A Network Communication Protocol for Distributed Virtual Environment Systems

G. Drew Kessler and Larry F. Hodges
Graphics, Visualization, and Usability Center
College of Computing, Georgia Institute of Technology
Atlanta, GA 30332 USA
{drew, hodges}@cc.gatech.edu

(To appear in the Virtual Reality Annual International Symposium, 1996, Santa Clara, CA)

Abstract

Virtual environment (VE) applications involve many different tasks, including interfacing with input and output devices, providing responsive user interaction, and simulating a dynamic environment. The variety and number of tasks lends the application to a distributed computing system, where different tasks are performed by different computing resources. A critical issue that arises from such a design is how information is communicated between tasks. In particular, for virtual environments, how information is communicated promptly *is the critical issue. In this work, we describe a pattern of communication common between VE tasks which is not addressed by other communication protocols, namely the communication of state information that continuously changes. We describe a new protocol based on an* updatable queue *abstraction which allows obsolete state information to be discarded, and compare a prototype implementation of that abstraction with a standard communication protocol.*

1. Introduction

Virtual environment (VE) applications usually consist of many different tasks that modify or examine a shared environment. The tasks vary considerably in computing requirements, from obtaining tracking and hand input information through continuous communication with external devices, to rendering processes that require graphics intensive computation, to processes that perform collision detection between entities of the environment and simulate movement and behaviors for those entities, to processes that produce audio output. Clearly, VE systems benefit from a distributed computing design, allowing tasks to be performed by the computing resources that best match the task. The issue that arises from such a design is how information is communicated between tasks.

From our experience with distributed VE systems, we have categorized communication between tasks of a VE application into three types: state update messages, command messages, and event messages. Event messages convey information which cannot be discarded. Command messages are similar to event messages, but require a reply. State update messages, however convey the current state of a subset of the shared environment. A state update message becomes obsolete and extraneous when a new state update message is generated for the same subset, unless an event or command has occurred since the last state update, or the receiver requires a complete history of state changes. Distributed VE systems generally transmit a great deal of state update messages between tasks, and usually require only the most recent state available. In fact, old state information is not only useless in most cases, it is harmful. A task cannot afford to spend time sifting through old messages, and may never be able to "catch up" to read the most recent state update. Existing communication protocols, however, do not support the concept of message obsolescence.

We present here a message passing abstraction called an *updatable queue* which allows for messages to become obsolete when possible, reducing considerably the effort needed to obtain the latest state information, while supporting any combination of state update, event, and command communication. Using a protocol based on this abstraction, we describe a prototype implementation in *user* space, and present a performance comparison to another implementation using a standard protocol (TCP).

2. Related Work

Communication in current distributed VE systems usually involves one-to-one communication with processes that interface external devices, or one-to-many communication among processes that share a common state. Communication with processes that interface with external devices is usually done using a client-server connection using the TCP/IP protocol, as in the RubberRocks application[7] and the MR toolkit[10].

Systems that maintain a common state are organized in a client-server topology, in which the servers are connected to client task, as in the MR toolkit's EM system[15], BrickNet[11], and the RING system[8]; or connected peer to peer (fully connected), as in the DIVE system[5], and the systems based on the Distributed Interactive Simulation (DIS) protocol: SIMNET[4], VERN[3], and NPSNET[9]. These systems have addressed two main issues, consistency and message traffic, but rely on standard available protocols.

Most of these systems maintain consistency by identifying one process as the "owner" of an entity of the shared state, which is the only process that may change that entity. The DIVE system utilizes another communication package, the ISIS system[1], to guarantee consistency between peers. Others have argued, however, that systems such as ISIS that provide causally and totally ordered communication do not know enough about the application level semantic consistency requirements to be efficient or complete, since they are based on message traffic alone[6].

Systems that maintain a common state through a client-server topology address the issue of message traffic by filtering messages that pass through the server on their way to another client so that clients only receive the messages they need (based on geographic information, as in the RING system, or semantic information, as in BrickNet). The NPSNET system utilizes separate Multicast groups to divide the peers such that only geographically close peers receive messages from each other.

The systems based on the DIS protocol address the message traffic issue by using UDP broadcast or Multicast messages (avoiding each peer sending one message to every other peer), and by simulating other peers through a *dead reckoning* algorithm. Because this algorithm simulates state changes locally, a peer need only broadcast its state when it has changed within a threshold error from what other processes will simulate. The DIS protocol does not send obsolete messages because it does not provide guaranteed message passing (although obsolete messages could resided in the receiver's message buffer). It overcomes lost messages by broadcasting an entity's state at timed intervals (nominally 5 sec.). This protocol, however, is not a general solution to communication between VE tasks, as event and command communication require ordered, guaranteed message passing, and the receiver should not have to wait 5 sec. to receive the latest state information if the first send was unsuccessful. In addition, when a task stops transmitting state information, the associated state information at other tasks may be inconsistent.

In this work, we are less concerned with the topology used for a set of processes that make up a VE application then we are with the underlying communication protocol used between these processes. Systems such as PVM[14] and ISL[13] provide frameworks for distributed and parallel computation, but do not address the particular needs of VE applications, for which a more relaxed reliability requirement is desired to maintain responsiveness.

Most protocols, including TCP, do not allow messages to be deliberately dropped. Protocols that do drop messages, such as UDP, generally do not allow the application to decide which messages are dropped. The communication protocol described in [9] does provides control over message reliability by using unguaranteed message passing and allowing the *receiver* to decide if a lost message (once it is detected) needs to be resent. This method requires the use of "heartbeat" messages to ensure message loss detection, and message log processes to assist in message recovery. The protocol presented here allows the *sender* to decide which messages need to be resent at any point in time, and does not allow old messages to be buffered at the receiver where they could become obsolete.

3. Communication in Virtual Environments

Communication between tasks in a VE application can be categorized into three types: state update, events, and commands (Figure 1). A particular task may require each of these categories of communication operating in tandem, or just one.We discuss communication as being from one process to one process, but our description could be extended to one-to-many communication, and our implementation could be extended to take advantage of multicast communication.

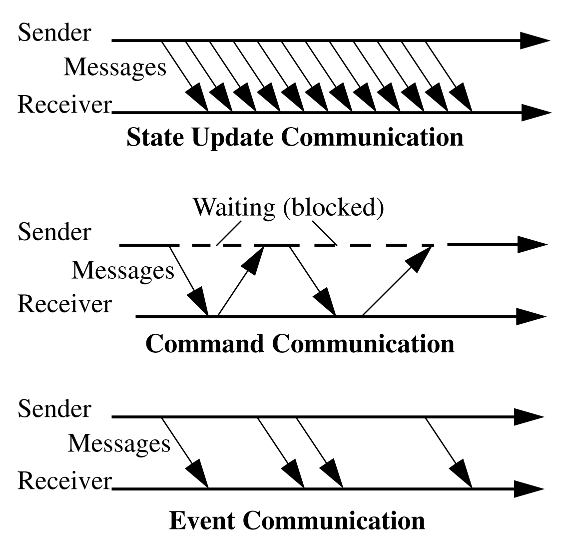

Figure 1: Communication categories

3.1. State Update

State update communication involves a stream of state descriptions, each of which describe a subset of a common state and *replace* previous descriptions for that subset. The position and orientation of a head tracking device would be an example of a state which is shared between the tracker device interface process and a task that maintains the current viewpoint. Using a continuous stream of state updates ensures that when the model maintainer or rendering task requires the tracking information, the latest possible information will be available from the communication channel. The DIS protocol discussed above, whose state includes entity position, velocity, etc., can be categorized as utilizing state update communication.

In practice, there are problems with sending a new state description every time the state changes (which occurs frequently, for tracking information and quickly moving entities). If the communication channel guarantees the delivery of every state description message, then the receiving task will need to read all of the obsolete information before obtaining the latest state information (which will most likely not be what was the latest state information when the task began to read messages from the channel). Implementing some type of flow control will alleviate the problem, but will increase the average *lag* time (delay between sending and receiving a message), as the state descriptions will be sent less frequently.

If the communication protocol does not guarantee delivery of messages, as is the case for the DIS protocol, the same overflow problem occurs, but less severely, as old messages will be automatically dropped if not received within a certain amount of time. Increasing the time between sending messages by having the receiver simulate the sender's state between messages, as in the dead reckoning algorithm, will prevent the receiver from being bogged down. This method works well for predictable behaviors (such as tank movements), but it is unclear that it works well for unpredictable behaviors (such as user head movements). In addition, using unguaranteed message passing introduces a new problem for state update communication when the stream of state descriptions is too slow to overcome the rate at which messages are lost.

The protocol described here uses unguaranteed messages, which are re-sent until acknowledged as received, *or* until a new state update of that type is to be sent (at which point, the message being re-sent is obsolete). The receiving task will, therefore, receive the latest state update message even if the state hasn't changed for a while because the last state update message will be re-sent until it is received.

3.2. Commands

Command messages convey instructions from one task to another, such as audio commands (play sound, stop sound, increase volume, etc.) and hardware device commands (open tracking device, change tracking hemisphere, close tracking device). They differ from state update communication in that a command does not become obsolete, as least not at the communication level. The communication protocol that carries command messages, therefore, must guarantee delivery in First-In-First-Out (FIFO) order. In addition, command messages require a response, although the response may as simple as "done."

Command messages can be used to obtain state information from another task through "polling" (Figure 2). One task, the "client," sends a *poll* command to another task, the "server," which responds with the state information. This method of communicating state information uses fewer messages than state update communication and receives the most current information possible. However, the client task must block and wait for a response.

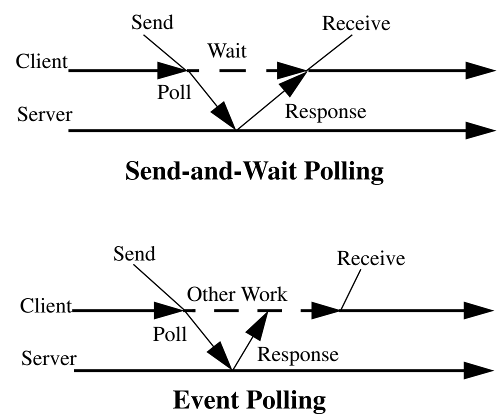

Figure 2: Polling Communication

3.3. Events

Event messages communicate the occurrence of events, such as key-presses, button presses, or object collisions. In addition, commands that do not require a response can be represented as events. Events could occur frequently, but like commands, are unique, do not become obsolete, and therefore require guaranteed, FIFO delivery. When processing events, a task often needs the context, or state, at the time of the event. As a result, the last state update messages before the event cannot become obsolete until after the event has been processed.

Like command messages, event messages can also be used to obtain state information from another task through polling. In this case, the *poll* and *response* messages are event messages. This method has an advantage over using command messages in that the client does not need to block for a response (the communication is asynchronous). The disadvantage is that the response may arrive early, and the information will become old. If the message travel time and time spent by the client doing other work is known, then the response could be timed to arrive when it is needed. In general, however, these times are difficult to predict.

4. Updatable Queue Abstraction

A useful abstraction for one-way, guaranteed, FIFO communication is a queue. It is easy to see that event communication can be represented as a queue. Communication of command messages could also use the queue abstraction by using a pair of queues, an outgoing one for the request, and an incoming queue for the response (pairing command and response by unique ID values). Event and command communication can be combined by using the outgoing queue for both events and command requests.

The communication of state update messages, however, has special requirements that a simple queue cannot provide. Therefore, we construct an *updatable queue* abstraction, which stores *keyed* data. Information that is placed in the queue with a particular key will overwrite the last information stored with that key (Figure 3), with two caveats. Information stored with a special key (0 in our implementation) will not overwrite any other information, and any information stored before the information with the special key (i.e. earlier in the queue) cannot be overwritten (Figure 4). These restrictions allow events and commands to co-exist in the same queue as state updates by assigning events and commands the special key. Placing an event or command on the updatable queue essentially archives the state updates stored before the event or command, until the state updates are removed from the queue (Figure 5).

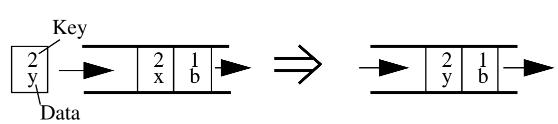

Figure 3: Adding (2, y) to Updatable Queue

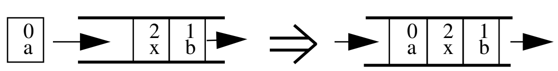

Figure 4: Adding (0, a) to Updatable Queue

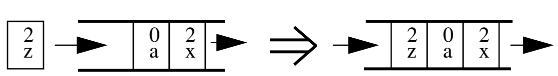

Figure 5: Adding (2, z) to Updatable Queue

5. Implementation

To allow for communication between tasks of a VE application which are being performed by separate processes on the same machine or on different, networked machines, we designed a prototype implementation of updatable queue pairs, which we have named the Inter-Process Queue (IPQ) library. Using the IPQ library, each task can send state update, event, and command messages. In addition, a task can read the latest command, event, or state update without waiting, if none has arrived. The functions to perform these tasks are the same for connections between processes on the same machine, or processes on separate machines, although the message passing is implemented differently.

Our implementation uses one-to-one communication either through shared memory (when the two processes are on the same physical machine) or UDP messages (when the processes are on separate physical machines). We have chosen not to utilize an IP-Multicast protocol due to the added complexity of providing guaranteed, FIFO message delivery. Future implementations may utilize this protocol to reduce message traffic between processes.

5.1. Making a Connection

A client-server model is used to establish a connection between two tasks, although once a connection is established, there is no distinction between the two tasks (either task can send or receive messages to or from the other task). The task which is to be the "server" end of the connection advertises that it is ready to be connected to by storing a name and a UDP listening port in a database contained in shared memory. The task which is to be the "client" discovers which port to connect to by querying the local database (if the connection will be between two local processes), or by querying a "name server" process on a remote machine. The sole purpose of the name server is to listen to a known port for address requests, and clean up when all advertised services on the machine are closed. This connection method is similar to an RPC implementation described in [2].

A process advertises a service with the `IPQ_advertiseService()` function. Note that the task specifies how large the outgoing queue should be to allow for efficient use of resources.

```
IPQ_service IPQ_advertiseService(char
    *serviceName, int serviceType, int
    outsize);
```

Using the returned service structure, the task can ask for a requested connection from another task using the `IPQ_getNewConnection()` function. If no task has requested to connect to that service and *wait* is FALSE, then no connection will be returned, otherwise the process will be blocked until a connection is requested.

```
IPQ_channel IPQ_getNewConnection(
    IPQ_service service, int wait);
```

The task which is to be the "client" end of the connection requests that a connection be established by giving the appropriate service name and machine location to `IPQ_openConnection()`. Note that the client task can also specify how large the outgoing queue should be.

```
IPQ_connect IPQ_openConnection(char
    *serviceName, int serviceType, char
    *machine, int outSize);
```

Once a connection has been established, the two processes can construct packets to send to each other, complete with the key value that indicates if a message is a command or event (0) or a state update (>0), and can receive and de-construct packets from the other task.

5.2. Communication Companion Process

For communication between tasks, we use UDP messages on top of Berkeley Sockets. We do this because we will not always need guaranteed delivery (state update messages that become obsolete no longer need to be guaranteed), and a task may require many open sockets. The common, guaranteed message passing protocol on top of Berkeley Sockets, TCP, is limited by the resources made available to it by the kernel, and therefore does not scale well to many simultaneous socket connections[1]. One of the factors that affects the reliability of UDP message passing is how long a received message has to wait before actually being read by the recipient. The longer the wait, the more likely the message will be dropped to make room for other arriving messages, or other resource needs.

Our implementation, therefore, creates a separate process for each task process which deals solely with incoming and outgoing UDP messages, freeing the job of message communication from the task process, and increasing the reliability of message passing. This method can also be found in BrickNet [11], the EM system of the MR toolkit[15], and PVM[14]. Communication between the task process and its companion communication process is done through Unix pipes and pairs of bounded, updatable queues in shared memory. The communication process waits for an incoming remote message or a message from the task precess using the *select()* command, thereby avoiding busy-waiting.

5.3. Process to Process Communication

In addition to handling all communication from a task process to processes on separate machines, the communication process also assists in making the connection between two task processes, whether they are on the same machine or on separate machines. After the connection is made, the task process will have two updatable queues, one for outgoing messages, and one for incoming messages, which reside in shared memory blocks. If the two task processes reside on the same machine, the updatable queues are direct connections between the task processes (Figure 6). If the task processes reside on separate machines, the queues actually connect the task process to its communication process (Figure 7). Messages to the other task process are read from the shared memory queue and sent from the task's communication process to the communication process of the other task, which forwards it to the task process. Messages received from the other task's communication process are placed on the task's incoming queue by the task's communication process.

Shared memory is utilized for communication between processes on the same machine for two reasons. First, data passed through shared memory does not need to pass through the kernel, which would slow down the transfer. Second, messages sent through the shared memory updatable queues are inherently guaranteed, as the queue is one way (only one process entering data), and flags are used for mutual exclusion.

UDP messages, however, are not guaranteed. Our implementation provides guaranteed message passing with UDP messages by using sequence numbers, and a stop and wait method with an adaptive time-out and an exponential back off. This method, called Jacobson's algorithm, is described in [12]. In our implementation, the method is slightly modified so that if a message fails to be acknowledged, and the information in the message becomes obsolete (because it is of the state update type, and new information has arrived), then *we give up on the obsolete information*, and continue trying with the new information instead. In addition, if a send does not succeed, it is not immediately re-sent if there are other messages waiting to be sent to different destinations. The other waiting messages are sent first to prevent a bad connection from completely stopping transmission through other connections.

6. Performance

In order to judge the performance of the protocol described here, we created a benchmark application which we implemented using the IPQ library and using the TCP/IP Internet protocol. The benchmark application consists of two processes, one which sends a stream of 1000 messages, and one which receives the messages. Each message contains the (machine local) time the message was sent, an ID value, and an array of 16 floating point numbers. This benchmark simulates a pair of processes, in which one process informs the other of a 3D position and orientation of an entity (a common VE task). The processes were run on SparcStation 2 workstations connected via 10Mb/s Ethernet.

We have, in fact, three different implementations. One

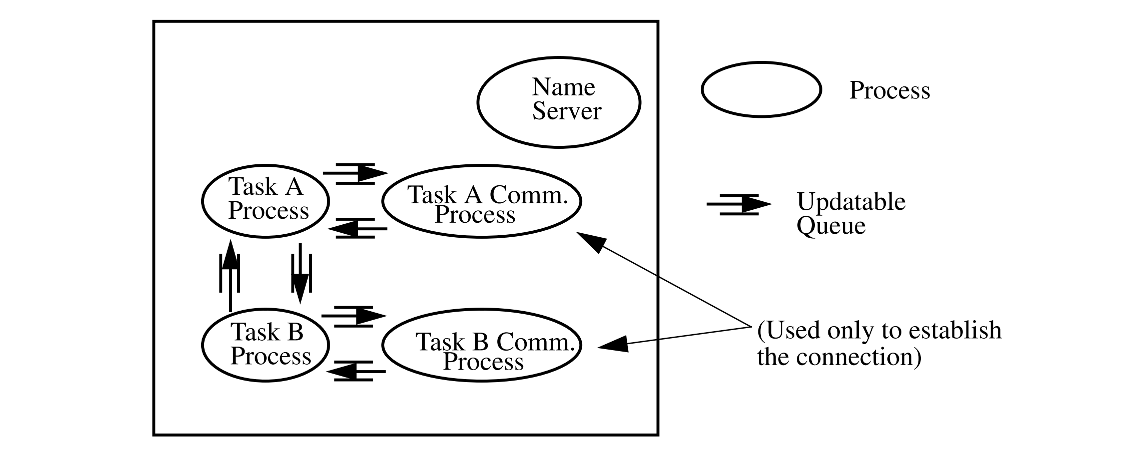

Figure 6: Communication between tasks on the same machine.

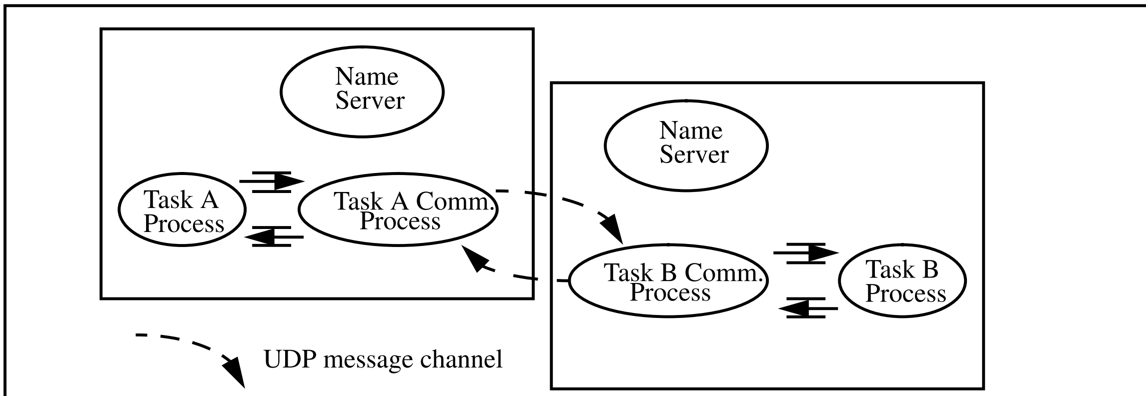

Figure 7: Communication between tasks on different machines.

implementation uses TCP messages, which guarantees message delivery in FIFO order. Another implementation uses the IPQ library and sends each message as a "unique" (or *event*) message, meaning that it cannot be discarded. These two implementations perform practically the same steps. The third implementation also uses the IPQ library, but sends each message as an "updatable" (or *state update*) message, meaning that the creation of a new message makes any old messages obsolete.

We used one additional independent variable to represent processes which spend different amounts of time working on other problems than communication. The variable represents a delay in seconds that the receiving process has between examining the incoming data. Obviously, if the messages are guaranteed, the receiver will fall more and more behind if it only reads one message between delays. On the other hand, the receiver cannot spend all of its time clearing the incoming queue of messages. Our implementation compromises by only reading up to 10 messages between delays. The delay times, 0, 0.033, 0.05, and 0.1 secs., were chosen to represent standard frame rates (30, 20, and 10 frames/sec).

We ran each implementation four times for each delay time using two cases: one in which two processes were on the same physical machine (local), and one in which the processes were on different physical machines (remote). The results are given in Figure 8, where the given lines cross through the average of the four measurement values. Figures 8a and 8b show the lag (delay) time between the time the last message was sent and when it was actually read and processed. Note that for the *remote* case, the lag time can only be used for comparative purposes, as there is no guarantee that the two machine clocks are in synch. Figures 8c and 8d show the amount of time spent by the sender process in sending the 1000 messages.

The main result of our comparison is that, as the receiver spends more time doing other computation, the TCP implementation has a lag time and a send time that increases linearly, while the IPQ-Update implementation (where every message can become obsolete) shows an approximately constant lag time and send time. Recall that the lag time is for the *last* of 1000 messages. If more messages are sent, the lag time will increase linearly for the TCP and IPQ-Unique cases. We observed, however, that

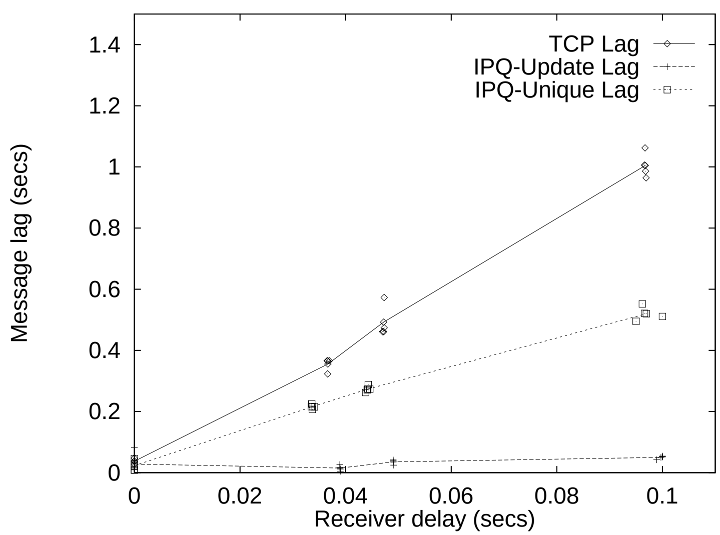

Figure 8a: Message lag for *local* processes given that the receiver spends time doing other things (the delay time)

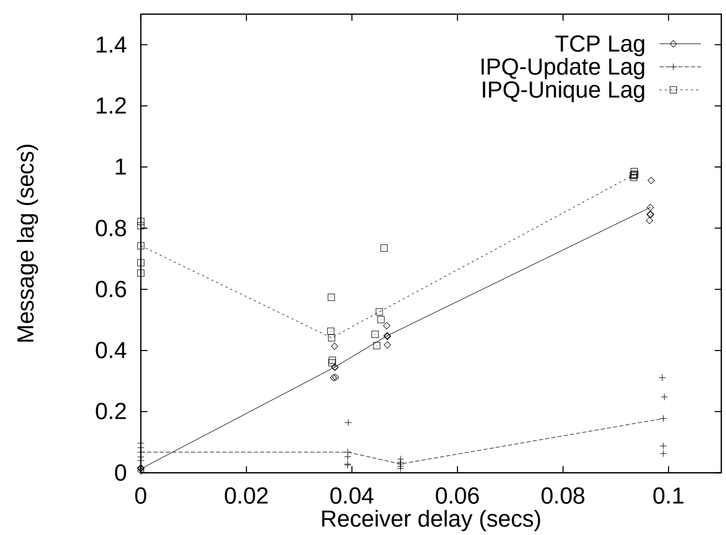

Figure 8b: Message lag for *remote* processes given that the receiver spends time doing other things (the delay time)

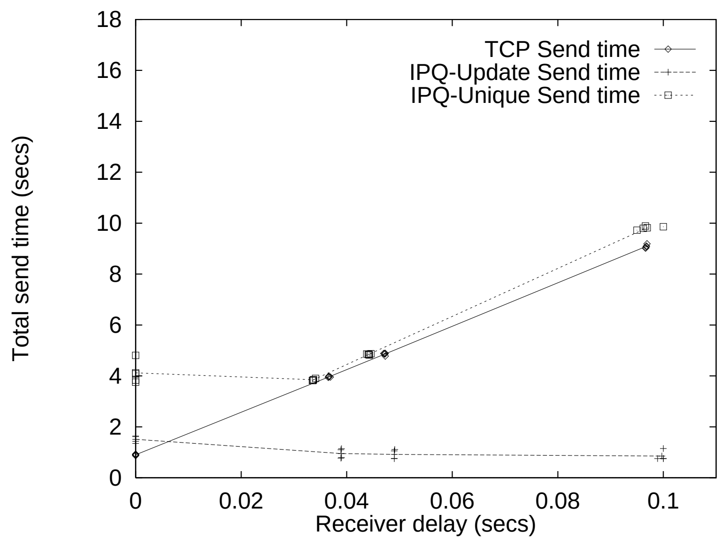

Figure 8c: Time spent by the sender sending 1000 messages to a *local* process given that the receiver spends time doing other things (the delay time)

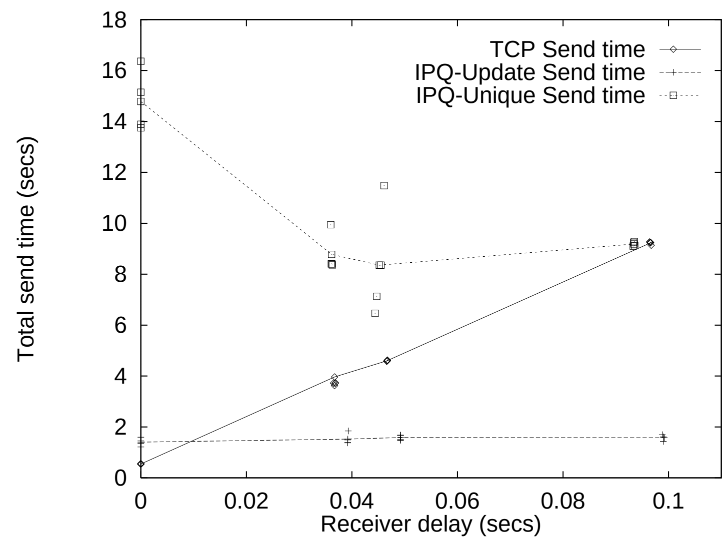

Figure 8d: Time spent by the sender sending 1000 messages to a *remote* process given that the receiver spends time doing other things (the delay time)

the better performance for the IPQ-Update implementation comes at the cost of throughput. In our runs of the IPQ-Update case, only 1-2% of the messages sent were actually read. The lost messages, though, are precisely the ones that the receiver is not interested in, as they have become obsolete due to the inability of the receiver to keep up.

Another significant result from our performance tests was the effect the delay had on the amount of time required for the sender to send 1000 messages. Once again, the TCP and IPQ-Unique implementations show worse performance for higher delay times, and the IPQ-Update implementation has an approximately constant performance. This can be attributed to the sender having to block and wait because the receiver's buffer is full. The sender can only continue when the receiver reads from its buffer, freeing buffer space.

When we compare the TCP implementation with the IPQ-Unique implementation, in which no messages can be discarded, we see that when the processes are local, the IPQ implementation introduces less lag. This supports the claim that avoiding passing data through the kernel (by using shared memory) produces an improved transfer rate. We also see that, in the other cases, the TCP implementation performs considerably better for small delays in the receiving process, and somewhat better with larger delays. This result is expected as TCP is implemented in kernel space and has been optimized to perform guaranteed message communication. The poor send times for the IPQ-Unique implementation with no delay can be attributed, in part, to contention for the shared memory queues.

7. Conclusions

The goal of most VE applications is to provide a dynamic environment that is interactive with the user or users. Meeting these requirements can involve a distributed design, with communication that reflects the most current state available. Existing communication protocols, such as TCP, do not allow messages to become obsolete, and therefore do not allow for state update messages to overwrite older state update messages. We have described an *updatable queue* abstraction that we use to develop a protocol that *does* allow messages to be discarded because they become obsolete. Using a prototype implementation of the protocol, we have shown that, when state update communication is a substantial component of the communication between tasks of a VE application, our protocol provides better performance than a traditional TCP message passing implementation.

8. Acknowledgment

We would like to acknowledge the extensive discussion and advice provided by Brad Topol during the development of this protocol.

9. References

[1] Birman, K., A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, *9*, 3, (Aug., 1991), pp. 272-314.

[2] Birrell, A. D. and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, *2*, 1, (Feb., 1984), pp. 39-59.

[3] Blau, B., C. E. Hughes, J. M. Moshell, and C. Lisle. Networked Virtual Environments. *Symposium on Interactive 3D Graphics* (1992), pp. 157-160.

[4] Calvin, J., A. Dickens, B. Gaines, P. Metzger, D. Miller, and D. Owen. The SIMNET Virtual World Architecture. *Proc. of IEEE VRAIS* (Seattle, WA, Sept., 1993), pp. 450-455.

[5] Carlsson, C. and O. Hagsand, DIVE - A Platform for Multi-user Virtual Environments. *Computers & Graphics*, *17*, 6, (1993), pp. 663-669.

[6] Cheriton, D. R. and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. *ACM SOSP* (Dec., 1993), pp. 44-57.

[7] Codella, C., R. Jalili, L. Koved, J. B. Lewis, D. T. Ling, J. S. Lipscomb, D. A. Rabenhorst, C. P. Wang. Interactive Simulation in a Multi-Person Virtual World. *Proc. of ACM CHI* (May, 1992), pp. 329-334

[8] Funkhouser, T. A. RING: A Client-Server System for Multi-User Virtual Environments. *Symposium on Interactive 3D Graphics* (1995), pp. 85-92.

[9] Holbrook, Hugh W., Sandeep K. Singhal, and David R. Cheriton. Log-Based Receiver-Reliable Multicast for Distributed Interactive Simulation. *Proc. of ACM SIGCOMM* (Cambridge, MA, Aug. 1995), pp. 328-341.

[10] Macedonia, M. R., M. J. Zyda, D. R. Pratt, D. P. Brutzman, and P. T. Barham. Exploiting Reality with Multicast Groups: A Network Architecture for Large-scale Virtual Environments. *Proc. of IEEE VRAIS* (RTP, NC, Mar., 1995), pp. 2-10.

[11] Shaw, C., J. Liang, M. Green, and Y. Sun. The Decoupled Simulation Model for Virtual Reality Systems. *Proc. of ACM CHI* (May, 1992), pp. 321-328.

[12] Singh, G., W. Png, A. Wong, and L. Serra. Networked Virtual Worlds. *Proc. of Computer Animation '95* (Geneva, Switzerland, Apr., 1995), pp.44-49.

[13] Stevens, W. R. *UNIX Network Programming*. Englewood Cliffs, NJ: Prentice-Hall (1990).

[14] Sunderam, V. S. An Inclusive Session Level Protocol for Distributed Applications. *Proc. of ACM SIGCOMM '90; (Special Issue Computer Communication Review)*, *20*, 4, (Sept., 1990), pp. 307-316.

[15] Sunderam, V. S. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice & Experience*, *2*, 4, (Dec., 1990), 315-339.

[16] Wang, Q., M. Green, and C. Shaw. EM - An Environment Manager for Building Networked Virtual Environments. *Proc. of IEEE VRAIS* (RTP, NC, Mar., 1995), pp. 11-18.