The MR Toolkit Peers Package and Experiment [*]

Chris Shaw and Mark Green

Department of Computing Science
University of Alberta
Edmonton, Alberta, T6G 2H1, CANADA
cdshaw,mark@cs.ualberta.ca

November 17, 1994

Abstract

The MR Toolkit Peer Package is an extension to the MR Toolkit that allows multiple independent MR Toolkit applications to communicate with one another across the Internet. The master process of an MR Toolkit application can transmit device data to other remote applications, and receive device data from remote applications. Application-specific data can also be shared between independent applications. Nominally, any number of peers may communicate together in order to run a multi-processing application, and peers can join or leave the collaborative application at any time. This paper will introduce the peer package, and will explain the theory of its operation. The last part of the paper will discuss our experience with a demo program we have written called multi-player handball that uses the peer package.

1 Introduction

One of the main research thrusts in the study of virtual environments is to examine how many people can work together on a common virtual application. The basic idea is to allow people who are geographically dispersed to collaborate on some task in one common virtual space. Software that enables this kind of interaction must support both adequate single-person VR style interaction, and must support communications to other computers. The presence or absence of other users in the simulation should have no effect on the visual presentation, and should not induce undue lags or delays in the interaction.

The MR Toolkit Peer Package is an extension to the MR Toolkit [7, 6] that allows multiple independent MR Toolkit applications to communicate with one another across the Internet. The peer package allows the master process of an MR Toolkit application to transmit device data to other remote processes, and to receive device data from remote processes. Application-specific data can also be shared between independent MR Toolkit applications. Nominally, any number of peers may communicate together in order to run a multi-processing application, and peers can join or leave the collaborative application at any time. Thus, the peer package allows each user to have a smooth interaction with the virtual environment, while updating each remote peer with the local device data.

2 Previous Work

The most famous example of a multi-user virtual environment is SIMNET [1], which is an interactive network system for real-time person-in-the-loop battle engagement simulation and war-gaming. SIMNET uses wide-area ethernets and satellite links to join hundreds of independent simulated components such as

[*]This paper appears in *IEEE Virtual Reality Annual International Symposium (VRAIS '93)*, 463-469, September 18-22 1993

tanks, aircraft, command posts and the like. SIMNET relies on a high-bandwidth and a low-latency communications infrastructure to achieve its goals of accurate military simulation. SIMNET does not have the very low-latency display requirements of head-mounted displays, which may make communications latency a non-issue. Also, my personal experience with the SIMNET Abrams tank simulator is that visual lag is more than 250 milliseconds.

Researchers at IBM [3, 2] have been building simulation systems for multi-person virtual worlds. Their emphasis is on the simulation of moderately complex real-time dynamics systems which are modeled as point masses connected by springs and dampers. The simulation in their system is centralized, and users connect to the central simulation server to generate their own view of the system in action. Users interact in concert with the simulated objects, and they can exert simulated forces on the various "rubber rocks" in the virtual environment. The IBM system is implemented in a local-area network configuration, and is susceptible to communications lags.

Our own work on the MR Toolkit [7, 6] focuses on providing software that supports real-time interaction with a virtual environment using a head-mounted display. Similar to the IBM system, the MR Toolkit drives devices such as trackers and gloves using the client-server model, and has a centralized process which manages the interaction. The programmer of an MR Toolkit application writes code for the *master* process, which collects device data from the servers, and which dispatches device and application data to any *slave* process which may be necessary. A slave process is responsible for providing visual output of the current state of the application. When a head-mounted display is used, the master drives one eye, and an identical slave drives the other eye. Depending on the application, the master may also collect application data from a *computation* process that is performing the application's computations on another machine.

The goal of the MR Toolkit is to provide a software foundation onto which a programmer can easily build single-user virtual environments. The MR Toolkit provides low-latency interaction with virtual environments by eliminating almost all unnecessary lags in the system. The main lags that can be eliminated are tracker lag [5], system throughput lag, and application lag. In cases where lag cannot be eliminated, the MR Toolkit supplies methods of structuring the virtual environment so that these lags do not interfere with the smooth animation of the scene in the head-mounted display.

3 The Peer Package

The Peer Package is an extension of the MR Toolkit that allows a master process to communicate with other master processes on other machines. Any running MR Toolkit program may start up the peer package at any time, and may initiate and quit communications with other processes at will. Each MR Toolkit master program, or *peer*, maintains a list of other peers that it is connected to. Peers are connected pairwise, and a peer may send a message to any or all other peers at once.

To start the peer package, the master process calls the *MR_peer_init(PORT, NAME)* routine, which declares the process's availability on the given UDP *PORT* number using the given application *NAME*. The initialization call returns an *MR_port* data structure which contains a distinct list of remote peers, which are in some state of connectedness. To start communications with a remote peer, the master process sends a connection solicitation message to the remote machine on the given port. If there is a process bound to this port on the remote machine, then the remote process sends back a solicitation message. Receipt of a solicitation message from a remote peer makes the local state of the peer *active*, which allows messages other than solicitations to be sent to the remote peer. If a solicitation message has not arrived, a process will keep sending solicitation messages at longer and longer intervals until the remote peer responds. Note that the local process does not block while this is happening. All connection solicitation and other peer traffic takes place asynchronously.

A local process can update all *active* remote peers with the most up-to-date local device data using *MR_peer_update()*. The *MR_peer_rcv()* call accepts all incoming peer messages, automatically updating local copies of remote device locations, sorting incoming shared data, and managing connections. The peer package detects remote peer idleness using a 10 second timeout, and supplies a callback facility for automatic execution of user-supplied code when any remote peer's state changes.

There is also a facility for sharing application-specific data with remote peers. Peer shared data is the basic tool for transmitting and receiving application-dependent data from an attached peer. The shared data model is used because the concept and some of the routines are already familiar to programmers of single-user MR Toolkit applications. Peer shared data creates a record that points to the storage area of the data to be sent or received from remote peers. This record also defines a packet with an associated ID that is to be sent to other peers. When a shared data message arrives, the appropriate storage area is automatically updated, and this area is automatically read when the appropriate message is to be sent to remote peers. The peer shared data facility also allows callbacks to be defined whenever a shared data item is updated by a remote peer.

3.1 Peer Network Implementation

The peer package is based on the User Datagram Protocol (UDP) for network communication. UDP is a connectionless transport service that uses the Internet Protocol network layer. The UDP header contains 16 error-detection bits, thus allowing erroneous packets to be detected and silently discarded by the receiver. UDP packets may be received out of the order they were sent, as there is no sequencing mechanism in UDP. UDP is used by the peer package for three reasons: connection management is simple, packet communication is one-way, and communications errors do not result in error returns from data sending calls.

The UDP protocol is connectionless, which means that one process can send a datagram to another process without first establishing a connection. If there is no receiver present, the datagram silently disappears. The peer package places a simple race-free connection maintenance protocol on top of UDP to maintain connection between two peers. The advantage of using UDP is that making and breaking connections is easy to handle, and there are no race conditions when two peers seek connections with each other simultaneously.

One-way packet communication is quite valuable in real-time applications. We have found that using one-way packet traffic for sending device data can reduce device-centered lag substantially [5]. Similarly, in situations where total packet travel time is a few seconds through congested Internet links, UDP datagrams will typically arrive in the user space of the receiving process sooner that TCP/IP's reliable messages. The problem with TCP in this situation is that the TCP protocol will send a number of packets, then wait for their respective acknowledgements. The result is that packets arrive in bursts, with a correspondingly bad effect on lag.

The main disadvantage of UDP is that it is not a "reliable" protocol, which means that it does not guarantee that the receiver will get all data that was transmitted, nor does it guarantee that data packets will be received in the same order that they were transmitted. The UDP protocol does guarantee that there are no errors in packets that are received by a process, so a peer is guaranteed to get no wrong data. The peer package does not currently perform any reliability protocol, so peer-based shared data carries some risk that shared data may arrive out of order or may not arrive at all. This risk is minuscule on most local-area networks, but for wider area networks, there may be a nontrivial level of packet loss. For packets containing device data, this is not necessarily a problem, since device packets are arriving many times per second.

3.2 Connection Topology

The peer package by default maintains a complete graph connection topology, which means that each peer is explicitly connected to all other peers. This is done by having the local peer inform all active remote peers that it has received a new remote connection. When the remote peers get this message, they each solicit a connection from the new peer. This connection propagation policy can be turned off at any time by the programmer. The complete graph topology was chosen because the peer software therefore need not concern itself with routing, or with forwarding incoming peer packets to other peers. A hub-and-spoke system can be supported by turning off the connection propagation policy, and writing a peer that automatically forwards incoming data to the other connected peers.

The clear disadvantage of the complete graph topology is that the number of packets transmitted is on the order of the square of the number of participants. With less than five participants, this is not an outrageous amount of network traffic, but beyond five, traffic growth becomes unmanageable, resulting in growing network delays.

In many respects, it would be best if the peer package used the Internet multicast facility [4, 1], since this would allow one message to be addressed to many users at once, thus saving network bandwidth. In contrast to complete graph unicast, multicast packet traffic grows linearly with the number of participants. The reason why multicast wasn't used is that a certain amount of system management is required to support multicasting beyond a local-area network. In particular, multicast currently requires either that all intervening hosts know how to handle multicast messages (not the case), or that the sending and receiving machines set up a *tunnel* through which multicast messages are sent disguised as ordinary unicast messages. Neither option is practical for people without super-user privileges, which means that wide user acceptance may be difficult to achieve. As it stands, the peer package could be re-implemented using multicast without the user being aware of the change.

4 The Multiple Player Handball Demo

The multi-player handball program allows one or more people to play a simple game of handball inside a ball court that is a 2 metre cube. Each player uses a head-mounted display to view the ball bouncing around inside the ball court. Each player uses a DataGlove or a free-range tracker to hit the ball as it moves about the room. The hand acts as a passive, perfect reflector. When the ball reflects off the plane of the palm of the hand, the ball increases its speed by one centimetre per second. The player does not impart any other change in velocity to the ball, so hand velocity is not important to the play of the game. That is, whacking the ball hard is not useful. Both forehand and backhand hits are allowed, and the reflection is off the same plane.

At the front of the court is an array of 64 bricks. When the ball hits a brick, the ball bounces off the brick, and the brick disappears. The object of the game is to eliminate all of the bricks before running out of the 10 balls that have been allocated. Fifteen centimetres behind the array of bricks is the actual front wall, and the ball can bounce in between the front wall and the array of bricks, knocking out bricks along the way.

The right wall of the ball court contains a score panel [7, 6] that displays the current score for each player. When the ball hits a brick, the score is updated, and as new players join the game, their names are added to the score panel. When a player quits, that player's name is removed from the score panel.

When the ball hits the yellow wall at the back of the room, the ball is lost, and a new ball is shot into the room from the centre of the yellow wall. The game is over when 10 balls have hit the yellow wall.

The floor of the room has a tile pattern on it, and a circle is drawn on the surface of the floor to indicate the position of the ball that is overhead. This fake drop shadow is quite useful in locating the current position of the ball, and is also helpful in determining the ball's velocity. Typically, consulting the drop shadow gives a player an accurate idea of the ball's progress towards the front or the back of the court.

The program has two modes: single player mode and multiple player mode. In single player mode, the simulation of the ball dynamics and the player interaction takes place in the master process of a single-user MR Toolkit application. In multiple player mode, the handball program expects at least one MR Toolkit Peer connection with another handball program. Up to 10 players may play together in the same court. Each player is represented visually by a face and a hand, which are drawn using peer package calls. The peer routine to draw a face looks up each remote user's name in a database of pictures on the master's machine, and if there is a user picture, and if the machine supports texture mapping, then the player's face is texture mapped onto a rectangle, as shown in figure 1. If there is no picture, a standard cartoon happy face is drawn. Each player updates the other players every 5 local updates, so while a player's own view of the court and one's own hand update as fast as possible, everyone else looks as if they are executing rather jerky motion. This slow updating of other players is done to conserve precious network bandwidth, a significant factor when playing with sites connected by slow links.

When the multi-player game first starts up, and when at least two players have achieved an active peer connection, the ball starts moving from its starting point at the yellow wall. The color of the ball indicates

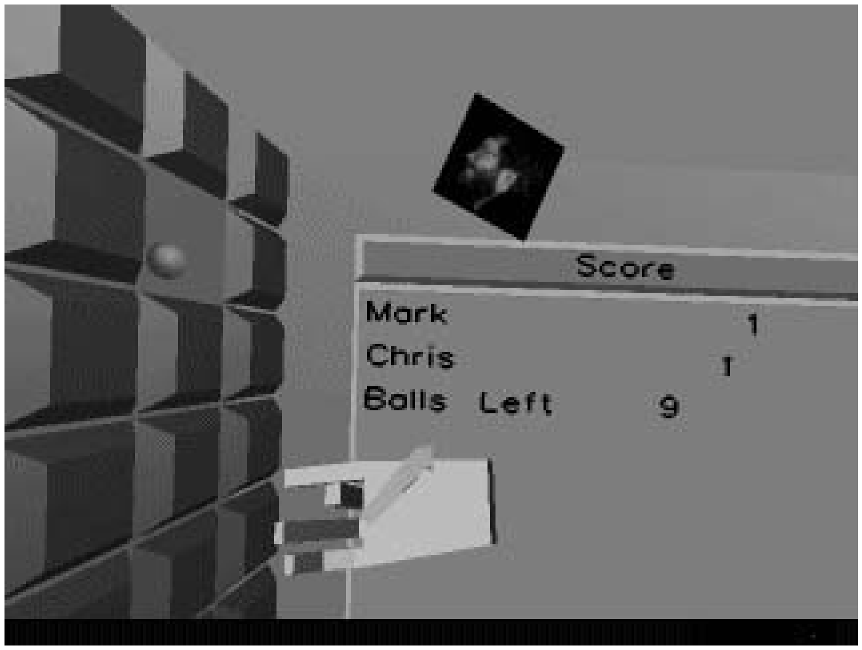

Figure 1: A Two-Player game in progress. The first author is shown in profile

which player is to attempt to hit the ball. The player that sees the ball as *blue* is the *owner* of the ball, and must attempt to hit the ball. When the player hits the ball, the ball changes color, and some other player becomes the owner. This new owner sees the ball as blue.

After a player has hit the ball, all bricks that are knocked out by the ball count towards that player's score. When the next player hits the ball, this next player starts collecting points. A player can hit the ball only once before the ownership token is passed along, so a player has a vested interest in hitting the ball towards the bricks. When the ball hits the yellow wall, the loss of the ball counts against the owning player (the current blue player). In effect, *ownership* of the ball and its scoring capabilities is passed from player to player in a round-robin fashion.

The ball motion is simulated in the same round-robin fashion. The program that currently owns the ball conducts the simulation using only the local player's hand position and the walls for potential reflections. All of the opposing (non-owning) players are ignored for the purposes of simulation. When a hand reflection is detected, the new velocity of the ball is calculated, and the ownership of the ball is passed to the next program in the list. In effect, the multiple player simulation is conducted as a round-robin allocation of single player simulations. Each player is simulated on his/her own machine, and when the ball hits a brick, one point is added to the previous owner's score. At the start of the game there is no previous owner, so no player gets credit for bricks that are knocked out before the first hand intersection.

While the owning player is simulating the ball motion, it updates the other players by sending out ball position and velocity packets every five local updates, or when the ball hits the bricks. The remote players each execute a smoothed dead reckoning algorithm [1] in which the incoming ball packet is used as a base for linear extrapolation of position. New incoming ball packets are used to smoothly adjust the speed so that non-smooth arrival of packets does not make the ball jump backwards or forwards too much.

4.1 Program Implementation

The main loop of the single-player program collects local device data, calculates the ball trajectory and any intersections, and then draws the scene based on the user's viewpoint. Th multi-player program grew out of a single-player program onto which was added the calls to share the various aspects of the simulation with other players. Thus, the multi-player version first declares shared data structures for the ball, the bricks, the player scores and the simulation ownership token. In the main loop, if the peer it owns the simulation, it calculates the ball's motion and updates the other peers. If the player hits the ball, the simulation token is passed to the next player, and the old owner sets a flag indicating that acknowledgement is expected. The local device data is sent out by every peer every 5 updates, and every peer checks for incoming messages from other peers.

The major effort in converting this program to multiple players lies in managing the round-robin simulation. Every peer maintains a list of other peers sorted by process id, so that all peers agree who is the next peer to receive the simulation. Also, a callback procedure is called whenever a new peer gets connected, or when a peer quits, so that it can be correctly added to or deleted from the sorted list. There are also two simulation token callbacks, one of which is called when the simulation token arrives, the other is called when the acknowledgement arrives. This saves having to deal with these messages within the animation loop.

By contrast, the simulation code itself remained essentially unchanged. Two variables indicating collision state had to be made available to the simulation management code. Also, the simulation code previously managed the ball's color, which is now handled by the simulation management code.

4.2 Dealing With Lag

In situations where there is only a small communication lag between each player's machine, the round-robin simulation scheme is not necessary, because one central site can simulate effectively for all players. However, when lag is significant (100 milliseconds or more), centralized simulation does not work. The problem is simply that a central site would simulate based on hand positions that are N seconds old, and will generate reflections that don't match with what the user sees. Perhaps more frustratingly, the user may miss the ball even when it *looks* like there was an intersection, because by the time the hand arrives at the ball's lagged visual location, the simulation has moved the ball N seconds ahead of this lagged location.

Users traditionally deal with lag by slowing their movements down, or by adopting a move-and-wait strategy [9, 8]. Neither of these approaches work because it is the *simulation* that is behind, not the user's hand and eye position. Therefore, the user would need to slow down or stop the simulation instead of the user's own motions, and since this is not possible, the simulation must be structured in such a way as to avoid or eliminate the lag problem.

The advantage this game has is that it can be structured such that only one person is active at a time. The rest of the players watch the action, awaiting their turn to hit the ball. While the owning player is trying to hit the ball, the opposing players see both the ball position and the owning player's hand with equal lag. When the owner hits the ball, the owning process sends the new information to the next owner, and then locally simulates the ball at a much reduced speed. Communication lag is evident during this ownership transfer.

When the new owner sends the acknowledgement to the old owner, the old owner sees the ball speed up and continue along the expected path. The time lag between when the ball first bounces off the hand to when it regains full speed is equal to the total round-trip packet time between the old and new owners. All the other players except the new owner also see this slowed motion, because the old owner stops broadcasting simulation output until the new owner gets the ownership message. Then the new owner starts simulating, which resumes the motion. The new owner does not see the slowdown because as soon as the new owner gets the ownership message, it starts simulating, and ignores incoming (out-of-order) simulation messages. As a result, the only player that will be adversely affected by lag is automatically prevented from seeing lag when it matters.

4.3 Real-World Test

This program was first run between the Banff Centre for the Arts and the University of Alberta in December 1992. The Banff Centre is connected to an intervening Internet link with a 14400 baud modem running SLIP. The intervening Internet link connects to the University of Alberta with a 56Kbps line. The average message round-trip time, measured using *ping*, was between 3 and 5 seconds during the day when the test was done. The interaction was reasonably satisfactory, although the lag in passing the simulation was a little disconcerting to players who were used to the smooth action of the single-player game. Each player saw the entire world move about them at an update rate of 20 or 25 frames per second (depending on tracker configuration), with lag between head motion and visual update in the range of 50-100 milliseconds.

The only perceptible ugliness was the remote update of the ball, which was somewhat more jerky than would be the case with the peer running on the local-area network. This was caused by the variable packet arrival times, but was not a factor in the play of the game, since the remote players don't need to hit the ball when it is moving at seemingly variable speeds.

A more involved demonstration occurred on May 1993 during the Third International Conference on Cyberspace at Austin, Texas. The four sites that played the game were U of Alberta, the Banff Centre, SRI International, and University of Texas in Austin. The game was played for about two hours across the Internet. A site could drop out at any time to switch players or make other local adjustments, then restart to see the other three players still playing. The biggest bottleneck in terms of bandwidth occurred between the U of Alberta and the rest of the world, due to a rather heavily-loaded link.

In an earlier version of this program, a problem arose with the simulation deadlocking due to a lost ownership message. If the ownership message disappeared on its way to the new owner, both the old owner and the new owner would sit waiting for simulation results. Both would see the current head and hand locations of the other player, but the ball would not move because each thought that the other was doing the simulation. This was fixed by requiring an acknowledgement from the new owner, with a 1 second timeout. If the old owner doesn't receive the acknowledgement within the timeout, it re-sends the ownership message until an acknowledgement arrives.

5 Conclusions and Future Work

We have briefly described the peer package, which allows multiple independent VR applications to communicate with each other. We have also introduced a sample application based on this package called multiple-player handball, which demonstrates the flexibility of the peer package. The peer package allows the remote peers to update the local peer's data asynchronously, without any effect on the local user's ability to rapidly turn his/her head to look for the ball.

Future work includes adding a reliability protocol to the peer package. Our experience with lost ownership messages indicates that this will be useful in some circumstances. Some of the specialized facilities used in the handball program are in the process of being moved into the peers package.

References

[1] Earl A Alluisi. The Development of Technology for Collective Training: SIMNET, a Case History. *Human Factors*, 33(3):343–362, June 1991.

[2] Perry A. Appino, J. Bryan Lewis, Lawrence Koved, Daniel T. Ling, and Christopher F. Codella. An Architecture for Virtual Worlds. *PRESENCE: Teleoperators and Virtual Environments*, 1(1):1–17, Winter 1991.

[3] Christopher F. Codella, Reza Jalili, Lawrence Koved, J. Bryan Lewis, Daniel T. Ling, James S. Lipscomb, David A. Rabenhorst, Chu P. Wang, Alan Norton, Paula Sweeney, and Greg Turk. Interactive Simulation in a Multi-Person Virtual World. In *Human Factors in Computing Systems CHI'92 Conference Proceedings*, pages 329–334, Monterey, California, May 1992. ACM SIGCHI.

[4] S. Deering. Host Extensions for IP Multicasting. *RFC 1112*, August 1989.

[5] Jiandong Liang, Chris Shaw, and Mark Green. On Temporal-Spatial Realism in the Virtual Reality Environment. In *UIST 1991 Proceedings*, pages 19–25, Hilton Head, South Carolina, November 1991. ACM SIGGRAPH/SIGCHI.

[6] Chris Shaw, Mark Green, Jiandong Liang, and Yunqi Sun. Decoupled Simulation in Virtual Reality with The MR Toolkit. *ACM Transactions on Information Systems*, 11(3):287–317, July 1993.

[7] Chris Shaw, Jiandong Liang, Mark Green, and Yunqi Sun. The Decoupled Simulation Model for Virtual Reality Systems. In *Human Factors in Computing Systems CHI'92 Conference Proceedings*, pages 321–328, Monterey, California, May 1992. ACM SIGCHI.

[8] Thomas B. Sheridan. Supervisory Control of Remote Manipulators, Vehicles and Dynamic Processes: Experiments in Command and Display Aiding. In *Advances in Man-Machine Systems Research*, volume 1, pages 49–137. JAI Press, 1984.

[9] Thomas B. Sheridan and W. R. Ferrell. Remote Manipulative Control with Transmission Delay. *IEEE Transactions on Human Factors in Electronics*, HFE-4(1), 1963.