An Active Self-Optimizing Multiplayer Gaming Architecture

V. Ramakrishna, Max Robinson, Kevin Eustice and Peter Reiher
Laboratory for Advanced Systems Research
Department of Computer Science
University of California, Los Angeles, CA 90095
{vrama, max, kfe, reiher}@cs.ucla.edu

Abstract

Multiplayer games are representative of a large class of distributed applications that suffer from redundant communication, bottlenecks and poor reactivity to changing network conditions. Many of these problems can be alleviated through simple network adaptation at the infrastructure level. In our model, game packets are directed along the edges of a tree connecting the players, aggregated and multicast as necessary. This tree is heuristically formed, and is dynamically adjusted in response to changes in network conditions.

We have designed and implemented a prototype using ANTS that performs these adaptations for unmodified DOOM clients. Active networks is currently the only open architecture suitable for these types of applications. We present analytical results that illustrate the reduction in communications overhead, and show that the tree can quickly adjust to changing network conditions. The overhead of the active networks layer is acceptable, especially in wide-area networks.

1. Introduction

In recent years, the multiplayer gaming industry has exploded, enabling millions of gamers around the world to play games like EverQuest, StarCraft and Quake with one another. Each of these games supports thousands of players. These games typically have real-time constraints.

Much work has gone into improving the performance and scaling of these games, largely focusing on improving response time while maintaining consistent state among player nodes assuming unreliable packet delivery [1,2]. As graphics and animation quality improve, delivering only the essential data for an individual client has been an important research focus. Little work has been done to improve the underlying game network infrastructure, which could not only provide throughput gains but also improve consistency and interest management.

Traditionally, game world designers have used one of two models: peer-to-peer or client-server. In the former, each player performs multiple unicasts of the game state to all other players; this provides optimal response time to the players and is feasible in a broadcast medium like a LAN, but fails to scale much beyond that. Also, identical game state is sent repeatedly, resulting in redundant communication. In client-server architectures, each player sends updates to a server, which computes new state and sends relevant information to all the players. The average response time perceived by players is sub-optimal, but this approach scales well. Unfortunately, the server becomes a bottleneck and a single point of failure. Nonetheless, this model is popular with game companies since it allows them to retain administrative control.

Response times in these architectures tend to be highly skewed in favor of some players. The structures are also static, and cannot respond well to changes in network and node conditions, and players joining and leaving.

Our approach retains most of the virtues of both the models and eliminates many of their drawbacks. We construct a multicast tree connecting the players that has a low average node-to-node latency. A tree is rooted at a centrally located node. Player packets are aggregated at the tree branch points and propagated upwards. The root multicasts an aggregated packet to the clients, who extract the game packets. The root monitors network conditions and changes the tree structure, relocating the root, when conditions change. This infrastructure is hidden from the application, so that the game need not be modified. This infrastructure enhances reliability and performance.

These techniques can be applied to a larger class of applications, including interest management in distributed simulations [3] and publish-subscribe systems.

We use active networks [4] to enable computation at both end nodes and intermediate nodes through the use of injected code. The intermediate nodes in a connection can perform computation on the data stream, in addition to routing packets. Active networks facilitate dynamic code execution, new protocol deployment and creation of overlay networks and support load-balancing.

2. Related Work

Our work has been influenced by two systems previously designed in our lab. Panda [5] is an active networks-based adaptation framework that enables intelligent adaptation of unaware network applications. Panda responds in real-time to changing network conditions and deploys active network adapters to optimize UDP communications. Conductor [6] is a TCP-based open architecture framework that provides a distributed, coordinated, application-transparent adaptation facility.

Various projects use active networks to perform routing adaptations, including multicasting. The ARRCANE project [11] investigates active routing in mobile ad-hoc networks, which are in constant flux; the protocol is resilient to changes in network conditions. Reliable and customized multicasting using active networks have been investigated in [12,13]. Gathercast, similar to packet aggregation in our middleware, has been implemented using active networks [14].

Much work has been done to improve gaming architectures. The MiMaze architecture [7] is based on a peer-to-peer model, but uses IP multicast for packet delivery; its topology is very static, reliability is not a concern, and its traffic reduction is sub-optimal. The design of a mirrored-server architecture that uses a reliable multicast protocol (CRIMP) for packet delivery and a mechanism for clients to locate the servers nearest to them is described in [8]. A common drawback of all these systems is that they require the game to be extensively re-modeled.

There is much work in building dynamic and fault tolerant multicast trees. Revere [9] builds overlay networks that forward security updates, handling reconfiguration for broken connections and failed nodes. [10] describes a distributed algorithm for building multicast trees that adapt to group members joining and leaving the tree during execution. Most other reliable multicast work has focused on ensuring that each packet eventually reaches each group member [16]. We aim for something weaker; a small amount of packet loss is not a concern so long as the tree is repaired (or just adjusted) quickly.

3. Design

3.1. Gaming Infrastructure

The gaming infrastructure combines the benefits of the peer-to-peer and the client-server models while eliminating some of their drawbacks. We connect all the game nodes to form a tree network, similar to building a multicast tree. One of these nodes, at a "central" location with respect to all the player nodes, is selected to be the root of the tree, similar to the core in a core-based multicast tree. The definition of central could vary; in our case, we use latency to measure distance between nodes. The center must be chosen to minimize its latency from all players. This heuristic ensures that none of the players perceive much worse response time from the average, and that all game packets pass through this node.

Figure 1 illustrates how packets are routed in this model. Each player sends out a packet containing its update of the game state along its upward link, i.e., the link along the path to the root of the tree. When a branch node on the tree receives packets from each of its incoming links, it aggregates them into a single packet and forwards them along its upward link. This routing goes on until the root node receives packets from each incoming link. It aggregates these packets into a single packet and multicasts them to the player nodes; duplicating received packets at every branch node on the tree and sending them along all downward links of that node, i.e. those links that are not an upward link. Since each leaf node must be a player, this process terminates when each leaf receives an aggregated packet. This packet is deaggregated into individual game packets and delivered to the game client.

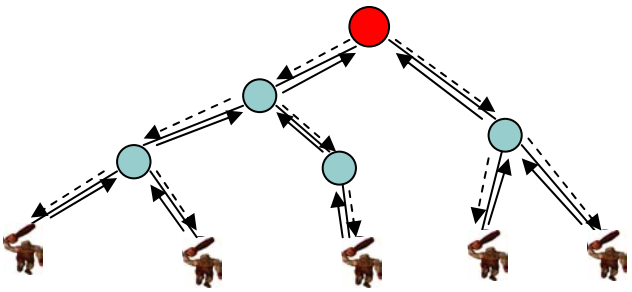


Figure 1. A tree connecting game players

3.2. Tree Building and Center Location

Multicast trees are of two types: source-specific and group-shared. In the former, a single node is the only multicast traffic source, while group-shared multicast trees allow every multicast group member to be a potential source. Most multicast routing protocols in use today are source-specific, but the latter is more suitable for our application, where every player must deliver packets to the other players and determining the optimal location of the root (server) is part of the problem. The general problem of finding an optimal, minimal delay, group-shared multicast tree is a well known NP-complete problem known as the Steiner tree problem [17].

A simple, $O(n^3)$ heuristic solution in the worst case, is twice as bad as the optimal, NP-Complete, solution [18]. Our algorithm is an iterative application of Dijkstra's single-source shortest path algorithm to build shortest-path trees to the multicast group members for every potential source, selecting the optimal tree out of the set of

potential trees. A min-priority queue implementation of Dijkstra means each iteration is $O(n^2)$.

Once the multicast tree has been built for the given set of game players, a center node is picked to be the root. The *eccentricity* of a graph vertex is the longest distance from that vertex to any other node in the graph. The *radius* of a graph is the smallest value of eccentricity among all vertices. The *center* of a graph is a subset of its vertices that have their eccentricity equal to the radius; there are at most two center vertices for a tree. We mark the center of the multicast tree as the root; if there are two candidates, one is chosen at random.

3.3. Network Monitoring

The multicast tree infrastructure performs continuous network monitoring to detect when the current tree structure becomes suboptimal with respect to average latency between nodes. When a change in network conditions is detected, a new tree is constructed and a new root is marked; the player nodes remain where they were before, but could play different roles in the new tree, for example, a player who was a leaf in the old tree could be a branch point performing aggregation and duplication in the new tree. The entire multicast tree is now relocated to the new one, which becomes a routing medium as soon as all nodes have been given the updated information.

It is the responsibility of the root node to monitor network conditions, and also execute the tree building and the root location algorithms. As this root performs more work than the other nodes in the tree, it can be visualized as a virtual server, and the modification of the tree can be considered to be a server relocation operation. After the initial tree is formed, no external intervention is needed.

3.4. Role of Active Networks

This infrastructure is built using active networks. Tree building requires knowledge of a set of active nodes as input along with the location of the players. All nodes in the tree must be active; this is necessary for them to be able to perform the necessary functions.

Game packets will be intercepted by the active networks-based middleware and queued to the virtual (overlay) network layer, which performs packet forwarding independent of the lower IP layer. The game packet is encapsulated as an active packet, the only addition being a header that contains application-specific information. Adapter code is deployed at every active node, which is executed upon receiving an active packet. In our infrastructure, aggregators, duplicators and deaggregators are deployed at the nodes. Each node knows its immediate neighbors in the tree and has routing information for them. The node also has a set of *roles*, i.e. that of a player, a branch point or a monitor. It can take on any subset of these roles. The node also maintains game state. If it is performing aggregation, it needs to wait for packets to arrive from all its children; it queues them for aggregation until all arrive. At this point, packets are aggregated and sent to the parent. Duplication and deaggregation are performed just from the knowledge of its roles.

4. Implementation

The target multiplayer game was DOOM. The game protocol proceeds in lock-step. Each player computes its state periodically and sends it to other players. When a player has received an update from every other player, his game state advances. A fast-paced game, DOOM requires real-time updates to maintain a smooth flow.

We chose a peer-to-peer, UDP-based version of DOOM, due to the relative ease of adapting peer-to-peer games rather than server-based ones, and because one of our goals was to eliminate a centralized server; also, we could examine the routing infrastructure in isolation.

We used the ANTS active networks platform [15], a Java-based toolkit that provides an execution environment and a protocol programming model allowing customization of packet forwarding. We implemented under Linux, using the IPcept kernel module designed for Conductor and Panda to perform transparent socket proxying and masquerading.

A typical system contains a set of ANTS-enabled nodes, including the game clients. Initially a static tree must be built, with roles assigned to each node, and a root node chosen manually. Each active node stores the adapter code and maintains a routing table for known active nodes, as well as a neighbor list consisting of active nodes located one hop away. All the active nodes in the vicinity interested in participating in the infrastructure must send registry capsules to the root.

When the DOOM client sends out multiple packets to other players, these packets are intercepted by IPcept, which passes them to the middleware layer. Since these packets contain identical data with only the destination address being different, only one packet is actually encapsulated and forwarded; the tree structure is responsible for sending the packet to all the other clients. When capsules containing game packets reach a tree branch point, they are aggregated into a single capsule; aggregation consists of extracting the game packets from the received capsules, concatenating them and appending the capsule header. Every nodes performing aggregation maintain an ANTS-defined NodeCache object, in which packets can be temporarily stored until it is time for aggregation. Because of real-time constraints, we have set a timeout period; if all expected packets do not arrive within that period, the existing packets are aggregated and forwarded. Deaggregation is the reverse of aggregation: the ANTS

header is stripped off and the game packets are extracted based on knowledge of their sizes (for DOOM, they are typically 16 bytes).

For latency monitoring, each node "pings" its active neighbors periodically and sends its observations as capsules to the root. The root now has a set of nodes and edges with weights to work with. Each edge has two weight values, as perceived by the two end-points; we take the conservative approach of choosing the higher value. Based on this information, the root executes the tree building algorithm as outlined in Section 3.2. If the new tree is different from the existing one, control capsules are sent to the new tree nodes asking them to assume their new roles. Once all the updates have been received, the new tree comes into effect and packets are routed through it. The old tree nodes are not deactivated, so any packets still in flight will be routed to the clients, preventing any packet loss.

5. Analysis of Benefits

Our architecture achieves reduction in data communication compared to both existing models.

Consider the total number of packets that game players send out into the network, given that the number of game players is n. For a pure peer-to-peer model, each player must send n-1 packets, one to every other node. The total number of packets sent out into the network is $n*(n-1)$, which is $O(n^2)$. In a client-server model, each player sends one packet to the server, which then sends n packets, one to each client. The total number of packets is $O(n)$. In our dynamic multicast tree, each player sends out one packet, so the total number of packets is $O(n)$.

The network traffic generated per round of game state updates is the total number of packets traversing network links. This metric is difficult to measure, since it is highly topology and routing table dependent; we demonstrate this through an example. Figure 2 shows a network of nodes, with a tree connecting the player nodes. (Note:

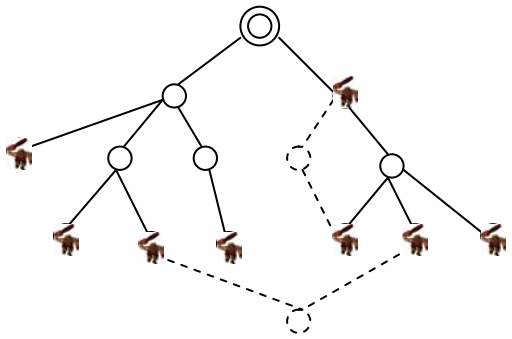


Figure 2. A multicast tree

This tree is not representative of the actual one that would be constructed using our algorithm). Consider the packet communication during one round of updates for each model, and assume the tree root for the dynamic multicast acts as the server in the client-server case. Except for the dynamic multicast case, all communication take place along the shortest path between peers or from client server. Table 1 shows the reduction in network traffic that dynamic multicasting achieves.

	Peer-to-Peer	Client-Server	Dynamic Multicast
# packets sent out by parties	56	16	8
# packets in the network	207	40	26

Table 1. Comparison of models based on network in Figure 2

The amount of transmitted data can vary depending on the size of the packets. Aggregation achieves packet reduction, but the total byte content remains the same (in fact, it increases slightly due to the appended capsule headers). Thus our comparison of the client-server model with the multicast model is not strictly fair, since multiple packets transmitted over a link in the former case might contain less data than an aggregated active packet. But aggregation reduces the number of packets, and real-time games packets are the order of a few tens of bytes (16 bytes in DOOM), so there is less chance of congestion with an aggregated packet, unless the number of players is very large. Fewer packets also means less work at routers, so the overall latency is reduced. For large number of nodes, the packets could be aggregated only so long as they remain with a fixed size limit. A server in the client-server model would rarely be in the same place as the root of a multicast tree constructed by our algorithm, because it is static and not chosen relative to the position of the clients. Therefore, packets may traverse more edges than in our multicast tree, leading to increased traffic. In the worst case, the multicast tree root may have to handle as much data as the server in a client-server model, creating a potential bottleneck.

The dynamism and self-adjustment of the game infrastructure is a step towards ubiquitous gaming environments. Fault-tolerance is also enhanced. With small adjustments, this architecture could handle failure of the active nodes and the virtual links between them. If reliability can be increased to a great extent, it would offset the disadvantages of the tree adjustment overhead.

There is no centralized server node in our infrastructure that is absolutely essential for game play. The root is a type of server, but with very restricted functionality that can be easily moved from one site to another.

The average response time latency is nearly equal, on average, for all players, because of the central root loca-

tion. All packets pass through this root, ensuring that two players never perceive widely inconsistent game state due to very different response times.

6. Experimental Results

We have designed and implemented a prototype of the middleware in our laboratory. This middleware was deployed on HP-Omnibook 4150 laptops running the Linux operating system. The operating system kernels were hacked to provide support for the IPcept module that performs transparent proxying and masquerading of asockets. (This functionality can also be performed in kernels of version 2.4.x and higher using the netfilter framework and setting suitable firewalling rules using the IPtables toolkit). The laptops communicated with each other through the UCLA CS department ethernet.

We performed a variety of tests using the test bed described above. A variety of network topologies of active nodes were tried out, with a subset of them being the game players. These nodes and connections were defined in an ANTS-understandable manner, with the routing tables being constructed automatically by the toolkit. (This user-level routing table functionality was also used extensively in our middleware for tree building).

We were most interested in observing and measuring the performance gains from two perspectives: one, how our dynamic multicasting framework compared with the traditional peer-to-peer and client-server models, and two, how feasible is active networks as a platform for building these kinds of frameworks. To measure the former, we considered network traffic as a metric, an informal argument about which was made in the previous section. To measure the latter, we considered the overhead incurred by the system in time units; we were also able to observe how quickly the tree modified itself when needed.

The technique used for measuring architectural and active networks performance also varied. The system performance could be measured using the resources we had, i.e. a few laptops. On the other hand, network traffic measurements would be meaningful only if the number of nodes was reasonably large; simulation was the only was to perform this task.

The results of the experiments we performed are given below.

To measure the base cost of using the middleware, we used a simple topology that directly connected game playing nodes as described in Figure 3. The comparison was made between the middleware and no-middleware cases. First, without the middleware, the average time difference between successive packets received by a DOOM client was observed to be about 28.5 msec. With the middleware in place, this time difference rose to about 31 msec on average, though this varied widely; the reason for this variation can be traced to the fact that a bunch of packets reach the middleware at the client node, which are then serialized and queued immediately to DOOM. Thus, an overhead of about 1.5 msec was observed on average; this could rise under certain circumstances, as we shall explain later.

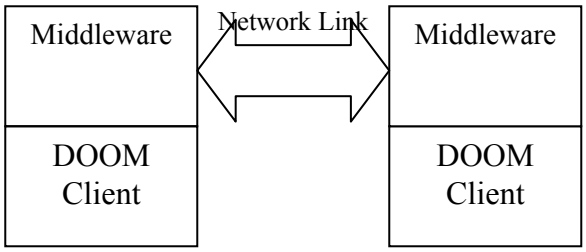


Figure 3. Simple topology for overhead measurement

This overhead was also measured in another way, by noting the execution time of the middleware layer; this turned out to be about 1.3 msec on average, though it varied widely between 1 to 2 msec. This reading seems to agree with the one observed above.

We must also mention that hardly any difference in quality was perceived when DOOM was played over active networks; the game playing experience remained almost as good as it was without the middleware.

Similar observations were made for other topologies consisting of more nodes. With a network of 3 nodes connected in a chain, with the end nodes being game players and the middle one being the root, the overhead suffered by the middleware at the game clients still remained about 1.3 msec. The aggregation and duplication adapters at the root node incurred about 2 msec overhead.

We also observed that the overhead could increase much beyond the average value. At the points where the monitoring of network conditions is done, the overhead could rise to a few tens of milliseconds, very rarely being as much as 50 msec. Also, game players observed slight jitter during these instants. The reason for this is that our network monitoring code is implemented entirely at the user level process; we are confident that using a kernel daemon to do this would yield much better results.

Another observation we were very interested in was the smoothness of transition from one tree to another. Network condition change was emulated and the time taken to transition from one tree root to another was measured for the topology in Figure 4 using the same techniques as outlined above. Again, there was an observable jitter, with transition overhead being a few hundred milliseconds, with the maximum observed being 700 msec.

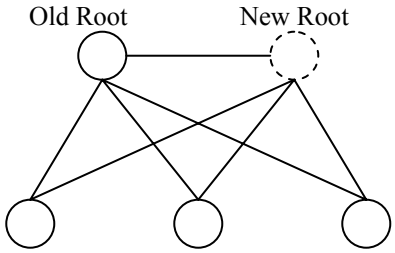
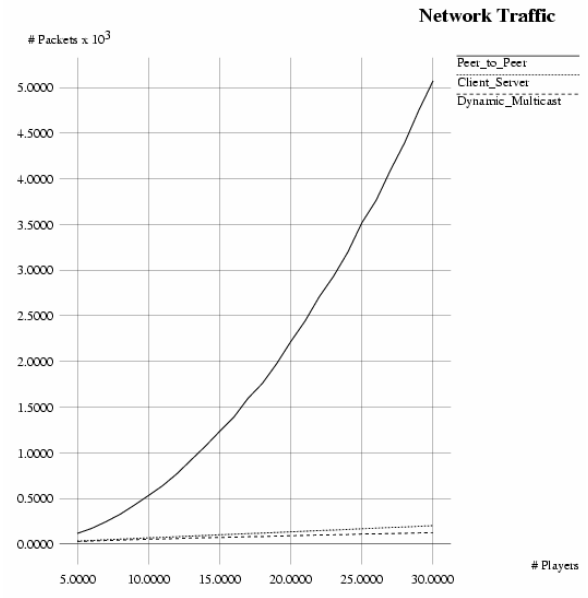


Figure 4. Topology for testing transition overhead; bottom three nodes are players

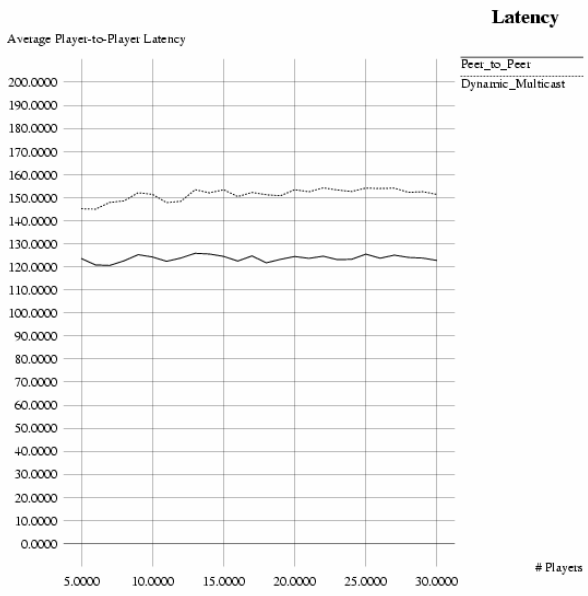
To analyze the above observations, we must keep in mind that these observations were made in a LAN environment, where the average node-to-node latency was under 1msec on average. The comparison between adapted and unadapted DOOM was all the more pronounced. For MANs and small WANs, where the communication latency could run into tens of milliseconds, the observed overhead would be negligible. Also, considering that the overhead at an individual active node remained somewhat constant for different topologies, a case could be made for the scalability of our approach.

The other experiment performed was a simulation to measure benefits in communication overhead, measured as the total number of packets seen by the network during a single round of message passing. Firstly, algorithms for peer-to-peer and client-server models were implemented, in addition to the multicasting framework. For the client-server case, the server was selected to be the same node as the root of the multicast tree. In addition to the network traffic, average node-to-node latency was also measured just to get an idea of how multicasting suffers as compared to peer-to-peer broadcast, which uses shortest paths for communication.

For simulation of graphs and multicast groups, we used the Georgia Tech topology generator. We designed four random weighted graphs of 250 nodes each: two of them were transit-stub graphs, one was a purely random graph using a Waxman model for connections, and another a 3-level hierarchical graph. All nodes of the graphs were considered active for the purpose of simulation. Multicast group size varied from 5 to 30, with a hundred random groups chosen for each size and the average reading taken. The comparison of network latency and average latency for the four graphs are shown in the figures below.

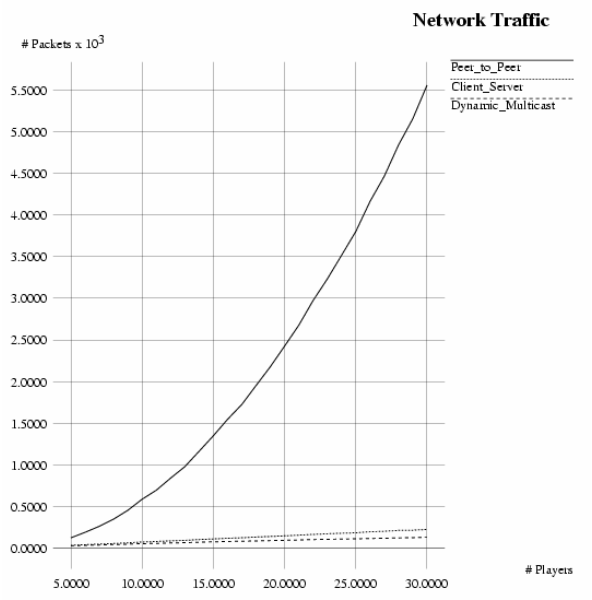


(a)

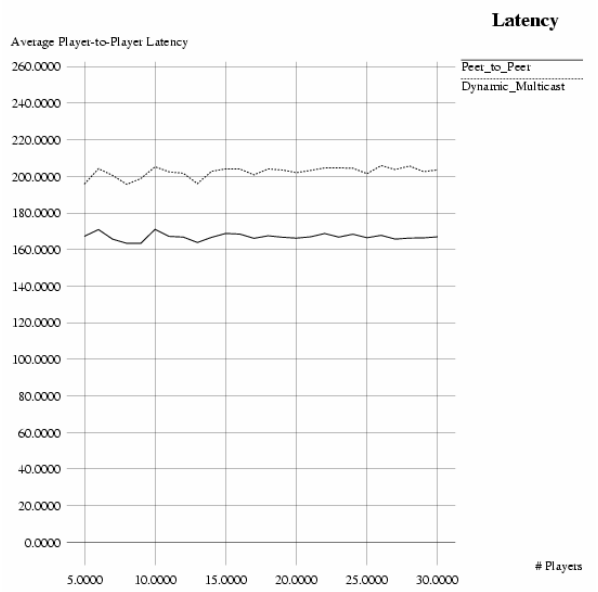


(b)

Figure 5. Transit-Stub graph: One transit domain with 5 nodes on average; each transit node has 7 stub graphs on average; each stub domain has 7 nodes on average

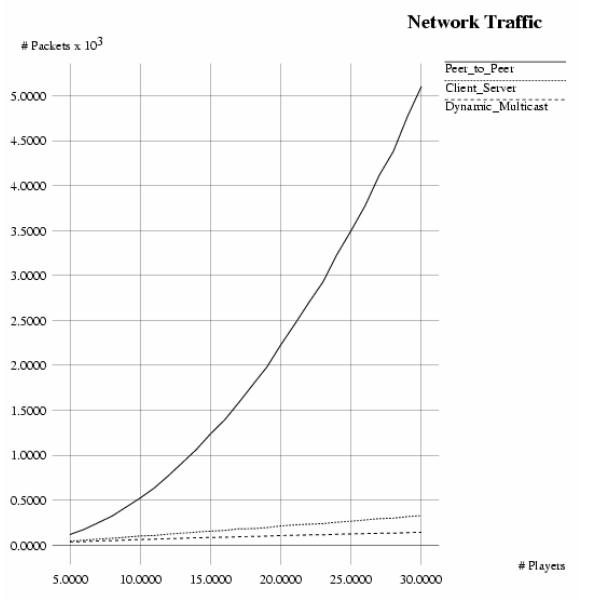


(a)

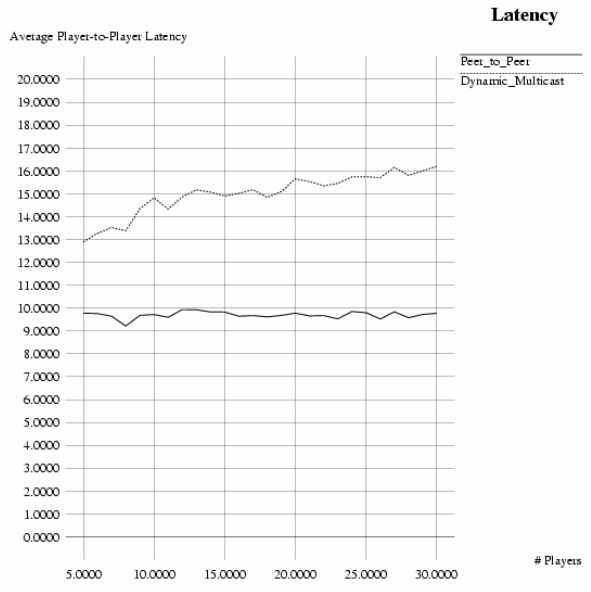


(b)

Figure 6. Transit-Stub graph: 2 transit domains with 5 nodes on average; each transit node has 6 stub graphs on average; each stub domain has 4 nodes on average



(a)



(b)

Figure 7. Random graph: 250 nodes and 1260 edges; Waxman parameters 0.3 and 0.2

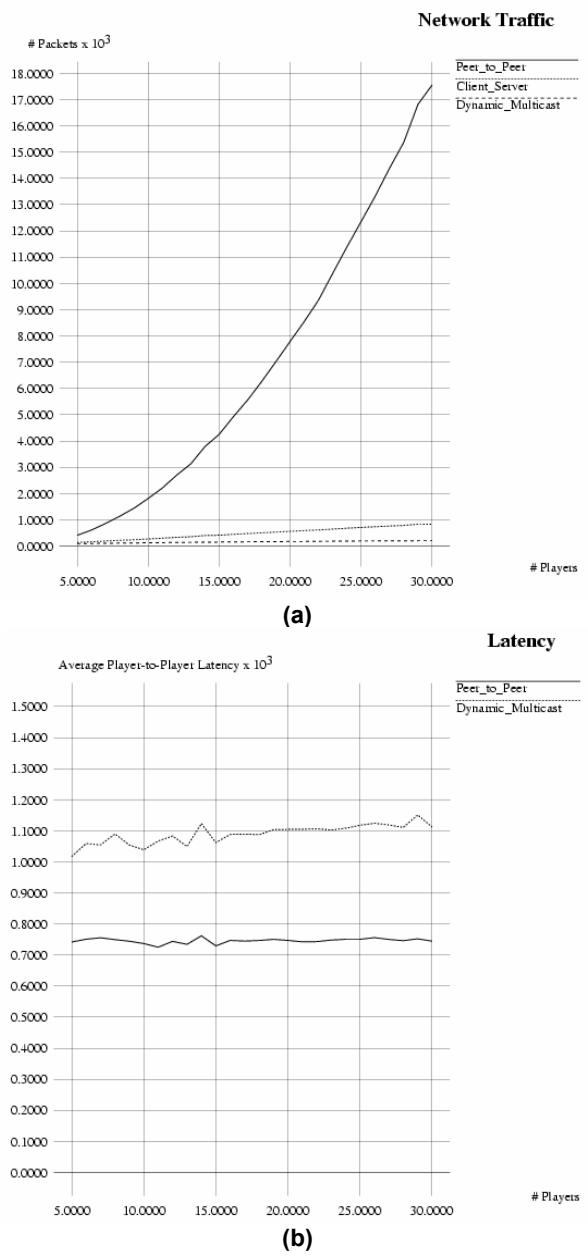


Figure 8. 3-level hierarchical graph: 5 nodes with edge prob. 0.4 at the highest level; 5 nodes on average with edge prob. 0.3 at the next level; 10 nodes on average with edge prob. 0.6 at the bottom level

Figures 5(a) to 8(a) show conclusively that our model performs better than client-server and much better than peer-to-peer for all multicast group sizes and the difference increases with increase in that size. On the other hand, the average node-to-node latency is somewhat lower in all cases for the peer-to-peer model, which is to be expected. This difference remains almost constant for all group sizes, and the latencies of both models are of the same order. Considering the huge gains in communication given by dynamic multicast over the peer-to-peer framework, this latency difference is a small price to pay. Also, it will not affect game-playing experience in any noticeable manner.

6. Future Work

There are various directions in which our system could be extended. Tree building methodology need not be based only on link latency; other factors like load the different nodes, congestion along links could be used to optimize the tree.

The current reliability of the system can be enhanced by replicating the monitor at multiple sites. If one monitor fails, others will take over and obtain a new position for the root. Scalability can be increased by replicating the root functionality at multiple nodes. Intermediate nodes could perform filtering and interest management, with more knowledge about the game, reducing the data to be communicated. Our architecture allows players to join and leave easily, but DOOM does not support this.

With wide deployment of active networks, independent game clusters could be built based on node proximity. These clusters could be formed without any manual intervention, with the tree roots deciding whether to admit a new player, and connecting him to his closest cluster.

7. Conclusion

We have designed and implemented a self-adjusting architecture for multiplayer games that can be deployed on both local and wide area networks. We have shown that active networks can be used to perform routing adaptations for multiplayer games, as we have seen in section 6. Our simulation results also prove conclusively the benefits in communication overhead over traditional models. As our techniques are application-transparent, the model is applicable to both new and legacy games. We believe that a wide variety of multiplayer games will benefit from using our architectural model. With currently available technology, it would possibly be a better option to implement a custom infrastructure, maybe at the network layer, in order to obtain better performance. Active networks, though, is still an evolving technology; we could expect that in a few years, it would become the *de facto* platform for deployment of new protocols.

Our approach is not restricted to the gaming world; it can also benefit a wider class of applications like distributed simulations. The performance impact on non-real time applications will be even greater than for multiplayer games.

8. References

[1] J. Steinman, J.W. Wallace, D. Davani and D. Elizandro, “Scalable distributed military simulations using the SPEEDES object-oriented simulation framework”, *Proc. of Object-Oriented Simulation Conference (OOS’98)*, pp. 3-23, 1998.

[2] E. Cronin, B. Filstrup, A.R. Kurc, S. Jamin, “An efficient synchronization mechanism for mirrored game architectures”, *Proc. Of the first workshop on Network and system support for games*, pp. 67-73, 2002.

[3] K.L. Morse, “Interest Management in Large-Scale Distributed Simulations”, *Technical Report ICS-TR-96-27*, Dept. of Information & Computer Science, Univ. of California at Irvine, 1996.

[4] D. Tennenhouse and D. Wetherall, "Towards an Active Network Architecture," *ACM Computer Communication Review*, Volume 26, No.2, pp. 5-18, April 1996.

[5] V. Ferreria, A. Rudenko, K. Eustice, R. Guy, V. Ramakrishna and Peter Reiher, "Panda: Middleware to Provide the Benefits of Active Networks to Legacy Applications," *DANCE 02* , May 2002.

[6] M. Yarvis, P. Reiher and G. Popek, “Conductor: A Framework for Distributed Adaptation”, *Proc. 7th Workshop on Hot Topics in Operating Systems*, 1999.

[7] MiMaze - http://www-sop.inria.fr/rodeo/MiMaze/Archi.html

[8] E. Cronin, B. Filstrup and A.R. Kurc, “A distributed multiplayer game server system”, *UM EECS589 Course Project report*, http://www.eecs.umich.edu/~bfilstru/quakefinal.pdf, May 2001.

[9] Revere project - http://lever.cs.ucla.edu/revere/

[10] F. Adelstein, G. Richard III and L. Schwiebert, “Building Dynamic Multicast Trees in Mobile Networks”, *ICPP Workshop*, pp. 17-, 1999

[11] ARRCANE project - http://www.docs.uu.se/arrcane/

[12] L. H. Lehman, S. J. Garland and David L. Tennenhouse, "Active Reliable Multicast", Proc*. of the 17th INFOCOM*, pp. 581-589, March 1998.

[13] S. Ramabhadran and J. Pasquale, “A framework for application-specific customization of network services”, *Proc. of the Fourth Annual International Workshop on Active Middleware Services*, pp. 35-40, July 2002.

[14] Y. He, C.S. Raghavendra and S. Berson, “Gathercast with Active Networks”, *Proc. of the Fourth Annual International Workshop on Active Middleware Services*, pp. 61-66, July 2002.

[15] D. Wetherall, J. Guttag and D. Tennenhouse, "ANTS: A toolkit for building and dynamically deploying network protocols," *Ph.D. dissertation,* University of Washington, 1998.

[16] B. Levine and J. J. Garcia-Luna-Aceves, "A comparison of reliable multicast protocols", *Multimedia Systems*, Volume 6, No. 5, pp. 334--348, 1998.

[17] R. M. Karp, *Complexity of Computer Computations*, New York: Plenum, pp. 85-103, 1972.

[18] S. Ali and A. Khokhar, “Distributed Center Location Algorithm for Fault-Tolerant Multicast in Wide-Area Networks,” *Workshop on Advances in Parallel and Distributed Computing, IEEE Symposium on Reliable Distributed Computing*, Oct. 1998.