# Distributed Architectures for Massively-Multiplayer Online Games

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

## 1 Introduction

Traditionally, multi-player games 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]. Last, this architecture is limited by the computational power and storage capacity of the server. While we can throw technology at most of these problems in the form of more servers, disk farms, and higher bandwidth lines, this solution incurs significant cost.

To address these problems, I propose researching fully distributed, peer-to-peer architectures for massively-multiplayer[2] online games (MMGs[3]). This architecture allows peers to send messages directly to each other, reducing the delay for messages and eliminating localized congestion. It allows players to start their own games without the incredible investment in resources required by a client/server architecture. Furthermore, this architecture allows games to overcome the bottleneck of server-only computation and storage

[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.

[2]According to the game industry, massive means anything on the order of $10^4$ or greater

[3]MMO is the current industry acronym, though I think MMG is easier to remember. Other acronyms include MMPG, MMOG, and MMPOG.

by harnessing the processing power and storage capacity of the players' machines through peer-to-peer . Finally, this architecture is more resilient and available because it does not have a single point of failure.

However, in order to develop a peer-to-peer architecture for MMGs, we must understand the research in three fundamental areas: network communication, peer-to-peer storage, and peer-to-peer computation. In this paper, I cover the primary results from these areas and discuss their relevance to the peer-to-peer architecture for MMGs.

## 2 Definition of an MMG

A massively multiplayer online game is a networked game with two distinguishing features. First, the magnitude of the number of concurrent players is typically on the order of $10^4$ or more. Second, MMGs have persistent state. This means that an MMG, unlike other networked games which end after some goal is completed, can continue indefinitely. Players join the game and play until they are ready to quit, at which point the state of their alter-ego in the game is saved. When they return, the state is restored. This also holds true for the virtual world. For example, Alice might own a house in an MMG which other players can visit even when she is not online.

MMGs typically fall into three different archetypes, which we define as *first-person shooters* (FPS), *role-playing games* (RPG), and *real-time strategy games* (RTS). In an FPS, the main goal is typically to kill the other players with various weapons from the virtual world. A player can only sustain several hits, depending on the weapon, which results in the need for fast reflexes and reaction times. The *twitch* nature of FPSs requires a low network latency of 150-250ms.

In an RPG, one of the primary goals in the game is to develop one's alter-ego by increasing abilities and gathering new equipment. These goals are achieved by grouping with other players to explore new lands, kill monsters, and sometimes by fighting with other players. However, combat in RPGs is resolved through a mathematical system based on the alter-ego's abilities and equipment. Thus, players do not need the same kind of responsiveness from a game.

In an RTS, a player is in control of *units* in a virtual world, where a unit might be a soldier, vehicle, or building. The player acts as the commander of the units and instructs them on what actions to take. Players compete with each other to either destroy all the units of the other player, or capture some vital piece in a game. As with RPGs, combat is not resolved simply by

| Archetype | Tolerated Latency | Example MMG |
|---|---|---|
| First-person shooter (FPS) | $150 - 250ms$ | Planetside |
| Role-playing game (RPG) | $500 - 1000ms$ | Everquest |
| Real-time strategy (RTS) | $< 1.5s$ | *None developed* |

Table 1: Archetypes for MMGs, tolerated latencies, and industry examples

clicking on another player to shoot her, but by ordering the units to attack other units. Thus, players have been reported to tolerate latencies up to 1 second, without it detracting from the game [BT01]. Table 1 summarizes these types of games.

Considering these latencies, a peer-to-peer architecture must provide low-latency communication so that the interactive nature of the MMG is maintained. In addition, the architecture needs to provide long-term, reliable persistence of the game state. Finally, the architecture must distribute and coordinate process execution among the peers.

The rest of this paper is organized as follows: Section 3 covers the fundamental issues in designing any distributed system. Section 4 covers network communication and considerations when designing low-latency protocols for MMGs. Section 5 discusses peer-to-peer storage research while Section 6 covers peer-to-peer computation. Section 7 covers scientific research that specifically covers games. Finally, Section 8 summarizes how these areas relate to a P2P architecture for MMGs and briefly discusses the direction I am interested in.

## 3 Foundations of Distributed Systems

In this section, I describe the definition of a distributed system with respect to how it applies to my research area. I then describe the fundamental problems and results of distributed systems research: synchronization, consistency, and event ordering.

### 3.1 Definition of a Distributed System

In a traditional monolithic system, a single processor interleaves the execution of multiple processes running in the system. All processes share the same processor, caches, memory, and disk space. When multiple processes must communicate and coordinate with each other, a monolithic system uses shared memory or message passing through the operating system. The advantage of the monolithic system is that certain events, such as loading and storing a variable in memory, are easily designed to be atomic at the hardware level—hence, we can implement a variety of synchronization mechanisms at the hardware level.

A distributed system is defined as one in which the processes only coordinate through message passing [CDK01]. This definition is appropriate because it implies that no hardware will be used to synchronize processes or maintain consistency. In other words, I am interested only in those systems that are distributed over a network such as the Internet, and not multiprocessor systems which can use hardware to synchronize and maintain coherence in the system. This narrower definition makes sense in the context of distributed real-time interactive applications where the whole point of the application is for collaboration over a wide-area network.

Distributed systems are divided into two models: synchronous and asynchronous. In the synchronous model, processes execution occurs in synchronous rounds (i.e., they proceed in lockstep)[Lyn96] according to a global clock. The advantage of the synchronous model is that it is easier to reason about, with the caveat that most real distributed systems are not completely synchronous. In the asynchronous model, processes execute local instructions at arbitrary speeds. This model has the advantage that algorithms designed for it can run on *all* types of networks without timing guarantees. The disadvantage of the asynchronous model is that some problems are more difficult, if not impossible, in the asynchronous system [Lyn96].

Regardless of the distributed system model, three fundamental problems plague the design of distributed systems: Ordering of events, synchronization of processes, and consistency of data. Event ordering ensures that events can be ordered as needed in the absence of any global timing mechanism. Synchronization of processes enforces the correct sequencing of instructions by all processes that both the program specifies and the programmer expects. Consistency of data ensures that all processes agree on the current state of the data.

These three problems are related. For example, if we can totally order events between all processes, then by implementing reading and writing shared variables as events, one can trivially ensure consistency through the ordering of those reads and writes.

## 3.2 Event Ordering

Event ordering can be classified as *strong* or *weak* event ordering. In strong event ordering, all events in a distributed system, whether they are local or shared events, are totally ordered. Lamport defined this ordering as *sequential consistency* [Lam79][4]. Dubois et al. describe weak ordering as a total ordering on global events, with local events occurring in some arbitrary interleaving with respect to each other [DSB88]. For instance, only accesses to shared memory in a multiprocessor system need to be totally ordered, while other events that occur locally on the processors can be interleaved in any order[5].

### 3.2.1 Strong Event Ordering

To address strong event ordering, Lamport defined the *happened before* relation in [Lam78]. The *happened before* relation, or $\rightarrow$, is defined as:

1. If events $a$ and $b$ are in the same process, and $a$ occurred before $b$, then $a \rightarrow b$.
2. If $a$ is the event that a message was sent from process $P1$ and $b$ is the event of receiving that message in $P2$, then $a \rightarrow b$.

With this definition, the definition of $a$ and $b$ occurring concurrently is when $a \nrightarrow b$ and $b \nrightarrow a$.

Lamport then described a logical clock $C$ as having the condition that for any events $a$, $b$, if $a \Rightarrow b$, then $C\langle a\rangle < C\langle b\rangle$. To order events totally, one can provide an arbitrary ordering on any two events which occur during the same logical time. However, this arbitrary ordering may lead to an unexpected total ordering, therefore Lamport described the *strong clock condition* as: for any events $a$, $b$ in the system, if $a \rightarrowtail b$ then $C\langle a\rangle < C\langle b\rangle$, where $\rightarrowtail$ is a relation that orders events in 'real' time. In order to achieve the strong clock condition, Lamport describes how to use real clocks, synchronized to within $\mu$ (the smallest message delay between two processes), so that events in the system can be totally ordered.

The importance of this work cannot be overstated: Lamport provided an algorithm for totally ordering events in a distributed system with the use of real clocks. He also provided an algorithm to synchronize clocks with the messages so that anomalous behavior (in the form of incorrectly

[4]Lamport actually defined this much more rigorously as, "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program." [Lam79]

[5]Weak ordering can be seen as the sequencing of shared events, where local events occur *between* the ticks of the clock of the shared events.

ordered events) does not occur. In essence, Lamport's algorithm allows us to transform an asynchronous system into a synchronous system.

### 3.2.2 Weak Event Ordering

Jefferson described Virtual Time in [Jef85] as a more natural ordering of events for a distributed system than strong event ordering. Jefferson's concept of virtual time is conceptually like Lamport's regular *clock condition*[Lam78], except that local events do not advance the time (in other words, the 1st condition of the *happened before* relation is not used to define virtual time). Using virtual time, Jefferson defines the Time Warp algorithm that is used to synchronize distributed systems.

Time warp works as follows:

1. All messages are timestamped according to the current virtual time locally.

2. Virtual time is advanced to the timestamp of the next queued message to be processed.

Time warp gets its name because a system can roll-back all events that have occurred whenever it receives a message from another system with a time earlier than its current virtual time. Thus, it can execute arbitrarily far into the future, even sending out events to other systems, but must roll-back time whenever it receives an event from an earlier period.

Jefferson also provides an algorithm for estimating the global virtual time. The global virtual time (GVT) is the earliest virtual time of any process currently executing in the system. The GVT can be used to determine how long old events must be tracked because a system will *never* be rolled back to a time earlier than the GVT. The GVT can also be used to determine when to commit to I/O or when an global snapshot of the system has completed.

Time warp seems to be particularly interesting to distributed discrete event simulation, where any system can execute arbitrarily into the future. The GVT is in some ways like lazy synchronization. In the system, each process may execute arbitrarily ahead, but as a whole the system is known to be synchronized at any point in time before the GVT.

## 3.3 Synchronization

Synchronization is the correct sequencing of processes to ensure mutually exclusive access to shared writable structures [DSB88]. Mutual exclusion is necessary for writable structures to prevent inconsistent state among the processes. We can divide synchronization mechanisms into hardware and software categories.

Because we are interested in distributed systems that only use message passing to between processes, we preclude the use of hardware mechanisms such as atomic load and stores, suspend-locks, and hardware compare-and-swap instructions [DSB88]. On a parallel system with shared memory, we might use semaphores or barriers. However, with only message passing, to provide synchronization, we have to use distributed mutual exclusion algorithms.

A number of distributed mutual exclusion algorithms are presented in [CDK01]. A simple ring-based algorithm logically arranges nodes in a ring and passes a token that allows a member of the system access to the shared writable structure. Other possibilities are multicast with logical clocks or distributed voting [CDK01].

## 3.4 Consistency

Consistency algorithms are designed to solve two fundamental issues: memory consistency or coherence, with respect to memory reads and writes [Lam79, DSB88, Lyn96, CDK01], and state consistency. Traditionally, consistency has focused on memory coherence. In a multi-processor system, each processor might have a private cache, which may lead to inconsistent states depending on the read and write policies of the memory system. In a fully distributed system, each system has its own local memory and must update others concerning state changes that are shared by all participants.

State consistency, on the other hand, requires that all processes agree on some state or value in the system. Lamport, Shostak, and Pease were some of the first to address this issue formally as distributed consensus in [LSP82]. They abstract the problem as the (now famous) Byzantine Generals Problem where a general needs to tell his lieutenants whether to attack or retreat from an upcoming battle. However, any of the participants may be a traitor, including the general. The goal then of a distributed consensus algorithm is to make sure that all non-traitors arrive at the same conclusion by the end of the algorithm.

Lamport et al. describe two algorithms: one that uses oral messages only, called $OM(m)$, and one that uses unforgeable signed messages, called $SM(m)$. Their results show that for $m$ traitors, $OM(m)$ requires $3m + 1$ generals and $m$ rounds of message passing to complete. $SM(m)$, on the other hand, requires only $m + 2$ generals, though still up to $m$ rounds of message passing. In addition, they show that distributed consensus is *impossible* in an asynchronous system.

#### 3.4.1 Group Membership

One particular problem in distributed systems involves determining the membership of the system at a given point in time. Chandra et al. note that this problem is similar to distributed consensus in that we want all nodes to agree on a value, which in this case is the membership list of the group[CHTCB96]. As with distributed consensus, they show that this problem is *impossible* in an asynchronous system, and tractable in a synchronous one.

### 3.5 Relevance to Research

Event ordering, synchronization, and consistency form the backbone of any distributed system. These inter-related components are necessary to design any distributed system. From this background work, we can draw several conclusions.

First, we know that a distributed architecture for MMOs requires a synchronous system in order to maintain consistency and to determine group membership. Lamport's work shows us how to transform an asynchronous system into a synchronous one. Second, the Byzantine Generals Problem maps directly to the problem of having a cheating player in a game that is maliciously trying to corrupt the system. Any algorithms that ignore the results from Lamport et al. will be subject to this kind of corruption. Third, the problem of interactive consistency, in which every process proposes a single value and the goal is for each process to agree on a vector of values, can be solved by running a distributed consensus algorithm from each process [CDK01]. Last, the concept of a global virtual time allows us to see that beyond an event horizon back in time, we can ensure that no changes will be necessary to the ordering of events.

With this background knowledge in hand, we now discuss research directly related to peer-to-peer architectures for massively-multiplayer games.

## 4 Distributed Communication

Our definition of a distributed system states that communication between nodes only occurs through message passing over a network. Therefore, in this section we discuss the network communication that is relevant to MMGs.

## 4.1 IP Multicast

While all current games use unicast for communication, and typically build on top of UDP, understanding multicast is important. Game players typically have less bandwidth for communication than game companies which host servers. Multicast could significantly reduce network traffic for distributed games. In addition, solving the problem of totally ordered, reliable multicast also solves the Byzantine General's Problem [CDK01].

The original concepts behind multicast were formalized by Deering and Cheriton in [DC90]. The goal of their design was to make multicast as similar to unicast as possible. To summarize their design, multicast uses groups that can be addressed by a single address, groups are open (knowledge of group membership is not necessary), hosts can join and leave at will, and hosts can belong to more than one group at a time. The first large scale multicast sessions began with the MBone, a set of tunnels over the Internet that connected to LANs that were multicast capable [Alm00].

The primary problem with the original multicast routing protocol (or DVMRP) is that it is efficient only if a large number of nodes in the system are participating, due to the amount of message passing that must occur to maintain multicast trees [DC90]. This led to research that focused on *sparse* protocols, where membership in the system only comprises a small percentage of the total number of nodes. Protocol Independent Multicast (PIM) [DEF$^+$94] and Core Based Trees [BFC93] are examples of sparse multicast protocols.

In building multicast trees, two primary methods have been investigated: shortest path trees and core based trees. Shortest path trees are rooted at the source with receivers at the leaves. Core based trees pick a central node in a minimum spanning tree that acts as the core. Sources unicast messages to the core which are then multicast to all group members. Core placement was studied extensively in [CZD, TR97].

Unfortunately PIM and CBT did not address several fundamental problems. First, neither considered inter-domain multicast, which is important since ISPs wish to do policy-based routing. The MASC/BGMP architecture addressed this by allowing intra-domain multicast to use whatever multicast protocol it deemed necessary and by building a bi-directional core-based tree rooted in the domain of the source of the multicast group [KRT$^+$98].

Second, some researchers felt that the multicast model was too general. Thus, single-source multicast (SSM) was created in the form of EXPRESS multicast [HC99]. In SSM, only one source can multicast messages to the group and an SPT is built rooted at the source. Holbrook and Cheriton argued that SSM was applicable to many multicast problems such multimedia broadcast [HC99]. Zappala and Fabbri added proxies to SSM in order to extend the SSM model to the general multicast model; in other words, the use of proxies allows any number of sources to send over the SSM tree and allows receivers to subscribe to the additional sources [ZF01].

## 4.2 Reliable Multicast

IP multicast assumes best-effort delivery and does not guarantee all packets will arrive at the receivers. Various researchers designed reliable multicast variants [HSC95, LP96, FJL$^+$97, LESZ98, KKT00, KHTK00]. These designs typically involve using some nodes to cache data that can then later be recovered. In addition, reliable multicast requires that receivers can actually detect lost packets. This implies that packets are either sequenced or that receivers expect packets periodically and can therefore detect missing packets.

In LBRM, a special logging server is added to the network and records all multicast packets [6] [HSC95]. RMTP is similar to LBRM in that certain receivers are designated to cache packets. In SRM, however, any receiver can respond to a lost packet message [LP96, LESZ98]. RMX uses a designated host at the LAN level and the multicast tree is built over the RMX receivers [KKT00]. Each RMX node limits the data flow according to the reception abilities of its receivers. Finally, Kasera et al. propose using multiple multicast groups for error recovery [KHTK00]. In their proposition, nodes subscribe to alternate channels to recover missing packets.

## 4.3 Congestion Controlled Multicast

In addition to reliability, scientists have also studied congestion control with multicast. Congestion control is vital to the sustainability of the Internet as Jacobson described in [Jac88]. In order to provide congestion control for multicast effectively, two problems need to be addressed. First, we have to make sure to only count one packet as lost when a single packet is lost over multiple paths. Second, we have to adjust the sending rate based on the congestion window for *all* receivers [BTK98, GS99]. This means that a source needs to track the congestion window for each receiver individually in order to keep performance at a maximum without overwhelming any single user.

[6] LBRM was designed in the context of distributed interactive simulations, where adding logging servers to the system was not considered a serious issue.

RLM addresses the congestion issue by assuming that we can divide a multicast stream into multiple layers [MJV96]. For instance, a media stream can be layered, with each successive stream adding higher quality data to the receiver. MTCP, on the other hand, uses specific receivers to collect data about their children in the multicast tree [RBR99]. This data is collected and collated up the tree until the source receives a report that represents the state of the multicast tree. The source then adjusts the sending rate based on the report.

### 4.3.1 Application Layer Multicast

While a large amount of research went into designing multicast, one questions why it has not been widely deployed. First, ISPs have been leery to enable new technology that has not been standardized (no one has agreed on the *best* form of multicast). Second, billing in the multicast model is a difficult problem. A single source can send just one packet that is duplicated thousands of times in another domain. Thus, one ISP would could use very little resources while generating a large volume of traffic in another.

By building multicast at the application layer, instead of at the network layer, researchers hope to make multicast available on the Internet. They also argue that the network layer is not an appropriate place for multicast when you consider that multicast requires state to be kept on each router (which is the part of the end-to-end principle [SRC84]). In order to design application layer multicast, an overlay over the physical topology must be built. These overlays are either unstructured, where the routing maintenance algorithms do not try to build some sort of structure on the overlay, or they are structured. However, from these overlays, application layer multicast builds a tree for distribution. Narada was one of the first application layer multicast protocols [CRZ00] and was shown to be capable of handling media streaming in [CRSZ01]. Narada works by building a mesh between members of a group and then building a tree for message distribution. SCRIBE, on the other hand, uses Pastry ([RD01]) to build a peer-to-peer network with the group members. Each multicast group maps to a particular node in the overlay which then acts as the root of the multicast tree [CDKR02]. We also note that several other application layer multicast protocols have been designed, and these two are simply representatives of the research in application layer multicast.

## 4.4 Secure Peer-to-Peer Communication

The final area in network communication research that we examine is that of secure communication in peer-to-peer networks. The issue we examine here is how can we build a peer-to-peer network such that a malicious node cannot disrupt the services it provides. Note that some research has focused on building secure P2P networks for the purpose of disseminating security updates ([LRP04]) or for ensuring that communications are available in a time of crisis ([KMR02]). These systems have a different purpose than general P2P communication in that they operate on a different time scale and are more *heavy-weight* than an MMG can afford.

A peer-to-peer network works at the application layer by building a virtual overlay on top of the physical Internet—a problem not unlike application layer multicast. Peers in the network follow some routing invariant that allows peers to route messages on the network based on the contents of an object. Typically, an object is hashed to produce an identifier which is then used as an address that can be routed to in the P2P network.

In [Wal02] and [CDG$^+$02], scientists address how to create secure routing in a P2P network. Castro et al. define secure routing as:

1. A malicious node cannot alter routing tables
2. A malicious node cannot affect locating a key by dropping or manipulating messages

Their solution can be summarized as using certified IDs, so that joining a P2P network becomes costly in terms of time or money; mapping IDs in the key space, and not basing it on network locality; using the first two assumptions and a detector to discover nodes that are violating these two assumptions.

In addition to these problems, Wallach adds that we can audit nodes periodically to ensure that they are correctly storing their prescribed values and routing messages properly [Wal02]. He also suggests signing content to ensure that it is not tampered with, but this is more of a problem with secure P2P storage and not routing. We discuss this problem in Section 5.

## 4.5 Key Results

In the previous sub-sections, we have covered a large amount of research related to networking in general. From these, we now summarize the key results.

Large-scale multicast was shown to be *possible* at all levels of the network, from the LAN [DC90] to intra-domain [BFC93, DEF$^+$94, Alm00, CZD, TR97] to inter-domain [KRT$^+$98]. However, as IP layer (or *native*) multicast has yet to be widely deployed, researchers have developed and demonstrated the feasibility of application layer multicast [CRZ00, CRSZ01, CDKR02]. While the major concern with application layer multicast has been the increased delay from overlay topologies that do not match the physical topology, the results from research have shown that delays are typically about twice that of native multicast.

The results from reliable multicast [HSC95, LP96, FJL$^+$97, LESZ98, KKT00, KHTK00] have shown that reliable multicast is possible, but that some form of caching is necessary for receivers to be able to recover lost packets. In multicast congestion control, the key result is that sources must track individual congestion windows for *all* receivers, though the congestion window information may be aggregated up towards the receiver [MJV96, GS99, BTK98, RBR99].

In peer-to-peer routing security, the main results appear to be that nodes should not be able to arbitrarily choose their own ID in the key space and that routing invariants should be verified at run-time [Wal02, CDKR02].

## 4.6 Relevance to Research

Past network research on multicast and peer-to-peer routing is important for a P2P architecture for MMGs. In order to reduce network communication, a P2P architecture would ideally use some form of multicast. Realistically, application layer multicast would be used due to the problem that native multicast is not widely deployed.

Games, like multicast recipients, are faced with the fact that receivers do not all have equal bandwidth. Therefore, an architecture should consider how to provide congestion control and reliability for all users. However, games do not require that *all* packets are transfered reliably. In addition, currently games do *not* use congestion control. History has shown us that congestion control is necessary for the sustainability of the Internet. Maintaining low latency and providing congestion control seem to be two conflicting interests, unfortunately.

Security in games is a major concern, and in particular, security flaws that allow players to *cheat*. We define cheating as any action taken by a player that gives them an unfair advantage over other players. If a peer-to-peer communication network is built, it will undoubtedly need to take cheating into consideration to prevent players from violating routing and message passing invariants.

## 5 Peer-to-Peer Storage

The second component of a peer-to-peer architecture for MMGs is storage. In a typical distributed system, each node has local storage, but data may be migrated to other nodes for performance reasons, or replicated for reliability. Having distributed storage creates the need for locks or mutual exclusion to ensure the consistency of data.

Perhaps the most notorious peer-to-peer storage system was Napster, which gained its notoriety from its users illegally sharing music files [nap]. Though users transferred files directly from each other, Napster was not entirely peer-to-peer. Instead, Napster used a central server that clients connected to in order to locate music stored by other peers.

Gnutella, on the other hand, is a completely peer-to-peer network [gnu] and serves the same purpose as Napster. The success of Napster, and other so called *unstructured* peer-to-peer networks, has prompted the creation of other peer-to-peer systems and motivated the research community to explore the viability of peer-to-peer storage. However, researchers feel that storage and lookup in unstructured peer-to-peer networks suffers from poor performance. In Gnutella, for example, the only way to locate data is to broadcast a request to the peers you know about–and even then, you are not guaranteed to discover the location of the data.

### 5.1 Structured Peer-to-Peer Storage

The two key functions required for peer-to-peer storage are simply the storage and retrieval of data. Similar to a file system, *structured* peer-to-peer systems take an object, or its name, and translate it into an address in the network[7]. This basic functionality can be seen simply as a *distributed hash table*; names are hashed into addresses so that they can be stored and retrieved from those addresses in the peer-to-peer network.

All DHTs work similarly. DHTs have a *key space*, which is specified as some number of bits. Objects that are to be placed in the network use some kind of cryptographically secure hashing function to return a hash value that fits in the key space. Nodes in the peer-to-peer network are also hashed into

[7]Of course a file system uses a table that maps names or file identifiers to locations on the disk, which is slightly different than simply hashing an object with a mathematical function to get an address

| **DHT** | **route length in hops** | **topologically sensitive** |
|---|---|---|
| Chord | $O(lgN)$ | |
| CAN | $O(N^{1/d})$ | ✓ |
| Pastry | $O(lgN)$ | ✓ |
| Tapestry | $O(lgN)$ | ✓ |

Table 2: Features of DHTs. $N$ is the number of nodes in the DHT and $d$ is the dimension chosen for CAN.

the key space. Thus, to determine where to store an item on the network, one simply hashes the object and routes to the node whose ID is closest to the object.

In order to route objects or requests to other nodes in the network, peers need to maintain routing tables. This maintenance is dependent on the DHT algorithms, but typically the DHT forms some sort of logical structure. Chord ([SMK+01]), for example, is a logical ring, while CAN ([RFH+01]) is a $d$-dimensional torus. The differences between route management in these DHTs are in routing guarantees (how long it takes to route to an item) and network locality (how well does the overlay map to the underlying network). Table 5.1 lists the differences.

DHTs, such as Chord [SMK+01], CAN [RFH+01], Pastry [RD01] and Tapestry [ZHS+04], are the cornerstone for any peer-to-peer storage system. At its core, Oceanstore uses a DHT, and builds global-scale storage services around it to provide consistency, reliability, and security [KBC+00]. Other services have also been built on top of DHTs, such as multicast (SCRIBE on top of Pastry, for example [CDKR02]), which use the DHT primarily for routing.

## 5.2 Key Results

The key results from research in DHTs and peer-to-peer storage have primarily come from simulation or small Planetlab *live tests* with a few hundred nodes. The simulations seem reasonable and demonstrate the ability for DHTs to scale well with the number of nodes in the system. This is due to the small amount of routing state (typically $O(lgn)$, where $n$ is the number of nodes) and minimizing the number of hops required to store and retrieve data.

Reliability of data in these systems comes typically from replication. Since peers may join and leave frequently, DHTs need to handle unexpected departures gracefully. The main problem with unexpected departures is that the data stored by the exiting node is lost. Any system planning on using a DHT for permanent state cannot expect data owners to constantly refresh the system (because, for instance, the owners may no longer be on the system). Thus, reliability is typically addressed through replication. Theoretically, if we have enough nodes and enough replication, the odds of data loss can be reduced to an arbitrarily small number.

Some DHTs are easier to replicate on than others. For example, Chord suggests modifying the basic Chord protocol so that data is stored on the $k$ closest nodes to the key [SMK$^+$01]. Because routing in Chord keeps neighbor lists, whenever a neighbor has crashed, the node can store its replicated data on an additional node. Other DHTs, such as CAN, usually depend on the data owner periodically replicating the data by using different hash functions. Unfortunately, this leaves data reliability in the hands of the data owner.

The reachability of data (or in some sense the reliability of routing) in DHTs is addressed nicely by CAN. While we can rely on the routing mechanisms of the DHT to ensure that data is reachable, DHTs that use proximity metrics for routing may all choose the same close broken link. CAN suggests a work around to this problem by using additional *realities*. A reality in CAN is a mapping of nodes to the key space. Any number of realities can be created by simply using additional hashing functions on the node IDs. Every object is then stored in each reality and querying for an object can occur simultaneously in each reality, thereby reducing the chance of failure and decreasing the number of hops to an object since we are now routing in more than one reality.

Another major result comes in the area of security in DHTs. Sit and Morris contributed a set of guidelines on security considerations when building a DHT [SM02]. These guidelines are:

- Define verifiable system invariants. For example, if a DHT has a specific routing invariant, make sure that it can be verified and then verify it at runtime.

- Allow a querier to observe the lookup progress. This prevents a malicious node from routing to an incorrect hop.

- Assign keys to nodes in a verifiable way. This prevents a node from choosing its ID such that it can *own* the data that it is interested in on the DHT.

- Do not allow server selection in routing. Pastry, for example, chooses the next hop based on a *proximity* metric, such as round-trip estimates.

Even if a querier can observe the lookup process, they cannot tell which route is closer to the malicious node. Therefore, a malicious node might purposely choose the farthest route.

- Cross-check routing tables using random queries (however, this can be prevented if a querier can observe the lookup progress).
- Avoid single points of responsibility. In other words, storing only a single item on the DHT without replication allows a single node to refuse to serve the data to other members in the system.

These guidelines address the major problems with current DHTs. For example, none of the DHTs previously listed allow a querier to observe the lookup progress. Some DHTs allow server selection (Pastry, Tapestry), and some DHTs do not assign keys in a verifiable way (CAN lets a node randomly assign its own identifier in the key space).

### 5.3 Relevance to Research

Designing a peer-to-peer storage system for MMGs has several unique characteristics. First, players expect that data stored in the MMG, such as their character information, *never* gets lost. Second, we need to prevent other players from tampering with the stored information.

This research addresses these problems by showing that DHTs are indeed scalable with the number of nodes in the system. Second, reliability can be increased through replication. Theoretically, we can replicate enough times so that the probability that data is lost is less than a database will be destroyed in a client/server architecture (assuming a sufficiently large number of nodes). Third, by following the guidelines presented in [SM02], we can address some of the problems of players tampering with data or trying to manipulate the storage or retrieval of data in the P2P storage network.

Ultimately, the success of using a DHT or similar structure for peer-to-peer storage in an MMG depends on its ability to quickly retrieve and store data, and its resilience in the face of high *churn*, or the rapid joining and leaving of nodes in the system.

## 6 Peer-to-Peer Computation

Peer-to-peer computation is a form of distributed computing over a peer-to-peer network. Scheduling occurs in a completely distributed manner and peers must both discover resources and verify results. A distinguishing feature of peer-to-peer computing is that peers are not set up for the *sole* purpose of supporting the system. Instead, peer-to-peer computing is trying to take advantage of the idle cycles of peers. The main problems in peer-to-peer computing are trying to discover idle cycles in a timely manner, process migration, and result verification.

The goal of peer-to-peer computing in an MMG is to schedule tasks that the server, in client/server architecture, normally handles. For instance, monsters in an MMG are controlled by an artificial intelligence (AI) process that determines how the monster reacts to players and how it fights during combat. The computation component of the architecture needs to schedule this AI process to be executed by players of the game.

## 6.1 Relevant Research

Recently, SETI@home ([ACK+02]) and the Stanford Folding project ([fol]) have received attention as large scale distributed computing projects. However, both of these projects have one common feature: the computation is easily subdivided into a countless number of problems and no communication between subproblems has to occur. Therefore, a client/server architecture for these distributed problems works well. Each client simply downloads a new problem set, solves it, and returns results to the server–all without needing to communicate with other clients. Obviously, distributed computing is much more difficult when peers must communicate.

Some distributed systems use a centralized scheduler. In Condor, for instance, a single system is used to locate machines with available cycles for users that are requesting additional cycles [LLM88], but scheduling is limited to a single LAN. Their work demonstrated that centralized scheduling could take advantage of the majority of idle cycles available. Butt et al. added the ability for schedulers in Condor to communicate with each other and schedule across domains [BZH03]. This *flock of Condors* uses Pastry for resource discovery. Recall that Pastry nodes keep a routing table that is topologically sensitive to the underlying network. In a flock of Condors, a scheduler advertises resources to all nodes in its Pastry routing table, allowing scheduling to occur on closer locations and ideally saving time.

SHARP (Secure, Highly Available Resource Peering) is a system to address trading and sharing resources [FCC+03]. SHARP places no restrictions on the underlying mechanisms for communication, but shows how digital signatures can be used in order to guarantee resources. Their resource sharing mechanism is conceptually simple. A node that has resources signs a message stating that it has granted a resource to another system for some given time. The other system then uses this receipt to redeem the resources. The interesting property of SHARP is that resources can be traded, divided, and duplicated (oversubscribed). The digital signatures provide evidence for repudiation if the resource provider fails to fulfill its contract–though SHARP does not suggest how repudiation should occur.

However, Butt et al. in [BFH+03] suggest that cycles are too dynamic to advertise because the information will be too stale by the time cycle requesters receive the information. Instead, they propose using a cycle discovery protocol. In their system, they simply forward requests to nodes and nodes which are available schedule the process for execution.

Lo et al. in [LZZ+04] describe four classes computation that are candidates for peer-to-peer computing in their system called *Cluster Computing on the Fly* (CCOF): infinite workpile, workpile with deadlines, tree-based search algorithms, and point-of-presence applications. They envision a number of community based overlays for the purpose of scheduling. Each community schedules locally with a local coordinator and coordinators use CAN ([RFH+01]) to schedule processes globally.

## 6.2 Key Results

Condor showed us that remote scheduling was effective in taking advantage of idle cycles, but uses centralized scheduling [LLM88]. While a flock of Condors allows remote scheduling over a WAN, each Condor system must be manually set up and managed.

Fu et al. discussed the need for over-subscription of resources in order to maintain a high load in the system [FCC+03]. They argue that systems should oversubscribe because any system requesting resources might crash or leave, thereby leaving cycles unused.

The question of whether to advertise idle cycles or to discover them is still an open problem. [BFH+03] suggested that resource discovery was probably the best method since advertisement information was very likely too stale to use. Lo et al. stated that they had found rendezvous points (a form of advertising) worked best with respect to heavy workloads [LZZ+04].

## 6.3 Relevance to Research

Comparing the requirements of P2P computation in an MMG with the given research is somewhat difficult. In an MMG, a process would interact in real-time with some set of players in the game. In the discussed research, we are trying to discover idle cycles to schedule long-running processes and often to take advantage of as many idle machines as possible.

Scheduling techniques and resource discovery have obvious applications in designing a P2P MMG scheduler. Perhaps the work in [LZZ+04] is most relevant to our research. Their point-of-presence application is similar in requirement (strategically located nodes in the P2P network that are used to execute processes) to our needs. The scheduler, in this instance, would need to locate process execution strategically to the needs of the game.

## 7 Game Research

Research in interactive games has gained popularity recently. However, some research, in particular that related to distributed interactive simulations (DIS), has had an influence on previously developed games. In this section, I first discuss the relevance of DIS [8] followed by research which has specifically focused on games.

### 7.1 Distributed Interactive Simulations

Distributed Interactive Simulation (DIS) was born from a DARPA initiative to build large scale, distributed, interactive simulations for creating theater of war scenarios. Over the years, the goal was to create larger and larger scenarios with hundreds of thousands of units (though not all necessarily interactive). The result of this initiative was a workshop dedicated to DIS[9] and a slew of standards located in difficult to access standards documents [dis95a, dis95b][10].

All DISs run to date have been designed for large supercomputers, harnessing hundreds of processors connected by high-speed links. More recently, grid technology, such as Globus [glo], has been used to manage the connection and process execution on the connected supercomputers.

Part of the work that came out of the design of DIS was that by Van Hook et al. in [HCNF94]. In this paper, two important algorithms were developed. The first of these is grid-based filtering, where each node calculates its area of interest based on dividing the virtual world into a grid. This area is then

[8] One might also include networked virtual environments, or NVEs, which were born from the DIS initiative. Basically NVEs are non-military related DIS applications. MMGs could be considered a subset of these applications.

[9] The last workshop was held in 1996.

[10] The goal of the DIS workshop was to develop standards for intercommunication between different simulation models.

relayed to an application gateway (or AG), which appropriately filters all updates to a local LAN based on the union of the areas sent to it by local nodes.

The second algorithm is called *rethresholding*. In DIS, entities are dead-reckoned, meaning that their position was predicted, by other entities, between network updates. Each entity has a threshold, or margin of error, that it allows other entities to experience before it is forced to send out a new update. This meant that slowly moving entities, or those that were moving in an accurately predictable manner, sent fewer updates, thereby reducing traffic significantly.

These two algorithms had a huge influence on networking design for games. Games use both techniques to reduce traffic, albeit at the expense of making it easier for players to cheat through the network protocols .

LBRM, which is log-based receiver-reliable multicast, is designed specifically for DIS [HSC95]. In the context of DIS, separate loggers for logging all multicast packets are appropriate. Further, LBRM also introduces a technique where entities send a heartbeat update and then back off exponentially, unless a change to the entity occurs. This technique further reduces the messaging requirements in DIS.

Léty and Turletti describe a multicast based communication architecture in [LT99]. They divide the virtual world into cells, which can be dynamically resized, split, and joined depending on the number of multicast addresses available and the density of players in a cell. Their work introduces a *satisfaction* metric, which is a measure of the amount of relevant traffic versus irrelevant traffic. They argue that a user is more satisfied when they receive a higher ratio of relevant traffic to irrelevant traffic. Through simulation, they show that static splitting of the virtual environment results in a significantly lower satisfaction than when dynamic cell adjustment was used.

## 7.2 Protocols

Designing game protocols requires an approach different than just trying to achieve the lowest latency and fastest speed. In particular, the problem of cheating by manipulating packets plagues modern games. The main cheats that occur through packet manipulation are the *fixed-delay cheat*, where a fixed amount of delay is added to each packet; the *timestamp cheat*, where timestamps are changed on packets to alter when events actually occurred; the *suppressed update cheat*, where updates are purposely not sent to other players; and the *inconsistency cheat*[11], where different updates from one player are sent to different players. These cheats can be prevented by designing the protocol appropriately.

Diot, Gautier and Kurose described the first protocol for distributed games in [DG99] and [GDK99] 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 to maintain state consistency among players. The MiMaze protocol uses multicast to exchange packets between players, resulting in a low latency.

At the other end of the spectrum, Baughman and Levine presented the *lockstep* protocol to address the problem of cheating with dead-reckoning [BL01]. Lockstep works by dividing game time into rounds, during which players reliably send a cryptographic hash of their move to all other players. Once every player has received the hash, the plain-text move is then reliably sent to all players. The game proceeds to the next round only after the hash and the move have been received by all players.

Lockstep is a major advance in distributed protocols because it is provably secure against several cheats. The drawback of lockstep is that its *playout latency*, which is the time from when an update is sent out to when the update can be displayed to other players, is unacceptably high for real-time games. The use of reliable transport bounds its playout latency at three times the maximum delay between any two players, assuming no packets are lost.

To mitigate the playout latency, *asynchronous synchronization* (AS) was developed using lockstep [BL01]. AS uses spheres of influence that are dilated each round and allows players that are virtually far apart to proceed asynchronously in round progression. Once the spheres intersect, actions must be resolved. This technique can be used with any protocol as a method of reducing communication, but it requires that every player know and keep track of every other player. Unfortunately, using AS subjects lockstep to collusion cheats.

Cronin et al. designed the sliding pipeline protocol[CFJ03] in order to improve the lockstep protocol. An adaptive pipeline is added 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 depth is designed to grow with the maximum latency between players so that *jitter*, or inter-packet arrival time, is reduced.

[11]This cheat is the same problem as Lamport's Byzantine Agreement problem.[LSP82]

While sliding pipeline reduces jitter and dead-reckoning, it still has the same playout latency as lockstep. In terms of security, the protocol prevents the lookahead cheat, but allows a player to use the *suppressed update* cheat. Even with an adaptive pipeline to help detect this cheat, it can falsely label someone with an increased delay as a cheater. Furthermore, a cheater can use the suppressed update every other round and not be detected.

In the game industry, very few networked games are fully distributed. One notable exception is Age of Empires (AoE) [Stu], in which games are synchronized across clients and peer-to-peer communication is used [BT01]. 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 all but the inconsistency cheat (because players periodically exchange hashes of the game state with other players to detect inconsistencies).

## 7.3 Architectures

Some researchers have recently suggested subscriber/publisher models for data distribution with distributed games. A subscriber/publisher model works by creating a mechanism whereby the game creates different types of content to be published in the game. Interested parties *subscribe* to the publisher of the information they desire. Ideally only the messages that a player desires are sent to them while they publish messages only to other players that are interested. The primary difficulty with the publisher/subscriber model is how quick new material for publishing can be advertised on the network, and how fast a player can get the information he or she is interested in.

Bharambe et al. describe their system, Mercury, as a scalable publish-subscribe system for distributed Internet games [BRS02]. Unlike Léty and Turletti's work in which subscription channels are based only on game position, Mercury allows channels to be of any subject. Mercury uses a subscription language, which is a subset of a relational database query language, such as SQL, and uses rendezvous points (RPs) which gather publications. Publishers send their data to the RPs and subscribers send their queries to the RPs to receive the publications. Bharambe et al. borrow heavily from the ideas in Chord [SMK$^+$01] and organize the RPs in a logical ring, using similar routing mechanisms, with the primary difference being that their subscription language allows them to match several RPs which contain the requested subscription while Chord can only perform exact matches on queries. Unfortunately, the results from the research on Mercury show that it is unable to meet the necessary performance requirements of multi-player online games.

Knutsson et al. also designed a publish/subscribe system [KLXH04] using Pastry [RD01] and Scribe [CDKR02]. 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.

Butterfly.net [But03] is a grid-based solution for developing MMOGs. Using the Globus toolkit [glo], developers are able to lease computer time and space from Butterfly.net to run their MMOGs from. Terazona [Zon02], 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.

## 7.4 Key Results

Grid-based filtering and rethresholding are both extremely important ideas that resulted from DIS [HCNF94]. Grid-based filtering spawned other techniques for *interest management*, where a node only receives updates about the entities that it is interested in. Rethresholding also spawned research into other techniques for dead-reckoning.

MiMaze demonstrated that bucket synchronization, dead-reckoning and multicast helped provide a communication protocol that was usable by interactive games [DG99]. Lockstep, on the other hand, demonstrated that network-level cheats could be prevented by the protocol itself [BL01].

Research in publish/subscribe systems presented a new concept in distributed architectures for MMGs [BRS02, KLXH04]. Whether these architectures are truly feasible has yet to be demonstrated.

## 7.5 Relevance to Research Area

Obviously, the research discussed in this section is all relevant to P2P architectures for MMGs.

## 8 Conclusions and Future Work

The first question that one must answer when considering an area for research is whether the area is a viable one for research in the first place. Looking at the list of papers in Section 7 shows that very little work has been done directly concerning my topic. In fact, the listed papers represent the main scientific work that contributes to game research in general[12].

Any architecture for massively-multiplayer games touches on just about every area in computer science, including networking, distributed computing, and theory. By dividing the architecture into three main components (communication, storage, and computation), we can begin to see the numerous research questions that arise in each component.

Application-layer multicast and peer-to-peer routing generate ideas about how to design the communication component. Secure DHTs help us see that P2P storage is possible. P2P computing help demonstrate some of the key ideas necessary to solving the problem of scheduling in a distributed MMG.

What is obvious about this area is that it is too large for any single person to complete in a finite amount of time. As such, I plan to focus my research on the communication component, with the possibility of exploring the computation component. A major open problem that none of the research in network communication addresses is how to handle the cheating while keeping a game running at an interactive rate. Furthermore, a fully distributed architecture is *useless* for games if it allows players to cheat freely and easily. I also argue that cheating must be addressed at the architectural level, and that doing so fundamentally changes the design of your system. However, once we address cheating, how can we build a communication system that scales with the number of players? Last, none of the game communication protocols address *playing fair* with TCP. In essence, they do not use congestion control, which past history has shown us to be a fatal problem on the Internet!

To conclude, I believe these three areas (communication, storage, and computation) are the three primary areas that a fully distributed, peer-to-peer architecture for massively-multiplayer games. Research from these areas will help in the design of this architecture. I also believe that each of these areas has a number of open, interesting research problems so that an exploration in these areas will provide a significant body of research for a Ph.D. dissertation.

[12]Aside from computer graphics, that is, under which *real-time rendering* typically covers computer graphics for games.

## 9 Acknowledgements

I would like to thank my committee for their help and suggestions on this work: Daniel Zappala (chair), Virginia Lo, and Jun Li.

## References

[ACK$^+$02] D. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45:56–61, 2002.

[Alm00] Kevin Almeroth. The Evolution of Multicast: From the MBone to Inter-Domain Multicast to Internet2 Deployment. *IEEE Network*, 14:10–20, Jan/Feb 2000.

[BFC93] Tony Ballardie, Paul Francis, and Jon Crowcroft. Core based trees (CBT). In *Conference proceedings on Communications architectures, protocols and applications*, pages 85–95. ACM Press, 1993.

[BFH$^+$03] A. Butt, X. Fang, Y. Hu, S. Midkiff, and J. Vitek. An Open Peer-to-Peer Infrastructure for Cycle Sharing. Poster in ACM Symposium on Operating System Principles, October 2003.

[BL01] Nathaniel E. Baughman and Brian Neil Levine. Cheat-proof Playout for Centralized and Distributed Online Games. In *INFOCOM*, pages 104–113, 2001.

[BRS02] A. R. Bharambe, S. Rao, and S. Seshan. Mercury: A Scalable Publish-Subscribe System for Internet Games. In *Proceedings of the First Workshop on Network and System Support for Games.*, April 2002.

[BT01] P. Bettner and M. Terrano. 1500 archers on a 28.8: Network programming in the Age of Empires and beyond. In *GDC 2001*, March 2001.

[BTK98] S. Bhattacharyya, D. Towsley, and J. Kurose. The Loss Path Multiplicity Problem for Multicast Congestion Control. In *Proceedings of INFOCOM*, pages 856–863, 1998.

[But03] Butterfly.net, Inc. The Butterfly Grid: A distributed platform for online games. http://www.butterfly.net/platform/, 2003.

[BZH03] A. Butt, R. Zhang, and Y. Hu. A Self-Organizing Flock of Condors. In *Proceedings of the Super Computing Conference*, 2003.

[CDG$^+$02] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. In *OSDI*, December 2002.

[CDK01] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concepts and Design, 3rd Edition*. Addison-Wesley, 2001.

[CDKR02] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communications (JSAC)*, 2002.

[CFJ03] Eric Cronin, Burton Filstrup, and Sugih Jamin. Cheat-Proofing Dead Reckoned Multiplayer Games. In *International Conference on Application and Development of Computer Games*, January 2003.

[CHTCB96] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 322–330. ACM Press, 1996.

[CRSZ01] Yang Chu, Sanjay Rao, Srinivasan Seshan, and Hui Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 55–67. ACM Press, 2001.

[CRZ00] Yang Chu, Sanjay G. Rao, and Hui Zhang. A Case for End System Multicast. In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–12. ACM Press, 2000.

[CZD] Kenneth L. Calvert, Ellen Witte Zegura, and Michael J. Donahoo. Core Selection Methods for Multicast Routing. In *IEEE ICCCN*.

[DC90] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, 1990.

[DEF$^+$94] Stephen Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei. An architecture for wide-area multicast routing. In *Proceedings of the conference on Communications architectures, protocols and applications (ACM SIGCOMM*, pages 126–135. ACM Press, 1994.

[DG99] C. Diot and L. Gautier. A Distributed Architecture for Multiplayer Interactive Applications on the Internet. *IEEE Networks magazine*, 13(4), July/August 1999.

[dis95a] IEEE Standard for Distributed Interactive Simulation – Application Protocols (IEEE STD 1278.1-1995). IEEE Computer Society, 1995.

[dis95b] IEEE Standard for Distributed Interactive Simulation – Communication Services and Profiles (IEEE Std 1278.2-1995). IEEE Computer Society, 1995.

[DSB88] Michel Dubois, Christoph Scheurich, and Fayé A. Briggs. Synchronization, Coherence, and Event Ordering in Multiprocessors. *Computer*, 21(2):9–21, 1988.

[FCC$^+$03] Yun Fu, Jeffrey Chase, Brent Chun, Stephen Schwab, and Amin Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Symposium on Operating System Principles*, October 2003.

[FJL$^+$97] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for lightweight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5(6):784–803, 1997.

[fol] Folding@home distributed computing. http://folding.stanford.edu.

[GDK99] L. Gautier, C. Diot, and J. Kurose. End-to-end transmission control mechanisms for multiparty interactive applications on the internet. In *IEEE Infocom*, 1999.

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

[gnu] Gnutella. http://www.gnutella.com.

[GS99] S. Jamaloddin Golestani and Krishan K. Sabnani. Fundamental Observations on Multicast Congestion Control in the Internet. In *INFOCOM*, pages 990–1000, 1999.

[HC99] Hugh W. Holbrook and David R. Cheriton. IP multicast channels: EXPRESS support for large-scale single-source applications. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 65–78. ACM Press, 1999.

[HCNF94] Daniel J. Van Hook, James O. Calvin, Michael K. Newton, and David A. Fusco. An Approach to DIS Scaleability. In *11th DIS Workshop*, September 1994.

[HSC95] Hugh W. Holbrook, Sandeep K. Singhal, and David R. Cheriton. Log-based receiver-reliable multicast for distributed interactive simulation. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 328–341. ACM Press, 1995.

[Jac88] V. Jacobson. Congestion avoidance and control. In *Symposium proceedings on Communications architectures and protocols*, pages 314–329. ACM Press, 1988.

[Jef85] David R. Jefferson. Virtual time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[KBC$^+$00] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201. ACM Press, 2000.

[KHTK00] Sneha Kumar Kasera, Gísli Hjálmtýsson, Donald F. Towsley, and James F. Kurose. Scalable reliable multicast using multiple multicast channels. *IEEE/ACM Trans. Netw.*, 8(3):294–310, 2000.

[KKT00] S. K. Kasera, J. Kurose, and D. Towsley. RMX: Reliable Multicast for Heterogeneous Networks. In *IEEE Infocom*, 2000.

[KLXH04] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-Peer Support for Massively Multiplayer Games. In *IEEE Infocom*, March 2004.

[KMR02] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proceedings of ACM SIGCOMM*, 2002.

[KRT$^+$98] Satish Kumar, Pavlin Radoslavov, David Thaler, Cengiz Alaettinoğlu, Deborah Estrin, and Mark Handley. The MASC/BGMP architecture for inter-domain multicast routing. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 93–104. ACM Press, 1998.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam79] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Computers*, pages 690–691, September 1979.

[LESZ98] Ching-Gung Liu, Deborah Estrin, Scott Shenker, and Lixia Zhang. Local error recovery in SRM: comparison of two approaches. *IEEE/ACM Trans. Netw.*, 6(6):686–699, 1998.

[LLM88] M. Litzkow, M. Livny, and M. W. Mutka. Condor - A Hunter of Idle Workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, June 1988.

[LP96] J. C. Lin and S. Paul. RMTP: A Reliable Multicast Transport Protocol. In *IEEE Infocom*, pages 1414–1424, March 1996.

[LRP04] J. Li, P. Reiher, and G. Popek. Resilient Self-Organizing Overlay Networks for Security Update Delivery. *IEEE Journal on Selected Areas in Communication, special issue on Service Overlay Networks*, January 2004.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[LT99] Emmanual Léty and Thierry Turletti. Issues in Designing a Communication Architecture for Large-Scale Virtual Environments. In *Networked Group Communications*, pages 54–71, 1999.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[LZZ$^+$04] Virginia Lo, Daniel Zappala, Dayi Zhou, Yuhong Liu, and Shanyu Zhao. Cluster Computing on the Fly: P2P Scheduling of Idle Cycles on the Internet. In *International Workshop on Peer-to-Peer Systems*, 2004.

[MJV96] Steven McCanne, Van Jacobson, and Martin Vetterli. Receiver-driven layered multicast. In *Conference proceedings on Applications, technologies, architectures, and protocols for computer communications*, pages 117–130. ACM Press, 1996.

[nap] Napster. http://www.napster.com.

[RBR99] Injong Rhee, Nallathambi Ballaguru, and George Rouskas. MTCP: Scalable TCP-like Congestion Control for Reliable Multicast. In *INFOCOM*, March 1999.

[RD01] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*, pages 329–350. Springer-Verlag, 2001.

[RFH$^+$01] 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.

[SM02] Emil Sit and Robert Morris. Security Considerations for Peer-to-Peer Distributed Hash Tables. In *Peer-to-Peer Systems: First International Workshop (IPTPS 2002)*, volume 608 of *Lecture Notes in Computer Science*, pages 261–269. Springer-Verlag Heidelberg, March 2002.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[SRC84] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, nov 1984.

[Stu] Ensemble Studios. Age of Empires. http://www.ensemblestudios.com/aoe.htm.

[TR97] David Thaler and Chinya V. Ravishankar. Distributed Center-Location Algorithms. *IEEE Journal on Selected Areas in Communications*, 15(3):291–303, 1997.

[Wal02] Dan S. Wallach. A Survey of Peer-to-Peer Security Issues. In *International Symposium on Software Security*, November 2002.

[ZF01] Daniel Zappala and Aaron Fabbri. Using SSM Proxies to Provide Efficient Multiple-Source Multicast Delivery. In *Proceedings from IEEE Global Internet Symposium (Globecom)*, November 2001.

[ZHS$^+$04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: A Resilient Global-SCale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Januray 2004.

[Zon02] Zona Inc. Terazona: Zona application framework whitepaper. www.zona.net/whitepaper/Zonawhitepaper.pdf, 2002.