Cluster Comput (2006) 9: 201–215
DOI 10.1007/s10586-006-7564-2

An active self-optimizing multiplayer gaming architecture

V. Ramakrishna · Max Robinson · Kevin Eustice · Peter Reiher

Received: October 2003 / Revised: May 2004 / Accepted: January 2005

Abstract Multiplayer games are representative of a large class of distributed applications that suffer from redundant communication, bottlenecks, single points of failure and poor reactivity to changing network conditions. Many of these problems can be alleviated through simple network adaptations at the infrastructure level. In this paper, we describe a model in which game packets are directed along the edges of a rooted tree connecting the players, aggregated during the upstream flight and multicast from the root to the leaves. This tree is constructed based on a heuristic, and can dynamically adjust itself in response to changes in network conditions. This gaming infrastructure is built and maintained using active networks, which is currently the only open architecture suitable for these types of applications.

We have designed and implemented a prototype using ANTS that performs these adaptations for unmodified DOOM clients. We present analytical and simulation 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 network-based middleware is acceptable, especially in wide-area networks.

Keywords Multiplayer games · Active networks · Self-optimizing middleware · Multicast · Packet aggregation

V. Ramakrishna · M. Robinson · K. Eustice · P. Reiher
Laboratory for Advanced Systems Research, Department of Computer Science, University of California, Los Angeles, CA 90095

V. Ramakrishna
e-mail: vrama@cs.ucla.edu

M. Robinson
e-mail: max@cs.ucla.edu

K. Eustice
e-mail: kfe@cs.ucla.edu

P. Reiher
e-mail: reiher@cs.ucla.edu

1. Introduction

In recent years, the computer gaming industry has exploded, creating billions of dollars of revenue, even rivaling the movie industry. Multiplayer games, or games that involve two or more human entities, usually enjoy the most popularity with users [26]. Although a large part of this industry in the past few years has been taken over by console-based gaming such as PlayStation, Xbox and GameCube, the network-based PC game market continues to dominate and expand. Millions of gamers around the world play *EverQuest, StarCraft* and *Counter Strike* everyday. Older games like *Quake* and *DOOM* provide the best user satisfaction when played on a LAN, and support, at most, tens of players. Nowadays massively multiplayer online role-playing games like *EverQuest* rule the roost; thousands of players can be supported in a gaming environment, with hundreds of such environments running simultaneously over the Internet.

A number of technical issues are involved in making multiplayer gaming a pleasant experience. Most of these games require that the entire game state, or a large subset of it, 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. Graphics and animation quality improve every year, and game environments become more detailed and intricate. Consequently, the amount of state information that needs to be transferred

among players keeps increasing. The network, therefore, becomes the performance bottleneck.

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 delaying communication [8, 10, 13]. Delivering only the essential data for an individual client has become an important research focus. On the other hand, 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 and 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 from the overall system perspective. 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. Also, a static topology causes the players to experience skewed response times. Nonetheless, this model is popular with game companies since it allows them to retain administrative control.

Though the Internet provides the universal connectivity that is essential for multiplayer games, it has some inherent drawbacks that create difficulties for game designers. No guarantees of packet delivery are offered, let alone quality of service. Network and end-host characteristics are heterogeneous. Broadband is getting common, and game companies can afford to buy more bandwidth to their servers, but the average game player is still likely to have a dialup or a low-bandwidth DSL connection. Experiencing variable bandwidth, high latency and significant jitter could make game playing a bad experience. These problems occur irrespective of the underlying system infrastructure. Also, 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, nor to the joining and leaving of players, which would significantly change the game network topology.

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 models; 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 low; as in the peer-to-peer case, latency is significantly reduced. The topology remains static though; 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 the peer-to-peer and the client-server models and eliminates many of their drawbacks. We construct a multicast tree connecting the players that has a low average player-to-player latency. This 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 periodically; when changes in network conditions are detected, the tree structure is changed and the root is relocated, with the aim being minimization of average player-to-player latency. All the nodes in the tree, including the players and intermediate nodes, participate in the building and monitoring of the infrastructure, though it is only the root that makes the decisions. This infrastructure is transparent to the application, so that the game need not be modified. It also enhances reliability and performance. Practically, such a framework would require a number of "intelligent" nodes distributed throughout the network that could be dynamically selected to be part of the tree at any point of time, in addition to "intelligence" at all player nodes. An actual deployment would require the cooperation of Internet Service Providers for placement of such nodes at strategic points in a network.

The main goals of our research are to enable the building of self-configuring and self-optimizing structures for the routing of packets between game clients. It is our aim to ensure that the number of packets containing game information transmitted through the network be minimal while also ensuring that the average time taken by a packet to move from one client node to another be as low as possible. Scalability is not a primary aim of our research. The basic approach we describe in this paper should scale for tens to hundreds of clients. Our approach, though, can easily be extended or adapted to create scalable infrastructures for massively multiplayer games; this topic is dealt with in a later section in this paper.

The techniques we describe here are not just applicable to networked games, but also to a larger class of distributed applications, of which games are a part. As we shall see, our methods are similar to those used to achieve interest management, or filtering and dispersion of relevant information, in distributed simulations [11] and publish-subscribe systems. A scheme that uses 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. Each node in such a network is "active", i.e. it can perform computation on a data stream, in addition to routing packets. Among open architectures, only active networks offers the flexibility and the range of functionality like dynamic code execution, new protocol deployment, creation of overlay networks, and support for load-balancing, all of which are necessary for the multiplayer gaming infrastructure that we have described above. In addition, active network-based frameworks like Panda [24] can adapt legacy applications with minimal effort and little or no modification.

This paper is organized in the following way. In section 2, we will describe current and former projects that are related in some way to ours, both as an adaptation middleware and as a gaming infrastructure. Section 3 contains a description of the design of our framework, and section 4 the implementation details of an ANTS-based prototype. In section 5, we will analyze the advantages and disadvantages of our system. Section 6 contains actual test and simulation-based results that will prove the conclusions obtained in section 5. Towards the end, we will describe different ways in which our design could be modified to support other kinds of games and distributed applications, and to provide added functionality. In the following sections, we will sometimes refer to our architecture as *dynamic multicast*.

2. Related work

Our work has been influenced by two systems previously designed in our lab. Panda [24] is an active networks-based 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. It possesses a security architecture as well as a reliability model. Where we significantly differ from both Conductor and Panda is that we perform infrastructural adaptations that can change the topology and routing behavior dynamically, rather than just adaptations of the data stream flowing through an end-to-end connection.

Various projects have used 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 has been investigated in [14, 23]. The feasibility of using active networks for multicasting has been investigated in [16]. Gathercast is a generalized network protocol that optimizes communication by aggregating packets that are flowing in the same direction; the result is far fewer packets, reducing the amount of processing that routers need to do. This protocol, which is similar to packet aggregation in our middleware, has been implemented using active networks [25].

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 adapts 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.

There has been a lot of work done in recent years on the design of gaming infrastructures that enhance performance. The MiMaze architecture [18] is a completely distributed peer-to-peer architecture that uses IP multicast for packet delivery, being built on top of the MBone, which is a wide area overlay of multicast capable nodes. It uses RTP/UDP/IP communication protocols and has support for recovery from packet loss. Synchronization mechanisms allow game participants to compute coherent game states at the same moment. Unfortunately, it suffers from many drawbacks: its topology is very static, scalability is limited and traffic reduction is suboptimal.

Cronin et. al. [7] have designed and implemented a mirrored server architecture for multiplayer games with a focus on providing increased scalability along with low latency packet delivery. To achieve low latency, the infrastructure relies on a custom-built reliable multicast protocol called CRIMP. The infrastructure, along with a description of the implementation of multiplayer Quake is described in [7]. Coherency in game state among clients is ensured by the trailing state synchronization protocol [8]. During bootstrapping, clients are allowed to locate the nearest game servers using two methods: i) a master server that locates candidate game servers, and ii) using a server selection service named *qm-find*. This architecture provides benefits like scalability and decreased latency, but has no mechanisms for fault tolerance; each of the servers is still a single point of failure. The network topology is also static and does not respond to changes in conditions.

The *ButterflyGrid* [28] project uses latest grid computing technology to provide an infrastructure that massively multiplayer games can run on. The system developers claim that this platform is highly scalable and dynamic; it can support potentially "unlimited" number of players in a game, and manage multiple games across the same resource base.

The Butterfly Grid is a completely distributed system, with decentralized control. The architecture is designed in an object-oriented fashion, with well-defined layers from the grid network up to the layer that recognizes and manages game objects. It is flexible, allowing game designers full customization of the network protocols that they need to use. The open source Globus toolkit that implements the Open Grid Services Architecture implements the lowest layer. The grid acts as a dynamic resource allocator for the infrastructure. Game play is instantiated by game servers that are responsible for certain regions. Servers that are part of the same game are connected using a multicast tree. In this respect, the infrastructure resembles a mirrored-server. Servers can be added and removed dynamically without interrupting game-play. On the downside, this platform requires wide-scale deployment of grid services. It is also somewhat tightly coupled with the game engine, and cannot support legacy games.

The WorldForge project [29] aims to build an engine for massively multiplayer online games that provides a much better gaming experience than currently available with games like Ultima Online and EverQuest. The architecture is based on the client-server model, with a WorldForge server managing player and non-player characters (in a role playing game), client interactions and world events. The focus here is not on building an optimized infrastructure, but rather on improving server performance and scalability. One of the advantages of this project, as most gamers will appreciate, is the flexibility to modify the clients, the software being completely open source; currently, this works only in a Linux/Unix environment. On the downside, the infrastructure does not attempt to address any of the problems that currently plague Internet games, as discussed in the section 1.

A common drawback to all these systems is that they require extensively remodeling of the game, or a completely new game design. Our framework supports unaware adaptations for legacy games; this has been our philosophy from the beginning, and one of the reasons for using active networks to build the middleware.

3. Design

One of the key aims of our design, and one which also is one of its biggest advantage, is that the game application is left unmodified to the greatest extent, ideally not at all. This does not preclude the active network-based infrastructure from being customized to the needs of specific games, and indeed, other kinds of applications. This infrastructure is designed as a middleware that physically lies beneath the application layer and above or parallel to the IP layer, depending on the implementation of the active network (currently, most implementations build application layer overlays). This middleware captures packets that are sent out by the game client, routes them through the network and queues them up to the application at the receiving ends. Logically, we can conceive of the infrastructure as having the following components: an overlay network with a multicast tree that routes packets, an algorithm based on some cost metric that is used to construct a tree connecting the players on a graph, and a monitor for detection of change in network conditions and connectivity. Each of these components can be designed separately and "plugged" in the framework as per the designer's requirements.

3.1. Game packet routing infrastructure

We connect all the game, or player, 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 because all game packets pass through this node.

Figures 1(a) through 1(g) illustrate how packets are routed in this model. Fig. (a) shows each player sending 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; we can observe this in Figs. (b) and (c). 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 along the edges of the tree; the first step in the multicast process can be observed in Fig. (d). Every intermediate node waits for packets to arrive from all its children. Packets that arrive early are cached; the root caches the packet from its right child in Fig. (c) while it waits for a packet to arrive from its left child. Received packets are duplicated at every branch node on the tree and sent along all downward links of that node, i.e., every link that is not an upward link. Figs. (d), (e) and (f) illustrate the multicasting. 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; we can observe this in Figs. (f) and (g).

3.2. Tree building and center location

As we have emphasized, minimization of communication latency is of paramount importance in a real-time game. The parameter that determines packet communication

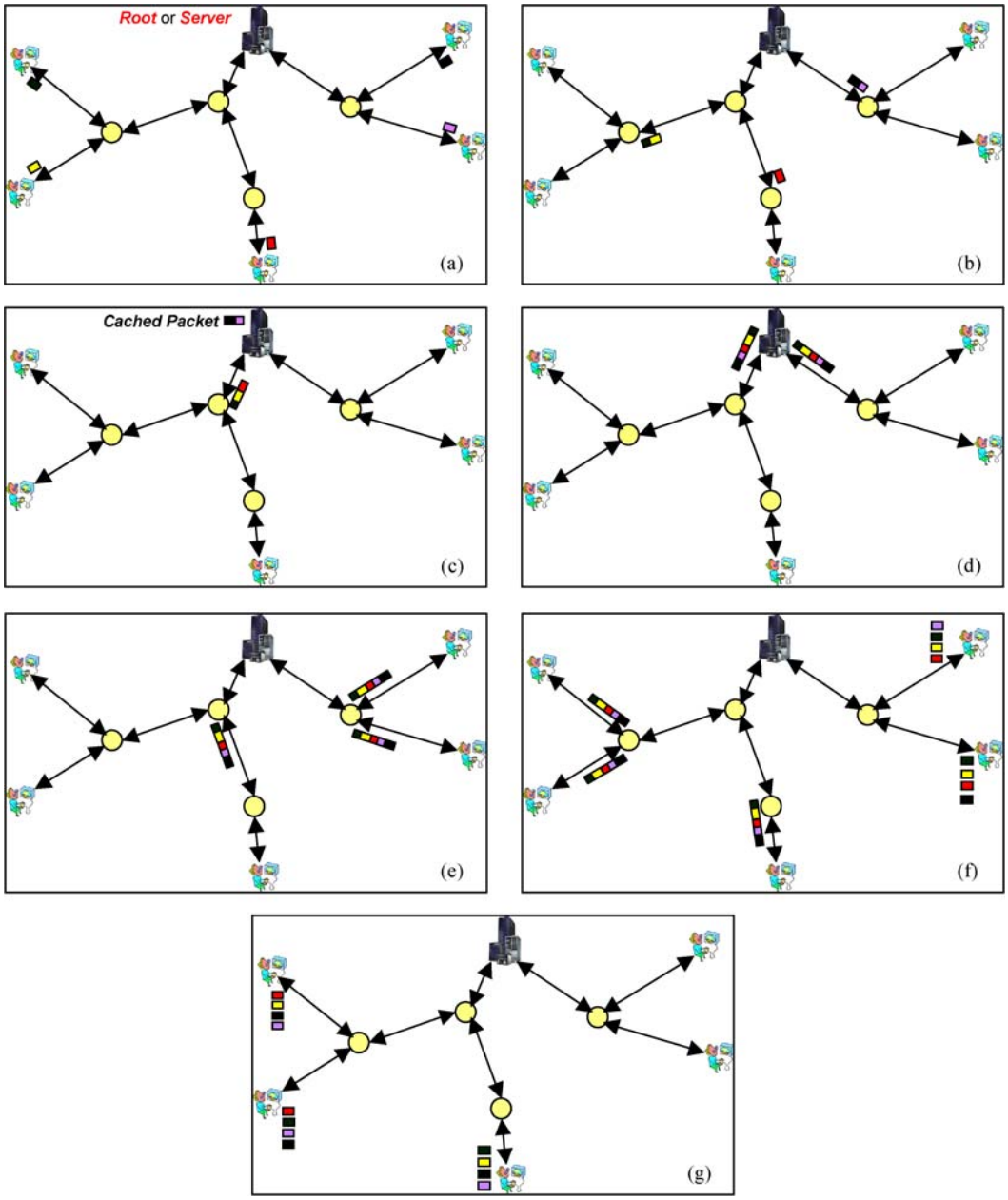

Fig. 1 Illustration of a single game step in an example multicast tree. (Note: A player can be an intermediate node also)

latency must be taken into account when constructing the tree connecting the players. There has been much research done in the area of multicast tree building, which we leverage for the purpose of selecting a suitable algorithm for our infrastructure.

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, allow any group member to be a traffic source and 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 the well-known NP-complete Steiner tree problem [4, 20]. Any practical algorithm for building multicast trees must involve tradeoffs between algorithmic complexity and the quality of tree produced. For example, 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 [22].

We use a centralized algorithm, based on Dijkstra's single source shortest path algorithm; this is run by a pre-selected root. Initially we have a network of nodes. In the general case, a subset of nodes will be active, but for the purpose of this discussion, we will consider only the overlay active network. We run Dijkstra's algorithm iteratively over the nodes of this network, each node being the source in a particular iteration. For each resulting tree, the cost can be calculated as the sum of edge packet communication latencies. The lowest cost tree among these is selected for the gaming infrastructure. A min-priority queue implementation of Dijkstra's algorithm implies that each of the n iterations runs in $O(n^2)$ time, n being the number of nodes. Therefore this algorithm terminates in $O(n^3)$ time. This complexity will not have a huge bearing on the performance of our gaming infrastructure, since construction or reconstruction of a tree is not expected to be a frequent event. It will occur only during bootstrap and when changes occur in network or game conditions, and this overhead is a price worth paying for the added reliability and fault tolerance.

Once the multicast tree has been built for the given set of game players, a *central* node is selected to be the root. This roughly implies that the average distance from every node in the network to the root must be minimized. The graph-theoretic definition of *center* fits our requirements perfectly, and we apply it directly to our algorithm.

First, we define some terms that 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. Our algorithm calculates the center of the tree based on this definition. If there are two candidates, one is chosen at random to be the root.

3.3. Network monitoring

The multicast tree infrastructure performs continuous network monitoring to detect change in conditions that might cause the current tree structure to become suboptimal with respect to average latency between nodes. Changes in conditions could be any of the following: addition or loss of connectivity between two nodes in the underlying network, congestion, failure of intermediate nodes, or players joining and leaving the game. When such a change is detected, the current routing structure may not remain optimal; in this case, 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 become 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 and assigned their new roles.

It is the responsibility of the root node to monitor network conditions, and also to execute the tree-reconstruction 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 this (overlay) 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 and is executed upon receiving an active packet. In our infrastructure, aggregation, duplication or deaggregation adapters are deployed at the nodes. Every node knows its immediate neighbors in the tree and maintains routing information. The node also has a set of *roles*, i.e., that of a player, a branch point or a monitor. It can take on one or more of these roles. It must also maintain 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 based on knowledge of roles and current game state, e.g., the node is waiting for a packet to arrive from its parent in the tree.

Using active networks as an application-independent platform enables the middleware to support multiple game sessions in parallel. A number of different game trees could run on a single overlay network simultaneously. Multiple packets of different game sessions could be aggregated, providing valuable savings in network bandwidth.

4. Implementation

For the purpose of evaluating our approach, we designed and implemented a prototype gaming infrastructure in our lab.

The multiplayer game chosen for our implementation was DOOM, a first person shooter game, and one of the earliest

ones of its genre. 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, this enabled us to examine the routing infrastructure in isolation from the other factors that could influence performance. This version of DOOM supports only a handful of players. This was sufficient for the building and demonstration of a prototype on our laboratory LAN, and for overhead measurements.

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. The ANTS execution environment was layered on top of the Janos NodeOS [27]. 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. These nodes communicate using ANTS packets, which are referred to as *capsules*. Initially, a static tree is 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. A computation of the best tree and root position could be done during the bootstrapping phase by the manually chosen root; alternatively, tree computation could be done by hand, since this step is performed only once. After the initial tree is chosen, the system behaves autonomously. Once the root is selected, all the active nodes in the vicinity that are interested in participating in the infrastructure must send registry capsules to the root. The root maintains at all times a list of nodes that could potentially be part of the multicast tree. Every other node maintains a list of pointers to its parent and its children.

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 routing the packet to all the other clients. When capsules containing game packets reach a tree branch point, they are aggregated into a single capsule and forwarded to the parent. 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 (the typical size of a DOOM packet is 16 bytes).

The value of the timeout period must be chosen carefully, taking application and typical network conditions into account. Large timeouts could disrupt smooth flow of the game, whereas small timeouts would reduce the number of aggregations. Having a timeout period is not mandatory, as it can introduce jitter, but is advisable in less than ideal network conditions. Timeouts also do not affect the synchrony of the underlying application as the middleware and the application are independent of each other. In our implementation, every DOOM client would simply wait for packets to arrive from all its neighbors before progressing to a new state.

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. No accumulation of state information takes place at any active node in the long term; old state (or role) information is just replaced by new information.

5. Analysis of benefits

In this section, we give theoretical and analytical 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 $O(n)$. Also, in the client-server case, the server must handle $O(n)$ messages, as compared to $O(b)$

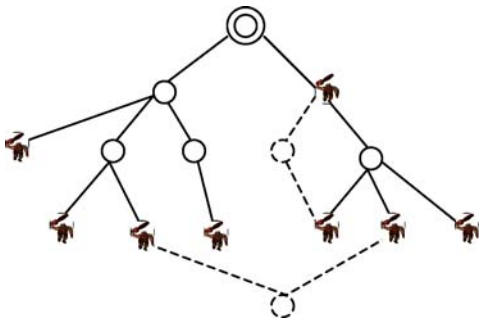

Fig. 2 Example multicast tree

messages by the root in dynamic multicast, where b is the branching factor at the root and $b \leq n$; asymptotically, the two orders are equal to $O(n)$. Analyzing a general mirrored-server architecture is difficult, since it depends on the amount of server replication. If the number of servers is small compared to the number of players, the behavior will be similar to the client-server model, with performance increasing by at most a constant factor; if the number of servers is relatively large, the behavior will be similar to a P2P network.

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 a 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].

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. Fig. 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.

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

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

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 multicast tree aggregation model should scale for a few tens or hundreds of players, but possibly not for massively multiplayer games supporting thousands of players as the aggregated packet size will become too large to escape fragmentation. The framework could be made more scalable by having multiple *roots*, each of which takes responsibility for a certain portion of the tree. Detailed investigation of this enhancement is beyond the scope of this paper.

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. Increased reliability 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 queuing 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

We have designed and implemented a prototype of the adaptation middleware 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 selected nodes in each network acting 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. We do not present a quantitative or measurement analysis of other features of our infrastructure, like aggregation, which has been investigated by He et al. [25].

The technique used for comparing architectures was different from that used to measure overhead. System overhead could be measured using the actual implementation on our test bed. 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 Fig. 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 the average. 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, as we can observe in Fig. 4. 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.

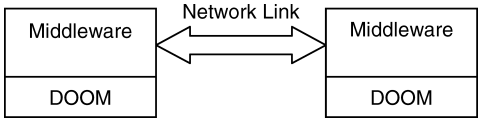

Fig. 3 Simple topology for overhead measurement

The numbers obtained above (∼2 milliseconds) seem to agree with naked eye observations. Hardly any difference in quality was perceived when DOOM was played over active networks; game-playing experience remained almost as good as it was without the middleware. Slight jitter was observed periodically, 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, as can be observed from Fig. 5. 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 Fig. 6 was a few hundreds of milliseconds, about 800 milliseconds in the worst case. As in the network monitoring case, game players observed some jitter.

We must keep in mind that these observations were made in a LAN environment, where the average node-to-node

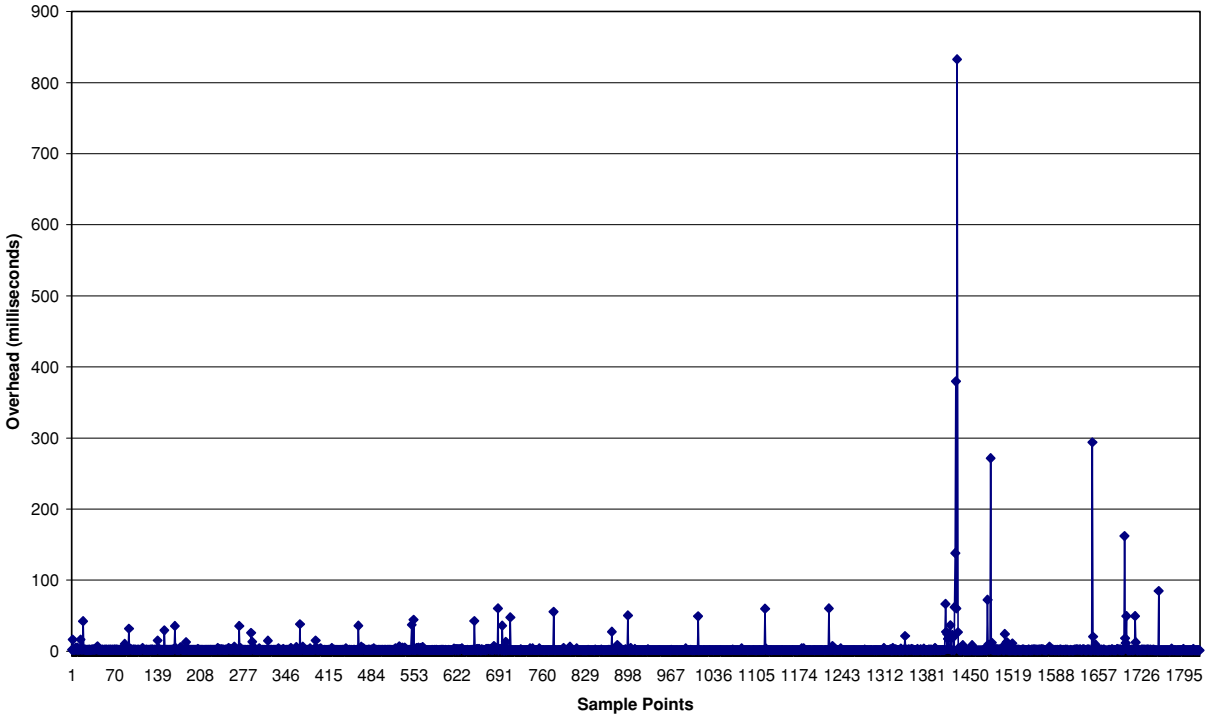

Fig. 4 Overhead due to middleware in the 2-node case.

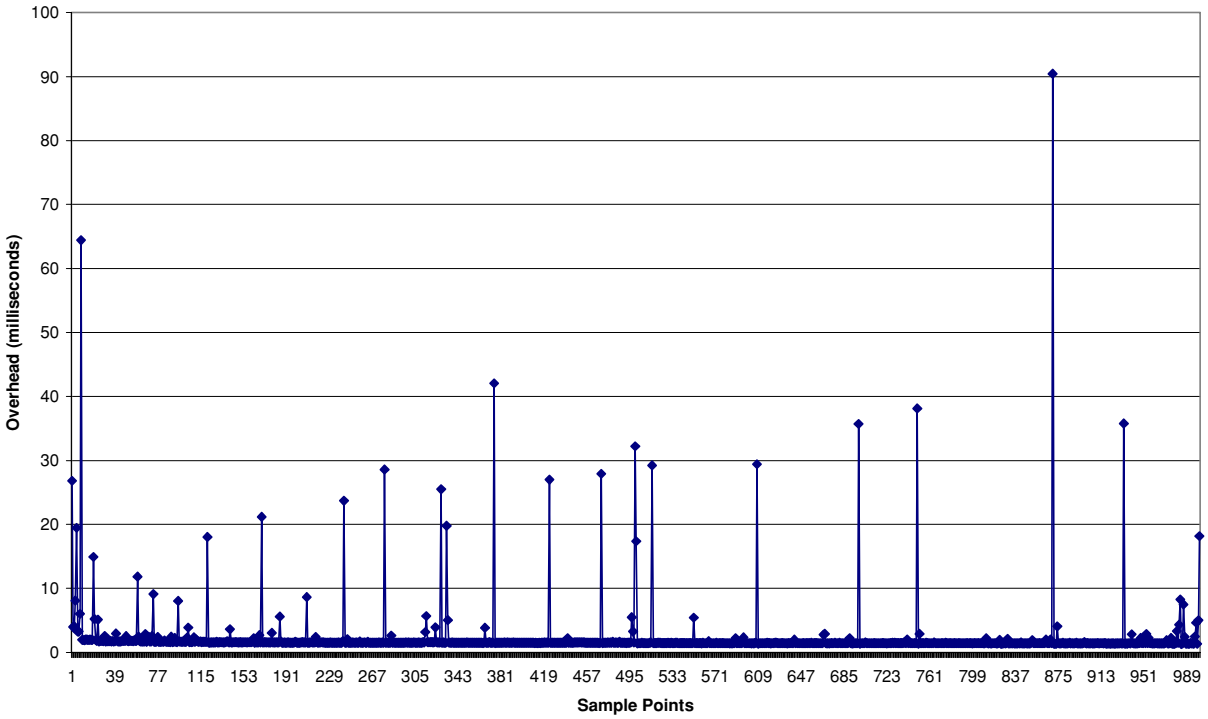

Fig. 5 Overhead at the root in the 3-node case.

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.

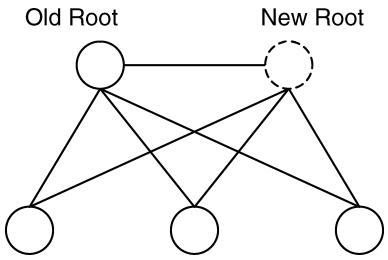

Fig. 6 Topology for testing transition overhead. Bottom three nodes are players.

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 (see 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; 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 Fig. 8 in order to obtain a better perspective. All data points are shown with 99% confidence intervals.

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, Fig. 9 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 7 and 8 show the comparison of network traffic among the three models. We can observe from Fig. 7 that the P2P model generates much more traffic that the other two models. Fig. 8 is a close-up of the comparison of client-server and dynamic multicast, which is not clearly observable in Fig. 7. We can observe from this figure that the dynamic multicasting model generates less network traffic than the client-server model. The difference between the models increases monotonically with multicast group size. The P2P curve appears to be quadratic, and the other two curves appear to be linear; this corresponds with the observations we made in section 5 regarding the complexity of the three models. 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 Fig. 9, 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 indicative of player-to-player communication latencies. Considering the huge gains in communication given by dynamic multicast over the peer-to-peer framework, a small

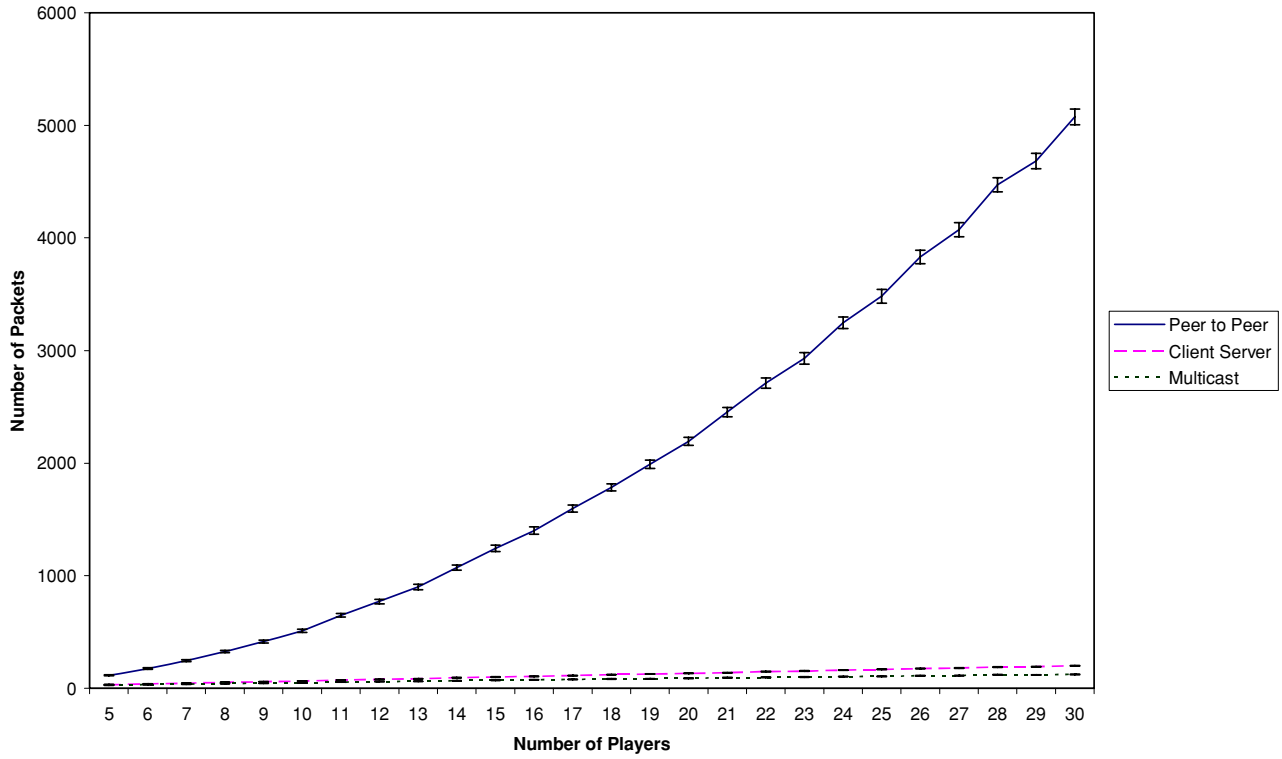

Fig. 7 Comparison of network traffic for the three gaming infrastructures.

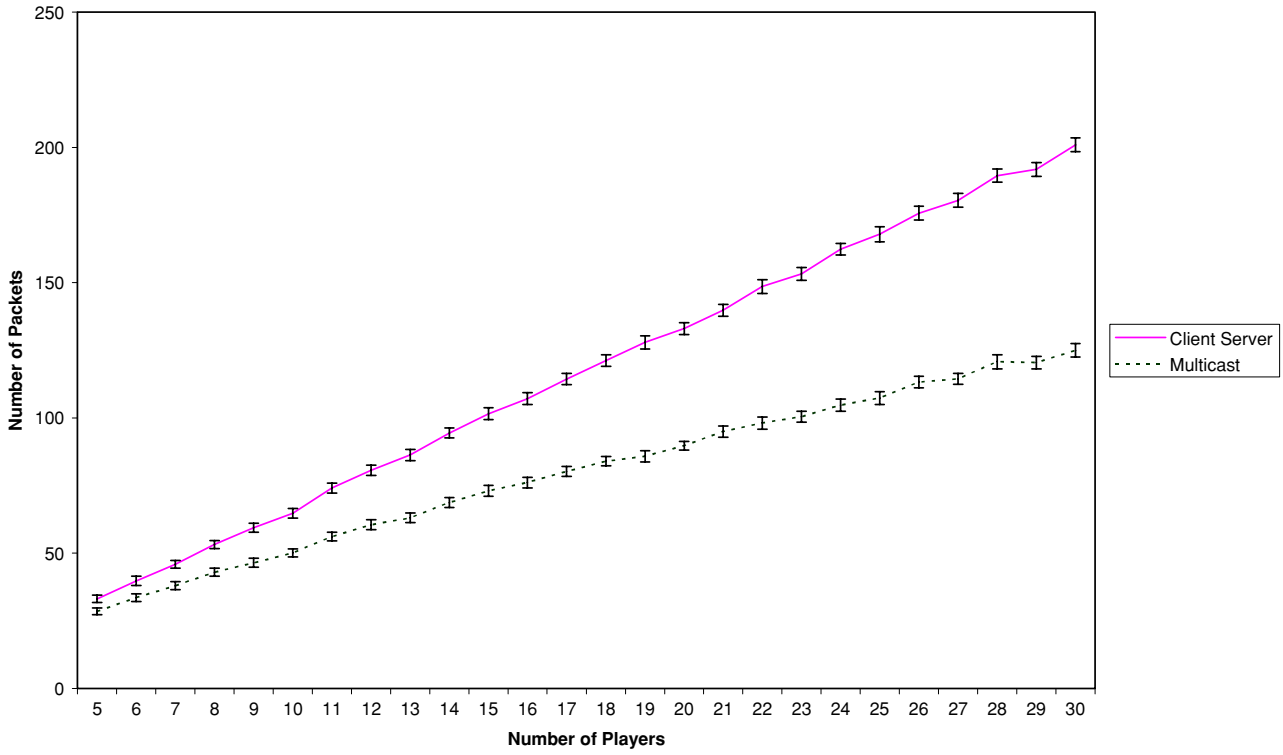

Fig. 8 A close-up of network traffic for the client-server and multicast architectures.

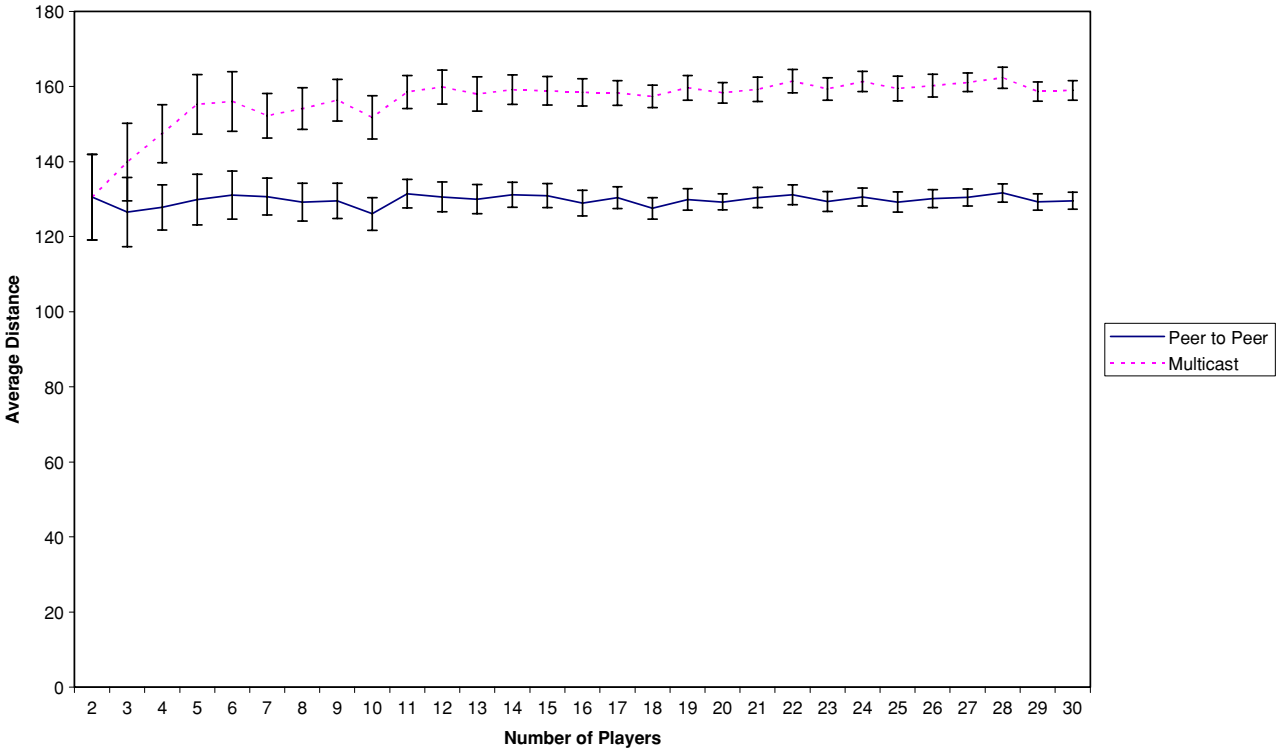

Fig. 9 Average player-to-player distance, averaged over all pairs of players for each multicast group.

constant difference in latency is a reasonable price to pay. Also, it will not affect game-playing experience in any noticeable manner.

7. Future work and possible extensions

There are various directions in which our system could be extended or modified to meet the needs of other applications. Some of our ideas are mentioned briefly in the following subsections.

7.1. Tree-building metrics

Our current system uses the simple link latency metric for building multicast trees. Other metrics could be used for building trees: link congestion could be monitored, with heavily congested links getting lowest priority for selection in the tree; load-balancing could be a criterion, with heavily loaded nodes being eliminated from trees, or assigned roles that involve less work on their part.

7.2. Fault tolerance

We have shown how our system can be tolerant to failures in the underlying network. Depending on topology, failures may not even produce degradation in service, since the system strives to build the best possible tree at all times. Still, a lot more needs to be done before our infrastructure can be completely fault tolerant. If any of the active nodes, especially the root, fail for some reason, the game could stall. There are simple techniques that we could use to overcome this. Nodes already monitor links to their neighbors; it would be easy to find out if a particular node becomes unreachable. If it is any node other than the root, reconstruction of the tree would be straightforward, based on our system design. Failure of the root is more serious, and is discussed in the following subsection.

Players joining and leaving could be mapped to the node failure case, and can be dealt with in the same manner. Our architecture supports this, but we cannot demonstrate this here as DOOM does not support this facility.

7.3. Replicated server architecture

The current reliability of the system can be enhanced by replicating the monitor at multiple sites, in effect having a multi-rooted tree (somewhat similar to a mirrored server architecture). If one monitor fails, others will take over and obtain a new position for the root. This enhancement has the added advantage of making the infrastructure more scalable. These roots would ideally be distributed uniformly in the game network so that each has to handle roughly equal load.

An enhancement to this model could be to distribute the roots throughout the game's virtual space, with a neighboring root taking over a larger virtual area in case of failure.

Intermediate nodes could also perform filtering and interest management, thereby reducing data communication.

7.4. Cluster-based infrastructure

With wide deployment of active networks, independent game clusters could be built based on node proximity. Active nodes could coordinate among themselves and form groups based on proximity. They could elect roots, and admit player nodes if the communication latency to those players fits the game-playing constraints. These clusters could be formed without any manual intervention, with the tree roots deciding whether to admit a new player, and then connecting it to its closest cluster. If a node cannot be admitted, its request could be forwarded to another cluster that might admit it; alternatively two or more free player nodes could be hooked up with each other to form an ad-hoc cluster.

7.5. Adaptation of client-server games

Adapting a legacy client-server game while retaining the advantages of a dynamic topology may pose some challenges, depending on the kind of game. If the game server is logically and physically separate from all the clients, it could be made to run on the same physical location as the root of the multicast tree. The rest of the architecture remains the same. Failure of the root will be more serious in this case, since it handles all state information. Server replication, as discussed in section 7.3, is probably the most appropriate solution for handling of root failure. Adaptation of any mirrored-server based game will proceed along the same lines.

For a game like Quake, especially the earlier versions, one of the players is expected to be the server. Therefore, if the server has to be co-located with the tree root at all times, the root must necessarily be static, since a player node cannot relocate. Alternatively, we could treat the server as just another client, and use the same architecture that we employ for DOOM. The resulting routing framework may or may not provide traffic reduction, depending on the extent of aggregation performed in the tree.

7.6. Publish-subscribe applications

A lot of work has been done in building infrastructure for distributed applications that have participant nodes *publishing* updates about new events, and other nodes *subscribing* for those updates. Without a routing framework, there will be a lot of redundant communication, and there would be no guarantee of a node ever being informed about the occurrence of a particular event. By using a framework based on our design, all the interested parties could be connected by a multicast tree that guarantees message delivery in the shortest possible time. Active networks could create a lot of possibilities in the publish-subscribe area; this is a promising topic for future research.

8. Conclusion

The most significant contribution of our research is an infrastructure for multiplayer games that reduces the number of communication channels by virtue of a tree structure and eliminates duplicate data transfer, with a marginal increase in latency. Additionally, this architecture is self-adjusting and can be deployed on both local and wide area networks. Our simulation results illustrate the reduction in communication overhead versus more traditional models. Active networks clearly provide a substantial benefit to gamers when used to perform network-level adaptations. Additionally, the application-transparent nature of our technique allows it to be applied to both new and legacy distributed applications. In the past, game designers concentrated on building better applications without regard to the underlying system and networking characteristics; we have shown that tighter coupling with the network infrastructure will enhance flexibility, functionality and have a positive effect on performance. Many existing multiplayer games would benefit by using this architecture; however in the absence of widely deployed active networks, our model would provide more efficient service when integrated into the design of multiplayer game infrastructure. As active networks mature, it will be possible to leverage their computational power to enable the deployment of an adaptation model such as the one described herein.

Our approach is not restricted to the gaming world; it will also benefit a wider class of applications like distributed simulations. Since the only drawback of our system is the overhead, or the latency, the performance benefits should be more apparent in non-real-time applications than in multiplayer games.

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, 6(5) (1998) 334–348.
3. B. M. Waxman, Routing of multipoint connections, IEEE J. Select. Areas Commun., 6(9) (December 1988) 1617–1622.
4. D.S. Johnson, J. K. Lenstra, and A. H. G. R. Kan, The Complexity of the Network Design Problem, Networks, 8 (Winter 1978) 279–85.
5. D. Tennenhouse and D. Wetherall, Towards an Active Network Architecture, ACM Computer Communication Review, Volume 26(2) (April 1996) 5–18.
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* (2002) pp. 67–73.
9. F. Adelstein, G. Richard III and L. Schwiebert, Building Dynamic Multicast Trees in Mobile Networks, *ICPP Workshop* (1999) pp. 17–.
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)* (1998) pp. 3–23.
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* 3 (March 1999).
14. L. Lehman, S. J. Garland and D. L. Tennenhouse, Active Reliable Multicast, *Proc. of the 17th INFOCOM* (March 1998) pp. 581–589.
15. L.H. Sahasrabuddhe and B. Mukherjee, Multicast Routing Algorithms and Protocols: A Tutorial, IEEE Network (January/February 2000) 90–102.
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* (July 2002) pp. 67–74.
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. L. Gautier and C. Dior, Design and Evaluation of MiMaze, a Multiplayer Game on the Internet, in *Proc. of the IEEE International Conference on Multimedia Computin 9 and Systems* (1998) pp. 233–236.
19. Modeling Topology of Large Internetworks – http://www.cc.gatech.edu/fac/Ellen.Zegura/graphs.html.
20. R. M. Karp, *Complexity of Computer Computations*, Plenum: New York (1972) pp. 85–103.
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* (July 2002) pp. 35–40.
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* (July 2002) pp. 61–66.
26. Game Research - http://www.game-research.com/art_myths_of_gaming.asp
27. P. Tullmann, M. Hibler and J. Lepreau, Janos: A Java-oriented OS for active network nodes, IEEE Journal on Selected Areas in Communications 19, 3 (March 2001) 501–510.
28. IDC, White Paper - Butterfly.net: Powering Next-Generation Gaming with Computing On-Demand, *e-business Case Study*, http://www.butterfly.net/platform/technology/idc.pdf
29. World Forge project – http://www.worldforge.org