Computer Science Department
University of Crete

LEMP: A Peer-to-Peer Video Streaming Protocol

Konstantinos Katertzis

Thesis Submitted to the faculty of the
University of Crete in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE
In
Computer Science

Heraklion, June 2009

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

LEMP: A Peer-to-Peer Video Streaming Protocol

Εργασία που υποβλήθηκε από τον
Κωνσταντίνο Κατερτζή
ως μερική εκπλήρωση των απαιτήσεων για την απόκτηση
ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΕΙΔΙΚΕΥΣΗΣ

Συγγραφέας :

_____

Κωνσταντίνος Κατερτζής
Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Εισηγητική Επιτροπή:

_____

Ευάγγελος Μαρκάτος
Καθηγητής, Επόπτης

_____

Μέμα Ρουσσοπούλου
Επίκουρος Καθηγήτρια, Μέλος

_____

Αθανάσιος Μουχτάρης
Επίκουρος Καθηγητής, Μέλος

Δεκτή :

_____

Πάνος Τραχανιάς, Καθηγητής
Διευθυντής Μεταπτυχιακών Σπουδών στο Τμήμα Επιστήμης Υπολογιστών
του Πανεπιστημίου Κρήτης

Ηράκλειο, Ιούνιος 2009

Abstract - In this work we propose *LEMP*, a new scalable application-layer protocol, specifically designed for video streaming applications with large client sets. *LEMP* is a pull-based protocol in which peers act both as clients and partial servers. The novelty of our protocol is its use of a semi-hierarchical overlay which, in a sense, "brings back to life" the traditional tree-based approach but this time from a specific only point of view. We emphasize four salient features of *LEMP* design and implementation: 1) fault-tolerance and self-recovery so as to successfully adapt to high churn environments, 2) minimum simultaneous video server channels, 3) manageable network traffic per client and 4) minimum control overhead. We present a detailed design justification, implementation and evaluation of our prototype on real-world PlanetLab testbed and large-scale emulation on the Emulab testbed. Our comparative analysis with an existing widely deployed commercial system shows that *LEMP* preserves high *QoS* and minimum control overhead.

Περίληψη – Στην παρούσα εργασία παρουσιάζουμε το *LEMP*, ένα νέο κλιμακούμενο πρωτόκολλο επιπέδου εφαρμογής, το οποίο είναι σχεδιασμένο ειδικά για video streaming εφαρμογές με μεγάλο αριθμό πελατών. Το *LEMP* είναι ένα pull-based πρωτόκολλο στο οποίο οι μετέχοντες λειτουργούν σαν πελάτες και μερικώς σαν διακομιστές. Η πρωτοπορία του πρωτοκόλλου μας είναι η χρήση της ημι-ιεραρχικής του τοπολογίας η οποία κατά κάποιο τρόπο επαναφέρει στο προσκήνιο την παραδοσιακή δενδροειδή προσέγγιση, αλλά μόνο από μία συγκεκριμένη οπτική γωνία αυτή την φορά. Τονίζουμε τέσσερα βασικά χαρακτηριστικά της σχεδίασης και υλοποίησης του *LEMP*: 1) ευρωστία και αυτό-ανάρρωση έτσι ώστε να προσαρμόζεται επιτυχώς σε ιδιαιτέρως δυναμικά περιβάλλοντα, 2) ελαχιστοποίηση του απαιτούμενου εύρους ζώνης από τον διακομιστή, 3) ανεκτή δικτυακή κίνηση ανά πελάτη και 4) ελαχιστοποίηση του φόρτου της ιεραρχίας ελέγχου. Παρουσιάζουμε μία λεπτομερή αιτιολόγηση του σχεδιασμού, υλοποίηση και αξιολόγηση του πρωτοτύπου μας στο PlanetLab testbed καθώς και εξομοίωση μεγάλης κλίμακας στο Emulab testbed. Η συγκριτική μας ανάλυση με ένα ήδη υπάρχον και ευρέως διαδεδομένο αντίστοιχο εμπορικό σύστημα δείχνει ότι το *LEMP* διατηρεί υψηλή ποιότητα υπηρεσιών και περιορισμένο κόστος ελέγχου.

Acknowledgements

I would like to thank my supervisor, Prof. Mema Roussopoulos, for her feedback and her useful comments on the text. I also thank Vassilis Lekakis for his time, patience and ideas on improving the protocol as well as for his help regarding the PlanetLab testbed experiments.

I would like to thank some friends from Heraklion who gave me strength when the “battery was running low”. Gabriel, Natasa, Panagioti, George (and family), Katerina and James thank you all for your company.

Finally, profound thanks to my parents, my sister and my brothers for always being there.

Table of Contents

LEMP: A Peer-to-Peer Video Streaming Protocol

Katertzis Kostas

Department of Computer Science
University of Crete
Heraklion, Crete, Greece
kater@csd.uoc.gr

Abstract - In this work we propose *LEMP*, a new scalable application-layer protocol, specifically designed for video streaming applications with large client sets. *LEMP* is a pull-based protocol in which peers act both as clients and partial servers. The novelty of our protocol is its use of a semi-hierarchical overlay which, in a sense, "brings back to life" the traditional tree-based approach but this time from a specific only point of view. We emphasize four salient features of *LEMP* design and implementation: 1) fault-tolerance and self-recovery so as to successfully adapt to high churn environments, 2) minimum simultaneous video server channels, 3) manageable network traffic per client and 4) minimum control overhead. We present a detailed design justification, implementation and evaluation of our prototype on real-world PlanetLab testbed and large-scale emulation on the Emulab testbed. Our comparative analysis with an existing widely deployed commercial system shows that *LEMP* preserves high *QoS* and minimum control overhead.

1. Introduction

Over the past few years, streaming technology has become one of the leading edges in worldwide network communications. The last added corner stone in this trend is video streaming, which gains momentum not only due to the penetration of broadband Internet access into households but also because it seems to perfectly fit the multimedia social framework of our days. In July 2006, more than 100 million videos were being watched every day and this number continues to increase [23]. On the other side of this coin, the traditional client-server architecture, where each client is allocated a dedicated stream from the server, is both costly (e.g., YouTube's bandwidth costs are estimated to 1 million a day) and has difficulty scaling to large client sets.

Bearing in mind the above, several proposals have been made to adapt the advantages of peer-to-peer (P2P) architecture to video streaming technology. P2P approaches have already been successfully applied to file transfer systems and now developers are focusing on the more demanding area (in terms of delay, churn and real-time constraints) of video streaming. Roughly speaking, the basic approach of these proposals is as follows: each peer participating in the system has the dual role of receiving and serving the video as well. Some of the differentiating aspects of these systems are whether or not there is a central server that facilitates the distribution amongst peers, whether the video is a live feed or a stored video, the architecture of the overlay (e.g. tree, mesh, cell, unstructured, etc.), whether video parts are buffered temporarily (sliding window buffering) or they are locally stored for the longterm, whether a push-based or a pull-based method is preferred, whether the system is intended for a controlled environment or not and whether multiple downloads are supported (e.g. video segmentation). Each of the proposals that have been made thus far exhibits a set of tradeoffs.

The main tradeoff that has to be dealt with is that between control overhead and delay, while achieving the goal of video continuity and smooth playback. In most systems (specifically those involving video segmentation over a pull-based method), each peer has to immediately inform its neighbors of a packet reception (buffer maps exchange) and the neighbors, interested in this packet, should make a straight-forward request for the packet. In this way, the delay is minimized but a noticeable control overhead is added to the system. For the overhead to be reduced, a peer might resort to less frequent buffer maps exchanges (e.g. after dozens of new packets have been received) but this can lead to extended delays and possible video interruptions.

Although P2P models based on file segmentation have already been successfully implemented in a large scale (e.g. BitTorrent), this is not still the case for (live) video streaming. The main difference is that, in P2P video streaming systems, peers are more interested in low delays (initial buffering delay and playback delay) and video continuity rather than in maximizing the aggregated download throughput of their link. Moreover, peers in existing proposals often have to carry the triple burden of serving other peers (which incurs bandwidth), storing video parts for long periods of time (as in the case of push and static local store methods) and reconstructing received video parts for playback (which incurs computational effort). Thus, the demands of this relatively new area are more challenging than in traditional P2P systems. The new challenges become more difficult due to strict timing and bandwidth requirements, peers' unpredictable behavior and extremely large flash crowds.

In this thesis, we propose LEMP a new scalable application-layer protocol, specifically designed for video streaming applications with large client sets. A preliminary version of the protocol was proposed in previous work but had limited evaluation and a number of limitations [24]. We thus present a complete rehaul of the LEMP protocol along with a detailed design justification, implementation and evaluation of the protocol on real-world PlanetLab testbed and large-scale emulation on the Emulab testbed.

We believe our protocol to be the key for minimizing overall video server network bandwidth, while simultaneously maintaining the latency of service to client requests at a minimum. We assume that client bandwidth is slightly larger than the playback rate for incoming traffic, that clients may be characterized by heterogeneity (computing power, buffer size) and that clients may join or leave the P2P overlay at any time either by choice or due to network failures.

Our protocol is a pull-based protocol in which peers act both as clients (receiving the video) and partial servers (serving the video). The novelty of our protocol is its use of a semi-hierarchical overlay which, in a sense, "brings back to life" the traditional tree-based approach but this time from a specific only point of view. Furthermore, we emphasize on the mechanism for the join and departure of clients, building it so as to be fault-tolerant, self-recoverable while keeping the control overhead to minimum levels. A full description of the LEMP protocol in its initial stages is available in [24]. In this work, we provide a high-level description of its salient features and then focus on a number of design improvements we added to the protocol as we built a new implementation prototype.

The rest of the document is organized as follows. In Section 2 we formulate the problem. In Section 3 we present the original LEMP protocol. In Section 4 we present the enhancements applied on the original protocol. In Sections 5 and 6 we present an evaluation of our revised LEMP protocol using detailed measurements of an implementation on the PlanetLab and Emulab testbeds respectively. Finally, in Section 7 we provide a summary regarding related work and in Section 8 we conclude.

2. Problem Formulation

For simplicity we assume one video server S, containing a set of videos, with D the duration of each video. C is the set of all clients, with 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 $d < D$.

There is no limit to the number of clients which can make requests, except that only one request per client may be outstanding or served at any moment. Client requests for video m are denoted by r_{jm}, where $1 < j \leq n_{cm}$ and may arrive at any time. The server 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 goal is to leverage as much as possible the available memory and bandwidth of the clients that are already being served by the server. 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. Thus, at some time point t_i, there are n_{cm} clients requesting the same video grouped in i

levels, namely L_1, …, L_i. The server is always at level L_0 and broadcasts to the clients of level L_1.

Each client has only one channel for video reception and b channels for video broadcasting at slightly larger than the playback rate. These are the *data* or *video* channels.

All clients use unicasting to broadcast video; hence, b is a positive integer, depending on the upper limit of video channels per client. It is possible for clients to fail, withdraw or operate in a lossy network environment.

Consequently, such pipelines would break and the system should try to remedy the situation. Therefore, a solution must satisfy the following characteristics:

- Be simple and fast to adapt quickly to the changing circumstances
- Minimize simultaneous video server channels for the same video
- Ensure that no client waits longer than t_w for service
- Ensure manageable network traffic per client
- Provide speedy recovery for client or network failures
- Have minimal requirements regarding client computing power

3. Proposed Solution

3.1 LEMP Hierarchy

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

Contrary to other proposals [1, 8, 11], the data and control paths are different: The data path follows a tree-like arrangement where a client at level L_i provides a set of up to b unicast streams to a group of clients at level L_{i+1}. These streams do not have to be synchronized; they may be transmitting different parts of the buffer content.

Assuming 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} same-sized groups, if possible, assigning each group to a client at level L_{i-1}. This forms a tree structure, used for video streams.

The control path on the other hand is twofold: all the clients at level L_i are organized in a star-like structure. One of the clients at each level L_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. The rest of the clients at level L_i maintain and exchange control information with their respective LR_i. This arrangement allows quick response in the event of client failures.

As for the arrangement of clients, 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. 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 the second closest client is selected as the *BackupLocal Representative* (BLR_i). This happens because 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. Similarly, the *BLR* is selected in order to replace quickly a failed *LR*. This hierarchy is presented in Fig. 1, for the first two levels of clients.

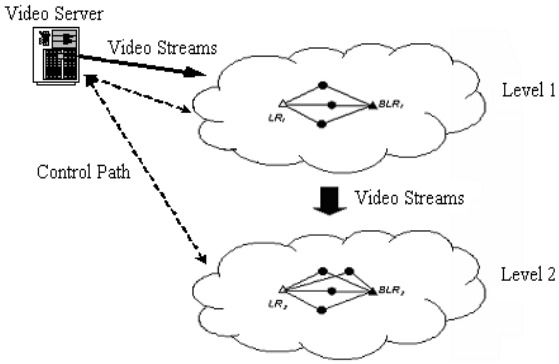

Figure 1. *LEMP* Control and Data Hierarchy

3.2 Protocol Operations

Under LEMP there are three phases for any client: *Join*, *Work* and *Leave*.

3.2.1 Join Phase

Under *Join*, a peer *v* requests a video from the server. This request, as well as all other requests made by other clients within a specific time period (t_{i-1}, $t_{i-1} + t_w$), are gathered from the server who creates an ascending sorted list of clients, based on the end-to-end latency between each client and itself.

Next, the server determines whether there is already a broadcast to at least one client, currently receiving the first part of the video (level L_{i-1}). If none exists, a new broadcast is scheduled by the server; otherwise, the new level L_i and identity of LR_i and BLR_i are determined.

This information is sent to the LR_i and BLR_i of level L_i. Each client v_i only receives the identity of LR_i, BLR_i LR_{i-1}, BLR_{i-1} and its parent. Thus, the size of these messages is constant.

Finally, the server divides the clients at the new level L_i into n_{i-1} groups so as to ensure that they are equally distributed to the parents of the previous level. This information is sent to each parent at level L_{i-1}, and its child at level L_i, such that each client at level L_{i-1} knows its children at level L_i. Thus, a forest of trees is formed, augmenting the data path. If possible, LR_i and BLR_i are not assigned any children due to their additional administrative load.

3.2.2 Work Phase

During this phase, the clients at level L_{i-1} broadcast the video in their buffers to their respective children at level L_i. Apart from data, control information is exchanged in order to detect any possible problems.

First, all clients send periodically an *Alive* message to their respective *LR*. If no such message arrives to *LR* from any client *v* within a certain time interval t_δ, then *v* is considered to have failed. Each of these *Alive* messages includes the client's identity and load. Thus, a list of potential parents is formed, sorted according to their load, in case of regular parents fail.

Finally, LR_i periodically exchanges a special *LRAlive* message with the BLR_i and the video server, containing all information updates regarding the state of clients at the particular level. This is sent so that the server or BLR_i can detect potential failure of its peer and synchronize control information.

3.2.3 Leave Phase

A client *v* may leave the overlay either by choice, in which case it sends a *Quit* message to LR_i and also to its parent *p* and children, or unexpectedly (e.g. due to network failure) so it no longer serves its children and does not send *Alive* messages to its *LR*.

In both cases, *p* removes *v* from the list of its children. Furthermore, *p* stops broadcasting video to *v*. The *LR* updates its information, accordingly.

- Orphans and Recovery

  The departure of a peer at level L_i has the result that some peers at level L_{i+1} are now orphans. Since they know LR_i, they send an *OJoin (Orphan Join)* message to LR_i. LR_i determines potential parents and replies by sending *ODirect (Orphan Direct)* messages, directing them to the appropriate new parents. If LR_i has failed, the orphans try the same process with BLR_i.

  If no new parent is found or both LR_i and BLR_i have failed at the same time, the orphans contact the server, which schedules a new broadcast to them.

- Uncertainty of Client Failures

  The control communication pattern is fairly distributed and unreliable. It is possible that no *Alive* message by *v* reaches its respective *LR* within the time interval $t_\delta/2$. This is a partial failure; one or more network links have failed to deliver the *Alive* message, but client *v* operates properly.

  In this case the *LR* simply deletes *v* from its list of potential parents, although it keeps waiting for *Alive* messages for another time interval $t_\delta/2$ (a total of t_δ). It is, thus, hoped that the link with *v* will operate again soon, in which case *v* is re-instated as a potential parent by the *LR*; otherwise, *v* is permanently deleted from its list

4. Protocol Enhancements

Thus far we have described the salient features of the original LEMP protocol. In this section, we

describe features that we have added to the basic protocol as part of our newly implemented prototype.

An under the hood analysis was the main reason that lead us to the recently applied enhancements on the original protocol. This insightful analysis consisted of a detailed practical revision of the protocol and a complete re-implementation of it.

The improvements made on the basic version of our new protocol (rehauled *LEMP*) are summarized below:

- "On the fly" service of client requests
- Client arrangement assigned to both server and *LRs*
- Absence of *BLR; step_LR* role introduced
- *LRs'* control partial independence from server
- Absence of *OJoin* messages
- Improved failure inspection
- Non-propagating failures
- Support for *live* video

We refer to the basic *LEMP* protocol plus these improvements as "*LEMP on the fly*".

Further refining enhancements applied to the rehauled *LEMP* protocol result in the latest version of our protocol (*"LEMP heuristic"*). These improvements include:

- Different heuristic for *LR* selection
- An actual *heuristic for step_LR* selection
- A fair heuristic for parent selection
- Different handling of orphans

4.1 On the fly service

In the basic *LEMP* design described in Section 3, the server gathers all client requests made during a time period (t_{i-1}, $t_{i-1} + t_w$) and then tries to serve them. It is, thus, possible for a client to have to wait up to t_w before being served (e.g. if his request was made at the beginning of the period). Although t_w is not numerically set (or at least proposed) in *LEMP*, we envision a protocol with minimum startup delay and this potential wait of t_w must be reduced.

In our revised protocol, we propose and implement an "on the fly" service of clients. That is, the moment the server receives a client request at level L_i, it decides whether to serve it directly (e.g. when no peers exist at level L_{i-1}) or, most importantly, it can immediately forward it to LR_{i-1}. Once such a forwarding is made, the LR_{i-1} will immediately find a parent for the new client (see paragraph 4.2) leading to a minimum startup delay experienced by the client. This procedure depends on the improvements described below for its functionality.

4.2 Client Arrangement

As it is probably already suspected, contrary to *LEMP*, the arrangement of clients is not a server only responsibility.

Specifically, client arrangement is now performed in two steps; each step is performed by a different protocol component. The server is still responsible for "nominating" the role of peers, regarding their *hierarchical* position, that is if they are *LRs* or normal peers. The second arrangement step, on the other hand, is the responsibility of the *LRs*. *LRs* are now those who assign children (peers at level L_i) to parents (peers at level L_{i-1}) and thus decide on peers' *functional* role into the protocol. The new procedure is as follows: let us assume that all requests made within the period (t_{i-1}, $t_{i-1} + t_w$) are successfully served and that the server moves to the next period, waiting for new requests. At this point the server has already created the ascending sorted list of peers (based on latency) at level L_{i-1} and has also chosen the LR_{i-1}, which it informs of its role. We assume that a peer *v* requests the video (from the server) within the time period (t_i, $t_i + t_w$). After the *hierarchical* role (*LRs* or not) of peers at level L_{i-1} was set by the server, is now time for the LR_{i-1} to set their *functional* role. The server simply forwards the request of peer *v* to LR_{i-1} and the responsibility of parent-to-child assignment is now in its hands. Likewise to *LEMP*, *LRs* try to be fair and preserve streaming load balance among parents.

In this way, not only is the startup delay minimized but a workload balance is also achieved, since the server shares its "burden" with *LRs*.

4.3 *Step_LRs* instead of *BLR*

In the original protocol, there is supposed to be a *Local Representative* (*LR*) and a *Backup Local Representative* (*BLR*) at each level. Our first, though a bit rough, implementation revealed that setting t_w to a medium value, renders the presence of *BLR* almost unnecessary, as far as successful

query forwarding to the *LR* is concerned. Such a t_w value however, keeps number of peers at each level respectively low and furthermore creates a more flexible hierarchy in terms of level maintenance.

Moreover, the cost of keeping *BLR* updated didn't seem to pay back as often as initially believed, since the (only one) *BLR* itself had equal chances to undergo a failure/ departure. Hence, we preferred a more sophisticated approach which further increases protocol robustness while minimizing control overhead.

The newly introduced role is that of *step_LRs*. Instead of having a single one backup *LR* which is continuously updated by the original *LR*, we choose to have a plethora of possible step *LRs*, one of which will be informed of its new role upon *LR*'s departure. More specifically, when LR_i is about to leave, it will nominate an "alive" peer of its level as new *LR* (*step_LR*), by informing him of their level status. The same "hot potato" procedure will take place upon the departure of the *step_LR*, such that a *LR* is always present and up to date.

Note that *step_LR* implementation requires no special synchronization between the *LR* and the *step_LR*, while control overhead is kept to minimum levels.

4.4 Partial Control Independence of *LR*

As described in *LEMP*, LR_i periodically exchanges a special *LRAlive* message with the video server, containing all information updates regarding the state of clients at the particular level.

However, our analysis showed that this information is rather useless on the server side, after the before mentioned improvements have been applied. To be more precise, under all circumstances, the maintenance of level L_i is now under LR_i's control. That means that the server is no longer responsible for changes or failures at the different levels.

Still, there is a catch regarding LR_i's control information at the server side. The server needs to be actually informed about LR_i status, up till the point LR_i waits for forwarded requests from the server; that is during the period $(t_{i+1}, t_{i+1} + t_w)$. If LR_i is up, then the server keeps on forwarding client requests (of level L_{i+1}) normally; if not, the server is held responsible for incoming requests and accordingly serves them directly. In this way, we achieve, even as a side effect, an even lighter control hierarchy.

4.5 Absence of *OJoin* messages

Under the departure of a peer at level L_i, orphans of level L_{i+1} send an *OJoin* message to LR_i, according to the original *LEMP* protocol. However, this is not the case in our enhanced new protocol. Instead, the recover procedure takes place automatically and no extra messages are required.

The departure of a peer at level L_i, will be noticed by LR_i after the absence of alive messages within the t_δ interval (case of an unexpected departure) or by an explicit *Quit* message sent by the leaving peer (case of normal departure). LR_i will then find a new parent for the orphan. This choice is currently made based on the number of children each parent has; more precisely, the parent with the fewest children is chosen from LR_i as the step-parent of the orphan.

We believe that *OJoin* and *Alive* messages serve the exact same purpose, in the case of an unexpected departure, so we totally removed *OJoin* messages from our new protocol.

4.6 Improved failure inspection

In the first protocol approach, unexpected failures are inspected only by the absence of *Alive* messages. Not much is mentioned nor specified for the special case of orphans under the absence of LR_i (and BLR_i in the original protocol).

In the revised approach, we propose a 2-tier mechanism against failures (e.g. broken link). We believe that this part of our system is vital for the overall system robustness and self-recovery, under challenging and unexpected situations.

We have already elaborated on the first tier, which is consisted of the *Alive* messages. These messages are sent periodically by peers at level L_i to their LR_i. If no *Alive* messages are received by LR_i, that is within the time interval t_δ, the corresponding peer is assumed to be "dead" and the recover procedure takes place. Note that a peer failure is no longer indicated by the absence of **two** successive *Alive* messages, since our control hierarchy is now built atop the *TCP* protocol, meaning no packets are lost, and possible high delays are dealt by setting an elastic timeout on the respective socket.

However, it is possible that the LR_i itself might leave the overlay, before video ends. In this case, there has to be a backup procedure running in all peers, such that they are able of self-detecting a failure at the level above (L_{i-1}). In light of this, each peer has a timeout, regarding newly received buffers. If no packets are received before the expiration of this timeout, the peer considers itself as an orphan and requests the video directly from the server.

For this mechanism to be functional, the "*buffer timeout*" is set slightly larger than the value of t_δ, such that its expiration certainly indicates a *LR* failure.

4.7 Non-propagating failures

One problem that came to surface, while testing our first implementation, was propagating failures. Consider a peer *A* at level *i*, who is facing bandwidth problems but is able to keep up with control message traffic. In other words, consider a node which is normally responding to LR_i's alive messages but for some reason, it is unable of serving the video to his children at level *i+1*. Since, peer *A* appears to be alive, no recovery procedure will begin until the "buffer timeout" at *A*'s children has expired. However, this "buffer timeout" will be propagated to all peers downstream peer *A*. In all, a single failure/ error might be propagated down to the leaves of the data tree, leading to a large number of orphans and rendering the protocol unstable.

For this weakness to be dealt, while a peer undergoes an error or a "buffer timeout" instead of normally replying to the *LR*'s "alive" message, it replies with a "temp_off" message to inform the *LR* of the temporary problem. *LR*, consequently, will temporarily remove this peer from the list of parents and will redirect any possible orphans to step-parents of its level. In this way, errors are not propagated to successive levels.

4.8 *Live* video streaming

The original protocol was theoretically designed for *VoD* services only. However, the advance of broadband connections has changed the perspectives of video streaming. A decade ago, IPTV or live broadcasting of a special event (e.g. football match, rare physical phenomena) to a large number of simultaneous peers seemed to be too ambitious. Nowadays, the ground for such systems is definitely more solid and able to hold the burden of their demands.

We have, thus, focused our implementation on *live* video streaming. We use the term *live* to refer to the simultaneous distribution of the same content to all clients. The content itself might be a live feed (e.g. live webcam or recording camera) or a playback. Consequently, each peer unicasts the exact content to all his children, contrary to the original protocol.

4.9 Different heuristic for *LR* selection

In the original protocol, *LR* selection was strictly based on minimum *RTT*. However, the *LR* is responsible for maintaining the control hierarchy which normally involves higher *cpu* usage demands. Thus, a more sophisticated heuristic based on a weighted average of *RTT* and peer's *cpu* usage percentage was chosen.

4.10 Heuristic for *step_LR* selection

The "hot potato" scheme engaged in *"LEMP on the fly"* was quite arbitrary, as far as the *step_LR* selection is regarded. Trying to keep things fair, we introduce an actual heuristic for *step_LR* selection based on a weighted average of peers' current *cpu* usage and *cpu* usage history.

4.11 A fair heuristic for parent selection

A key aspect of our protocol's feasibility and fairness is definitely parent selection, which was initially based on a very simple algorithm; children were equally distributed among parents without taking into account any bandwidth limitations. To provide a more effective approach for parent selection, children assignment to parents is now performed based on parents' available bandwidth (exported with *Iperf*) and the number of already assigned children to them.

4.12 Different handling of orphans

In the original protocol, upon the reception of an orphan message, the server sent the video directly to the peer that issued the orphan request. With this enhancement, each client that undergoes a buffer control timeout (not handled by the *LR* of its level) will issue a first-orphan message to the server and will be redirected to the current (latest) level *LR* to

receive the video again. If the peer undergoes a second buffer control timeout during his lifetime, he will request and receive the video directly from the server. Note that, this handling concerns only buffer control timeout orphan requests and not orphan requests issued upon the reception of an explicit *parent_out* message (these orphans still receive the video directly from the server). We find that with this solution server stress is improved without heavily worsening video loss percentage.

5. Planet-Based Performance Evaluation

Our first prototype (*LEMP*) was implemented in the *Java* language, using the *Java Media Framework* (*JMF) API*. However, the nature of *Java* renders it a relatively heavy tool for such demanding applications. Moreover, *JMF* after its first release, received little attention, despite the promises and initial envision of *Sun* for a new powerful *API* for multimedia applications. Its last update was made available in 2004 and since then the *API* is untypically considered as obsolete. Still, this first implementation armed us with significant experience and knowledge, while at the same time revealed problems and disadvantages of the initial protocol.

After applying the new design changes to the basic *LEMP* protocol, we implemented in full a new prototype (*LEMP-heuristic*). Our new prototype is implemented in the *Python* language in conjunction with the *GStreamer Python* bindings. Our experience has shown that these new tools are lightweight, flexible and reliable in terms of video streaming and network communication.

5.1 Experiment Setup

We conducted an experiment on PlanetLab to evaluate the performance of our new protocol. The total number of participating PlanetLab nodes is 390, which is almost the largest scale of nodes available on PlanetLab, and all results are extracted from one PlanetLab run. The video stream is 3000 seconds long and the streaming rate is 150 kpbs, unless stated otherwise.

According to [21], when it comes to live media streaming, the nature of interactions between users and objects is fundamentally different from stored media streaming. The greatest difference is that access to stored objects is user driven (since the media is "always there" and is only up to client's will when he will access it), while access to live objects is object driven (since the timely constrained availability of the content drives user behavior). In light of this, our workload was strictly based on the model for live media workload generation, provided by [21], with peers inter-arrival time following a *pareto* distribution (α = 2.52, b = 1.55) and peers lifetime modeled by a *lognormal* (μ = 5.19, σ = 1.44).

Table 1. PlanetLab Experiment (Default) Parameters

Parameter	Value
Total Number of Clients	390
Streaming Bitrate	150 Kbps
Video Length	3000 seconds
Client Arrangement Interval (t_w)	60 seconds
Alive Message Interval (t_δ)	3 seconds
Client Interarrival Time	Pareto α = 2.52, b = 1.55
Client Lifetime	Lognormal μ = 5.19, σ = 1.44
Unexpected Failures	5%

Note that, since our experiments are conducted in a real network, unexpected failures due to link congestion/break are present by definition. Furthermore, no special choice over PlanetLab nodes was performed, that is nodes might be heavily loaded both in terms of bandwidth consumption and cpu usage. In light of these, artificially injected failure probability was set to 5% in all experiments, unless stated otherwise.

5.2 Metrics

We evaluated our prototype under the following metrics:

- **Startup Delay:** time period elapsed between the time the video request is issued and the time packets start flowing in our pipeline.
- **Video Loss Percentage (Discontinuity Index):** this metric concerns the maintenance of continuous playback. We define as video discontinuity the percentage of time that no new data is flowing in the buffer to the total time the peer is supposed to watch the video ("lifetime" – "startup delay"). In other words, it is a time estimator of the number of segments that arrive (if at all) after playback deadlines over the total number of segments.

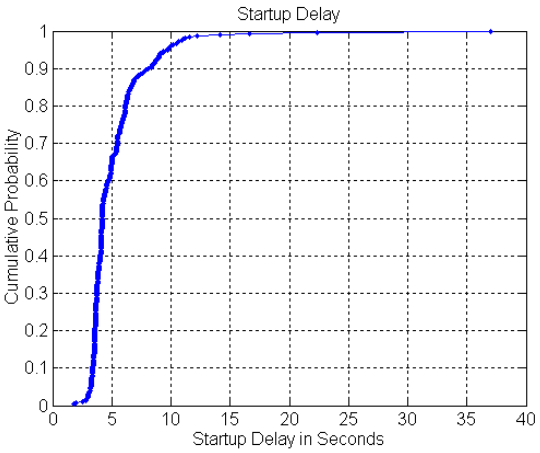

Figure 2. Startup delay in seconds

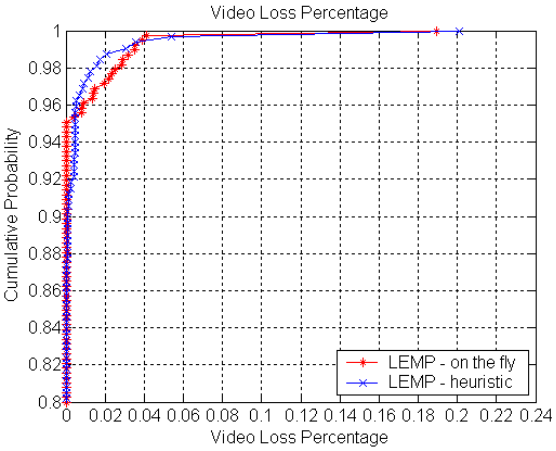

Figure 3. Video loss percentage

- **Percentage of video received directly from server:** we have each peer tracking down the time periods that, either due to problems (i.e. orphan client) or due to the nature of our protocol (i.e. clients at level 1), it receives the video directly from the server. This metric is an indicator of server stress in terms of time. As above, this metric is the ratio of the time the peer was served directly from the server to the total time the peer was watching the video ("lifetime" – "startup delay")

- **Total Outgoing Rate:** Each peer runs an instance of *Iptraf* to monitor its bandwidth usage. This metric concerns the total (data and control) uploading rate of each peer.

- **Data Outgoing Rate:** A second instance of *Iptraf*, providing more detailed information (i.e. port numbers), runs on every peer. This metric represents the data (video) uploading rate of each peer.

- **Control Outgoing Rate:** Similarly to above, this metric depicts the control overhead of each peer in terms of upload bandwidth consumption.

- **Server Stress Saving Percentage:** This metric refers to the protocol's server bandwidth that is saved compared to the traditional client-server model.

5.3 Experimental Results

Unless stated otherwise, the results presented below refer to our latest protocol version, namely *"LEMP heuristic"*.

5.3.1 Startup Delay

Startup delay (Figure 2) is clearly kept in low levels for the majority of population. More specifically, 50% of peers received the video in less than 4 seconds and 90% of them in no more than 9 seconds. Note that only 1% of the population received the video after 15 seconds or more.

5.3.2 Video Loss Percentage

Ideally, we would expect our protocol to achieve zero video loss percentage for almost all clients. That is, despite the undergoing operations (departure, failure, rejoin, etc.) we envision the majority of peers observe no video discontinuity. Indeed, our prototype (Figure 3 *"LEMP heuristic"*) presents 0% video loss for 91% of the population, while 98% of peers lost no more than 1.7% of the video. In order to clarify the impact of the different handling of orphans on video loss, we also choose to present video loss percentage for *"LEMP on the fly"*. As Figure 3 (*"LEMP on the fly"*) indicates, 95% of peers underwent 0% video loss and 98% of peers lost no more than 3% of the video. This comparison reveals the effectiveness of our protocol in terms of video continuity since video loss percentage remains at low levels, even under the new and more demanding handling of orphans.

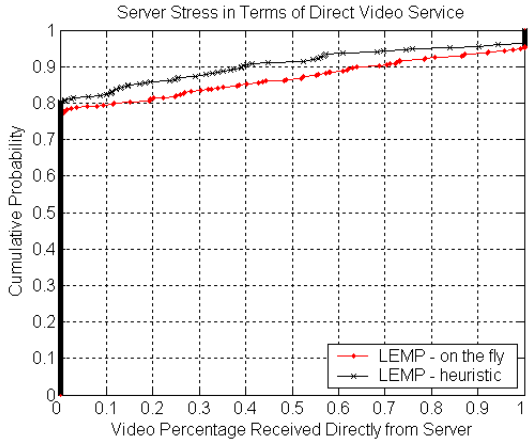

Figure 4. Video percentage received directly from server

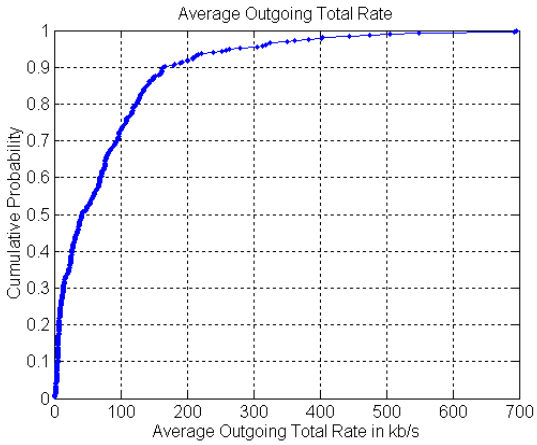

Figure 5. Average Outgoing Total Rate

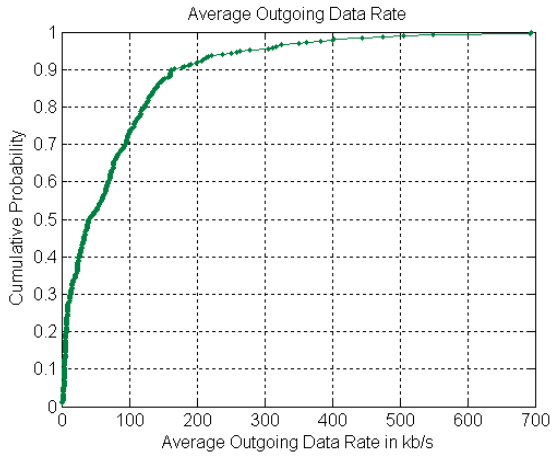

Figure 6. Average Outgoing Data Rate

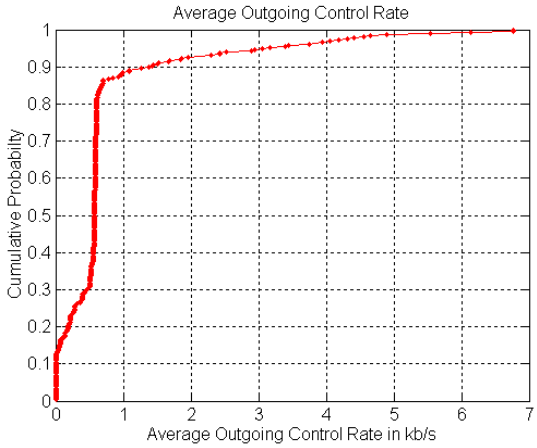

Figure 7. Average Outgoing Control Rate

5.3.3 *Percentage of video received directly from server*

As for server stress in terms of direct video service percentage, a remarkable improvement is achieved since a portion of the orphans population is now handled by other peers and not by the server. Figure 4 (*"LEMP heuristic"*) shows that not only is there an increase in the number of clients receiving the video exclusively from other peers (81% in contrast to the 77% of *"LEMP on the fly"*), but more importantly, 90% of the peers received less than or equal to 40% of the video directly from the server, while in *"LEMP on the fly"* the same percentage of total population received less or equal to 65% of the video directly from the server. Moreover, only 5% of peers received the video directly from the server, right from the beginning. One should note that these percentages are over client's lifetime, which varies among clients. These results show the protocol's ability to relieve server stress effectively.

5.3.4 *Total & Data Outgoing Rate*

To demonstrate the protocol's feasibility, Figure 5 shows the upload bandwidth consumption of participating peers. We see that 60% of peers contributed less than 85kb/s on average and 90% of peer contributed less or equal to 190kb/s. Moreover, 99% of the population had outgoing total rate less or equal to 500kb/s while only 4 peers experienced higher outgoing rate values. As expected, data outgoing rate (Figure 6) is almost identical to the total rate, since control rate is kept to minimum levels. Bearing in mind the worldwide invasion of broadband internet connections in households, we expect the majority of home users to successfully carry the burden of such uplink demands and only a small percentage of peers remaining on dial-up modem connections having their upload bandwidth stressed.

5.3.5 *Control Outgoing Rate*

An important aspect of our protocol's implementation is control overhead. Figure 7 shows that 90% of peers experienced control

overhead of less than 1.4kb/s, while the biggest value was only 6.77kb/s. Figure 8 provides an illustration of total, data and control average outgoing rate, so as to better depict the magnitude difference between control and data outgoing rates. Moreover, since each node mainly interacts with the *LR* of its level (via alive and parent/child messages) and every control message has a fixed size, the average control overhead of each node does not grow with the overlay size or with the streaming bitrate.

5.3.6 *LRs and non-LRs*

To compare the stress placed on both *LRs* and non-*LRs*, Figures 9 and 10 present the average outgoing data and control rate for *LRs* and simple (non-LR) peers respectively. Figure 9 demonstrates the fairness of our protocol, since server stress appears to be equally distributed among *LRs* and normal peers. Note that this performance was achieved only after the parent selection heuristic was applied on our protocol and that in our protocol's previous version (*"LEMP on the fly"*) *LRs* were clearly more stressed than non-*LR* peers. When it comes to control overhead, we expected *LRs* to be more loaded than normal peers, since it's their responsibility to handle their level. Indeed, according to Figure 10, *LRs* spend more bandwidth than non-*LRs* but still the fluctuations are modest. Most importantly, the knee of non-*LRs* demonstrates that the protocol places a bounded amount of control overhead on non-*LR* peers.

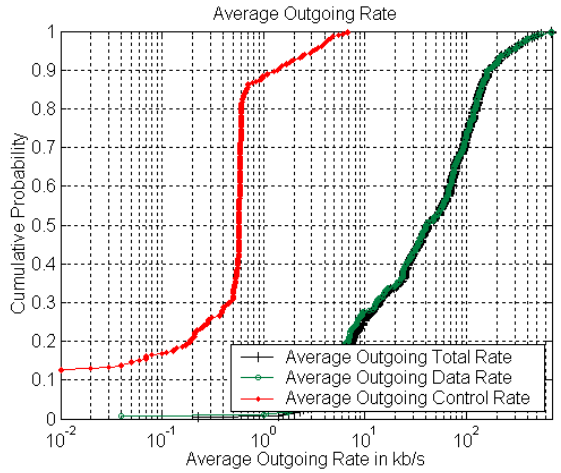

Figure 8. Average outgoing total/data/control rate

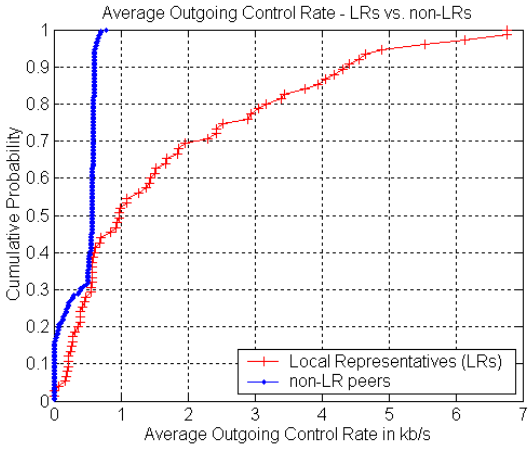

Figure 10. Average Outgoing Control Rate for LRs and non-LRs

5.3.7 *Increased Unexpected Failures*

To evaluate the protocol under worst-case scenarios, we conducted experiments with peers failing unexpectedly with various (increasing) probabilities (5%, 10%, 15%, 20%). We would like to highlight once more that since all experiments take place on *PlanetLab*, unexpected failures due to link congestion/break are present by definition. The following three experiments, thus, push the protocol to its limits so as to further examine its robustness and performance.

Figures 11, 12 and 13 present a comparison of the differentiating metrics of these experiments, so as to better present and compare the protocol's performance under the different setups.

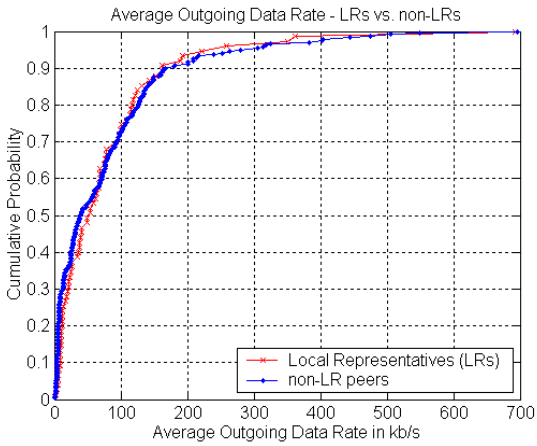

Figure 9. Average Outgoing Data Rate for LRs and non-LRs

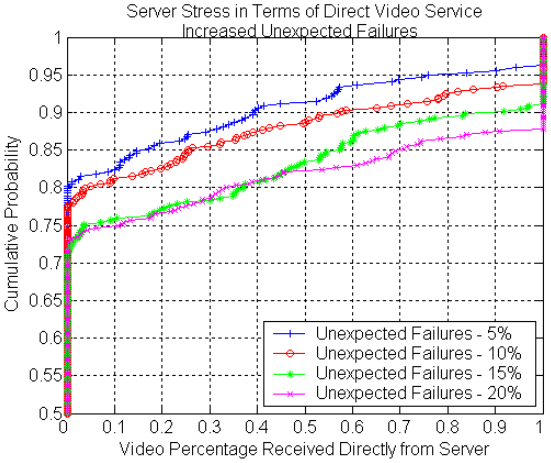

Figure 11. Video percentage received directly from server under different percentages of unexpected failures

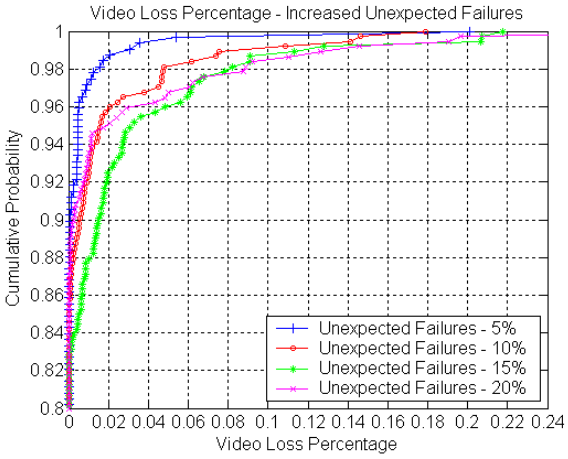

Figure 12. Video loss percentage

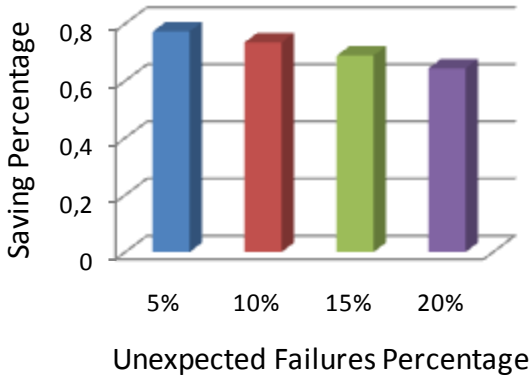

Figure 13. Server Stress Saving Percentage

5.3.7.1 *Percentage of video received directly from server (unexpected failures)*

It is normal to expect that as unexpected failures increase, the server will have to carry some extra burden, regarding peers' service; especially in the case of unexpected *LR* failures. Our concern was the extent to which the protocol preserves a high performance under such demanding circumstances. According to Figure 11, the protocol's performance degrades as unexpected failures increase but still the majority of peers is served in an efficient p2p way. Even with 20% unexpected failures, 73% of peers receives the video exclusively from other peers. Moreover the number of peers that receive the video right from the beginning of their session directly from the server is analogous to the percentage of unexpected failures, which is the best possible scenario.

5.3.7.2 *Server Stress Saving Percentage*

The extra server burden, which is described above, is clearly depicted in Figure 13. The plot presents bandwidth savings at the server side in comparison to the traditional client-server model and clarifies the consequences of the increased unexpected failures. With 5% unexpected failures, the server saves 77% of the bandwidth that would had spent under the client-server model, while under 20% unexpected failures, server savings are reduced to 64%. We choose to present percentages of server savings in terms of bandwidth demands, rather than absolute values of server stress, since these values are closely related to the overlay size and the streaming rate which varies among applications.

5.3.7.3 *Video Loss Percentage (unexpected failures)*

As for video loss percentage (Figure 12), though there is a degradation, 84-91% of peers experience no video losses, while for all cases 96% of peers loses no more than 5% of the video.

5.3.8 *Comparative Analysis*

To better evaluate our protocol's performance, we also provide a comparison with one of the most widely deployed and commercially available P2P live-video streaming systems, namely *CoolStreaming* [22]. Since the source code is not publicly available and as several technical information and parameters are absent for an implementation of this system to be feasible by a third party, we choose to evaluate our protocol using the same parametric conditions and experimental environment (PlanetLab) used in their evaluation so as to achieve a fair comparison. Specifically, the client set is consisted of 200 PlanetLab nodes and the default video bitrate is 500 Kbps. In the dynamic environment, peer lifetime is exponentially distributed with an average of *T* seconds while unexpected failures are set to 5%.

5.3.8.1 *Stable Environment*

We first evaluate continuous playback and control overhead in a stable environment; that is peers join the overlay and stay on until the end of the video stream. Regarding video playback continuity under different streaming rates (Figure 14), our protocol outperforms *CoolStreaming* mainly due to two reasons. First, our protocol's data hierarchy construction distributes children to parents based

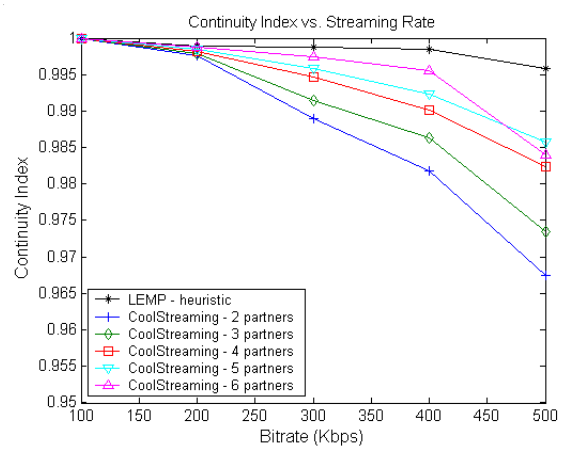

Figure 14. Continuity index vs. streaming rates in stable environment

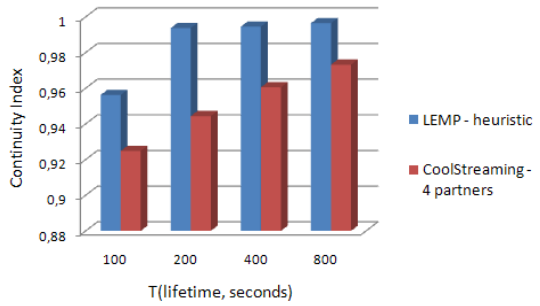

Figure 16. Continuity index under dynamic environment – Streaming Bitrate 500 Kbps

on their throughtput and the number of already assigned children to them such that bandwidth outages are minimized. Second, peers are normally bound to undergo no more than two video discontinuity malfunctions, since after the second one they will request the video directly from the server. This limit was set so as to both decrease server stress and keep video loss percentages at low levels.

Control overhead (Figure 15) concerns less than 0.8% of video traffic volume and falls between the respective CoolStreaming overhead for 2 and 3 partners.

5.3.8.2 Dynamic Environment

We then conduct experiments under the dynamic environment described in [10]. According to this setup, peers may leave (or fail) and join freely and their lifetime is exponentially distributed with an average of T seconds. As Figure 16 indicates, our protocol presents higher continuity index than *CoolStreaming* mainly because most departures/ failures are caught by the *LR* before discontinuity malfunctions occur at lower levels. Furthermore, in the absence of *LR*, a leaving peer will directly

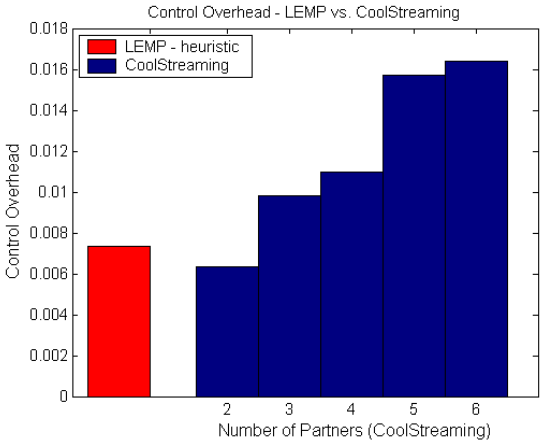

Figure 15. Control Overhead – Streaming Bitrate 500Kbps

inform his children and they will consequently request the video from the server without undergoing any video loss.

6. Emulation

Since scalability is a key issue to the success of *P2P* video streaming systems, we also evaluate *LEMP* rehauled under an extensive emulation performed on the *Emulab* testbed.

6.1 Emulation Methodology

To allow experiments with a very large number of nodes, Emulab provides a *multiplexed virtual node* implementation (hereafter known as just "virtual nodes").

Virtual nodes fall between simulated nodes and real, dedicated machines in terms of accuracy of modeling the real world. A virtual node is just a lightweight virtual machine running on top of a regular operating system. In particular, virtual nodes are based on the FreeBSD jail mechanism, that allows groups of processes to be isolated from each other while running on the same physical machine. Emulab virtual nodes provide isolation of the filesystem, process, network, and account namespaces. That is to say, each virtual node has its own private filesystem, process hierarchy, network interfaces and IP addresses, and set of users and groups. This level of virtualization allows unmodified applications to run as though they were on a real machine. Virtual network interfaces are used to form an arbitrary number of virtual network links. These links may be individually shaped and may be multiplexed over physical links or used to connect virtual nodes within a single physical node.

Once the *vnodes* implementation was made clear, there were several other issues to be solved for the *Emulab* experiments to be feasible. These issues had mainly to do with restrictions associated with the hosting *OS* for *FreeBSD* jails, namely *FreeBSD4.10*. In particular, due to some limitations, *gstreamer* could not be installed so we had to simulate the media stream using a dummy data generation module of our own. Moreover, *iptraf* is strictly designed for *Linux* architecture and cannot be installed on *FreeBSD*. For this, the *iptraf* bandwidth monitoring was replaced by a more primitive approach based on the *netstat* utility. Note however that several medium-scale *Emulab* experiments were performed so as to ensure that the before mentioned changes don't affect our protocol's behavior and performance.

Our transit-stub topologies of 1000 and 1800 nodes were generated using the *GT-ITM* generator. Link delays were automatically assigned using the *sgb2ns* tool and are proportional to the distance between nodes with their values ranging from 2-187 ms. Peers inter-arrival time and lifetime follow the same *pareto* and *lognormal* distributions mentioned in paragraph 5.1. The video stream is 6000 seconds long and the streaming rate is 150 kbps. Finally, all experiments were performed with the latest protocol version (*"LEMP heuristic"*) and unexpected failures were set to 5%, unless stated otherwise.

6.2 Emulation Results

Startup delay measurements do not show significant differences with the PlanetLab results, with 90% of peers receiving the video in no more than 10 seconds and only 4% of the population having to wait for more than 16 seconds to receive the video. This observation indicates that startup delay is independent of the total number of peers.

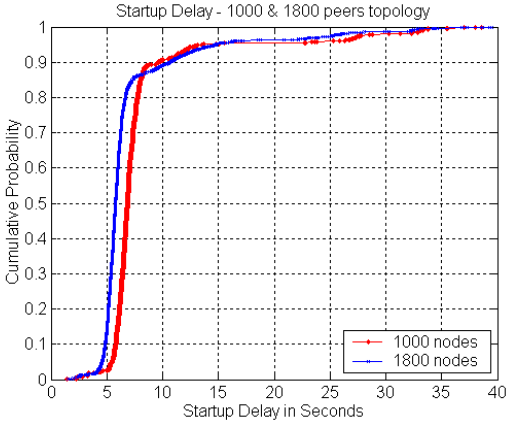

Figure 17. Emulab Startup delay in seconds

Video loss percentage remains at very low levels with 95-96% of the population experiencing no video loss and 99% of the clients losing less or equal to 6% of the video during their lifetime.

Stress stress (as measured by the number of peers receiving the video directly from the server) is displayed in Figure 19. We see that server stress is slightly improved compared with previous experiments with 88% of peers receiving the video exclusively from other peers and 90-91% of the total population receiving less than or equal to 30% of the video directly from the server.

Similarly to previous experiments, only a small percentage of peers (4%) received the video directly from the server. Figure 19 thus, demonstrates once again the scalability of the protocol.

Average outgoing total, data and control rates present no significant differences from the previous experiments. This observation leads us to the conclusion that peers' stress is stable and has nothing to do with the overall peer population,

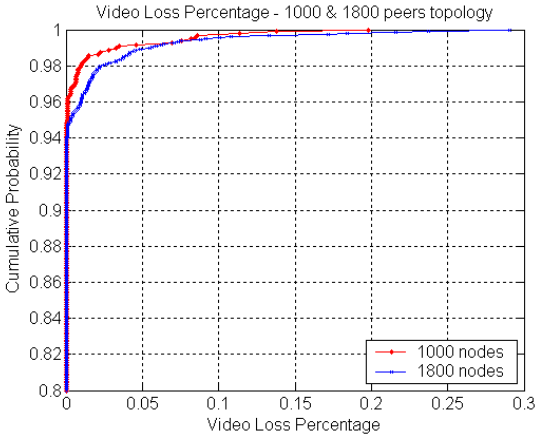

Figure 18. Emulab Video Loss Percentage

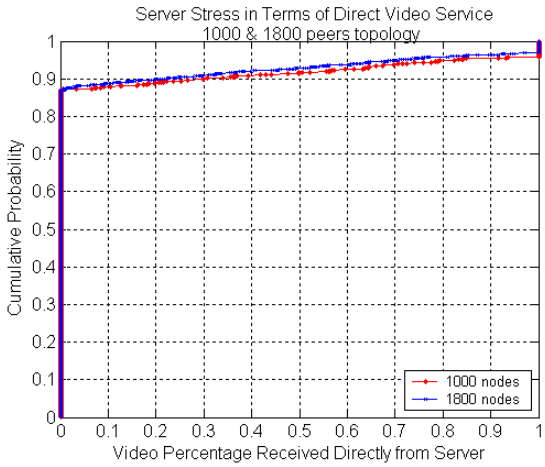

Figure 19. Emulab server stress savings as measured from video percentage served directly from the server for each peer

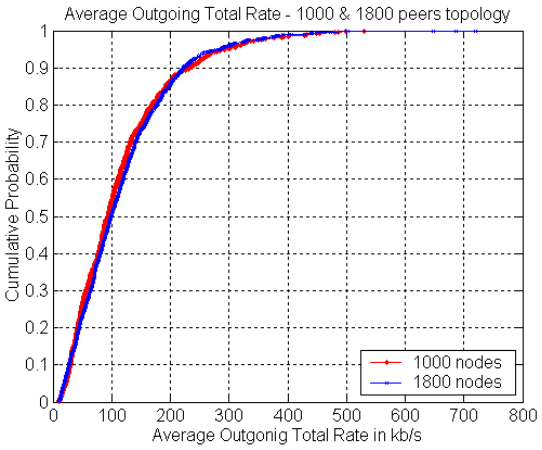

Figure 20. Emulab Average Outgoing Total Rate

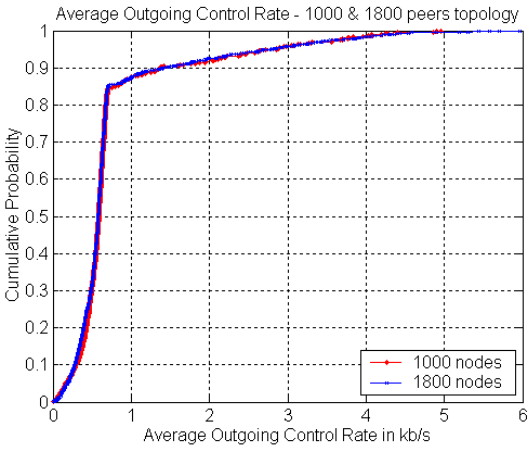

Figure 22. Emulab Average Outgoing Control Rate

since the protocol organizes clients into levels, with each peer (p_i) being associated only with its own level (L_i) and the next one (L_{i+1}).

An issue that required attention was whether the achieved fairness of the protocol (*"LEMP heuristic"*) will be sustained for larger scale experiments. According to Figure 23, the protocol appears to be strictly fair as far as *LR*s' and normal peers' stress is concerned, with the two distributions being almost identical.

Since the control hierarchy remains unchanged, it is normal to expect that the average outgoing control rate among *LR*s and normal peers (Figure 24) will present the observed variations, with normal peers experiencing a pretty stable control overhead (around 0.55kb/s) and *LR*s being clearly more stressed. Still, the average control outgoing rate remains at low levels similarly to previous experiments.

Being interested in evaluating our protocol's robustness under abnormal conditions, we present plots of the two metrics likely to be affected under

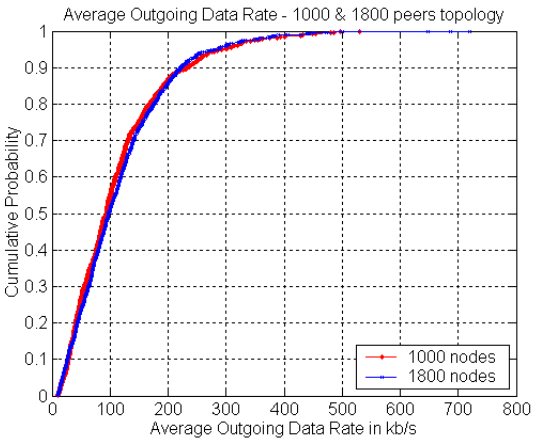

Figure 21. Emulab Average Outgoing Data Rate

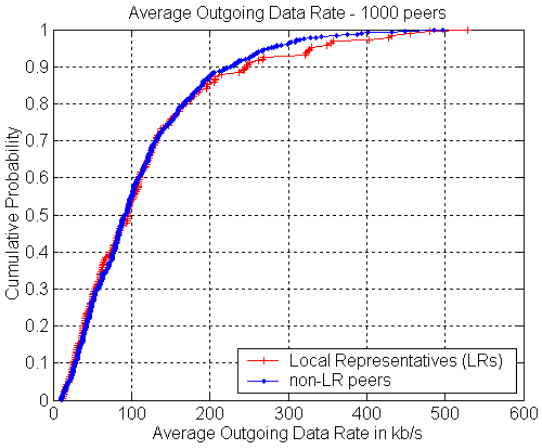

Figure 23. Emulab Average Outgoing Data Rate for LRs and non-LRs

increased unexpected failures, namely video loss and server stress in terms of direct video service.

As Figure 25 depicts, increased unexpected failures have an impact on video loss percentage but still the high majority of peers receives the video with minimum video discontinuities. Even under heavy unexpected failures (20%), 90% of total population loses no more than 2% of the video and 96% of peers loses less or equal to 5% of the video. Higher percentages may be considered as outliers since all video loss percentages are over peers' lifetime, meaning several clients might have lost just a few segments of the video but their lifetime was too short. In order to prove our claim we have plotted video loss percentages and peers lifetime for all four unexpected failure scenarios. Figure 26 shows that the vast majority of high video loss percentages (i.e. more than 10%) mainly concern peers with short lifetime.

When it comes to video percentage received directly from the server, our system appears to be robust and stable despite the demanding scenarios

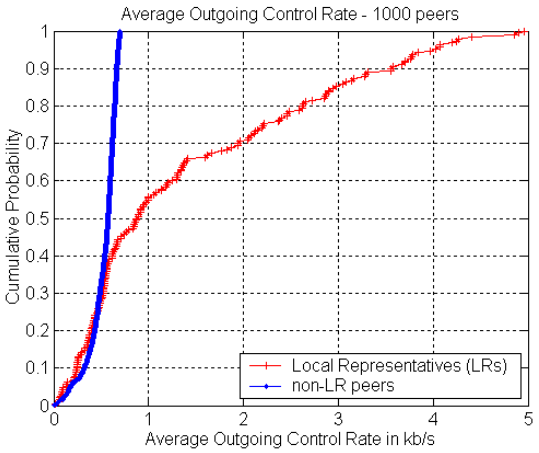

Figure 24. Emulab Average Outgoing Control Rate for LRs and non-LRs

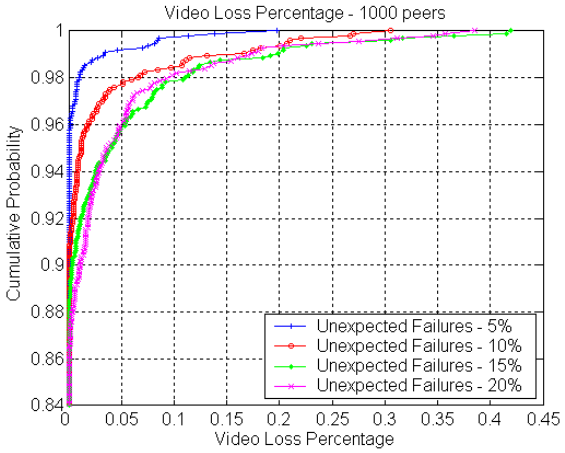

Figure 25. Emulab Video Loss under increased unexpected failures

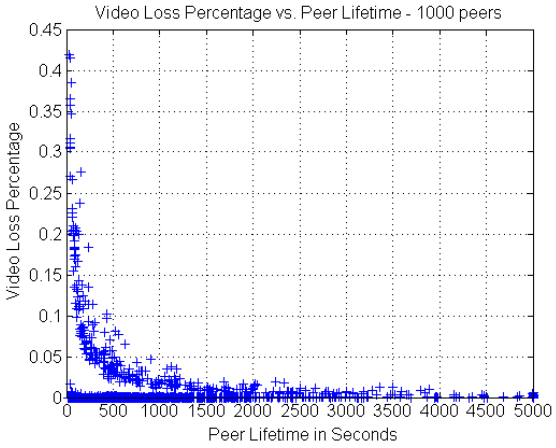

Figure 26. Emulab Video Loss and Peers' Lifetime for all unexpected failures scenarios

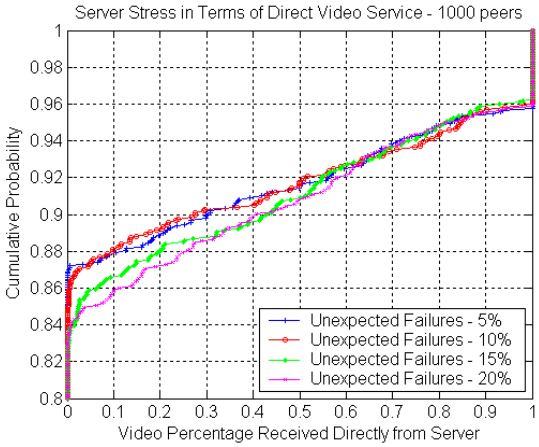

Figure 27. Emulab server stress in terms of direct video service under increased unexpected failures

of increased unexpected failures. According to Figure 27, unexpected failures up to 10% don't seem to affect server stress while higher percentages of failures (15% and 20%) have only a moderate impact on server bandwidth demands with 84% of peers receiving the video exclusively from other peers and 90% of the population receiving no more than 44% of the video directly from server. Still the percentage of peers receiving the whole video directly from the server remains low (4%) and quite unchanged.

7. Related Work

Adapting the advantages of peer to peer architecture on (video) streaming technology is a relatively unexplored research field which however has already attracted some attention. Several proposals [1, 2, 3, 4, 5, 12, 13, 20] focus on *VoD* media while other approaches [6, 7, 8, 9, 10, 11] deal with *live* video streaming.

In most proposed approaches, peers are organized in an application layer multicast tree. In [1, 2, 6, 7, 9, 11, 13] such a multicast tree is used as the building block for the peers' coordination. The main difference with our protocol is that in these systems, data and control hierarchy are unified; *LEMP* however, uses a tree-like hierarchy for data flow and a dual star-like architecture for control messages. Other differences concern client arrangement, as in [1], [2], [7] and [11] where control management and maintenance are performed totally by the server and no "special" roles are assigned to peers (e.g. *LRs* in *LEMP*). Moreover, some proposals ([2], [7]) first exhaust the server and then reside to the P2P approach while at the same time they deal with service interruptions only theoretically, unlike *LEMP*. [11] and [13] use techniques such as patching and shifted stream forwarding in order to handle disruptions while Venkata N. Padmanabhan et al. [9] use multiple diverse distribution trees (and thus reassure redundancy in network paths) as well as Multiple Description Coding (meaning redundancy in data). Similarly to *LEMP*, the authors of [6], propose the partitioning of clients to clusters with each cluster having a head and an associate-head

responsible for monitoring the membership of the cluster and transmitting the content to cluster members respectively. However, over a new client request the server has to run an algorithm so as to forward the request to another peer, while in our system this is done immediately, since the request is straightly forwarded to the *LR* of the previous level, who is already known.

Having explored the tree-based approaches, we'll now look into p2p video streaming approaches which are based on different building blocks (e.g. cell, mesh, unstructured overlays).

An inspired proposal is the one made in [3], in which the authors insert the *cell* definition in order to deal with two challenges; namely how can a host find enough video pieces to assemble a complete video and what part of a video a host should cache, given a limited buffer size. A cell is a cluster of hosts which together can supply a complete video. Peers' discovery is made easier and data redundancy is clearly supported. Still, only simulation results are presented while vital metrics such as delays and server stress are not addressed. A proposal based on the JXTA project (which hides/ tackles all streaming routing and storage re-allocation details) and RSVP is presented in [4], which however only provides simulation results and does not address scalability. In contrast to *LEMP*, Alok Nandan et al. [5] and Mohamed Hefeeda et al. [12] propose prototypes which are independent of the underlying p2p substrate and may be deployed over other overlays (e.g. Pastry, Chord, Can). The latter infers and exploits properties of the underlying network (topology and performance), resulting in higher system performance. Systems based on mesh overlays are presented in [10] and [18]. In the first case, an inter-overlay scheme (based on multiple tree structures) is used and parallel downloads are supported via the use of a *buffer manager*. In the second case the mesh-based approach is used in conjunction with other DHT systems, such as Chord or Pastry, while peers adopt static local storage of data (and not a "*cache and relay*" scheme as in *LEMP*).

Mea Wang et al. [14] and Meng Zhang et al [15] propose a random packet scheduling strategy for their protocols. Specifically, in [14], a simple and probabilistic scheme ("*perfect collaboration"*) is combined with video segmentation under a pure push-based approach, unlike our pull-based protocol. In this way, explicit requests are eliminated, control overhead is minimized and there is shorter initial buffering delay (10-20 seconds).. In [15], both a pure pull-based and a pull-push hybrid protocol are examined, using the metric of peer bandwidth utilization and system throughput. However, we believe that in such time-constrained systems, metrics such as delay and video continuity should also be taken into account.

Two proposals applicable to controlled environments are presented in [17] and [19], while our protocol is mainly intended for a wider deployment. The first one uses a push-pull based method and video segmentation while the latter focuses on IPTV over FTTN/xDSL networks. Another protocol based on a pull-push approach is the one presented in [8]. Their GridMedia prototype is built over an unstructured overlay (unlike *LEMP*) where each peer starts with the pull method (and delivers a packet only when there is an explicit request) and then turns to push method (that is relay packets to its neighbors for predefined time intervals).

Stefan Birrer et al. [16] present a comparison of resilient overlay multicast approaches and specifically examine three different techniques: cross-link, in-tree and multiple tree redundancy. The paper mentions the advantages of each method and concludes (via both simulation and real-experiments on PlanetLab) that the combination of in-tree and multiple-tree redundancy achieves highest delivery ratio under different failure scenarios; this is something that will be taken into account in our future work. Finally, Yan Huang et al. present challenges and architectural design issues of a large scale P2P *VoD* system, based on real experiments on their *PPLive* system. Segmentation, replication, content discovery, piece selection, transmission strategy and NAT/ firewall issues are taken into consideration. The authors only focus on *VoD* but their work includes extensive theoretical analysis as well as experimental results and thus provided a guideline for our research as well.

8. Conclusion and Future Work

In this thesis, we presented the design of *LEMP* with emphasis on a set of improvements that resulted in a complete rehaul of the original protocol. These improvements aim to a fast, light-

weighted, fault-tolerant and self-recoverable application-layer *P2P* protocol for live video streaming.

We extensively evaluated the performance of *LEMP* rehauled on the PlanetLab and Emulab testbeds, over a number of metrics and the results demonstrated that our protocol is effective in terms of startup delay, video continuity, control overhead and server stress savings. Moreover, our comparative analysis showed that *LEMP* rehauled outperforms CoolStreaming as continuity index and control overhead are concerned.

For our future work, we are exploring parallel download support approaches to achieve higher utilization without heavily interfering continuous playback and control overhead.

References

[1] Cheng Huang, Jin Li, and Keith Ross, "Peer Assisted VoD: Making Internet Video Distribution Cheap" IPTPS, Bellevue, WA, Feb. 2007.

[2] Anwar Al Hamra, Ernst W. Biersack, Guillaume Urvoy-Keller, "A Pull-Based Approach for a VoD Service in P2P Networks" IEEE HSNMC, Toulouse, France, Jul. 2004.

[3] Ying Cai, Zhan Chen, Wallapak Tavanapong, "Video Management in Peer-to-Peer Systems" IEEE International Conference on Peer-to-Peer Computing, Germany, 2005.

[4] Chris Loeser, Peter Altenbernd, Michael Ditze, Wolfgang Mueller, "Distributed Video on Demand Services on Peer to Peer Basis" In Proceedings of the First International Workshop on Real-Time LANs in the Internet Age (RTLIA2002).

[5] Alok Nandan, Giovanni Pau, and Paola Salomoni, "GhostShare – Reliable and Anonymous P2P Video Distribution" *Global Telecommunications Conference Workshops, 2004. GlobeCom Workshops 2004. IEEE* (2004)

[6] Duc A Tran, Kien A. Hua, Tai T. Do, "A Peer to Peer Architecture for Media Streaming" Selected Areas in Communications, IEEE Journal on, 2004

[7] H. Deshpande, M. Bawa, H. Garcia-Molina, "Streaming Live Media over a Peer to Peer Network" Stanford database group technical report (2001-20), Aug. 2001

[8] Meng Zhang, Li Zhao, Yun Tang, Jian-Guang Luo, and Shi-Qiang Yang, "Large-Scale Live Media Streaming over Peer to Peer Networks through Global Internet" *P2PMMS'05*, November 11, 2005, Singapore.

[9] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, "Resilient Peer to Peer Streaming" Technical Report MSR-TR-2003-11, Microsoft Research, Redmond, WA, March 2003.

[10] Xiaofei Liao, Hai Jin, Yunhao Liu, Lionel M. Ni, and Dafu Deng, "AnySee: Peer to Peer Live Streaming" INFOCOM 2006. 25th IEEE International Conference on Computer Communications.

[11] Meng Guo, Mostafa H. Ammar, "Scalable Live Video Streaming to Cooperative Clients Using Time Shifting and Video Patching" in Proc. of IEEE INFOCOM, 2004.

[12] Mohamed Hefeeda, Ahsan Habib, Boyan Botev, Dongyan Xu, Bharat Bhargava, "PROMISE: Peer to Peer Media Streaming Using CollectCast" *ACM Multimedia 2003*, Berkeley, CA, 2003,

[13] Yang Guo, Kyoungwon Suh, Jim Kurose, and Don Towsley, "P2Cast: Peer to Peer Patching Scheme for VoD Service" Proceedings of the 12th international conference on World Wide Web, Budapest Hungary , 2003

[14] Mea Wang, Baochun Li, "R^2: Random Push with Random Network Coding in Live Peer-to-Peer Streaming" Selected Areas of Communications - Advances in Peer-to-Peer Streaming Systems, IEEE Journal on, December 2007

[15] Meng Zhang, Qian Zhang, Lifeng Sun, Shiqiang Yang, "Understanding the Power of Pull-based Streaming Protocol: Can We Do Better?" Selected Areas of Communications - Advances in Peer-to-Peer Streaming Systems, IEEE Journal on, December 2007

[16] Stefan Birrer, Fabian E. Bustamante, “A Comparison of Resilient Overlay Multicast Approaches” Selected Areas of Communications - Advances in Peer-to-Peer Streaming Systems, IEEE Journal on, December 2007

[17] Kyoungwon Suh, Cristophe Diot, Jim Kurose, Laurent Massoulié, Cristoph Neumann, Don Towsley, Matteo Varvello, “Push-to-Peer Video-on-Demand system: design and evaluation” Selected Areas of Communications - Advances in Peer-to-Peer Streaming Systems, IEEE Journal on, December 2007

[18] W.-P. Ken Yiu, Xing Jin, S.-H Gary Chan, “VMesh: Distributed Segment Storage for Peer-to-Peer Interactive Video Streaming” Selected Areas of Communications - Advances in Peer-to-Peer Streaming Systems, IEEE Journal on, December 2007

[19] Yennun Huang, Yih-Farn Chen, Rittwik Jana, Hongbo Jiang, Michael Rabinovich, Amy Reibman, Bin Wei, and Zhen Xiao, “Capacity Analysis of MediaGrid: a P2P IPTV Platform for Fiber to the Node (FTTN) Networks” Selected Areas of Communications - Peer-to-Peer Communications and Applications, IEEE Journal on, January 2007

[20] Yan Huang, Tom Z. J. Fu, Dah-Ming Chiu, John C. S. Lui and Cheng Huang, “Challenges, Design and Analysis of a Large-scale P2P-VoD System” in Proc. of Sigcomm 2008.

[21] Eveline Veloso, Virgılio Almeida, Wagner Meira, Azer Bestavros and Shudong Jin, “A Hierarchical Characterization of a Live Streaming Media Workload”, In *Proceedings of ACM SIGCOMM Internet Measurement Workshop (IMW)*, November 2002.

[22] X. Zhang, J. Liu, B. Li, and T. P. Yum, "DONET: A Data-Driven Overlay Network for Efficient Live Media Streaming", in Proceedings of IEEE INFOCOM, 2005.

[23] "YouTube serves up 100 million videos a day online". USA Today. 2006-07-16. http://www.usatoday.com/tech/news/2006-07-16-youtube-views_x.htm. Retrieved on 2008-11-29.

[24] Panayotis Fouliras , Spiros Xanthos , Nikolaos Tsantalis , Athanasios Manitsaris, “LEMP: Lightweight Efficient Multicast Protocol for video on demand”, Proceedings of the 2004 ACM symposium on Applied computing, March 14-17, 2004, Nicosia, Cyprus