Supporting Spectators in Online Multiplayer Games

Ashwin R. Bharambe*
Carnegie Mellon University

Venkata N. Padmanabhan
Microsoft Research

Srinivasan Seshan
Carnegie Mellon University

ABSTRACT

We examine network support for spectators in online multiplayer games. We consider how the associated challenges are different from those in traditional audio/video streaming. Specifically, we consider two problems: *spectating* (distributing the gaming stream reliably to spectators, who may outnumber the players by several orders of magnitude) and *cheering* (delivering the spectators' audio cheers to the players as well as to other spectators). We point out the many unique challenges and opportunities for optimization that arise in this context in terms of resilience, bandwidth adaptation, and dynamically varying user interest. We outline a solution based on overlay networking and quantify some of our design arguments with a preliminary evaluation of Quake III, a popular first-person shooting game.

1. INTRODUCTION

Spectator-mode gaming, or *spectating*, has emerged as a popular paradigm for online multiplayer gaming. Spectating allows users to watch the proceedings of an online game akin to how television allows people to watch real-world games and sports. The number of spectators can be in the thousands or more, i.e., several orders of magnitude larger than the number of players. Popular games that either support or have been extended to support spectators include Half-life and Quake.

In this paper, we consider network support needed for spectator-mode gaming. The basic question is how to deliver the gaming stream to a large, dynamic, and heterogeneous population of spectators in a reliable and cost-effective manner. On the face of it, this problem is akin to that of distributing other streaming content (e.g., audio/video) to a large client population. However, spectating differs from audio/video (A/V) streaming in a number of ways that presents unqiue challenges as well as opportunities for optimization.

First, the game view can be disaggregated into its constituent entities, which can be treated differentially based on their importance, to optimize the use of the available bandwidth. Such differential treatment is far more fine-grained than differentiation at the granularity of layers in the A/V context. Second, the game state is persistent, which allows masking lost updates but also requires reliable delivery of certain updates (e.g., entity creation and deletion). Third, it is quite likely that there will be multiple camera views of interest, given the natural fit with the gaming context (e.g., each player's view of the game) and the ease with which the views can be generated. Taken together, these characteristics of spectating enable a rich and flexible set of strategies for bandwidth optimization, ranging from adjusting the frequency of updates to adjusting the field of view.

In addition, spectators need not be passive viewers like a TV audience. They can actively "cheer" the players, just as in a real game or sporting event. While such *cheering* can take different forms, audio is perhaps the most natural form, and is the focus of our discussion here. At first glance, cheering might appear to be no different from "conference" mode (as opposed to "lecture" mode) in the A/V context. However, cheering differs in a few key ways. First, there is typically no floor control. Multiple members of the audience are typically speaking at the same time, and what the players hear is the superposition of the individual cheers, just as in a sports stadium. Second, since the camera views of each spectator may be different, it might be natural to deliver their cheers only to the players that they are viewing. Finally, while cheering introduces a delay constraint on the spectating-cheering feedback loop, this constraint is typically much less stringent than that needed to support an interactive conversation in an A/V conference.

In this paper, we focus on the client-server game model, which is the basis for most online games. Although the server may be able to support a dozen or so players, it is unlikely to have the bandwidth to forward the spectating stream to or receive the cheering streams from thousands of spectators. Overlay networking is therefore an attractive solution. The spectating stream flows down overlay multicast trees while the cheering streams flow up the trees. We employ a combination of striping across multiple trees and in-network filtering and aggregation to accomplish multiple goals, including resilience, load balancing, multiple camera views, bandwidth adaptation, and avoiding an implosion of cheering input at the root.

In the remainder of this paper, we elaborate on the characteristics and requirements of spectating and cheering. We then discuss various strategies for delivering the spectating stream to clients. We quantify some of our design arguments with a preliminary evaluation of Quake III, a popular online game. We also discuss the question of defining an appropriate metric to quantify the quality of a spectating stream.

2. BACKGROUND

There are three main categories of online games: role-playing games (RPG), real-time simulations (RTS), and first-person shooters (FPS). The genre of a particular game heavily influences the

*The author was an intern at MSR during part of this work.

design of spectating support for it. For example, critical factors such as the number of unique views of the game world, the rate of game updates, the size of the game world, the spatial distribution of activity, the number of active players, and even typical deployment patterns are largely dictated by a game's genre. In this section, we describe the relevant properties of each genre and provide background about Quake III, the game that we use in our preliminary tests.

While each genre has unique properties, there are some important common aspects to all games. In most games, the player controls one or more entities (characters, units, etc.) within a 2- or 3-D virtual world. The game is responsible for determining the actions of a collection of computer-controlled entities as well as resolving the results of any game-play interactions. While other architectures have been employed, most current games rely on a client-server design, with servers run by the game vendors (common for Role Playing Games (RPGs)) or by one of the players (common for Real-Time Simulation (RTS) and First-Person Shooter (FPS) games). Finally, each game has a unique rendering engine that generates a display of the relevant game state. Below, we describe how the genres differ in other aspects.

In RPGs (e.g., Everquest), 10s to 1000s of players interact in a relatively large virtual world. Individual players can join the game world at any time. Both player characters and the game world are persistent – i.e., changes persist across individual game sessions. This forces any spectator system to efficiently download the current state to the spectator. Action in these games is quite slow, with player movements and the game world updates occurring at most a few times per second. While servers are able to host a large number of players, spectators may be more difficult to support. For example, players are typically given a view of their surroundings but spectators may be given an overview of the entire game world as well as detailed views of many smaller regions of the game world. One of the key challenges is supporting the large number of different views that the collection of spectators may demand.

Unlike RPGs, RTS games (e.g., Age of Empires) do not involve persistent state. All players join before the game begins and the game session lasts until there is a winner. Because of this synchronous start requirement, only a small group of individuals (at most 30 players) play in a particular game session. Usually, in many RTS designs, the game server simply acts as a message relay and clients typically keep a copy of the entire game state. This is possible due to low bandwidth requirements of the game, resulting from the limited number of players, the limited size of the game world needed for the low player counts, and the low update rates typical of RTS games. Spectators can be easily supported by providing them with a copy of this low bandwidth stream, which would be sufficient to generate arbitrary views of the game world.

FPS games (e.g., Quake, CounterStrike) are much more arcade-like than either RTS games or RPGs. Action in the game is much faster and game state updates are much more frequent. Most sports games fall into this category. FPS game players can freely join and leave a particular game session. Most FPS games limit play to a small number of players (about 30) either to minimize overhead at the server or due to game play limitations. In our initial studies of spectating, we use Quake III, a popular FPS game. Like most modern FPS games, Quake III uses a central server architecture. The server is responsible for maintaining the state of the virtual world and all entities within it. Clients connect to a server and the server continuously transmits the current state of entities within the current view of the client. To reduce the bandwidth requirement, each entity update transmission is delta-encoded against the previous state that the client has already received. Like RPGs, spectators may be given multiple views of the game. However, while the size of the game limits the number of interesting views, the high update rates make each view more difficult to support. Although we use a FPS game in our preliminary design, our goal is to support all genres of games. However, certain aspects of our design may be most applicable to particular genres.

3. SPECTATING CHALLENGES

In this section, we discuss the challenges in delivering a gaming stream to a large, dynamic, and heterogeneous population of spectators in a reliable and cost-effective manner. We consider the virtual game world divided into multiple areas or *game views*. A game view consists of a set of game objects called *entities*, each of which has various *attributes* (e.g., location) An update of the game view sent to the spectator is called a *frame*.

3.1 Differences compared to A/V streaming

While spectating shares much in common with A/V streaming, it differs in a number of ways:

Scene disaggregation: Each frame of the game view can be disaggregated into its constituent entities. The entities, and possibly individual attributes of the entities, can be treated differentially with regard to the reliability, granularity, and timeliness of updates, which provides opportunties for reducing bandwidth usage without compromising the user-perceived quality of the spectating stream. For instance, the position of the cars in a racing game is likely to be far more important than changes in the appearance of the cars say due to bumps and scratches. While A/V streams can also support some differentiation (e.g., base-layer versus enhancement-layer bits in a layered codec), it is hard to perform the entity-level disaggregation that is possible with game streams.

Persistence: The game state maintained by the spectating client is persistent (i.e., entities continue to exist until they are destroyed). A spectating client can mask lost updates for an entity by re-using or extrapolating the entity's state from a previous update. Such *selective* patching at the granularity of entities would be hard to do for an A/V stream, where the entire frame may have to be repeated instead. However, persistence forces entity creation and deletion messages to be delivered reliably to all spectators. For instance, a new car could enter a race or the bumper could break off a car, creating a new entity. A/V streams deal with this issue in a bandwidth-inefficient way — by repeating the *entire* scene continuously.

Multiple camera views: Multiple camera views (e.g., the views from each player's perspective) of a gaming session are easy to generate and are of great interest to spectators. Multi-

ple views are especially useful in a game world that is too large to fit within a single view. Overlapping camera views could save bandwidth by sharing substreams. Although there might be multiple camera views even in an A/V context, this scenario is faced with greater hurdles (e.g., the need for actual cameras). Also, the difficulty in disaggregating the scene makes it hard to save bandwidth across overlapping views.

Flexible strategies for bandwidth adaptation: Bandwidth adaptation is important in both the spectating and A/V settings to deal with bandwidth heterogeneity and dynamic congestion. However, the spectating setting provides a richer set of adaptation possibilities given the scene disaggregation property and the availability of multiple camera views. For instance, a user could choose to receive less frequent updates for some or all entities, or instead switch to a more narrow camera view.

There are other spectating-related issues that, while important, are not our focus here. For instance, the ability of spectators to switch between camera views opens up the possibility of players impersonating as or colluding with spectators to cheat. Delaying the spectating stream can alleviate this problem.

3.2 Metrics

The quality of the spectating experience depends on how faithfully the received stream reproduces the true scene. In the A/V context, the *peak signal-to-noise ratio* (PSNR) is a commonly used metric. PSNR is a function of the *distortion*, i.e., the difference between the *bitmaps* rendered at the source and receiver.

One end of the scale of possibilities is to use the PSNR of the rendered bitmap itself as the metric of goodness for spectating. The main advantage of this metric is that it is game-independent and builds on the existing work on PSNR in the context of video. However, a key limitation of this metric is that it fails to account for the differential importance of various entities in a gaming scene. For instance, in a car racing game, getting the positions and speeds of the cars right may be far more critical than getting details of the background right. Also, the relative positions of entities (e.g., race cars) is perhaps more important than their absolute positions.

Another way to address the above issues is to define a game-specific metric that is cognizant of the relative importance of entities and groups of entities. While such a metric could well be very accurate, it suffers from the drawback that it is tied to a specific game, hence making it difficult to compare the spectating experience across games.

The compromise we advocate is a *weighted* PSNR metric, which uses a PSNR-like calculation on the rendered bitmaps but incorporates an *importance* value (weight) with each bit that reflects its importance to gameplay. For example, in a racing game, the car bitmaps would have higher weights than the background. While it is not as powerful as a game-specific metric in capturing aspects such as the relative position of entities, we believe that weighted PSNR strikes an acceptable balance between accuracy and generality.

4. DESIGNING A SPECTATING SYSTEM

We now outline our design of a spectating system that addresses the issues discussed in Section 3.1. Our focus is on discussing the unique and novel aspects of the solution rather than describing the complete solution.

Our solution relies on end-host-based overlay multicast, an approach pioneered by [4]. For our discussion here, we assume that the set of players is represented by a single "source" node, although our design can easily be adapted to the case where there is no such single point of aggregation. We construct a set of overlay multicast trees, each rooted at the source and spanning the set of interested spectators. The spectating stream is then distributed over these trees, with each spectator node replicating the stream to its children. The reason for employing multiple trees rather than a single one is not only the improved resilience [9] and load distribution [3] that this provides but also the flexible bandwidth adaptation that this enables.

We now discuss the alternatives for the client software architecture, delta-encoding, bandwidth adaptation, and reliable message delivery, in turn.

4.1 Client Software Architecture

The client software architecture we advocate is a game-specific spectating client with a generic communication library. The spectating client would typically be derived from the players' version of the game and, hence, would include all of the game-specific optimizations such as fast physics and rendering engines. The generic communication library, on the other hand, would provide support for aspects that are common across all games — delta-encoding, reliable message delivery, and bandwidth adaptation. While the alternative of having a generic game-independent spectating client (such as Windows Media Player for A/V streams) has its advantages, it would typically be less bandwidth-efficient and be unable to leverage game-specific optimizations.

4.2 Distributed Delta Encoding

Delta encoding is motivated by the observation that typically only a small number of entities change from one game update (i.e., *frame*) to the next. A similar observation is leveraged in the A/V streaming context by encoding frames as differences with respect to other frames (e.g., the P and B frames in MPEG). However, the scene disaggregation and persistence properties of game spectating make delta-encoding far more efficient — only entities that have been updated with respect to the reference frame need be transmitted, and furthermore it may be possible to condense these updates further with game-specific knowledge. Note that current games do incorporate delta encoding protocols for game play updates. However, our goal in this section is to demonstrate the importance of this mechanism and to illustrate the difficulties involved in effectively supporting it in a spectating context.

The precise scheme for delta-encoding a frame is as follows: first, the sender maintains a fixed-size history buffer of frames the receiver has received. When transmitting a new frame, the sender first identifies the subset of entities that have changed since the last frame. Each changed entity is encoded with respect to the frame (termed the *base* frame) that provides the minimum delta-encoded size. Figure 1 shows the impact of history size on the the mean bandwidth (over a 5-minute session) of a Quake III game data stream for various number of com-

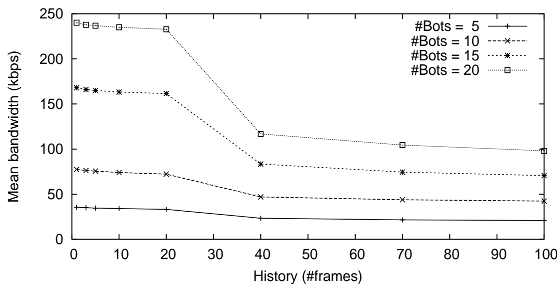

Figure 1: Impact of history on delta-encoding performance.

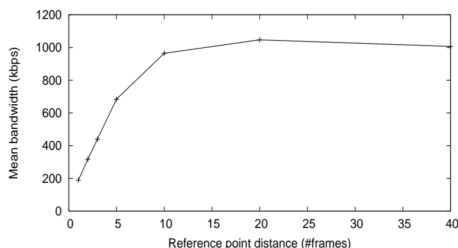

Figure 2: Impact of reference frame selection on delta-encoding performance.

puter AI-based players (*bots*). If updates are *not* delta-encoded, the bandwidth requirement for even a 5-bot game is well over 3Mbps (not shown), implying that delta-encoding, with even just the most recent frame, provides huge bandwidth savings. The graph also shows that the required bandwidth decreases as the history size increases (i.e., more base frames are available for encoding with respect to). There is a significant increase in savings at around 20 frames. This is because in the game missiles are shot approximately every 20 frames and new missles can be encoded efficiently if frame number $f - 20$ is available as a base frame for encoding frame f. Similarly, Figure 2 plots the mean bandwidth when encoding a frame f with respect to a *single* previous frame, one that is a distance d in the past (i.e., frame $f - d$). It is clear that encoding with respect to a more recent frame is better. Taken together, Figures 1 and 2 indicate that it is important to have the most recent frames available for efficient delta encoding, but sometimes also having an older reference frame available (in addition to the recent frames) can yield significant additional compression.

A simple delta-encoding scheme would be for the source (game server) to compute the deltas and transmit them to all the spectators via overlay multicast. However, this approach suffers from a few drawbacks. Due to packet loss and node failures, the "state" of each spectator (i.e., the set of frames it has received) could be different. So if the delta is computed with respect to the previous frame for maximum bandwidth savings, the spectator nodes may be unable to decode the delta frame (and subsequent delta frames) if they missed the previous frame. On the other hand, if we computed the deltas with respect to a full "key" frame (which is transmitted periodically), it would alleviate this problem but will make the deltas less efficient (as shown in Figure 2).

Instead, we propose a *distributed* delta-encoding scheme where the deltas are computed on a hop-by-hop basis, i.e., a parent node computes the delta with respect to the most recent frame that it shares in common with a child node. The fact that the spectating frame can be disaggregated into its constituent entities makes such a hop-by-hop delta-encoding scheme less expensive than hop-by-hop transcoding in the A/V context.

In general, it would be expensive in terms of both messaging cost and time for a parent node to determine which frames its child node has before computing and transmitting each delta encoded frame. An alternative would be to employ an *optimistic* delta-encoding scheme, where the parent transmits a delta with respect to the previous (delta) frame, assuming optimistically that the child has received and has been able to decode the previous frame. This is motivated by the observation in Figure 2 that deltas with respect to the previous frame are much more compact. In the unlikely event that it has not received the previous frame, the child can send its parent a list of recent frames that it has received and ask for a delta with respect to one of those frames.

4.3 Reliable Message Delivery

The persistence of the game state maintained by the spectating clients means that any missing state information is likely to have a lingering effect. For instance, missing the creation message for a new entity might make it difficult or impossible for a spectator to interpret future updates for that entity. Likewise, an entity whose deletion was missed by a spectator would linger on forever. Thus, we need a mechanism for the reliable delivery of certain critical messages, such as those corresponding to entity creation and deletion.

In the A/V context, reliability is attained by frequently repeating an entire "key" frame (e.g., the I frame in an MPEG sequence). However, this bandwidth inefficient strategy is inappropriate and unnecessary in the spectating context.

The distributed delta-encoding scheme discussed above can easily accommodate reliable message delivery. Reliable messages are marked as such, and unlike normal (unreliable) updates, they are *not* subsumed by later updates. Thus, so long as a child has not received a specific reliable message (e.g., entity creation message), its parent would include the reliable message (in *addition* to the most recent update, if any, for that entity) in the next delta-encoded frame that it transmits to the child. This procedure guarantees that the child will eventually receive the reliable message.

In some cases, one reliable message could "cancel" another (e.g., the creation of an entity followed by the deletion of the same entity). When a new spectator joins, there may be a large accumulation of such "cancelled" messages. Rather than deliver all such messages and have the new spectator process them, we modify the delta-encoding computation to ignore such cancelled messages.

4.4 Strategies for Bandwidth Adaptation

As noted in Section 3.1, there are two basic strategies for adaptating to limited bandwidth availability due to link constraints or congestion — *temporal* subsampling and *spatial* subsampling.

Temporal Subsampling: The idea in this technique is to reduce the frequency of updates received (e.g., receive every k^{th} update). This is akin to reducing the frame rate in the A/V context with the added flexibility that the client can choose to receive more frequent updates for certain entities and less frequent updates for the rest.

It may be possible to mask some of the jerkiness that would normally be associated with infrequent updates using interpolation techniques typically employed by a game rendering engine. The effectiveness of such a technique would depend on the movement pattern of the entities and the subsampling frequency. Furthermore, not all attributes might be amenable to interpolation.

Temporal subsampling can also reduce the effectiveness of delta-encoding, since the delta will have to be with respect to an older frame. For example, if only 1 in 10 frames is received, the bandwidth consumption is about 5 times larger than that between successive frames, according to Figure 2. Furthermore, the optimistic delta-encoding protocol outlined in Section 4.2 would have to be modified to let the parent know that the child is only receiving a fraction of the frames. (The parent does not have this information automatically, since the child could be receiving additional temporal slices from other parents, if multiple trees are used for dissemination.) With this information in hand, the parent can avoid sending deltas with respect to frames that the child would not have received.

Spatial Subsampling: The idea here is to deliver updates to the client for just a subset of the entities, thereby saving bandwidth. In fact, this capability is of interest for supporting multiple camera views, even when bandwidth adaptation is not the primary consideration.

The main advantage of spatial subsampling over temporal subsampling is that the quality of the spectating stream is undiminished *for the subset of entities* included in the stream. Also, the efficiency of delta-encoding is not affected by the subsampling.

Striping vs. Filtering: In general, we would like to support both temporal and spatial subsampling to give the spectating clients maximum flexibility. This requires a mechanism for "demultiplexing" the full stream of updates. We consider two possibilities: *striping* and *filtering*.

Striping - The basic idea is to construct multiple distribution trees and to transmit a different substream down each tree. Only the nodes that are members of a tree receive the corresponding substream. The substreams could either be temporal slices or spatial slices.

Filtering - In this case, we have a single logical tree, where each node transmits a filtered version of the received stream to each child. Again, filtering could be along temporal as well as spatial dimensions. The filter applied is based on the interest expressed by a child and is, in general, different for each child.

Filtering assumes that the interests of the clients at higher levels of the tree (i.e., close to the root) are a superset of those further away. Thus, it is inefficient to support both temporal and spatial subsampling with just a filtering-based approach. An alternative would be to use striping along both dimensions, i.e., construct a separate distribution stree for each combination of temporal and spatial slices. This would require as many trees as the product of the number of temporal slices (T) and spatial slices (S), with the associated overhead of maintaining each tree and its constituent links.

The alternative we propose is temporal striping coupled with spatial filtering. In other words, we use a separate distribution tree for each temporal slice and use filtering along the spatial dimension within each tree.[1] The spatial dimension could refer to different attributes (e.g., the physical space or perhaps the set of entities) depending on the game. Regardless, the space can be successively divided into "quadrants" that are organized into a hierarchy. The position of a node in the tree reflects the position of its spatial slice of interest in the hierarchy. Thus nodes that are interested in the entire space would be at the upper levels of the tree whereas nodes with narrower interests would be lower down. If a node's slice of interest straddles multiple "quadrants", the node simply attaches itself at multiple points in the tree.

We believe that this combination of temporal striping and spatial filtering offers several advantages over alternative approaches. Compared to a purely striping-based solution, our hybrid approach offers almost the same flexibility in demultiplexing with much less overhead in terms of the number of peer-to-peer relationships that need to be maintained. For instance, in our hybrid scheme, a node interested in a single temporal slice of the bottom-right quadrant of the game space only needs to attach itself at the appropriate location in the corresponding tree and install a filter. It needs to establish just one peer-to-peer relationship, i.e., that with its parent in the one tree that it joins. On the other hand, a purely striping-based approach would have a separate tree for each combination of temporal slice and "atomic" spatial slices, so subscribing to one temporal slice of the bottom-right quadrant would require joining each of the trees corresponding to the atomic spatial slices that constitute the bottom-right quadrant for the temporal slice of interest.

It is important to note that we have deliberately not considered the choice of spatial striping and temporal filtering. The advantage of temporal striping over spatial striping is that it achieves better load balancing across the trees, since it is not affected by the spread in the size or frequency of updates across different entities. Also, since the number of spatial slices is typically much larger than the number of temporal slices, handling the demultiplexing along the spatial dimension using filtering is likely to incur less overhead.

5. CHEERING

A simple form of cheering could be enabled by allowing spectators to simply vote for different pre-defined actions such as yell, stay quiet, and jump. The game could collect a sampling of all spectators and simply produce a roughly representative audience as part of the game. Such designs could be built using polling techniques that have been developed for conferencing systems [1]. In this paper, we focus on a more sophisticated form of cheering in which spectators can provide arbitrary au-

[1]We could employ multiple trees for each temporal slice for reasons of resilience and load-balancing, as in [9, 3]. However, we restrict ourselves to a single tree per temporal slice for our discussion here.

dio feeds, and the game then mixes the audio for each player to accurately represent the input of the spectators that are "closest" to the player. Given the diversity in phrases chanted, languages and accents, it would be impossible to mimic such cheers using a voting-based scheme.

In some ways this design is much like a general conferencing application where all participants can speak. However, there are some critical differences. First, floor control mechanisms are not needed since everyone is allowed to cheer simultaneously. Second, cheering presents a different view to each player receiving cheers since each player hears a possibly unique combination of all cheering inputs. Finally, the delay requirements for cheering are less stringent than conferencing since the players and spectators interact relatively slowly. However, cheering does require some synchronization to enable chanting (e.g., the entire audience yelling "go team").

A naive approach to supporting cheering is to deliver each spectator's cheers individually to all players within range of the spectator. However, this has some obvious bandwidth scaling problems. Performing in-network aggregation of cheers is a possibility; however, it is challenging since each player may compute a different aggregate of inputs. To address this problem, we could group players by vicinity and reduce the number of unique cheering views supported. In other words, the system could force players to share inputs for reducing overhead. In the worst case, there would only be a single view and all players would receive the same combination of cheering inputs.

To enable chanting, we could employ a combination of "cheerleading" and synchronized clocks (e.g. using GPS or NTP). An in-game cheerleader could initiate the chant much like they do at real-world sporting events. Using synchronized clocks, the playback of the cheerleader could be synchronized on all spectator machines. Similarly, the cheers of the spectators could be timestamped to indicate their relationship to each other. Players could merge the cheering inputs using these timestamps.

Note that it is the relaxed maximum delay constraint on the spectating-cheering feedback loop that permits the use of an overlay to deliver the cheering streams. However, many games may also place a high minimum delay constraint on this feedback to prevent spectators from providing illicit feedback to any player. This would limit the effectiveness or value of any cheering system.

6. RELATED WORK

Spectator-support is a relatively recent feature added to commercial games and hence, only a few such systems have been implemented in popular games. Currently, the only well-known game that supports spectators is Half-Life [6], which calls its spectator support Half-Life Television (HLTV). There is unofficial support for spectators in both Unreal Tournament [11] and Quake III as well. In all these systems, spectator-support is enabled by running a special server which streams game data to spectators either using IP Unicast or IP Multicast. These spectator systems typically allow spectators to choose from a few *viewing modes*. For example, HLTV supports the following modes: Overview, Free-Look, First-Person and Third-Person.

The most important disadvantage of these systems is that the spectator server bandwidth scales linearly with the number of clients (assuming low availability of IP Multicast). In order to remove this bottleneck and provide a robust, resilient spectator system, we leverage a significant amount of research work done for scalable application-level multicast ([4], [9], [3]).

Support for receiver heterogeneity (bandwidth adaptation) has been widely studied in the A/V multicast context ([8], [2], [7], [10]). Our system incorporates solutions that build on the core control-theoretic ideas presented in these papers. However, as discussed in Section 3.1, spectating provides a richer set of specific scenarios and mechanisms for adaptation.

Finally, the early work on scalable realiable multicast (SRM) [5] for applications such as distributed whiteboard also worked with a disaggregated view of the scene, which permitted the retransmission of individual updates such as ink strokes. However, SRM was based on a network-layer multicast substrate and depended on broadcast-based recovery mechanisms, which are not suitable for a high-bandwidth spectating stream.

7. CONCLUSION

In this paper, we presented the networking challenges arising from spectating and cheering in the context of online multiplayer games. We have pointed out how spectating and cheering differ from traditional A/V streaming and conferencing, and have discussed the challenges that arise in providing resilient and bandwidth-efficient support for them. We have outlined a design that incorporates a number of elements, including distributed delta encoding, reliable message support, a hybrid bandwidth adaptation strategy based on temporal and spatial subsampling, and in-network aggregation. We are currently in the process of building and evaluating our solution in the context of the Quake III game.

Acknowledgements

We thank Aaron Ogus and Phil Chou of Microsoft for early discussions on spectating.

8. REFERENCES

[1] E. Amir, S. McCanne, and R. H. Katz. Receiver-Driven Bandwidth Adaptation for Light-Weight Sessions. In *ACM Multimedia*, 1997.
[2] J. Byers, M. Mitzenmacher, and M. Luby. Fine-Grained Layered Multicast. In *IEEE INFOCOM*, April 2001.
[3] M. Castro, P. Druschel, A. M. Kermarrec, A. Nandi, A. Rowstron, and Singh A. Splitstream: High-bandwidth multicast in a cooperative environment. In *ACM SOSP*, October 2003.
[4] Y. Chu, S. G. Rao, and H. Zhang. A Case for End System Multicast. In *ACM SIGMETRICS*, June 2000.
[5] S. Floyd, V. Jacobson, S. McCanne, C. G. Liu, and L. Zhang. A Reliable Multicast Framework for Light-weight Sessions and Application Level Framing. In *ACM SIGCOMM*, September 1995.
[6] Half-Life TV. "http://planethalflife.com/features/articles/hltv/".
[7] G. Kwon and J. W. Byers. Smooth Multirate Multicast Congestion Control. In *IEEE INFOCOM*, 2003.
[8] S. McCanne, V. Jacobson, and M. Vetterli. Receiver-driven Layered Multicast. In *ACM SIGCOMM*, August 1996.
[9] V. N. Padmanabhan, H. Wang, and P. A. Chou. Resilient Peer-to-Peer Streaming. In *IEEE ICNP*, November 2003.
[10] V. N. Padmanabhan, H. J. Wang, and P. A. Chou. Supporting Heterogeneity and Congestion Control in Peer-to-Peer Multicast Streaming. In *IPTPS*, February 2004.
[11] Unreal Tournament TV. "http://utv.clan-sy.com/help.html".