Kangaroo: Video Seeking in P2P Systems

Xiaoyuan Yang[†], Minas Gjoka[‡], Parminder Chhabra[†], Athina Markopoulou[‡], Pablo Rodriguez[†]
[†] Telefonica Research, {*yxiao, pchhabra,pablorr*}*@tid.es*
[‡] University of California Irvine, {*mgjoka, athina*}*@uci.edu*

Abstract

A key challenge faced by peer-to-peer (P2P) video-on-demand (VoD) systems is their ability, or lack thereof, to provide DVD-like functionality, such as pause, forward and backward seeking (or *jumps*). Such operations can significantly degrade the performance of a P2P system as arbitrary video segments may need to be served timely on demand. Currently, little is known of the impact that these operations can have on the swarm efficiency, user experience and server load. In this paper, we design and implement a novel P2P system called *Kangaroo*, which supports DVD-like jumps. Using a carefully designed peer topology management, hybrid block scheduling algorithms, and a smart tracker, *Kangaroo* provides low buffering times and high swarming throughput under jump operations, without the need for overly provisioned peers or server. We experimentally evaluate the performance of the system using live VoD traces captured from a large commercial IPTV network.

1 Introduction

Peer-to-peer (P2P) systems have been successful in distributing files to a large number of users. P2P systems are also widely used for distribution of video content, including video downloads (where users need to completely download the file before they can watch the video) and live media streaming (such as Coolstreaming). Recently, new systems [1, 10, 11] have been designed to support video-on-demand (VoD) using P2P. Although such VoD systems allow users to start watching the video at any time, they assume that viewers watch from start to finish without any jumps. Supporting DVD functionality is a natural requirement for most VoD systems. For example, the most popular centralized VoD services, such as YouTube or home theater solutions, support such seeking functionality. However, DVD functions have been largely ignored by many P2P VoD systems. Understanding the performance impact of realistic DVD operations and optimizing the system design for their support is a step forward towards fully functional P2P VoD systems.

Designing a VoD system with DVD features using P2P technologies is non-trivial. The main difference in designing for DVD operations vs. standard VoD is that in VoD, younger peers can pull resources from older ones that contain early parts of the video, thus creating a "cascade" of peers based on age. However, with DVD operations such aged-based cascades do not work well. Instead, peers' neighborhoods need to be constantly readjusted to match peers that have roughly the same playing point. If one does not match peers with each other properly as they jump, the chance of sharing video blocks decreases, and receiving the video in a timely manner often requires pulling data from the server. This is especially critical during flash crowds, since the server is the only node in the system with a copy of the file.

The challenge is to design a P2P system that meets the user VoD requirements (low delay and sustained playout rate), while minimizing the amount of data that has to be delivered from the server (and thus the server capacity) under arbitrary jump patterns. In an ideal scenario, the server should only be used if there are not enough peers in the system to satisfy the demand created by jump operations. However, scheduling data, finding the right peers or freeing their resources on time is a non-trivial task.

To this extent, we design, build and deploy *Kangaroo*: a P2P mesh-based system that can support jumps efficiently. The goal is to provide users with high-quality DVD services, i.e., with low latency after jump operations and high throughput. To achieve this goal, Kangaroo implements (i) a hybrid scheduling policy that combines selfish (sequential segment downloads for continuous playback) with altruistic (local rarest to improve segment diversity) behavior, (ii) a neighborhood manager that constantly re-visits the peer's neighborhood and decides which are the best peers to get/push data from/to and (iii) a scalable tracker implementation, which helps to find new peers with similar playback points with little delay when jumps occur.

2 Architecture

Kangaroo resembles a typical mesh-based P2P system in that it consists of *Peers* coordinated by a *Tracker*. Content is hosted by a special peer called a *server* or *seeder*. There is one swarm per media file. A media file is split into segments of equal size. By default, we choose a segment size of 64KBytes to keep the buffering times low. Each peer downloads data in parallel from a small number of neighbors through *data connections*. Peers also maintain a number of *control connections* which are used to exchange information about available segments in a neighborhood, thus enabling the peer to infer the popularity and location of the segment for scheduling. We strive to keep the number of control connections bounded as the overhead of exchanging control information with a large number of peers could be significant. Data connections are established with a subset of peers with whom the peer has control connections. We use persistent HTTP/1.1 connections to allow for seamless integration with web proxies: segments can be cached by transparent proxies in the path, which in turn can behave as extra seeding resources for the swarm.

2.1 Peers

A peer consists of several sub-components, the Segment Scheduler, the Peer Selection Scheduler, and the Neighborhood Manager, which we discuss next. The Segment scheduler decides what segment should be scheduled for download next, while the Peer Selection Scheduler decides which neighbor peer(s) to schedule the download from and the Neighborhood manager decides the composition of a peer's neighborhood.

Segment Scheduler. The segment scheduler faces the following trade-off: on one hand, a peer wants to download the next segments for its own continuous playback; on the other hand it wants to download the local-rarest segment to help the swarm performance. We choose to use a *hybrid* sequential and local-rarest policy. Since each peer can have a maximum of five data connections, it initially starts by downloading the next four segments from its play point (greedy strategy) to minimize jump delay and ensure continuous playback; it also downloads one local-rarest segment to be altruistic to the neighbors. Furthermore, over the course of the download, and depending on the swarm performance and the number of segments in the buffer, the allocation of segments between sequential and local-rarest policies varies dynamically. Since our algorithm constantly adapts the ratio of sequential vs. local-rarest segments, depending on the buffer size and the deadline of each segment, it differs from other works that statically combine greedy with local-rarest [3]. Pseudo-code of the algorithm implementing such adaptive hybrid scheduling policy is shown in Algorithm 1.

Algorithm 1 Hybrid Scheduling Algorithm

```
1: T_i = Time-stamp when the system needs segment i.
2: T̂_i = Time-stamp when the system expects to receive segment i based on
   current download rates.
3: N = Max. num of simultaneous connections.
4: G = Num of greedy-download connections.
5: R = Num of local-rarest download connections.
6: R ← 1
7: In a Window of size W closest to the current play point
8: while R < N do
9:     Find T̂_i for segment i in W
10:    if T̂_i < T_i then
11:        R ← R + 1
12:    end if
13: end while
14: G = N - R
15: Schedule G sequential segments and R local-rarest segments in window W
```

Neighborhood Manager. This sub-component essentially does admission control on requests arriving from other peers. It may deny peer requests if there are no more connections available, i.e., when the number of active connections is equal to the maximum number of connections. However, when a peer refuses to upload a segment, its neighborhood manager also suggests the location of another peer that has the segment in order to facilitate peer matching. Even if the maximum number of connections has been reached, the seeder can make an exception and accept the extra connection: a request is always accepted if the segment has exceeded a delay-tolerance parameter, which captures the maximum delay that a peer can tolerate for each segment. If the delay tolerance is exceeded and the peer cannot find the segment in its neighborhood, then the peer can go to the seeder as a last resort; this policy bounds the jump delay at the expense of potentially more capacity at the seeder.

The peer neighborhood manager also reshuffles a peer's neighborhood when necessary and maintains it "healthy". In particular, each peer periodically calculates the *health factor* h of its neighborhood, which is defined as the percentage of useful segments that a peer's neighborhood has. We define *useful segments* to be those that are within a window W of the currently playing segment. If h falls below a pre-defined threshold t, then: more collaborators are requested from the tracker, the connections with the non-useful peers are reset and new neighbor relationships are created. A careful choice of parameters t and W is necessary to strike a good balance between getting data from the *server* and often contacting the *tracker*. E.g., if W is too short or t is too high, frequent updates will be triggered to keep a highly healthy neighborhood; conversely, a small value of t will result in more segments being requested from the server since neighbor peers will not hold interesting content.

Peer Selection Scheduler. Given a neighborhood, a peer decides to request a segment from neighbors that have the smallest number of useful segments. The rationale is to avoid overloading peers that have segments that can potentially be useful for other peers too. Since the server holds all the segments, this also prevents peers from going to the server unless strictly necessary (e.g., when the server is the only node in the neighborhood that holds the missing segment) and avoids a synchronization

effect where multiple peers request the same segment repeatedly from the server.

Another key functionality of the peer selection scheduler is the propagation of *Have* messages. Every time a peer successfully downloads a segment, it issues *Have* messages to let its neighbors know of the existence of that segment. Because flooding the neighbors with *Have* messages would cause a high overhead, we try to minimize the number of such updates as follows. When a peer knows that a neighbor peer already has the segment, the *Have* message of that segment is batched. As soon as the peer downloads a segment that the neighbor does not have, it sends the *Have* message for the downloaded segment together with all previously batched messages.

Other considerations. Due to space limitations, there are certain aspects of the system that we have not considered in this paper. In particular, we do not consider incentive mechanisms for cooperation in VoD P2P systems where different peers may target different parts of the file with different playout deadlines. However, there are a number of research efforts in this area such as "give to get" [15]. Similarly, we do not consider security threats that could arise from peers lying about segments, number of connections. Our system could, however, be deployed as is under a closed and controlled system where incentives are not required and tampering the system is harder (e.g. on top of IPTV set-top-boxes).

2.2 Tracker

A key function and design feature of Kangaroo is its tracker implementation. The role of the tracker is to allow quickly discovery and mesh together peers that have content to exchange (which is particularly critical during jump operations) as well as to maintain healthy neighborhoods. This function has to scale with the number of peers and possible jump points.

In file sharing applications, this is easy: a random selection works well because peers are interested in the entire file. In P2P DVD systems, peers' needs are guided by their current playback point. Hence, a tracker needs to carefully choose and provide a list of neighbors so as to ensure a good performance and a low jump delay. In essence, the tracker needs to estimate the current playback point of all peers in the system and suggest suitable peers that have the needed content; and this has to scale with the number of peers and possible jump points.

We base the design of our tracker on the following two intuitive observations: (i) users who are roughly at the same playback point benefit by collaborating, and (ii) in between jumps, users play the video sequentially. Every time a peer needs a list of neighbors, it contacts the tracker with its current playback point. Based on the playback point, the neighborhood selection algorithm at the tracker invokes two mechanisms: (1) the Smart Neighborhood Selection (SNS) that selects a list of peers that are at the same playback point as the playback time in the request and (2) the History-based Neighbor Selection (HNS) that selects peers, which are not at the same playback point but contain the video portions of a given play point. These two may not be the same as peers may perform arbitrary jumps and may hold non-contiguous parts of the video. If no suitable peers are found by (1) or (2), then, we attempt to get the peers that are as close as possible to the play point of the requesting peer.

For every peer request at the tracker, the tracker would potentially need to search over all peers and all potential play points to find the right matches, which is an expensive operation and clearly does not scale. Next, we describe how the Kangaroo tracker implements SNS and HNS in an scalable way.

Smart Neighborhood Selection (SNS). When the tracker receives a peer's request for a list of neighbors, it needs to know the point of the video this peer plays. Since the play point increases with time, a naive tracker would have to re-calculate the play point of every peer at every request, thus incurring a very high overhead. To avoid a per-peer computation, we created a hash table that keeps in each entry all peers that are progressing together in the same part of the video. We use a hash key that is a function of the video time when the jump happened, the final jump point and the session jump time.

For a jump operation at video time T_j, peer j reports the jump point, P_j to the tracker. The tracker generates a hash key K_j as: $K_j = \frac{P_j - T_j}{C}$ where $C > 1$ is the granularity of the mechanism to predict the play point of peers; a good choice for C depends on the length of the movie and the user behavior. Then the peer is removed from the old hash entry and inserted into the new entry with key as K_j. Note that the hash key K_j is static and the tracker does not need to update the peer in the hash table until the next jump operation.

In Fig.1 we demonstate our approach using a simple example. Peer A begins watching a video at time 2 and at time 6 jumps to play point 7. Assuming $C = 1$, the hash key for peer A at the jump is 1. The hash key of peer A is the projection of its play point at jump time. We assume that Peer B has started playing at time 4. At time 8, Peer B jumps to play point 9. At time 9, peer B also has a hash key 1 and the tracker will return A as a neighborhood peer for B.

The value C in the Kangaroo is set to 30 seconds, according to our user behavior trace and for a movie length of 120mins; we also tested other smaller values of C without significant changes. Additionally, to ensure that the tracker does not include stale information (e.g. from peers who left the system), every peer periodically contacts the tracker to inform it of its current play position.

History Neighborhood Selection (HNS): While SNS only knows about the current play point of a peer, HNS knows about past playing points. This is more complicated as it requires a separate history table containing whether some peers visited a given segment. Keeping history about a peer in the order of fractions of a minute

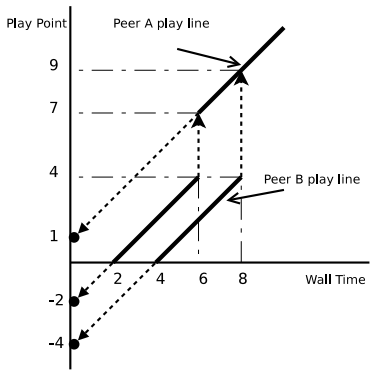

Figure 1: An example that demonstrates the logic behind the SNS grouping mechanism of the tracker.

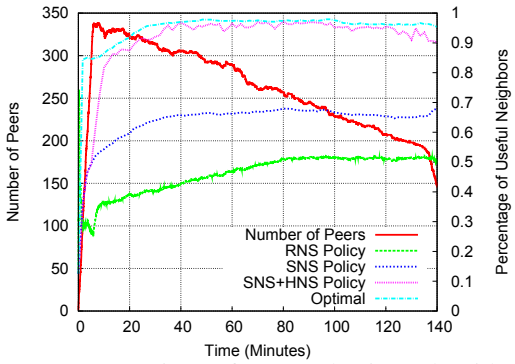

Figure 2: Comparison of Peer Selection Algorithms at the Tracker.

is memory intensive; searching over it is resource intensive as well. In our implementation, we divide videos in time-fragments of size C as before. The memory requirement is $N_p * \frac{L}{C} *$ size(P_{id}), where L is the length of the movie, P_{id} is the unique peer-id of each peer and N_p is the total number of peers. Each peer has a unique id of 32 bits. Thus, the tracker would require about 18MB of memory to build the history for 20K peers, for a movie length of 120 minutes and with $C = 30$ seconds.

To generate the history table of the content of each peer, peers do not need to periodically report their content to the tracker. Instead, by estimating their current play point the tracker can infer the content of each peer and refresh the history table accordingly. The refresh period affects the prediction: for a large refresh period, the tracker may under-count the content in the peers, thus maintaining an incorrect history. In the experimental evaluation (results not presented due to lack of space), we saw that a refresh time of 1 minute was sufficient to minimize the prediction error for finding peers with requested neighborhoods by current playpoint.

3 Experimental Evaluation

In this section, we evaluate the sub-components of Kangaroo as well as the system as a whole under realistic jump operations. First, we evaluate the performance of the tracker in terms of its effectiveness to quickly provide useful peers for various workloads and peer neighborhood management algorithms. Next, we study the impact of various block scheduling policies. To evaluate Kangaroo, we use a real VoD-DVD trace, which was collected from a live commercial IPTV service run by a large Telco in Europe, with more than half a million subscribers. The trace was collected at one of the VoD servers that provide service over ADSL to one of the 17 regions within the country. The trace spanned a period of 109 days from Feb. 3, 2007 to May 24, 2007, and included a total of 65,498 sessions and 700 unique movies. In the trace, 60% and 90% of the forward and backward jump distances are longer than 1 and 10 minutes respectively and most viewers (90%) perform 10 or less jumps during the period of a movie. Some of the information about these jump patterns is used as part of the input to the experiments described next.

3.1 Tracker Performance

Here, we evaluate the performance of the peer matching algorithms implemented in the Kangaroo tracker. In particular, we compare our two proposed algorithms SNS (smart neighbor selection) and HNS (history-based neighbor selection) against two baselines. The first baseline is a simple algorithm, which we call RNS (random neighbor selection). RNS is the standard algorithm used by most P2P tracker systems today: the tracker provides the requesting peer with a list of n peers, selected at random from a list of active peers. RNS is clearly the most scalable algorithm since it does not require any sophisticated data structure to maintain the candidate list of peers. However, this simplicity comes at the cost of neighborhoods where the peers cannot share enough data, which eventually leads to poor performance. The second baseline for comparison is the optimal neighborhood selection algorithm, which assumes global knowledge of every segment at every peer and performs exhaustive search over all peers to return the peers with the most relevant segments. Although optimal in terms of peer neighborhood formation, this algorithm is prohibitively expensive in terms of the tracker's processing load: for each request, the tracker needs to check all N peers to generate the best neighborhood.

Fig. 2 compares four algorithms, namely RNS, SNS, SNS+HNS, and Optimal. These results are obtained through simulations. We used all 350 sessions from the most popular video of the VoD trace. Peers arrive according to a Poisson process with $\lambda = 1$ peers/sec. Time indicates progress as users jump according to the trace. Most users arrive within a few minutes and the peak swarm size is close to 350 peers. We observe that simpler neighborhood selection strategies perform poorly. Initially, both SNS+HNS and SNS algorithms return about the same percentage of useful neighbors. However, over time, as users jump, the fraction of useful neighbors returned by the tracker is quite small for both SNS and

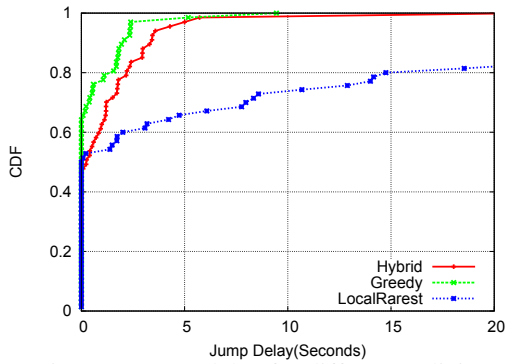

Figure 3: Jump delay for different policies.

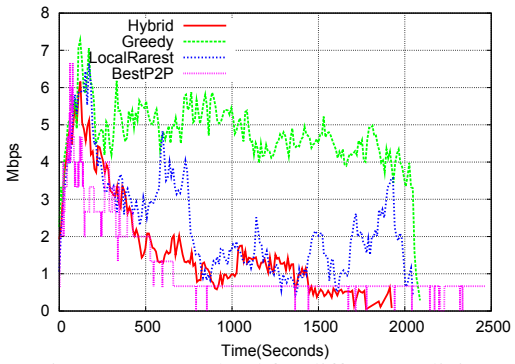

Figure 4: Server load for different policies

RNS, but is kept very high and close to the optimal peer selection strategy for SNS+HNS. SNS does much better than RNS since SNS selects peers that are close to the playback point of the requesting peer and therefore has a higher likelihood of forming useful neighborhoods. SNS+HNS performs nearly optimally and much better than SNS, since HNS can return peers who are not playing at the same playback point but have the requested segment. On the other hand, the SNS policy might not be able to return enough useful neighbors, in which case the tracker complements the returned list with peers playing as close as possible but not close enough to the current play point.

The number of requests received by a tracker is an important test for the scalability of the system. Peers contact the tracker (i) at every jump operation and (ii) when triggered by neighborhood health evaluation. The former depends on the workload while the latter depends on the value of the threshold t which represents an acceptable health factor. To strike a trade-off between tracker response time and good topology connectivity, we picked a value of t = 0.2 . To test the scalability of the tracker, we ran experiments where we used an off-the shelf commodity PC for the tracker and produced requests from jump patterns of up to $16,000$ "dumb" peers at the same time. The tracker is able to respond with less than 0.1 sec delay, a negligible value compared to the transmission time of the first segments of the video (2 sec for a 64 KB segment and a 1 Mbps link). The maximum memory usage observed was 32 Mbytes with CPU usage peaking at 74%. More details of the experiments for health factor impact and tracker scalability are omitted due to lack of space.

3.2 Scheduler Performance

In this section we test another critical component of the Kangaroo system, the segment scheduler. From a user's point of view, we are interested in small delays after each jump operation, thus termed *jump delay*. From the system's point of view, we are also interested in keeping the capacity/bandwidth low at the seeder.

Experimental Setup. We ran experiments in a cluster of 10 machines connected inside a Gigabit LAN and used Modelnet [8] to simulate a realistic Internet environment. Each cluster machine (Intel Core Duo, 2.13 GHz CPU, 2GB RAM) ran at most 10 peer instances. We first used INET [9] to create an Internet topology of 3500 nodes. Link delays in the topology were assigned based on link lengths derived from the INET node location. Each peer instance was connected to the network topology through a different client-stub link in the topology. All packets from the 10-host cluster were routed through a Modelnet emulator which shaped them according to the specified delay and bandwidth of the link in the network topology. We rate limited each peer's access capacity using the modelnet emulator machine to 1.5Mbps. But we left the upload capacity unrestricted at the server (only) ; the demand we will observe over time will then give us an indication of what capacity we need to provision the server with, so as to meet user requirements.

Below, we show results of an experiment with 60 peers for a video file with duration 1024 seconds, split into 2048 segments of 64KB each, and playback rate of 1Mbps. Each peer connects to no more than 15 neighbors. We use jump patterns extracted from the most popular video from the trace and peer arrivals according to a Poisson distribution with $\lambda = 1$ peers/sec. We ran the same experiment for three policies: greedy, local-rarest and the proposed adaptive hybrid policy.

Results. Fig.3 shows the jump delay for the three strategies. The greedy policy achieves the lowest delay because peers use all their bandwidth to download only the segments they need, whereas local-rarest achieves the highest delay because peers are fetching segments for the neighbors. The hybrid policy is very close but not better than greedy because it allocates at least some bandwidth to download rare segments.

However, if algorithms are not efficient, low latency could potentially come at the expense of high server load. Fig.4 shows the server load in the three cases as well as for an idealized baseline, which we call *BestP2P*. In BestP2P, peers have infinite capacity and number of active upload connections, which allows them to instantaneously disseminate all segments once provided by the server. Clearly, BestP2P abstracts away the inefficiencies of any real P2P system and thus outperforms any

P2P system that could be considered as baseline. It captures the minimum possible load on the server, which depends only on the jump patterns in our trace.

We observe that the greedy policy provides a low jump latency at the expense of a very high server load, since peers in the system that do not find the blocks that they need turn to the server. In fact, the average server load is 4.5Mbps and remains high for 2000 seconds. We also observe that the local-rarest policy achieves a lower server load,[1] because it keeps all video segments well represented, but at the expense of high jump delay, since peers do not download segments for their own continuous playback. Our proposed hybrid policy achieves the best compromise between low server load (close to BestP2P) and low jump delay.

4 Related Work

The first P2P video systems were built for live video streaming and included tree-based overlays, such as SplitStream, and mesh-based overlays, such as CoolStreaming and PPLive. The next generation video P2P systems were designed to support VoD including BitOS [11], BASS [10], Redcarpet [1], Toast [14], but without optimizing the system for efficient DVD-like operations. In [6], the authors present an analytical formulation of the impact of various scheduling policies to optimize VoD performance. In [3], the authors describe the challenges faced by a commercial P2P live video system deployed by PPLive, and propose content discovery, replication, and scheduling algorithms to deal with these challenges.

Recent work [12] [7] and [2] discussed some of the issues that can arise when designing P2P systems that support DVD-like functionality. For instance, [12] introduced the concept of anchors to prefetch data in predefined points of the video and allow for jumps to such points. In contrast, the users of our system may jump to any point in the video. In [7], the authors proposed an aggressive prefetching strategy to proactively create multiple copies of every segment on an overlay, thus reducing the dependence on the source. The goal was to ensure that all blocks are replicated regardless of when the set of active peers in the overlay would need them to support current playback. Kangaroo does not require such pre-fetching, but instead provides careful allocation of swarm resources when needed. In [2], the authors propose a gossip protocol over a ring, where each peer keeps some nearby neighbors as well as some remote neighbors following a power-law radius; however, no block scheduling or peer management is considered. In comparison to the above, our contributions are: an entirely mesh-based architecture which includes an adaptive hybrid segment scheduling, a "least useful" peer selection scheduling and a scalable tracker that implements quick and efficient neighborhood matching algorithms.

[1] Interestingly, the load is not as low as one might expect. In the beginning, there is only one copy of the video (at the server) and peers try to increase the diversity of all segments. For a video size 2048Mbits, and an average rate of 4Mbps, the server needs just 512 seconds to upload all the segments. However, as seen in the plot, the server load remains high even after 512 seconds. This is because peer departures reduce the number of replicas for some segments, which need to be reseeded by the server.

5 Conclusions

We have analyzed the impact of realistic DVD operations on the design of P2P systems, and designed and deployed *Kangaroo* – a P2P VoD system optimized for jumps. Using traces captured from a commercial IPTV system, we experimentally evaluated the performance and provisioning of servers and peers. We have carefully optimized Kangaroo to handle DVD operations with minimum delays, network overhead, and server resources. We have also tested our system with several hundred real users during the 2008 Olympic games; the results were similar to what we observed using the trace in this paper. Overall, we believe that *Kangaroo* is a step forward towards fully functional P2P VoD systems and can pave the way to the next generation of IPTV architectures.

References

[1] S. Annapureddy, S. Guha, C. Gkantsidis, D. Gunawardena and P. Rodriguez. Is High-Quality VoD feasible using P2P Swarming. In *WWW*, 2007.

[2] B. Cheng, H. Jin and X. Liao. Supporting VCR functions in P2P VoD Services Using Ring-Assisted Overlays. In *ICC*, 2007.

[3] Y. Huang, T. Z. J. Fu, D.M. Chiu, J.C.S. Lui and C. Huang. Challenges, Design and Analysis of a Large-scale P2P VoD System. In *Proc. of Sigcomm*, 2008.

[4] A. Hu. Video-on-demand broadcasting protocols: A comprehensive study. In *IEEE Infocom*, 2001.

[5] K. Almeroth, and M. Ammar. On the use of multicast delivery to provide a scalable and interactive Video-on-Demand service. In *JSAC*, 1996.

[6] Y. Zhou, D. Chiu and J. Lui. A Simple Model for Analyzing P2P Streaming Protocols. In *Proc. of ICNP*, 2007.

[7] N. Vratonjic, P. Gupta, N. Knezevic, D. Kostic, A. Rowstron. Enabling DVD-like features in P2P Video-on-Demand-Systems. In *ACM P2P-TV Workshop*, 2007.

[8] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kosti, J. Chase, D. Becker Scalability and Accuracy in a Large-Scale Network Emulator In *Proc. of OSDI*, 2002.

[9] C. Jin, Q. Chen, S. Jamin. Inet: Internet topology generator. *Univ. of Michigan TR CSE-TR-433-00*, 2000.

[10] C. Dana, D. LI, D. Harrison, and C. Chuah. BASS: BitTorrent assisted streaming system for VoD. In *Proc. of MMSP*, 2005.

[11] A. Vlavianos, M. Iliofotou, and M. Faloutsos. Enhancing BitTorrent for supporting streaming applications. In *IEEE Global Internet*, 2006.

[12] B. Cheng, X. Liu, Z. Zhang, and H. Jin. A Measurement Study of a Peer-to-Peer Video-on-Demand System. in *IPTPS'07*.

[13] P Marciniak, N Liogkas, A Legout, E Kohler. Small Is Not Always Beautiful. In *IPTPS'08*.

[14] Yung Ryn Choe, Derek L. Schuff, Jagadeesh M. Dyaberi, Vijay S. Pai. Improving VoD server efficiency with bittorrent. In *Proc. of IEEE Multimedia* 2007

[15] J.J.D. Mol, J.A. Pouwelse, M. Meulpolder, D.H.J. Epema, and H.J. Sips. Give-to-Get: Free-riding-resilient Video-on-Demand in P2P Systems. In *MMCN'08*.