Scalable Consistency Protocols for Massively Multiplayer Games

Pavan Edara, Rajesh Krishna Balan, and Anindya Datta
Singapore Management University
{pavanedara, rajesh, anindya}@smu.edu.sg

Abstract

The Massively Multiplayer Game (MMOG) market, exemplified by games such as World of Warcraft, is a substantial and rapidly growing subsegment of the multi-billion dollar Multiplayer Online Game (MOG) industry. A key challenge in MOGs is providing real time response latencies to the large number of concurrent game players. This has attracted substantial commercial and research attention, resulting in widely used methods such as *dead-reckoning* and *local-lag*. The problem with scaling these approaches is well known however, and it is widely accepted that they simply will not work in the scale required for MMOGs. In this paper, we suggest a variety of ways to manage consistency in MMOGs with the objective of providing desired latency levels while still ensuring that every player has a "good-enough" view of their game environment. The contribution of this paper is in presenting a family of consistency protocols for MMOGs that address two key challenges: (1) when an event occurs, how do we figure out the set of clients that need to be informed of this event?, and (2) how do we schedule the dissemination of updates to clients such that latencies and consistency requirements are satisfied? We show, through analytical methods and simulations, that our protocols are (a) "optimal" in computing the set of clients that need to be informed of an event, and (b) that they can scale to a large number of clients while still satisfying all latency and consistency requirements.

1 Introduction and Background

Multiplayer Online Games (MOGs), such as Half-Life [28] are collaborative games with simultaneous participation by multiple players. In these games, both the players and the game objects controlled by the players are mobile, making *view consistency* an important requirement. Such consistency requires that players must have consistent views of game objects, i.e., if a game event causes an object to change state (move), other players much be made aware of this change quickly. However, due to the various latencies associated with component systems of such games (e.g., network latency on the Internet and computational latency on the game servers), a player of a MOG may perceive significant delay in receiving updated state information of game objects as they move around. The rule of thumb in this community is that update latencies greater than 200 ms are unacceptable. To reduce the effects of such delays, certain restrictions are imposed on the game; (a) players can only attack enemy objects and not each other for instance, or (b) enemy objects can move minimally when attacked, to name a few. Clearly, such rules impose significant reductions of game features and participant experience.

While current commercial MOGs have player loads from the tens (e.g., Quake 3 and Half-life) to possibly the low hundreds , a class of games called Massively Mutiplayer Online Games (MMOGs), such as World of Warcraft [8] and Final Fantasy XI [25], are played by hundreds of thousands of concurrent users. Clearly, consistency maintenance gets exponentially harder in these games. For example, in Final Fantasy XI, a player usually receives position updates of other players with almost a second delay [20], severely limiting the interactivity of the experience. Indeed, it is well accepted that providing consistent views to players of MOGs and MMOGs is a difficult problem and developing scalable consistency maintenance mechanisms for MMOGs is even harder. Existing consistency mechanisms in both commercial games as well as in work reported in research literature have either borrowed from extensive available work in distributed systems consistency maintenance, or, very recently attempted specialized mechanisms. We summarize these approaches below.

Consistency control in distributed applications has been explored from a variety of dimensions, including distributed shared memory (DSM) systems [2, 4, 18, 21, 31],, database systems [1, 7, 9, 15, 30] and, most relevant to this work, collaborative visualization systems [11]. This last class of systems allow geographically separated users to access a shared virtual environments to visualize and manipulate datasets for problem solving – much like in the case of MMOGs. Examples of collaborative visualization occurs in a variety of physical environments such as fluid dynamics [13], medical data visualization [29], and collaborative editing [26]. A unique characteristic of collaborative applications is the need to distribute state updates to remote sites over the network to update the state of shared objects at these sites. Because of delivery latencies, different remote sites may receive data with varying delays, causing view discrepancies. This problem is almost exactly the same consistency issue that arises in multiplayer games, with one critical difference. In the traditional collaborative visualization case, solutions borrowed from DSM systems and databases, still work well, *as long as they ensure that the updates arrive at the different replicas in the correct order*. This is because the interarrival time of updates are usually large compared to network and computational latencies.

However, MMOGs belong to a class of collaborative application that involve *time-dependent data* and *continuous user interaction*. Thus the update streams in these applications are *continuous* in nature [20, 24] and it is required that these updates are presented to the users with either no delay, or with very stringent short deadlines (100 - 200 ms). Early attempts to address this problem has resulted in reports describing this problem and its impacts on interactive applications [5], or rudimentary solutions like *user side adaptation methods* (inform the user of the delivery latency and let him figure out how to factor it in) or *system side adaptation*, where the system attempts to abstract this delay away from the user by using various methods. The most popular of these methods is *dead-reckoning* [3, 12, 23], that attempts to predict the motion of game objects and present to users a *predicted future state* of the object. None of these methods, are exact, and don't work very well in precise game scenarios.

Very recently, specialized consistency control and synchronization methods for multiplayer games have been proposed in [20, 22]. This is perhaps the most relevant research in this area. In [20], the authors propose a reference simulator for each dynamic object in a game running at the server. Each of the clients interested in that object executes a "gradual synchronization process" on its local copy of the object to align its motion to that of the reference simulator running at the server. While making significant enhancements to the state of the art, this method suffers from high error during the period of an occurrence of an event and the communication of this event to the relevant clients.

In [22], the authors extend their work in [20], and propose a "trajectory preserving" synchronization method that reduces these errors. However this assumes connection oriented network protocols and still leaves one important question unanswered: when an event occurs how does the system determine which clients are interested in this event? Also, these mechanisms ignore the "real-time" nature of update delivery – given that different clients have different view parameters (e.g., clients might have "fuzzy" regions of visibilities) and have tight deadlines by which they must receive updates, these protocols do not emphasize meeting of this deadlines or satisfying the appropriate view parameters.

In this paper we suggest a family of consistency protocols for multiplayer games that explicitly addresses the two issues of concern when enforcing consistency: (a) when an event occurs, how do we figure out the set of clients that need to be informed of this event?, and (b) how do we schedule the dissemination of updates to meet both update receipt deadlines, as well as view parameters. In doing so, we end up proposing one of the first "realistic" consistency control approaches for multiplayer games, that takes into account not only the issue of collaborative visualization, but also game specific visualization parameters as well as the attendant stringent latency requirements. Our protocols have the following properties (a) they are "optimal" in computing the set of clients that need to be informed of an event, and (b) they scale significantly beyond the current state of the art in multiplayer games.

This paper is organized as follows: Section 2 presents the system model of MMOGs. Section 3 explains the components of consistency maintenance. In Section 4 we describe our update dissemination algorithms. In Section 5, we compute theoretical upper bounds for the update dissemination schemes. Section 6 describes the experiments that were performed using our consistency maintenance mechanisms. We present our conclusions and directions for future work in Section 7.

2 System Model

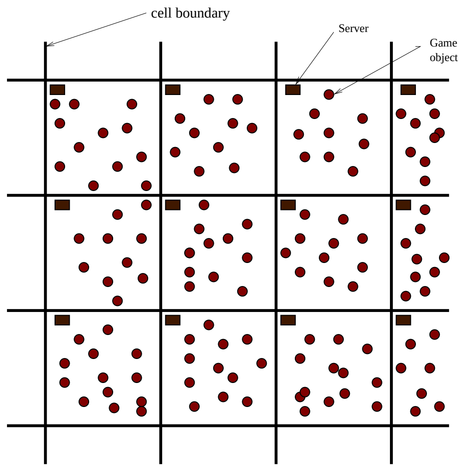

Figure 1: View of the Game Space in an MMOG

In this section we provide an overview of the system model of a multiplayer game used in this paper, which is consistent with models available in the literature [6, 10]. In these, the universe (the entire game space) is modeled as a (possibly unbounded) geospatial area, where every location is uniquely identifiable by its co-ordinates. In a MOG, multiple players, and game objects controlled by these players, are situated at various locations across this game universe. Note that henceforth we use the term *objects* to refer to both players and other game objects controlled by these players, without any loss of generality. The universe is divided into contiguous, non-overlapping cells. Objects are mobile and enter and leave cells freely. Figure 1 depicts the above scenario. Each cell is associated with a game server, which, manages the game activities in that cell. The game server performs multiple management tasks by making control decisions and communicating these decisions to players and other game servers.

Players, or *clients*, perform activities in the game universe. The two most important aspects of a client, with respect to this work, are its *mobility*, and its *view* of the universe. While mobility simply means that clients and objects controlled by clients move about the game space, the notion of a client *view* deserves explanation. View refers to how a client is made "visually" aware of the game universe and is typically based on the notion of *region of visibility* (ROV). Typically, a client is not made aware of the entire universe; rather it is shown slices of the universe with varying degrees of clarity. This is shown in Figure 2. Here the client view consists of 3 concentric circles of radii r_1, r_2 and r_3. All objects inside the inner r_1 circle is completely visible, while objects in the two outer rings are exposed with increasing fuzziness. Formally, the ROV of a client consists of x concentric circles, with radii $r_1, r_2, ..., r_x$, and may be represented as a set of 2-tuples $\{[r_1, f_1], [r_2, f_2], ..., [r_x, f_x]\}$. With each r_i is associated a *fuzziness factor* (FF) f_i, that denotes the degree of clarity of the associated view. In the innermost circle with radius r_1, it is common for the client to have complete clarity, meaning f_1 is typically 0. This is achieved by the server sending the client *every update* that occurs in that region. In the outer rings, corresponding to the radii r_2 through r_x, the server provides progressively lower clarity, signifying $f_1 \leq f_2 \leq ... \leq f_x$. This is achieved by the server not sending the client every update in that region. How many updates the server needs to send the client is determined by the value of the FF for a specific region. For instance, in Figure 2, in the ring associated with radius r_3, the FF is f_3, indicating that the server need only disseminate a fraction $(1 - f_3)$ of all the updates that occur corresponding to this region. In Figure 2, the values of f_1, f_2 and f_3 are 0.0, 0.20 and 0.30 respectively.

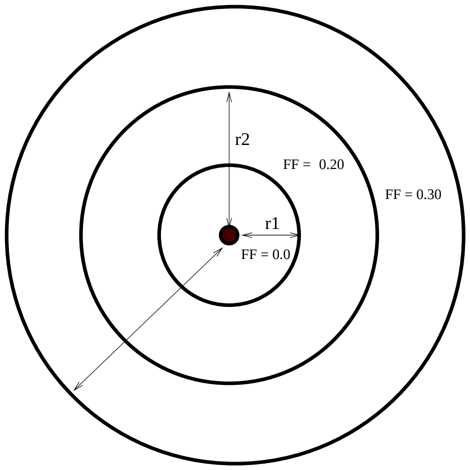

Figure 2: A client with multiple radii of visibility

Actions by clients cause *events*, typically due to motion of game objects, to occur in the game. When an event occurs the client responsible for causing the event sends an update message to the server, which in turn disseminates this update to all other clients affected by this event. Clients must receive these updates within acceptable time bounds, i.e., deadlines, after the associated event occurs, for the application of these updates to be valid. These deadlines are known as *allowable latencies*.

With the notions of ROVs and allowable latencies for clients, we now define the notion of *consistent view* for a client. A client in a MMOG is said to have a consistent view of the universe, if the game system can support its view parameters within the appropriate latency bounds, i.e., corresponding to a radii r_i in its ROV, if the server is able to maintain an update rate that satisfies the f_i parameter while simultaneously satisfying the client's allowable latency bound l. It is of utmost importance to be able to provide clients with consistent views. Virtually all work in game consistency, in one form or another, advocate the following consistency enforcing procedure, given a client-server game architecture.

- Each player (client) has her own replica of the real world, and interacts only with it
- Any change to the game world, say caused by player P, is first recorded in P's local replica and then sent to the game server
- The server forwards this update to all the other interested clients, who then update their own respective replicas

Inconsistency arises dues to the fact that there exists a delay between the occurrence of an event (say, motion of a game game object), and all players being informed of this event so that the replicas are synchronized.

3 Consistency Maintenance Mechanism

We now describe our procedure to maintain view consistency across clients in a cell. As mentioned in the previous section, all clients inside a cell are controlled by a single server. Whenever an event of interest (such as client movement) occurs, the server receives a state update from

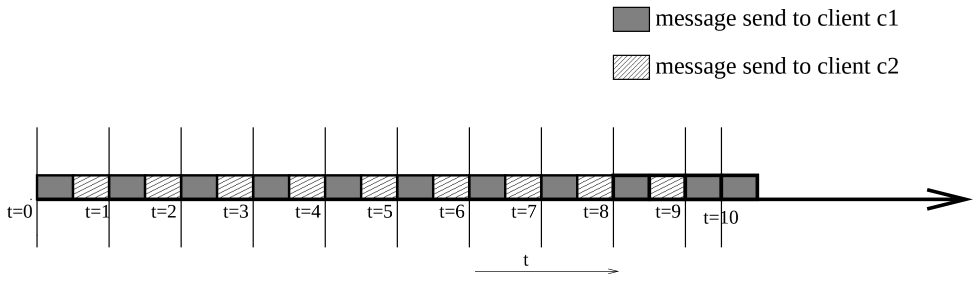

Figure 3: Using fuzziness factors to do intelligent scheduling of updates

the corresponding object. At this juncture, our consistency maintenance procedure is launched. It consists of two sequential tasks.

1. **Relevant Set Computation** As soon as the server receives an event update message, it must compute the *relevant set* of clients affected by this event, i.e., clients whose views would be compromised if this event were not communicated to it. Simply stated, an event L, is relevant to a client c_x, if L occurs within the ROV of c_x. Thus, the relevant set of an event is the set of all clients for which the event is relevant.

2. **Update Dissemination to the Relevant Set** Upon computation of the relevant set for an event, the server must disseminate the state update to each client in that set. However, since different clients might view the event with different fuzziness factors (and, possibly, might have different allowable latencies) the server must determine an update dissemination schedule that maintains consistency across the clients. This step computes that schedule.

3.1 Finding Relevant Sets

Let us assume that a client c_x has a simple single-radial ROV $\{r_x, f_x\}$ and allowable latency l_x. This implies that a fraction $(1 - f_x)$ of the events that take place in the circle (or sphere for a 3-D game), centered at the location of c_x with radius r_x are reported to c_x while also obeying the latency bound l_x. Let us consider two instants of time, t_{old} and t_{new} ($t_{new} > t_{old}$), and further assume that motion occurs for c_x across these two intervals, i.e., at t_{old}, c_x was located at (x_{old}, y_{old}) and at t_{new}, c_x is at (x_{new}, y_{new}). Such motion is an example of an event. Let us call this motion event E. we would like to find the relevant set for the event E corresponding to the location change of c_x at time t_{new}, i.e., the set of clients that need to be informed of this event. We assume for now that the client's region of visibility both at t_{old} and t_{new} does not cross cell boundaries. In this section, we outline two different methods to compute the relevant set of E.

Naive Method

Let the total number of clients in a cell be n. Let $\mathcal{C}$ denote the set of all clients that are in the cell under consideration. Figure 4 shows the pseudocode for this approach.

Time complexity of the naive method:

From the algorithm, it is clear that in order to find the relevant set for an event created by the movement of a single client, we require $O(n)$ time.

Using an R-Tree

We now present a method that uses an R-Tree [14], a spatial index structure that can be used to fasten the retrieval of data items according to their spatial coordinates. The clients in a MMOG are spatial objects that can be approximated by rectangles. We also approximate the ROV of each client by an *MBR* (Minimum Bounding Rectangle). An R-Tree organizes these rectangles in a spatial index structure such that all MBRs that are contained in another MBR are found in the subtree below it.

As in the naive method, let E denote the event caused by the movement of player p_x to a new location at time t_{new}. Our algorithm first constructs the new MBR for p_x. To find the relevant set for E, we update the R-Tree with the new MBR for p_x. We then search the R-Tree for the infinitesimally small MBR, MBR_x, corresponding to the object p_x. All MBRs in the R-Tree that lie in the path from the root of the R-Tree to the node containing MBR_x contain MBR_x. The players corresponding to these MBRs denote the relevant set of E. Figure 5 outlines this approach.

Time complexity of the R-tree based method:

For the R-tree based method, at t_0 when the game starts up, the R-Tree needs to be created based on the initial positions of all clients. Since the initial positions of all the clients is typically known beforehand, the R-Tree can be hardcoded into the code. Each time a client moves, the R-tree needs to be updated. Updating the tree requires $O(lg(n))$ time. Searching for all rectangles that contain the COV of p_x at time t_{new} requires $O(lg(n))$ time. The total asymptotic time complexity of the R-Tree based method is $O(lg(n))$. Thus, the R-Tree based method is much better than the naive method.

Algorithm
3.1: NAIVE-METHOD($p_x, \mathcal{C}$)

```
Comment: Compute Relevant set for event $E$ caused by movement of client $p$ corresponding to clients in cell $\mathcal{C}$ at $t_{new}$
Initialize $R(E) = \emptyset$
for each pair of clients $(p_x, p), where\ p \in \mathcal{C}\ and\ p <> p_x$
  do
    Compute $d$, distance between the locations of $p_x$ and $p$
    Compute the difference $\delta = r_x - d$
    if $\delta <= 0$
      then
        Comment: Client $p$ lies in the ROV of $p_x$
        $R(E) = R(E) \cup p$
      else
        Comment: Client $p$ does not lie in the ROV of $p_x$
Return $R(E)$
```

Figure 4: Pseudocode for the naive method for calculating Relevant Sets

Algorithm
3.2: R-TREE METHOD($p, \mathcal{C}$)

```
Comment: Compute Relevant set for event $E$ caused by movement of client $p$ corresponding to clients in cell $\mathcal{C}$ at $t_{new}$
Initialize $R(E) = \emptyset$
Update the R-Tree with the new MBR for $p_x$. // This MBR corresponds to the ROV of $p_x$
Search the R-Tree for the MBR, $MBR_x$, of $p_x$. // $MBR_x$ is the infinitesimally small rectangle corresponding to the object $p_x$.
Set of all MBRs that contain $MBR_x$ is $R(E)$
Return $R(E)$
```

Figure 5: Pseudocode for the R-Tree method for calculating Relevant Sets

4 Scheduling Updates

In this section, we present algorithms for efficiently disseminating updates to game clients. The problem can be restated as follows: Given a certain event and a set of clients to which the event is relevant (with different fuzziness factors), find an optimal ordering of updates such that consistency requirements are met for the maximum number of clients and the average *latency* of all these consistent clients is minimized.

We define latency for any given client as the amount of time elapsed between the reception of an event by the server and the delivery of that event to that client. We observe that the fuzziness factors can be used to schedule message transmission so that more number of clients can be kept consistent.

To elucidate this idea, consider two clients c_1 and c_2 with respective fuzziness factors 0 and 0.20. The clients' views can be kept consistent by scheduling the message sends as shown in Figure 3. Notice that at time $t = 5$, c_2 has received 100% of the packets even though a clarity of 80% (fuzziness factor is 0.20) only is required. Hence, the server now has the luxury to not send packets to c_2 when it gets overloaded. This situation occurs at time $t = 9$ and $t = 10$, and the server stops sending updates to c_2 – concentrating instead on ensuring that c_1 (with fuzziness factor of 0) receives every packet. Note that at the end (time $t = 10$), all clarity requirements have been met as c_1 received 100% and c_2 received 80% (8 of 10 packets) of the packets.

We model this problem of optimal packet dissemination as a scheduling problem with each client c_i as a job j_i. The allowable latency l_i of the client is used as the scheduling deadline. The problem then becomes finding a schedule that honours the deadlines for as many clients as possible while achieving certain criteria.

In this paper, we present two different scheduling algorithms that optimize different criteria. We assume that message transmission is atomic and this results in jobs which are non-preemptive. Also, without loss of generality, we assume that all jobs take equal amount of processing time. We present our algorithms in the rest of this section and their comparative analytical performance in Section 5. The two algorithms are:

- maximize the number of jobs that are completed by their deadlines while minimizing total average latency for the completed jobs. We call this criterion **Max-Job-Count** and present it in Section 4.1.
- maximize the *profit* obtained by completing a certain

number of jobs while minimizing the total average latency for the completed jobs. We present our notion of profit and present our profit function in Section 4.2. We call this criterion **Max-Profit** and present it in Section 4.2.

4.1 Max-Job-Count Scheduling

Consider a total of n clients for which a certain event is relevant. This results in n jobs, $j_1, j_2, j_3, \ldots, j_n$. Let $l_1, l_2, l_3, \ldots, l_n$ be the respective allowable latencies for the n clients. These are the deadlines for the n jobs. Formally, the Max-Job-Count criteria can be captured as follows: Given a schedule S, define $D_{j_i} = 1$ if the completion time of j_i in S is ${C_{j_i}}^S \leq l_i$ and $D_{j_i} = 0$ otherwise. Thus, the optimality criteria is to maximize $\sum D_j$ while minimizing total average latency.

To achieve this, first we order the jobs by their deadlines such that $j_1, j_2, \ldots, j_n$ have nondecreasing deadlines $l_1 \leq l_2 \ldots \leq l_n$. Any optimal schedule can be converted to one which has this property and has the same maximum value for $\sum_j D_j$. We denote by t_m, the time required for message transmission. According to our definition, a schedule only consists of jobs that are completed by their respective deadlines.

To find a schedule that maximizes $\sum D_j$ while minimizing total completion time, we observe that this problem has the optimal substructure property that allows a dynamic programming based solution. To exploit this, we define $T(s,t)$ as the minimum completion time of a schedule with s or more jobs from a set of t jobs $j_1, j_2, \ldots, j_t$.

Let S[1] be the schedule of jobs for which the minimum completion time is $T(s,t+1)$ i.e. $S \subseteq j_1, j_2, \ldots, j_{t+1}$. S may or may not contain j_{t+1}. If S does not contain j_{t+1}, then $S \subseteq j_1, j_2, \ldots, j_t$. S is thus a schedule of s or more jobs of a set of t jobs $j_1, j_2, \ldots, j_t$. By definition, S can be completed in $T(s,t)$ i.e.

$$T(s,t+1) = T(s,t)$$

If $j_{t+1} \in S$, j_{t+1} can be scheduled to execute as the last job (since a schedule that minimizes average latency can be obtained by sorting jobs in nondecreasing order of deadlines). The rest of the jobs form a schedule of $s-1$ or more jobs, which leads to a set of t jobs $j_1, j_2, \ldots, j_t$ that have minimum completion time $T(s-1,t)$. This gives a minimum completion time of $T(s-1,t) + t_m$ for S. It follows that:

$$T(s,t+1) = \begin{cases} min(T(s,t), T(s-1,t)+t_m) & if\ T(s-1,t)+t_m \leq l_{t+1} \\ T(s,t) & otherwise \end{cases}$$

[1]For simplicity we also consider S as a set

Theorem 1 *Space complexity and time complexity of Scheduling based on Max-Job-Count is $O(n^2)$.*

Proof The size of the schedule is bounded by the total number of jobs n. Thus, we progressively calculate values of $T(s,t)$ for $s = 0..n$ and $t = 0..n$ and store them in a table for reuse in larger subproblems. Filling each table entry requires $O(1)$ time. Thus, the space complexity and time complexity of this approach are both $O(n^2)$. With all the values of $T(s,t)$, we can find the maximum size subset of jobs by identifying the largest value of s for which some $T(s,t)$ is finite. The maximum value of s for which $T(s,n)$ is finite is the maximum number of jobs (or equivalently maximum number of clients to which messages can be sent) that can be completed by their completion date.

4.2 Max-Profit Scheduling

For this criteria, instead of maximizing the number of jobs that are completed within their deadlines, we maximize the sum of profits obtained by the completion of jobs while also minimizing the total average latency. As in Section 4.1, since the jobs are of equal duration, any schedule of jobs gives the same total average latency, and is thus trivially satisfied. We define the profit for a job based on the fuzziness factor associated with the client.

In particular, a client needs to be consistent, to events that occur, with fuzziness f and latency l. This means that at any point of time, the client should have received at least a fraction $1-f$ of the total relevant updates in that region within a latency bound l. At time t, the profit $p(t)$ is given by:

$$p(t) = \begin{cases} 1 + d(t)/e(t) - (1-f) & if\ d(t)/e(t) >= (1-f) \\ 0 & otherwise \end{cases}$$

Where:

- $e(t)$ is the total number of events that occurred in that region till time t.
- $d(t)$ the total number of events delivered to the client within the allowable latency till time t.

Note that $d(t)/e(t)$ signifies the clarity with which the particular region of visibility is seen by the client at time t. The profit associated with sending a message is the extra clarity over the required clarity that is provided to the client as a result of the update being communicated to it.

The problem of scheduling the updates can be restated as: Given a set of n jobs $j_1, j_2, \ldots, j_n$ where each job j_i has an allowable latency (deadline) l_i and profit p_i if the job finishes by deadline l_i or profit 0 if it finishes after the deadline, find a schedule of jobs such that the profit is maximized. Note that all jobs are of equal duration, t_m. Any schedule of r jobs will thus have job completion times of $t_m, 2 \times t_m, 3 \times t_m..., r \times t_m$.

We again present a dynamic programming based solution for this problem. We first order the jobs in nondecreasing order of deadlines, such that $a_1, a_2, ..., a_n$ have deadlines $l_1 \leq l_2 \leq ... \leq l_n$. Any schedule that has maximum profit can be converted to a schedule that has jobs in nondecreasing order of deadlines and the same profit. The optimal substructure property of this problem is to maximize the profit by scheduling a subset of u jobs $j_1, j_2, ..., j_u$ with a final deadline of l_u. It is known that this problem is NP-complete for non-integer values of deadlines. We assume that it is possible to scale these deadlines and job durations (t_m) to integer values so that a polynomial time solution is possible. For the purposes of this discussion we assume that l_u and t_m are both integers. We denote by $P(u,v)$, the maximum profit obtained by scheduling a subset of the set of jobs $j_1, j_2, ..., j_u$ by time v. The goal is to find the schedule that maximizes the profit $P(n, l_n)$. The recursive definition for profit is as follows:

$$P(u,v) = \begin{cases} 0 & \text{if } u = 0 \text{ } or \text{ } v = 0 \\ P(u-1, v) & \text{if } t_m > v \text{ or } l_u < v \\ max(P(u-1,v), P(u-1, v-1) + p_u) & \text{otherwise} \end{cases}$$

Theorem 2 *Scheduling under the Max-Profit criteria has a space complexity and time complexity of* $O(n \times l_n)$

Proof We build the $(n+1) \times (l_n + 1)$ profit matrix P to store the optimal value of profit for each subproblem. Also, we build a second matrix S to construct the best schedule $S(u,v)$; defined as follows:

$$S(u,v) = \begin{cases} 0 & \text{if } u \text{ or } v = 0 \\ u & \text{if } u, v > 0, \text{ and } P(u-1, j-1) + p_u \text{ is maximum} \\ u-1 & \text{if } u, v > 0, \text{ and } P(u-1, j) \text{ is maximum} \end{cases}$$

Filling each entry of the profit table $P(u,v)$ requires $O(1)$ time if all entries $P(x,y)$ with $x < u$ and $y < v$ have already been filled. Thus, computing $P(n, l_u)$ requires $O(n \times l_u)$ time. Constructing the schedule that gives maximum profit requires $O(n)$ time. Thus, the total time complexity is $O(n \times l_n)$. Since P and S are both $(n+1) \times (l_n + 1)$ matrices, the space complexity of this algorithm is $O(n \times l_n)$.

It may be noted that the **Max-Profit** method requires the server to maintain more per-client state information than the **Max-Job-Count**. In particular, the server has to maintain the instantaneous values of $e(t)$ and $d(t)$ for each client. This overhead is in the order of a few MB only even for games of large scale. We believe that this overhead is acceptable given the performance benefits (See Section 6) provided by the **Max-Profit** method.

5 Analysis of Both Schemes

In this section, we determine theoretical upper bounds on the maximum number of clients in each sphere of visibility that can be handled by the server at a given point of time for each of two criteria described in Section 4. Sections 5.1 and 5.2 provide the theoretical upper bounds for the Max-Job-Count and Max-Profit criteria respectively.

For both analysis, we let $x_1, x_2, ..., x_c$ ($x_i \geq 0$) be the maximum number of clients (for which an update is relevant) corresponding to radii of visibility $r_1, r_2, ..., r_c$ to which the update can be disseminated by the server. The server disseminates these updates to these clients within their allowable latencies $l_1, l_2, ..., l_c$ respectively where $l_1 \leq l_2 \leq ... \leq l_c$. Let the maximum bandwidth of the server be b bits/s, i.e. a server can transmit a maximum of b bits per second.

5.1 Upper bounds for Max-Job-Count

From Section 4.2, we observe that any schedule that has maximum profit can be converted to a schedule where the allowable latencies of the jobs (sending messages to clients) are in non-increasing order. Since each of the x_1 clients has an allowable latency l_1, for the schedule to be an acceptable one, the message send to all the x_i clients should be completed by l_i. The duration of each job is t_m (the time needed for the server to send a message to any of its clients).

If the average size (in bits) of any message from server to client is m and b is the server bandwidth, then:

$$t_m x_1 \leq l_1$$

where

$$t_m = m/b$$

For the schedule to be acceptable, messages sent to all x_2 clients should complete by l_2. This gives the equation:

$$t_m x_1 + t_m x_2 \leq l_2$$

and the following set of inequalities:

$$\begin{aligned} t_m x_1 &\leq l_1 \\ t_m(x_1 + x_2) &\leq l_2 \\ &. \\ &. \\ &. \\ t_m(x_1 + x_2 + ... + x_c) &\leq l_c \end{aligned}$$

Or equivalently

$$\begin{aligned} x_1 &\le l_1/t_m \\ (x_1+x_2) &\le l_2/t_m \\ &\dots \\ &\dots \\ &\dots \\ (x_1+x_2+\dots+x_c) &\le l_c/t_m \end{aligned}$$

In the matrix form, this can be represented as:

$$Ax \le b \quad (1)$$

$$where\ \ A = \begin{bmatrix} 1 & 0 & 0 & \dots & 0 & 0 \\ 1 & 1 & 0 & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 1 & 1 & 1 & \dots & 1 & 1 \end{bmatrix} \quad (2)$$

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_c \end{bmatrix} \quad (3)$$

$$b = \begin{bmatrix} l_1/t_m \\ l_2/t_m \\ \dots \\ l_c/t_m \end{bmatrix} \quad (4)$$

$$\forall i,\ \ x_i > 0 \quad (5)$$

The objective function for **Max-Job-Count** is:

$$maximize\ (x_1+x_2+\dots+x_c) \quad (6)$$

subject to the conditions of Equation (1) and Equation (5). From the constraints, it is clear that this is a linear program that can be solved using well known techniques.

Solving this linear program shows that the maximum value for $(x_1+x_2+\dots+x_c)$ is l_c/t_m. Thus, the values of $x_1, x_2, \dots, x_c$ (the maximum number of clients per ROV that can be handled by the server) for the **Max-Job-Count** criterion are given by $\lfloor (l_1 \times b/m) \rfloor, \lfloor (l_2 - l_1) \times b/m \rfloor, \lfloor (l_3 - l_2 - l_1) \times b/m \rfloor, \dots, \lfloor (l_c - l_{c-1} - \dots - l_2 - l_1) \times b/m \rfloor$ respectively.

5.2 Upper bounds for Max-Profit

We use a method similar to that in Section 5.1 to find upper bounds on the number of clients in each sphere of visibility for the Max-Profit criteria. In this criteria, the sum of the profits of the jobs needs to be maximized. By definition, it is known that the profit associated with a job is related to the additional clarity that is provided to the client over the required clarity. At any time t, without loss of generality it can be assumed that all clients in a given sphere of visibility provide the same extra clarity over the required value (this can be obtained by scheduling the "message send" jobs appropriately). The profit obtained by a server that handles $x_1, x_2, \dots, x_c$ clients corresponding to radii of visibility $r_1, r_2, \dots, r_c$ and fuzziness $f_1, f_2, \dots, f_c$ can thus be written as:

$$(1+Sg(\frac{d_1(t)}{e_1(t)} - (1-f_1)))x_1 + (1+Sg(\frac{d_2(t)}{e_2(t)} - (1-f_2)))x_2 + \dots + (1+Sg(\frac{d_c(t)}{e_c(t)} - (1-f_c)))x_c \quad (7)$$

$$Where\ \ Sg(x) = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

The objective is to **maximize** Equation (7)'s value. Since, in general, for the innermost sphere of visibility, the value of fuzziness f_1 is 0, the objective function can be rewritten as:

$$maximize\ \ [x_1 + (1+Sg(\frac{d_2(t)}{e_2(t)} - (1-f_2)))x_2 + \dots + (1+Sg(\frac{d_c(t)}{e_c(t)} - (1-f_c)))x_c] \quad (8)$$

subject to the constraints of Equations (2) and (5).

Since the values of $e(t)$, $d(t)$, fuzziness f are all known for any time t (for any setup), it is easy to compute the maximum value of the objective function using the simplex method.

6 Experimental Results

This section describes the evaluation of the algorithms presented in Sections 4.

6.1 Experimental Setup

We performed the evaluation of the algorithms using the TOSSIM [19, 27] simulator. TOSSIM was originally designed to test the TinyOS [16] sensor network operating system and we used it as a) it can accurately simulate a large number of small nodes that are exchanging information (which is highly representative of game environments), and b) it is a high-fidelity accurate simulator that is used by many other researchers. Hence, we are confident the the results obtained using TOSSIM will show the correct patterns and trends.

A key input to our simulation was the packet send rate between the clients and server as well as the size of packets exchanged between the clients and server. To obtain realistic values, we analyzed the Quake 2 [17] game and used the values from that game. We used Quake 2 as a) the source is available, b) it is a financially successful commercial multiplayer game, and c) it is representative of a large class of multiplayer action games.

An UPDATE message sent from a client to the server consists of a dead reckoning vector. The vector contains information about the current position of the client in 3-D and the trajectory of the entity in terms of the velocity component in three dimensions. The sending client [2] also identifies itself to the server by tagging the UPDATE message with its identity. The server node sends an UPDATE message to the client with the received dead reckoning vectors.

6.2 Metrics

To analyze our algorithms, we define the following metric:

Fidelity: We define fidelity as the fraction of time for which the player's awareness of the game world correctly reflected the actual game state. For example, a fidelity value of 1.0 indicates that the player was never inconsistent with the game while a value of 0.5 indicates that the player had an inconsistent view of the game half the time. The fidelity metric can be mathematically represented as follows:

$$Fidelity,\ f = 1 - \frac{i}{t} \qquad (9)$$

where:

- i, the infidel time, is the amount of time for which the player's view of the game is not same as the canonical game state.
- t is the total amount of elapsed game time.

Average Fidelity is the average of fidelity across all clients in the game space.

6.3 Experiments Performed

We performed two types of experiments to evaluate our algorithms:

- **Scheme Comparison**: In this experiment, we compared the performance of the schemes in Sections 4.1 and 4.2. In particular, we compared the percentage of total clients that could be kept consistent with each scheme for different sizes of the region of visibility and different client to server event rates. Note that the server to client event rates are governed by the radii of visibility and the client to server event rates. These results are presented in Section 6.4.
- **Metric Evaluation**: In this experiment, we compared the fidelity achieved by each of our schemes for different numbers of clients. These results are presented in Section 6.5.

[2]We use the terms client and nodes interchangeably

6.4 Results: Scheme Comparison

In this set of experiments we are interested in discovering how well our procedures maintain client consistency. We compare three different schemes:

1. **Max-Job-Count**: In this procedure the server computes Relevant Sets (RS) of events using the R-Tree based procedure shown in Figure 5 and then uses the Max-Job-Count scheduling algorithm to schedule the updates to the members of the RS.
2. **Max-Profit**: In this procedure the server computes Relevant Sets (RS) of events using the R-Tree based procedure and then uses the Max-Profit scheduling algorithm to schedule the updates to the members of the RS
3. **EDF**: In this procedure the server computes Relevant Sets (RS) of events using the R-Tree based procedure and then uses the Earliest-Deadline-First (EDF) scheduling algorithm schedule the updates to the members of the RS

Since the EDF algorithm is known to perform well under a large variety of cases, we feel it provides a reasonable baseline to benchmark our algorithms. In the EDF scheduling method, we schedule all those "message sends" which have the least allowable latency ahead of those with higher allowable latency. If two "message send" jobs have the same allowable latency, these jobs are scheduled in arbitrarily random order. We use a setup of 500 nodes with one of them acting as the server node. At regular intervals (called *epoch*), each node generates three random numbers from a Gaussian distribution with mean 0 and variance 1. The right most 16 bits of these random numbers are used as the new values of the X, Y and Z coordinates of the location of the node (i.e. the location to which the node moves). This movement corresponds to an event in the game space and thus governs the event rate. The epoch size is set such that different values of event rates, ranging from 10 events per sec to 80 events per second were generated. Three different values of radii of visibility, $r_1 = 5$, $r_2 = 15$ and $r_3 = 25$, were used. The fuzziness factors for r_1, r_2 and r_3 are 0.0, 0.20 and 0.30 respectively and the corresponding allowable latencies are 100 *ms*, 200 *ms* and 300 *ms*. Each update message had a size of 30 bytes. With the implementation of TOSSIM that we used, each message transmission takes approximately 1 msec. In our equations this corresponds to the value of the t_m parameter.

6.4.1 Summary of Results

We computed the percentage of clients that are provided with a consistent view of the game space in each of the three spheres of visibility defined by r_1, r_2, r_3. This is done by computing the percentage of consistent clients at the end of every epoch and computing the average of percentages recorded over the duration of the game. The results of this experiment are shown in Figure 6. From the figure it is seen that as the event rates increase, the percentage of consistent clients decreases in all the three schemes. This can be explained as follows: As the number of clients increases the number of update messages received by the server increases and given the latency bounds of the clients, scheduling all message sends within a given amount of time becomes increasingly infeasible. The Max-Profit scheme provides 100% consistency for update rates of up to 50 events per second. It is worth noting that popular games such as Quake 2, scale up about 30 events per second. The fact that our algorithms demonstrate 100% consistency upto event rates of 50/sec speaks to their increased scalability properties. Indeed, if 90% consistency can be tolerated in a game (perfectly acceptable by current standards), then Max-Profit provides acceptable performance up to 70 events/sec, a more than 2X improvement in allowable event rates. Given that event rates typically scale much worse than number of clients, this translates to much more attractive allowable participation counts.

The two comparative findings of note from this experiment are (a) EDF performs worse than both our procedures, and (b) Max-Profit performs appreciably better than Max-Job-Count. The first phenomenon is easy to explain – EDF acts as a greedy heuristic, arranging the dissemination schedule purely based on deadline, while the other two attempt to factor in semantic knowledge available regarding the application (i.e., the game). The second phenomenon is more subtle and leads to the discovery of an interesting property of this class of system. Effectively, Max-Job-Count attempts to maximize number of jobs completed within their deadlines, discriminating among jobs based on their completion time. Max-Profit on the other hand, discriminates among the jobs based on their ”usefulness”. It does so by considering fuzziness – if sending an update will not improve the clarity of a particular visibility slice (because its clarity parameters are already satisfied), Max-Profit will schedule this after (indeed, this update may even get dropped) more ”important” updates that must be executed to maintain view parameters. The interesting finding is that at high event loads, such discrimination pays off.

Upper Bound Comparison: We also ran experiments to test how our theoretical upper bounds analysis reported earlier in the paper stacks up against real upper bounds discovered in our tests. We first calculated the theoretical maximum number of clients that can be kept consistent using the analysis reported in section 5. Plugging in the parameters used in these tests into our theoretical expressions, we get the following: 100 clients in the innermost sphere with radius r_1, 100 clients in the sphere with radius r_2 and 100 clients in the sphere with radius r_3 can be kept consistent. Armed with these numbers, we ran experiments with 100 nodes in each of the three slices of visibility. If our theoretical analysis is valid we would expect to observe virtually 100% consistency in these experiments. Indeed we observe, for Max-Job-Count, number of consistent clients as 90, 85 and 95 for the innermost to the outermost visibility slices. However, for Max-Profit we observe 100% consistency across all three slices. This prompts us to conclude that our theoretical analysis is indeed valid.

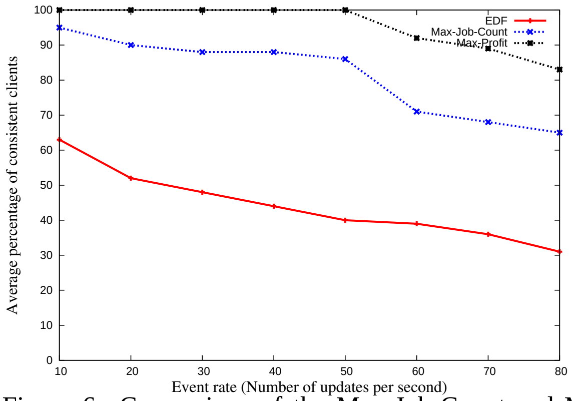

Figure 6: Comparison of the Max-Job-Count and Max-Profit scheduling algorithms with EDF

6.5 Results: Metric Evaluation

We computed the fidelity that is given by the two schemes by varying the number of clients. Experiments were run with 250, 500, 600, 800 and 100 clients each with an event rate of 30 updates per second. The radii of visibility, fuzziness factors and allowable latencies were set to the same values as in Section 6.4. Figure 7 shows the fidelity comparison for the two schemes. The experiments were run for 3600 seconds, a sufficiently long time for the values of fidelity to be statistically significant.

6.5.1 Summary of Results

From the figure it is clear that as the number of clients increases fidelity delivered by the three schemes decreases. This is due to the increased number of messages in the system with increasing number of clients. The **Max-Profit** scheme makes use of fuzziness factors to do intelligent scheduling and thus produces higher fidelities.

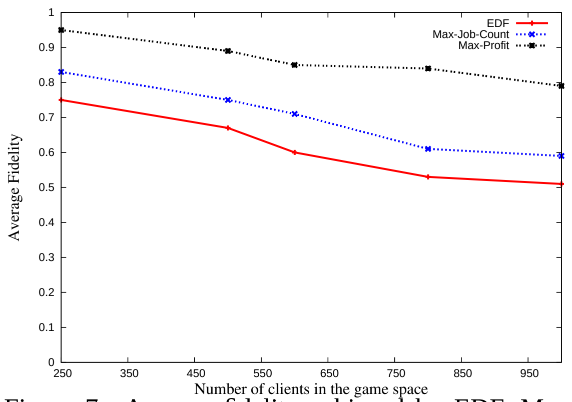

Figure 7: Average fidelity achieved by EDF, Max-Job-Count and Max-Profit scheduling algorithms

6.6 Results: Relevant Set Computation

We presented our R-Tree based method for computing relevant sets in Section 3. This method is both optimal and scalable. **Optimal**: Our R-Tree method returns the exact list of relevant clients. Hence, we never omit relevant clients or send send updates to irrelevant clients. From our experiments, we noticed that the average relevant set size depended heavily on the ROV used and contained about 10% to 50% of the total client population. To understand the effect of non-optimal relevant sets, we ran experiments where different relevant sets were used. As the number of unnecessary clients in the relevant set increases, the overall performance decreases. This is because sending to more clients would mean allowable latencies for not all of them are honored. This leads to decrease in the percentage of consistent clients as the number of non-relevant nodes increase. Hence, optimality is required to attain the best possible results.

Scalable: Our R-Tree method requires $O(lg(n))$, where n is the number of clients, running time. Hence, it requires very little extra computational time as the number of clients increases. More importantly, its running time does not increase as the *client density* changes. i.e., it is equally fast regardless of whether 10% or 80% of the clients are relevant. This scalability property is a crucial result as client movement patterns and dynamic ROV changes result in highly dynamic client relevant sets.

These two properties of our R-Tree based relevant set computation method make us confident that it can be used in real MMOGs where a) it is important that every player that needs to know of an event receives it, b) the updates of events must be sent in real time, and c) the number of players and player density can change by orders of magnitude in just a few minutes.

6.7 Discussion of Results

From our experiments, we conclude that the scheduling mechanism that maximizes profit provides consistent view of the game space to a higher number of clients than the mechanism based on maximizing job count. It is worth noting that the Max-Job-Count scheme that proactively tries to keep higher number of clients consistent is worse in comparison to the Max-Profit scheme, which only tries to maximize profit without worrying about the number of clients. We attribute this, as already discussed, to the use of fuzziness factors to discriminate among jobs. The interesting property that emerges is as follows: given a set of jobs with deadlines and a set of clients that consume these jobs, if the clients can survive without a particular job being delivered to it, it is always preferable to deprioritize that job below “clients cannot survive without” jobs, even at the cost of not maximizing the number of scheduled jobs on any specific schedule. This may result in low priority job being dropped from the schedule, leading to a schedule which may not always maximize the number of completed jobs.

From the above discussion it is clear that the Max-Profit scheme gives the best performance. In our experiments we monitored the CPU usage of each of our schemes. Results are shown in Figure 8. From the figure, we see that the high performance of Max-Profit comes with only a small amount of extra CPU usage, thus justifying our conclusion that Max-Profit is better. Also, our relevant set computation method is both optimal and scalable. Hence, we conclude that our algorithms are perfectly suited for real MMOG environments.

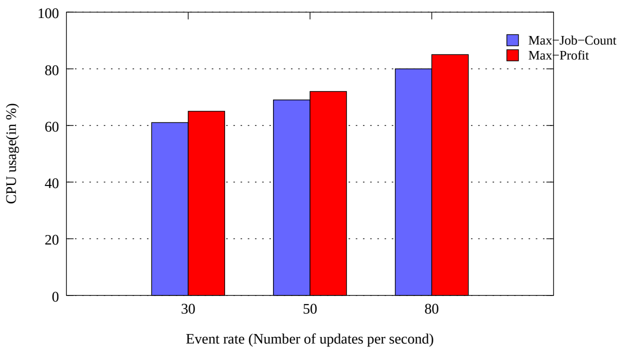

Figure 8: Comparison of CPU usage for Max-Job-Count and Max-Profit scheduling algorithms

7 Conclusions and Future Work

In this paper, we described various algorithms for achieving consistent game performance on a single server. We addressed both the update dissemination requirements as well as the consistency set determination requirements of consistency protocols. Our model is able to scale to realistic scenarios where different clients could have different consistency requirements. We showed, via simulations, that our algorithms perform well under realistic game conditions.

Our work is an important first step in addressing the difficult yet increasingly important problem of achieving fast yet consistent massively multiplayer games. We plan to extend this work in many different dimensions. In particular, we plan to build consistency protocols that are provably correct and efficient when more than one server is being used and clients are able to migrate and interact across the servers. We also plan to address the consistency requirements for truly massively multiplayer games that have millions of players, thousands of servers, gigabytes of traffic, and highly diverse consistency requirements. For example, most events will only require local consistency but some events, such as events affecting the game's internal economy, will have to be globally consistent. Finally, we plan to implement and test our algorithms in real multiplayer games.

References

[1] Adya, A. and Liskov, B. Lazy consistency using loosely synchronized clocks. *Proc. 16th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, Santa Barbara, CA, Aug. 1997.

[2] Agarwal, A., Chaiken, D., Johnson, K., Kranz, D., Kubiatowicz, J., Kurihara, K., Lim, B.-H., Maa, G., and Nussbaum, D. The MIT alewife machine : A large-scale distributed-memory multiprocessor. *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*, Toronto, CA, May 1991. Kluwer Academic.

[3] Aggarwal, S., Banavar, H., Khandelwal, A., Mukherjee, S., and Rangarajan, S. Accuracy in dead-reckoning based distributed multiplayer games. *Proc. 3rd ACM SIGCOMM workshop on Network and system support for games (NetGames)*, Aug. 2004.

[4] Amza, C., Cox, A. L., Dwarkadas, S., Keleher, P., Lu, H., Rajamony, R., Yu, W., and Zwaenepoel, W. Treadmarks: shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, Feb. 1996.

[5] Armitage, G. Lag over 150 milliseconds is unacceptable. `http://gja.space4me.com/things/quake3-latency-051701.html`, May 2001.

[6] Balan, R. K., Ebling, M., Castro, P., and Misra, A. Matrix: Adaptive middleware for distributed multiplayer games. *Proceedings of the 6th ACM/IFIP/USENIX International Middleware Conference (Middleware)*, Nov. 2005. Short Paper.

[7] Bernstein, P. A. and Goodman, N. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185, June 1981.

[8] Blizzard Entertainment. *World of Warcraft*, Sept. 2004. `http://www.worldofwarcraft.com`.

[9] Breitbart, Y., Komondoor, R., Rastogi, R., Seshadri, S., and Silberschatz, A. Update propagation protocols for replicated databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 28(2):97–108, 1999.

[10] Butterfly.net. *The Butterfly Grid*, Sept. 2000.

[11] Chan, A., Lau, R. W. H., and Li, L. Motion prediction in gesture-based collaborative design environments. *Proc. IEEE Virtual Reality Conference (VR)*, Alexandria, VA, Mar. 2006.

[12] Gautier, L. and Diot, C. Design and evaluation of mimaze, a multi-player game on the internet. *Proc. IEEE International Conference on Multimedia Computing and Systems (ICMS)*, Austin, TX, June 1998.

[13] Gerald-Yamasaki, M. J. Cooperative visualization of computational fluid dynamics. *Proc. 1st international conference on Parallel and distributed information systems (PDIS)*, Dec. 1991.

[14] Guttman, A. R-trees: A dynamic index structure for spatial searching. *Proc. ACM SIGMOD Conf.*, Boston, MA, June 1984.

[15] Haritsa, J. R., Ramamritham, K., and Gupta, R. Real-time commit processing. *Real-Time Database Systems*, pages 227–243, 2001.

[16] Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., and Pister, K. System architecture directions for network sensors. *ASPLOS*, November 2000.

[17] Id Software. *Quake 2 Source Code*, Apr. 2002. `http://www.idsoftware.com/business/techdownloads/` (V3.21).

[18] Lenoski, D., Laudon, J., Joe, T., Nakahira, D., Stevens, L., Gupta, A., and Hennessy, J. The DASH prototype: Implementation and performance. *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 92–103, Gold Coast, Australia, May 1992.

[19] Levis, P., Lee, N., Welsh, M., and Culler, D. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. *First ACM Conference on Embedded Networked Sensor Systems(SenSys)*, 2003.

[20] Li, F., Li, L., and Lau, R. Supporting continuous consistency in multiplayer online games. *Proc. ACM Multimedia*, New York City, NY, Sept. 2004.

[21] Li, K. and Hudak, P. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, Nov. 1989.

[22] Li, L. W. F., Li, F. W. B., and Lau, R. W. H. A trajectory-preserving synchronization method for collaborative visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):989–996, 2006.

[23] Mauve, M. Consistency in replicated continuous interactive media. *Proc. ACM conference on Computer supported cooperative work (CSCW)*, Dec. 2000.

[24] Mauve, M., Vogel, J., Hilt, V., and Effelsberg, W. Local-lag and timewarp: providing consistency for replicated continuous applications. *IEEE Transactions on Multimedia*, 6(1):47–57, 2004.

[25] Square Enix. *Final Fantasy XI Online*, Oct. 2003.

[26] Sun, C. and Chen, D. Consistency maintenance in real-time collaborative graphics editing systems. *ACM Trans. Comput.-Hum. Interact.*, 9(1):1–41, 2002.

[27] TinyOS Website. `http://www.tinyos.net`.

[28] Valve Corporation. *Half Life 2*, Nov. 2004.

[29] Wood, J., Wright, H., and Brodlie, K. Collaborative visualization. *Proc. 8th conference on Visualization (VIS)*, Oct. 1997.

[30] Xiong, M., Ramamritham, K., Haritsa, J., and Stankovic, J. A. Mirror: A state-conscious concurrency control protocol for replicated real-time databases. *rtas*, 00:100, 1999.

[31] Zekauskas, M. J., Sawdon, W. A., and Bershad, B. N. Software write detection for distributed shared memory. *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*, Monterey, CA, Nov. 1994.