A Distributed Architecture for Massively Multiplayer Online Games*

Chris GauthierDickey†
Department of Computer Science
1202 University of Oregon
Eugene, OR 97403-1202
chrisg@cs.uoregon.edu

Daniel Zappala
Department of Computer Science
1202 University of Oregon
Eugene, OR 97403-1202
zappala@cs.uoregon.edu

Virginia Lo
Department of Computer Science
1202 University of Oregon
Eugene, OR 97403-1202
lo@cs.uoregon.edu

ABSTRACT

To date, massively-multiplayer online games (MMOGs) have been inherently limited by the client/server architecture. While some researchers and companies have proposed using grid or clustered servers, we propose a fully decentralized architecture. Our unique contribution is an architecture that scales with the number of players in communication, storage, and computation. Further, the architecture uniquely allows a single individual to deploy a MMOG without the incredible investment required by previous architectures.

1. INTRODUCTION

Traditionally, massively multi-player online games, called MMOGs, have used a client/server communication architecture. This architecture has the advantage that a single authority orders events, resolves conflicts in the simulation, acts as a central repository for data, and is easy to secure. On the other hand, this architecture has several disadvantages. First, it introduces delay because messages between players are always forwarded through the server. Second, traffic at the server increases with the number of players, creating localized congestion[1]. Third, the size of the virtual world and the quantity the data it contains is limited by the storage capacity of the server. Fourth, the complexity and richness of the game is limited by the computational power of the server. While we can throw technology at most of these problems in the form of more servers and higher bandwidth lines, this solution incurs significant cost.

A distributed architecture for MMOGs has the potential to overcome these problems. Players can send messages directly to each other, thereby reducing delay and eliminating localized congestion. The storage capacity created by combining the storage resources of all players can easily exceed that of a single server. Furthermore, the ability to execute thousands of remote processes in parallel on user machines provides unparalleled computational power for MMOGs. Finally, this architecture would allow individuals to start their own MMOG without the incredible investment in resources required by client/server architectures.

However, a distributed architecture has several fundamental and challenging problems that must be overcome:

- Authenticating players and granting access rights to the game.
- Maintaining consistency, ordering events, and propagating events to intended recipients
- Providing tamper-resistant storage of characters and game state
- Scheduling computations across the players
- Providing responsiveness to interactive elements of the game
- Ensuring the architecture is *cheat-proof* by preventing players from taking advantage of the architecture itself in order to cheat.

In this paper, we present our distributed, peer-to-peer architecture for MMOGs. We explain the choices we made in our design process and point out alternatives. We then conclude with open problems and future work related to designing a distributed, peer-to-peer architecture for MMOGs.

2. BACKGROUND

In this section, we provide an overview of past research related to distributed architectures for multiplayer games. We describe work specifically on game architectures first and follow with research generally related to distributed architectures.

2.1 Game Specific Research

Diot, Gautier and Kurose described the first protocol for distributed games in [7, 11] and built a game called MiMaze to demonstrate its feasibility. Their work is important because they developed a technique called bucket synchronization, in which game time is divided into 'buckets', in order

*This is a draft paper, please do not redistribute.

†Supported by a NSF GRF

[1]One local game developer states that the bandwidth requirement for their massively-multiplayer game is equivalent to the city of Eugene's telephone bandwidth.

to maintain state consistency among players. The MiMaze protocol uses multicast to exchange packets between players, resulting in a low latency; however, it does not address the problem of cheating.

At the other end of the spectrum, Baughman and Levine designed the *lockstep* protocol to address the problem of protocol level cheats [2]. Lockstep uses rounds for time, which are broken into two steps: first, everyone reliably sends a cryptographic hash of their move, then everyone sends the plain-text version of their move. This forces everyone to *commit* their move, without revealing it, thereby preventing anyone from knowing someone else's move ahead of time. To mitigate the problem of delay introduced by reliable transport, event scoping (called asynchronous synchronization) was added to lockstep. Lockstep is a major advance in communication protocols for distributed games because it is provably secure against several cheats. Unfortunately, time progresses in the lockstep protocol at the speed of the slowest player and therefore cannot meet the bounded time requirements necessary for games.

Cronin et al. designed the Sliding Pipeline (SP) protocol[6] in order to reduce *jitter* in the lockstep protocol. They add an adaptive pipeline that allows players to send out several moves in advance without waiting for ACKs from the other players, reducing the time that is dead-reckoned between rounds. The pipeline does not reduce the latency inherent in lockstep, and furthermore is subject to cheating.

Bharambe et al. have proposed Mercury, a distributed publish/subscribe communication architecture [1]. Mercury provides channels, which can be of any subject, uses a subscription language (that is a subset of relational database query languages), and uses rendezvous points (RPs) to gather and disseminate publications. Unfortunately their results show that it cannot meet the performance requirements for MMOGs due to the routing delay introduced by their architecture [1].

Knutsson et al. also designed a publish/subscribe system [12] using Pastry [18] and Scribe [19]. The virtual world is divided into regions, and players in each region form a group. Each region maps to a multicast group through Scribe so that updates from the players are multicast to the group. Consistency is achieved through the use of *coordinators*. Every object in the game is assigned to a coordinator; therefore, any updates to an object must be sent to the coordinator who resolves any consistency problems. Fault tolerance is achieved through replication. However, their system does not fully take cheating into consideration in its design, but instead relegates it to future work.

In the game industry, very few networked games are fully distributed. One notable exception is Age of Empires (AoE) [3], in which games are synchronized across clients and peer-to-peer communication is used. AoE's protocol is similar to bucket synchronization, except that unicast is used. While AoE is a commercial success for distributed game protocols, it is subject to many cheats at the protocol level.

Butterfly.net [4] is a grid-based solution for developing MMOGs. Using the Globus toolkit [23], developers are able to lease computer time and space from Butterfly.net to run their MMOGs from. Terazona [22], on the other hand, is a cluster based architecture that allows developers to more easily create clustered solutions to act as the servers in a traditional client/server architecture. Neither of these products have the same scope and goals of our architecture.

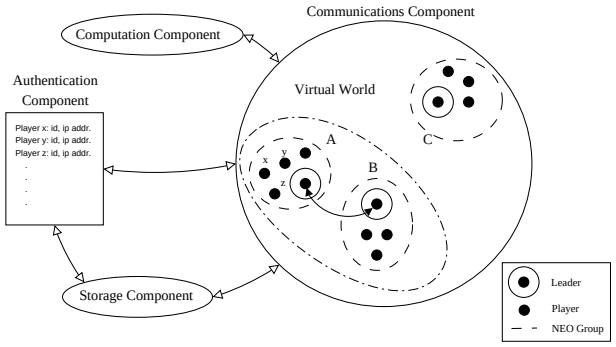

Figure 1: Our peer-to-peer game architecture

Finally, we note that the area of distributed interactive simulation (DIS) addresses some of the same issues, but all participants are trusted, so the DIS protocols are not concerned with cheating participants.

2.2 General Related Work

Because we are proposing a distributed architecture, past research related to distributed systems directly applies to our research. Foremost of these is the consistency impossibility results from Lamport et al. in [15]. These results show that without some form of digital signatures on messages, it is impossible to agree on the state of a distributed system if $n < 3f$, where n is the number of participants and f is the number of failed systems.

Every player in a distributed game must have the same ordering of events in order to maintain sequential consistency, which is defined by Lamport in [14]. Sequential consistency is where the results of any execution in a distributed system are the same as they would be if executed on a single system in some given order.

Carter et al. describe release consistency in [5]. In release consistency, events between synchronizations points are unordered, but ordering is enforced across synchronization points. This technique is similar to bucket synchronization in [7]. Last, we note that Dubois et al., provide a general survey of coherence and synchronization for distributed systems in [8].

3. A DISTRIBUTED, PEER-TO-PEER GAME ARCHITECTURE

We now present our architecture and more thoroughly describe the challenges in building a distributed system. We divide the architecture into four main components: authentication, communications, storage, and computation. First, the authentication component is responsible for controlling access to the game. The communication component determines how players send messages to each other and the storage component provides long-term storage of world state. Last, the computation component schedules computations across the player base. Figure 1 is an abstract view of our architecture.

3.1 Authentication

Authentication and access control are necessary for any multiplayer game. For example, a game which requires a monthly subscription should only allow players who have

paid to join the game. A player who has been banned from a game should not be allowed to rejoin the game.

The challenging problems with distributed authentication are security and scalability. How can authentication be distributed and how can it scale as the number of users to authenticate increases? Research related to distributed authentication typically makes the assumption that distributing authentication simply means that we wish to have multiple trusted authentication servers such that as long as a minority of them are not compromised, we can still securely authenticate and prevent unauthorized access [21]. However, in our architecture, we assume that someone deploying a game might only have one server and would wish to distribute authentication to the players of the game. This presents the much more difficult research problem of how we can distribute authentication over untrusted servers.

We propose using a trusted server to bootstrap player accounts, combined with a distributed hash table (DHT) [20, 17] to provide scalability as the system grows.

The trusted server assigns each player a unique ID when they join the game for the first time. The server then cryptographically signs two pairs: (player_id, expiration_date) and (player_id, public_key). Digital signatures prevent anyone from tampering with the pairs and because every player knows the trusted server's signature, they can easily determine the validity of a pair. The (player_id, expiration_date) pair acts as a player's game key. The expiration date shows when a key expires so that valid keys can still be rejected based on their expiration date. The second pair, which has the public key of a player, allows players to verify the identities of each other.

In addition to the pairs that the trusted server generates, it also can generate a signed pair, (player_id, banned), that can be published to prove that a player has been banned from the game. This feature is necessary because any multiplayer game needs the ability to remove players from the game which are misbehaving or who are no longer authenticated.

To help with scalability, once the pairs are generated, they are stored on a DHT by the player or server. This allows players to efficiently look up the public key of another player without needing to contact the server. It also allows players to quickly discover if a player has been banned who they are currently interacting with. The advantage of using a DHT is that as the number of players increase, the storage capacity increases. Ideally, the trusted server will only need to be contacted on rare occasion.

We now present several scenarios to demonstrate how the authentication component (AC) works:

1. *A player starts a new account:* In this scenario, a new player joins the game. For our example, we will assume that the game has a monthly subscription fee. The player pays their subscription online and provides a user name and public key for her game account. The payment system communicates with the AC and sends it the new player's user name and public key. The AC then generates the two pairs (player_id, expiration_date) and (player_id, public_key) and sends them back to the player. Once the player has these keys, she publishes them in the DHT.

2. *A player joins the game:* When a player who already has an account joins the game, she queries the DHT to make sure her keys exist. If they don't exist, she publishes them so that other players can discover her public key efficiently without needing to contact her or the AC.

3. *A player receives a message:* When a player receives a message from another player that she has not exchanged messages with recently, she queries the DHT to make sure that the game key is valid. If she finds a banned key, she ignores all messages from the player. Note that the only key that can be invalid is one that was previously valid but for some reason has been marked as invalid (such as because the player was caught cheating or their account has expired). This holds due to our assumption that keys are digitally signed and cannot be forged. Forged keys are easily detected by players since every player can identify the trusted server's signature.

4. *A player is caught cheating:* Whenever the system can prove a player has cheated, for example by sending two inconsistent packets to different players, other players can send this proof to the AC. The AC then invalidates the players ID and game key.

3.2 Communication Component

The primary advantages of using a distributed communication component are scalability, responsiveness and resiliency. At this time, most client/server architectures support several thousand players concurrently. A distributed communication architecture has the possibility of supporting millions of players. In addition, with client/server architectures, all communications must first be sent to the server before being sent to players, introducing additional delay. Distributed communication allows players to send messages directly to each other, creating a more responsive game. Last, the client/server architecture results in a single point of failure in the network. Even a grid-based architecture, where clients communicate with several servers, does not provide the resiliency that distributed communication can achieve and causes localized congestion close to the servers. Ideally, with distributed communication, a system that fails only affects that single player.

The key challenges in developing a communications component for a distributed architecture are consistency, event ordering, interactive responsiveness, security, and scalability. Consistency is required because if two players have different state, then their simulations will arrive at different conclusions, possibly creating a situation where they can no longer play together. Event ordering, which is closely related to consistency, is required so that players can determine when events occurred that affect them. For example, if two players open a chest and try to take out a treasure, events must be ordered so that only one of these players will succeed. Communication must also be bounded by time so that the game will remain interactive. Protocols which do not explicitly bound how long it will take to complete an exchange of messages run the risk of being too slow to maintain interactive rates of play. Security is also extremely important for communications in games. Experience has shown that when players are given the chance to cheat in a game, they most certainly will. In addition to these challenges, the communication component must scale with the number of

players. With a distributed communications component, we hope to scale to millions of players.

Current massively multiplayer games divide a virtual world into *zones*, which are areas created by game designers that have specific content in the game. For example a zone might represent a city, dungeon, wilderness area, or even a single building. Each zone typically has its own set of monsters, non-player characters, items, and events. In addition, most zones are isolated. For example, when a group of players enters a building, they no longer receive updates from players outside of the building. Furthermore, zones are often nested. Using our building example, the building might be a single zone, but can further be subdivided by using a zone for each floor of the building.

Dividing communication between zones is an obvious choice. Since events rarely escape the boundaries of a zone, this immediately reduces some communication between players. However, the number of a zones is typically very small compared to the number of players in a game, especially when we are considering millions of players. Zone boundaries are also fixed statically during the design of a game. If every player in the virtual world decided to visit the same zone concurrently, then communication between all players would quickly break down.

Because zones are insufficient for surmounting the scalability problems with communication, we subdivide each zone in the virtual world into a self-organizing hierarchy of peer-to-peer groups. This model works well with games because events in one part of a zone typically do not affect other parts of the zone.

To organize the peer-to-peer networks into a hierarchy of groups, we use distributed election protocols to elect one or more leaders to act as *nodes* in the hierarchy. The nodes assist in event propagation and group location. To help build the hierarchy of groups, we propose using a DHT to map unique zone IDs to the list of nodes in a given zone. As players enter new zones, they query the DHT to discover the nodes of the hierarchy so that they can locate the appropriate group they belong in. Some events affect multiple groups, such as the lights going off in a building; in these cases, the event is propagated through the world hierarchy by group leaders in each peer-to-peer network.

While a hierarchy can be built in a number of ways, we propose using a quadtree (or octree for 3-dimensional partitioning) [9] where each leaf in the quadtree maps to a peer-to-peer group in the game. We believe using a quadtree might allow for efficient forming and dividing of groups since quadtrees can be easily split and joined in O(lg n) steps, where n is the number of groups in the zone. However, we note that efficient grouping algorithms are still open research questions.

At the lowest level of granularity, our provides consistency and event ordering using a new protocol called NEO [10]. We have focused on developing NEO as the first building block of our architecture because the viability of any game depends on timely and secure event ordering. NEO ensures low playout latency and also prevents cheating by using several techniques (majority voting, cryptography, and game state hashes). Unlike the lockstep protocol [2] which constrains playout latency by the delays incurred by the worst message latency (slowest player), NEO uses fixed-length rounds to ensure that messages arrive in a timely manner.

3.3 Storage Component

Increasing the storage capacity of a MMOG creates many interesting possibilities. We claim that the design of current areas in MMOGs is restricted by the storage capacities of the client/server architecture. One example is the *expiration* of items left in an area of the virtual world. In order to save memory and storage, items left 'lying around' on centralized games disappear. Given a much larger storage capacity, a game could let objects decompose naturally, creating a more compelling playing experience. One could easily imagine how littered a city would become in a MMOG if dropped items took a long time to decay! In addition, the quantity of objects in a game is simply limited by the amount of memory a server has, which explains the sparsity of most games. Distributing storage of game state could overcome the current limitations of the client/server architecture.

To provide distributed storage, several fundamental problems must be solved, including consistency, reliability, security, and scalability. To maintain consistency, writes must be ordered so that the final results in storage are the results that all players expect. Storage must also be available and reliable. Data that is stored should always be available for retrieval. Players would not be acquiescent if their characters were lost because the person that was storing their character left the game! Stored data must also be secure so that it can not be tampered with. Last, for a MMOG to scale to millions of players, the storage capacity of the system must scale appropriately.

Recent research in peer-to-peer systems has shows that peer-to-peer storage systems can meet most of the storage requirements of a distributed game. Structured peer-to-peer systems, such as DHTs like CAN [17], Chord [20] and Pastry [18], along with distributed file systems such as OceanStore [13], have been shown to *increase* in reliability, availability, and scalability as the number of participants increase. However, one problem faced by these systems is *churn*, or the rate at which new parties join and leave the system. As churn increases, the reliability decreases since data is lost with the leaving member. Further, network traffic increases with churn because data is relocated in order to distribute it over the peer-to-peer network.

Our architecture addresses the problem of churn by separating storage needs into two categories: permanent and ephemeral data. Permanent data must *always* be available to the player while ephemeral data can be lost. Zone data is an example of ephemeral data, since this data can be regenerated by players from the zone itself. For example, a player drops her sword in the forest. The sword is stored as ephemeral data because if the player leaves the game and this data is lost, it does not affect the overall game (though she might think it does!). Note that long-term persistence is a distinguishing feature of massively-multiplayer games. Character data, for example, is stored for the lifetime of a player's account–which is often years.

Ephemeral data makes up the bulk of data in a MMOG, but churn does not affect ephemeral data since it can be lost. On the other hand, the storage system must be more resilient with respect to permanent data.

Permanent data can further be broken down into data which always exists (which we call *existential data*) and data that exists only when the player is in the game (which we call *participatory data*). Character data and inventories are examples of participatory data because it exists only when

the player is in the game. A player-owned house is an example of existential data that always needs to exist, even when the player is not online.

3.3.1 Ephemeral Data

The storage component handles ephemeral data in the following manner. Data which is ephemeral is simply stored on the DHT, indexed by its geographical area in the virtual world. Players that enter an area, query the DHT for all ephemeral data. Because the players are using consistency protocols for communication, we can assume that the players will agree on the state of the data. As changes to the data occur, these changes are saved back to the ephemeral data in the DHT.

Note that zones in a MMOG are periodically *reset*. Resetting is when the the objects, non-player characters, and processes associated with a zone are re-created. This is necessary because players consume the resources of a zone as they play in it (ie, they kill the monsters and take the treasure). When a zone is reset, its data is stored as ephemeral data in the DHT. In the case where some part of the DHT fails and data is lost, the zone will be replenished on the following reset.

3.3.2 Permanent Data

To address permanent storage, we propose the following methods. For participatory data, players can store this data locally and on the DHT. This allows the player to make sure the data is always available when they are online, but provides a backup in case their local data becomes corrupted. However, local storage cannot be done without considering the security implications. If a player can store their own character data, what prevents them from altering the data to their advantage?

When the game is sufficiently small, the trusted server can act as a witness to player actions and digitally sign their character data. Only signed characters are allowed to join the game. To provide scalability as the game grows, other players are allowed to sign character data. Since we assume that NEO maintains consistency, then all players in a NEO group agree on the state of the game and can therefore reliably sign character data. To prevent collusion, randomly chosen players also witness and sign character data. However, what prevents a group of colluding players from signing each of their character data as they choose? How can a player prove that their character data has been signed by unbiased other players and not a group of friends? This remains an open problem in designing the storage for a distributed system.

One interesting problem that needs to be addressed with this system is how to provide privacy. Using our technique, any player can query the DHT for another player's character. In competitive games, this would allow to players to determine the exact abilities and inventory that a competing player has. Under this scenario, a system would need to be enabled that divided players into their respective competing groups. The DHT would instead store an encrypted version of the character data. A key to decrypt the data would then be sent only when players needed to interact with each other.

To understand the encrypted system more fully, let us assume we have two players, Alice and Eve, which are about to engage in combat with each other. First, Alice and Eve send each other keys that decrypt only the ability portion of their character data plus their currently equipped items so that each player could fully simulate the combat (and make sure the other player was not cheating). However, inventories remain encrypted. Now, let us assume that during the combat, Alice decides to equip her machine gun. At that moment, Alice sends the key that decrypts her inventory so that Eve can see that indeed Alice does have a rounds that she can use!

3.4 Computation Component

The last component, the computation component, is used to schedule game computations among the players. The ability to harness millions of players' processors creates exciting possibilities for the virtual experience that players can participate in. A common complaint in MMOGs is the relative 'dumbness' of monsters. The server in a client/server architecture typically does not have enough power to simulate complex interactions between players and the artificial intelligence (AI) behind monsters and other non-player characters. Using on-the-fly scheduling of processes for AI among the millions of players could lead to a richer gaming experience. Furthermore, being able to harness these processors in parallel could open new possibilities for game simulations such as more complex weather simulations, simulating complex population models for monsters and vegetation, and other global-scale events.

Distributed computation has several difficult hurdles before it comes to fruition. First, scheduling processes in a decentralized system is a difficult problem. Recent work in Cluster Computing on the Fly [16] has shown promising avenues. Second, inter-process communication is slow over the internet. For some processes, this does not pose a problem. For example, an AI process that is interacting with players does not need instantaneous responsiveness. Third, the distributed processes must be secure from cheating. Specifically, how can we trust players to execute processes accurately?

The problem with allowing a player to execute processes local to them is evident in the following example. Assume the player Eve desires to cheat. She can simply move to an isolated location in the virtual world and since no other player is simulating the processes in her zone, she can have the processes execute however she pleases. Perhaps she has monsters give her all of their treasure and then kill themselves.

To combat the problem of local process execution, a set of randomly chosen players executes interactive processes for each group. We randomly choose players to reduce the chance that the group can collude. Players can also locally execute processes in order to verify the results produced by the remote players. Inconsistent results produced by remote players indicates that one of them is cheating. Since all messages are digitally signed, a cheating player can be detected and the proof of her cheating (in the form of her signed messages) can be submitted to the AC for appropriate action (such as invalidating their account). Since players are only responsible for a small number of remote processes, the total number of processes the MMOG can execute scales with the number of players.

To determine how many players we need to randomly choose for process execution, we first decide what our tolerance level, T, is for cheating. This means that we need to

choose r of n total players such that the chance that all r players are colluding is less that T. If we assume that up to c players will collude, then we should pick r such that:

$$\frac{\binom{c}{r}}{\binom{n}{r}} < T$$

Equation 3.4 shows that the probability of a cheater escaping detection is the number of ways we can choose r players from c over the number of ways we can choose r players from n. As r increases, the probability that cheaters can escape detection is reduced. Consequently, when $r > c$, then we will always detect cheaters.

4. FUTURE WORK

In this paper we have outlined the possibilities of fully distributing a massively-multiplayer online game. We discussed the problems inherent in using a distributed system and methods for circumventing these issues. We presented our solutions as an architecture for distributed MMOGs.

With this architecture as a possible framework for distributed MMOGs, our future work is to research the many open problems in this architecture. Specifically, we are investigating how we can scalably build a cheat-proof hierarchy of NEO groups in a virtual world, while maintaining real-time interactiveness. Our next step is to form a hierarchy of NEO groups using a quadtree to determine whether this structure can be both responsive and secure.

Many open problems still exist in this architecture. First, can we distribute authentication to untrusted players? Second, how can we prove that modifications to permanent data are valid in light of the possibility of collusion between players? Third, how can we schedule processes efficiently in the network? Finally, what other methods besides replication can be used to verify computations by untrusted players?

5. REFERENCES

[1] S. S. A. R. Bharambe, S. Rao. Mercury: A scalable publish-subscribe system for internet games. In *Proceedings of the First Workshop on Network and System Support for Games.*, April 2002.

[2] N. E. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *INFOCOM 2001*, pages 104–113, 2001.

[3] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in the age of empires and beyond. In *Game Developer's Conference*, March 2001.

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

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Techniques for reducing consistency-related communication in distributed shared-memory systems. *ACM Transactions on Computer Systems*, 13(3):205–243, 1995.

[6] E. Cronin, B. Filstrup, and S. Jamin. Cheat-proofing dead reckoned multiplayer games. In *International Conference on Application and Development of Computer Games*, January 2003.

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

[8] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, coherence, and event ordering in multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[9] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics*, chapter 12, pages 550–555. Addison-Wesley, 1996.

[10] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low Latency and Cheat-proof Event Ordering for Peer-to-Peer Games. In *NOSSDAV*, June 2004.

[11] L. Gautier, C. Diot, and J. Kurose. End-to-end transmission control mechanisms for multiparty interactive applications on the internet. In *IEEE Infocom*, 1999.

[12] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-Peer Support for Massively Multiplayer Games. In *IEEE Infocom*, March 2004.

[13] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ASPLOS*, November 2000.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. C-28(9):690–691, September 1979.

[15] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. In *ACM Transactions on Programming Languages and Systems*, pages 382–401, July 1982.

[16] V. Lo, D. Zappala, D. Zhou, Y. Liu, and S. Zhao. Cluster Computing on the Fly: P2P Scheduling of Idle Cycles in the Internet. In *Third International Workshop on Peer-to-Peer Systems (IPTPS)*, February 2004.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of ACM Sigcomm*, pages 161–172. ACM Press, 2001.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. 18th IFIP/ACM Int'l Conf. on Distributed Systems Platforms*, Nov. 2001.

[19] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[20] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of ACM Sigcomm*, pages 149–160. ACM Press, 2001.

[21] T. X. Y. Woo and S. S. Lam. Authentication for Distributed Systems. In *Computer*, volume 25, pages 39–52, January 1992.

[22] Zona Inc. Terazona: Zona application framework whitepaper. www.zona.net/whitepaper/Zonawhitepaper.pdf, 2002.

[23] The Globus Alliance. http://www.globus.org.