Peer-to-Peer Support for Massively Multiplayer Games

Björn Knutsson (bjornk@dsl.cis.upenn.edu)
Honghui Lu (hhl@cis.upenn.edu)
Wei Xu (xuw@seas.upenn.edu)
Bryan Hopkins (bwh@stwing.upenn.edu)
Department of Computer and Information Science, University of Pennsylvania

Abstract— **We present an approach to support massively multi-player games on peer-to-peer overlays. Our approach exploits the fact that players in MMGs display locality of interest, and therefore can form self-organizing groups based on their locations in the virtual world. To this end, we have designed scalable mechanisms to distribute the game state to the participating players and to maintain consistency in the face of node failures. The resulting system dynamically scales with the number of online players. It is more flexible and has a lower deployment cost than centralized games servers. We have implemented a simple game we call SimMud, and experimented with up to 4000 players to demonstrate the applicability of this approach.**

I. Introduction

We propose the use of peer-to-peer (P2P) overlays to support massively multi-player games (MMGs) on the Internet. Players participating in the game form an overlay on which many of the game functions are implemented. The players thus contribute the memory, CPU cycles and bandwidth to manage the shared game state.

The premise of most MMGs is that of a large shared game world inhabited by thousands of players. The emphasis is often on social interactions and exciting story lines. Games like Lineage have recorded two million registered players, and 180K concurrent players in one night.

Online MMGs are traditionally supported by a client-server architecture, where the server keeps both player account information and handles game state. Scalability is achieved by employing server clusters. The servers can either be connected by LANs, as in Terazona [38], or they can form a computing grid, as in Butterfly.net [7]. Although this architecture scales with the number of players, it lacks flexibility and the server has to be over-provisioned to handle peak loads. Furthermore, the client-server model limits the deployment of user-designed games, which is an important trend in game design. While games like EverQuest allow limited user designed game extensions, security and performance concerns will limit the scope of such extensions since they would need to be hosted on the game servers handling the core game.

Massively multiplayer online games are natural applications for peer-to-peer overlays. We take advantage of the self-organizing characteristic of P2P overlays to create a system that dynamically scales up and down with the number of players. Game players also have incentives to join the overlay, because the participation is limited to the duration of the player's game play.

Games are different from previous P2P applications that focus on the harnessing of idle storage and network bandwidth, including storage systems [12], [33], [11], content distribution [9], [20] and instant messaging [26]. Games utilize the memory and CPU cycles of peers to maintain the shared game state. Three potential problems must be addressed to make this approach fully applicable in practice:

- **Performance**—games have frequent updates, that must be propagated under certain time constraints. Furthermore, peers have limited bandwidth since they are located at the edge of the network.
- **Availability**—replicating game states to improve availability has two potential problems. First, once a peer goes offline, its state quickly becomes stale and the replica becomes invalid. Secondly, because of high update frequency, maintaining a large set of replicas is a potential performance bottleneck.
- **Security**—both the prevention of account thefts and the prevention of cheating during game play should be considered. Distributing game states to the peers increases the opportunities for cheating.

This paper discusses the first two problems in detail. Our design prevents account thefts, the problem most important to the game industry, by centralizing the account management at the server, and only distributing game states to the peers. Although cheat-prevention is a major concern for online games [4], it is a separate issue from the basic performance and availability of P2P gaming. We will make note of instances where cheat-prevention has influenced our design, but the details and particulars are a subject of ongoing and future work.

The primary technical contributions of this work are architectural and evaluative. We preset a novel architecture marrying massively multiplayer games with peer-to-peer networking technologies, and we provide a detailed performance study to demonstrate the feasibility of our design.

The key to the feasibility of a P2P game architecture is locality of interest [27]. Games are designed such that while the game world is large, the area of interest to a single player is limited, typically correlating to the sensory capabilities of the game characters being modeled. The players, in turn, can be arranged into groups with coinciding areas of interest. This self-organizing property of MMGs matches the self-organizing character of peer-to-peer networks. In particular, nearby (as defined in game terms) players can form peer groups, and keep updates to game state within the group.

The rest of this paper is organized as follows: Sections II and III provide background material on online MMGs and peer-to-peer overlays, respectively. Section IV discusses our design choices. Section V describes the algorithm for keeping game state consistent. Section VI details the implementation of our system. Section VII presents the experimental results. Section VIII discusses related work. We conclude and discuss future work in Section IX.

II. Online Massively Multiplayer Games

Massively multiplayer online games (MMGs) distinguish themselves from other online games by allowing thousands of players to share a single game world. Most existing MMGs are role-playing games (RPG) or real-time strategy (RTS)/RPG hybrids. Typical examples of MMGs include EverQuest, Ultima Online, There.com, Star Wars Galaxies, The Sims Online and Simcountry. Although First Person Shooter (FPS) games like Quake and Doom also have large numbers of concurrent players, they are usually divided into many small isolated game sessions with a handful of players each, and the same is true for many networked RTS games like Warcraft III.

The basic premise in most MMGs is that the player assumes the role of a character in a virtual world. The classical setting for MMGs is Middle Earth-like, with characters belonging to different races like dwarves, humans, elves and classes like magicians, fighters and priests, but could be an arbitrary world — past, present or future. The player experiences the game world through a game *avatar* — the representation of his character in the game — and is typically limited to seeing, hearing and doing things through his avatar.

The typical game involves the game character taking on missions or quests, alone or as part of a group, that require him to travel to different parts of the game world, interacting with various players, and finding objects or earning money, while accumulating experience (often abstracted into abilities and experience points).

Superficial differences aside, much of the underlying game mechanics, data structures and communication patterns are similar. We review the common characteristics of MMGs and existing network implementations in the rest of this section.

A. Game states

A typical multiplayer game *world* is made up of immutable landscape information (the *terrain*), characters controlled by players (*PCs*), *mutable objects* such as food, tools, weapons, mutable landscape information (e.g., breakable windows), and non-player characters (*NPCs*) that are controlled by automated algorithms. NPCs can be either allies, bystanders or enemies, and are not always immediately distinguishable from PCs, except by their interaction.

The terrain consists of all immutable elements in the game. Graphic elements for the terrain are typically installed as part of the game client software, and updated using the normal software update mechanisms. The abstract description of the terrain of a region can be created offline and inserted into the system dynamically.

The state of a player includes his position in the world and the state of his game *avatar*, such as its abilities, health and possessions. Avatar states are often persistent and can be carried along from one login session to another. Similar states exists for NPCs and game objects, and depending on their role in the game, they may either be persistent or temporary.

In general, a player is allowed three kinds of actions: *position change*, *player-object interaction*, and *player-player interaction*. Players interacting with objects (including NPCs) or other players may, subject to game rules, change their state as well as the state of their avatar. For example, drinking from a bottle would change the state of the bottle object from full to empty, and decrease the thirst parameter of the player object. Similarly, if the player fights another player, both player objects' health parameters would change.

Except for persistent player states and the terrain, most other game states are periodically rebuilt. This is because as the game progresses, all NPC opponents will eventually be killed, all food be eaten and all quests solved. The rebuild is either implemented as a game-wide "reset", or as a periodic "respawn" of individual NPCs and objects.

The resulting world is huge, and is typically statically divided into *regions* connected with each other, possibly taking the player from one game server to another. These connections can be implemented using game mechanics, e.g. a tunnel. Each region can be further subdivided to keep the amount of data that the client handles small enough to fit in memory.

B. Existing system support

The client-server architecture is the predominant paradigm for implementing online MMGs. In this model, players connect to a centralized server using their client software. The clients can be anything from text terminals to advanced 3D rendering systems that allows the player to see the world in which he is playing. The server is typically responsible for both maintaining and disseminating game state to the players, as well as account management and player authentication [38], [7].

The main reason for hosting game state on a centralized server is to allow the players to share the same virtual world. Scalability is approached in a number of different ways; large dedicated servers are used to allow a single server to handle thousands of simultaneous players. Further scalability is achieved by clustering servers, and by dividing the game universe into multiple different, or parallel, worlds and spreading the users over them. A typical single machine server can support 2000 to 6000 concurrent clients, and the cluster solution allows TeraZona [38] to support up to 32,000 players.

Some multiplayer games, such as MiMaze [14] and Age of Empires [28], are implemented using a decentralized model. These designs have severe limitations on scalability because the game states are broadcast to all players. AMaze [5] and Mercury [6] are based on group communications, where nearby players in the game world form a multicast group. These multicast-based games are first person shooter games. They are different from MMGs in that there are no mutable objects such as food and drinks. The lack of shared game state allows a simple distributed implementation that only sends the positions of players and objects.

C. The effect of latency on player performance

Player tolerance for network latency (a.k.a lag) varies from game to game. In general, games where a player is guiding an avatar rather than directly controlling the game action can better tolerate latency. It is therefore unsurprising that this is how most MMGs are designed.

First person shooter games (such as Quake 3) where the player directly controls the avatar can only tolerate latencies less than 180 milliseconds [3]. Real time strategy games like Warcraft III, on the other hand, can tolerate up to several seconds of network latency [35], because the emphasis is on strategy rather than direct interaction. Although there is no study of latency tolerance for MMGs, role playing games are generally considered to have a latency tolerance similar to RTS games, since the player controls the game by telling his avatar *what* to do, e.g. "pick up object" or "attack monster", rather than *how* to do it.

III. Peer-to-Peer Infrastructure

A number of peer-to-peer overlays have recently been proposed, including CAN [29], Chord [36], Tapestry [37] and Pastry [32]. These self-organizing, decentralized systems provide the functionality of a scalable distributed hash table, by reliably mapping a given object key to a unique live node in the network. The systems balance object hosting and query load, transparently reconfigure after node failures, and provide efficient routing of queries [31].

We built our application on top of Pastry [32], a widely used P2P overlay, and we use Scribe [10], the multicast infrastructure built on top of Pastry, to disseminate game state.

A. Pastry

Pastry maps both the participating nodes and the application objects to random, uniformly distributed IDs from a circular 128-bit name space, and implements a distributed hash table to support object insertion and lookup.

Objects are mapped on the live nodes whose ID is numerically closest to the object ID. For example, if we have four nodes with IDs 1, 3, 7 and 10, then message 4 will be routed to node 3, and message 8 to node 7. Each node also has "pointers" to its closest neighbors on both sides. ("Closeness" in this context is limited to the numerical ID, no geographical or topological closeness is implied.)

Pastry routes a message toward its destination within an expected $log_{2^b}N$ routing steps (where b is a configuration parameter). For example, in a network of 10,000 nodes with $b = 4$, an average message would route through three intermediate nodes. Each Pastry node maintains a leaf set, consisting of l nodes whose IDs are numerically closest to and centered around the local node ID. The leaf set ensures reliable message delivery even when multiple failures happens. Despite the possibility of concurrent failures, eventual message delivery is guaranteed unless $l/2$ nodes with adjacent node IDs in a leaf set fail simultaneously, where l has a typical value of $8 * log_{2^b}N$. Node additions and fail-stop node failures are handled efficiently, and Pastry routing invariants are quickly restored.

B. Scribe

Scribe [10] is a scalable application level multicast infrastructure built on top of Pastry. Multicast groups are mapped to the same 128-bit ring of identifiers. A multicast tree associated with the group is formed by the union of the Pastry routes from each group member to the group ID's root, which also serves as the root of the multicast tree. Messages are multicast from the root to the members using reverse path forwarding.

Group member management in Scribe is decentralized and highly efficient because it leverages the existing Pastry overlay. Adding a member to a group merely involves routing toward the group ID until the message reaches a member of the tree, followed by adding the route traversed by the message to the group multicast tree. As a result, Scribe can efficiently support large numbers of groups, arbitrary numbers of group members, and groups with highly dynamic membership.

IV. General Distributed Game Design

This section discusses how to partition the world and how to synchronize game states in a general decentralized system with best-effort multicast capabilities. In the next section we discuss how to implement the general design on a P2P overlay and deal with features of P2P networks such as addressing and replication.

Our idea is to distribute the transient game state of the MMGs on a peer-to-peer network, while persistent user state (payment information and character experience) is handled by a central server. The important property of our system is that it allows a centralized server to delegate the bandwidth and processing intensive game state management to the peer-to-peer network formed by clients participating in the game, while retaining control over the less frequently updated persistent game state.

The system could also be used without a server for ad hoc game sessions when hundreds or thousands of players gather together to play a game for a few hours, and where all game data is transient.

A. Partition of the game world

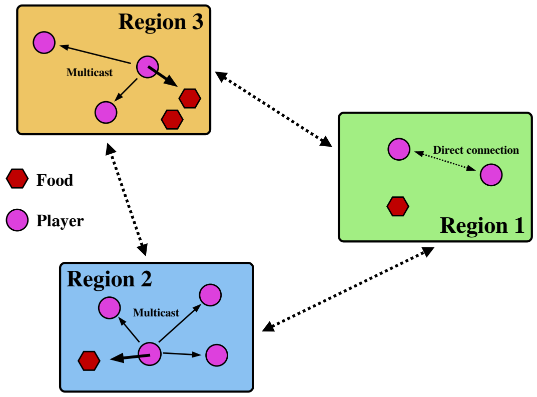

Fig. 1. Game Design

Our design is based on the fact that players in games have limited movement speed and sensing capabilities, thus the data access in games exhibits both temporal and spatial localities. Networked games and distributed real-time simulations have exploited this property and applied *interest management* [27] to game state. Interest management allows us to limit the amount of state any given player has access to, so that we can both distribute the game world at a fine granularity and localize the communication.

We partition the world into regions based on the limited sensing capabilities of a player's avatar. Players in the same region form an interest group for that portion of the map, so that state updates relevant to that part are disseminated only within the group. A player changes group when he moves from one region to another, as illustrated in the design overview in Figure 1. He could also be allowed to listen to updates in other, e.g. adjacent, regions if motivated or required by game mechanics.

Additionally, objects residing in a given region only need to communicate the part of their state that is *visible* to players. For example, a chest in a dungeon must communicate its location and appearance to players, but not its status as locked or unlocked, or its content. This also helps preventing cheating by players snooping on traffic or in-client memory.

Interest management details are highly application dependent [27]. To keep our prototype simple, we use fixed size regions and limit a player to listening to one region at a time. We vary the group size and the frequency of group changes to emulate the network effect of various interest management approaches.

B. Game state consistency

The game state must be consistent among the players withing the same region. If, for example, player A drank half of a bottle of wine, player B who later arrives on the scene will only be able to drink the remaining half.

This section only considers consistency under failure-free environments. Fault tolerance will be discussed in the next section, because it is closely related to features of the P2P overlay.

Our basic approach employs coordinators to resolve update conflicts. Since different game states have different access patterns and consistency requirements, we split game state management into the classes presented below.

1) Player state: Player state is accessed in a single-writer multiple-reader pattern. Each player updates his own location as he moves around. Player-player interactions, such as fighting and trading, only affect the states, e.g., life points, of the players involved.

Because position change is the most common event in a game, the position of each player is multicast at a fixed interval to all other players in the same region. The interval is determined during game design, based on the requirements of the game. We use best effort multicast to disseminate position updates. Although additional reliability can be added by implementing receiver-reliable multicast [19], it is usually unnecessary. The loss and delay of messages can be masked by application level mechanisms such as dead reckoning [5], that interpolates or extrapolates player positions. An alternative to periodic updates is to multicast the position only if it has changed or is significantly different from what dead reckoning would predict. This approach potentially reduces the network traffic, but incurs additional overhead for detecting lost or delayed messages.

Players usually have to be in close proximity in the virtual world in order to interact. Player-player interaction often involve multiple actions in quick succession, e.g. in a heated battle, and often require fast responses [28]. The increased communication requirements are, however, limited to the involved players and, possibly, the players in the immediate vicinity (i.e. to allow them see a fight).

2) Object State: We use a coordinator-based mechanism to keep shared objects consistent. Each object is assigned a coordinator, to which all updates are sent (the distribution and replication of coordinators are discussed in the next section). The coordinator both resolves conflicting updates, and is a repository for the current value of the object.

Successful updates are multicast to the region to keep each player's local copy fresh. We use best-effort multicast, and provide functions to allow a client to probe the value to verify the current value. Timely delivery of object state is necessary, but like position updates, missing information can be corrected with subsequent messages.

3) The Map: Graphic elements for the terrain and players are typically installed as part of the game client software, and can be updated using the normal software update mechanisms. A map is a non-graphical, abstract description of the terrain of a region. Maps are considered read-only because they remain unchanged during the game play. They can be created offline and inserted into the system dynamically. Dynamic map elements are handled as objects.

V. Distributed Game On A P2P Overlay

This section discusses how to map distributed game states to an P2P overlay and how to replicate game states to improve availability. We base our discussions on Pastry [32] and Scribe [10], the two P2P infrastructures on which we implemented SimMud. Our algorithm can, however, readily be extended to other hashing functions, and it can even be simplified for a more deterministic routing algorithm like Chord [36].

A. Mapping games states on to peers

We group players and objects by regions, and distribute the game regions onto different peers by mapping them to the Pastry key space. Each region is assigned an ID, computed by hashing the region's textual name using a collision resistant hash function (e.g. SHA-1). A live node whose ID is the closest to the region ID serves as the coordinator for the region. In our current design, the coordinator not only coordinates all shared objects in the region, but also serves as the root of the multicast tree, as well as the distribution server for the region map. Although mapping all synchronization responsibilities to the same node simplifies the design, it might incur a high network load on the coordinator. However, the load can be distributed by creating a different ID for each type of object in the region, thus mapping them on to different peers.

Because of the random mapping, the coordinator of a region is unlikely to be a member of the region, but the lack of locality actually works to our benefit for a number of reasons. First, it reduces the opportunities for cheating by separating the shared objects from the players that access them. Second, instead of handing off the coordinator when the corresponding player leave the region, the random mapping limits coordinator hand-offs to when a player either joins or leaves the game. Finally, random mapping improves robustness by reducing the impact of localized (game and real world) events. For example, multiple disconnects in the same region do not typically result in losing the region's state.

B. The fault-tolerance problem

There is one major obstacle that all P2P-systems must overcome: Participating machines can be expected to disconnect (or crash) in a much less controllable fashion than pampered servers in a corporate data center. We must therefore make fault-tolerance and efficient failover a priority.

Pastry and Scribe provide limited fault-tolerance in that their routing is resilient to network and node failures, but game states still need to be replicated to improve their availability. Furthermore, the replicas must be kept consistent upon node and network failures. Since an efficient general solution to this problem is impossible to construct, we make some assumptions based on our target application and the expected configuration of a sufficiently large P2P network.

1) Node failures are independent: The node ID assignment in P2P networks is quasi-random, and ensures that at large scales, there is no correlation between the node ID and the node's geographic or network topological location or ownership. It follows that a set of nodes with adjacent node IDs are highly likely to have independent failures.

2) The failure frequency is relatively low: We expect players to be online for extended periods of time and have incentives not to disconnect except when they exit (gracefully) from the game. This allows us to use lazy node failure discovery, i.e. use existing game events to discover node failures, instead of actively probing. It also means we need fewer replicas of data to maintain consistency in the face of node failures.

3) Messages will be routed to the correct node: The low failure frequency implies that a key will almost always be routed to the node whose ID is numerically closest to key. P2P systems such as Chord have demonstrated this property even when half of the nodes fail simultaneously [36]. With a much lower failure frequency, it is reasonable to assume that messages eventually reach the correct node.

C. Shared state replication

We design a lightweight primary-backup mechanism to tolerate fail-stop failures of the network and nodes. Failures are detected using regular game events, without any additional network traffic. We dynamically replicate the coordinator once a failure is detected. The algorithm tries to keep at least one replica up under all circumstances, to prevent losses. Furthermore, our replication algorithm does not currently distinguish graceful departures (i.e. quitting) from fail-stop failures (i.e. node crash/disconnect), however keep in mind that graceful departures *can* be handled more efficiently.

Our discussion is based on a single replica, but can be extended to multiple replicas to cope with higher failure frequencies, at the cost of increased bandwidth usage for replication messages. Additional replicas can be added/removed dynamically, in response to join/leave rates.

Failover and replication use the property of the P2P communication subsystem that routes messages with key K to the node whose ID N is closest to K.

As stated above, the object coordinator will be the node whose ID is closest to the ID of the object. Given an object with key K, then the numerically closest node N will be its coordinator. We will similarly make the next numerically closest node M the object replica. ($N, M : \forall I : |N - K| \leq |M - K| \leq |I - K|$.)

This means that a Pastry message with key K will always be routed to the corresponding coordinator N, and should N fail, this message will instead be routed to the replica M. Handling of failover thus becomes very simple—if node M receives a query for K, this implies that node N has failed and that node M should become the new coordinator. The only delay incurred is the time it takes the underlying P2P routing to determine that N has failed.

Since N is the node closest to K, we know that the replica M must be the closest node on either side of N. Pastry keeps the leaf set of N updated, so N just sends to whichever of its nearest neighbors is closest to K. This way, should a replica node fail, the next eligible replica node will automatically become the replica as soon as the leaf set in N is updated. Similarly, if a node joins which is closer to K than M, it will automatically become the new replica.

A newly joined node can also become a coordinator. If a node T where $|N - K| \geq |T - K|$ is added, then T will now receive update requests for object K. On receiving a request for an object K that T does not coordinate, T will find the current coordinator N and request a transfer of all state of K. Until the transfer is complete, T will continue to forward updates to N, but keep copies. Once the transfer is complete, it will apply all the stored updates to the transferred object and take over as coordinator. Thus, should the new coordinator die during a state transfer, the old coordinator will just continue as coordinator. When the new coordinator takes over, the old coordinator becomes the new replica and the old replica will be retired.

With this approach, accesses to the shared objects do not need to block during the data transfer between the old and the new replicas. Data replication can be done in the background, and allows the game to progress with no noticeable delays for the client.

D. Discussions of the replication algorithm

When all but one replica or coordinator is lost, there exists a window of vulnerability during which only a single copy of the consistent state exists, and should that copy also be lost before recovery, consistency is lost.

The size of this window is a sum of the failure detection and the recovery times. Since we use normal game events to detect failures, the failure detection time depends on how often the coordinator is contacted, which is proportional to the number of players in the group. The recovery time is the time to transfer data to the new coordinator or replica, which depends on both the size of the game state and the network characteristics.

We do not know of any comprehensible study of session times in RPGs, but through interviews with players, we have gathered some anecdotal data. Large quests often take multiple hours to complete, with averages of four to five hours. During the quest, players have strong incentives not to leave the game, since doing so may lose them their investment in time. Novice players start with smaller quests that take less time, but session times are not necessarily shorter, partly because of the need and excitement of exploring and experimenting with a new game.

Lacking hard data on RPGs, we have instead used real life measurement of the Gnutella P2P system. It has been found that over a period of 60 hours, the average session time was 2.3 hours [34]. We realize that file sharing and online gaming session lengths are apples and oranges, but given our anecdotal data, 2.3 hours seems conservative, if anything.

Equating failure and exits, this gives a failure percentage per minute of 0.007, i.e. out of 1000 nodes, 7 will fail or exit any given minute.

As we will show in Section VII-G, the window of vulnerability for SimMud is small enough that the replication algorithm can tolerate a relatively high failure frequency without loss of consistency.

Should a catastrophic failure happen, i.e. coordinator and all replicas fail simultaneously, we can still recover the state, although without consistency guarantees. Since every node that has registered itself as interested in the (now lost) state caches it, we can regenerate the lost state from the caches with a high probability.

Isolated network outages are indistinguishable from node failures, except at the affected node, and handled as such. Larger scale outages, such as those that lead to network partitions, are more troublesome. The system can continue to allow shared state access, but with no communication between partitions, the original world would split into two or more parallel worlds, likely with loss of consistency. Brewer neatly codifies the issue in his "CAP conjuncture" [17] which states that a distributed system can enjoy only two out of three of the following properties: Consistency, Availability and tolerance of network Partitions.

One option is to sacrifice consistency for availability and tolerance of network partitions. The real challenge is when the partitions again merge. Since the parallel worlds may have diverged a lot by this time, they cannot be merged back without potentially causing paradoxes. If, however, the game world has a limited lifetime, it could be better to just let the parallel world exist until the next reset and then attempt to sort out any remaining paradoxes in persistent data.

The approach we take is to require that coordinators be "blessed" by the central server, i.e., the node that handles account information and persistent player states. Thus, even if the partition is only partial, meaning that nodes in one partition cannot reach nodes in another partition, but both can reach the central server, the central server refuses to "bless" concurrent coordinators for any given object.

This solution insists on consistency, but will lose availability in the face of network partition. Since the centralized account server is already a single point of failure in our scheme, we do not make the situation worse. Our goal of using a distributed P2P platform was never to improve resiliency, but rather to improve flexibility and scalability. With this approach, availability will be no worse than that of a traditional client-server solution.

VI. Implementation details

We have implemented SimMud on top of FreePastry, an open source version of Pastry [32]. SimMud is purposely simple and mostly unoptimized, since we do not want to obscure the effects we want to study.

A. Map and objects

Each region is described by a two-dimensional array of terrain information, and an object array that tracks the one kind of mutable object, food, that our game models. Each food object consists of a counter that keeps track of how much nutrition (food units) the object consists of.

The player object handles the player's current position and other states. Players can perform three different actions—moving, eating and fighting. Eating is a typical player/object interaction, and fighting is a typical inter-player interaction.

We distribute the computational and communication load by mapping regions to the Pastry key space, as described in the previous section.

Each multicast message for position updates includes player ID, the current location on the map, and a player specific sequence number. The sequence number can be used to detect re-ordered or missing packets.

B. Inter-player interaction

Inter-player interactions are implemented with direct UDP messages. To reduce the opportunity of cheating, all actions are executed on all participating parties, with the same input and algorithm, and the results are exchanged for comparison. In fighting, the damage to each player is usually calculated based on the capability (e.g., skill, health) of each player and a pseudo-random number. The pseudo random number is generated by each player with the same seed, which can be agreed upon either by applying a simple deterministic function over the two player IDs, or from a third party. The coordinator can be used as arbiter where event ordering is important.

C. Object updates

The implementation of object updates aims to reduce the number of message round trips between players and the coordinator while maintaining fairness among the players. We recognize two basic types of object updates. The first type is initialization, where the old value does not matter, e.g., a spell that restores your health. The second type is read-write updates, where the update value depends on the result from the preceding read.

Initializations can be implemented trivially, by just sending the object name and the new value to the coordinator.

For read-write updates, the update is only valid if the current actual value matches the client's cached value. An example would be for a spell that doubles your strength. These updates are implemented by sending the object name, the original value $V1$, and the new value $V2$ to the coordinator. The coordinator keeps a queue of requests. For each request at the top of the queue, the coordinator compares $V1$ with the object's current value to ensure that it is up-to-date. The update will proceed only if two match, otherwise the request is rejected and the current value is sent back to the player. The player will then examine the current value and decide whether to request again.

In some cases, object version numbers are used instead of its value. To illustrate how version numbers are used and why, consider an object that has a value X. A player P1 updates X to X'. Lets assume that a player P2 now updates X' to $X'' = X$. If, at this time, a third player P3 wants to update the value X of the object, then without any additional mechanisms, he would succeed despite the fact that the object has been updated twice compared to the object in P3's cache, and it is only because of the coincidence that $X == X''$ that he would succeed. If this behavior is unacceptable, historic dependency is needed, and this is implemented by adding a monotonically increasing version number to the object. An update will only succeed if the version number provided matches the version number of the object.

VII. Experimental Results

This section presents the experimental results we have obtained with a prototype implementation of SimMud on FreePastry, the open source version of Pastry [32]. We use the network emulation environment in FreePastry to experiment with large networks of Pastry nodes, through which the instances of the node software communicate. The network emulator is largely transparent to the Pastry implementation. We concentrate on the networking aspect of results because the computing load on each node is negligible compared with the bandwidth requirement.

We configured Pastry nodes to run in a single Java VM. All experiments are performed on dual-processor Dell PowerEdge 1550 with 1.2 GHz Pentium III CPUs and 2 GB of main memory. The machines run Linux 2.4.17 and Sun JDK 1.4.1. Both SimMud and our extension of FreePastry are written in Java, and built on top of FreePastry 1.1. We run FreePastry with a key base of 2^4 and a leaf set of 32. The maximum simulation size is constrained by the memory available, and for our system translates to a practical limit of about 4000 virtual nodes.

We analyze the effect of total population, group size, network dynamics, and of our aggregation optimizations on the communication load and message delay of our system. In every instance, the results corroborate our hypothesis that our system has the scalability properties we designed it for.

Since real traces for role-playing and strategy games are not available, we generate our own traces based on a simple, but aggressive, model of typical movements in online role-playing games such as EverQuest. In our model, our simulated players on average eat and fight every 20 seconds and only remain in the same region for 40 seconds. This is more sustained activity than we'd expect from a real player, but compensates for the relative simplicity of our game.

Similarly, in our game, two or three position updates per second would be more than adequate, but we multicast position updates every 150 milliseconds to really stress our system (for comparison, the first-person shooter Quake II broadcasts player position updates every 50ms).

Each region consists of a 200x300 map grid and is described using a 60 KB array. Associated with each region is also a 60KB object array, describing the type and amount of each object. In the base implementation, position and object updates are sent in 200 byte serialized Java records. The map and object arrays are inherently sparse, and in packed format the messages to transfer maps and objects are only $2 * 6$ KB. One replica is kept for each coordinator. For the simulations, we randomize the actions taken by each simulated player, and average the results of 5 runs for each data point.

A. Base Results

Our first measurements of the system are when the game is at a stable state, i.e. when no players join or leave the game, and with no optimizations applied. These measurements will allow us to evaluate the basic performance of the platform. In our first experiments, we measure 300 seconds of simulated game play with an average of 10 players per region. In a real game, we would strive to have an average of less than one player per region by dividing the world into more regions. (For comparison, EverQuest groups are limited to six members, which means our simulation would correspond to having almost two full parties of players present in every region.) In order to obtain message delays, we use a randomly generated network topology in which each link between two nodes is assigned a random link delay in the range of 3–100 ms.

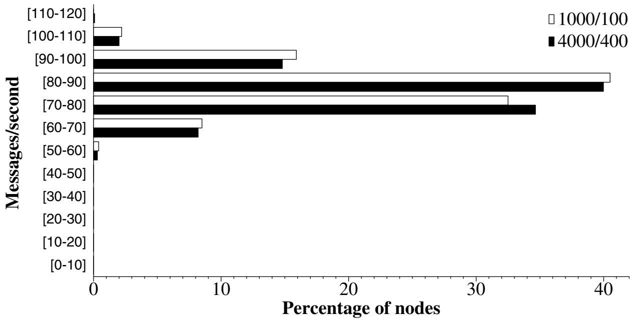

Fig. 2. Distributions of message rate. Average group size is 10. No message aggregation.

Figure 2 presents the distribution of message rates for 1000 and 4000 players with 100 and 400 regions, respectively. Each node receives between 50 and 120 messages per second, which matches our expectations given the region density (10 players/region) and update frequency (about 7/second) yielding $10 * 7$ update messages per second[1]. Eating and fighting take place at intervals of 20 seconds, region change at intervals of 40 seconds, and generate a much smaller number of messages.

The distribution of unicast message hops is similarly illustrated in Figure 3. We find that practically all unicast messages are delivered within six hops. With our simulated delays, we see a maximum delay of about 400ms, with most messages being delivered in less than 200ms.

Most multicast messages are also delivered using six or less hops, but the distribution, illustrated in Figure 4, has a long tail with about 1% taking more than 18 hops and going well beyond 50 hops in the 4000 node experiment. This translates to delays of up to several seconds. This behavior is due to idiosyncrasies in Scribe multicast routing, and although the Scribe authors suggest a solution to this problem [10], it is not implemented in the software available to us.

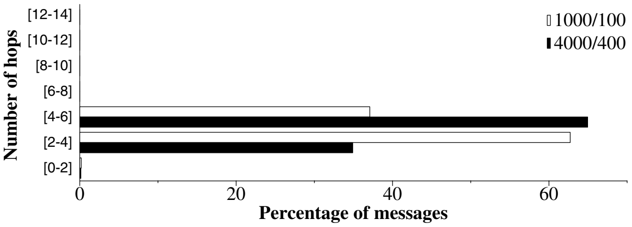

Fig. 3. Distributions of unicast message hops. Average group size is 10. No message aggregation.

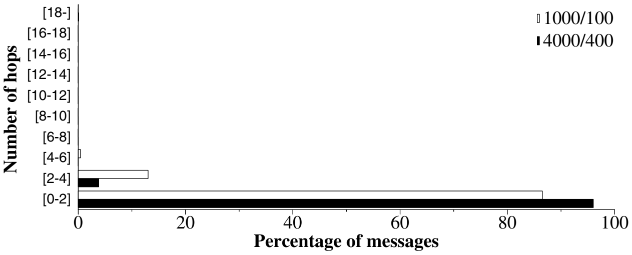

Fig. 4. Distributions of multicast message hops. Average group size is 10. No message aggregation.

B. Breakdown of messages

Table I presents a breakdown of the per second message rate by their functionalities. Without message aggregation (an optimization discussed in Section VII-E), more than 99% of messages are position updates. The remaining messages are split between object updates, player-player interactions and moves across regions. The message rate for object updates is higher than that of player-player interactions, even though they take place at the same frequency. This is because successful object updates are multicast to the region, as well as sent to the replica, while player-player interactions only affect the participating players. Although region changes are the most infrequent events in the system, due to the amount of data involved in this event, they consume more bandwidth than the rest of the operations.

Although the communication is dominated by position updates, it is also important to understand the distribution of other actions so that we can generalize our results to games with different characteristics. Among the rest of the operations, position update is the most likely to generate unbalanced load. Figure 5 presents the message distribution of object updates. It demonstrates that the coordinator task is concentrated on about 10% of the nodes, with a small percentage of the coordinators handling more than one region.

[1] At 200 bytes per message, this translates into a 14KB/s flow of messages, or twice the capacity of a modem. We will explore how to reduce this without resorting to the obvious reductions in message size or update frequency.

Number of Nodes		1000	1000	4000	4000	1000	1000	1000
Number of Regions		100	100	400	400	25	25	100
Message Aggregation		Yes	No	Yes	No	Yes	No	Yes
Failure (node(s) per sec)		0	0	0	0	0	0	1
Total Messages	Average	24.12	82.17	26.98	82.12	26.58	283.13	24.52
	Max	227.62	115.57	216.14	125.18	383.11	341.71	221.38
Average Application Messages		14.76	81.84	14.73	81.7	17.46	282.81	15.09
Average Position Updates		13.34	80.33	13.34	80.18	13.34	278.89	13.31
Object Update Messages	Average	0.85	0.84	0.93	0.95	2.31	2.28	1.07
	Max	8.24	7.82	11.31	10.75	13.92	14.42	13.48

TABLE I

BREAKDOWN OF MESSAGE RATE BY FUNCTIONALITIES.

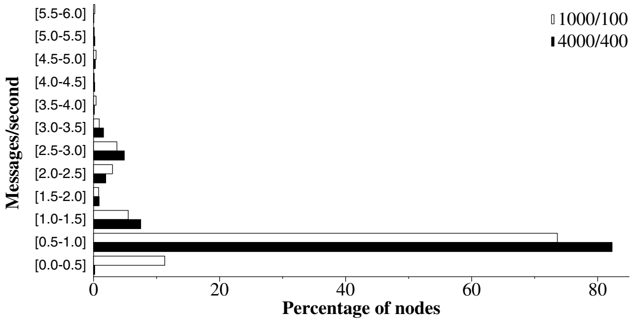

Fig. 5. Distribution of object update messages.

C. Effect of Population Growth

Revisiting the experiments illustrated in Figures 2, 3 and 4, we see that for an average of 10 players per region, the results for 1000 players and 4000 players are quite similar. We can thus conclude that as long as average region density is kept constant, increases in player population can be handled. This is consistent with the approach existing online games take—they expand their world to keep a comfortable population density.

The message delay largely depends on the underlying overlay routing. Pastry routing will typically route a message within $log(N)$ hops, where N is the total number of nodes. This means that routing times will increase with population size, but only very slowly. As discussed in the base results, the long delay in the 4000 player case is largely due to the Scribe's multicast algorithm.

D. Effect of Population Density

We have also evaluated the effect of population density by re-running the experiment with 1000 players and 25 regions, thus increasing the average player density to 40 per region. As we would expect, the number of position updates received by each node increases linearly with the increase in density. Table I shows that, without message aggregation, the number of total message increases by four times when the average players per region is increased from 10 to 40.

Obviously, our scheme is much more sensitive to player density than to the total number of players in the game. This also indicates that non-uniform player distributions could hurt our performance. We propose three ways to deal with this, presented in increasing order of implementation complexity.

The first is to simply use game mechanics to enforce limits on player density, for example by having a gate keeper that will not let players pass unless they are carrying the right amulet. If every region can be "guarded" in this way, that in itself will solve the problem. It will, however, place arbitrary and sometimes strange limitations on what can be done.

The second way requires that the region division be non-uniform. That is, different regions can have different size. This allow us to statically partition regions with higher expected density into many smaller regions. This would make sense e.g. for villages and cities, where walls would naturally create barriers limiting the player's sensory abilities. This approach can be combined with the game mechanics above.

The third way is to allow the system to dynamically re-partition regions when the player density increases. This is significantly more complex than the above two solutions, and harder to combine with game mechanics, but adds the ability to handle unexpected aggregations of players, e.g., if players decide to re-enact Woodstock on an otherwise empty field.

E. Effects of message aggregation

Since updates are multicast, this allows us to aggregate messages in the root before relaying them. This allows us to both reduce the number of messages and to amortize the per-message overhead over more payload, significantly reducing the average per-node load.

We have altered our system so that position updates from all players present in a region are aggregated before being sent out. This means that in the case with 10 players/region, ten update messages will be aggregated into a single message. The aggregated update is 1 KB, about 50% of the size of 10 individual messages, cutting bandwidth requirements by half.

Since the position updates are aggregated at the root of the multicast tree, the number of messages received by the root remains unchanged, while the regular nodes observe a decrease of messages proportional to the number of nodes in the group. This effect is demonstrated in Figure 6. For a majority of the nodes, the message rate drops from 60-

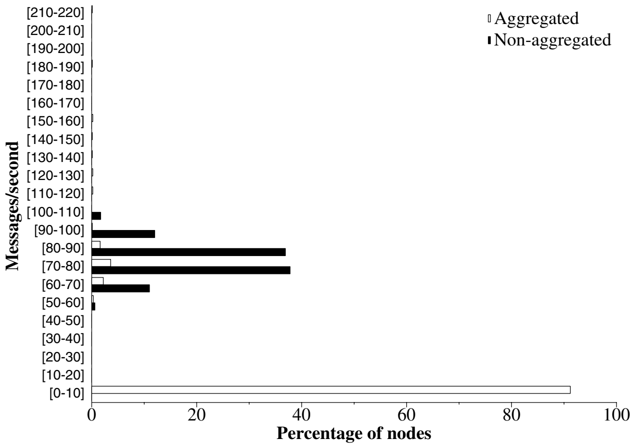

Fig. 6. Effect of message aggregations. 1000 players, 100 regions.

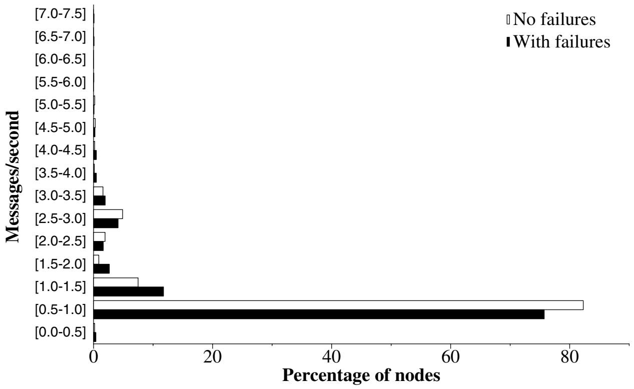

Fig. 7. Coordinator handling overhead with node joins and leaves. 1000 players, 100 regions. One node join and one node leave each second on average.

110 per second to 0-10 per second, leaving only the 10% of nodes that are root nodes to handle the same number of messages as before.

For the same reasons that one node may coordinate multiple regions, a node may also become the root of multiple multicast trees, resulting in a large amount of messages being handled by one node. This mechanism can be further improved by performing aggregation on several nodes, using techniques similar to those of SplitStream [9]. This could allow us to distribute traffic more evenly across the network.

F. Effect of Network Dynamics

Our experiments so far have excluded effects caused by node join and departures. One of the major challenges in distributing an MMG over a peer-to-peer network is that we can expect nodes to be added and removed randomly.

As we've outlined earlier, our failover mechanism will recover from node losses transparently and quickly, but each node join and departure will cause an overhead for transferring state to new replicas and coordinators above what is motivated by normal player movements.

To measure this, we repeat the 1000/100 experiment, but with nodes joining and leaving during the experiment. We simulate one random node join and node depart event per second on average, thus keeping the total number of nodes at approximately 1000 for the duration of the experiment.

This evaluates to a per-node failure rate of 0.06 per minute and an average session length of 16.7 minutes. This is close to the median session length of 18 minutes reported by Henderson and Bhatti on a set of popular Half Life servers [18]. Half Life is a first person shooter (FPS), and we expect significantly longer sessions in RPGs. Using this short session length will, however, allow us to generate a measurable amount of transfers of coordinators and replicas.

In our experiment, the stipulated join/leave rate leads to 112 coordinator migrations, and 173 replica migrations during the 300 second run. The average message rate with node joins and departures is higher than with a stable system, and the overhead caused can be pinpointed by examining the message distribution attributed to handling coordinators, illustrated in Figure 7. The average message rate is increased from 24.12 to 24.52. Even with this artificially high failure frequency, we see that for the majority of nodes, there is little change in the message rate.

G. The possibility of catastrophic failures

The loss of consistent states results from losing both a coordinator and all its replicas in the time-window before the loss is detected and a state transfer can be completed. We define this scenario a *catastrophic* failure. Our system detects the failure of a coordinator only when processing a update request. In our experiments we have a single replica, and region coordinators receives update requests every 1.5 to 2 seconds on average. The transfer size of a region is 12 KB, which means it can be transmitted in less than two seconds even at modem speeds. To account for re-transmission attempts, we will stipulate a window of vulnerability W of 10 seconds.

This means that if, for example, the coordinator crashes, the replica will be the sole repository of region data for the next 10 seconds, after which it will have taken over as coordinator *and* created a new replica. Only if the replica also crashes before it can take over as coordinator and create a new replica will data potentially be permanently lost.

The failure frequency R in our previous experiment is one failure per second, or 60 failures per minute, and the ratio of coordinators to nodes N is $1/10$. The coordinator failure frequency is thus $R*N$, or 6 coordinator failures per minute. If during this window W the corresponding replica dies, we have a catastrophic failure. Since we know that the failure frequency L of nodes in the experiment is 0.06, we can now calculate the catastrophic failure frequency $C = (R*N)*(W*L)$. For our experiment it thus comes to $(60*(1/10))*((10/60)*0.06) = 0.06$, or once every 16 minutes.

Using the more realistic session length of 2.3 hours (see Section V-D for details) and again interpreting all session ends as node failures, we get $1000*0.007 = 7$ node failures per minute. This gives us a catastrophic failure frequency of $(7 * (1/10) * ((10/60) * 0.007) = 0.0008$, or about once every 20 hours.

Assuming a fast network and shorter timeouts before the replica takes over, the window of vulnerability can be shrunk to around two seconds, which would cause the catastrophic failure frequency to drop to once every four hours for our exaggerated failure frequency and once every four days using the more realistic rate derived from the 2.3 hour average session length. Similarly, if we keep the 10 second window of vulnerability and add a second replica, we get catastrophic failures once every 4 1/2 hours and 121 days respectively.

VIII. Related Work

Our target games are Massively Multiplayer Games such as EverQuest and Ultima Online. Most existing MMGs are based on a client/server model, and employ server clusters to improve scalability. Butterfly.net [38] and Zona's TeraZona [38] both provide middleware for grid and cluster based game servers. The client-server approach uses dedicated server resources instead of resources provided by the game players as in our P2P approach. Although the P2P approach is more flexible, and lowers the deployment cost of user designed games, it also incurs a higher security risk because of the game state is distributed to the peers.

Group communications and interest management are also used in many other distributed game implementations, including AMaze [5], Mercury [6], and by Fiedler et al. [16]. Interest management is also used in DIS [8] and HLA [13], distributed interactive simulations that feature large scale virtual environments. Although SimMud shares many techniques from previous efforts in games and distributed simulations, we distinguish ourselves from previous work in that SimMud uses the cycles and bandwidth of the player machines instead of dedicated servers. As a result, state replication is an integral part of SimMud, but has not been considered in previous systems.

Replication is an integral part in peer-to-peer file sharing [33], [12], [22] for both availability and performance. However, those are read-only file systems, while SimMud applies frequent update to shared data. As a result, SimMud must maintain data consistency while tolerating network and node failures. Our approach to always maintain a constant number of replicas is similar to that of CFS [12]. However, the consistency requirements in SimMud, require us to design a replication mechanism that has a small window of vulnerability.

Fault tolerant consistent data services can be built with quorum systems [23], [24], [25]. Unlike SimMud which requires the central server to "bless" one of the partitions, these approaches are completely distributed. In quorum systems, updates cannot proceed if the number of nodes in the partition is not large enough to form a quorum. Fault tolerant consistent data services can also be built on top of view-synchronous group communication [15], [30] using global totally ordered broadcast services [21], [1]. Those approaches provide stronger consistency guarantees than our approach, but may incur much higher performance costs because of the use of totally ordered multicast.

Finally, similar to Seti@home[2] and distributed.net, peers in SimMud contribute not only network bandwidth, but also memory and CPU cycles. In particular, the coordinator contributes compute power to manage shared states. Although this contribution is dwarfed by the corresponding contribution in network bandwidth, consistency mechanism in SimMud can be applied to distributed peer-to-per computing. This will allow peer-to-peer computing platforms to support more sophisticated computing than their current staple—embarrassingly parallel programs.

IX. Conclusions And Future Work

This paper presents the design and implementation of SimMud, a simple massively multiplayer game, on a peer-to-peer overlay. We take advantage of the interest management characteristic in games, and design a scalable mechanism to distribute and map game states to the peers. Furthermore, we present a lightweight replication algorithm that has a narrow window of vulnerability, and can tolerate high failure rates with a small number of replicas.

Measurements with up to 4000 players show that SimMud scales with the number of players. The average message delay of 150ms can be easily tolerated by massively multiplayer games. The bandwidth requirement on a peer is 7.2KB/sec on average, and peaks at 22.34KB/sec. This is well within the capacity of consumer broadband services like DSL. Moreover, with only one coordinator backup, SimMud can sustain a practical failure rate for up to 20 hours, exceeding the interval for which game states are refreshed.

In conclusion, we have demonstrated that a new application, massively multiplayer games, can be supported on peer-to-peer overlays. This application is significantly different from existing P2P applications, which focus on file sharing and content distribution. The shared state distribution and replication mechanism presented in this paper not only can handle games, but can also be extended to handle other forms of peer-to-peer computing.

Much work need to be done beyond this proof-of-concept demonstration. To begin with, SimMud is a simple game and our network emulation assumes uniform latency. We are experimenting with games with larger amount of states on globally distributed network platforms. In addition to further validating this idea, the result may also motivate optimizations of state transfer and latency reduction. Furthermore, cheating is a serious problem in online games, and the problem is exacerbated in the P2P architecture, because a large portion of the game function are executed on untrusted peers. We plan to prevent cheating by detection. Since cheat detection is a resource consuming job, we can distribute the load to the peers. The P2P architecture is potentially suitable for cheat detection, because locality of interest and the basic replication scheme apply to both the game and monitoring of game states.

Acknowledgment

We would like to thank Michael Greenwald, Jonathan Smith and the anonymous referees for their many suggestions that improved the content and presentation of this paper.

This research is supported in part by the DoD University Research Initiative (URI) program administered by the Office of Naval Research under Grant N00014-01-1-0795.

References

[1] Y. Amir, D. Dolev, P. Melliar-Smith, and L. Moser. Robust and efficient replication using group communication. Technical Report CS94-20, The Hebrew Univ. of Jerusalem, 1994.

[2] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: An experiment in public-resource computing. *Communications of the ACM*, 45(17), nov 2002.

[3] Grenville Armitage. An experimental estimation of latency sensitivity in multiplayer Quake 3. In *Proceedings of the 11th IEEE International Conference on Networks (ICON 2003)*, Sydney, Australia, September 2003.

[4] Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof playout for centralized and distributed online games. In *INFOCOM*, pages 104–113, 2001.

[5] E. J. Berglund and D. R. Cheriton. Amaze: A multiplayer computer game. *IEEE Software*, 2(1), 1985.

[6] Ashwin R. Bharambe, Sanjay Rao, and Srinivasan Seshan. Mercury: a scalable publish-subscribe system for internet games. In *Proceedings of the first workshop on Network and system support for games*, pages 3–9. ACM Press, 2002.

[7] Butterfly.net, Inc. The butterfly grid: A distributed platform for online games, 2003. www.butterfly.net/platform/.

[8] C. Bouwens. The DIS Vision: A Map to the Future of Distributed Simulation. *Inst. for Simulation and Training*, 1993.

[9] M. Castro, P. Druschel, A-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. Splitstream: High-bandwidth content distribution in a cooperative environment. In *IPTPS'03*, 2003.

[10] Miguel Castro, Michael B. Jones, Anne-Marie Kermarrec, Antony Rowstron, Marvin Theimer, Helen Wang, and Alec Wolman. An evaluation of scalable application-level multicast built using peer-to-peer overlays. In *Infocom'03*, April 2003.

[11] Clip2. The gnutella protocol specification v0.4 document revision 1.2, 2000. www9.limewire.com/developer/.

[12] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *Proceedings of SOSP'01*, October 2001.

[13] Judith S. Dahmann, Richard Fujimoto, and Richard M. Weatherly. The department of defense high level architecture. In *Winter Simulation Conference*, pages 142–149, 1997.

[14] C. Diot and L. Gautier. A distributed architecture for multiplayer interactive applications on the internet. *IEEE Networks magazine*, 13(4), July/August 1999.

[15] Alan Fekete, Nancy Lynch, and Alex Shvartsman. Specifying and using a partitionable group communication service. *ACM Transactions on Computer Systems*, 19(2):171–216, 2001.

[16] Stefan Fiedler, Michael Wallner, and Michael Weber. A communication architecture for massive multiplayer games. In *Proceedings of the first workshop on Network and system support for games*, pages 14–22. ACM Press, 2002.

[17] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Sigact News*, 33(2), June 2002.

[18] Tristan Henderson and Saleem Bhatti. Modelling user behaviour in networked games. In *Proceedings of ACM Multimedia 2001*, pages 212–220, October 2001.

[19] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. In *SIGCOMM*, pages 328–341, 1995.

[20] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: A decentralized peer-to-peer web cache. In *PODC*, July 2002.

[21] I. Keidar. A highly available paradigm for consistent object replication. Master's thesis, The Hebrew Univ. of Jerusalem, 1994. Technical Report CS95-5.

[22] John Kubiatowicz, David Bindel, Yan Chen, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westly Weimer, Christopher Wells, and Ben Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS*. ACM, November 2000.

[23] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[24] N. Lynch and A. Shvartsman. Rambo: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Symposium on Distributed Computing*, pages 173–190, 2002.

[25] Nancy Lynch, Dahlia Malkhi, and David Ratajczak. Atomic data access in content addressable networks. In *Proceedings of the 1st International Workshop on Peer-to-Peer*, March 2002.

[26] Alan Mislove, Ansley Post, Charles Reis, Paul Willmann, Peter Druschel, Dan S. Wallach, Xavier Bonnaire, Pierre Sens, Jean-Michel Busca, and Luciana Arantes-Bezerra. Post: A secure, resilient, cooperative messaging system. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, may 2003.

[27] Katherine L. Morse. Interest management in large-scale distributed simulations. Technical Report ICS-TR-96-27, University of California, Irvine, 1996.

[28] Paul Bettner and Mark Terrano. GDC 2001: 1500 Archers on a 28.8: Network Programming in Age of Empires and Beyond, March 2001. www.gamasutra.com/features/20010322/terrano_01.htm.

[29] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[30] Robbert Van Renesse, Kenneth P. Birman, Bradford B. Glade, Katie Guo, Mark Hayden, Takako Hickey, Dalia Malki, Alex Vaysburd, and Werner Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Cornell University, 23, 1995.

[31] Sean Rhea, Timothy Roscoe, and John Kubiatowicz. Structured peer-to-peer overlays need application-driven benchmarks. In *Proceedings of 2nd International Workshop on Peer-to-Peer Systems*, February 2003.

[32] Antony Rowstron and Peter Druschel. Pastry: scalable, decentraized object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, November 2001.

[33] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, A large-scale, persistent peer-to-peer storage utility. In Greg Ganger, editor, *Proceedings of SOSP-01*, volume 35, 5 of *ACM SIGOPS Operating Systems Review*, pages 188–201, New York, October 21–24 2001. ACM Press.

[34] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[35] Nathan Sheldon, Eric Girard, Seth Borg, Mark Claypool, and Emmanuel Agu. The effect of latency on user performance in Warcraft III. In *Proceedings of the 2nd Workshop on Network and System Support for Games (NetGames 2003)*, pages 3–14, 2003.

[36] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In Roch Guerin, editor, *Proceedings of SIGCOMM-01*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York, August 27–31 2001. ACM Press.

[37] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, UC Berkeley, April 2001.

[38] Zona Inc. Terazona: Zona application frame work white paper, 2002. www.zona.net/whitepaper/Zonawhitepaper.pdf.