IPQ: A Communication System for Distributed Virtual Environments with a Network Protocol using a Semi-optimistic, Sender Initiated Acknowledgement Scheme

Scott Frees and G. Drew Kessler
Lehigh University
5/17/02

Abstract

Virtual Environment (VE) applications are generally comprised of a number of different tasks that are responsible for such things like tracking user motion, rendering screen output, or performing collision detection in the environment. These tasks are of different priority and often require different computing requirements. It is for this reason that VE applications lend themselves to a distributed computing environment. Since producing an effective VE application is greatly dependent on the speed of the system, the efficiency of information passing between tasks becomes a very important concern. It is out of this concern that the Inter-Process-Queue (IPQ) was developed. IPQ works under the idea that many of the messages sent between VE tasks are simply state updates and a task on the receiving end of these messages is only concerned with the most recent updates. By implementing an information passing abstraction called an updateable queue, IPQ automatically drops obsolete and extraneous state updates while en route to the receiving task.

In addition to a detailed overview of the basic ideas and structure of the IPQ system, we present the new networking protocol used by IPQ to communicate with remote machines. We compare two implementations of a reliable UDP protocol using sender initiated acknowledgment schemes. The first implementation is a more pessimistic approach, requiring an acknowledgment to be sent each time a message is received. The second, new and more optimistic approach uses cumulative positive and negative acknowledgment scheme to cut down on network traffic. We present performance results comparing these two protocols along with tests comparing TCP against IPQ using this new protocol.

1. Introduction

Virtual Environment (VE) applications are generally comprised of a number of different tasks that are responsible for such things like tracking user motion, rendering screen output, or performing collision detection in an environment. These tasks are of different priority and often require different computing requirements, which allow many VE applications to lend themselves to a distributed computing environment.

Since producing effective VE applications is greatly dependent on the speed of the system, the efficiency of information passing between tasks becomes a very important concern. These multiple tasks (or processes) could be executing on the same machine, or on separate machines connected by a network. It is out of this concern that we are developing the Inter-Process-Queue (IPQ). IPQ works under the idea that many of the messages sent between VE tasks are simply state updates and a task on the receiving end of these messages is only concerned with the most recent update. By implementing an information passing abstraction called an updateable queue, IPQ is able to automatically drop obsolete and extraneous state updates while they are en route to the receiving task. By doing this, IPQ guarantees that each time a receiving process looks for an update message it will find the most relevant and up-to-date one available. It will
also be shown that by eliminating theses obsolete messages IPQ can decrease a system's communication latency.

Although some of the work that went into the development of the original implementation of IPQ is presented here (section 3), this paper is mainly focused on a modification in the way in which IPQ sends individual packets over a network. The original protocol, based on UDP, serializes each individual packet sent by assigning a sequence number to it. Each time a process sends a message packet it waits for an acknowledgment from the receiver. By requiring an acknowledgment for each individual packet, IPQ ensures reliable, in-order delivery. However, there is an inherent drawback in this approach in that the common case, successful packet delivery, requires confirmation, which consumes valuable time. A new implementation of IPQ's network protocol is presented here that takes a more optimistic approach by allowing a sending process to send multiple packets before requesting an acknowledgment from the destination process, thus eliminating the need to wait for confirmation on every message sent. When an acknowledgment is requested, a receiver either sends a positive acknowledgment (ACK) or a negative response (NACK), which will inform the sending process which packets have been lost since the last acknowledgment. By cutting down on the time a sending process waits for acknowledgments we are able to increase the performance of IPQ while maintaining the reliable delivery and processing order of data messages.

This new protocol was implemented under the Windows 2000 environment. Since IPQ was originally implemented for UNIX, we also present some of the modifications of IPQ that resulted from the port to Windows. This topic is discussed in section 4, while an in-depth discussion of the new, more optimistic network protocol along with a quantitative analysis of the results are presented in section 5. In section 6 we will present a quantitative comparison of the performance of IPQ versus TCP using a benchmark application.

2. Related Work

In the computer science field, one of the most exciting and active areas is in Virtual Reality (VR) and the study of how Virtual Environments (VE's) can be used. One of the areas that hold the most potential for VR involves collaboration between multiple people at multiple sites, working together in a common VE. These systems, often referred to as Distributed Virtual Environments (DVE’s), present a number of unique problems that need to be solved and are the subjects of much research. One of the more important issues in DVE’s is the method of communication between sites across a network (both LAN and WAN). VE’s have a unique combination of needs in that there is often a large amount of data being sent between sites and this communication often must be very close to “real time” for the system to preserve a sense of presence for the users. When network delay becomes a problem users have been observed to lose their sense of immersion and become confused as to the cause of the inconsistencies observed in “world”; often this can cause users to lose trust in the system or attribute the inconsistencies to problems caused by things other than network delay [Fraser-00].

There are a number of VR systems that support collaboration among users at multiple sites, with a wide variety of methods of communication ranging from simple TCP connections, reliable and unreliable UDP, and multicast. One such system is MASSIVE-3 [Greenhalgh-00]. MASSIVE provides support for data consistency and interest management between multiple users inside possibly very large worlds comprised of 'locales’. This system uses TCP to implement multicast, mainly for TCP’s
maturity as a reliable protocol and its wide availability (as opposed to traditional IP multicast [Greenhalg-00] [Roehl-95]). In contrast, CAVERNsoft G2 [Park-00] provides a programmer with high-level and low-level support for networking which allow the programmer to decide what type of protocol is desired (TCP, UDP, and various high level abstractions such as parallel TCP). Another system, NVE [Joslin-00] uses both TCP and UDP for network communication. NVE implements both reliable and unreliable UDP for data transfers associated with streams and updates, while utilizing TCP for larger file transfers and control messages. Bamboo [Watsen-98] is a multi-platform VR toolkit that utilizes a foundation class library (ACE) to provide a layer of abstraction above the operating system. Bamboo uses this library to provide unicast (TCP and UDP) and multicast support to programmers. Finally, the Octopus [Hartling-01] system is a cross platform API for constructing Collaborative Virtual Environments (CVE’s). The Octopus library uses unreliable UDP for network communication due to the protocol’s low overhead costs and the fact that object update messages are sent so frequently that a dropped message has a minimal effect on the consistency of the system (it should be noted that implementing reliable UDP is listed as a goal in the authors’ future work).

IPQ [Kessler-96] is a tool for DVE’s, but its goals are different from the systems mentioned above. The systems discussed above are full-featured VR systems, with networking being only one aspect of the overall system. IPQ is not a general purpose VR system however; its purpose is to provide a communication system and framework between multiple applications, specifically tailored for the unique needs of DVEs. IPQ, and/or the ideas behind the system [Jeffay-01], can be incorporated into existing systems like those discussed above. Since IPQ does not have all the responsibilities of the systems discussed, more attention has been paid to the actual network protocol and the following discussion presents the concepts behind the different networking schemes that have previously been developed.

At the core of the communication between remote machines is the transport layer. The most popular transport protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). These protocols sit on opposite ends of the spectrum of the numerous protocols (TP4, NETBLT, RTP, XTP, etc)[Sami-99] that have been developed.

TCP implements a reliable transmission of messages at the transport layer level, virtually invisible from the programmer’s perspective. With this reliability, however, comes a price; there is a considerable amount of overhead involved with each TCP transmission. In addition to this overhead (and at times due to it [Clark-82]) network delays can become unsuitable for many applications [Roehl-95][Kessler-96], including DVE applications.

UDP is unreliable, messages are not guaranteed to reach their destinations and may arrive out of order. Verification, and retransmission if necessary, is up to the application. UDP's advantage is that there is very little overhead involved and congestion is often avoided. [Roehl-95].

Comparing UDP with TCP, we see that part of the problem with using TCP in DVE applications is the fact that TCP is invisible from the programmer's perspective; removing the possibility of adapting the protocol to suit the needs of the particular application (i.e. the needs of DVEs). It is for this reason that IPQ uses UDP; IPQ retransmits intelligently, in that instead of retransmitting the same update message upon packet loss, it will attempt to find a more up to date message to send instead.
Implementing reliable UDP is the subject of much research and there is much to be learned from studying the low level implementation of TCP [Allman-99][Clark-82][Sami-99]. Among systems that use unicast, as opposed to multicast, there are few that have chosen to primarily use UDP, instead opting for TCP. However, reliable multicast implementations are generally based on unreliable UDP [Brutzman-95]. This research is informative when implementing UDP unicast as well, since many of the algorithms that go into implementing reliable multicast can also be used in reliable unicast.

There are two main issues to be considered when implementing reliable UDP: flow control, and acknowledgement schemes.

One such commonly used flow control scheme, found in TCP, is the window-based flow control [Mishra-98][Zhang-01][Clark-82][Sami-99]. This algorithm uses a congestion window to determine how many unacknowledged messages can be allowed in transit, regulating how many messages a sending process will send before receiving acknowledgements from the receiver. This congestion window is varied throughout the life of the connection in order to handle changing network conditions. In our new network communication protocol for IPQ we use a similar concept, except that this ‘window’ of unacknowledged messages is static with its size known by the sender and the receiver.

Most reliable UDP protocols incorporate the notion of sequence numbers, which establish an ordering of messages sent over the network. Each message has a corresponding sequence number; verification of it reaching its destination is handled through an acknowledgment scheme. There have also been a number of acknowledgement schemes proposed, many corresponding to multicast protocols. Two classifications are defined in [Levine-98]: sender initiated and receiver initiated. Two general types of acknowledgements (ACK’s) are also defined [Rhee-99][Mishra-98][Levine-98]: positive acknowledgements (PA) and negative acknowledgements (NA). Generally, protocols using PA schemes detect message loss when the receiver hasn’t returned an ACK for the message after a certain timeout period. In NA protocols however, the receiver will explicitly request retransmission by sending a negative ACK to the sender when gaps between sequence numbers are detected. There have been variations on how and when these acknowledgments are sent in order to further reduce network congestion and reduce the processing time involved with handling each ACK. The concept of block acknowledgement was described in [Brown-89]. This reduces the number of PA’s sent by the receiver by constructing the PA with two sequence numbers that represent a range of messages that have been received successfully. Another scheme is a variation on the NA protocol, involving cumulative NA’s, or CNAK’s. [Poo-01] This scheme reduces the number of ACK’s by constructing NA’s with a series of missing sequence numbers. Our communication protocol in IPQ draws from these concepts by combining them. After a predetermined number of messages have been sent, the receiver either sends a positive acknowledgement with the last received sequence number (the last ACKed sequence number implicitly defines the lower bound of the range) or a NACK, consisting the missing sequence numbers between the latest sequence number and the last ACKed message.
3. Initial Design of IPQ

In response to the unique needs of distributed virtual environments, we have developed a network communication framework was developed which is called the Inter-Process Queue (IPQ). There are two concepts that IPQ utilizes to provide for quick and efficient communication between remote or local processes: message type differentiation and updateable queues. In this section, we will discuss these two concepts, a description of the implementation, and give a comparison of the results obtained using the IPQ prototype versus TCP.

3.1 Message Types

Communication between tasks in a virtual environment application is achieved through discrete messages that can naturally be broken down into two categories, state update messages and event messages.

3.1.1 State Update Messages

State update communication usually involves a relatively steady stream of messages that describe some attribute of an entity that changes with time. An example of this can be a VR application sending position coordinates of a thrown ball to another VR application, or a head mounted display communicating with a rendering process. In both these cases, the most recent state is of the most value. There are a number of problems that arise when sending frequent state messages. One problem arises when using a protocol that guarantees message delivery, such as TCP. When state updates are sent frequently, the messages can become backed up due to the automatic retransmission of lost packets, causing the receiving application to process and discard obsolete messages until it gets to the most recent one. IPQ resolves these issues by using an un-guaranteed message delivery (UDP) protocol. Each message is sent and resent until an acknowledgment is received from the receiver or until a new state update describing the same entity is ready to be sent. By replacing obsolete state messages that have not been acknowledged, IPQ is able to resolve some of the overflow problems associated with these type messages.

3.1.2 Event Messages

Of course, not all messages sent between VR applications are state updates; some messages represent specific events that occur in an environment, or command messages such as a keystroke. These “event” messages must be delivered successfully and in the order they occur. IPQ handles these messages by resending them until they are acknowledged in the same fashion as sending an update message; the major difference being that no other messages are sent until the message is acknowledged or the connection is deemed to be lost. The reception of an event message also archives all previously received update messages, preventing them from being overwritten in the queue (see below discussion on updateable queues).

3.2 Updateable Queue Abstraction

One structure that facilitates one-way, FIFO communication of messages is a queue. A simple queue can clearly handle the passing of event messages but there are modifications necessary in order to handle state updates more effectively. To handle passing these update messages, an updateable queue abstraction was developed.
Messages passed through the queue are comprised of a key value and data. Messages that enter the queue will replace messages already in it if the keys match. There are two conditions where messages are not overwritten however.

1. Messages with a special key (zero in this implementation) value do not replace any information. Messages with these special keys represent event messages.

2. Once a message with this special key is stored in a queue, all messages earlier in the queue are effectively archived, meaning they cannot be overwritten by any newly arriving packets.

It is this scheme that allows a single queue to handle both event messages and update message. Figure 1 illustrates a few of the key scenarios involved in this process.

![Figure 1: Different types of messages being added to an updateable queue.](image)

**3.3 Implementation**

The goal of IPQ is to provide a quick and efficient network protocol for distributed virtual environments. IPQ was originally developed under UNIX and the system consisted of three separate heavyweight processes: the Task process, Communications process, and the Send process.

**3.3.1 Task Process**

The Task process represents the parent process or the actual application process. The Task process uses the Communications process and the Send process to send and
receive messages from other applications using IPQ on remote machines. For remote connections the Task process sends and receives event and update messages via updateable queues as shown in Figure 2.

When two tasks on the same local machine wish to communicate, the communications and sending processes are not needed for the connection and the two task processes are directly connected by updateable queues. This is illustrated in Figure 3. It is also important to note that the fact that task processes communicate differently depending on whether they are local or remote is invisible to the programmer using IPQ’s API.

![Figure 2: When communicating with a remote IPQ process, the task process uses the send and communication processes, which are connected by updateable queues.](image1)

![Figure 3: Two IPQ processes located on the same physical machine, they communicate directly with each other via two updateable queues.](image2)
3.3.2 Communications Process

The Communications process works to relieve the VR application of the responsibility of handling incoming messages from remote machines. After a remote connection is established, the Communications process listens on the associated sockets for messages. Once a message is received and decoded, it is placed on the appropriate queue, which the task application then reads. By creating a separate process to handle incoming network events, the task process is free to perform other responsibilities and can read information from the incoming queue at its “leisure”. In addition, the Communications process can continuously wait and process incoming messages, helping prevent the buildup of messages often caused by a receiving process performing all its other responsibilities while trying to process messages.

3.3.3 Send Process

The Send process’s purpose is basically the same as the Communications process in that it works to relieve the task process of the responsibility of sending messages to remote machines. When a new remote connection is established an updateable queue is created between the Task and Send process. The task simply places messages on this queue while the Send process reads the other end and takes care of sending the messages over the network and implementing the actual network protocol (sending messages and receiving acknowledgements). Once again, another advantage of this is that since it is a separate process, the Send process can continuously check the outgoing queues and communicate with remote process without needing to perform any other actions.

3.4 Programming Interface

IPQ provides the programmer with a simple but powerful set of routines to communicate with other VR processes. These routines require the programmer to have little knowledge as to the network protocol implemented or the actual workings of the updateable queues.

3.4.1 Opening a connection

The connection process between two applications starts out with a client-server organization (peer to peer once the connection is made). One process (server) must first advertise its availability to the network by calling IPQ_advertiseService(). This stores the name of the service the application is providing along with the appropriate port number in a local database called the name server, which is available to any remote application by utilizing a known port number. Once the service is advertised, the process can then call IPQ_getNewConnection() to allow a “client process” to connect to it. The client process connects to an advertised service by retrieving the port number and service name from the name server. It then calls IPQ_openConnection() with the appropriate parameters (machine name, service type, and port). Once the connection is established the idea of a server and client is no longer valid, each task process can send and receive messages between each other freely. This use of the name server to pass out port numbers for advertised services allows for location independence among the machines running IPQ, while still allowing peer to peer connections to be established.
3.4.2 Reading and Writing Messages

IPQ provides a number of routines to receive and send information to other processes. When sending information to another process a programmer first must call IPQ_beginWrite or IPQ_beginGroupWrite with the connection specified as a parameter. After this, the programmer can begin writing information by using the following routines:

- IPQ_writeChar
- IPQ_writeInt
- IPQ_writeFloat
- IPQ_writeArray
- IPQ_writeXDR

The first three routines allow the programmer to write specific primitives while IPQ_writeArray allows the programmer to write an array of any of these data types by passing a procedure for writing the primitive as a parameter. The fifth write function, IPQ_writeXDR allows the programmer to write any programmer defined data structure by passing a procedure to the function that can encode the data structure appropriately.

Finally, the programmer must call IPQ_endWrite, which actually writes all the data to the outgoing queue. Receiving information from another process is performed by using similar routines, which are shown below.

- IPQ_readChar
- IPQ_readInt
- IPQ_readFloat
- IPQ_readArray
- IPQ_readXDR

Just like when writing data, the programmer must begin a reading “session” by calling IPQ_beginRead and must finish reading from a connection by calling IPQ_endRead.

3.5 Results

A representative application [Kessler-96] was developed in order to perform comparisons between TCP and IPQ and the results were quit encouraging. This application consists of two processes, one sends 1000 messages and another receives the messages. Each message consists of the (local machine) time, an ID value, and an array of 16 floating point numbers, which represent the 3D position and orientation of arbitrary entity in a VE. The application tracks the time it takes for a sending process to send all 1000 messages along with the time between sending and receiving (and processing) the last message (message lag). An additional independent variable, receiver delay, is also introduced to simulate the time a receiving process spends on other tasks aside from communication (e.g. rendering, etc.). The lag time between messages sent over a network was improved and remained constant as the time the task process took performing other responsibilities was increased. This was in contrast to the lag times created when using TCP, which increased substantially when the task process spent more time performing other things. This result is important because VR applications are generally very processor intensive, which generally causes the process of receiving messages to fall behind the sending process. Another result was that the number of state update messages sent and what was actually received by the other application was reduced. This result is expected since one of the main purposes of IPQ is to eliminate obsolete messages before the receiving task process gets them, freeing it...
from trying to determine the most recent information. For a quantitative summary of the results of our first IPQ implementation we refer the reader to [Kessler-96].

4. Porting IPQ From Unix to Windows

When implementing IPQ for Windows we made the decision to take advantage of threads instead of using separate, heavyweight processes for the Task, Communication and Send processes. This suggests a modified way of communicating between the Task, Send and Communication threads since the threads now belong to the same global address space.

4.1 Communication between the Task, Send, and Communication threads

The three processes of IPQ under Unix communicate in two ways: through shared memory (the queues) and via pipes when sending command messages to each other. These two methods are modified in the windows version to take advantage of the Windows API.

4.1.1 Shared Memory Queues

The original Unix version of IPQ uses queues located in shared memory for communication between the Task, Communication, and Send process. Also, when two IPQ applications run on the same local machine, a shared memory queue is used for direct communication between each Task process. For Unix this is the most efficient and logical way of inter-process communication when the principals of IPQ are taken into consideration. Windows however, has allowed these three processes to run as threads under the same process. This fact allows for the queues involved in communicating between threads to be taken from shared memory space and into a global address space.

There is a benefit to not using shared memory for these queues aside from eliminating the need to create and attach to the section each time a thread wants to use the queues. Where as Unix uses shared memory frequently and provides quick and efficient function calls for its use, Windows does not naturally lend itself to using shared memory for these purposes. In fact, the function calls used to implement shared memory in Windows are the same as those used for file I/O operations. This suggests that Windows may not perform shared memory operations as efficiently as Unix. Since there is no real need to use shared memory for communication between threads and because the queues are the lifeblood of IPQ and are used so frequently, any increase in efficiency in this area is quite valuable.

For a remote connection, the Task process communicates to the connection by placing and reading from the incoming and outgoing queues as seen in figure 2. However, when a local connection is made the Send and Communication threads no longer play a role and a queue is made directly between the two Task threads of the separate applications. For this type of queue, the best solution remains the shared memory queue. The introduction of two types of queues adds complexity to IPQ's internal procedures involved in opening, reading, and writing from queues. The solution used provides a mapping of the queue identifiers to determine whether the queue is located in shared memory (local connection) or in the global address space (remote connections). The queue data structure is shown below.
 typedef struct srShQueueStruct {
    char              id
    int      memID
} srShQueueStruct, *srShQueue;

Here, the primary identifier is the memID; the use of the id field remains the same (the id field identifies a reading queue versus a producing). In the Unix implementation memID was a unique integer assigned by the operating system for the block of shared memory containing the queue. The goal of the redesign of the queues was to allow the memID to identify a block of shared memory or a pointer to a block of memory in the global address space while limiting the changes to the upper levels of IPQ. To provide for this, when IPQ initially creates the write-enabled end of a queue structure it calls:

srShQueue createSRSQProducer(unsigned short dataSize, int shared)

If the shared parameter is set to true this function will return a queue residing in shared memory (local connection), otherwise the queue is created in the global address space. The returned srShQueue's memID field can then be used to access the queue in memory when creating the read-enabled end of the queue or during reading and writing. To create the read-enabled end of a queue, the following routine must be called:

srShQueue createSRSQReader(int memID)

Here, the memID parameter is the memID of the queue returned by the createSRSQProducer function. When reading and writing to queues, IPQ uses the following routines:

addToSRShQueue(srShQueue queue, void *data,...)
removeFromSRShQueue(srShQueue queue,...)

All three of these routines first examine the memID field in the provided srShQueue structure (or directly passed as an integer) in order to attach to the appropriate memory block. A mapping is used to determine if the block is in shared memory or not. A memID assigned an integer less than MEM_ID_MAX implies that the memory was created in shared memory. Since Windows uses strings to identify shared memory, the integer is translated into string form and then the shared memory is accessed. A memID greater than MEM_ID_MAX however, tells the function that the memory is in the global address space. An index is then created by subtracting an offset from the memID. This index is used to look up a pointer in a global lookup table to the memory block that was assigned when the queue was created.

By assigning the memID appropriately in the createSRSQProducer routine and performing this mapping in the various routines of the queues, the idea of different types of queues is almost completely transparent to the rest of IPQ, with the only exception being the initial creation of the producing end of the queue.

4.1.2 Pipes

The initial implementation of IPQ used twelve pipes to communicate internal command messages between the Task, Communication, and Send processes.

ipq_sendFromComm ( ipq_comm2Send
Pipes are used quite efficiently in Unix but just as shared memory, the Windows API does not provide the same functionality. At first glance, it seems that pipes between the three threads may be unnecessary because the threads share the same address space. A possible implementation was designed using a linked list structure in global memory to provided for the passing of commands between threads. Although this implementation was indeed implemented at first, there were a number of difficulties involved with memory management; specifically synchronization problems involved with allocation and de-allocation by the two threads were causing the system to become unstable. It was decided to use anonymous pipes to avoid these difficulties. Using anonymous pipes in Windows does not map directly from Unix however because Windows does not naturally provide a method for efficiently waiting on pipes in a similar manner as the Unix select command. For this reason a data structure was designed to provide the needed functionality.

```c
typedef struct waitablePipe {
    HANDLE readEvent;
    HANDLE readPipe;
    HANDLE writePipe;
    HANDLE mutex;
    char * name;
} waitablePipe;
```

The `readPipe` and `writePipe` handles represent two anonymous, one way pipes. The fusion of the read and write pipes allows for six “waitablePipes” to be used in IPQ.

```c
CommToTask
CommToSend
SendToTask
SendToComm
TaskToSend
TaskToComm
```

The `readEvent` field is an event object that is set when data is on the pipe, ready to be read (analogous to `fd_read`). The `mutex` field allows for synchronization while setting or resetting this event object. IPQ procedures can efficiently wait on the read end of any of these “waitable” pipes by passing the `event` field to one of the Windows API wait functions such as `WaitForSingleObject()`.

To make the use of these pipes more clear, the pseudocode for reading and writing are presented.

```c
readWPipe(waitablePipe * pipe, void * data, ...)
{
    /*Read Data From Pipe */
    ReadFile(pipe->readPipe, data, ...)
    P(pipe->mutex) /* Enter pipe's critical section*/
}```
/* Check to see if any more messages are on pipe */

if (peekSizeRead = PeekNamedPipe(pipe->readPipe) == 0)
    /* Nothing left on the pipe, reset the event object */
    ResetEvent(pipe->event)
    V(pipe->mutex)  /* Exit pipe's critical section */

return

writeWPipe(waitablePipe * pipe, void * data, ...)
{
    P(pipe->mutex)  /* Enter pipe's critical section */

    /* Write Data to Pipe */
    WriteFile(pipe->writePipe, data, ...)

    /* There is now something to read from the pipe, signal it*/
    SetEvent(pipe->event)
    V(pipe->mutex)  /* Exit pipe's critical section */

}

4.2 Results

This Windows implementation was tested using the same type application as described in section 3.5. Our results were consistent with the results found in the original Unix implementation. Message lag times and send times followed the same important trends in that they remained constant as receiver delay was increased. Qualitative results comparing this implementation (pessimistic) with our new protocol implementation are presented in section 5.2.2. That said, we did find that when sending update messages the overall throughput decreased. A possible reason for this further drop in throughput is the way Windows schedules the three different threads. Throughput is maximized when the three threads receive more equal processor time; therefore the throughput of the system may be largely affected by the different scheduling mechanisms used in Windows and Unix. For our benchmark application, messages were continuously being generated (by the Task thread), preventing the send thread from sending most of the messages before they were overwritten in the outgoing queues (update messages). In order to better simulate an application sending updates to remote machines we ran another test case with the same application, although this time the application waited for varied times between generating new updates. This time between update generations represents the time a real VR system would spend doing other jobs in a real-world setting. The effects on the overall throughput are summarized in figure 4.

These results are only included here to give the reader an idea of how many update messages actually get through to the receiving process’s task thread. The throughput however is not really our concern in these types of situations, we are more concerned with making sure the receiver gets the most relevant and recent updates when it gets a chance to look for them. The messages that are not getting through to the receiver are in fact precisely the ones that are useless to the receiver (obsolete) and in later sections we will show how this lower throughput enables IPQ to provide lower latencies even when the sender’s rate is much higher than the receiver’s.
5. A Modified Protocol

In this section we first describe the network protocol used in the prior implementation of IPQ and then present a more optimistic variation, which will be discussed in detail. Both protocols are implemented using UDP sockets with added support to ensure reliable and in-order delivery of messages to their destination processes.

5.1 A Pessimistic Approach

The first network protocol implemented for IPQ used a “lock step” approach for each message sent. Each packet sent by a process was constructed from the data to be sent, along with a unique sequence number, used to preserve the order of the messages. Each packet sent by a process required the receiving process to reply with an acknowledgment, which was the negation of the sequence number. A sending process would wait for the correct acknowledgment for a specific period of time, which is determined by an exponential back off algorithm [Stevens-90]. If the correct acknowledgment is not received within the given time period, a new time-out period is calculated and the sending procedure returns, indicating the message had timed out. The receiving process in turn would process any message with a sequence number greater than the last received sequence number, dropping any packets with sequence numbers less than the last acknowledged sequence number. In any event however, the receiver would always send an acknowledgment to each packet received in order to signal the sender to proceed. Packets that did not receive acknowledgments continued to be resent until a correct acknowledgment was received or it was determined that the connection was been lost.
Although this method of communication ensured reliable, in order message delivery between two remote processes, the acknowledgment scheme used has two drawbacks. The first drawback is clearly increased network traffic; each message to be sent to a remote machine requires a minimum of two distinct network packets to be sent, one containing the sequence number and data, and another sent by the receiver in the form of an acknowledgment. A second drawback is that a sending process must wait for an acknowledgment each time it sends a message. This overhead can be quite costly.

The fundamental cause of these limitations is that this protocol takes a somewhat pessimistic approach toward each packet. The sending process works under the expectation that every packet may be lost, and thus works to assure each packet is received individually, for each send. Another approach is a more optimistic one, where a sending process works under the assumption that most packets do reach their destination. With this approach, the sender needs only to verify that packets have reached their destination periodically.

5.2 An Optimistic Approach

A solution to the limitations described above is to only request acknowledgments for a subset of the packets sent over the network. We must do this however, without sacrificing reliability or the ordering of packets. We have developed a scheme that uses sequence numbers for each packet in a slightly modified way and introduces two distinct types of acknowledgments, an \texttt{ACK} and a \texttt{NACK}. In this approach, the sending process will decide when an acknowledgment is necessary and inform the receiver by negating the associated sequence number before sending it. The sender will request an acknowledgment after a predetermined number of packets have been sent to the destination without an acknowledgment request (in our implementation, this number is 5) or if the specific message is determined to be critical and needs an immediate response. Upon receiving packets, the receiving process checks the sequence number to see whether it is requesting an acknowledgment (a negative sequence number). If so, the receiver determines if it has received each sequence number since the last successfully acknowledged sequence number. If it has, it responds to the packet by sending an \texttt{ACK} with the sequence number back to the sending process. If it determines that there are missing packets, it constructs a \texttt{NACK}, which contains a list of the missing sequence numbers, and forwards it to the sender.

```c
struct ACK {
    int ack = ACK;
    long int seqNum;
};

struct NACK {
    int ack = NACK;
    long int seqNums[PACK_MAX];  // PACK_MAX = 5
};
```

When the sending process requests an acknowledgment from the receiver it waits for the response in a similar fashion as in the first implementation. If it receives an \texttt{ACK} response, it simply proceeds as normal. In the event that a \texttt{NACK} is received, it will need to resend the packets missed, requesting a response on the last missing packet specified by the \texttt{NACK}. If no response to the original acknowledgment is found from the receiver, it is treated as if a \texttt{NACK} was sent with the most recent sequence number missing, thus the current sequence number and data are again negated and sent. Figure 5 illustrates the old protocol (5a) and two possible scenarios in the new protocol (5b, 5c).

It should be pointed out here that messages sent by the sending process aren’t the only packets that can be dropped; the acknowledgements sent by the receiver could be lost as well (this can happen in both the pessimistic and optimistic approaches). Our
protocol handles this occurrence in a relatively seamless way. In the event the receiver sends an acknowledgement to the sender, but the message gets lost, the sender will retransmit the last data and sequence number (the same exact step it would take if packet actually never reached the receiver). Once the receiver gets this retransmission, it will immediately see that its sequence number corresponds to a previously acknowledged packet and will send the same acknowledgement (either ACK or NACK) back to the receiver again. This simple solution is able to handle this type of event because a sender requesting an acknowledgement for a sequence number $N$ can never send a packet with sequence number $N+1$ until receiving the acknowledgement to $N$ from the receiver. The implementation of this scenario is shown in section 5.2.1.2 when the receiving algorithm is presented.

By only requiring responses from the receiver periodically, we can cut network traffic down by a factor of $2N/(N+1)$, where $N$ is the number of packets sent between acknowledgment requests assuming that no packets are dropped. Fortunately, even when packets are dropped, network traffic will continue to be lower than the first implementation because of the ability to resend a number of missed packets without requesting responses from the receiver. Similarly, the amount of time a sending process spends waiting for acknowledgments is clearly reduced. In the worst case scenario (illustrated in figure 5d), when a single message gets lost a number of times repeatedly, the protocol’s efficiency will degrade to being approximately equal to the pessimistic approach, forcing the sender to send one message and wait for a response from the receiver.

These performance gains, however, introduce considerable complexity in order to preserve reliable and in order delivery. Specifically, there is much more bookkeeping involved on both ends in order to keep track of the received and missed sequence numbers. This complexity is now described in the implementation section.
Figure 5a: Sending packets (sequence numbers $x_0, x_1, x_2$) using the pessimistic approach. The packets with dotted lines represent the receiver's acknowledgments.

Figure 5b: Packets being sent using the more optimistic approach. Here every fifth packet is acknowledged.

Figure 5c: Packets sent with the optimistic approach. Here, packets $x_1$ and $x_3$ were dropped and the receiver and sender recover by processing the

Figure 5d: In the worst-case scenario, where a single message continues to get lost repeatedly, the algorithm temporarily degrades to a pessimistic approach, waiting for an acknowledgment to $x_2$ until a response is gotten from the receiver (or until the sender determines the connection is lost).
5.2.1 Implementation

For clarity the description of the implementation is now broken up into two sections addressing the sending and the receiving procedures respectively.

5.2.1.1 Sending

The process of sending messages to remote machines is broken up into two levels, an upper level that is integrated into the main routine of the send thread of IPQ and a lower level consisting of UDP routines. The upper level is in control of the bookkeeping and sequence number assignment associated with each message and the lower level is responsible for sending a message over a socket, and if needed, receiving an acknowledgment and returning it to the upper level. The lower level is relatively straightforward in that it simply hides the socket level details from the upper level.

The upper level is implemented in the IPQ function forwardData, which is called regularly by the send thread. This function sends one message to each remote connection currently opened each time it is called, with the procedure for sending to each connection being identical. Each remote connection has a data structure, remoteOutConnectStruct, associated with it.

```
typedef struct remoteOutConnectStruct
{
    ipq_udpOutPort outPort;
    srShQueue  fromTask;
    void    *sendData;
    short int  sendDataLen;
    unsigned long  sendDataKey;
    long int  sendDataSeqNum;
    long int  lastSuccessSeqNum;
    long int  seqNum;
    long int  waitingOnSeqNum;
    int   recovering;
    int   lastMissing;
    ipq_udpOutPacket outMsgs[SET_MAX];
    /* SET_MAX = five in our implementation */
} remoteOutConnectStruct, *remoteOutConnect;
```

The above structure holds all the state information required for an individual remote connection. The first data member, outport, holds the associated destination socket and all the information needed to send messages to it. The fromTask member represents an IPQ queue (described above) used to communicate with the Task process (one way communication, Task -> Send). Data is taken from this queue and sent over the network to its remote destination. The outMsg list is used to store complete copies of each message until it has been successfully acknowledged in an ACK, since it may need to be resent later if the receiver sends a NACK. (Note that a reference to the data taken form the queue is stored, not a separate copy)

```
typedef struct ipq_udpOutPacket
{
    long int seqNum;
    void  *data;
    int   key;
    int   size;  int missing;
```
For each remote connection, there is also a notion of a packet set, which is a range of packets starting with the lastSuccessSeqNum and up to the next packet that will request an acknowledgment (typically lastSuccessSeqNum + 5). Upon opening a new remote connection, some of the state variables must be initialized: lastSuccessSeqNum, waitingOnSeqNum, recovering, and lastMissing are all set to zero, each outMsg is initialized to not missing, and finally, seqNum is set to 1.

Figure 6 shows the general algorithm that is followed in sequence for each open remote connection. This algorithm starts in State A as soon as a remote connection is opened. One aspect of the algorithm is left out of figure 6 however; since only one message is sent to each remote destination each time the send thread calls forwardData, the state diagram really has two entry states. Each time forwardData begins working with a remote connection, it first looks at its remoteOutConnectStruct’s recovering flag. If this flag is set, forwardData begins executing the algorithm in State F, while starting in State A under any other conditions. The following discussion describes how each state works, along with how the following state is determined.
State A:
This state begins by getting new data from the `fromTask` queue, using the function `removeFromSRShQueue`. It then follows the procedure outlined in to pseudocode below:

```plaintext
if (fromTask !empty)
{
    Fill in sendData, sendDataLen, and sendDataKey with information read from queue (fromTask).

    if (seqNum - lastSuccessSeqNum >= 5)
    {
        // time to request acknowledgment
        requestAck = TRUE
        sendDataSeqNum = -seqNum  // signals ack request
        waitingOnSeqNum = seqNum
    }
    else
    {
        sendDataSeqNum = seqNum
    }

    Save record of sendData, sendDataLen, sendDataKey, and sendDataSeqNum in outMsg[seqNum - lastSuccessSeqNum + 1] in case a retransmission is needed later

    sendMessage();
}
else
{
    Exit the algorithm, move on to next remote connection
}
```

If there was no data, `forwardData` will simply go on to the next connection. However, when there is data ready `forwardData` performs the bookkeeping described above and calls `sendMessage()`, which implements most of States B and C as well as part of State F.

State B:
At the end of State A, `sendMessage` is called, which sends the algorithm into State B. In this state, `sendMessage` takes `sendData, sendDataLen, sendDataKey, and sendDataSeqNum`, constructs a packet from them, and sends it to its destination. After sending the packet, the `requestAck` flag is examined; if set, the algorithm continues on to State C where it will wait for the acknowledgement.

If the flag was not set, control is simply returned to `forwardData`, which increments `seqNum` and then moves on to the next remote connection. The next time `forwardData` works with this remote connection it will begin the algorithm at State A.

State C:
If an acknowledgement is needed, the `sendMessage` routine will wait for an acknowledgment for a period of time and then construct and return (to `forwardData`) a
udpStatus data structure from the receiver’s response (or lack of one). This structure holds all the necessary data needed for forwardData to determine what to do next.

```c
typedef struct udpStatus {
    int status;
    int numDropped;
    int droppedSeqNums[];
} udpStatus;
```

Upon examining the udpStatus structure, forwardData either moves to State D (udpStatus.status = ACK) or on to State E (udpStatus.status = NACK or TIMEOUT).

**State D:**
If the algorithm reaches State D, the udpStatus.status field has been set to ACK, which means the receiver has received all the packets in the set. Consequently, the remoteOutConnectStruct’s internal variables are reset as shown below:

```c
lastSuccessSeqNum = seqNum
recovering = FALSE
seqNum ++
for (i = 0; i < 5; i++)
{
    outMsg[i].data = NULL
    outMsg[i].missing = FALSE
}
// other outMsg fields are just overwritten when they are used again
```

After resetting these fields, forwardData moves on to the next remote connection. The next time forwardData processes this remote connection it will begin the algorithm in State A.

**State E:**
If the algorithm reaches state E, the udpStatus.status field has been set to NACK or TIMEOUT, which means either the receiver has indicated that it has missed packets in the current set (NACK) or a response was never received in State C (TIMEOUT). The following pseudocode represents what happens in this state:

```c
if (udpStatus.status = NACK)
{
    recovering = TRUE
    lastMissing = last sequence number in udpStatus.droppedSeqNums list
    for (i = 0; i < 5; i++)
    {
        if (outMsg[i].seqNum in udpStatus.droppedSeqNums list)
        {
            outMsg[i].missing = TRUE
        }
    }
}
```
else  // TIMEOUT
{
    CalculateNewTimeoutValue(remoteOutConnectStruct)
    if (CalculateNewTimeoutValue signals "connection lost")
    {
        close remote connection
    }
    else
    {
        recovering = TRUE
        lastMissing = waitingOnSeqNum
        outMsg[waitingOnSeqNum].missing = TRUE
    }
}

After the above bookkeeping is performed, forwardData moves on to the next remote connection. When forwardData begins working on this remote connection the next time however, it will begin the algorithm in State F.

**State F:**
As mentioned earlier, the first thing forwardData does when it begins working on a remote connection is look at the recovering flag. If the flag is set, it begins that algorithm in State F.

Here, rather than looking for new data on the fromTask queue, the sendData* fields are set according to the first outMsg in the list that has its missing flag set to true. The copy of the packet is kept in the outMsg list in case this send fails as well, although its missing flag is reset to false. If the sendDataSeqNum now equals lastMissing, (set when a NACK is received) it is negated, requestAck is set to true, and waitingOnSeqNum is set to sendDataSeqNum, since this is the last missing packet and therefore must be acknowledged.

It is important to note here that instead of combining all the missing packets we send each one individually. Although this may seem more efficient, sending larger packets over a network that is already dropping packets could make the situation even worse.

forwardData’s next calls sendMessage(), just like in State A. If an acknowledgement is needed, the algorithm once again proceeds to State C.

If no acknowledgement was necessary (requestAck = FALSE), sendMessage returns and forwardData moves to the next remote connection. The next time forwardData works with this remote connection it will begin the algorithm in State F again since the recovery flag will still be set

This algorithm is followed throughout the life of the remote connection and is able to handle any state the connection might be in.

**5.2.1.2 Receiving**

The receiving routines are organized somewhat differently than the sending routines in that they are not broken up into different levels. IPQ, specifically the Communications thread, simply waits for an event object (discussed below) to be set for an
ipq_udpInPort structure. When the event is set, it calls the main receiving routine, ipq_recvUDP, which will return a message if available.

Just like the sending process, the receiving process has a fundamental data structure that holds a connection’s state and sequence number information. This structure, shown below, is the ipq_msgData structure.

typedef struct ipq_msgDataStruct
{
    u_long address // used for identifying the remote connection
    u_short port // used for identifying the remote connection
    long int lastSequenceNumber // last sequence number received, initialized to zero
    long int lastAckedSequenceNumber // last successfully acknowledged sequence number, initialized to 0
    long int upTo // Specifies the sequence number that has requested a reply, used in recovery from NACK
    int missingPacket // set to true as soon as the receiver detects a gap in the sequence numbers, remains set until all missing packets are found
    int recovering // set to true after NACK has been sent to sender, reset once ACK is sent
    int dataBuffered // set to true when buffered data is available, which means that a new packet has filled a gap
    inMsg inPackets[SET_MAX] // records incoming messages, keeping track of their sequence numbers and possibly their data as well
} ipq_msgDataStruct;

Another structure, ipq_udpInPortStruct, holds socket information and a list of the ipq_msgData structures. Each incoming connection has an associated ipq_msgData structure.

typedef struct ipq_udpInPortStruct {
    SOCKET fd;
    u_short port;
    L_list lastMsgList;
    HANDLE dataBuffered;
} ipq_udpInPortStruct, *ipq_udpInPort;

Also like the send routines, the receive procedure uses the set idea to describe the range of sequence numbers from lastAckedSequenceNumber to upTo (defaults to lastAckedSequenceNumber + 5).

While working in a particular set, the receiver maintains a record of the received messages in ipq_msgData’s inMesg list (inPackets). This list holds structures similar to the outMsgs structure. The structure holds a complete record of a received packet. Unlike the outMsgs though, these structures only hold the packet’s data when a missing packet has been detected.
typedef struct inMsg {
    int received;
    void *data;
    int bytes;
} inMsg;

A significant difference between the new and old protocols is that we must now process out of order packets, instead of simply dropping them. In figure 7, x0, x1, x2 ... are sequence numbers for incoming packets. Time is represented by the horizontal lines running left to right.

![Sending Process](image1)

The first responsibility of ipq_recvUDP is to check to see if there is buffered data that can be returned immediately in any of the msgData structures. Data can only be returned when there are no outstanding sequence numbers that come before it. In the case described in figure 7, the packet with sequence number x3 could not be returned to the caller until x2 had been returned. Below is a list of the sequential events a receiving thread would go through while waiting for the inport events based on the figure 7.

In this case, the receiver will receive sequence numbers x0, x1, and then x3, creating a gap between x1 and x3. This gap is filled when the missing packet, x2, is received. Once this happens, the internal state of the msgData structure must be updated to reflect the fact that all packets up to the last in order sequence number are received, for example, msgData->lastSequenceNumber would be set to x3.

An IPQ thread (typically the Communications thread) waits for one of two event objects associated with an inport to be set, which signals that there is data ready to be processed. The first event object is the fd_read event associated with the inport's socket. This event is set by the operating system when there is new data on the socket. The second event object is the inport's dataBuffered object. Each inport has a list of ipq_msgData structures corresponding to each connection that has previously been established. The inport's dataBuffered event is set when any one of the ipq_msgData structures associated with the inport has its dataBuffered flag set. When either of an inport's events are set, ipq_recvUDP is called for the inport.
1. `inport->fd_read` set (new data on the socket)
Action: call `ipq_recvUDP`, returns data from packet x0.

2. `inport->fd_read` set (new data on the socket)
Action: call `ipq_recvUDP`, returns data from packet x1.

3. `inport->fd_read` set (new data on the socket)
Action: call `ipq_recvUDP`, returns NULL. The new packet was x3, this data could not be returned since x2 is outstanding.

4. `inport->fd_read` set (new data on the socket)
Action: call `ipq_recvUDP`, returns data from packet x2. The new packet was x2, which could be returned since x1 was already processed.

5. `inport->bufferedData` set
Action: call `ipq_recvUDP`, returns data from packet x3. The data from x3 can now be returned since there are no older outstanding packets.

6. `inport->fd_read` set (new data on the socket)
Action: call `ipq_recvUDP`, returns data from packet x4.

For clarity, before describing the details of the algorithm followed by `ipq_recvUDP`, we first present the following utility routines that are used in the algorithm.

- **void * getBufferedMsgData(inport)**
  Searches through the given inport's `ipq_msgData` list and returns the first data packet it finds that has been buffered and can now be returned.

- **int outstandingPackets(ipq_msgData msgData)**
  Examines the `msgData` structure and returns true if there are outstanding packets in the set and false if all packets are marked as received.

- **int recoverOrder(ipq_msgData msgData, long int seqNum)**
  One of the primary bookkeeping routines in the receiving process and is called when a packet is received and the `msgData`'s `missingPacket` field is set to true. By examining the `seqNum` and `msgData` parameters, this function detects when a packet fills a gap.

- **void sendNack(ipq_udpInPort inport, ipq_msgData msgData, ...)**
  Searches the `msgData`'s `inPackets` for missing packets. It will then construct a NACK containing the associated missing sequence numbers and send it to the sender.

- **void sendAckAndReset(inport, msgData, ..., int seqNum)**
  Sends an ACK for the given sequence number back to the sender. Since this marks the end of a set, the internal state of `msgData` is also reset. All of the `inPackets` have their received flags set to false. The `lastAckedSequenceNumber` and `lastSequenceNumber` fields are set to `lastReceived`, which is not necessarily the `seqNum` being acknowledged, since if the connection was recovering from a NACK it would be the last missing packet. Also, `upTo` is set to `lastReceived + 5`, `missingPacket` and `recovering` are set to false.
Figure 8 outlines the general steps in the receiving algorithm, which is now described in detail. Because of the added complexity of the receiving procedure, it is difficult to describe in the same manner the sending procedure was presented; thus the receiving procedure is mainly described through pseudocode.

Figure 8: The receiving process. The flowchart begins when ipq_recvUDP is called and ends when the routine completes.
Once called, `ipq_recvUDP` first goes though some initial steps before handling any incoming packets. This step is presented in the pseudocode below:

```c
ipq_recvUDP(ipq_udpInPortStruct inport) 
{
  void *    packet
  ipq_msgData       msgData
  int    inDex

  if (inport.bufferedData == TRUE)
    return getBufferedMsgData(inport)

  /* else, get new packet */
  /* the incoming bytes are placed in 'packet' */
  n = recvfrom(inport->fd, packet, &recvAddr, &recvLen)

  /* find the corresponding ipq_msgData for this source */
  msgData = getMsgData(inport, &recvAddr);

  if (msgData == NULL) /* new connection, create a new structure */
    msgData = create_ipq_msgData(recvAddr)

  seqNum = packet.seqNum
  inMessage = packet.data

  inDex = ABS(seqNum) - msgData.lastAckedSequenceNumber - 1

  if (! msgData.missingPacket) /* No previously missing packets*/
  {
    msgData.inPackets[inDex].received = TRUE

    if (seqNum == msgData.lastSequenceNumber + 1)
      /* Packet is in-order */
    {
      msgData.inPackets[inDex].data = NULL

      /* Packet marked as received, no buffering necessary
         since it was in order and can be returned immediately */
      
      msgData.lastSequenceNumber++
      return inMessage
    }
  }
```

Once `ipq_recvUDP` has separated the new packet into its sequence number and data and found the appropriate `ipq_msgData` structure, it can begin processing the packet. The first step taken is to look at the `seqNum`: if it is positive, the packet does not need to be acknowledged, if it is negative however, the sender is expecting an acknowledgement. We begin by presenting the pseudocode for dealing with a packet that does not need an acknowledgment.
else if (seqNum > msgData.lastSequenceNumber + 1)
  /* This packet is out of order */
  {
    msgData.missingPacket = TRUE
    msgData.inPackets[inDex].data = inMessage

    /* Packet marked as received but data must be buffered since there are missing packets */

    /* Don’t increment lastSequenceNumber, since lastSequenceNumber represents the last in-order packet’s sequence number */

    return NULL
  }
else
  {
    /* Any sequence number less than the lastSequenceNumber + 1 and NOT requesting an acknowledgment can be ignored, since it is guaranteed that we have already received it. The case if it is requesting an acknowledgment is addressed shortly */

    return NULL
  }
else
  /* a missing packet had already been detecting */

  msgData.inPackets[inDex].received = TRUE
  if (recoverOrder(msgData))
    {
      msgData.inPackets[inDex].data = NULL
      return inMessage

      /* inPort.dataBuffered would be set if there are now other packets to be returned due to this message filling a gap */
    }
else    /* This packet did not fill the existing gap in the sequence numbers, data is buffered */
    {
      msgData.inPackets[inDex].data = inMessage

      return NULL
    }
}
The second possibility is if seqNum is negative, meaning the packet is to be acknowledged. This part of the algorithm is described by the following pseudocode:

```plaintext
seqNum = ABS(seqNum)
if (seqNum <= lastAckedSequenceNumber)
{
    /* This is an indication that the last ACK we have sent never reached the sender, simply bounce an ACK back:
    sendAck(lastAckedSequenceNumber) */
}
else if (! msgData.recovering)
/* This is the first acknowledgment requested for this set. */
{
    if(!msgData.missingPacket && seqNum= msgData.lastSequenceNumber + 1)
    {
        /* There were no missing packets previously detected and this packet is in-order */
        sendAckAndReset(inport, msgData, seqNum)
        return inMessage
    }
    else
    /* either a packet was already missing or this packet is out of order. Note that this packet could not possibly fill an existing gap, since the sender can not send a sequence number greater than this one until receiving an acknowledgment */
    {
        msgData.missingPacket = TRUE
        msgData.inPackets[inDex].received = TRUE
        msgData.inPackets[inDex].data = inMessage
        msgData.recovering = TRUE
        msgData.upTo = seqNum
        sendNack(inport, msgData, seqNum)
        return NULL
    }
}
else /* The recovery flag had already been set, we have already sent a NACK to the sender */
{
    msgData.inPackets[inDex].received = TRUE
    if (outstandingPackets(msgData))
    /* There are still missing packets */
    {
        msgData.inPackets[inDex].data = inMessage
        sendNack(inport, msgData, seqNum)
        recoverOrder(msgData)
        return (inMessage or NULL)
    /* This packet could have filled a gap, which would allow us to return it and the bufferedData flag might be enabled to
```
By following the flow of control presented in figure 8 and the pseudocode above, the receiver can handle all the possible scenarios while maintaining the ability to recover from missing packets.

5.2.2 Performance

In order to compare the performance of the two protocols presented we ran the benchmark application previously developed to test IPQ. For these trials we ran the application using only event messages in order to highlight the effects of the two protocols by maximizing the number of packets (messages) actually sent across the network (we will report tests on different update/event message mixes in section 6). Ten trials were run on two Windows 2000, Pentium III workstations connected with 100Mb/s Ethernet, under ideal network conditions. Since we are comparing only the network protocols presented, all tests were done with two processes on remote machines. (Local processes would communicate via shared memory)

Figure 9 shows the results of both the send time and message lag times we compiled. In the graph showing the message lag times it is important to note the third dashed line that is introduced: minimum expected lag time. This is a very important aspect of the test(s) in that it illustrates the real reasoning behind IPQ and the advantages of using updates message. With the introduction of receiver delay (typical in real-world systems) between reading each message, a minimum amount of lag time is produced since the sender is sending out these messages very rapidly. Although the communications thread receives all the messages sent by the sender and puts them on the queues quite quickly, the receiver delay delays these messages from being read by the task thread. The minimum lag time introduce will be approximately the receiver delay variable multiplied by the number of messages received, which in these tests is 1000. As we will see in section 6, the problem of the sender flooding the receiving process introduces significant lag times for TCP and IPQ (event messages only) alike and in fact the only way to really help the situation is to start dropping messages. This of course is where IPQ’s use of update messages comes into play and is the basis for comparing IPQ with TCP.
Figure 9: Send times and message lag times (of the last message received) for 1000 event messages.
Figure 9’s results show that although we see send times decrease by a fair amount, there is no significant performance gain between the two protocols in terms of message lag except for the case of zero receiver delay. It appears that performance gains are in effect “washed out” by the receiver delay between messages. To further emphasize the effects of the receiver delay, figure 9b graphs the message lag times for the same test trials, normalized to eliminate the effects of the minimum expected lag time. Here it is much more clear that we do in fact see a substantial performance increase with the optimistic protocol when the receiver delay is at zero. We suspect that this washing out effect would be less when the sender was sending messages at a rate closer to the receiver, although this theory will have to be tested further.

What we do see from figure 9’s results is that when the receiver is accepting incoming messages at a maximum rate our optimistic approach can clearly deliver messages more quickly than the pessimistic protocol. The increase in performance using the optimistic approach is to be expected under ideal network conditions since the assumption that messages are reaching their destination is almost always correct. A more interesting comparison between the two protocols comes with the introduction of network load. To simulate this, another series of tests were run in which a variable amount of packets were artificially dropped. For simplicity only data messages from the sender were dropped, acknowledgements from the receiver were not; their loss does not affect the overall performance of the protocol much differently than the loss of data messages (the handling of lost acknowledgements was addressed in previous sections). This test was again run using only event messages (unique) and receiver delay was not taken into account (zero receiver delay). Five trials were run for each of the five different drop percentages simulated: 1%, 2%, 4%, 10%, and 20%. The send times and message lag results of these trials are shown in figure 10.

![Message Lag - 1000 Event Messages (Normalized)](image_url)

Figure 9b: Message lag normalized to take into account the minimum expected lag time (see figure 9).
Figure 10: Send time and message lag times (of last message received) for 1000 event messages with simulated network load. Receiver delay is zero for all tests.
These results are quite interesting in that the performance gains achieved with the optimistic protocol are not lost as the network conditions worsen. In fact, not only are the lag and send times better, but the optimistic protocol’s performance also degrades less quickly than with the pessimistic approach.

The reason behind this is that each time a packet is dropped the pessimistic approach pays the penalty of waiting a specified time for a response that never arrives (in our implementation, the first timeout period is one second) and then resending the packet. In contrast, the optimistic approach does not pay an equal penalty for each dropped packet (even though both implementations wait for acknowledgements for the same amount of time). Each time a packet is dropped, the minimum penalty is the time it takes to resend the packet, although unlike with the pessimistic approach, the new protocol will only pay the cost of waiting for the acknowledgement that never arrives if the packet dropped was requesting one. Since approximately one out every five packets requests an acknowledgement in our implementation, the optimistic protocol is able to absorb much of the cost of missed packets.

6. IPQ’s Performance vs. TCP

The networking protocol introduced in this paper is only a part of the IPQ system. IPQ’s main purpose and contribution is its ability to remove extraneous state updates en route to the receiving process. This ability works towards solving the problems that occur when the receiver is being flooded with updates and must sift through incoming message to find the most recent one, increasing message lag times. The updateable queues, communication and send threads, and the networking protocol all play roles in working to keep message lag times down and in this section we present a comparison of IPQ using the new networking protocol versus a TCP implementation of the benchmark application used in the previous sections.

For this comparison, we ran our benchmark application with varying state update to event message ratios in order to show the effects of IPQ’s use of state updates. Once again, 10 trials (each) were run for receiver delay times of 0, 0.02, 0.04, 0.06, 0.08, and 0.1 seconds. The sending process once again generated messages (updates or events) continuously, without delay. Although this is somewhat unrealistic for a sending process in the real world, this is in fact the most taxing situation for a networking system to handle due to the flood of messages arriving on the receivers end – building up more the longer the receiver delays. This is precisely the type of situation that can significantly hurt the performance of DVE’s and is what IPQ is designed to help. Figure 11 shows the send times and message lag times (associated with the last message received) for IPQ using a varying mix of update/event messages along with the results of the TCP trials. Once again, just as in section 5.2.2, we have added a dashed line to the message lag graph denoting the minimum expected lag, which is equal to the receiver delay multiplied by the number of messages (expected) to be received.
Figure 11a: Sent times for different event/update message ratios (1000 total messages) along with TCP.
Figure 11b: Message Lag times (of last message) for different event/update message ratios (1000 total messages) along with TCP.
In figure 11a we can see that for all trials the time it took to send out all messages was quite small, however TCP send times are significantly lower. We believe that there are two main reasons for IPQ’s lower performance in this experiment. First off, the times reflect that there is indeed overhead involved with sending messages out that is not incurred with TCP. This overhead can be coming from a variety of places, from the reading and writing of queues to the networking protocol algorithms. A second reason for this lower performance has to do with the implementation of the TCP protocol itself. When an application attempts to send a TCP message, the send routine will return immediately as long as there is buffer space available at the transport layer level. In these tests it appears that the TCP buffers are not being overloaded, and thus the application is never forced to block on the TCP send. In our new protocol we have attempted to lessen the time spent waiting for acknowledgements, yet we have not eliminated the overhead completely. In the future we intend to run more tests to find out when the TCP buffers will begin to fill up and how IPQ will compare in those situations. One encouraging result in regard to IPQ’s send times is that they stay essentially constant while receiver delay increases. This is due to the communications thread remaining unaffected by the receiving task’s delay, continually receiving and acknowledging incoming network traffic independently.

Although we do see lower performance in send times when compared to TCP, our main concern remains the message lag times. Figure 11b shows that as the portion of state updates sent rises, IPQ gives us much lower latency. This of course is because IPQ is able to eliminate the obsolete state messages. By enabling the receiving task to process only the relevant messages, IPQ is able to bring lag times down well below the expected minimum time shown by the dashed line. (It is important to remember here that although many updates are discarded before reaching the receiving task, the last message the receiving task gets is always the last message sent by the sender) These latencies are in stark contrast to the TCP trials, which rise quickly as receiver delay is increased. These results are also consistent with earlier results on previous implementations of IPQ under Unix

Perhaps the most encouraging observation from figure 11b however is that even when sending 100% event messages, thus preventing IPQ from taking advantage of any state updates, message lag times mirror those produced with TCP. In this situation, IPQ is essentially “handcuffed” and in prior implementations, IPQ was unable to match TCP’s performance in this situation. These results are particularly important because it shows that even an application producing large numbers of event messages instead of making use of state updates will incur no added latency above that which TCP would produce.

7. Conclusions

In this work we have primarily built upon the previous development of the Inter-Process Queue (IPQ). We have developed an implementation that works in the Windows environment and produced successful results. This windows implementation was then modified by the implementation of a different network protocol. We have shown that by using a more optimistic approach to guarantee reliable, in-order delivery of packets we can cut down on network traffic and enhance the performance of the system as a whole, particularly in less than ideal network conditions. Our results suggest that in addition to the logical conclusion that a more optimistic protocol works best under ideal conditions, a more optimistic approach can also realize performance gains as network
conditions worsen. We have shown that although there is overhead (book keeping) involved with this optimistic approach, the performance benefits outweigh the increased complexity.

Finally, we have also shown the overall effectiveness of IPQ by comparing it with TCP. For our results it is clear that by taking advantage of the nature of state update messages, IPQ can continue to provide low latencies even as the receiver delay increases – where TCP cannot. In addition, we have also shown that even in the absence of state updates, IPQ can provide latencies similar to that of TCP.

8. Future Work

Although it seems that the optimistic protocol is more efficient, more research is needed to identify how many packets should be sent between acknowledgments. In our implementation we have chosen this number to be five, but more experimentation may show that different numbers may be more suited for different network conditions. Ideally, our protocol might dynamically adjust this number in response to the changing network load.

In addition to continued work with the networking protocol, one issue we must address is determining a way to regulate the rate of the sending process when a receiver’s queue (Communications to Task thread) becomes full. Currently, when the Communication thread cannot put an incoming message onto the queue, it immediately discards the message; the protocol continues as though the message was lost by the network. Although this simplistic approach handled situations well in the past, as our communication speeds have increased (due to the new protocol), this has become a more significant problem since more packets are getting to the receiver before it can make room, in turn generating more retransmissions. In the future, we will need to implement algorithms that will allow the sender to be notified that the receiver needs time to process queued messages, therefore solving the problem in a more pro-active manner.

We are also exploring ways of giving programmers more flexible and powerful ways to define event and update messages. Particularly, we would like to allow the user to define semantic groups of messages that the queues will be able to handle accordingly. We have already begun a new implementation of the internal structures of the queues to provide the flexibility we will need to implement the grouping mechanisms. It is also our eventual goal to integrate IPQ with an existing VR library, namely the Simple Virtual Environment (SVE) [Kessler-00] library.
8. References


