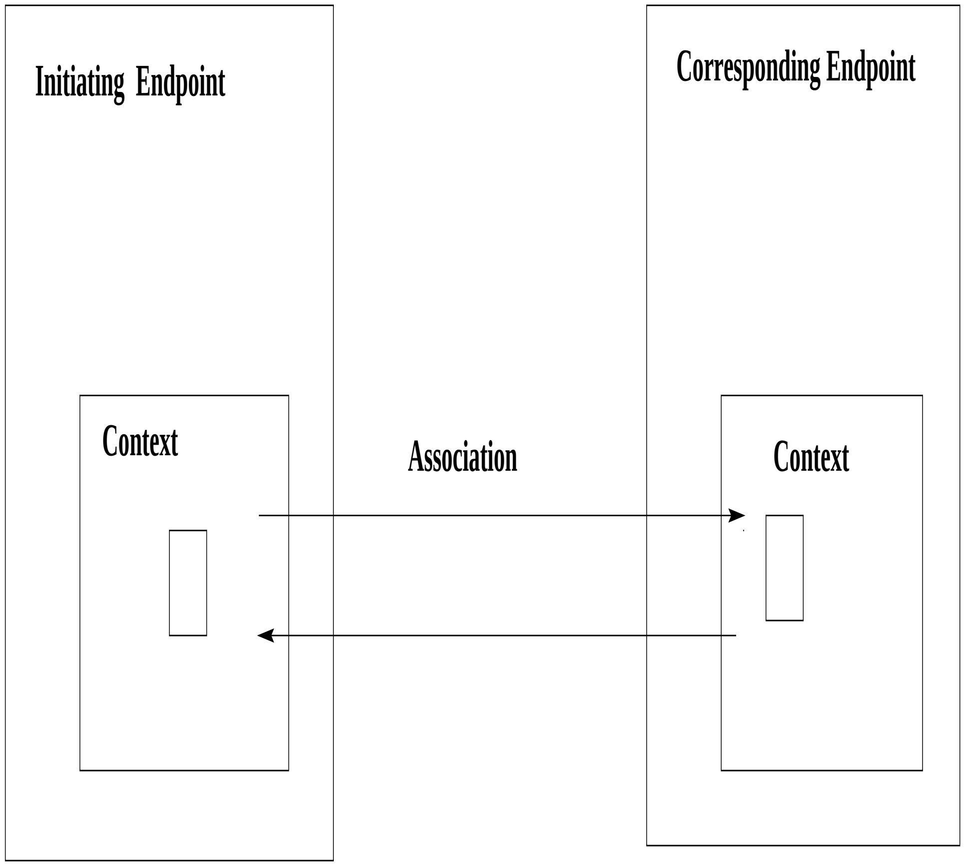

Figure 4.1: QMP Communication Model

FIRST packet with explicit addressing and service information obtained fromthe 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.

Now, after a little overview on the protocol, we will modularly discuss the various aspects of QMP. This chapter is divided into following subsections :

- Header Format
- Different Types of Packets
- Unicasting in QMP
- qbuf

4.1 Header Format

A packet is the basic unit for information exchange between the end points of an association. The fields within a packet hold the information pertaining either to the state of the association or to the data being transferred. The layout, or structure, of the packet facilitates retrieval of the information held within; there is no inherent structure to the packet once it is handed off to the underlying data delivery service. The receiving end point parses the packet only by knowing the packet structures a priori.

Each QMP packet carries a common header structure followed by a payload seg-ment, as shown in Figure 2-1. There are two basic types of packets in QMP, control packets and information packets. The control packets carry a Control Segment and are used to exchange protocol state information between contexts in an association. The information in control packets is not given directly to the user, but is used by the context to effect the control algorithms. User information, including user data and protocol diagnostic messages, are carried in the Information Segment of information packets.

All QMP packets use a fixed header syntax consisting of the following fields: key,cmd (command), dlen (data length), check (checksum), sync (synchronizing number), and seq (sequence number). The key field steers the packet to the proper destination context. The cmd field dictates how the packet is to be processed. The dlen and seq fields identify this packet' s contents with respect to the data stream. The check and sync fields are used to determine the validity of the packet, and the sort field orders the act of parsing the packet among all contending activities. The header and its fields are shown in Figure 2-2.

4.1.1 Key Field

The key field associates a packet with a context. It is similar to the connection ortransaction identifier found in most protocols. The lifetime of a key value is the same as the lifetime of the associated context: a key is considered "active" while its context is in anystate but quiescence, and "inactive" when its context is quiescent. The key field is always interpreted and must contain a meaningful value in all packet types.The key field is 32 bits wide.

When a context is instantiated, a key value is assigned to that context and the 32-bit value is unique within that host's QMP implementation. When the context sends a packet, it fills in the key field of the packet header with its key value so that the receiver of the packet can, along with other information, determine which context sent the packet and infer, therefore, which context is to receive the packet (see section ,"Full Context Lookup"). The receiving context notes the packet's key value anduses it, with the

RTN flag set, in all of its outgoing packets. Such a packet, when received,can be mapped directly to the appropriate context because that key value was generated for that context, and is therefore unique within the host (see Section 4.3.2, "Abbreviated Context Lookup"). This is illustrated in Figure 4.2, where K is the key value generated at Context A. The packets sent from Context B use the return key value K'.

4.1.2 Command Field

The 16-bit cmd field carries the options set for this packet, the packet type, and the format of the packet. The cmd field must always contain meaningful values for all its subfields and bits.

- Command Options

  The bitflags in the options field are given in Table 4.1. These bits select QMP operating modes and mechanisms. The logic convention is positive: a function is enabled if the corresponding bit is set (value is 1), it is disabled if the bit is cleared (value is 0). Zero or more bits may be set in each header.

  MULTI : When set, this bit indicates use of multicast mode (see Section 5, "Multicast Functional Specification"). The value of this bit must be the same in all packets over the lifetime of the association.

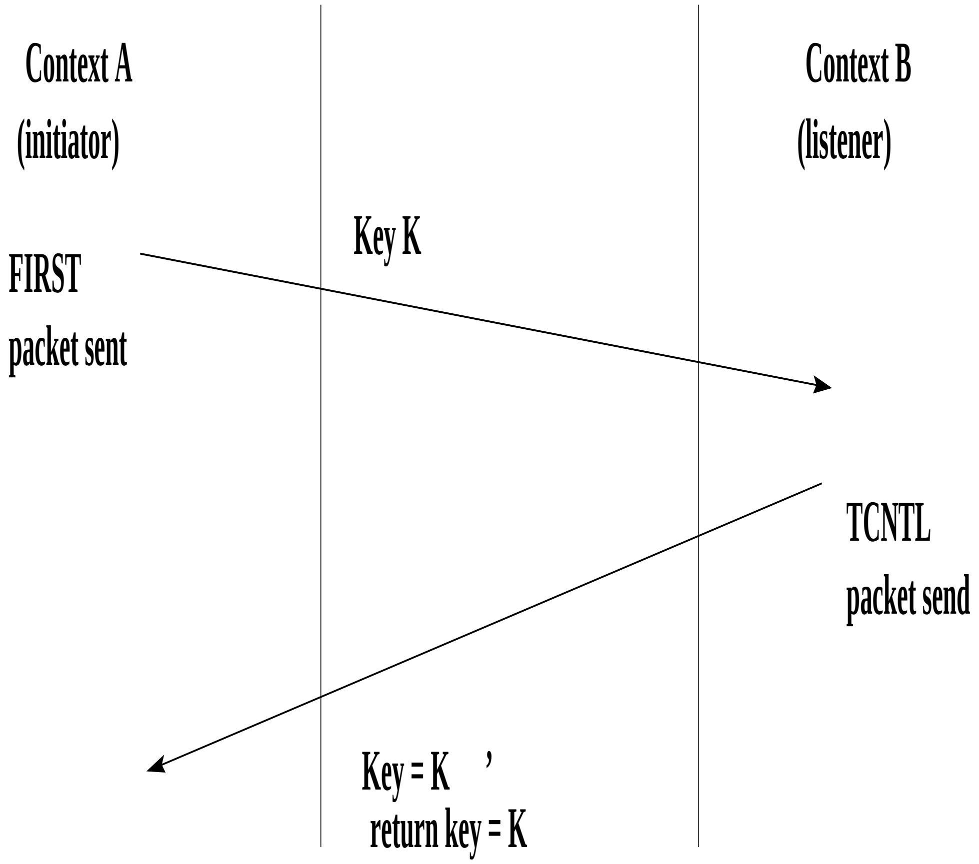

Figure 4.2: Key Exchange

SREQ : When set, the receiver must respond immediately with a control packet. The SREQ bit is set by an QMP sender according to its output acknowledgement policy or when responses are needed to recover from errors.

DREQ : When set in a data-bearing packet, the receiver must send a control packet after all enqueued data, up to and including any in this packet, have been delivered to the higher-layer application. When set in a control packet, this bit indicates that the sender requests that the receiver send a control packet after all data with sequence number less than the value in the seq field of this packet have been delivered to the higher-layer application.

WCLOSE and RCLOSE : These bits are the basis for disconnect handshakes carried out by the close state machines (see Section 4.2.7, "Association T ermination"). The WCLOSE bit indicates that no more data will be written to the outgoing data stream. The RCLOSE bit indicates that all bytes written to the incoming stream

Bit	Mask	Description	Potential change
RTN	0x80	Return Bit	Per association
MULTI	0x40	Multicast Mode	Per association
SREQ	0x20	Status Requested	Per packet
DREQ	0x10	Data delivery request	Per packet
RCLOSE	0x08	Reader Closed	Per packet
WCLOSE	0x04	Writer Closing	Per packet
EOM	0x02	End of Message	Per packet
END	0x01	End of Association	Once

Table 4.1 Command Options Fields

have been received with respect to the error control policy in use.

EOM : This bit is used to delimit message boundaries in a data stream; when set, this bit denotes the end of a message. A data-bearing packet containing EOM carries the trailing bytes of a message. Control packets must not carry an EOM bit. The EOM bit is not manipulated by QMP, nor does QMP create or remove EOM bits from a data stream. The bit is asserted by a sending application through its service interface and delivered to the receiving application by the receiving QMP.

END : When set, this bit indicates that the sending context is being released. It is used in the last packet of a closing handshake and also when an association is aborted (seeSection 4.3.2, "Association Termination").

- Packet Type Field

The least-significant byte of the cmd field is called the ptype field. Within this byte,bits 0 through 4, called the pformat field, identify the XTP packet type. Bits 5 through 7 indicate the XTP version (ver field). The pformat field is always interpreted and must contain a meaningful value. The ver field is also always interpreted and, in QMP, must be set to binary 001.

The pformat values are enumerated in Table 2-2. FIRST, DATA, JOIN, and DIAG packets use an Information Segment. The FIRST (Section 4.2.1) packet is the initial packet of an association and contains an Address Segment , a Traffic Specifier (Section 4.2.4), and optionally a Data Segment . DATA packets (Section 4.2.2) are used for subsequent data transfers, and contain only the Data Segment. These are the two packet types responsible for user data transfer.

CNTL (Section 4.2.3), ECNTL (Section 4.2.4), and TCNTL (Section 4.2.5) packets use a Control Segment. The CNTL packet conveys control information such as flow control window values through the Common Control segment. The ECNTL packet additionally conveys error control information through its Error Control segment . The TCNTL packet is used to exchange keys during connection establishment. These are the three packet types responsible for state information exchanges between contexts.

4.1.3 Data Length Field

The dlen field specifies the number of bytes immediately following the header, thatis, the number of bytes in the payload segment. Since zero-length DATA packets are not allowed, the dlen field will always have a non-zero value.

4.1.4 Synchronizing Handshake Field

The 16-bit sync field provides the basis for the synchronizing handshake (see Sections 4.6.3 an 4.2). Each context keeps a record of the last sent sync value in a variable saved_sync. When a packet is sent with its SREQ bit set, the saved_sync value is incremented by one, and this new saved_sync value is placed into the sync field of the outgoing packet. The rules for a sender are as follows.

1. The value of the sync field in outgoing packets increases only when the SREQbit is set in an outgoing packet, and remains constant at the most recently transmitted value for all other packets.

2. Retransmitted packets must also abide by Rule 1; the sync values used are not the ones originally transmitted, but values generated under Rule 1.

An QMP receiver saves the highest received sync values into a local variable rcvd_sync.

The value of rcvd_sync is later placed into the echo field of each outgoing control packet. The rules for a receiver are as follows:

1. The local context variable rcvd_sync is initialized with the value of the sync field from the first packet received for this association.

2. The rcvd_sync value is placed into the echo field of all outgoing control packets.

When a control packet is received, the sync field from the control packet is compared to the value in rcvd_sync; if the control packet contains a sync value that is greater than or equal to the rcvd_sync value, the control packet is processed normally. If not, the control packet has information older than information received in previous control packets. In this case, the SREQ and DREQ bits should be responded to if set, but no other processingshould be done on the control packet.

If the WTIMER expires, indicating a status request (sent SREQ bit) has not been answered, the context enters the synchronizing handshake.

4.1.5 Sequence Number Field

The seq field is an unsigned 32-bit quantity representing a sequence number for the outgoing data stream. A sequence number is associated with every byte of data in that stream. The seq field is meaningful and interpreted in all packet types with two exceptions:in the FIRST packet, the seq field is used to convey an initial allocation value for the return data stream, and in the JOIN packet sent from a multicast receiver wishing to join a multi-cast association, where the seq field is used to convey the receiver's identity.

For DATA packets, the value in the seq field is the sequence number of the f irstmeaningful byte of the Information Segment. The range of sequence numbers representing the data in a DATA packet begins with seq and extends to seq + dlen - 1.

The value of seq in control packets is the next sequence number to be sent on theoutgoing data stream. Since control packets do not carry data, no sequence numbers are consumed by a control packet's payload.

Associations always begin both data streams at sequence number 0. Since a sequence number is not necessary to identify the first byte of the outgoing data stream (it is always zero), the seq field in a FIRST packet is used to indicate the do-not-exceed sequence number for the incoming data stream of the sender of the FIRST packet. This is the initial allocation value for the return data stream; subsequent control packets carry updated allocation information thereafter.

4.1.6 Control Segment

A Control Segment reports the state of the context that sent it. QMP packets containing a Control Segment as their payload are referred to as control packets. Control packets are used to exchange state information between QMP endpoints.

The Control Segment is included in CNTL, ECNTL, and TCNTL packets. These three packet types correspond to the three possibilities for the Control Segment. A CNTL packet contains a Common Control segment with information most commonly needed, such as flow control information. The ECNTL contains an Error Control segment, which contains all of the fields of the Common Control segment but also holds error control information. The TCNTL packet contains a Traffic Control segment, and is used to supply the return key.

- Common Control Segment

  The format of a Common Control segment is shown in Figure 4.3. These three fields represent the state information for which a control packet is most commonly needed. The fields in this segment are common to all three control packet types. The first two fields, rseq and alloc, are flow control parameters and together define the flow control window. The third field, echo, is used to identify a particular control packet as a response to a packet sent with the SREQ bit set.

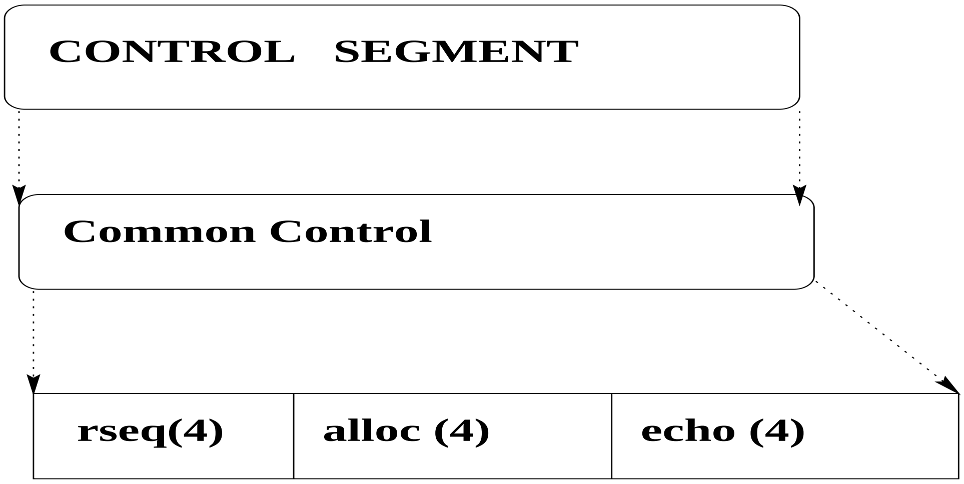

Figure 4.3: Common Control Segment

rseq field : The rseq field is interpreted and must contain a meaningful value in all control packets. The rseq field holds the sequence number of the next in-sequence byte expected on this data stream, so it is one greater than the highest contiguously received data byte.The rseq value serves as the lower edge of the flow control window and, consequently , acknowledges all data whose sequence numbers are less than the value of rseq. For a context in which no data have as yet been received, rseq is the starting sequence numberfor the data stream, which is zero.

alloc field : The alloc field is used in flow control. The value of the alloc field is a sequence number that limits the amount of data a sender may transmit; a sender may send data with sequence numbers up to, but not including, the sequence number in alloc. The alloc value represents the amount of data the receiver is willing to accept, which may or may not correspond to internal buffer space of the receiving QMP machine. When the RES bit is set, however , the alloc value represents the buffer space reserved for this association.

echo field : The echo field is valid in all control packets. It is used to match a control packet with the status request (sent SREQ bit) that may have caused the control packet to be generated.The value of the echo field is the highest seen sync value from all incoming packets (see Section 2.2.6). The highest sync value yet seen is held in a context variable called rcvd_sync; when a control packet is generated, the value of rcvd_sync is placed into the echo field of the control packet. Thus the value in the echo field of an incoming control packet indicates the highest seen sync value at the time of its transmission.

If the echo value is less than the saved_sync value, the control packet is not a response to the most recent status request. If the echo value is equal to the saved_sync value, the information in the incoming control packet is no older than the time at which last status request was generated. Therefore, the echo value is useful for:

1. Taking round-trip time samples

2. Stopping a synchronizing handshake, if one is in progress

3. Aggregating control information from the set of multicast receivers Round-trip time estimates can be collected if the current time is saved when a packet with the SREQ bit set is sent (and its sync value is recorded in saved_sync). This saved time is subtracted from the current time when a control packet arrives whose echo value is equal to the saved_sync value. (Note that this only works if the control packet is in response to an SREQ.) The difference in the times is a round-trip time observation. The value used to load the WTIMER is a value derived by smoothing observations of round-triptimes.

4.1.7 Traffic Control Segment

The format for the Traffic Control segment is given in Figure 4.4. A Traffic Control segment includes all of the fields of the Common Control segment with two additional fields and a Traffic Specifier segment. The two additional fields are rsvd (reserved) and xkey(the exchange key field).

Only time when traffic control segment is used is when we need to do key exchange. This occurs during connection setup. During connection setup, the keys are exchanged and these keys are used for furthur communication. So, we use this control segment only once.

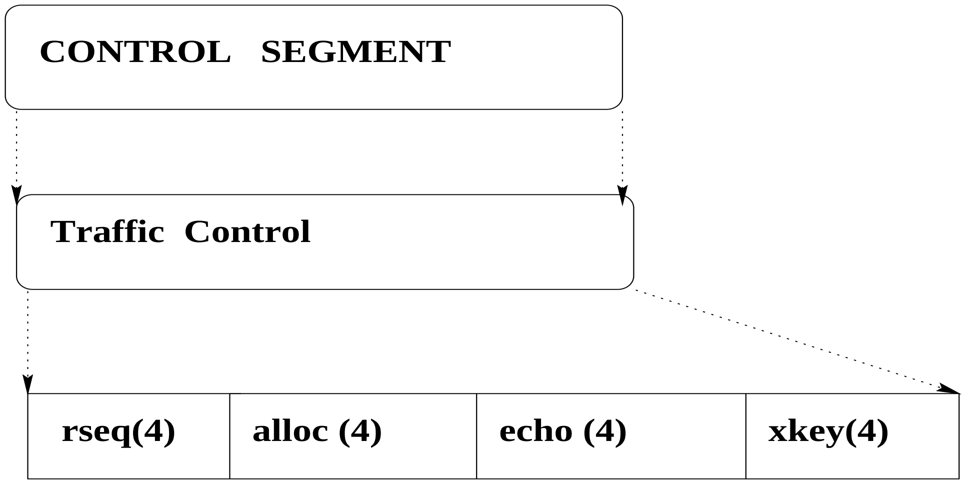

Figure 4.4: Traffic Control Segment

- Reserved Field

  The rsvd field is reserved and must be zero for all TCNTL packets.

- Exchange Key Field

  The xkey field is interpreted in all TCNTL packets. The value of the xkey field mustbe the return key value (i.e., the RTN bit must be set) for the context sending the packet.The value in the xkey field supports the key exchange mechanism described in Section.

- Traffic Specifier Segment

  We do not use this segment. This is used to specify traffic patterns in the communication. Used in WAN environment.

4.1.8 Information Segment

The Information Segment encapsulates user and other protocol and diagnostic information. QMP packets containing an Information Segment as their payload segment are called information packets. DATA, FIRST, JOIN, and DIAG packets are information packets corresponding to the possibilities in Figure 2-1. The first two types can contain higher layer (user) data. These are referred to as data-bearing packets. The other two contain transport layer messages only.

There are four segments used in information packets, the Data segment, the AddressSegment, the T raffic Specifier, and the Diagnostic Segment. These segments are used in various combinations to construct the several information packets. The Data Segment,Address Segment, and Diagnostic Segment are discussed below; the Traffic Specifier has already been defined in Section 4.1.7.

Information Segments of DATA and FIRST packets always consume sequence space; those of DIAG packets do not.

- Address Segment

  The Address Segment format is shown in Figure 4.5. The Address Segment contains destination and source addressing information. The packet type which require an Address Segment within their Information Segment is FIRST . An Address Segment of a FIRST packet consumes sequence space.

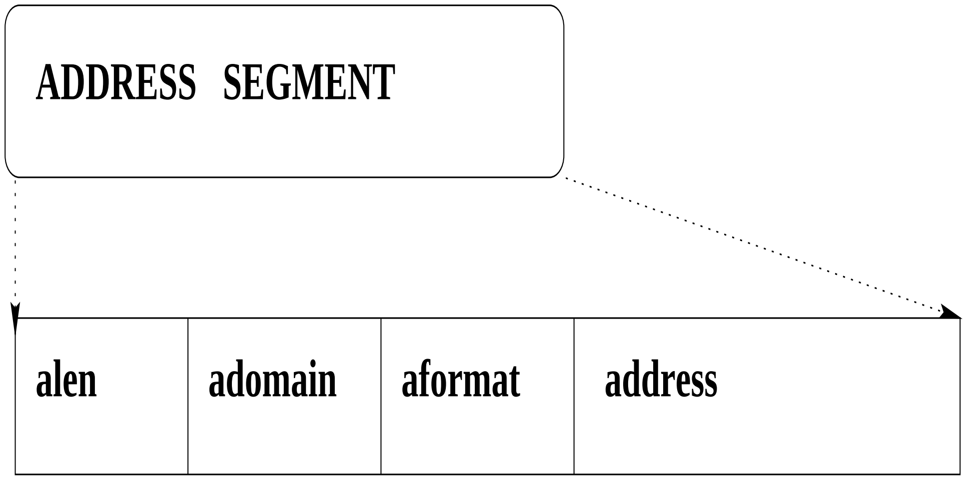

Figure 4.5: Address Segment

The Address Segment is carried in FIRST packets only. These packets are used to initiate or join an association; once the association has been established , the addressing information is no longer explicitly carried by packets in subsequent packet exchanges.

Address Format field : The **aformat** field identifies the address syntax.

Address Domain field : The **adomain** field is an address demultiplexer. Some address formats are actually used by two or more addressing domains. The adomain value disambiguates which address domain is being used.

Address Segment Length field : The **alen** field specifies the length of the whole Address Segment.

4.1.9 Data Segment

The Data Segment, show in Figure 4.6, is a variable-length segment. Data Segments must consume sequence space so that error control, which is based on sequence numbers, can be applied. Data Segments are intended for transmission of higher-layer application (user) data. Two packet types can include Data Segments: FIRST and DATA.

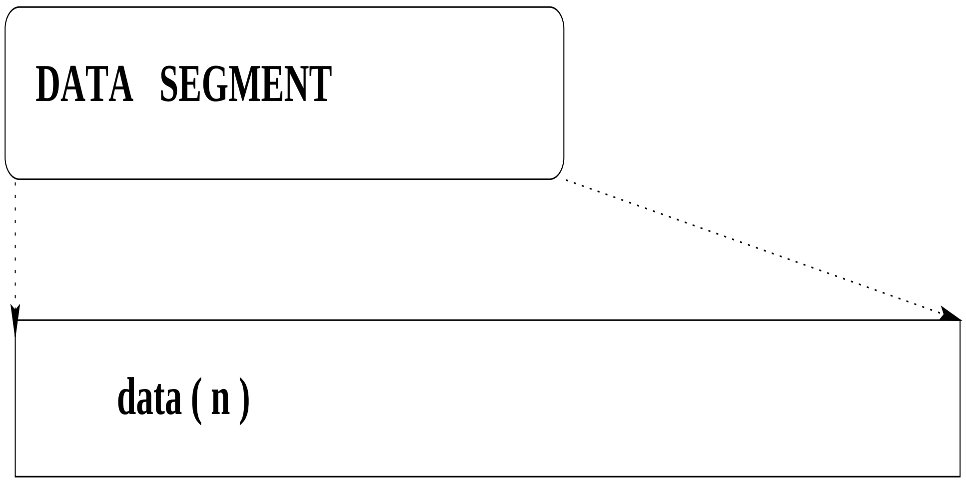

Figure 4.6: Data Segment

- Data Field :

  The data field of the Data Segment is a variable length field that holds user data.The contents of this field are not interpreted by QMP.

4.2 Packet Types

The seven packet types in XTP provide the mechanism for information exchange between endpoints in an association. This information consists of protocol and user data. Certain packet types are used exclusively to exchange protocol state information. These packets are generically called control packets. The other packet types are used to carry information, either in the form of user data or some non-state information. These packetsare called information packets.

The packet type is determined by the **pformat** field in the **cmd** field of the header. **FIRST** packets initiate an association. Subsequent data is transferred with the **DATA** packets. Flow control and other state information required often during an association are conveyed via the **CNTL** packet. **ECNTL** packets add error control information to the control information, and **TCNTL** packets add traffic control information. **DIAG** packets are used to notify the recipient of error conditions at the sending end-system.

The header syntax is common to all packets, and the fields within the header are usually parsed and interpreted in the same way for each packet type. The text will call out situations where this is not true.

In general, the key field is used to map the packet to the appropriate context. The **cmd** field carries options for this packet and the modes for the association, as well as the packet type identifier and the protocol version number. The **dlen** field is the length of the payload segment, which is anything that follows the header. The **sync** field contains the count of the number of packets with **SREQ** set sent from the context sending this packet. The **seq** field, last of the header , indicates the next byte expected on the sender's outgoing data stream for control packets, or the first byte in the information segment for data-bearing packets.

4.2.1 FIRST PACKET

The syntax for a **FIRST** packet is given in Figure 4.7. The **FIRST** packet carries all of the information necessary to find a listening context at a destination host, and establish an association between that context and the sending context. To effect this, the **FIRST** packet includes an address specification.

The entire payload of the **FIRST** packet is part of the sender's outgoing data stream ,so every byte after the header is assigned a sequence number. The **dlen** field indicates the number of bytes in the payload.Since data streams in both directions begin with sequence number zero, the **seq** field in the **FIRST** packet is used to specify the initial allocation value to be used for the return data stream.

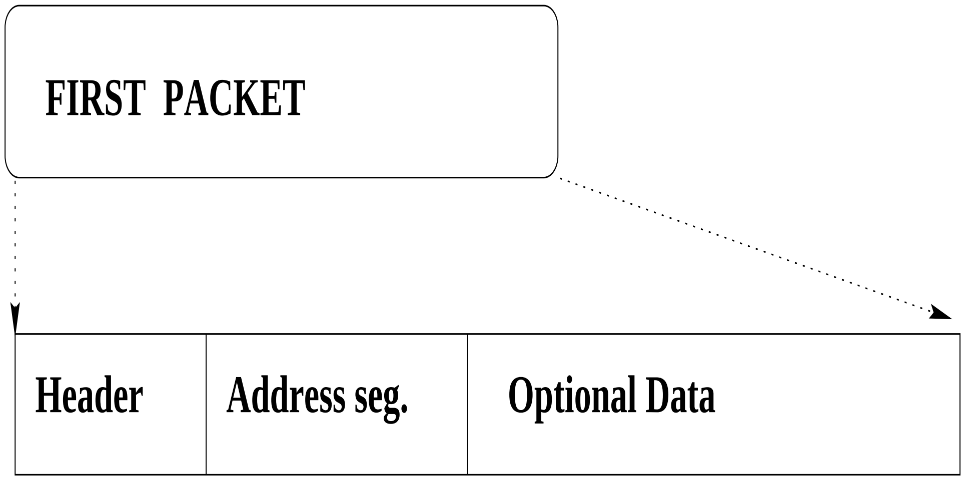

Figure 4.7: First Packet

The Address Segment must be matched against the address filter of each of the listening contexts at the destination host. Section 4.3.2, "FIRST Packet Matching," describes how a **FIRST** packet is matched with a listening context.

A **FIRST** packet is intended to be used only once during the lifetime of the association, assuming it or its acknowledgement is not lost. The address is needed only once, subsequent packets will use the key field to identify the proper recipient.

4.2.2 DATA PACKET

The syntax for the **DATA** packet is shown in Figure 4.8. After establishment of the association, subsequent data transfers in both directions use the **DATA** packet. The **seq** field indicates the beginning sequence number for the data contained in the **Data** Segment.

The **dlen** field gives the length of the data. Every byte in the **Data** Segment is counted toward consuming sequence space.

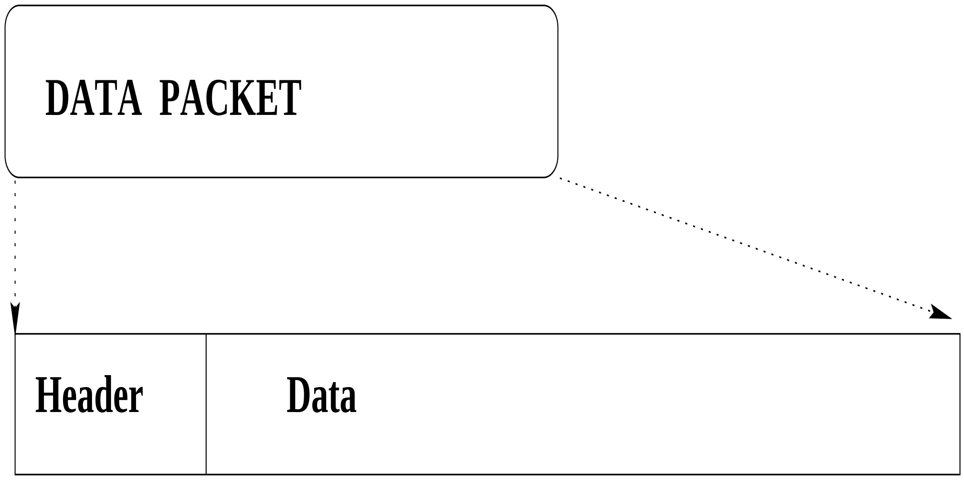

Figure 4.8: Data Packet

4.2.3 CNTL PACKET

The common control packet is the **CNTL** packet, shown in Figure 4.9. The **seq** field holds the sequence number of the next untransmitted byte to be sent on the outgoing data stream of the sender of this packet. The **CNTL** packet, however , does not consume sequence space. The receiver of the **CNTL** packet can use the **seq** value to compare it with the next sequence number expected on its incoming data stream , this comparison will indicate the need to generate an ECNTL packet as a fast negative acknowledgement.

The **rseq** and **alloc** fields are used to convey flow control information for the sending context's incoming data stream. The **rseq** field holds the highest sequence number for data received without gaps. It therefore represents the lower edge of the flow control window.The **alloc** field represents the upper edge of the flow control window . The value it holds is one past the highest sequence number acceptable on the incoming data stream. The **echo** field carries the highest **sync** value yet seen in any incoming packets. The act of echoing this monotonically increasing number back to its originator gives the originator some idea of how synchronized the endpoints are. The details of the synchronizing handshake procedure are discussed in detail in Sections 4.6.3 and 5.4.2.

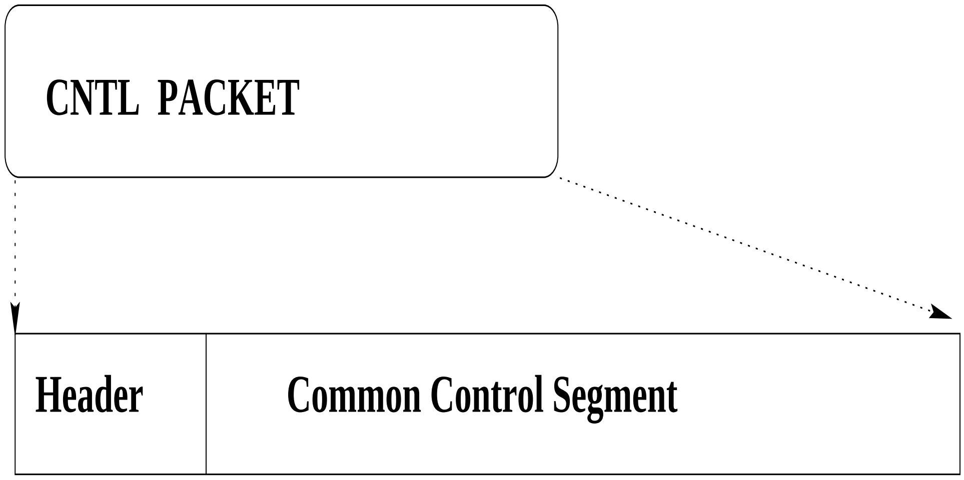

Figure 4.9: Control Packet

A **CNTL** packet is not used if an error in the data stream is encountered,**ECNTL** packets are used under these conditions. A **CNTL** packet may be generated at any time , but there are several specific conditions that cause a **CNTL** packet to be generated . While the data stream is error free, a **CNTL** packet is generated in response to the following conditions:

* the **SREQ** bit is set in any incoming packet, except during traffic specification negotiation (a **TCNTL** is used)

* all of the data received prior to the arrival of a set **DREQ** bit have been delivered to the user

If more than one of the above conditions exist, a single **CNTL** packet can be sent.

4.2.4 ECNTL Packet

The **ECNTL** packet is used to convey error control information as well as common control information. The syntax for an ECNTL packet is given in Figure 4.10. The **rseq** , **alloc, and echo** fields are interpreted just as in a **CNTL** packet. We use this type of packet when there is out of sequence packet. But since one of our basic assumption is that there would be very few losses because of highly reliable network, there would be very few such packets.

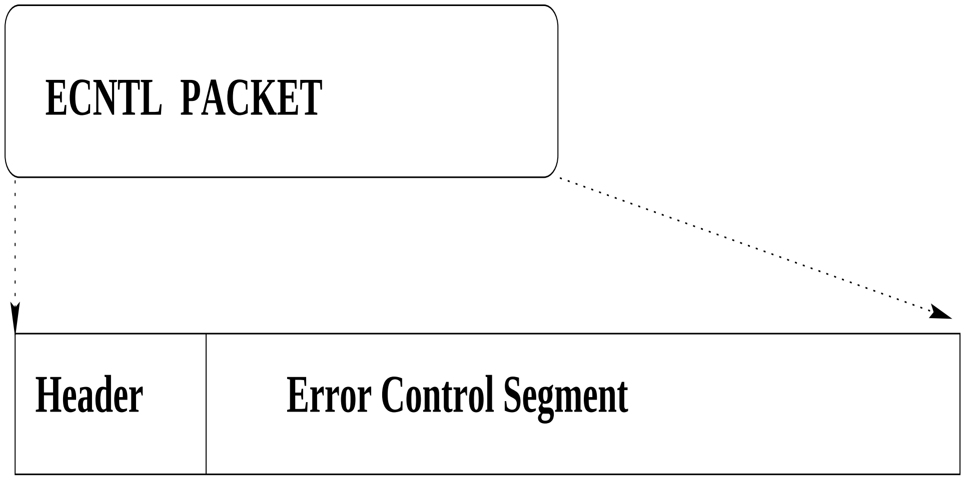

Figure 4.10: ECNTL Packet

4.2.5 TCNTL Packet

The syntax of **TCNTL** packet is shown in following figure 4.11. The **rseq, alloc and echo fields** are interpreted just as in **CNTL** packet. Main utility of **TCNTL** packet is its use in establishing connection. We use this packet during key-exchange. After connection establishment, key is used for addressing , so we need this packet only once and that is during connection establishment.

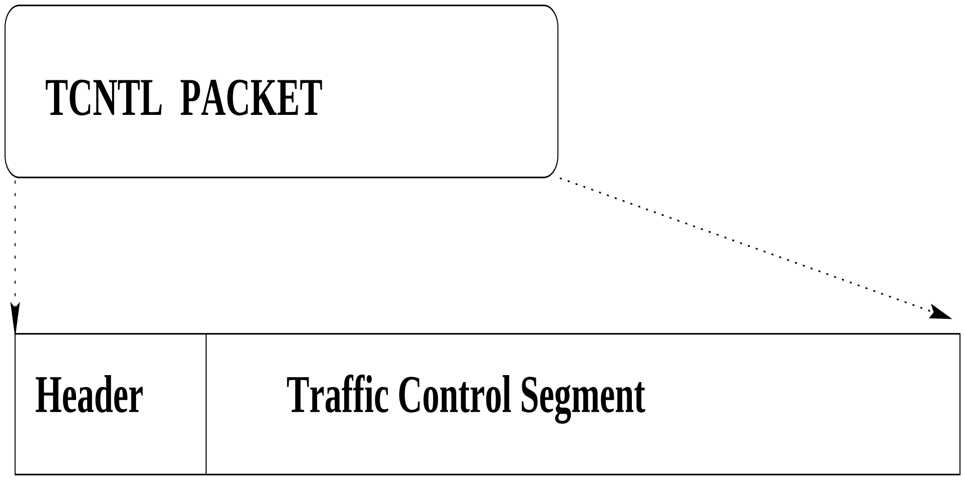

Figure 4.11: TCNTL Packet

4.3 Unicasting IN QMP

This section first defines the fundamental concepts in **QMP** , including the context and association state machines. Next, the procedures for unicast association management are discussed, complete with unicast association establishment and termination. The last subsection describe the particulars of the flow and error control procedures.

4.3.1 Protocol Fundamentals

Processes : Conceptually, an QMP implementation consists of four processes: a **receiver process, a sender process, a reader process, and a writer process**. The receiver and sender processes provide QMP' s interface to the underlying data delivery service. The reader and writer processes provide QMP's interface to the users (applications). Their relationship to one another and to associated structures is shown in Figure 4.12.

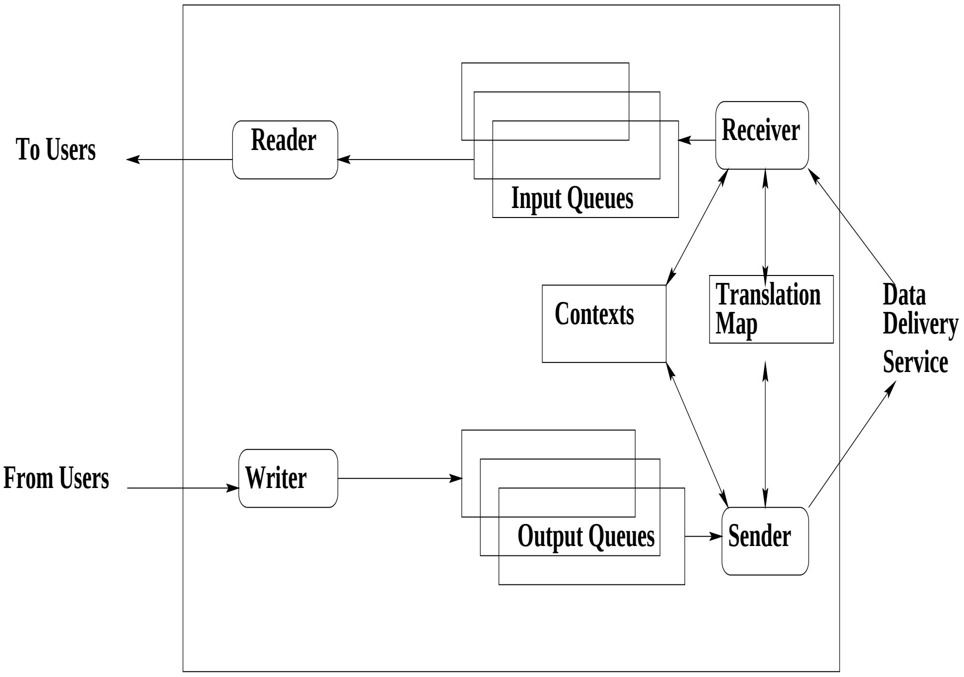

Figure 4.12: Host Architecture of QMP

The receiver process acts on packets received from QMP's underlying data delivery service, directing the packets to the appropriate contexts. A packet is matched to its context by using some packet header information to index into the translation map. Once found,the context parses the packet's control information. If the packet is a data-bearing packet, the data are placed directly into input queues pending commitment from the context. When the context is assured that the data are valid, it commits the data to the reader process. The users then access this data through the reader process.

The writer process acts upon output commands from the users, placing data into the output queues and informing the appropriate contexts of the data's presence. The contexts construct packets and give them to the sender process. The sender process pulls data fromthe output queues into the outgoing data-bearing packets, and gives the data and control packets to the underlying data delivery system.

Context State Machine : Figure 4.13 shows the state diagram for a context. All contexts in an implementation exist in a quiescent state until they are activated. When a user submits an input command, a quiescent context moves to the listening state. This must occur some time before another user at a remote host issues an output command. The output command causes a quiescent context at this remote host to move directly into the active state, and a **FIRST** packet to be generated and sent. When this **FIRST** packet is received by the context in the listening state (and all conditions are met for accepting this **FIRST** packet), the listening context moves into the active state. The contexts at both endpoints are now active, and data transfer may occur in both direction for an arbitrary length of time. When the contexts are finished sending and receiving data, they each move into the inactive state. When one of the contexts issues a packet with the **END** bit set, the association is terminated and both contexts return to quiescent states.

Association State Machine : The state diagram for an association is shown in Figure 4.14. This diagram is from the point of view of one of the contexts in the association. The association state machine begins with both the incoming and outgoing data streams in the active state. The dual nature of this state machine reflects the fact that an association controls two data streams. As data flows in both directions, both data streams remain in active states. The outgoing data stream side moves from output active to local writer closed when a packet is sent with the **WCLOSE** bit set. The local writer process may not transmit any new data (no sequence space can be consumed from this point on), but previously transmitted data may be retransmitted. When a packet is received with its **RCLOSE** bit set, the outgoing data stream moves from local writer closed to output inactive. Receiving an **RCLOSE** bit implies that all data have been received to the

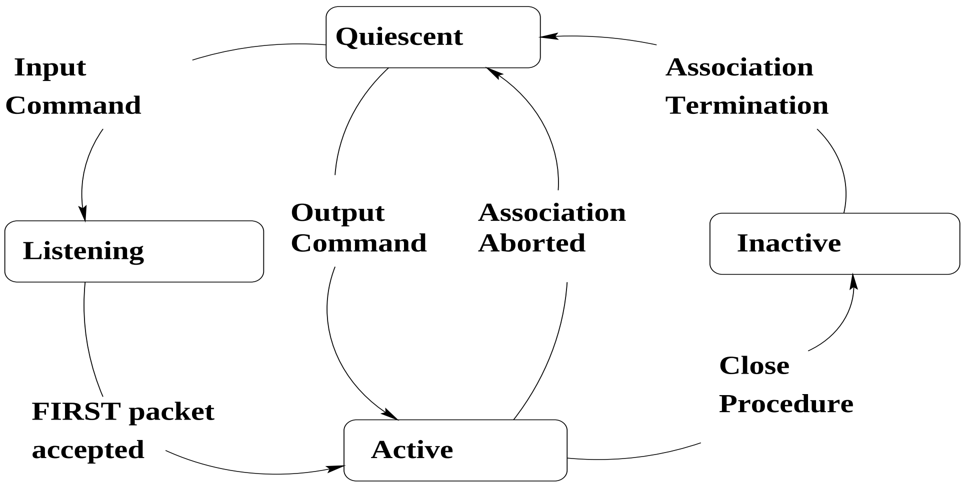

Figure 4.13: Context State Machine

satisfaction of the receiver , so no further **DATA** packets whatsoever can be sent on the outgoing stream. Independently, the incoming data stream moves from input active to remote writer closed when a packet with the **WCLOSE** bit set is received. This tells the context that no new data will be arriving on this data stream. When all data have been received, according to the error control parameters, a packet with the **RCLOSE** bit set is sent. This moves the incoming data stream into the input inactive state.

Either data stream may go inactive first, but once both data streams have gone inactive, they both move into the association inactive state. This is equivalent to the context inactive state. When one of the endpoints sends a packet with an **END** bit set, the association moves from inactive to closed, which is the condition for the context state machine to move from inactive back to quiescent.

Data Streams : A data stream is an abstraction for an arbitrary length string of sequenced bytes, where each byte is associated with a sequence number. The sequence space for a data stream starts at zero and continues indefinitely. Only specific fields in QMP packets consume sequence space. Sequence space is conceptually monotonically increasing, but due to finite sequence number representation, "rollover" may occur such that one byte may be further into the sequence space than another but have a smaller sequence number.

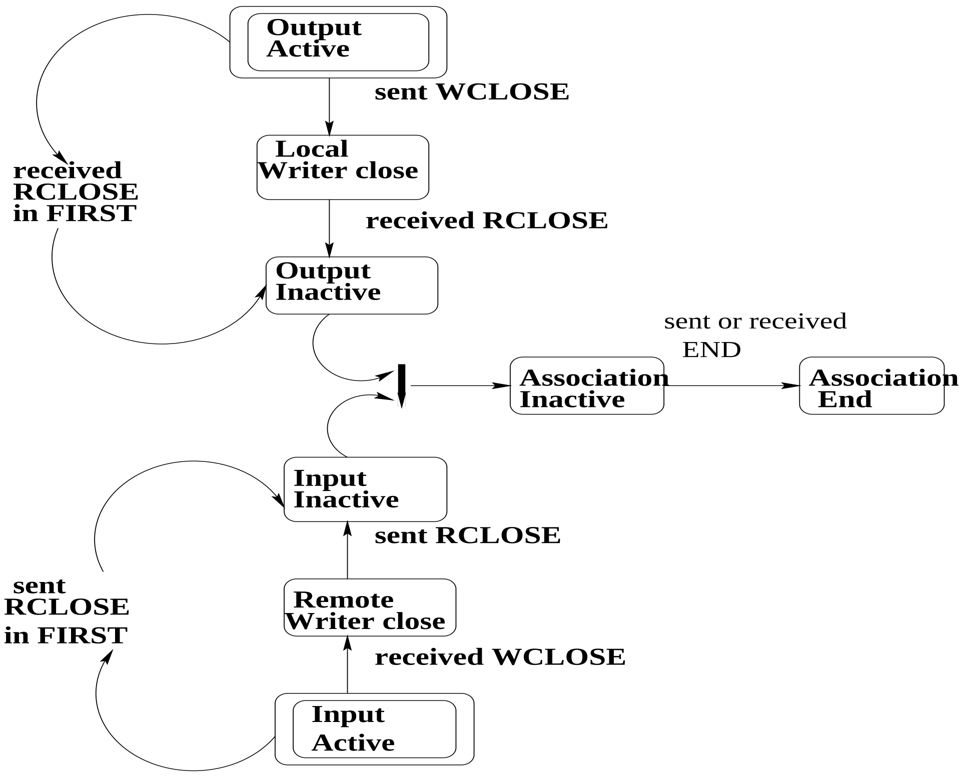

Figure 4.14: Association State Machine

4.3.2 Association Management

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. This is described in the section called **FIRST Packet Matching.** The **Full Context Lookup and Abbreviated Context Lookup** sections describe mapping incoming packets to the appropriate contexts. The **Key Exchange** section describes an optional optimization that eliminates the need for full context lookups for incoming packets. Whenthere is no longer a need for the association, the association is closed using procedures described in **Association Termination**.

- **FIRST Packet Matching** : Upon receipt of a **FIRST** packet, the receiving host takes the following steps:
  1. If a full context lookup on this **FIRST** packet finds an active context, the **FIRST** packet is a duplicate; the context to which this packet belongs should respond to the **SREQ** and **DREQ** bits, if set,and accept any additional data carried within, but

no new context becomes active.
2. If the **FIRST** packet is not a duplicate,the receiving host performs a series of comparisons to find an appropriate listening context:

* The address field within the Address Segment of the **FIRST** packet is matched against a set of filters.

* The settings of the option bits in the options field of the header of the **FIRST** packet are examined for acceptability.

3. If the **FIRST** packet's address and options are acceptable, the **FIRST** packet is given to this listening context.

4. Otherwise, the matching algorithm moves on to the next listening context, and the comparisons in Step 2 are repeated. If all of the listening contexts are examined but no acceptable match is made, the **FIRST** packet is rejected by using a **DIAG** packet whose code value is 1, "Context Refused."

The listening context is allowed to reject **FIRST** packets if the options are not acceptable. In particular,**MULTI, RES and RCLOSE** are options whose values, set or cleared, may be grounds for rejecting the **FIRST** packet.

- **Full Context Lookup**

  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 translation map structure holds the mapping from packet information to appropriate context. The packet information is used as an index into the translation map to retrieve the handle of the context.

  The key field of an incoming packet is checked to determine if it is a return key or a normal key. If it is a return key, the abbreviated context lookup procedure, described next, is used. If the key value is not a return key, that key value and the packet's source host'saddress (ethernet address) are used as a pair to index into the translation map. If a **FIRST** packet with this key value has been received from this packet's source host, a mapping will exist in the translation map. If not, the lookup will fail, and a **DIAG** packet (containing the code for **Invalid Context**) should be returned to this packet's sender.

- **Abbreviated Context Lookup**

  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.

  If the key field of a received packet holds a return key, the context manager knows immediately that the key value from which this return key is derived was generated at this host. Since the key value is unique within this host, there is a direct mapping from this key value to the context for whom the key value was generated.

- **Key Exchange**

  Packets sent in the return direction are matched with their intended context via the abbreviated context lookup procedure, but packets sent in the forward direction cannot be matched this way because these packets do not carry return keys. The key exchange procedure is the method by which the context that received the **FIRST** packet can tell the context that sent the **FIRST** packet what return key value to use, so that both sides can use the abbreviated context lookup procedure.

  At some point during the association, but usually in response to the first **SREQ** received from packets travelling in the forward direction, a key exchange can occur. The key exchange procedure uses the xkey field in a **TCNTL** packet. The context that received the **FIRST** packet places into the **TCNTL** packet that context's key value with the **RTN** bit set. This is that context's return key value. The context that sent the **FIRST** packet, upon receipt of this **TCNTL** packet, must note the value of the xkey field and must use this value as the key value in all outgoing packets from that point onward.

- **Association Termination**

  An QMP association between a pair of contexts, A and B, involves a pair of simplex data streams: A-to-B and B-to-A. Each context is the sender for one of the data streams and the receiver for the other one. In order for an association to be terminated, both contexts must be released. An association is closed by completing the packet exchanges with **WCLOSE, RCLOSE**, and **END** bits (see Figure 4.3.1a). Closure is achieved with various degrees of gracefulness, depending on the bits used and the order in which they are set, but we have implemented graceful termination in **QMP**.

  The **WCLOSE** bit is set when the sender has completed the transmission of all data in its outgoing data stream. Once the **WCLOSE** bit is set in an outgoing packet, all subsequent outgoing packets, except retransmitted **DATA** packets, must carry the set WCLOSEbit.

  In **Fully Graceful Independent Close**, context A initiates the close of its outgoing data stream with a packet with **WCLOSE** set. When all data are accounted for according to the error control parameters for the A-to-B data stream, Context B responds with a packet with **RCLOSE** set. At this point, the A-to-B data stream is gracefully closed, but the B-to-A data stream remains open. Later, Context B initiates the close of its outgoing data stream by setting the **WCLOSE** in an outgoing packet. Context A responds accordingly with an **RCLOSE**. At this point, Context B sends a packet with the **END** bit set, and the association is terminated.

4.3.3 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 reciever. For ex, if user sets this frequency to 1, then every we send a packets with **SREQ** bit set . This asks the receiver to acknowledge and report its status. This is shown in Fig. 4.15. Following figures show the flow control in two situations, first when SREQ frequency is per packet i.e. the sender asks for status request each packet sent. We also show the flow control mechanism when the SREQ frequency is 2 , i.e. we send one SREQ bit set packets in between 2 data packets. This is shown in Fig. 4.16. The data shown with each arrow from sender to receiver corresponds to quadraplet,**(FLAG, outbseq, outbrseq, outballoc)** . Here, FLAG shows the **option** bit flag set in the Header. **outbseq** corresponds to next sequence number the sender is sending. **outbrseq** corresponds to next sequence number the **receiver** is expecting. **outballoc** corresponds to the sequence number upto which **receiver** can accept data.

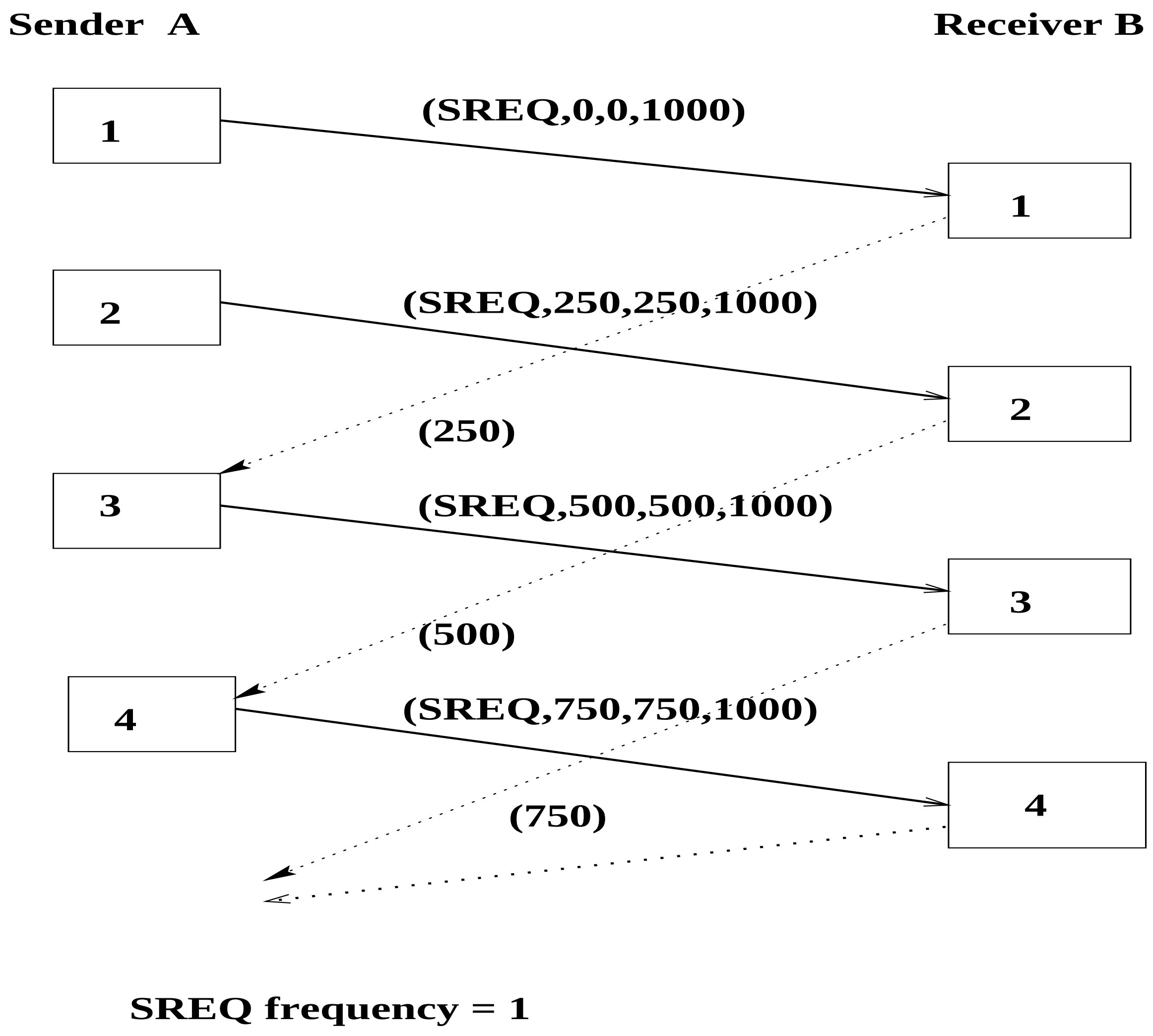

Figure 4.15: Flow Control with SREQ frequency =1

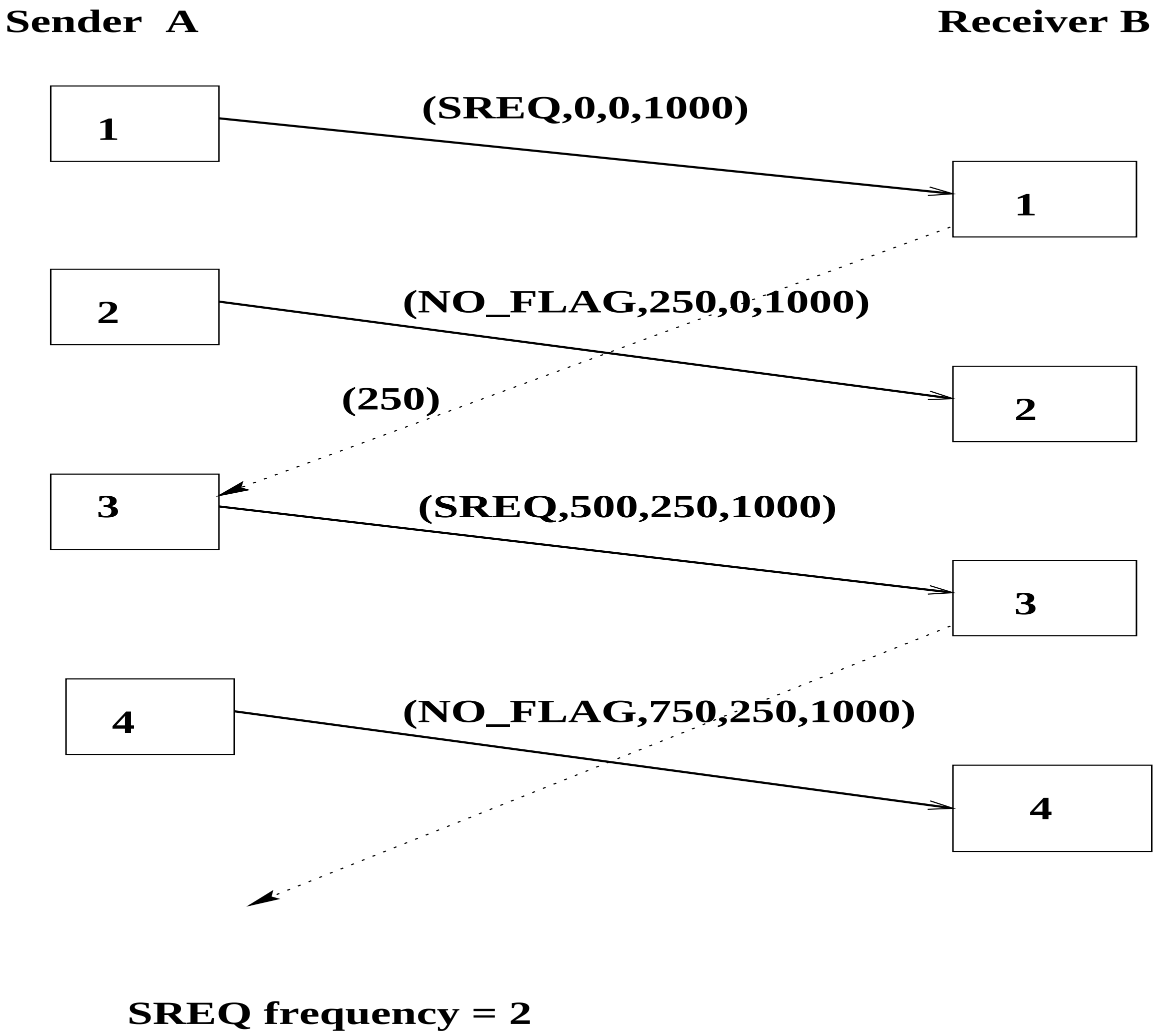

Figure 4.16: Flow Control with SREQ frequency = 2

4.3.4 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 , ie 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. Notification of other error conditions, such as unexpected packets or protocol errors, is made using DIAG packets.

In Figure 4.17 we show the error control mechanism employed. Here the SREQ frequency is set at 1, i.e. with each packet we set the SREQ bit.

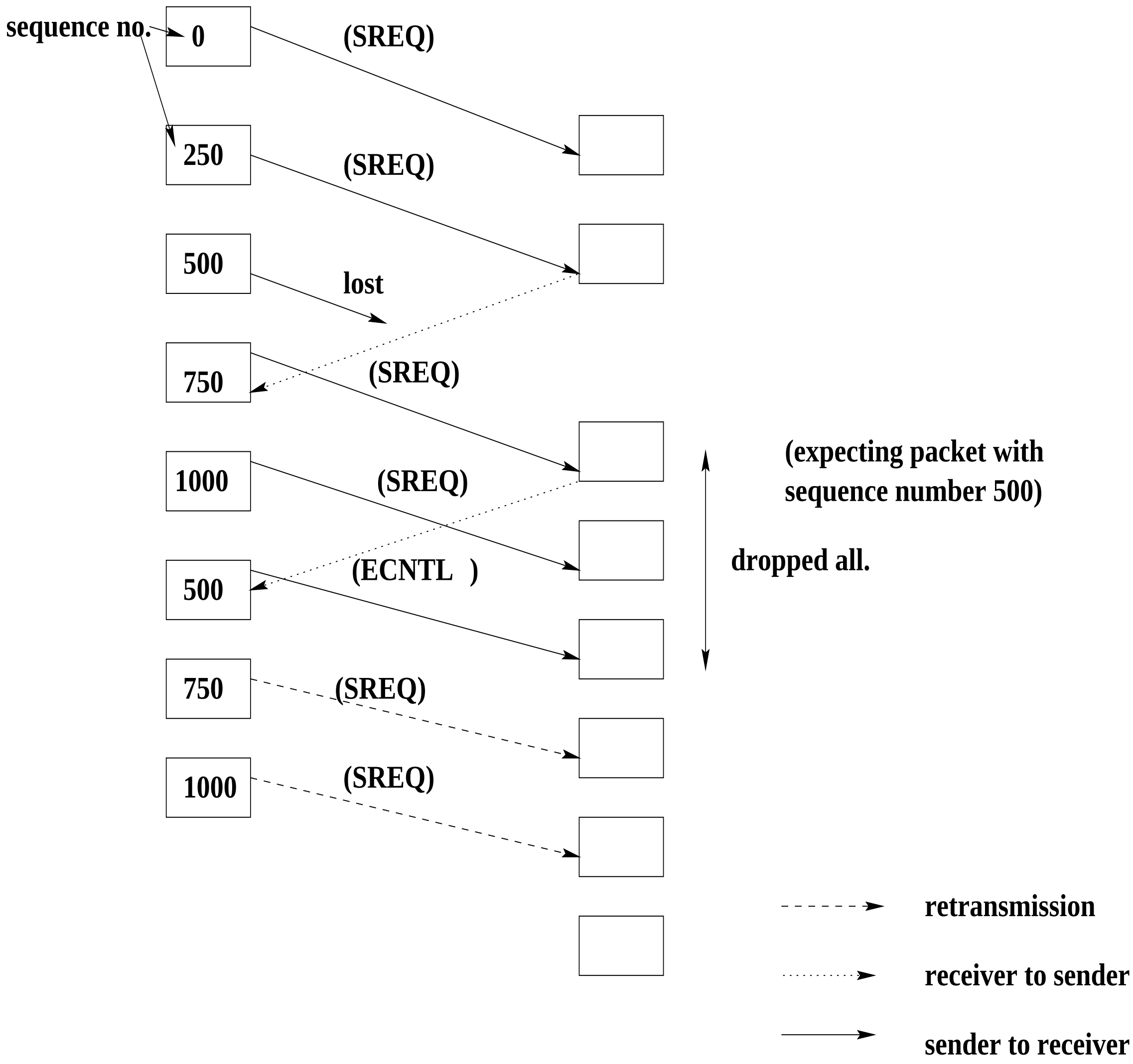

Figure 4.17: Error Control Control

Chapter 5

QBUF - A Zero-Copy I/O Framework

5.1 Background

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.

- 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.

Clearly, the overheads of data touching operations are most significant for large data transfers and reducing data touching costs can have considerable benefits for such cases. There have been previous efforts to reduce the number of copies in order to improve performance. Some of the schemes have been as follows :-

- Mark the pages of the user buffer as **Copy-On-Write** and do I/O on them. If the user process tries to write into that buffer while I/O is going on then the MM system of the kernel will copy that page and give the fresh copy to the user process. Even though this scheme looks attractive we have not adopted this because changing page table entries (like setting the Write-Protect flag) is an expensive operation, since we would need to flush the Translation-Lookaside-Buffer or **TLB**. Secondly if I/O is done only on a small fraction of the page the entire page is copied. Hence this might become more expensive than simply copying the user buffer into a kernel buffer.

- Do blocking I/O ie the user process blocks till actual I/O is over. This prevents the user process from writing to the buffers on which I/O is taking place. However this scheme suufers from a serious drawback since it leads to poor overlapping between computation and communication. Hence this scheme is not at all suitable for our objectives.

  We have adopted an entirely different approach in order to do zero copy I/O. Instead of **copying** buffer contents we **tranfer** 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.

5.2 The Design of the QBUF Memory Management scheme

The fundamental drawback in traditional I/O interfaces is that they expect explicit transfer of data from user-defined buffers and kernel buffers. The user expects the buffer supplied by it to be free after it has executed "read" or "write" system calls. This leads to copying of data between user and kernel spaces. An advantage of this scheme is that it allows the user process to reuse the buffer after it has executed the system call. However this leads to poor throughput.

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. 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. However it must be kept in mind that it is possible for the user process to write to that buffer after the system call, since no effort is made to unmap that buffer or mark it as Copy-On-Write. But the user process is expected to follow the semantics of the QBUF scheme and not access the buffer after the system call.

5.2.1 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 5.1. 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 concurently 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 consequetive 1's in the bitmap, but before it can change them to 0's, the process is pre-empted 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, dellocation must be done on the zone from which the buffer was first allocated. What should be done when that zone if locked ? In

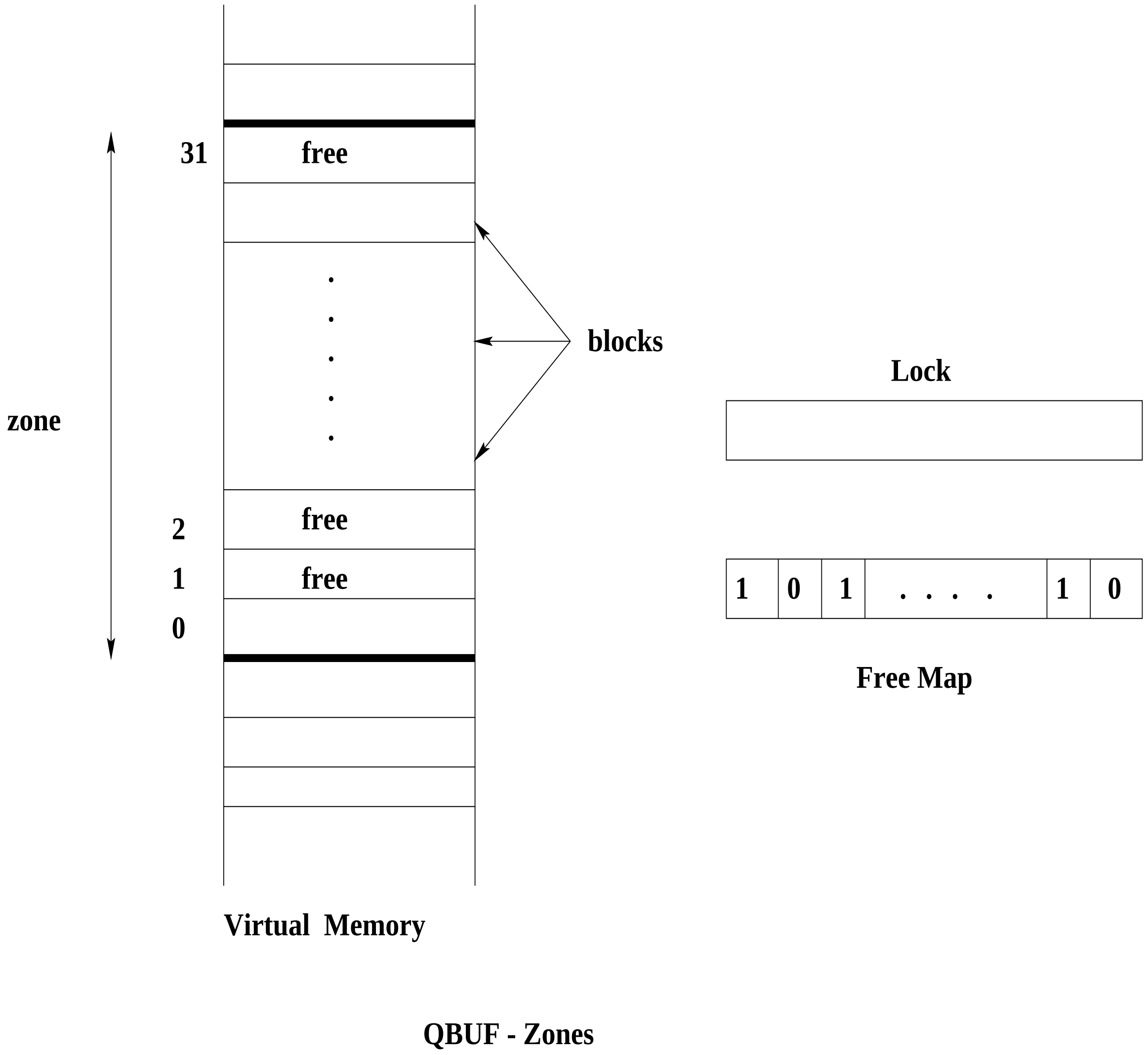

Figure 5.1: Qbuf Zone Model

our scheme we maintain a deferred deallocation work queue to handle such cases. If, while dellocating, 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.

5.2.2 Tranferring buffer ownership

As mentioned earlier the QBUF scheme does not copy buffer contents from user space to kernel space. Instead, it tranfers the **ownership** of buffers. 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 tranfers 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).

5.3 Implementation

The **QBUF** system has been implemented as a character device "/dev/qbuf". This was done so that we dont 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 ie 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.

5.3.1 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 us-

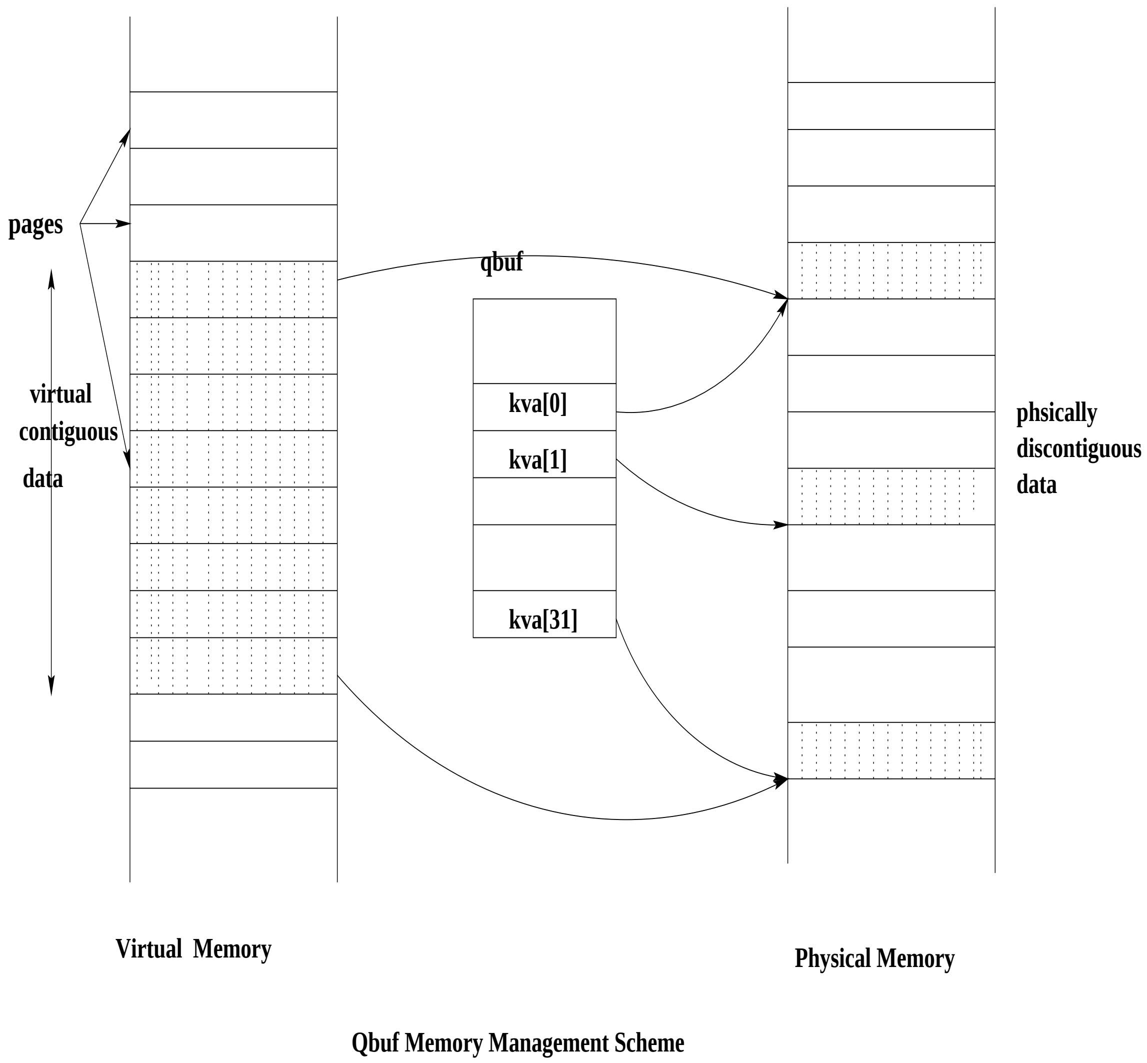

Figure 5.2: QBUF Architecture

ing tha 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 Fir 5.2. 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-¿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.

5.4 Interfaces

5.4.1 User interface

A process registers with the QBUF subsystem by calling qbuf_init(). This is a qbuf library function which can be called after linking with libqbuf.a . After initialization the process can allocate/free qbufs from the QBUF free pool. It allocates a qbuf by calling qbuf_alloc(int size). This function is just like malloc except that it allocates from the QBUF free pool.

There are two new system calls which transfer qbufs between the user process and the kernel. **"qb_write()"** transfers a qbuf from the user to the kernel. The qbuf should not be touched after calling qb_write(). It will be freed later on after I/O is complete. **"qb_read()"** transfers a qbuf from the kernel to the user process. As mentioned earlier this should be passed a pointer to a pointer, since it gives the address where the qbuf is mapped.

When a process obtains a qbuf from the kernel using qb_read() it frees the qbuf by calling qbuf_free(). This function returns the QBUF to the free pool.

5.4.2 Kernel Interface

The interface functions are :-

- qbuf_register() :- This function is called when the I/O subsystem registers with the QBUF subsystem. It returns a qbuf handle. The handle is used for all subsequent allocations/deallocations. For example we call qbuf_register() at socket creation time. The handle is stored in the QMP context data structure and is used subsequently for qbuf_wrap(), qbuf_vfree() and qbuf_valloc().

- qbuf_wrap() :- This function is used to encapsulate a user virtual address (obtained by the user process using qbuf_alloc()) in an internal data structure. This function wires all the underlying physical pages and stores the physical addresses in the kva[] vector. Hence after calling qbuf_wrap() we can access the physical addresses directly. This function is called in qb_write().

- qbuf_valloc() :- This function is called to allocate a qbuf in the kernel. It allocates a qbuf from the QBUF free pool and wires the underlying pages. After calling this function "read" I/O is done on the underlying physical pages.
- qbuf_vfree() :- This function is called to free a qbuf that has been allocated by a user process and encapsulated using qbuf_wrap(). It unlocks the underlying physical pages and frees the corresponding blocks in the free-list bitmap. However if the zone is locked the freeing is deferred in a work queue.

Chapter 6

QMP Implementation

In this chapter we will discuss the implementation issues. We will discuss the change that we introduced in sock data structure and sk_buff data structure. After that we will discuss about the **asynchronous send** which is another feature which we have added. Then, we will discuss the **context** data structure which is now the end point of communication. After that we will discuss the **qbuf** data structure and details of **qbuf** memory scheme.

6.1 Change in sock data structure

In Linux, sock is the end point of communication. It contains the protocol independant information for end-point, like receive_queue, which is a list of receive buffers and write_queue which is a list of send buffers.It has a union field “tp_pinfo” which holds transport layer protocol specific information. All transport layer protocols have an entry into that union. So, we introduced our protocol specific information data structure, “struct qmp_ctxt af_qmp” in this union . Now, this points to our context for each end-point which we can use to implement our protocol specific functionality.

6.2 Change in sk_buff data structure

In Linux,data is transmitted using network buffers called sk_buff’s.All data that is transmitted is first copied into sk_buff’s and then sent to driver. Also, the data contained in sk_buff is exptected to be contiguous. So, whenever we send any user data, we have to copy it into sk_buff . So, our user data cannot be discontiguous.This is an overhead. So, we introduced two fields in the sk_buff data structure, **udata and ulen**, where first one points to user data and other one contains the length of user data. So, now our header

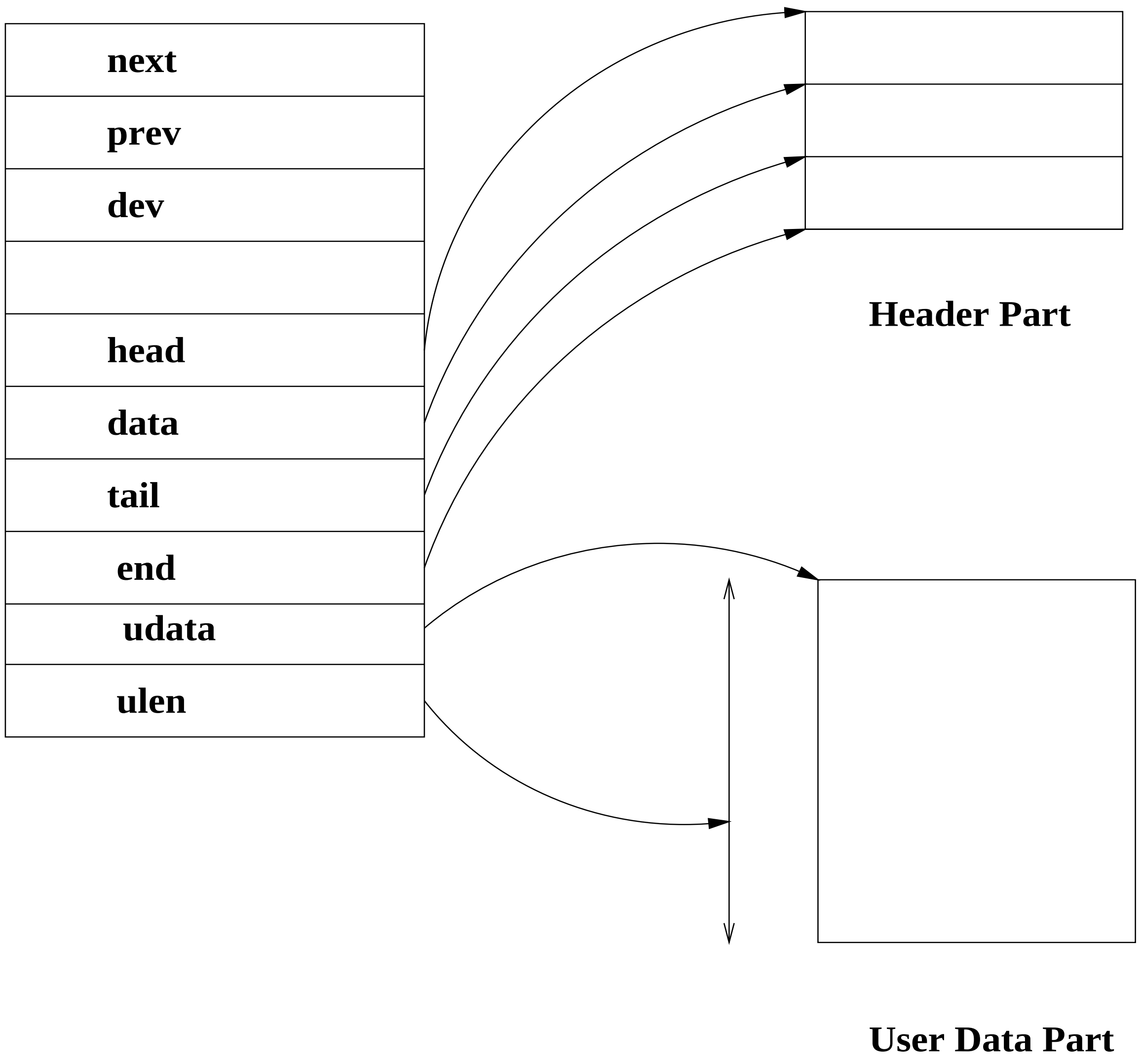

Figure 6.1: Modified skbuff data structure

and user data part are not contigous. The traditional pointers like **head, data, tail, end** still point to header part of new sk_buff , as they are used to identify the various header locations like ethernet header and qmp header. Following figure shows the modified sk_buff data structure.

6.3 Asynchronous Send

Conceptually send can be done in two ways, either synchronously or asynchronously. In synchronous send, the user process waits until the data has been sent . While in asynchronous send , the send system call returns as soon as data is put up in the write_queue of the sock data structure. All send processing is done at bottom half .This helps in reducing latency.

The socket send system call that we implemented, qmp_sendmsg() sends messages asynchronously. Following pseudo-code explains the working:

```
static qmp_sendmsg(){
            copy_from_user();
            put data into write_queue of sock;
           start_bh_atomic();
           now start sending;
           end_bh_atomic();
    }
```

In this, we first copy the user data from user level to kernel level. Then we insert that buffer into write_queue of sock data structure. Now, sending is done at bottom half time as the sending is surrounded by bottom half code.

6.4 Hash Tables

We have used three hash tables for correct demultiplexing of packets to their final endpoint.

- Listen Hash Table : This hash table is used by all listening contexts. For example, if the protocol receives a FIRST packet, then we search all contexts in the listening hash table . If a matching listening context is found, connnection is established and a TCNTL packets is sent. Otherwise, ECNTL packet is sent saying that Connection Refused.

- Established Hash Table : This hash table is used for demultiplexing of all other packets after a connection is established. Here, we use key and mac address to hash into the table and then find out the correct sock structure.

- Context Key Table : This table is used to identify a context given a key. This is used to get a key when a context is initialised and surrender a key when a context is removed.

Chapter 7

Performance And Results

The major objective of the project was to develop a communication protocol which can perform better than TCP/IP in a cluster environment, where the conditions are very favourable , eg low packet drop. So, we have to compare the performance of QMP with TCP/IP for evaluating its utility in cluster environment.

We have used following two benchmarks for evaluating its performance :

- **Bandwidth** : We can find out the bandwidth provided by a communication protocol by sending large size data and receiving large data . We can repeat this for a number of times and then take the average to find average bandwidth supported.
- **Latency** : We can find out the latency involved in sending data using QMP by sending and receiving a small sized packet. Again, we can take the average by measuring it number of times.

7.1 Bandwidth

In our protocol, we have provided BSD socket interface. So, we used sockets to communicate between two nodes. For measuring bandwidth, we send data from both sides and measured the time it took to send that message and receive same size message from another node. Now, we do this for different sizes of data , eg from 100 Byte data to 100,000 Bytes of data. We also evaluated the performance of TCP/IP using same procedure and found out that QMP outperforms TCP/IP. Following graph compares the bandwidth provided by both protocols. In fact, this graph shows the bandwidth provided when we are not

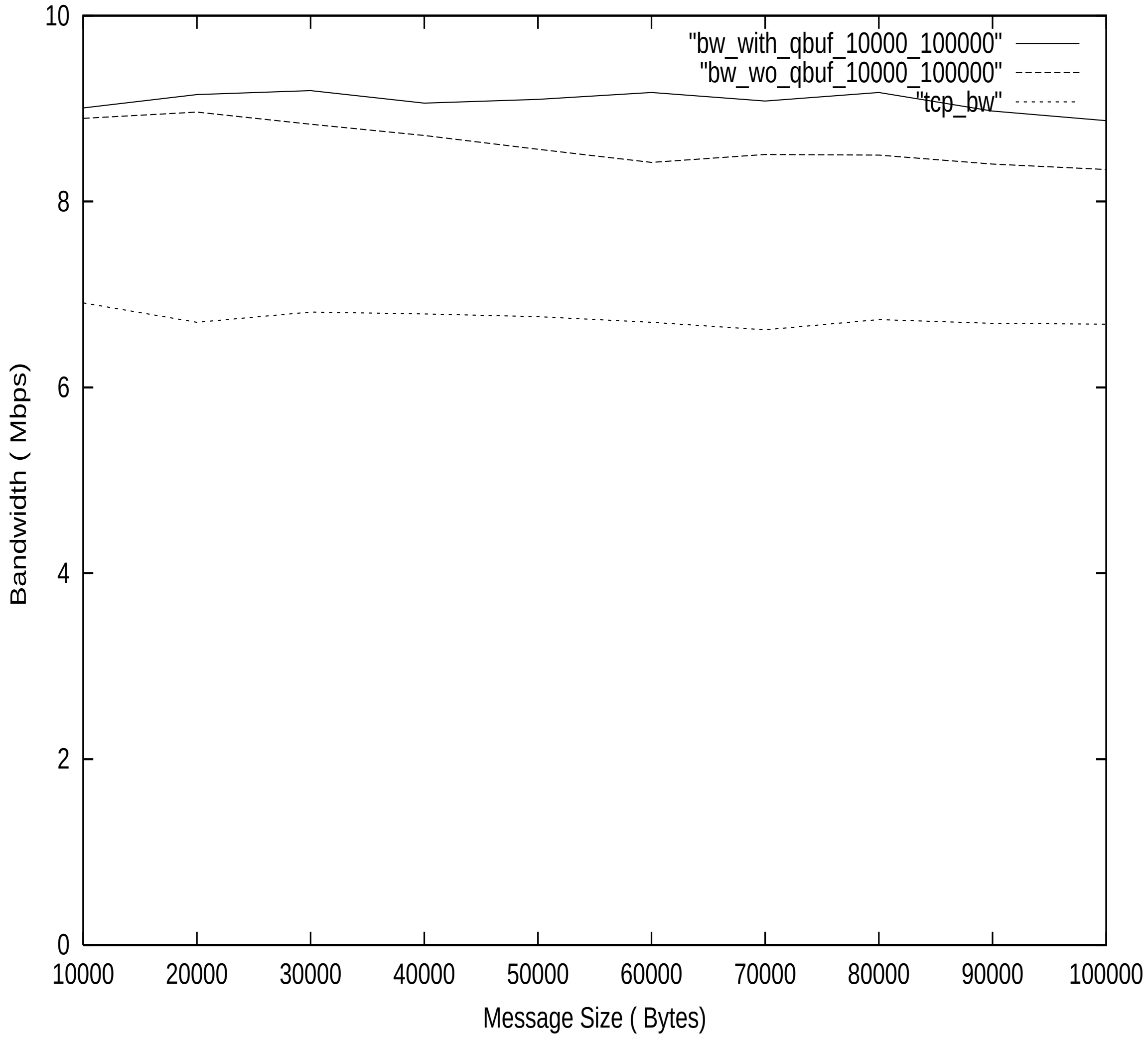

Figure 7.1: Comparing Bandwidth provided by QMP with and without Qbuf and TCP/IP

using our Qbuf memory manager. Using Qbuf memory management scheme, we will show in next pages that Qbuf is really a better protocol than TCP/IP in cluster environment.

The first graph compares the bandwidth provided by **QMP** with and without **Qbuf** and **TCP/IP**. There is a definite improvement when using QMP with large sized messages. Also , when we use qbuf for transferring large sized messages, the qmp provides as high as 9.5Mbps bandwidth on a 10 Mbps line. For small sized messages, using qbuf is not a good idea because we lock a page even if the message is very small and this overhead is costing us. This graph confirms that TCP/IP is not meant for cluster environment because of so many positive features like low error rate, packets arrive in sequence etc. We have achieved

30% improvement over TCP/IP by using our protocol.

This graph 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. So, we stress again that for small messages like checkpointing in distributed applications like pvm, we should use TCP/IP but for process migration ,where there is bulk transfer we should use QMP.

Next graph shows the performance of QMP over message size range of 100 Bytes to 100,000 Bytes . The graph shows logarithmically the performance as there would be many points for the range of Message size. As we can see the performance of QMP initially for small sized messages is low, but it increases as the message size increases and finally settles down at around 9.1 Mbps . This is considerable improvement over TCP/IP.

7.2 Latency

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

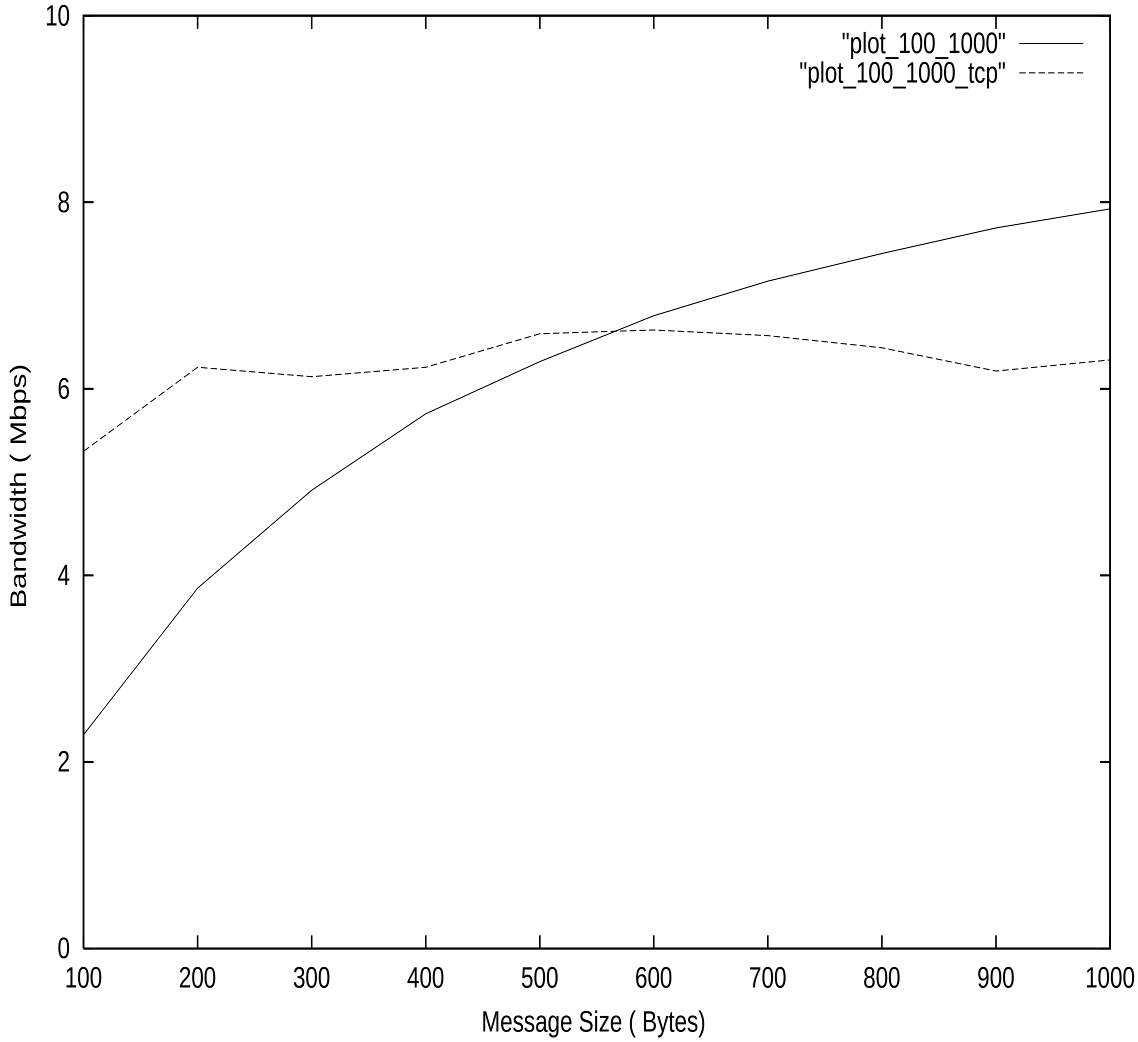

Figure 7.2: Comparing performance of QMP and TCP/IP for Message size from 100Byte to 1000 Bytes

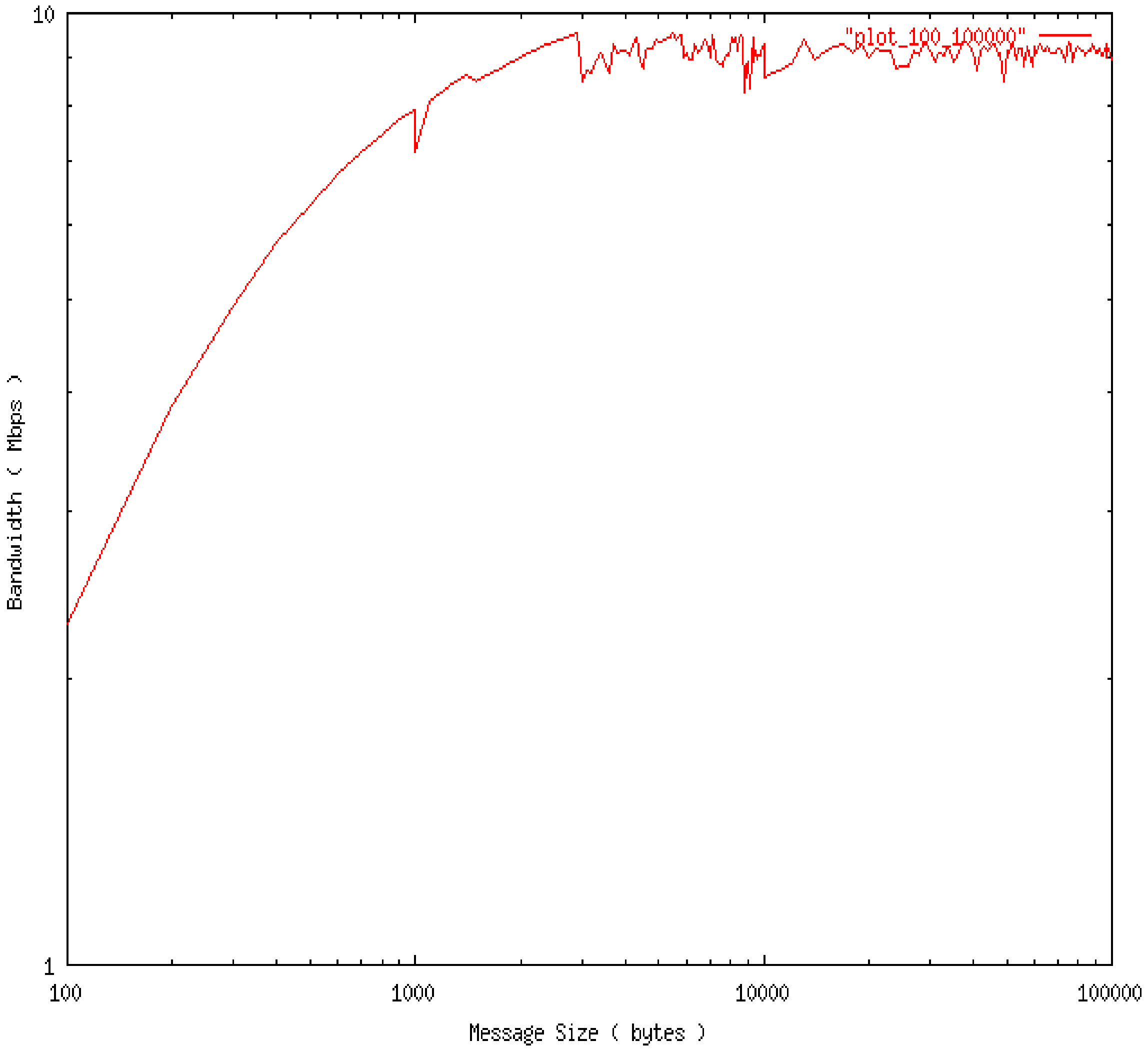

Figure 7.3: The performance of QMP in log scale

Chapter 8

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.

To reiterate QMP's scoring points :

- reduces data copying overheads.
- slashes software checksumming overhead.
- decreases the software latency.
- employs an optimistic flow control mechanism.
- introduces the concept of kernel-user sharable buffers, **qbuf's**
- provides **BSD Socket Interface**

[1]see Chapter 1. Motivation for details on cluster environment assumptions.

Chapter 9

Foreseeing Ahead

The QMP protocol is therefore, an effective alternative to the current generalised TCP and UDP communication protocols. This is just the start. The fact that it co-exists with the previous communication stack is a very useful one. It means that the same kernel provides both routes for data exchange. This opens up several possibilities for the future. Smart applications should be able to make an intelligent choice between QMP, TCP or UDP depending on the assumptions and constraints governing them.

The QMP must be enhanced to include multicasting feature into it . We have designed QMP to be used by PVM like applications where multicasting is a common feature. But currently it uses multiple unicast messages for sending to all the members in the group. So, if we can provide the multicasting facility at the protocol level, then this overhead is saved .The MPI (Messgae Passing Interface) is another standard software for distributed applications. The QMPI interface (QMP tailored interface for MPI) is another prospect for future work.

In effect, the QMP can be used as an all-encompassing protocol coexisting with TCP which will give better performance for certain applications for which it is tailored.

Bibliography

[1] Manish Verma , Tzicker Chieuh "A Low Latency Communication Subsystem"

[2] Hong Xu and Tom W. Fisher "Improving PVM performance using ATOMIC User-Level Protocol.

[3] David Clark , Van Jacobson , John Romkey , Howard Solwen "An analysis of Processing Overhead," IEEE Communications Magazine , June 1989.

[4] Thorsten Eicken , Anindya Basu , Vineet Buch , Werner Vogels "U-Net: A level Network Interface for Parallel and Distributed Computing, " Proc. of 5th ACM symposium on Operating System Principles, Copper Mountain, Colorado. 3-6 , 1995.

[5] D. Clark and D. Tennenhouse "Architectural Considerations for a New Generation of Protocols," Proc. of SICOMM-87, pages 353-359, Aug. 1987.

[6] T. E. Anderson , D. E. Culler , D. A. Patterson, et. al. "A Case for NOW (Networks of Workstations)," IEEE Micro, Feb. 1995, pages 54-64.

[7] J. B. Postel "Internetwork Protocol Approaches," IEEE Transactions on Communica tions, vol. Com-2B, no. 4, pages 605-611, April 1980.

[8] Jonathan Kay and Joseph Pasquale "Profiling and Reducing Processing Overheads in TCP/IP," IEEE/ACM Transactions on Networking, vol.4, no.6 Dec. 1996.

[9] Xpress Transfer Protocol Specification,Version 4.0, March 1995

[10] Moti N. Thadani,An Efficient Zero-Copy I/O Framework for UNIX,Sun Micro,May 1995