Using N-Trees for Scalable Event Ordering in Peer-to-Peer Games[*]

Chris GauthierDickey [†] and Virginia Lo
Department of Computer Science
University of Oregon
1202 University of Oregon
Eugene, OR 97403-1202
{chrisg | lo}@cs.uoregon.edu

Daniel Zappala
Computer Science Department
Brigham Young University
Provo, UT 84602-6576
zappala@cs.byu.edu

ABSTRACT

We are concerned with the fundamental problem of event ordering in multiplayer peer-to-peer games. Event ordering, even without faults, requires all-to-all message passing with at least two rounds of communication [12]. Multiplayer games add real-time constraints to this scenario. To meet this challenge, we develop an event scoping mechanism that uses N-Trees for event propagation. Unlike traditional application-layer multicast, N-Trees organize peers by their application-level scope of interest, instead of by their delay-based shortest-path tree. This organization allows peers which are close by in the virtual world to order events without needing to communicate with other peers that are farther away. We show the asymptotic analysis of N-Trees indicates that they will perform well for scalable peer-to-peer event ordering. We also provide an analysis of N-Trees in comparison to other distributed architectures for peer-to-peer games.

Categories and Subject Descriptors

C.2.4 [**Computer Systems Organization**]: Computer Communication Networks—*Distributed Systems*

General Terms

Algorithms, Design, Theory

Keywords

n-tree, peer-to-peer, multiplayer, games, event ordering

1. INTRODUCTION

Event ordering is a fundamental problem in multiplayer peer-to-peer games. Without event ordering, players will quickly generate inconsistent views of the virtual world as events are delayed over the network. Traditional event ordering protocols, such as the Paxos algorithm which only tolerates stopping failures [15], can require up to five rounds of communication. In fact, even without faults, distributed event ordering requires at least $\Omega(n^2)$ messages per event, and two rounds of communication between all peers [12].

Multiplayer games present the additional challenge that events must be ordered within a small time limit in order to maintain interactivity. Though some games do not need low latency, such as a game of online chess, the majority of multiplayer games on the Internet have latency requirements below 500ms [20]. Taking the latency requirements of games in conjunction with the best case scenario for distributed event ordering means that the naïve approach of all-to-all communication for event ordering will not scale with the number of peers in the game.

Previous work in this area has looked at some form of application layer multicast (ALM), such as the publish/subscribe system of Mercury [3] or using multicast with regions [13]. These systems organize their dissemination paths based on the proximity of peers on the network, instead of the proximity of peers in the virtual world. The result is that peers which are close in the virtual world but far apart in the network will have to route messages over the underlying ALM structure to order events. Even though these systems guarantee $O(\lg n)$ application layer hops with n peers, even a few additional hops can add considerable delay, making interactive event ordering infeasible.

To address this problem, we propose using N-Trees and *scoped events*. N-Trees are a generalization of the *octree*, from computer graphics [8], that recursively subdivide an N-dimensional space. For event ordering, the N-Tree subdivides the *application state space*, which is the application domain that all peers share and modify. In a peer-to-peer game, the application state space *is* the virtual world; hence N-Trees can be applied naturally to this state space.

Peers are organized into an N-Tree by their scope of interest in the virtual world. The N-Tree allows peers to efficiently move to new locations, discover other peers that are in close proximity, and propagate events to other parts of the virtual world. By joining the N-Tree, peers know which peers are close by, and can therefore order events directly with those peers without having to exchange events with other, further peers.

Peers generate scoped events, which are events that are labelled with a tuple representing the scope of impact that the event has in the virtual world. When a peer generates an event, it uses the N-Tree to propagate the event to other peers within the event's scope. If the scope of the event is contained within the leaf node, then

[*]This work sponsored in part by NSF ANI 9977524.
[†]Supported by an NSF Graduate Research Fellowship.

the peer only has to exchange events directly with the other peers in the leaf. Using scoped events, we loosen the requirements that all peers must exchange message to totally order events. In other words, scoped events and the N-Tree allow us to reduce communication between peers while quickly propagating events that need to reach far away peers.

This paper has three important contributions. First, we have created a novel event ordering system that organizes peers by their *interest* in the application domain (unlike traditional ALM), allowing peers to order events quickly. We describe operations for joining and leaving the N-Tree. Second, we provide an asymptotic analysis of the complexity of joining, leaving, moving between nodes, subdividing nodes, and collapsing nodes. We also compare it to existing simulations. Last, we demonstrate how N-Trees and event scoping can be used for scalable peer-to-peer games.

2. MOTIVATION AND BACKGROUND

We are motivated by the fact that neither strong or weak event ordering are scalable when the number of nodes and non-local events increase beyond a small value. In strong event ordering, all events in the system are totally ordered [14]. In weak event ordering, only non-local events are totally ordered [7]. For example, in a peer-to-peer game, with strong event ordering, a command to 'look at player inventory' would be ordered with every other command by every player in the game. This stringent event ordering is useful for debugging situations to know exactly when one event occurred. In a game with weak event ordering, only events that change the virtual world are totally ordered. Thus, the inventory command would not be ordered (and only executed locally), while a movement command would be ordered.

Unfortunately, even in the best cast scenario, any event ordering algorithm requires at least $\Omega(n^2)$ messages and at least 2 rounds of communication before consensus can be reached [12]. The Paxos algorithm [15], for example, requires up to five rounds of communication to agree on a value, and a majority of nodes must have reliable communication with the leader.

We are interested in N-Trees because they are optimized for event propagation so that the fewest number of nodes are contacted for event ordering. Existing peer-to-peer structures, such as Gnutella, Chord, CAN and Pastry [10, 17, 19, 21], are optimized for fast storage and retrieval of data using two functions, *insert(key, value)* and *lookup(key)*. DHTs do provide the kind of mapping we need for the application space – we might map the application space to the key space of a DHT, for example. However, propagating events to neighbors and relocating to new places in the DHT would be too slow for scalable event ordering.

N-Trees have a similar advantage when compared to using application layer multicast (ALM) such as Bayeux, CAN-multicast, Narada, NICE, and Scribe [1, 4, 6, 18, 23]. Multicast is optimized to send as few messages as possible in one-to-many (or many-to-many) communication. However, multicast trees are built based on end-to-end delays between hosts, not on the interests of group members. While multicast reduces messaging, all messages are sent to all group members and filtering messages to relevant members is not a trivial task. Further, in a peer-to-peer game, every member sends messages, requiring a shared tree (such as HMTP [22]), or n source-specific trees. On the other hand, ALM could be used in conjunction with N-Trees between leaders of each node in the tree, and even between nodes in each small group, to reduce messaging.

Mercury provides a publish/subscribe architecture to support massively multiplayer online games (MMOs) [3]. In this architecture, a multi-attribute query language is used so that game state is published in Mercury and players receive a small subset of state changes according to their subscriptions; thus, the total state that each player receives is reduced by Mercury. The primary difference is that Mercury uses a DHT to store and route information while N-Trees build a domain-specific tree. Joining an N-Tree at a particular node determines what data a peer receives, whereas with Mercury the subscription determines what data is routed to a peer.

Knutsson et al. statically divide a world into *regions* and use Scribe to multicast messages to all members of a region [13]. Peers join the multicast tree of the region they are interested in, but this also means that peers will have to forward unrelated traffic. This occurs because peers are members of the underlying Pastry DHT and will likely be on the path of other multicast trees (since a source-specific tree is built for each region). Further, the static division of the virtual world limits the scalability of the system if some areas suddenly become popular. N-Trees in this case could be used in each region to increase the scalability of their work.

3. DEFINITIONS

3.1 Peer-to-Peer Game

The term *peer-to-peer game* is defined as a multiplayer, networked game (running over the Internet) where each player, or peer, shares an equal responsibility in running the game. We are interested in large scale peer-to-peer games, since games with few players can simply run traditional event-ordering protocols, where large scale might mean several thousand players. Note that using a "back of the envelope" calculation, a peer-to-peer game of only 64 players would overwhelm a typical broadband connection.[1]

Throughout this paper and without a loss of generality, we assume a simple game that takes place on a 2-dimensional plane, where each peer represents a single player in the game, each of whom roams the virtual world in search of treasure. Players have the ability to cast a single spell, which creates a snow storm, to help escape the population of monsters (undoubtedly evil) that want to eat the players. This simple model represents the basic interactions of any multiplayer game. Players can interact with objects (taking a treasure) or each other (giving each other treasure), and objects can interact with players (monsters attacking players) or with each other (snow storm snowing over the monsters). Last, the snow storm represents an event that affects a portion of the virtual world, instead of just a single point such as taking a treasure would.

3.2 Application State Space

The *application state space* is defined as the domain of a distributed application that is hierarchically structured such that a domain has 2^N sub-domains, each sub-domain shares this domain as a sole parent, and each sub-domain can be recursively subdivided ad infinitum. We place the restriction that each division of a sub-domain has 2^N children solely for the purpose of mapping the application state space onto an N-Tree.

In a peer-to-peer game, the application state space is typically the virtual world and its contents, including the players of the game. Events occurring in the virtual world include player movement, taking an item in the virtual world, and larger events such

[1]Assuming 10 updates/second, 100 byte updates [5], unicast communication between players, each player would require 500Kbps upload and download speeds.

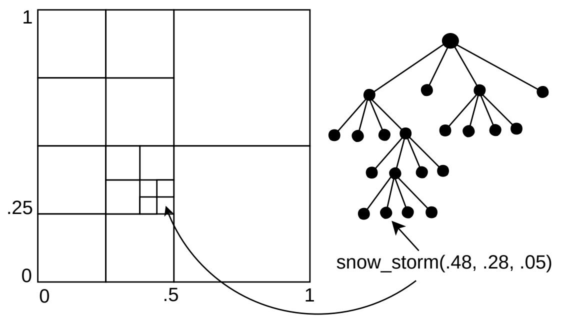

Figure 1: Cartesian application space and quadtree representation. In this example, a snow storm occurs at (.48, .28) with a radius of .05. The event occurs in the shown node and may be propagated to other nodes if the radius of the event exceeds the boundary of the node it occurs in.

as a snow storm in an area. The state space in this example includes two or three *dimensions*, or variables, which correspond to the actual dimensions in the virtual world.

3.3 Scoped Event

A *scoped event* is a tuple consisting of an action, the location of the action, and a function representing the scope of impact of that action in the application state space. The location of the event indicates where the event originally occurs while the scope indicates how broadly the event can impact the application state space.

For example, taking a treasure in a game creates an event that occupies a single point in space, which is the location of the treasure. A snow storm, on the other hand, would have a scope equal to the maximum radius of the storm. In this case, the scope is a circle centered at the location of the action, with a given radius. However, we do not limit scopes by circles, and instead use a function which defines the scope for a given event.

3.4 N-Tree

We define an N-Tree as a generalization of the *octree* with N dimensions instead of only three that an octree has, or two that a *quadtree* has [8].

An N-Tree is defined inductively as follows:

- **Base case**: a Leaf
- **Inductive case**: a Node with 2^N children, each an N-Tree

By this definition, nodes in the N-Tree are either parent nodes (simply called nodes), or leaves (a node with no children). An elected subset of peers manages the nodes and leaves of the N-Tree on a one-to-one basis. All peers reside at leaves and totally order events with other peers in the same leaf. We call the peers at a leaf a *group*. Events that have a scope that intersects the boundary of a leaf are propagated to other leaves in the N-Tree by being passed up through leaders until a higher level node is found that encompasses the entire scope of the event–at which point it is propagated back down the tree to appropriate nodes and leaves.

The purpose of the N-Tree is to divide the application state space into a series of nested game scopes so that peers can be organized by their interest in the game. This has the result of reducing the communication requirements between peers because peers that are far apart in the virtual world do not have to exchange events with each other. Each successive level below the root node represents a further subdivision of the virtual world by 2^N.

Peers, like events, have a location, and a function that determines their scope of interest in the virtual world. All peers reside at leaves in the N-Tree. The peer's scope of interest determines if it should be a member of more than one leaf, for example when it is on the boundary of two leaves. In this case, it would join both leaves in the N-Tree.

In Figure 1, we illustrate a 2-tree, or quadtree. In a quadtree, the state space is a Cartesian square that is subdivided as necessary to meet the scoping requirements of the application and the current peers in the network. For our model game, the 2 dimensional virtual world corresponds to a 2-tree so that each point in the world maps to a leaf in the 2-tree.

For example, a player is going to cast a snow storm centered at (.48, .28). She joins the N-Tree leaf that encompasses (.48, .28) (see Figure 1). She would then become a member of the group in that leaf so that she could totally order the casting of the spell with other events by the group members.

Because the main purpose of the N-Tree is to reduce communication and propagate events, leaves in the N-Tree are subdivided whenever the population of a leaf exceeds a given threshold t. This threshold is game specific and depends on the communication and latency requirements of game. For example, a first-person shooter type game, which has a high update rate and low latency requirements, might have a lower threshold t to make sure that players are interacting with as few other players as possible to keep from overwhelming their Internet connections. Thus, when the leader of a leaf discovers that the population of the node has exceeded the threshold, the leaf is subdivided into N children and the peers are placed into the appropriate leaves.

4. EVENT ORDERING

Using the N-Tree, we can assume that peers are organized according to their scope of interest in the virtual world. At the leaf level, all peers run a total event ordering protocol such as NEO or Lockstep [2, 9]. These protocols ensure that events are ordered without cheating. For example, two players might be competing to take a treasure from the game. NEO or Lockstep would resolve the situation so that only one of them could actually get the treasure.

At times when the scope of an event exceeds the boundary of the leaf that it was generated in, the N-Tree is used for event propagation (discussed further in Section 5). For example, in Figure 1, the snow storm event has a radius of .05, which intersects the boundary of the current leaf. This event would be totally ordered in the leaf at which it occurs in, and then propagated to neighboring leaves through the N-Tree. The problem then arises that events from two different leaves may preclude each other but one may be delayed on the network, so how can we order events in this situation?

Normally, if we used an event ordering protocol between all players, each event would have to have been *accepted* by all players before it actually occurred. Events with a large scope would then take a long time before they were accepted, as every player within that scope would have to be contacted so that the event could be totally ordered with their other events. In essence, we have a trade-off in the interactiveness of event ordering with the ability to totally order events.

In a peer-to-peer game, interactiveness has to be chosen, or else we loose the main purpose of the game: to interact with the virtual world and other players! Fortunately, games are forgiving in this respect because we can turn back time in the game to allow events that arrive late to occur and possibly preclude events that have already occurred, as with Jefferson's Time Warp algorithm [11]. In fact, modern day games already do just what has been described. When an event arrives late that precludes a previous event, the game time is *rolled back* to when the event should have occurred, and all later events are then replayed up to the current time. While this solution is not perfect, the alternative is intolerable latencies.

The NEO protocol synchronizes peers and uses time stamps for event ordering [9]. Events generated locally in a leaf are totally ordered and can be further ordered with other events in the game if their scope exceeds the boundary of the leaf . The Lockstep protocol would require the simple modification of adding timestamps to each round so that when the event was propagated through the N-Tree other peers could determine the ordering.

5. THE N-TREE PROTOCOL

In this section, we describe how we build and maintain an N-Tree while handling joins and leaves. We use a DHT as a bootstrap mechanism to discover nodes in the N-Tree. Nodes use the DHT to register their IP addresses with locations in the application space. When a new node wishes to join the system, they do a lookup on their location in the application space, which returns a list of nodes in that area.

The advantage of the DHT is that if some node in the DHT has failed and entries are missing, the next responsible node in the DHT will respond with the entries it has. Thus, as long as a single node in the N-Tree can be discovered on the DHT, new peers can join the system.

5.1 Joining the N-Tree

To join, a peer queries the DHT for some set of nodes in her location of the application space. She then sends a *join* message to one of the nodes that includes the peer's current scope in the application space. The receiving node looks at the scope of the join and determines whether to forward the message up or down the tree. Eventually, the *join* message reaches a node that is within the peer's event scope, and this node notifies the peer so that it can join the N-Tree at this location.

Each node further has the ability to divide its scope based on the communication needs of the application. The reason that subdivision is application specific is that some applications have a much higher event rate than others. Applications with a high event rate will probably need to subdivide the state space as much as possible so that the t-way communication between the t peers in a node is reduced as much as possible without hurting the performance (or security) of the application.

When a leaf node discovers that it contains more than a given threshold t for the application, it sends a *divide* message to the peers. The application space that this leaf represents is then subdivided evenly along each of the N dimensions. Each peer within the newly divided subspace then determines who will act as the leader (using any appropriate leader election protocol [16]), and the leaders notify their parent of their leadership status. These leaders represent the new leaves of the previous leaf that was subdivided. In this manner, the tree can be subdivided as necessary.

5.2 Leaving the N-Tree

To leave the N-Tree gracefully, a peer sends a *leave* message to all other peers in its node (or *group*), with a new leader being elected and the parent node notified if the leaving peer is the node's current leader. For an ungraceful leave, either a parent or a group will notice that a peer is missing when they try to forward an event to it. The case where the peer is not a leader is trivial. However, when the peer is a leader, the protocol must handle rejoining the group to the tree. If the group noticed that the leader is missing before the parent did, then they can simply re-elect a new leader and notify their parent.

On the other hand, if the tree is attempting to forward an event to a subtree, then it may discover that a child is not responding to an event. To handle this case, leaders should periodically send membership lists to their parent so that the parent can quickly send the event to another group member. Furthermore, the parent can queue events until the subtree is reconnected. However, even if the membership list of the parent is outdated, the parent can locate members of the subtree through the DHT.

If a peer leaves that is both a leader of a group and of one or more parent nodes, then the group will first elect a new leader and then the leader will contact other leaders to elect higher level node leaders. The DHT allows peers to discover other leaders if they do not already know of them.

When peers leave, the leader of the node determines if the subtree can be collapsed. Subtrees are collapsed whenever the total number of peers in the subtree is less than or equal to $t/2$ in order to keep the paths in the tree short and to prevent unnecessary subdivision and collapsing of branches in the tree.

5.3 Event Propagation

Event propagation occurs when the scope of an event exceeds the sub-domain of a node in the N-Tree. When a peer generates an event, the event is exchanged with other peers in the same node. When the leader receives the event, she checks the scope and compares it to the scope under her responsibility. If the scope of the event is not completely contained by her node, she forwards the event to her parent. The parent then forwards the event to all children whose scope intersects with the event scope, and additionally forwards it up the tree again.

5.4 Failures

Last, we consider the issue of failures. In an interactive game, the players will get frustrated if they have to wait for several seconds each time a leader is lost. One may ask whether we can ensure that leader loss does not cause the simulation to *pause* while a new leader is elected. In particular, as the game grows in population, the probability that a leader will be lost also increases.

For this case, we note that each N-Tree node keeps a membership list of the group members of its child nodes. If we assume the majority of events are local, then a lost leader will be detected with a high probability (and repaired) before the simulation is affected. However, to maintain interactiveness, each leaf group could elect two or more leaders. Events would then be forwarded through the additional leaders, with new leaders elected whenever one leaves.

6. ANALYSIS OF N-TREES

6.1 Asymptotic Analysis

In order to understand N-Trees as a communication structure for scoped events, we provide a simple analysis of their performance in terms of messaging. In general, all operations take at most logarithmic time in terms of the number of peers, while some only take constant time. Table 1 summarizes these results. In the following discussion, n is the number of nodes in the N-Tree, p is the number of peers, t is the threshold for subdividing a node, and d is the

Table 1: Asymptotic Messaging Costs

Operation	Player distribution	
	Pathological	Uniform
New Member Join	$O(\lg p) + O(h)$	$O(\lg p)$
Move to new node	$O(h)$	$O(\log_d n)$
Amortized movement	$O(1)$	$O(1)$
Leave	$O(1)$	$O(1)$
Collapsing branch	$O(1)$	$O(1)$
Subdividing leaf	$O(1)$	$O(1)$
Event propagation	$O(h)$	$O(\log_d n)$

p is number of peers, h is height, n number of nodes, $d = 2^N$ (or the number of leaves per branch) in the N-Tree.

number of leaves per branch (i.e., $d = 2^N$ in the N-tree), and h refers to the height of the tree.

As with binary trees, many operations on the N-Tree are based on its height, h. N-Trees do not try to balance themselves to maintain their height to be logarithmic to the number of nodes in the tree. Thus, in a worse case scenario, the height of the tree could be $O(p/t)$. However, this scenario only represents the case when all players are in the same location in the virtual world (a case we assume to be extremely rare or non-existent as p gets sufficiently large).

Unfortunately, no scientific measurements have been done that show the distribution of players in a game. Anecdotally, game designers want a uniform distribution of players in the virtual world to keep the level of inter-player messaging low (so that the game can scale to a large number of players). Given a uniform distribution, the N-Tree is balanced and $h = O(\log_d n)$. Note that the height of the N-Tree is based on the number of *nodes* in the tree, not the number of *peers* in the game. Assuming a uniform distribution, $n = O(p/t)$ since each leaf is subdivided when the number of peers in it exceeds the threshold value t.

Joining is a $O(\lg p) + O(h)$ operation. The $O(\lg p)$ timing comes from the search time for most DHTs (assuming all peers are registered in the DHT). Once a node is found for the N-tree, a peer will most likely be able to join the N-Tree in constant time. However, if the DHT is not up to date, it can take up to $O(h)$ time to locate an appropriate node through the N-Tree.

Once the N-Tree is joined for the first time, moving in the N-Tree is a much faster operation. The majority of player movement will be slow (in comparison to warping from one part of the world to another where a fresh join from the previous paragraph would be used), so a player simply leaves their node and joins a neighboring node, requiring only at most $O(h)$ time (in the case where we join the node that is the neighbor of a completely separate branch of the tree). We can show that a player moving from one end of the world to another has an *amortized* movement cost of $O(1)$, though we omit the proof due to space constraints.

Leaving, on the other hand, is a simple $O(t)$ operation, where t is the maximum number of members in a leaf (the application threshold before subdivision). When a node leaves, it must contact the other $t - 1$ members to notify them that it is leaving.

To analyze the complexity of subdivision, we simply examine leader election protocols. A constant number of messages are required to initiate subdivision. However, leader election using standard protocols can take from $O(1)$ to $O(t^2)$ messages, where t is the maximum number of peers in a leaf [16].

Subdivision has another cost in the amount of state that must be stored at each node. For N-dimensions, each node must store 2^N pointers to children. Applications designers should be motivated to reduce the application state space when possible to avoid a state explosion. We believe that most applications will not have a large number of dimensions in the application state space. In particular, we think that games can use 2 or 3 dimensions sufficiently to subdivide the application state space.

N-Tree collapsing is a $O(1)$ operation. We assume in this case that $O(2^N) = O(1)$ because the N value is a constant value set by the designers of the application. For peer-to-peer games, N would probably be 2 or 3, meaning 4 or 8 messages at most for collapsing a branch. Each node is periodically sending membership lists to its parent. When the parent notices that the number of peers in its subtree is less than or equal to t, the parent sends out a message to the peers and collapses the tree. To prevent repeated subdivision and collapsing, the minimum and maximum threshold for a leaf should be different.

Event propagation is a $O(h)$ operation, where h is the maximum height in the tree. In the worse case, a group at the lowest level in the tree generates a global event which must travel to the root of the tree and back to every other group in the application. As with joining, the worse case scenario is $h = O(p/t)$, which again represents a case where most players are in a single location in the game. With a uniform distribution, $h = O(\log_d n)$. However, even if most players are located in a single location, we believe that the majority of events will be local to that area, so that most events will not need to traverse the entire N-Tree. In most cases, events will travel through only a few levels of the tree, keeping the cost of event propagation low.

6.2 Performance Analysis

In addition to looking at the asymptotic performance of N-Trees, we wish to see how the performance of N-Trees affects the latency players would experience if N-Trees were used to propagate events, especially in comparison with similar architectures. In the architecture by Knutsson et al. experiments were run using their simulator with 1000 and 4000 players over a virtual world divided into 100 and 400 regions, with players uniformly distributed over the world [13]. Players were connected over a randomly generated topology with latencies between 3 to 100ms. For Mercury, the authors only simulated 20 and 40 players in each simulation, but uniformly distributed players and assumed a random way-point model of movement for players [3]. In the Mercury simulations, they assumed players were connected with a star topology, each with a 20ms delay between each other and the simulation field was about $2n$ times the maximum distance a player could move, for n players. Neither work considered events that were not local to the players or a particular object.

To analyze the performance of N-Trees in a similar setting, we assume a uniform distribution of players, only local events, and a 20ms delay between players, as with the Mercury simulations. We then extrapolate the data from [13] and [3] to get an idea of how these three architectures compare for the dissemination of messages.

Using this data, we plot our hypothesized performance based on N-Trees and using the simulation parameters common to Mercury and Multicast regions using Scribe. Please note that the graph is purely speculative, but demonstrates the effectiveness of organizing players by their application-level interest instead of by the shortest-path multicast tree. Figure 2 shows our hypothesis graphically.

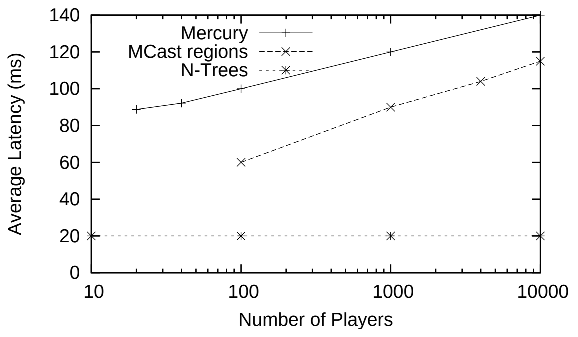

Figure 2: Theoretical performance of three different architectures. For Mercury, the first two data points at 20 and 40 players are taken from [3], with the rest of the points derived from their measured number of hops in the Mercury DHT. For Multicast regions, points 1000 and 4000 are taken from their simulations in [13], with the rest of the points extrapolated from their data and Scribe performance. N-Tree performance is hypothesized based on the simulation parameters of previous work in [3, 13]

We believe our hypothesis will hold for the simulation parameters used in [3, 13]. Our reasoning is if players are uniformly distributed, N-Trees perform optimally. The movement models used in simulations by Bharambe et al. and Knutsson et al. ensure that the population will stay uniformly distributed. Thus, the N-Tree will be perfectly balanced and subdivided so that players close together exchange events directly with each other. With a 20ms delay between all players, this results in an average 20ms delay, regardless of the population of the game.

Clearly, a more complicated and detailed set of experiments are needed to validate N-Trees. Indeed, the true test for N-Trees, Mercury, and Multicast regions, occurs when players have a non-uniform distribution and events have scopes that occupy more than a single point in space. Knutsson et al. suggest dynamic region adjustment for these situations, but do not describe how to accomplish this. Mercury, on the other hand, with its query language should be able to handle *crowded* situations more gracefully.

7. CONCLUSION AND FUTURE WORK

N-Trees and scoped events offer a scalable alternative to event ordering in peer-to-peer games. While not all peer-to-peer applications have events which can be scoped and require scalable event ordering in terms of the number of peers, our work provides a mechanism to make this class of applications scale.

Our hypothesis is that N-Trees and scoped events can help make peer-to-peer games possible by providing an efficient communication architecture as evidenced by the asymptotic analysis of the various operations on N-Trees. Our future work is to demonstrate this through a more detailed set of simulations that includes a more realistic model for player distribution, movements and event generation. These simulations should measure the cost of event propagation when event scopes fall outside of the leaf they occur in, delay when group leaders leave (or fail) and the cost of rebuilding the N-Tree, the cost of event ordering in terms of messaging and latency when players move throughout the virtual world, and the cost of event ordering when players gather in local areas (to test pathological cases). We plan to compare N-Trees further with other architectures for peer-to-peer event ordering. Finally, we hope to show that N-Trees can be generalized to other peer-to-peer applications requiring scalable event ordering.

8. REFERENCES

[1] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *ACM SIGCOMM*, pages 205–217. ACM Press, 2002.

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

[3] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting scalable multi-attribute range queries. In *ACM SIGCOMM*, August 2004.

[4] 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.

[5] W. chang Feng, F. Chang, W. chi Feng, and J. Walpole. Provisioning on-line games: A traffic analysis of a busy counter-strike server. In *ACM Internet Measurement Workshop*, 2002.

[6] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *ACM SIGMETRICS*, pages 1–12. ACM Press, 2000.

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

[8] J. D. Foley, A. Van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics*. Addison-Wesley, 1996.

[9] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *ACM NOSSDAV*, June 2004.

[10] Gnutella. `http://www.gnutella.com`.

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

[12] I. Keidar and S. Rajsbaum. On the cost of fault-tolerant consensus when there are no faults—a tutorial. MIT-LCS-TR-821. Technical Report MIT-LCS-TR-821, MIT, Massachusetts Institute of Technology, Cambridge, MA, 02139, May 2001.

[13] B. Knutsson, H. Lu, W. Xu, and B. Hopkins. Peer-to-peer support for massively multiplayer games. In *IEEE Infocom*, March 2004.

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

[15] L. Lamport. The part-time parliament. *ACM Trans. on Comp. Syst.*, 16(2):133–169, May 1998.

[16] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

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

[18] S. Ratnasamy, M. Handley, R. Karp, and S. Shenkar. Application-level multicast using content addressable networks. In *Network Group Communications*, November 2001.

[19] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350. Springer-Verlag, 2001.

[20] J. Smed, T. Kaukoranta, and H. Hakonen. Aspects of networking in multiplayer computer games. In *NetGames*, April 2002.

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

[22] B. Zhang, S. Jamin, and L. Zhang. Host multicast: A framework for delivering multicast to end users. In *IEEE Infocom*, June 2002.

[23] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *ACM NOSSDAV*, June 2001.