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 adaptations 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 communication overhead, and show that the multicast 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. Most of these games require that the entire, or a large subset of, game state be visible to each player at all times; this involves communicating a large quantity of data. For these games to be playable, a number of real-time constraints must be satisfied.

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; this is usually done by reducing the amount of data communicated or by delayed communication [8, 10, 13]. As graphics and animation quality improve, delivering only the essential data for an individual client has become an important research focus. Little work has been done to improve the underlying game network infrastructure, which would not only provide throughput gains and reduce communication overhead, 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 suboptimal, 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.

The general drawbacks from which all Internet games suffer, irrespective of the underlying infrastructure, are heterogeneity of network and end-host characteristics, variable bandwidth, high latency and significant jitter. For the same reasons, these games have scaling problems. With non-optimally positioned servers, response times tend to be highly skewed in favor of some players. The structures tend to be static and cannot respond well to changes in network and node conditions, and also to the joining and leaving of players. To improve scaling and reduce skew in response time, mirrored-server architectures are used widely, one example being the architecture used in *EverQuest*. A mirrored-server architecture [7] is a compromise between the client-sever and the peer-to-peer architecture; a number of servers performing identical functions are deployed on the Internet, with clients connecting to the servers nearest to them. As in the client-server case, administrative control can be retained and communication overhead kept down

to tolerable levels; as in the peer-to-peer case, latency is significantly reduced. In spite of these advantages, the problem of static topology still remains; in the face of heterogeneity and changing conditions, these architectures still fail to provide optimal structures.

The approach that we describe in this paper retains most of the virtues of both 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 [11] and publish-subscribe systems. Using multicast trees to group data for interest management is described in [12].

We use active networks [5] 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, creation of overlay networks, and support for load-balancing, all of which are necessary for the infrastructure that we have described above; in addition, legacy applications can be adapted with minimal effort.

2. Related Work

Our work has been influenced by two systems previously designed in our lab. Panda [24] 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; it also performs dynamic planning for adapter deployment. Conductor [17] 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 [1] 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 [14, 23]. The feasibility of using active networks for multicasting has been investigated in [16]. Gathercast, similar to packet aggregation in our middleware, has been implemented using active networks [25].

Much work has been done to improve gaming architectures. The MiMaze architecture [18] is based on a peer-to-peer model, but uses IP multicast for packet delivery. Unfortunately, it suffers from many drawbacks: its topology is very static, reliability is not a concern, and traffic reduction is suboptimal. 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 [7]. A common drawback to all these systems is that they require extensively remodeling of the game; our system supports unaware adaptations for legacy games.

Considerable work has been done in building dynamic and fault tolerant multicast trees. Revere [21] builds overlay networks that forward security updates, handling reconfiguration for broken connections and failed nodes. A distributed algorithm for building multicast trees that adapt to group members joining and leaving the tree during execution has been described in [9]. Most other reliable multicast work has focused on ensuring that each packet eventually reaches each group member [2]. We aim for something weaker; a small amount of packet loss is not a concern so long as the tree is repaired (or simply adjusted) quickly. As we shall see later in this paper, our infrastructure also manages to ensure that game packets are not lost during the tree reconfiguration phase.

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. How we achieve this is described below.

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 communication 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 experience a response time that is much worse than the average. The position of the center has an impact on performance; this is because all game packets pass through this node, as we shall see in the following paragraph.

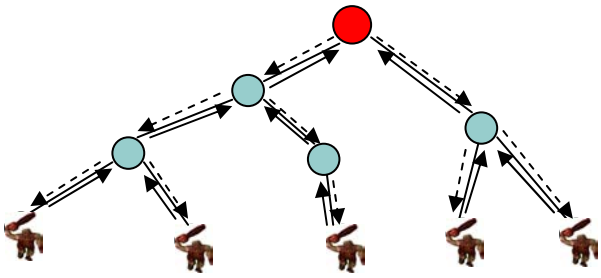

Figure 1. A tree connecting game players

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 through each incoming link. It aggregates these packets into a single packet and multicasts them to the player nodes. Received packets are duplicated at every branch node on the tree and sent along all downward links of that node, i.e., those links that are not an upward link. This process terminates when each leaf node receives an aggregated packet, since all leaf nodes must be players. At every player node, the received packet is deaggregated into individual game packets and delivered to the game client.

3.2. Tree Building and Center Location

Depending on topology and the tree construction algorithm, multicast trees can be classified into two categories – source-specific and group-shared [15]. In source-specific trees, a single node is the only multicast traffic source; this node initiates tree building. Group-shared multicast trees, also called core-based trees, have some arbitrary node appointed as the core so that all traffic passes through it. Most multicast routing protocols in use today are source-specific, but group-shared trees are 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 [4, 20]. Therefore, any practical algorithm for building multicast trees must involve tradeoffs between algorithmic complexity and the quality of tree produced. A simple, heuristic solution of complexity $O(n^3)$, in the worst case produces a tree that is twice as bad as the optimal NP-Complete solution, as mentioned in [22].

Our algorithm is an iterative application of Dijkstra's single-source shortest-path algorithm to build shortest-path trees to 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 implies that each of the n iterations is $O(n^2)$.

Once the multicast tree has been built for the given set of game players, a center node is selected to be the root. Before we describe the procedure for choosing the center, we must define some terms popular in graph theory; these definitions apply to weighted, undirected graphs. The *eccentricity* of the vertex of a graph 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, the eccentricity of each vertex being equal to the radius; there can be 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 change in conditions, which might cause the current tree structure to become suboptimal with respect to average latency between nodes. When a change is detected, a new tree is constructed based on current network information, following which a new root is selected. The player nodes remain where they were before, but can play different roles in the new tree. For example, a player who was a leaf in the old tree can 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 to 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 a server relocation operation. No external intervention is needed to perform this relocation; only the initial tree is configured statically.

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 perform the necessary adaptation functions.

Game packets are 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 roles and current game state, i.e., the node is waiting for a packet to arrive from its parent.

4. Implementation

The multiplayer game chosen for our implementation was DOOM. The game protocol proceeds in lock-step. Each player computes its state periodically and sends it to other players. The game state at a player's node advances only when he has received an update from every other player. 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 [6], 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 that are 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. Game packets are extracted from the received capsules, concatenated, and the capsule header appended. Every node performing aggregation maintains 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, typically 1 millisecond in our experiments. If all expected packets do not arrive within that period, the *cached* 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 weighted edges 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 instructing 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

In this section, we give theoretical arguments to show the benefits of using our architecture compared to existing peer-to-peer and client-server 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 would also be a measure of the total bandwidth consumption if all packet sizes were the same. In general, for naïve implementations, the bandwidth consumption is $O(n^2)$ for all the models. If a server uses interest management techniques to filter out irrelevant data, the bandwidth consumption drops to $O(n)$. This will also be true for peer-to-peer models if messages are sent

only to a subset of nodes that are in the region of interest of a node [7].

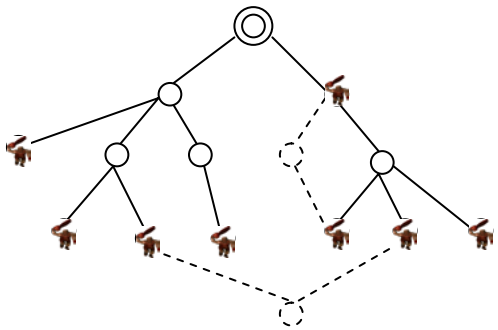

Figure 2. A multicast tree

On the other hand, network traffic as we have defined it is highly topology and routing table dependent, and we will not attempt a study of the complexity of the models in this respect. We shall demonstrate the difference between the three models through an example. Figure 2 shows a network of nodes, with a tree connecting the player nodes. The tree edges are marked as solid lines, whereas the other network edges are marked as dotted lines. (Note: 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 takes place along the shortest path between peers or from client to server. Table 1 shows the reduction in network traffic that dynamic multicasting achieves.

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

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

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 in our model 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 game packets are on 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 mean less work at routers, so the overall latency is reduced. For a large number of nodes, the packets can be aggregated only so long as they remain within a fixed size limit. As we saw earlier in this section, the bandwidth consumption upper bound is the same for all the models, but decreasing the number of packets will make a difference in the performance. As we will see in the following section, our experimental results will illustrate that peer-to-peer models are the worst in this respect, followed by client-server, with our dynamic multicast model being better than either of these. Another important point to be noted in our comparison is that a server in the client-server model would rarely be at the same place as the root of a multicast tree constructed by our algorithm because the server 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 toward ubiquitous gaming environments. Fault-tolerance is also enhanced. With small adjustments, this architecture can 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 for all players in most of the cases because of the central root location; we don't consider queueing effects at the routers. All packets pass through this root, ensuring that two players never perceive a widely inconsistent game state due to very different response times.

6. Experimental Results

In our laboratory, we have designed and implemented a prototype of the adaptation middleware, as described in Section 4. This middleware was deployed on HP Omnibook 4150 and Dell Inspiron 3500 laptops running Linux. The IPcept kernel module was used to perform transparent proxying and masquerading of sockets. (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 were connected by Ethernet cables.

We performed a variety of tests on the test bed described above. A number of network topologies of

active nodes were tried out, with some of them selected as game players. An ANTS utility called *makeroutes* was used to build the overlay network routing tables from the active network specifications; this user-level routing table functionality was also used extensively in our middleware for tree building.

We were interested in observing the performance of our system from two perspectives: 1) comparison with the traditional peer-to-peer and client-server models; and 2) feasibility of active networks as a platform for building this type of middleware. We chose network traffic to be a metric for the former, as described in Section 5. For the latter, we considered the time overhead incurred by the middleware; this also allowed us to observe how quickly the tree modified itself when required.

The technique used for comparing architectures was different from the one used to measure overhead. System overhead could be measured using the actual implementation on our test bed. Making network traffic comparisons would be meaningful only with a reasonably large number of nodes and an Internet-like topology; this was done by simulating the active network.

To measure the base cost, or overhead, of the middleware, we used a simple topology that directly connected game-playing nodes as described in Figure 3. Players on both machines played DOOM with one another for about 5 minutes, and the time taken to execute the adapter (ANTS) code at each client was recorded during every game step. The average overhead produced by the middleware code was observed to be approximately 4.1 milliseconds. But we also observed that about 93% of the overhead values were less than this average value. The readings were mostly in the range 1-2.5 milliseconds, with periodic bursts of a few tens of milliseconds to a couple of hundred milliseconds. Therefore it makes more sense to consider the median value, approximately 1.75 milliseconds, as the typical overhead introduced by the middleware at each client node. The reasons for the high variations will be explained shortly.

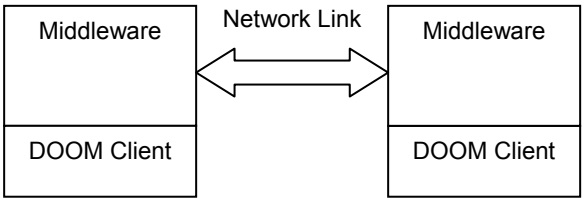

Figure 3. Simple topology for overhead

The numbers obtained above (~2 msec) also seem to agree with naked eye observations. 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. Slight jitter was obtained at periodic intervals, these points coinciding with the high overhead observations.

Observations for other topologies with a larger number of nodes were similar to the 2-node case. In a network of 3 nodes connected in a chain, with the end nodes being game players and the middle node being the root, the overhead at the player nodes was approximately 2.8 milliseconds on average, with a median value of 1.85 milliseconds, 92% of values being less than the average. The overhead due to the aggregation and duplication adapters at the root was approximately 2.3 milliseconds on average, with a median of about 1.5 milliseconds, 93% of values being less than the average. Variations in overhead similar to the 2-node case were observed at the client and root nodes, leading to the belief that the median, rather than the mean represents the typical overhead per game step.

The reason for the periodic spikes in overhead can be traced to the nature of the functionality that the adapters perform. Apart from processing and forwarding packets, network monitoring is done periodically by every active node. These occurrences of monitoring corresponded one-to-one with the instants at which high overhead was observed. We also simulated network condition change at certain intervals, leading to tree computation at the root, causing the overhead to rise to a couple of hundred milliseconds. We have implemented the monitoring code as a user-level application; moving this function to the kernel should reduce (or eliminate) spikes in overhead.

Another interesting observation was the smoothness of transition from one tree to another. Upon emulating latency change between nodes, we observed that the time taken to relocate the tree root for the topology in Figure 4 was a few hundreds of milliseconds, about 700 milliseconds in the worst case. As in the network monitoring case, game players observed some jitter.

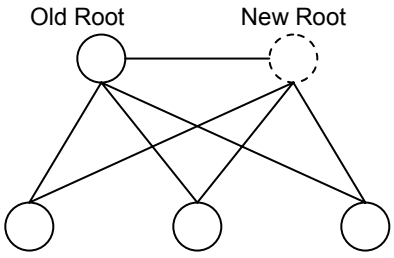

Figure 4. Topology for testing transition overhead. Bottom three nodes are players.

We must keep in mind that these observations were made in a LAN environment, where the average node-to-node communication latency was less than 1 millisecond. This only served to emphasize the difference between adapted and unadapted DOOM. 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 (an increase of 0.1 milliseconds from the 2-node topology to the 3-node one), this infrastructure promises to be quite scalable.

The other experiment performed was a simulation for comparison of network traffic, measured as the total number of packets seen by network links during a single round of message passing; as explained in section 5. Algorithms for the peer-to-peer and client-server models were implemented, in addition to our multicasting framework. In every experiment instance, the server for the client-server model was selected to be the same node as the root of the multicast tree. In addition to the network traffic, average player-to-player distance was also measured in order to compare the quality of paths in the multicast tree and the peer-to-peer model, which uses shortest paths for communication.

For simulation of networks and multicast groups, we used the Georgia Tech topology generator [19]. We generated four random undirected weighted graphs of 250 nodes each:

- TS1: A *transit-stub* graph – One transit domain with five nodes on average; each transit node has seven stub graphs on average; each stub domain has seven nodes on average.
- TS2: Another *transit-stub* graph – Two transit domains with five nodes on average; each transit node has six stub graphs on average; each stub domain has four nodes on average.
- R: A *random* graph generated using the Waxman model [3] – 1462 edges, with parameters values 0.1 and 0.2.
- H: A *three-level hierarchical* graph – Five nodes with edge prob. 0.4 at the highest level; five nodes on average with edge prob. 0.3 at the next level; ten nodes on average with edge prob. 0.6 at the bottom level.

All nodes of these graphs were considered active for the purpose of simulation. Edge weights represented physical distance, and if we ignore queuing effects at nodes, they could be considered as measures of inter-node communication latency. Multicast group size varied from 2 to 30, with a hundred random groups chosen for each size and the average reading taken. The comparison of network traffic and average player-to-player distance for the first transit stub graph (TS1) are shown in the figures below. The comparison between client-server and multicast models is also shown separately in figure 6 in order to obtain a better perspective. All data points are shown with 99% confidence intervals.

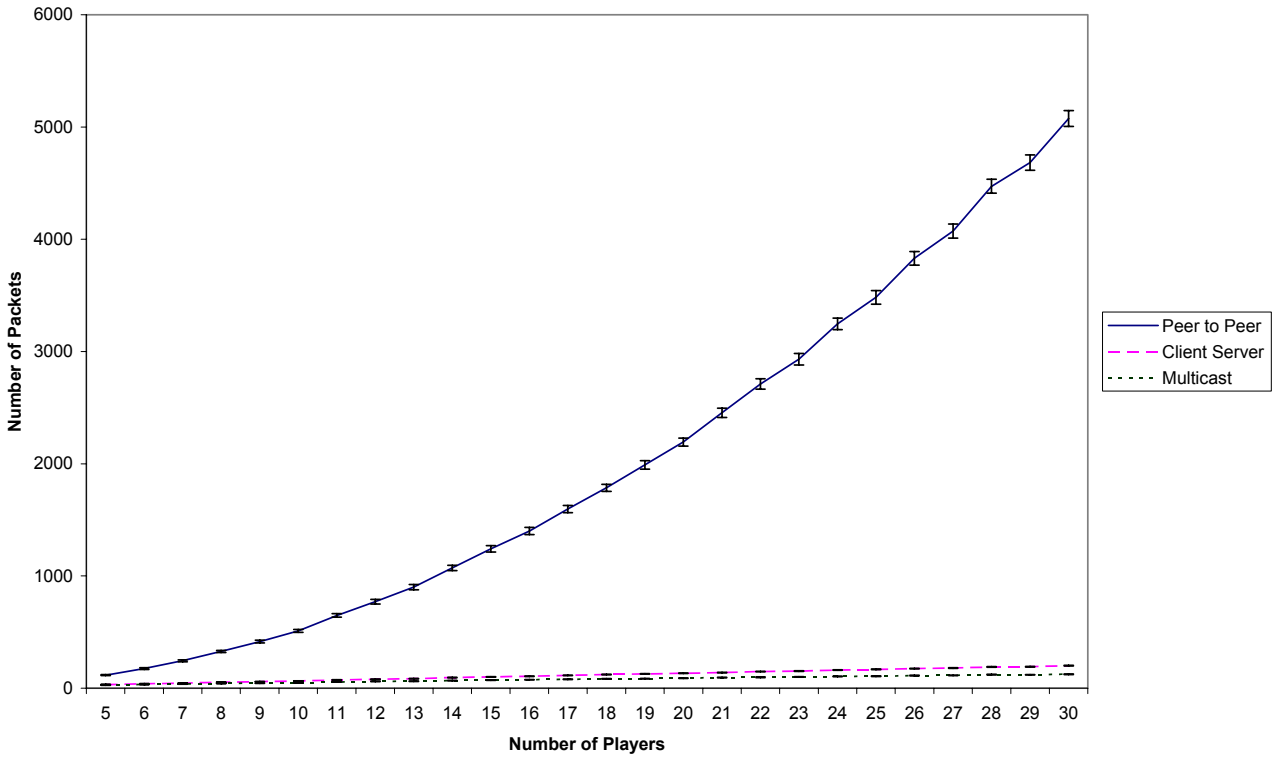

Figure 5. Comparison of network traffic for the three gaming infrastructures.

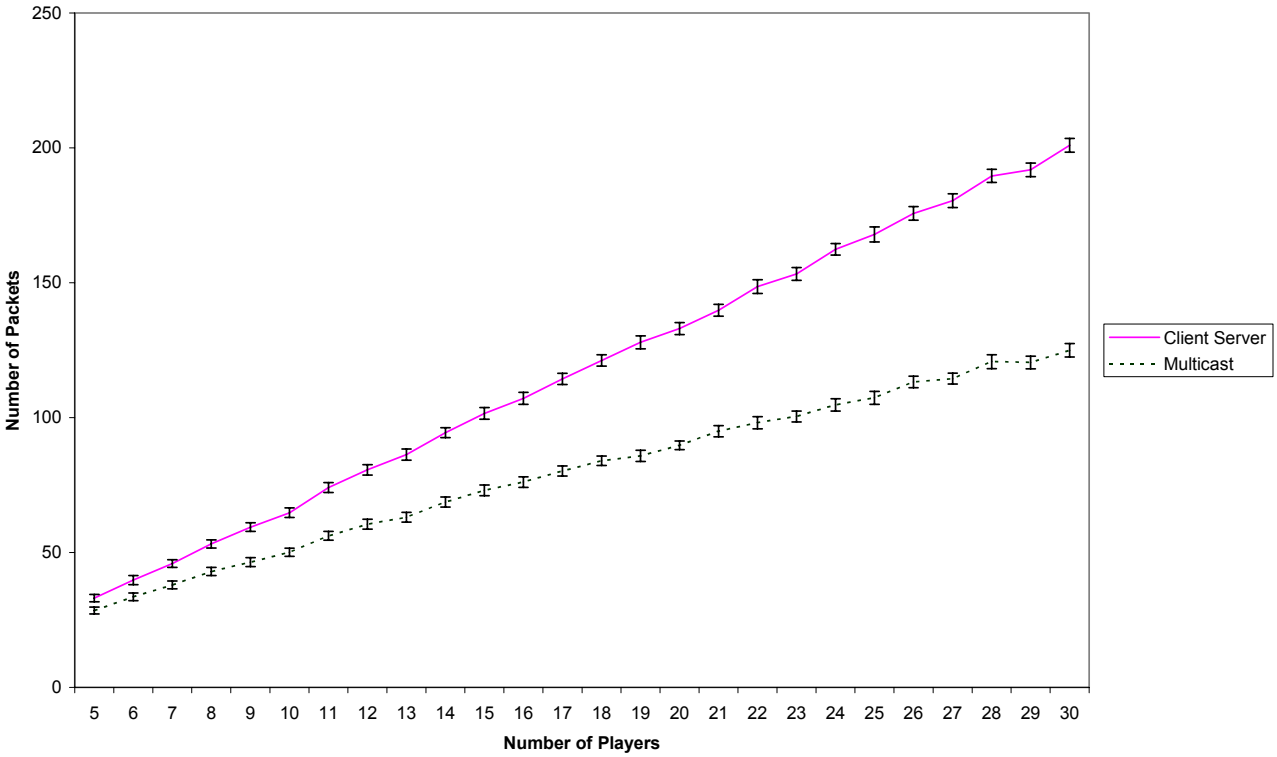

Figure 6. A close-up of network traffic for the client-server and multicast architectures.

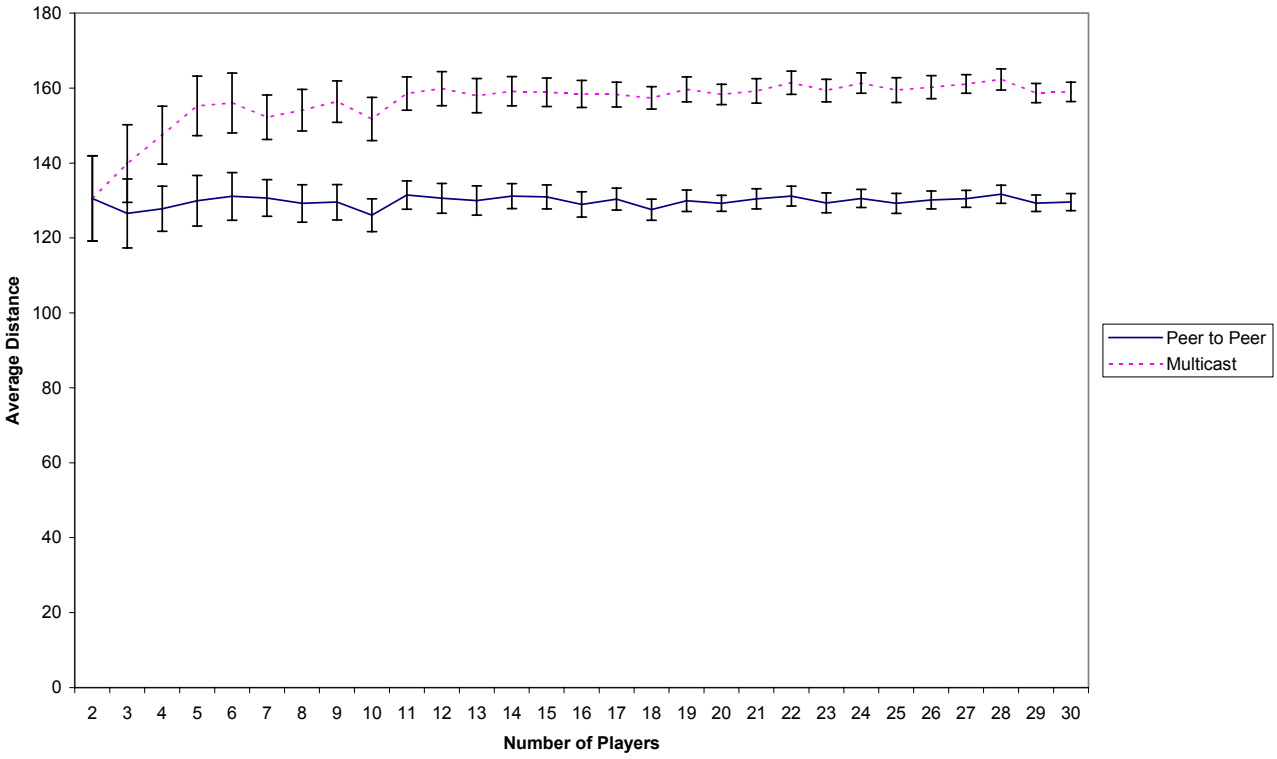

Figure 7. Average player-to-player distance, averaged over all pairs of players for each multicast group.

We obtained similar data for the three other graphs. Since the comparison charts for these were almost identical to the ones displayed above except for the scaling factor, we have not shown them in this paper. The reason for selecting a transit-stub graph was that it is representative of an Internet topology, unlike random or hierarchical graphs. The other data sets indicate that the observed trends are almost completely independent of the underlying network topology.

As we can see, figure 7 shows a comparison for only two of the models. This is because all data points for the client-server and the multicast cases coincided, the reason being that the server and the multicast tree root were always made to coincide. In the real world, we would expect to observe worse results for the client-server model, since the server is not likely to be located at an optimal position as the root in our multicast tree.

Figures 5 and 6 show that the dynamic multicasting model generates less network traffic than the client-server model and much lesser traffic than the peer-to-peer model; this difference constantly increases with increase in multicast group size. The 99% confidence intervals are extremely narrow, proving that the general trends indicated by the nature of the curves are highly accurate. Thus, processing at routers decreases significantly in the multicast case. These results would also be representative of bandwidth consumption, if interest management techniques are used. In real networks, a reduction in network traffic would translate to a reduction in communication overhead, since the chances of link congestion would be highly reduced.

From figure 7, we can observe that average player-to-player distance is equal for the peer-to-peer and the multicast models at group size 2, as expected. The difference continues to increase up to a group size of about 11. From 11 to 30, both curves can be considered almost flat, with a constant difference of about 30 distance units. These results are expected, as the peer-to-peer model generates shortest paths between all pairs of player nodes; this will be impossible to guarantee for a multicast tree since all paths must go through the root. On the other hand, the average distances for both the models are of the same order, and seem to remain constant with increase in group size. For a real network, these distances would also be, in most cases, measures of player-to-player communication latencies. Considering the huge gains in communication given by dynamic multicast over the peer-to-peer framework, a small constant difference in latency is a small price to pay. Also, it will not affect game-playing experience in any noticeable manner.

7. 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 on the different nodes or congestion along links, could be used to optimize the tree for communication.

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 we cannot demonstrate this here as DOOM does not support this facility.

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 then connecting him to his closest cluster.

8. 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. Our simulation results prove conclusively the benefits in communication overhead over traditional models. As our techniques are application-transparent, this model is applicable to both new and legacy games. We believe that a wide variety of multiplayer games will benefit by adopting this architecture. With currently available technology, a better option might be to implement a custom infrastructure, maybe at the network layer, in order to obtain better performance. Active networks, however, is still an evolving technology; we can expect that in a few years it will 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.

9. References

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

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

[3] B. M. Waxman, “Routing of multipoint connections,” *IEEE J. Select. Areas Commun.,* vol.6(9), pp.1617-1622, December 1988.

[4] D. S. Johnson, J. K. Lenstra, and A. H. G. R. Kan, “The Complexity of the Network Design Problem,” *Networks*, Volume 8, pp. 279-85, Winter 1978.

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

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

[7] 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.

[8] E. Cronin, B. Filstrup, A. R. Kurc and 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.

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

[10] 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.

[11] 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.

[12] K. L. Morse, L. Bic, M. Dillencourt, and K. Tsai, “Multicast grouping for dynamic data distribution management,” *Proc. of the 31st Society for Computer Simulation Conference*, 1999.

[13] L. Gautier, C. Diot and J. Kurose, “End-to-end transmission control mechanisms for multiparty interactive applications on the Internet,” *Proc. of IEEE Infocom 1999*, Volume 3, March 1999.

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

[15] L. H. Sahasrabuddhe and B. Mukherjee, “Multicast Routing Algorithms and Protocols: A Tutorial,” *IEEE Network*, pp. 90-102, January/February 2000.

[16] M. Maimour, J. Mazuy and C. Pham, “The cost of active services in active reliable multicast,” *Proc. of the Fourth Annual International Workshop on Active Middleware Services*, pp 67-74, July 2002.

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

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

[19] Modeling Topology of Large Internetworks – http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html.

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

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

[22] 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, October 1998.

[23] 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.

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

[25] 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.