Application Layer Protocol for Video-on-Demand

Panayotis Fouliras Athanasios Manitsaris

Department of Applied Informatics, University of Macedonia
54006 Thessaloniki, Greece
Email: {pfoul,manits}@uom.gr

Abstract. We propose a new scalable application-layer protocol, specifically designed for low-bandwidth, data streaming applications with large receiver sets. This is based upon a control hierarchy of successive hypercubes for the peers and is quite robust to peer and network failures. Dynamic load balancing is also incorporated. We present an analysis showing that it is near optimum in performance and use of server channels.

1 Introduction

The advance of communication technology has spawned a large number of services, previously too expensive or even impossible to access for the average user.

However, there are still services that require a large amount of bandwidth and the associated cost has not allowed them to achieve widespread use. Video-on-Demand (VoD) is one such service.

Under VoD there are at least three entities, namely the video server, the customer's client computer and the intermediate network. The client sends a request for a certain video title; the video server processes the request and, if possible, replies with a corresponding video stream.

There are several conflicting requirements: Less bandwidth per video means more video streams (*virtual channels*) available per video server. Also, due to the nature of the Internet, it is not easy to avoid jitters by establishing isochronous virtual channels between the server and the client. The memory space occupied by a single video is typically much larger than the available memory on the client. The client requests for the same video do not necessarily arrive within the same period, forcing the use of more channels. Moreover, in a typical video service the client expects additional capabilities, such as fast forward, pause and rewind, not to mention interactive video.

Many researchers have worked on these problems the past few years and have proposed several interesting ideas.

Some of the proposals are patching [1], skyscraper broadcasting [5], bandwidth skimming [6], SVD [7] and greedy disk-conserving broadcasting [11], all of which try to minimize the duration of broadcast or the number of additional server channels for the same video.

Other researchers try to utilize client memory in a simple but very hard to implement way (e.g. chaining [4]).

Other proposals have slightly different goals or assumptions, such as Variable Bit Rate (VBR) broadcasting [10, 12, 13, 14], or lossy network environments [2].

In most cases the goal is to minimize the delay of service for client requests, while at the same time minimizing server and overall network bandwidth. Unsurprisingly, the server uses multicasting in order to serve many client requests simultaneously for the same video. If all such requests can somehow be grouped together in a batch, only one video stream needs to be transmitted. This method corresponds to *Near Video on Demand (NVoD)*, where client requests close to each other in time are grouped to form batches. This is opposed to *True Video on Demand (TVoD)*, where a request is served as soon as possible.

The key problem we address in this paper is how to minimize overall video server network bandwidth, while simultaneously maintaining the latency of service to client requests minimal, for popular videos. We assume client bandwidth slightly higher than the playback rate for incoming traffic and outgoing traffic approximately the same if multicasting can be utilized. In the case of unicasting for clients, the number of video channels has an upper bound of b, where $b>1$. In all cases, multicasting is assumed for the video server itself.

We also assume that the client buffer size is very small (approximately 2% of the total video duration, which has a typical duration of 120 minutes). Furthermore, we assume that the clients can join or leave a broadcast at any time, either due to their choice or due to the problematic nature of the underlying network. These assumptions are quite realistic for present-day mobile clients (e.g. PDAs).

Based on these assumptions, we propose the use of clients not only as passive receivers of videos, but also as partial video servers for other clients through the use of their buffers. This is not a novel approach [4, 9, 15, 16], but the organization of the underlying access control mechanism is, since we propose a semi-hierarchical hypercube overlay against the tree-like arrangements proposed by other researchers [16]. We also propose a different mechanism for the join and departure of clients, with emphasis on fault-tolerance, recovery and dynamic load balancing.

The obvious advantages of such an approach are minimization of the load for the video server and the waiting time for clients which require the same popular video. The disadvantages are the realization of the proposed protocol, which is challenging, but not very difficult to accomplish and the possibility that the clients may not prefer to have some of their bandwidth used for broadcasting. The latter can be solved either as a term for using the video services (i.e., if a client wants to receive a video it has to use the appropriate software agent which forces broadcasting) or through other incentives. Of course, another problem may arise if a network connection is unbalanced – download bandwidth is much larger than the upload. Work is undergoing on this special issue, but this is not taken into account in the present work.

The rest of the paper is organized as follows. In section 2 we formulate the problem. In section 3 we present the *Application Layer Multicast protocol (ALM)*, together with analysis showing that it is near optimum, scalable and resistant to failures. Our conclusions follow in section 4.

2 Problem Formulation

We assume for simplicity that there is one video server *S*, which contains a set of videos *M* with cardinality n_M. The duration of each video is *D*. Also, *C* is the set of all clients, while C_m the set of clients requesting the same video *m* up to a certain time point. The cardinality of these sets is n_c and n_{cm}, respectively. The buffer size available at each client (expressed in playing time) is $d << D$, which we assume is the same for all clients.

There is no limit on the amount of clients who can make requests, except that only one request per client may be outstanding or served at any moment. Client requests are denoted by r_{jm}, where $1 < j \leq n_{cm}$ and may arrive at any time. The server receives such requests directly, but tries to serve them at discrete successive time points, t_i, t_{i+1}, …, so that t_{i+1}-$t_i \leq t_w$. The latter (t_w) is a constant that depends on the amount of time a client is willing to wait for service, before it decides to withdraw its request.

The server primarily provides the video service, but we try to use as much of the available memory on the clients that are already being served. Therefore, if at least one client receives the same video at successive time points with time difference $t_w < d$, it is possible to form a "chain" of successive video streams that serve all client requests up to the present time, using only a single server channel. Thus, at some time point t_i, there are n_{cm} clients for the same video grouped in *i* levels, namely L_1, … L_i. The server is always at level L_0.

Finally, each client has only one channel for video reception and only *b* channels for video broadcasting at playback rate. These are the *data* or *video* channels.

Multicasting is assumed for video broadcasting from a parent client to its children for the basic version of ALM, in which case *b* can be considered practically infinite.

Unicasting is also considered in the second version of ALM, in which case *b* is some positive integer, depending on the upper limit of video broadcasting channels per client.

It is possible for clients to fail, withdraw or operate in a lossy network environment. Consequently, such "chains" would break and the system should somehow try to remedy the situation. Therefore, a solution must satisfy the following characteristics:

- It must be simple and fast in order to adapt quickly to the changing circumstances
- It must minimize the amount of simultaneous video server broadcasts
- No client must wait for time > t_w to be served
- It must provide an access control mechanism for clients to join or leave quickly so that the amount of network traffic to any client is manageable
- It must provide speedy recovery for client or network failures

Since the client memory can be utilized by ALM only if the same video has been requested, in the rest of the paper we assume all client requests are for the same video, unless otherwise stated.

3 Problem Solution

The ALM protocol arranges clients into a hierarchy of i levels, where $0 < i \leq \lceil D / t_w \rceil$. The main operation is to create and maintain this hierarchy.

Contrary to other proposals [16], the data and control paths are not identical: The data path follows a tree-like arrangement where a client at level L_i, provides a multicast stream to a group of clients at level L_{i+1} (or a set of up to b unicast streams).

The control path is twofold: All the clients at level L_i are organized in a binary hypercube. They maintain and exchange control information with their neighbors at the same level, as well as their parents and children in the data path, which allows them to respond quickly in the event of local failures. One of the clients at each level i is the *Local Representative* (LR_i). This client, together with other *LRs* from the rest of the levels, communicates with the video server, forming a control topology of a star, keeping overall communication minimal.

In the rest of this section we describe the exact form of hierarchy for ALM and how it is used to establish scalable control and data paths.

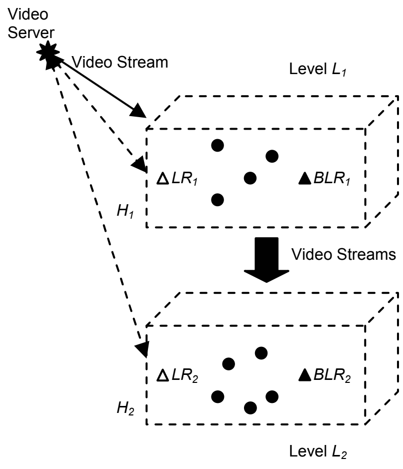

Fig. 1. Hierarchy under ALM

3.1 Arrangement of Clients

The control hierarchy is created by assigning members to different levels. From time t_i to $t_{i+}t_w$ the server receives client requests for the same video, which it groups into the level L_i, arranging them in a hypercube data structure. The arrangement is not random; the end-to-end latency of the path between a client and the server is used as criterion to select the *Local Representative* for this level (LR_i) and all other clients are placed closest to it in the hypercube. The second closest client is selected as the *Backup Local Representative* (BLR_i).

The rationale behind this arrangement is that the LR_i is the only client for level L_i communicating with the server under normal conditions; hence an effort is made to select the one closest to the server in terms of end-to-end latency. The *BLR* is selected for the case of a failed *LR*, since it can quickly take up its place.

The *LRs* and the server form a star with the server at the center. The total communication load on the server for this set of clients is relative to maximum number of levels $\lceil D/t_w \rceil$. This has the advantage that the server can detect *LR* problems quickly. Moreover, this arrangement is more reliable than any other scheme with message hopping from *LR* to *LR* until the server is reached, since *LRs* are clients that can withdraw at any moment without notice. This hierarchy is depicted in Fig.1, for the first two levels of clients grouped into hypercubes H_1 and H_2.

Assuming that there are n_{i-1}, n_i and n_{i+1} clients at levels L_{i-1}, L_i and L_{i+1} respectively, the server divides the clients at level L_i in n_{i-1} equal-sized subgroups, assigning each subgroup to a client at level L_{i-1}. This, progressively, forms a tree structure, which is used for the data path (i.e., video streams).

In the end, each client v at level L_i communicates for control purposes with its parent at level L_{i-1}, as well as its children at level L_{i+1}. Therefore, the control communication paths needed per client v under normal conditions are:

$$Control\_Paths(v) = log n_i + min(b, n_{i+1}) + 1 . \qquad (1)$$

The second term depends upon whether we have multicasting or unicasting.

3.2 Protocol Operations

The server and each client act both as data as well as control management servers, but only the server is considered reliable. Under the ALM protocol, there are three phases for any client: *Join, Work* and *Leave*.

3.2.1 Join Phase

Under the *Join* phase, a client v requests a video-clip from the video server at some time $t \in (t_{i-1}, t_{i-1}+t_w)$. The server gathers all requests R_j for the video and calculates the end-to-end latency between each client and itself, forming an ascending sorted list of clients. This list is used to create a virtual hypercube H_i for the group R_j of these clients.

Next, the server determines whether there is already a broadcast to at least one client, which is currently receiving the first part of the video. If none exists, a new broadcast is scheduled from the server; otherwise, the new level L_i and identity of the LR_i are determined.

This information, together with the above-calculated hypercube H_i is sent to the LR_i and BLR_i of level L_i. The rest of the clients only receive the list of their neighbors in H_i. Thus, the size of these messages is $O(log n_i)$.

Finally, as explained in the previous section, the server divides the clients at the new level L_i in n_{i-1} subgroups and sends this information to each client at level L_{i-1}, and each client at level L_i. In this way a forest of trees is formed where a client at level L_{i-1} is the parent and certain clients at level L_i are its children. This forest augments the data path, apart from the control path. If possible, the LR_i and BLR_i and their neighbors are not assigned any children due to their additional administrative load and the need to reserve a manageable amount of clients as backup for failure of other clients.

There are many possible assignments, but to keep the arrangement simple and faster to compute, the clients at level L_i ‘closer’ to the server are assigned to the closest client at level L_{i-1}. This is merely a calculated guess, since the network environment may change considerably over time.

This step concludes the *Join* phase. By now, each client at level L_i has the following information:

- An identity in H_i
- An initial state of H_i (only the LR_i and BLR_i)
- Its immediate neighbors in H_i
- Its parent in the data path, as well as the neighbors of its parent in H_{i-1}
- It knows whether it is the LR_i or BLR_i for level L_i
- It knows the LR_{i-1} and BLR_{i-1} for its parent level

In addition, each client at level L_{i-1} knows its children in H_i.

3.2.2 Work Phase

During the *Work* phase, the clients at level L_{i-1} broadcast the video content in their buffers to their respective children at level L_i.

Apart from the data, control information is exchanged in order to detect any possible problems.

First, all clients send periodically a simple *Alive* message to all their neighbors in the hypercube. If no such message arrives from any neighbor w, between successive transmissions of *Alive* messages by client v, then w is no longer considered neighbor of v. Each of these messages includes the parent identity of their neighbors and its respective load. Thus, a list of potential parents is formed, sorted according to their load. Only neighbors of their parent with load $< b$ are considered potential parents, so this list is kept in an ascending order.

Also, each client v sends periodically an *Alive* message to its parent p. This message informs p about its remaining children.

An *Alive* message from a parent p to its children is also necessary, so that it informs them on its current load (i.e., how many children it currently serves). This is also useful for load balancing.

Finally, the LR_i periodically exchanges a special *Alive* message with the BLR_i. This is sent so that either can detect potential failure of its peer.

3.2.3 Leave Phase

This phase deals with two cases: Normal and abnormal termination of client participation.

Under the first case, a client v that wishes to withdraw sends a *Quit* message to all its neighbors in the hypercube and also to its parent and children.

Under the second case, a client v no longer broadcasts video to its children and does not exchange *Alive* messages with its neighbors or parent.

In both cases the parent *p* removes *v* from the list of its children. Furthermore, if unicasting has been used, *p* stops broadcasting video to *v*. The neighbors of *v* update their information about the hypercube status, accordingly.

3.2.4 Orphans and Recovery

There are two problems that have to be solved now: The first problem is that the children of *p* at level L_{i+1} are now *orphans*. Since they know the immediate neighbors of *p*, they send a *LJoin* (Local Join) message to the first of them, say p_1. If p_1 is able to accept some of them, it replies sending *Alive* messages according to its current load (no more than *b* children under unicasting) and proximity. Thus, orphans are not necessarily accepted as children by any single neighbor of *p*.

If no *Alive* message arrives from any neighbor of *p*, the remaining orphans send a *LJoin* message to the LR_i, denoting their late parent. This is useful for grouping cases of orphans of the same failed parent.

LR_i deletes the *p* and all neighbors of *p* from its hypercube H_i. It then probes the clients at its perimeter in H_i to check the number of orphans anyone is able to accept. The reason is that such clients first receive orphans of their neighbors. Only those clients capable of accepting at least one child respond. Therefore, the moment that LR_i receives enough messages to allocate orphans it stops probing clients (if has not sent probes to all clients at its level) and sends this information to the orphans as it receives it. This is depicted in Fig. 2. The orphans manage to become children of *p*'s neighbor on the right, apart from one, which is not accepted by any neighbor. It asks the LR_i which finds another client, which accepts it as its child. Note that this information is periodically sent to BLR_i and the video server.

If LR_i has found no appropriate parent or it has failed, the orphans try the same process with LBR_i. The latter becomes LR_i and selects its closest neighbor with the least load as the new BLR_i through a special *LRSelect* message.

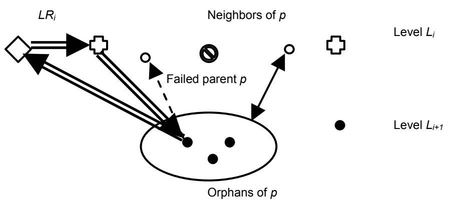

Fig. 2. Orphans and Recovery

If no new parent is found or both LR_i and BLR_i have failed at the same time, the orphans contact the server. The server schedules a new broadcast to the orphans and their descendants. It also calculates the new H_i possibly merging fragments of the hypercube and selects one of the remaining clients as the LR_i and another one as the BLR_i. It then sends the new H_i information to both of them and the updated neighbourhood information to the rest of the clients. Then the process continues as described above.

Note that LR_i does not calculate the new hypercube H_i; only quick modifications are performed to the old hypercube. Also, the server does not deal with such situations unless whole parts of the hypercube have failed.

3.2.5 Uncertainty due to Client Failures

Finally, a special problem arises from the fact that the control communication pattern is fairly distributed and unreliable. It is possible that no *Alive* message by *v* reaches some neighbor *w*. This is a partial failure: One or more network links have failed to deliver the *Alive* message, but client *v* and some of its links operate properly.

If client *w* is a neighbor of *v*, which has not received an *Alive* message by *v* within a certain amount of time, it simply deletes *v* from its list of active neighbors, although it keeps sending it *Alive* messages periodically. It is, thus, hoped that the link with *v* will operate again soon, in which case *v* is re-instated as an active neighbor of *w*. If this does not occur and *w* fails, the children of *w* do not send a request to *v* as a potential father. To avoid extreme cases, *v* is removed permanently from the list of *w*'s neighbors after a constant number of unanswered *Alive* messages.

3.3 Analysis - Experimental Results

As described earlier, there are at most $O(\lceil D/t_w \rceil)$ possible time-slots at which client requests may belong, requiring a separate video channel for their service. We shall focus our analysis in the worst case, where the server uses multicasting and the clients use unicasting.

Under ALM, the number of video server channels for a video *m* range from one (optimal case when at least one client per time slot) up to $\lceil D/2s \rceil$(worst case when client requests arrive every two time slots).

Using ALM, each level in the hierarchy must have at least one client to maintain it. Each client must exchange a pair of messages with every neighbor at the same level, another pair with its parent and each of its children. With a total of n_{cm} clients in the hierarchy and equation (1), we have for the normal case:

$$Client\_Messages = O(\log n_i + b)\,. \qquad (2)$$

Since $n_{cm} = \sum_{k=1}^{i} \mathrm{n}_k$ for *i* levels and in the worst case the number of clients at level L_i is:

$$n_i \le b^{(i-1)} * n_1\,. \qquad (3)$$

we can determine a stricter bound:

$$Client\_Messages = O(\log (b^i * n_1) + b)\,. \qquad (4)$$

In practice, we expect $2 \le b \le 4$. Hence, equation (4) now becomes:

$$Client\_Messages = O(i + \log n_1)\,. \qquad (5)$$

Thus, we determine that the number of participating clients in the hierarchy depends on the number of clients at the first level (3) and that the amount of messages per client is bounded by $logn_1$ and the amount of levels (5).

If the number of clients is approximately the same at each level or we are at the beginning of the hierarchy (i.e., i is small), equation (4) becomes:

$$Client\_Messages_{Avg} = O(\log n_1 + b)\ . \qquad \textbf{(6)}$$

In case of any parent p failure at level L_i, the worst case for the amount of messages per orphan or neighbor of p is $O(logn_i)$. Using (3) we find:

$$Orphan\_Messages = O(i^* \log b + \log n_1)\ . \qquad \textbf{(7)}$$

Using (4), (5) and (7) we find that the total amount of messages per client in the hierarchy is:

$$Total\_Messages = O(\log n_1 + i)\ . \qquad \textbf{(8)}$$

$$Total\_Messages_{Avg} = O(\log n_1 + b)\ . \qquad \textbf{(9)}$$

The only exception to the analysis above is the *LR* at each level. This has a higher burden than the rest of the clients, since it has to receive the initial and updated hypercube status by the server. It also needs to select and probe potential new parents for orphans. In the worst case these are as many as the clients in its perimeter area. Together with those for the *LBR* and the number of the orphans yields $O(logn_i+b)$ messages. From the previous discussion we end up at equations (8) and (9) above.

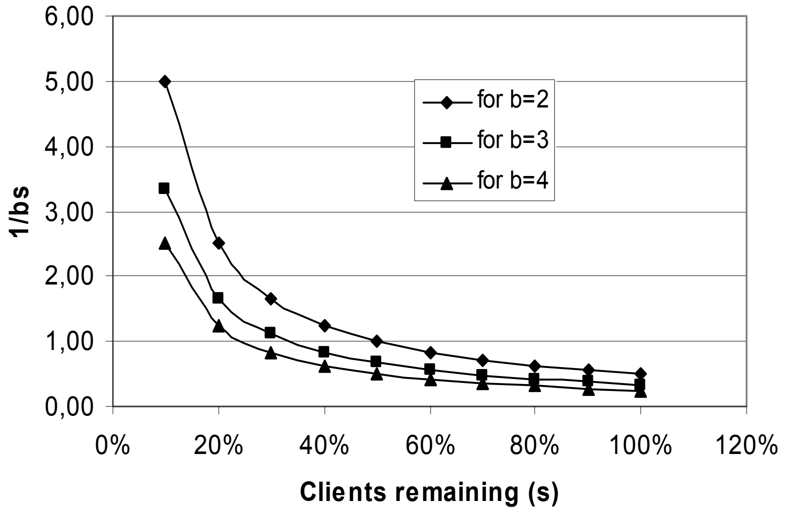

Fig. 3. Probability for Massive Failure

Of course, in the extreme case all parents at every second level fail. Thus, the server falls to the batching strategy, with $\lceil D/2t_w \rceil$ channels to accommodate the orphans, although not for the full duration of the video [3].

Based on the discussion above, we see that for many clients, additional server channels are required only in the case of massive adjacent client faults. If only partial faults take place and the clients are evenly distributed at each level, then only a single video stream is required (a speed-up of up to $\lceil D/2t_w \rceil$ times, which is near optimum).

If *s* is the cumulative percentage of non-failed clients during a complete video broadcast, then up to $(b*s*n_i)$ children at level L_{i+1} can be accommodated. Thus, massive failures are more probable when the ratio $n_{i+1}/1/(b*s*n_i)$ is close to 1. This can be rewritten as $n_i/n_{i+1} \leq 1/(b*s)$. That is, massive failures are likely to occur when the ratio $1/bs \geq 1$. This is depicted in Fig. 3 for possible values of *b*.

From this figure we see that massive failures are likely when $1/bs \geq 1$, which occurs with b=2 for s=50%, with b=3 for s=35% and with b=4 for s=25%.

Finally, the clients never calculate the hypercubes. This is performed initially by the server and then in two more extreme cases: Either both LR_i and BLR_i or the complete level L_i of clients have failed. In the first case, the server only needs to receive at least one message that both LR_i and BLR_i have failed and to verify itself this event by probing them. If more messages from orphans arrive, the server calculates the new hypercube and selects two of the remaining ones as the LR_i and BLR_i.

We, therefore, see that ALM is not only near optimal in terms of server channels usage, but also scalable and quite robust to failures.

Dynamic load balancing is straightforward to implement: Each client at the same level can easily find out through *Alive* messages, the current load of its neighbours. If a client has at least 2 children more than one of its neighbors, one of the children is "passed" to that neighbor. This process is easy to implement and does not require any central authority (e.g., the video server) to participate in it. Another advantage is that no disruption in the video stream being delivered occurs. The most important advantage, however, is that the children are more evenly distributed to data subgroups, reducing the probability of massive failures.

4 Conclusion

We have proposed ALM, a new multicast application layer protocol for NVoD, utilizing the available buffer of clients under a hierarchy of successive hypercubes, in a faulty environment, leading to better server network and channel utilization.

Preliminary analysis has shown that it is scalable and quite robust, for NVoD and relatively easy to implement, since it is less complex or demanding for clients compared to other proposals.

Work is in progress for a more detailed simulation with enhancements on the basic idea.

References

1. Kien A. Hua, et al: Patching: A Multicast Technique for True Video-on-Demand Services, *Proceedings ACM Multimedia Conference (SIGMM)*, (1998) 191-200

2. Mahanti L., et al: Scalable On-Demand Media Streaming with Packet Loss Recovery, *Proceedings ACM SIGCOMM Conference* (2001), 97-108
3. Jack Y. B. Lee: UVoD: An Unified Architecture for Video-on-Demand Services, *IEEE Communications Letters*, 3(9):277-279 (1999)
4. S. Sheu K. Hua, Tavanapong, W.: Chaining: A Generalized Batching Technique for Video-On-Demand Systems, *Proceedings ICMCS'97 Conference* (1997) 110-117
5. Hua K., Sheu, S.: Skyscraper Broadcasting: A New Broadcasting Scheme for Metropolitan Video-on-Demand Systems, *Proceedings ACM SIGCOMM Conference,* (1997) 89-99
6. Hua K., Cai Y., Sheu S.: Patching: A Multicast Technique for True Video-on-Demand *Services, Proceedings ACM Multimedia Conference (SIGMM)* (1998) 191-200
7. Eager D., et al: Optimal and Efficient Merging Schedules for Video-on-Demand Servers, *Proceedings ACM Multimedia Conference (SIGMM)* (1999) 199-202
8. Min-You Wu et al: Scheduled Video Delivery for Scalable on-Demand Service, *Proceedings ACM NOSDAV Conference* (2002) 167-175
9. Wang, James Z., Guha, Ratan K.: Data Allocation Algorithms for Distributed Video Servers, *Proceedings ACM Multimedia Conference (SIGMM)* (2000) 456-458
10. Loser C., et al: Distributed Video on Demand Services on Peer to Peer Basis, *Proceedings International Workshop on Real-Time LANS in the Internet Age (RTLIA)* (2002)
11. Saparilla D., Ross K.: Periodic Broadcasting with VBR-Encoded Video, *Proceedings IEEE Infocom Conference* (1999) 464-471
12. Lixin Gao et al: Efficient schemes for broadcasting popular videos, *Multimedia Systems*, 8:284-294, (2002)
13. Tantaoui M., et al: Interaction with Broadcast Video, *Proceedings ACM Multimedia Conference (SIGMM)* (2002)
14. Yanping Zhao, et al: Efficient Delivery Techniques for Variable Bit Rate Multimedia, *Proceedings MMCN Conference*, (2002)
15. Kien Hua, JungHwan Oh, Khanh Vu: An adaptive video multicast scheme for varying workloads, *Multimedia Systems*, 8:258-269, (2002)
16. Kien Hua, et al: Overlay Multicast for Video on Demand on the Internet, *Proceedings ACM SAC Conference,* (2003)