Internet Based Real-Time Multiuser Simulation: Ppong!

James "Bo" Begole and Clifford A. Shaffer
Department of Computer Science
Virginia Tech
Blacksburg, VA 24016
{begolej, shaffer}@cs.vt.edu

February 20, 1997

Abstract

There is a growing demand for real-time collaborative applications through the World Wide Web. New techniques are required for real-time applications to perform well in the face of changing network conditions that often include long delays. We present some of the key issues for implementors of real-time web-based applications, including choices on centralized versus distributed control, two user versus multi-user considerations, synchronous versus asynchronous message protocols, and simulation divergence. We present an implementation for a web-based version of the classic Pong game, called Ppong!. Ppong! was selected for implementation since it strips to its essentials many components of real-time collaborative simulation. A key feature of Ppong! is implementation of a heuristic for "retarding" a user's view of the simulation to accommodate network delays.

1 Introduction

The computing field is shifting from an emphasis on *personal* computing to one of *inter*-personal computing. Internet users communicate and interact asynchronously via email, news groups, and publishing or browsing documents on the World Wide Web (WWW). These commonplace asynchronous interactions do not allow users to interact simultaneously, however. There has always been an interest among users for more direct interactions with others. Potential applications include synchronous communications, joint editing of documents, joint exploration of simulations or data, and multi-player interactive games.

The advent of new World Wide Web (WWW) interactive runtime environments provides the opportunity for network-based synchronous collaboration. In particular, the Java programming language and its Application Programming Interface (API) libraries provide a platform-independent environment for creating interactive applications and applets that may be delivered over the World Wide Web [8, 5].

A number of toolkits are being developed, based on Java, for implementing synchronous collaborative applications [13, 15, 22]. These toolkits are appropriate for many synchronous activities, such as text chat and whiteboard, which are typically included in demonstration suites accompanying the toolkits. However, these demonstrations do not address the performance requirements of many forms of interaction. Examples include multi-player interactive games and real-time simulations of physical science experiments for multiple students. Like all collaborative applications, the application state changes with time and a visual representation of the current state is displayed simultaneously to all users as changes occur. Unlike whiteboards and multiuser chat sessions, however, realtime simulations and games require that the state of the application change within some implied time constraint.

This paper discusses software architectures and communications models for high performance synchronous collaboration on the Internet. Our particular interest is in developing high performance real-time multiuser interactive simulations. To motivate and test our ideas, we present an Internet version of the classic Atari "Pong" game, in Java, which we call Ppong!

We have identified several issues that need to be considered when designing a synchronous collaborative system. These issues are presented in Section 2. Section 3 then discusses these issues in the context of our Ppong! implementation under the Java runtime environment. Section 4 presents our conclusions.

2 Web-based Synchronous Applications

In addition to the challenges common to all multiuser synchronous applications, Internet based collaborative software must contend with potentially long network delays imposed by running on an international Wide Area Network (WAN). Network delay is not a significant problem in asynchronous collaborations where participant interactions may be separated by periods of hours or days (e.g., email, Usenet news). However, long network delays seriously impact a synchronous collaboration in which participants are interacting simultaneously.

2.1 Architecture

There is a long-running discussion of the merits and pitfalls of distributed versus centralized architectures for multiuser applications [1]. A centralized architecture enforces data consistency by requiring that all modifications to the data be routed through a central authority. However, this can lead to unacceptable performance due to communication delays. In a distributed architecture, copies of shared data are maintained by each participant. Performance can be improved over the centralized architecture by processing data locally before updating remote copies. However, consistency is more difficult to guarantee, since multiple copies of the data exist.

Centralized Model A centralized client-server model has a simple network connection layout. All participants (clients) only connect to and communicate with a central server, resulting in a star topology. The server maintains the state of the information the participants are sharing and sends updates to participants when modifications occur.

In addition to ensuring consistent data among participants, this client-server approach has other advantages. In this scheme, a more powerful server may relieve clients of some heavy computation, when modifying information is computationally intensive. The client-server model also supports the desirable separation of user interface (client) from application functionality (server).

On the other hand, any communications among participants must go through two sequential network transfers, one from the originating client to the server, and another from the server to each of the other clients. Additionally, the host machine of the server could be heavily loaded by several simultaneous sessions or non-related activities. Finally, a client-server model requires the development of two separate applications, namely the client and the server.

Distributed Model In a distributed model, the collaborating applications hold copies of the shared data and communicate directly with other peers. A distributed peer-to-peer model means no server must host a large amount of network traffic. In web-based collaboration, this is especially important as an application server would often be located on a computer dealing with many sessions of the application, or other network traffic unrelated to the application. The processing load also is distributed among the peers. Peer-to-peer communications can require only one network send for one peer to communicate with another. Finally, a peer-to-peer CSCW system requires the development of only one application replicated at each client. However, this single peer application may well be as large as the combined client and server of the centralized approach, since the peer will implement the functionality of both.

A peer-to-peer model results in more complex connection management for each peer as it must be able to communicate with the others. Participants may come and go during a session. However, this more complex connection management would have been handled by the server in a centralized model, so that portion of the server may be moved into the peer.

Data consistency among participants is more difficult to maintain in a distributed architecture. The problem is basically that of distributed shared memory (DSM) [19], although the granularity of the shared data is generally larger than memory addressable units. Greenberg and Marwood [12] point out that it is important to consider the impact of data consistency techniques on the usability of the system. Even when correct algorithms are implemented to ensure consistency, network lag can inhibit the performance of the system so that it is unusable. Furthermore, differing processing speed among participants can lead to perceived inconsistencies as some participants' machines will display updated data more quickly.

Two-user versus Multi-user considerations For a two-user interaction, the peer-to-peer architecture seems to be most appropriate. Unfortunately, security restrictions or other specifics of the networked environment may not permit the two parties to be true peers. We found this to be the case in our Java implementation of Ppong!, as described in Section 3.

When the application is meant to support more than two users, additional architectural concerns arise. In particular, there are the issues of the number of connections and number of message hops required depending on the configuration of the network at the application layer. A large body of literature exists on the relative merits of network topologies such as stars, rings, complete graphs, and intermediate solutions such as hypercubes. Any advantages gained at the application layer by a particular configuration are constrained and overridden by the physical layer configuration of the network. Since we have focused on a two-user application to explore basic concepts, we do not consider multiuser topologies further.

Preferred Architecture on WANs Due to the difficulties of implementing a distributed peer-to-peer system, many multiuser software systems, particularly shared application systems, use the centralized approach [7, 1, 18]. However, this architecture may be inappropriate for Web-based collaboration for two reasons.

First, when users have relatively high bandwidth network connections and/or low network delay the simplified data consistency of the centralized architecture outweighs its potentially slow performance due to network delays. Most synchronous collaborative software in use today is run over Local Area Networks (LAN) that deliver the throughput necessary for satisfactory performance. However, collaborators working together across the Internet do not generally have the bandwidth of LAN connections. In addition to relatively low modem connection speeds, Internet users may be located far apart, resulting in high network propagation delays.

Second, while the centralized approach is enabled by some underlying window systems on a LAN, it may be inappropriate for Web based collaboration. In particular, collaboration transparency systems (systems that support collaboration using applications written for a single user) tend toward the centralized architecture [20], especially those implemented on the X Window System [17]. X is well suited to this approach, since X has separation of display from computation as a fundamental feature of its architecture. The centralized approach requires distributing large amounts of display data from the central application to the participants. X also uses the model of a relatively "dumb" local display server supporting a remote client with potentially unlimited compute power. In contrast, the Java virtual machine requires that anyone loading a Java applet within a WWW browser have a new and separate instance of that applet. The philosophy is to distribute the complete applet, including both computation and display, to local machines as a unit. Thus, the local system must be "smarter," and network traffic is minimized.

2.2 Message Protocol

Updating the simulation state between participating applets should not be so heavy-weight as to unduly impact the performance [12]. An additional constraint for Web based collaborations is that participants may have low bandwidth connections (e.g., modems) as well as high network delays due to physical distance. Therefore, as few updates as necessary should be sent between participants.

In a synchronous protocol, the sender of a message blocks processing until a response is returned by the recipient. The response may be a simple acknowledgment or new data. A synchronous protocol is implicit in remote procedure call (RPC) and remote method invocation (RMI) [21] systems. As Mowbray and Zahavi explain, such synchronous communications facilities require added complexity using threads or callbacks when asynchronicity is desired [14]. While RPC and RMI provide a convenient abstraction for network communications programming, the processing delay imposed by the synchronous communications can severely impact the performance of the system. For example, in a semi-replicated real time groupware system that used a synchronous communications model, Graham, et al, [10] found that clients can spend up to 95% of the time blocked waiting for responses To address this, they combined memory access techniques of caching, prefetch and presend to reduce the blocked time.

Another approach involves using an *asynchronous* message protocol in which messages are sent, but no response is waited for. In many cases, such as sending updates, no response is expected. For those messages that do expect a response, a separate thread accepts them when they arrive some time after the request. An asynchronous protocol relieves the system of half the network messages when no acknowledgment is required. It also allows the sender to resume processing immediately after sending a message. This is especially important in a peer-to-peer distributed system. If an n-participant system used a synchronous protocol, the sender would need to wait for a reply from each of $n-1$ participants. In contrast, an asynchronous sender will spend only n units of time sending copies of the message. Asynchronous communications can be combined with the memory access techniques of caching, prefetch and presend to potentially realize even further gains than any of those techniques alone.

2.3 Simulation Divergence

Due to network lag in a peer-to-peer system, there is a delay from the time a simulation state change is sent and when it is received. Simulations may become unsynchronized during the delay, since participants may receive the update at different times. Measures must be taken to re-synchronize copies of the simulation. To allow the differing states to converge, a synchronization point may be introduced in the simulation so that all copies block until they share the same state and then continue. The pause caused by this approach may be unsatisfying to users in a real-time simulation. We discuss techniques to mask such pauses in Section 3.4.

2.4 Java

The Java programming language and its Application Programming Interface (API) libraries provide a platform-independent environment for creating interactive applications and applets that may be delivered over the World Wide Web [8, 5]. Java provides many standard features that facilitate developing synchronous collaborative applications on the WWW.

Java rose to popularity in part due to its applet delivery mechanism over HTTP. An `<applet>` tag in the Hypertext Mark-up Language (HTML) tells a Web browser to load specified code and reserve a screen rectangle for it. Start-up parameters for the applet may also be specified.

The WWW delivery of applets is useful for many kinds of software, but is especially relevant for collaborative applications. In many cases, people work collaboratively in spontaneous sessions. Using the Web model, a new member may join a collaborative session simply by clicking on a URL from within a Web browser, causing the user to join the session and download the needed application to the local machine. This is quite different from the situation where each a group of collaborators must arrange the collaboration in advance, and each member must explicitly initiate their portion of a collaborative environment on their own computer.

The Java API includes a set of classes to facilitate network communications. These include asynchronous UDP datagrams, creating input and output streams from TCP/IP sockets, and URL creation, manipulation, and connection.

3 A Minimal Multiuser Simulation

3.1 Ppong!

To investigate Internet based synchronous collaborative applications, we created a two-player game based on the classic Atari Pong, which we call Ppong! (pronounced "pee' pong", a contraction of "ping pong"). A screenshot of our implementation appears in Figure 1. Ppong! appealed to us as an experimental vehicle since it strips to its essentials many components of real-time collaborative simulation. If these features cannot be satisfied, more sophisticated real-time collaborations and interactive games over the Internet are unlikely.

Ppong! has two players that each control a paddle, with the players on opposing sides of a rectangle. There is a ball that moves between the players and which may be deflected in the opposite direction by intercepting the ball with a paddle. If the ball moves past a player's paddle, the opponent scores a point.

Ppong! may be considered the minimal application that an architecture for a real-time synchronous multiuser software must be able to implement. There are two users, real-time updates occur on the simulation state, and both players may affect the state of the simulation. However, Ppong! only tests some aspects of synchronous collaboration since it is inherently *turn taking*. A user may have a significant effect on the state of the simulation only when it is his turn.

Today, real-time chat and whiteboard applications are often used to demonstrate a collaborative application toolkit's capabilities [13, 15, 22]. However, these do not exercise the issues relevant to a multiuser

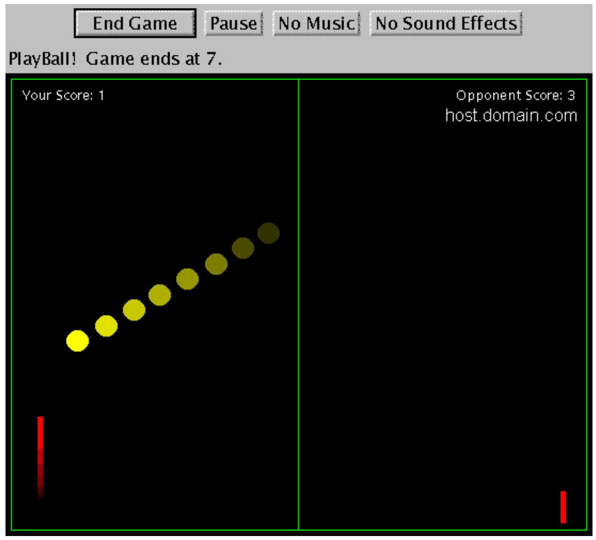

Figure 1: Ppong! game in progress.

real-time simulation. We suggest that an implementation of Ppong! become part of the standard demonstration suite of collaborative application frameworks and toolkits, alongside shared whiteboards and shared chat sessions.

3.2 Ppong! Architecture

As a general purpose programming language and API library, Java and its Network API allow both the centralized and distributed architectures described in Section 2.1. That is, applications written in Java may implement either paradigm. However, Java applets downloaded and executed within the runtime environment of a WWW browser may be restricted in functionality [6]. Whenever a potentially damaging operation is invoked (e.g., file read and write), Java asks the security manager whether to allow the operation to complete [4]. Many WWW browsers that implement Java, such as Netscape Navigator 3.01 and Microsoft Internet Explorer 2.0, restrict the capabilities of Web delivered Java applets by implementing a restrictive security manager. With respect to establishing network connections, WWW browser security managers typically disallow applets loaded over the net from creating a Socket connection to any host other than the one that originally served the applet (called the `codebase`). Furthermore, network-delivered applets may not instantiate a `ServerSocket` (a socket that accepts connections from other hosts). Applets loaded from the local file system are not restricted.

This restriction does not so much *enforce* security as it *avoids* secure implementations by blocking all potentially harmful capabilities. Since applets cannot connect to or accept connections from peers, we are restricted by the Security Manager from linking Web delivered applets in a collaborative session using a distributed architecture. The centralized architecture remains as our only choice, since the security model permits socket connections back to the originating applet server. Therefore, anyone publishing a potentially collaborative application on an HTTP server, also must establish a collaboration server on that host.

Netscape and other browser manufacturers may relax their Security Manager policies in the future, allowing the user to select security levels like JavaSoft's HotJava browser and the Java Development Kit's `appletviewer`. Additionally, the announced support for signed classes in Java 1.1 should allow Web browsers to relax security restrictions for*trusted* applets which are composed of such signed classes [16].

With this in mind, the client and server ends of Ppong! were designed to be pseudo-distributed. That is, while all messages must pass through the central server, the server simply passes the messages to the other participant. Only the clients maintain state. Since the clients appear to send and receive messages directly to and from each other, when/if the security restrictions are relaxed, it will be straightforward to connect the clients directly.

3.3 Ppong! Message Protocol

It is important that both players perceive the same game state. Equally important is that the simulation perform quickly enough to provide a challenging game. To that end, we considered it important to minimize network traffic between instances of the game (see Section 2.2).

Our approach is to allow all copies of the simulation to run simultaneously. If the simulation is deterministic, and each instance receives the same input, the copies should run to the same conclusion, although not necessarily at the same time. Generally, only **indeterminable** changes in the simulation state need be sent to all participants. Indeterminable changes are those that other participants cannot have known occurred, such as some user interaction. Any state change that occurs algorithmically as part of the simulation is **determinable**.

This solution leaves open the problem that not all participants will receive updates in the same order. If two users change the simulation simultaneously, their changes may conflict and leave the copies in different states [12]. One fix for such conflicts is to only permit one user at a time to modify the simulation. For Ppong!, this scheme is natural since only one user at a time has the opportunity to affect the simulation significantly (by hitting or missing the ball as it approaches).

There are three indeterminable changes that a particular instance of the game must send to its opponent. All other state changes are determined by the simulation algorithm.

1. **Hit** If the player hit the ball, the opponent is notified of the new ball position, velocity, and direction.
2. **Miss** If the player missed, the opponent is notified to start a new ball traveling toward the player that missed.
3. **Racket** When a player moves their paddle, an update is sent to the opponent.

3.4 Relieving Simulation Divergence

Ppong! provides a useful example of the simulation divergence between two participants described in Section 2.3. Figure 2 illustrates the following scenario. Suppose the ball is moving toward player 2's paddle on the right side of the court. When the ball arrives at the paddle, player 2 can either hit or miss. Player 1's simulation must wait for player 2's simulation to send the message. It can either pause the ball movement at that point and wait, or guess that player 2 hit or missed by moving the ball accordingly. Suppose the guess were that player 2 missed, when in fact player 2 actually hit the ball. When the "hit" update arrives at player 1, the game adjusts its state. Suddenly, the ball is moving toward player 1, despite the fact that player 1 *saw* player 2 miss. The affect is cumulative as now player 2's copy of the game simulation will run ahead of player 1's copy.

Despite this divergence, the game may be fairly played from this state. Player 1 still has ample time to respond to the ball now moving toward his paddle. However, when implemented this way, players complained about sudden dramatic changes in the state of the game that resulted from receiving an update from the opponent after the often significant network lag time between users of the Internet.

We considered three approaches to alleviate the sudden change in state. In each of these, the simulations are deliberately unsynchronized to mask the inconsistency. First, it is possible to **extrapolate** the state of the game by making guesses at indeterminable points until an update arrives. This is what occurs in Figure 2, which illustrates the problem when the extrapolation is wrong.

A second approach is to **interpolate** the game states between updates. That is, the simulation would not run past an indeterminable point. It would wait for an update and smoothly transition the game state to the state specified by the update. Therefore, no incorrect state occurs as was the case with an incorrect extrapolation. However, this

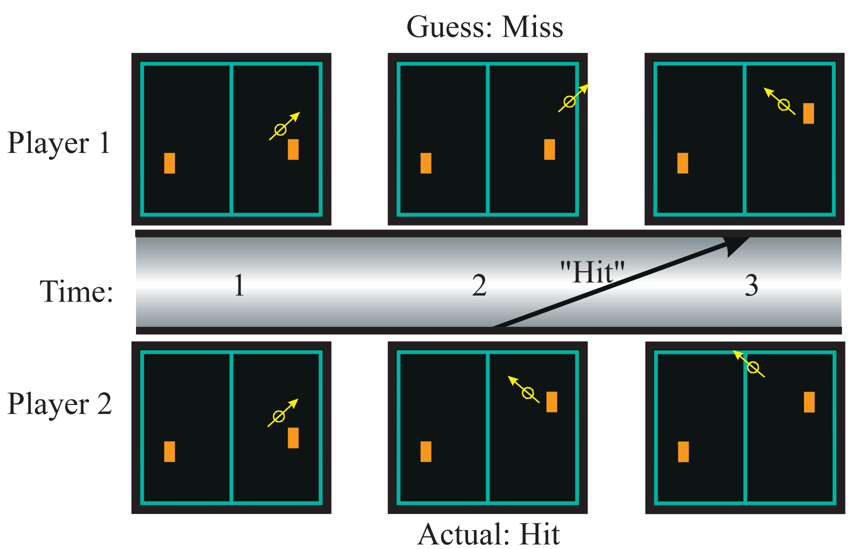

Figure 2: Simulation divergence. (1) The simulation copies are synchronized. (2) The simulations diverge. Player 1's simulation guesses that player 2 missed. Player 2 actually hits and sends update. (3) Player 1 receives the "hit" message.

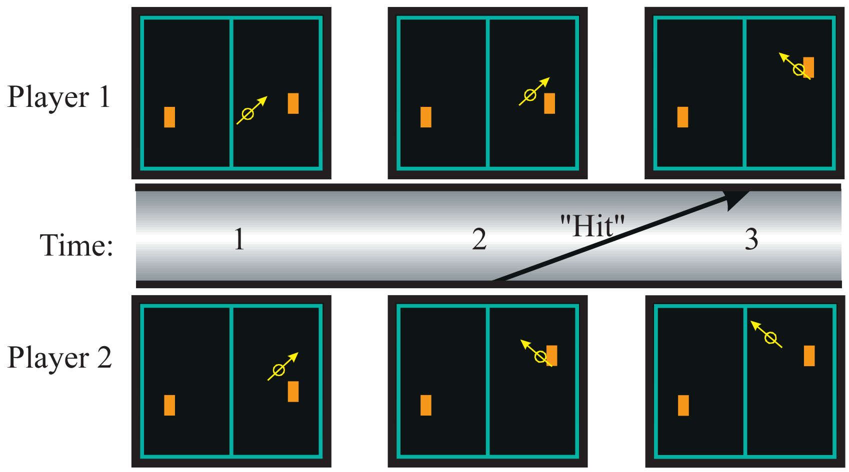

Figure 3: Simulation Retardation. (1) Player 1's simulation copy is retarded and lags Player 2's. (2)Player 2 hits the ball and sends update. The ball has not reached that point in Player 1's copy. (3) Player 1 receives the "hit" message at the point of time that the simulation needs that update information.

compounds the problem of divergence, since the simulations now suffer from the transition delay in addition to the network delay originally causing the simulation divergence.

A third approach, taken by Ppong!, is to **retard** the visual representation of the simulation for participants that are expecting to receive an indeterminable change from another participant. When the ball is moving away from a player, we use a heuristic to slow the movement of the ball slightly. This way, the opponent's copy of the game has time to either hit or miss the ball and to send the corresponding message to the first player. If the delay of the simulation is just right, the first player should receive the "hit" or "miss" at the same time that the ball appears at the opponent's paddle, as illustrated in Figure 3.

Ideally, the simulation should be retarded by an amount that will cause the simulation to reach the point of indeterminable change at the same time it receives the update. Since this cannot be known with certainty, our heuristic is based on the recent history of one-way network delays for sending messages from the originator to the receiver. If the simulation is of a nature that the point of the next indeterminable change cannot be predicted or whom the originator will be, this scheme does not apply. Ppong! meets these two criteria: (1) The point of an indeterminable change is the next time the ball arrives at a paddle line; and (2) who will generate the change (hit or miss) will be the opponent.

3.4.1 Simulation Retardation in Ppong

Ppong! continually estimates the one-way network delay by taking the average round trip delay for the last five timing tests and dividing by two. Ppong! runs the simulation in real time. That is, the speed of a player's processor affects the smoothness of the animation (frames per second), but does not affect the position of the ball.

The speed in the X direction (horizontal) is stored in variable `ball.XSpeed` and represents the number of pixels to move per second in the X direction. For each iteration of the simulation loop, the ball's current position is calculated relative to its starting point (`ball.startX`). The following algorithm (in Java) calculates the ball's X position (`ball.x`).

```
long now = System.currentTimeMillis();
long diffTime = now - startTime;
float numSeconds = diffTime/1000;
ball.x = ball.startX
    + Math.round(ball.xSpeed * numSeconds);
```

When the horizontal distance between the paddles is `W` pixels, the time to travel that distance, `T`, will be `W/ball.xSpeed` seconds (`T = W/ball.xSpeed`). With this and the one-way network delay time, `delay`, we can determine how much to retard the simulation. The simulation should take `T+delay` seconds to travel distance `W`. The new `ball.xSpeed` is determined by `ball.xSpeed = W/(T+delay)`.

Retarding the speed in the X direction is all that is required in order to synchronize the arrival of the ball at the opponent end with the arrival of the "hit" or "miss" message from the opponent. How the ball travels in the Y direction (vertically) will not affect this timing. However, we want the two players to have as similar a perception of the simulation as possible. Changing the X speed without a corresponding change in the Y speed will create a change in the ball's travel path. The Y speed should be changed by the same percent change as that of the X speed:

```
ball.ySpeed = ball.ySpeed*(ball.xSpeed/ball.oldxSpeed).
```

The following algorithm determines the Y position for each iteration of the simulation loop. It is similar to the previous algorithm for the X position, but different in that the ball reverses its Y direction when it hits a side wall. Therefore, the current Y position cannot be determined using the same `startTime` as for the X direction. Instead, the time when the ball last hit a wall is used as `startTime`.

```
long now = System.currentTimeMillis();
long diffTime = now - yStartTime;
float numSeconds = diffTime/1000;
ball.y = ball.startY
    + Math.round(ball.ySpeed * numSeconds);

// Check to see if it hit a side wall
// if so, reverse direction
if (ball.y < 0) {
    // hit the top wall
    ball.y = Math.abs(ball.y);
    ball.Starty = ball.y;
    ball.ySpeed = -ball.ySpeed;
    yStartTime = now;
} else if (ball.y > Court.height) {
    // hit the bottom wall
    ball.y = ball.y - 2*(ball.y-Court.height);
    ball.startY = ball.y;
    ball.ySpeed = -ball.ySpeed;
    yStartTime = now;
}
```

With this approach, players can successfully carry out a game, since the ball moves at normal speed when it approaches the local player's paddle. However, this approach may be unacceptable at some value of delay. For example, if the ball moves at less than half normal speed when approaching the opponent, the perceived delay may be unsatisfactory. We did not determine at what point the delay becomes more distracting than useful.

Unfortunately, determining the network delay from one instant to the next is not possible, since dynamic routing and network load change constantly. Therefore, despite having retarded the simulation, the update from the opponent may arrive before or after the time expected. If it is early, we accept the update and continue the simulation, despite the resulting discontinuity. The discontinuity could be relieved by accelerating the simulation to catch up smoothly. If the update is late, Ppong! continues to move the ball as if the opponent missed, resulting in the same discontinuity described in Section 3.4. It is also possible to stop the game and wait for the update message until exceeding the timeout period.

4 Conclusions

As a minimal capability real-time multiuser simulation, Ppong! illustrates several issues for Internet based synchronous collaborations.

Today, real-time chat and whiteboard applications are used to demonstrate a collaborative application toolkit's capabilities. However, they do not exercise issues relevant to a multiuser real-time simulation on a WAN. We propose that an implementation of Ppong! become part of the standard demonstration and test suite of collaborative application frameworks and toolkits.

Our work with Ppong! illustrates the problems with current security restrictions placed on Web-delivered applets. The necessary work-arounds required increase the development effort and result in less-than-optimal collaborative software architectures. Current limitations require collaborative systems to use a centralized server located on the HTTP host that delivered the applet to participants. As a result, each message is sent twice, first to the server and then to the opponent. These extra messages also add to the network and processing load imposed on the HTTP server, which may already be heavily loaded.

The nature of the Internet means there may be long network lag time between users. Therefore, it is important to minimize the number of messages passed among collaborators. With deterministic real-time simulations only the **indeterminable** changes need to be broadcast to participants. Indeterminable changes are those that other participating simulations cannot have known occurred, such as some user interaction.

Real-time interactive Internet-based applications must deal with significant network lag times, but without causing sudden dramatic changes in the state of the simulation resulting from receiving an update from another user. Our solution to this problem for Ppong! was to **retard** the receiver's simulation so that the receipt of an update message would more likely coincide with the point at which the update visually occurred. The simulations are not synchronized, but in practice players do not notice the inconsistency.

Ppong! Statistics From February 19, 1996 to Dec 8, 1996, the Ppong! URL has been accessed 17454 times, and 2589 games of Ppong! have been played with at least one player outside of the Virginia Tech Computer Science Department network domain. On March 28, 1996, we began to record final game scores. Of the 1768 games for which we have a score and where at least one player was outside of the Virginia Tech Computer Science network domain, in 852 games (48%), each player scored at least one point. A shutout score may indicate that one of the players was not actually playing the game, such as if one player had left her machine while waiting for an opponent.

References

[1] R. M. Baecker, *Readings in Groupware and Computer-Supported Cooperative Work: Assisting Human-Human Collaboration,* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1993

[2] J. Begole, *Ppong!*, February 1995.
URL http://simon.cs.vt.edu/~begolej/Ppong.html

[3] J. Begole, C. Struble and C. Shaffer, "Collaboration Transparency in Java through Event Broadcasting," submitted to *IEEE Internet Computing*.

[4] P. Chan and R. Lee, *The Java Class Libraries: An Annotated Reference,* Addison-Wesley, Reading, MA, 1997.

[5] D. Flanagan, *Java in a Nutshell,* O'Reilly & Associates, Inc., Sebastopol, CA, 1996.

[6] J. S. Fritzinger and M. Mueller, "Java Security" Sun Microsystems, Inc. Mountain View, CA, 1996.
URL http://www.javasoft.com/security/whitepaper.ps

[7] D. Garfinkel, B.C. Welti and T.W. Yip, "HP SharedX: A Tool for Real-Time Collaboration," *HP Journal*, April 1994, pp. 23-26.

[8] J. Gosling and H. McGilton, "The Java Language Environment: A White Paper," Sun Microsystems, Inc. Mountain View, CA, May 1995.

[9] J. Gosling, B. Joy and G. Steele, *The Java Language Specification,* Addison-Wesley, Reading, MA, 1996.

[10] T.C.N. Graham, T. Urnes, R. Nejabi, "Efficient Distributed Implementation of Semi-Replicated Synchronous Groupware," *Proceedings of the ACM Symposium on User Interface Software and Technology - UIST '96*, Seattle, WA, November 6-8, 1996, pp. 1-10.

[11] S. Greenberg, "Sharing Views and Interactions with Single-User Applications," *Proceedings of the ACM/IEEE Conference on Office Information Systems*, ACM Press, New York, 1990, pp. 227-237.

[12] S. Greenberg and D. Marwood, "Real Time Groupware as a Distributed System: Concurrency Control and its Effect on the Interface," *Proceedings, CSCW '94*, ACM, October 1994, pp 207-217.

[13] F.J. Hauck and U. Gall, "Promondia: A Java-Based Framework for Real-Time Group Communication in the Web," University of Erlangen, Technical Report TR-I4-96-07, URL http://www4.informatik.uni-erlangen.de/TR/ps/TR-I4-96-08.ps.Z

[14] T.J. Mowbray and R. Zahavi, *The Essential CORBA: System Integration Using Distributed Objects*,John Wiley & Sons, Inc. New York, 1995.

[15] NCSA, "NCSA Habanero, " June 1996.
URL http://www.ncsa.uiuc.edu/SDG/Software/Habanero/

[16] JavaSoft, *JDK 1.1 Preview,* October 1996.
URL http://www.javasoft.com/products/JDK/1.1/designspecs/index.html

[17] O. Jones, "Multidisplay Software in X: A Survey of Architectures," *The X Resource* (6), Spring 1993, pp. 97-113.

[18] J.C. Lauwers, T.A. Joseph, K.A. Lantz, L. Romanow, "Replicated Architectures for Shared Window Systems: A Critique," *Proceedings of the Conference on Office Information Systems*, ACM Press, March 1990, pp. 249-260.

[19] R.E. Kessler and M. Livny, "An Analysis of Distributed Shared Memory Algorithms," *Proceedings of the 9th International Conference on Distributed Computing Systems,* 1989, pp. 498-505.

[20] Microsoft, Inc., *Microsoft NetMeeting Platform*, October 1996.
URL http://www.microsoft.com/workshop/prog/msconf/msconf.htm

[21] Sun Microsystems, Inc., *Java Remote Method Invocation*, Revision 1.0, Draft October 2, 1996.
URL http://chatsubo.javasoft.com/current/doc/rmi-spec/rmi.ps

[22] TESSERACT Information Solutions Inc., "RealTime Collaborative Groupware for JAAS - Intro," May 1996.
URL http://www.tisinc.com/realtime/