Constructing Client-Server Multi-Player Asynchronous Networked Games Using a Single-Computer Model*

Daniel M. Zimmerman, Brian Rothstein,
Yevgeniy Kaganovich and Khai Pham

Computer Science 256-80
California Institute of Technology
Pasadena, California 91125
{*dmz, brianr, ymk, khai*}*@cs.caltech.edu*

1 August 1997

Abstract

We examine the process of creating asynchronous networked games by applying systematic transformations to their single-computer analogues, identify the need for such transformations, and propose a simple system of rules for them. In developing these rules, our primary concerns are comparing the flow of events in single-processor and networked games and examining the restrictions and limitations resulting from speed considerations. Although this paper only discusses games, the transformation rules may apply to any networked application with asynchronous data input and exchange.

Keywords: asynchronous message passing, networked multi-player games, distributed systems, client-server model, Java

*This work was supported under the Caltech Infospheres Project. The Caltech Infospheres Project is sponsored by the Air Force Office of Scientific Research under grant AFOSR F49620-94-1-0244, by the CISE directorate of the National Science Foundation under Problem Solving Environments grant CCR-9527130, by the NSF Center for Research on Parallel Computation under Cooperative Agreement Number CCR-9120008, and by Novell, Inc. and Parasoft Corporation.

1 Introduction

Asynchronous networked game programming is a new and exciting application of the Internet. While networked games are more interesting in terms of both gameplay and programming than their single-computer counterparts, they present many design complications [1]. These include fairness and synchronization, which are unique to networked games, and game speed, which ascends to a new level of complexity as a result of message passing and network speeds.

In response to the increasing demand for simplifying the construction of networked games, a few models have been proposed to deal with the problems of distributed programming and asynchronous message passing [3, 5]. Our model centers around a transformation from a multi-player, single-computer version of the game, in an attempt to provide an easier transition to the realm of networked games.

Single-computer multi-player games have two advantages over networked games: first, all players see the same display in a single-computer game, so everyone gets every piece of game information simultaneously; second, competing inputs are always processed in the correct order in a single-computer game because there is only one computer to arbitrate among the inputs.

The method we present in this paper makes it possible for networked games to have these properties too. In our model, a central server receives all event data from each player on the network, arranges the events in their proper order, and then broadcasts them to the players using a messaging protocol which preserves message ordering. The events received by the clients are then in the correct order, and can be processed just as in the single computer version.

The transformation rules presented are a set of clear and simple guidelines which help the programmer to correctly identify and handle the issues listed above. At the same time, they are flexible enough to be applied to most games. To test the viability of the proposed transformation rules, we have successfully applied them to three different networked games, which will later serve as examples of how our transformations apply to games with widely varying communication and gameplay requirements.

We begin by introducing some concepts and definitions which are used frequently throughout this paper. Next, we discuss the considerations which must be dealt with in designing networked asynchronous games, present our transformation rules in detail, and describe the successful application of these rules to three different games. Finally, we compare our method to others which have been proposed, and present our conclusions.

2 Concepts and Definitions

In order to devise a set of transformation rules for multi-player games, we make use of several conceptual and programmatic constructs. In this section, we describe these constructs and their functions.

2.1 PlayerTable

In a single-computer game, all players' states are controlled by the single computer. The programmer need not concern herself with consistency of states, because there are no unanticipated state changes being made. In a networked game, however, states may be changed simultaneously on many different computers. We use a construct called the *PlayerTable* to ensure consistency of states in a networked game. In addition to state information for each player, a PlayerTable contains contact information which allows this state information to be communicated between players and the central server. The server keeps the main PlayerTable for the game – when a single client makes a change to its copy of the PlayerTable, this change is sent to the server, which broadcasts it to the remaining players.

An entire PlayerTable can be sent over the network as a single message, which allows players who join the game late to be easily "brought up to speed" with respect to players already in the game.

2.2 Net Event

Asynchronous games, like many other applications, are event-driven – the gameplay results from players' actions. In a single-computer game, an event is synonymous with a change in the state of one of the players. A *net event* is the extension of this concept to a networked game, where a change in one player's state must be communicated to the other players on the network. Net events generated by a player are communicated using the contact information in that player's PlayerTable. A net event which modifies the game state is called a *shared event*.

2.3 Net Event Handler

A *net event handler* is a process that listens for net events and converts them into local events to be processed by the player's machine. If a language with thread support, such as Java, is used for programming, multiple net event handlers in separate threads can listen for incoming net event transmissions, which can improve communications efficiency.

2.4 Dispatcher

A *dispatcher* is a specific type of net event handler which listens for net events containing requests to join or leave the game. It is useful when the players are allowed to enter or leave in the middle of a game. The dispatcher on the server may allow or disallow the join/leave, and then broadcast the appropriate net event to the dispatchers on the other clients. It is possible for the programmer to disable the dispatcher in situations where it is undesirable to have players enter or leave the game (for instance, the middle of a hand of poker might be an inopportune time for a new player to join the table).

2.5 Game Clock

The players in an asynchronous networked game are likely to have network connections of varying speeds. In games with minimal synchronization, therefore, a *game clock* is necessary to ensure that no player is at a disadvantage as a result of variations in network performance. For instance, in a networked quiz game, if one player were to buzz in with the answer, it might be the case that the player had seen the question for a longer time than other players because the question message got to him first. To overcome this problem, each player keeps a local game clock. In the quiz game example, the clock starts when the question message is received and its value is sent to the server when the player buzzes in. When the server receives the buzz-in message, it then sends a request to all other clients with the time indicated on the buzz-in message. Each client responds by checking in with the server if they have not sent a buzz-in message by the time the specified time is reached. Once the server has received either a buzz-in message or a check-in message from every client, it knows that every player has had an equal opportunity to buzz in, and can thus determine the true winner based on the buzz-in message with the minimum clock time.

2.6 Shared Game Object

A *shared game object* is an object within the game whose state can be changed or observed by more than one player, simultaneously or at separate times. One example of a shared game object is the ball in the Pong game (described in Section 5.3), whose state can be changed by all players at separate times but is observed by all players simultaneously.

2.7 Danger Zone

Each game state in which multiple players need to access a shared object simultaneously, or in which multiple players are interacting directly with each other in a timing-critical way, is called a *danger zone*. The number and frequency of danger zones varies from game to game – in the Pong game, for example, danger zones occur only when a ball is close to a paddle. Within its danger zones, a game absolutely *must* be synchronized to prevent the creation of divergent game states.

2.8 State Buffering

To decrease message bandwidth, it is sometimes possible to allow a small divergence in the game state until a point where this divergence could cause a violation of the game rules. We call this *state buffering*. For instance, in the Tron game (described in Section 5.2), the player could send events every five moves, as long as it is not within five moves of colliding with any other player. Since what any particular player is really seeing with such state buffering is the other players' positions from five moves ago, it is possible for the other players to be anywhere within a five move radius of their positions on the player's screen.

3 Game Design Considerations

Networked games differ in the restrictions placed on them by the game rules and the allowable player interactions. Before a networked game can be effectively coded, possible performance bottlenecks resulting from particular aspects of the game must be identified and dealt with. Each of these presents a design problem for the programmer and must be considered *before* any coding takes place. In this section, we examine the design problems which arise in networked games, and discuss possible solutions to these problems.

3.1 Communication Volume

While modern computers are powerful enough to seamlessly handle most game-related computations, the amount of data communicated between players and the server is the most important restriction on networked game performance because of network latency. We call this amount of data, which should be the main factor in determining what information needs to be sent over the network and how it should be encoded, the *communication volume*. For games requiring a very small communication volume, it may be sufficient to communicate all known state information (thus greatly simplifying the game programming), but, more often than not, a game programmer will need to choose a subset of the state to be communicated.

The programmer must keep the communication volume in mind when choosing the communication protocol to be used by the game. While using a high-level communications layer may greatly simplify the programming process, some communication-intensive games require reverting to lower level protocols such as TCP and UDP. For instance, we have successfully used the communications layer provided by the Caltech Infospheres Group's *info.net* package [4] for two of our example games, but found it inadequate for the third, for which we used UDP directly.

The communication volume also determines the amount of data stored in the PlayerTable. For games with high communication volume, the main PlayerTable may be subdivided into smaller ones if there are states in which it is reasonable to broadcast only partial information.

3.2 Communication Density

Just as communication volume differs from game to game, so does *communication density*, the amount of communication which must take place within a certain amount of time. The communication density of a specific game, moreover, may vary wildly at different points during gameplay - even games with low average communication density may have some states with extremely high communication density, where more information than usual must be exchanged between clients. In a Pong-style game, for example, the states where the ball is close to a player's paddle require much more communication than the states where the ball is in the middle of the screen. If not handled properly, points of high communication density may cause an unacceptable decrease in game speed.

The programmer should try to make the communication density of a game as uniform as possible, as a way to minimize "spikes" of high communication density. One way to accomplish this is by subdividing the PlayerTable; another is to use state buffering as much as is permitted by fairness and synchronization considerations (described below).

3.3 Synchronization

Synchronization is one of the limiting factors in the optimization of communication density. The rules of any asynchronous game require synchronization to occur at some states, even if this synchronization is only to determine who wins. If the players are not already synchronized at that point, as when state buffering takes place, the buffering causes the communication density to increase. Hence, the frequency of synchronized states is a restriction on the usage of state buffering.

Synchronization is also important in terms of timing. Message delay is a big problem in asynchronous games, because the outcome of a player's actions often depends on the timing of these actions with respect to those of the other players. For example, in the Set card game only the person who first claims the set may get the point for finding it. However, we assume that for the game to be fair, the "speed" of his response should not depend on the speed of his connection or processor. In this case, synchronization is handled using a game clock to ensure that each player gets an equal opportunity to find the set.

3.4 Quality of Service

Quality of service considerations come into play when dealing with players who inhibit the gameplay significantly more then the rest. This may occur if a player is joining/leaving the game, or if he has problems with his connection or computer. These decisions are largely game-dependent, but may involve timeouts or special game states. For example, timeouts for slow/broken connections and time adjustments for players who entered the game in the middle of a hand (and are thus at a disadvantage compared to other players) are implemented in the Set game.

3.5 Communication Scoping

In some situations, not all players need to be notified of a certain net event. The restriction of communications such that only a subset of players is notified about a particular net event is called *communication scoping*. If a game requires unusually high communication volume for every net event, or if there are a large number of net events which are only important to one or two out of a group of players, communication scoping can save a lot of time and improve game responsiveness. For instance, a game where players are not always in each others' fields of view might benefit greatly from communication scoping, since unobservable movements would not need to be communicated as frequently to ensure state consistency.

4 Transformation Rules

This section details a set of transformation rules which take a single-computer multi-player game and produce a networked multi-player game.

1. **Save game code twice, as GameServer and GameClient**

   The first step in writing a client-server application is to have both a client and a server. Both the client and the server need to know some of the game rules, and transforming them both from the original source code of the single-computer game is the easiest way to give them this knowledge.

2. **Define shared state information to be stored in the PlayerTable**

   In any multi-player game, players need to know certain information about each other when they join the game (this may include other players' names, identifying numbers, positions in the game, and scores). All of this information should be stored in the PlayerTable, so that a player joining the game can learn the entire current state of the game just by receiving the PlayerTable.

3. **Implement net event handlers to communicate net events**

   This is the key to transforming the single-computer game into a networked game. In the single-computer model, when a user interacts with the game, an event is posted to that computer's event handler. In a networked game, events are posted to net event handlers when players interact with their respective clients.

   It may be desirable to have multiple net event handlers for multiple net event types. For instance, one could implement a membership handler, which handles all events generated when clients join or leave the game, and a game event handler, which handles all events related to actual gameplay. The use of multiple net event handlers allows for automatic sorting of net events, which reduces the overhead required to sort messages; however, each net event handler requires its own thread, which increases thread-related overhead on each client system.

4. **Determine which events need to be communicated to other clients and the server**

   Some local events in a game may not need to be shared with the server or the other clients, and can instead be handled solely by the local game client. These events include invalid game actions and any events which do not modify the state of the game. For instance, if a player wanted to look at her game statistics, the other clients would not be thrown into a divergent game state if they were not informed of this local event. To reduce communication volume, only events which change the game state should be communicated to the server.

5. **Determine which information needs to be centralized**

   Centralized information is data which the server needs to maintain and broadcast to the game clients. For instance, in a card game, it might be desirable to have the server keep track of the deck so that it can shuffle it and then deal cards to the clients. However, to save on communication volume, each client could have a copy of the deck. The server could then give each client a random number seed at the beginning of the game, and each client would shuffle the deck using this seed and deal the appropriate cards based on previous game events and the game rules. The use of previous game events and game rules to change or define the game state without receiving a message explicitly detailing the new state is called a *derived update*. As much as possible, the client should be written to use derived updates, because this minimizes the amount of communication between clients and the server.

6. **Have clients send all shared events to the server**

   Clients must send all shared events to the server, which will then send the events to all other clients and thereby keep the game state consistent among all clients. The server keeps track of the game state in the same way as the clients, by watching for shared events and using the game rules - it can therefore bring new clients up-to-date with the game state quickly, by sending them the current PlayerTable and any other centralized state information resulting from shared events.

7. **Determine the danger zones**

   Whenever a client enters one of the game's danger zones, it may become necessary for it to cease operations until it receives an update of the game state. This is necessary when a shared game object is being modified by a remote player. The client only knows what the remote player has done up until the last net event it has received from that player, but the remote player may have sent more event messages that have not been received. Before the client lets the remote player modify the shared object, it needs to determine exactly what the remote player is doing. This may present a bottleneck, and must be considered in the transformation process. Once the danger zones of the game, if any have been identified, there are various methods for handling the necessary synchronizations.

   One method, which may result in unacceptable overhead if danger zones are frequent, is to simply synchronize on every game iteration which takes place within a danger zone. Another is to stop play on all clients except the one which can potentially modify the shared object, which then records everything it does with the shared object and broadcasts an update to the other clients. Also, in some games, if a client sees that a shared object is headed for a danger zone it can slow the movement of the shared object on the local display until it receives the update message, so that there is a seamless transition rather than an abrupt halt while waiting for synchronization.

   If two or more players' danger zones overlap, then the only way to maintain a consistent state is to have these players synchronize on every game iteration. No single player can be responsible for recording the state, since multiple players could have had an effect on it.

Only a certain subset of games can really be classified as having clearly defined danger zones. Some games have the entire playing field (or nearly the entire playing field) as a synchronization region, and must therefore synchronize on every game iteration. Such games are the hardest to implement effectively in a network environment, because they require a very large communication volume.

5 Example Games: Set, Tron, Pong

5.1 Set

Set is a card game in which twelve cards are laid out on a table and players examine the cards trying to find a set of three which meets certain requirements. When a player spots a set, she calls it and has five seconds to pick up the cards comprising the set. If the cards are actually a set, she keeps the cards. If she is incorrect and they are not a set, or if her time expires, then she must put the cards back and also put three cards from her pile (if she has one) back into the deck. The winner of the game is the player who has the most cards in her pile when there are no cards left in the deck.

Set can be implemented as a single-computer game by giving each player a key to press when she wants to call a set. The PlayerTable contains each player's name and a list of the cards each player has won. When a player presses her key, the game announces who called the set and allows her to click on the cards comprising the set. If the player runs out of time or picks an invalid set of three cards then the game announces that the cards were invalid and leaves them on the table. It also puts three of the player's cards from his pile back in the deck and shuffles the deck. If the player gets a valid set of three cards, the game takes those cards off the table and puts them in the player's pile, and then deals a new set of three cards onto the table. When there are no cards left in the deck and no valid sets of three cards on the table, the game is over.

To convert this into a networked game, we first save the single-computer version as both the client and the server. As stated previously, the PlayerTable consists of the players' names and piles of cards. Since the PlayerTable construct was used in the single-computer version of the game, it is easier to transform the single-computer version to a networked version.

We use two net event handlers to handle all network messaging. One of the event handlers deals with players joining and leaving the game at any time. The other event handler deals with all other game actions. We determine the net events necessary for the game to be “Player Joins”, “Player Leaves”, “Player Claims Set”, “Player Checks In”, “Player Clicks Card”, “Player Runs Out of Time”, and “Server Deals Cards”. The join/leave event handler handles “Player Joins” and “Player Leaves” messages, and the action event handler handles all other game actions.

The client needs to share three events with other clients: “Player Claims Set”, “Player Clicks Card”, and “Player Runs Out of Time”. However, these events don’t need to be shared all the time. If a player has not claimed a set, she isn’t allowed to click on a card and therefore “Player Clicks Card” events don’t need to be shared with all other clients. Also, if a player is in the process of picking up cards, then no other player is allowed to claim a set and therefore “Player Claims Set” events don’t need to be shared.

Other changes in game state can be derived from these shared events. For instance, the server does not need to inform all the clients when one player has picked an invalid set of cards, because all the clients have received the three Player Clicks on Card messages and can determine for themselves whether the set was valid or not. If it was valid, they can remove the cards from the table and put them in the player’s pile without receiving an explicit message instructing them to do so, and then wait for the next three cards to be dealt. If the set wasn’t valid, the clients know to leave the three cards on the table and take three cards out of the player’s pile.

A player running out of time is something of which the local client needs to inform everyone, because only the local client can be sure that five seconds have passed without the player clicking on three cards. If other clients tried to determine that five seconds had passed before the player had clicked on three cards, they might do so incorrectly because a “Player Clicks on Card” message might still be in transit.

When the server receives game action messages, it forwards the messages on to the other clients and also keeps track of the game state itself. That is, when it receives a “Player Clicks on Card” message, it actually modifies its own copy of the game state to mark this event. This way, the server maintains the same state as the clients and is able to determine, using the rules of the game, when it should deal cards, remove cards, or take other game actions.

If the server didn't keep track of the game state itself, extra messages (such as requests from clients for cards to be dealt) would be necessary.

The server disallows joining the game (by suspending the join/leave event handler) when a player is in the process of picking up cards from the table. This makes it unnecessary to keep currently selected cards in the PlayerTable state information. If a client could join while another player was selecting cards, this client would need to know which cards were selected when it joined so that it can have a consistent game state.

The shared game object in Set is the card table. Players can affect it when they claim a set, so the clients need to synchronize whenever anyone tries to claim a set. This is implemented in a fair way using a game clock. Each client has a timer which keeps track of how long the current set of cards have been displayed (the time since the last three cards were dealt). When one of the clients attempts to claim a set, it sends the time on its game clock to the server as part of its "Player Claims Set" message. Each of the other clients then sends a "Player Checks In" message when its clock reaches this time (or if the clock has already exceeded this time) if the player has not attempted to claim a set herself. When the server receives either a "Player Claims Set" or "Player Checks In" message from every client, it determines the actual winner to be the client with the minimum clock value on its "Player Claims Set" message and gameplay continues.

5.2 Tron

Tron is a game in which players traverse the game board, leaving behind an impenetrable trail which no player may cross. If a player hits any trail, including her own, she loses. To implement this as a single-computer game, each player is assigned a different set of keys with which to control his movements.

To convert this into a networked game, we first save the single-computer game as both the client and the server. The PlayerTable contains the players' names and scores (how many times they've won by being the last left alive).

As in Set, we use two event-handlers. One handles players joining and leaving the game, and the other handles all other actions. Joining and leaving are disallowed in the middle of a game round, so players' positions and directions do not need to be stored in the PlayerTable. The only net event is "Player Turns", which contains the direction in which a player turned. All other gameplay information can be derived using this net event, since each player moves at a constant rate in the direction she was last facing. Therefore, each client can independently determine when collisions occur and handle them appropriately.

The server's main responsibility is to handle the joining and leaving of clients. The server only allows clients to join and leave between rounds; it keeps track of the game state in the same way as the clients, so it always knows when a round is over.

The main problem with Tron is the amount of synchronization required to maintain consistent game state across clients. If a client simply sends "Player Turns" messages indiscriminately, they can arrive at any time and the other clients will see that player turn at various different times, producing divergent game states. The shared game objects in Tron are the players' trails, so every time a player turns (or doesn't turn), she is affecting the game state by extending her trail. This would seem to indicate that players need to synchronize on every game iteration. To reduce communication volume, though, we note that a player's next move is only critically important if any other player is within one move of that player, because only then could it possibly cause a collision.

The implementation can take advantage of this by synchronizing only every n moves, where n is an arbitrary constant which will be chosen to give acceptable performance. This means that after every n moves, the player sends his last n moves to the server as a single message. The resulting movement of the player on other clients can be made seamless by displaying the n moves gradually (one per iteration). This means, however, that each client is actually seeing what the other players did n moves ago. If two clients are within n moves, they need to synchronize more closely to make sure that each player is not actually running through the other player's trail simply because the n move buffer hasn't caught up. The danger zone for Tron, therefore, is when two players are within n moves.

5.3 Pong

Pong is a game where multiple players control paddles situated on the sides of the screen. The goal for each player is to keep a moving ball from passing her paddle, by maneuvering the paddle such that the ball bounces off it. For this game, our transformations proceed along almost exactly the same lines as Tron. First, we write the single-computer game, with each player having a different set of keys. Next, we save the game as both the client and server.

Again, the PlayerTable contains only the players' names and scores.

The only action message is "Player Moves Paddle", and the shared game objects are the paddles and the ball. The position of the ball can be calculated based on its previous position and trajectory, taking into account the positions of all paddles, which can be calculated using the "Player Moves Paddle" messages. However, a client doesn't even need to know the exact paddle position for the other players unless it is possible for the ball to actually hit one of the paddles. Therefore, synchronization of clients need only occur when the ball enters the small strip of play area where a player's paddle may be. The danger zone for Pong, therefore, is that small strip of play area. The client of the player controlling this paddle records all "Player Moves Paddle" net events while the ball is in this region and synchronizes with the other clients. This is quite different from Tron, where every single movement made by each player must be communicated to the other players.

6 Comparison to Other Methods

The need for systematization of networked distributed programming is discussed by other authors in [3, 7]. The problems of network delays [1] and increased programming complexity [5] are also addressed elsewhere.

The process of transforming a single computer application into a networked one is examined in [5], mainly with regard to already existing applications. However, this examination largely neglects the benefits of this approach, instead regarding it as a problem that needs to be overcome. We, on the other hand, believe that this transformation technique is quite useful, and that the reason it has presented difficulty in the past is that most programs to which it has been applied were not specifically designed with such extension in mind.

Because of that, we have found the existent proposals of single-computer to multi-computer transformations to be rigid and unpermissive. The predominant attempt in the field has been to automatize the transformation as much as possible by creating an "absolute data sharing" model, as in [3], where all state information is shared by broadcasting every state change among all participating parties. We feel that this approach, while suitable for some specific applications, is impractical for most multi-player games, mainly because of speed constraints.

Many distributed computing models employ techniques similar to our net event handling. For instance, CORBA [7] technology employs stubs and skeletons to isolate the transformation of network communications into local data.

James Begole, of Virginia Tech, favorably compares the event broadcasting model to display broadcasting in [3] and suggests a view of a game as a sequence of states for each player in [1]. He has also identified the event-driven nature of the games and has developed a "retardation" technique [1], similar to the state buffering described in our paper. We have used these and other concepts to form a logical *system* of transformations, presenting an organized and intuitive method for identifying and employing these concepts.

Many of the concepts we have defined have already been implicitly utilized in other applications, but have not previously been clearly formulated or organized. In fact, many existing applications and frameworks can be used to actually implement the ideas presented here.

7 Conclusions

In this paper, we have described a process to systematize and simplify the construction of networked asynchronous games. Our approach relies on producing a single version of the game, which is easier to create in terms of programming and design, and subsequently transforming it into a networked client-server game. To that end, we have developed a set of constructs and transformation rules to address the issues which arise in multi-player networked gaming. Our constructs allow vital game information to be viewed as local events, so that the need for new code, and therefore the possibility of introducing bugs and other problems into the game, is minimized.

These transformation rules help the programmer by allowing him to implement synchronization points, shared states, and network events as extensions to the single computer version. At the same time, they are formulated so as to allow for game-specific optimizations and to not impose unnecessary limitations on the program design.

The usefulness of the transformation rules was demonstrated in the conversion of three different games, each representing a different degree of message and synchronization complexity. Our method, therefore, provides a convenient framework for the conversion of single-computer multi-player games into a networked client-server games, regardless of game complexity.

References

[1] Begole, J., and Shaffer, C.A. (1997). Internet Based Real-Time Multiuser Simulation: Ppong!. Technical Report, Virginia Tech Department of Computer Science.

[2] Begole, J., Struble, C.A., and Shaffer, C.A. (1997). Leveraging Java Applets: Toward Collaboration Transparency in Java. *IEEE Internet Computing*, volume 1, number 2, pp. 57-64.

[3] Begole, J., Struble, C.A., Shaffer, C.A., and Smith, R.B. (1997). Transparent Sharing of Java Applets: A Replicated Approach. In conference proceedings – the 1997 Conference on User Interface Software and Technology (UIST'97).

[4] Chandy, K.M., Kiniry, J., Rifkin, A., and Zimmerman, D. (1997). The Infospheres Infrastructure User's Guide. Technical Report, California Institute of Technology.

[5] Crowley, T., Miazzo, P., Baker, E., Forsdick, H., and Tomlinson, R. (1990) MMConf: An Infrastructure for Building Shared Multimedia Applications. In conference proceedings – the 1990 Conference on Computer-Supported Cooperative Work (CSCW'90).

[6] Grandmaster Technologies Corp. (1996). CyberSite Internet Collaboration Engine, URL `http://www2.tcc.net/CyberSite`

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

[8] Wilson, A. (1997). An Internet Game Server in Java. Web Techniques Magazine, March 1997.