# Quality-Adaptive Scheduling for Live Streaming over Multiple Access Networks

Kristian Evensen[1], Tomas Kupka[2], Dominik Kaspar[1],
Pål Halvorsen[1,2], Carsten Griwodz[1,2]
[1]Simula Research Laboratory, Norway [2]Department of Informatics, University of Oslo, Norway
{kristrev, tomasku, kaspar, paalh, griff}@simula.no

## ABSTRACT

Video streaming ranks among the most popular services offered through the Internet today. At the same time, accessing the Internet over public WiFi and 3G networks has become part of our everyday lives. However, streaming video in wireless environments is often subject to frequent periods of rebuffering and characterized by low picture quality. In particular, achieving smooth and quality-adaptive streaming of live video poses a big challenge in mobile scenarios.

Building on the observation that the subjective video experience on mobile devices decreases when quality changes are more frequent than every 1 to 2 seconds, we present a client-side scheduler that retrieves segments of several video encodings over heterogeneous network interfaces simultaneously. By extending the DAVVI streaming platform with support for multihoming, the proposed scheduler's performance is experimentally evaluated. The results show that our scheduler reduces the video interruptions and achieves a higher and more stable average quality over multiple, truly heterogeneous wireless interfaces.

## Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems - Client/Server

## General Terms

Performance

## 1. INTRODUCTION

Video streaming is rapidly increasing in popularity and has become one of the dominant services on the Internet today. For example, the video aggregation site YouTube streams millions of videos per day, and over 20 hours of content is uploaded every hour[1]. In addition, several live streaming services exist. Microsoft's SmoothStreaming [13] is for example used to deliver the 2010 Olympic games quasi-live to millions of people, with only a small time gap between the actual event and video display.

Streaming high-quality video requires a lot of bandwidth. For example, the bit rate of H264-compressed 1080p video is usually around 6 - 8 Mbit/s. If a user is connected to the Internet using a fixed connection, this may not be a problem, but if the connection is established using a wireless link, the available bandwidth often becomes a bottleneck. Even though the maximum bandwidth of most IEEE 802.11 standards (WLAN) exceeds the requirements for 1080p video, the experienced throughput is often insufficient. Link contention, interference or weak signal reception are some phenomena that cause a significant throughput reduction.

A possible way to alleviate the bandwidth problem caused by most wireless technologies is to increase the performance by combining the throughput of multiple network interfaces. Today, many client terminals are equipped with multiple interfaces and are often within coverage range of overlapping access networks. For example, laptops and smart phones can often connect to both WLAN and 3G networks. Delivering a video as a sequence of small, independent movie segments coded in several qualities, as done by Microsoft's torrent-like, HTTP-based SmoothStreaming, enables a stream to adapt to current network fluctuations and potentially allows a single video segment to be distributed over various paths for bandwidth aggregation.

Here, application protocol features, such as HTTP's *range retrieval requests*, allow a file to be logically segmented and to be downloaded over various access networks [5] or from redundant sources [10]. If a client's interfaces are connected to independent networks, the simultaneous use of multiple wireless links can achieve a total transfer bit rate close to the sum of all the individual interfaces' throughput. In addition, taking advantage of request pipelining [6, 9, 10], the aggregated throughput can be further optimized for small segments. However, obtaining a minimal time gap between the live event and the video display – in other words, increasing the *liveness* of a video stream – is a challenge that increases in complexity when scheduling video segments over network interfaces with heterogeneous characteristics.

This paper introduces an adaptive, pull-based scheduler that achieves smooth playback by scheduling requests for video segments of different quality levels over multiple interfaces simultaneously. The application-layer scheduler is implemented in our experimental testbed as an extension to the DAVVI [4] video streaming platform, which allows live multimedia streaming from commodity web servers.

[1]http://www.youtube.com/t/fact_sheet


*NOSSDAV'10,* June 2–4, 2010, Amsterdam, The Netherlands.

Our results show that the combined operation of multiple interfaces significantly enhances the quality of live video streaming. Even in a truly heterogeneous environment with WLAN and HSDPA links, the presented scheduler achieves an increased video quality and reduces playback interrupts.

The rest of this paper is organized as follows. Section 2 continues with a brief presentation of related work, before the architecture and testbed setup of our multilink-enabled live streaming platform are introduced in Sections 3 and 4. The results of our experiments are discussed in Section 5. Finally, our findings are summarized in Section 6.

# 2. RELATED WORK

File segmentation is a commonly used technique for improving the performance of multimedia streaming and content distribution systems. The allocation of video segments on multiple replica servers or proxies can significantly reduce response times and allows for scalable and dynamic adaption of video quality to changes in throughput. Commercial examples that employ a pull-based distribution of video segments include the solutions by Move Networks [7], Apple's QuickTime Streaming Server [1] and Microsoft's Smooth-Streaming (Internet Information Services) [13]. A comprehensive introduction to decentralized distribution infrastructures is provided in [3].

Choosing the server that provides the best user experience is a non-trivial problem that has been studied extensively. Parallel access schemes, such as those described in [10] and [12], attempt to automatically shift the load from congested servers to less utilized ones. These parallel download techniques are motivated by the fact that the throughput bottleneck is often caused by unequally popular servers or by overloaded routers. Analogously, in mobile scenarios, the bottleneck is usually at the wireless access link, which creates a similar scheduling problem. However, the parallel access schemes are not suitable for achieving quasi-live streaming (sometimes referred to as "progressive download"), because they lack a notion of deadlines when a data segment is required for playout. Additionally, an automatic adaptation of video quality to the currently experienced throughput – by choosing from a set of quality layers – introduces additional scheduling complexity that is not solved by these parallel access schemes.

Although our solution is readily extensible to support multiple servers, our current research targets client-based performance improvements of concurrently utilizing multiple network interfaces. A similar goal is pursued in [11], where a server is enabled to send live video streams over multiple TCP connections to a client. However, such push-based solutions cannot easily be extended to multiple servers. In addition, push-based solutions (e.g., [2]) have limited knowledge about the client-side connectivity and hardly adapt to situations of network interfaces going down or user devices moving out of the coverage area of an access network.

# 3. ARCHITECTURE

The user-experienced quality of live video streaming is highly dependent on the available bandwidth. To show how effective multiple independent links can be used in a live video streaming environment, we have enhanced the DAVVI streaming system [4] to support more than a single network interface. The functionality of DAVVI is therefore described in more detail in the following subsections, followed by an explanation of how the data delivery functionality was extended with multilink support and a request scheduler.

## 3.1 Video streaming and adaptation

DAVVI is an adaptive, segmented, torrent-like HTTP-based streaming system, where users can, in addition to traditional video playout, search for particular events and use the search results to compose a personalized, on-the-fly, content-based video summary. The basic idea of the streaming solution is to divide video objects into independent (closed-GOP), 2-second video segments. Furthermore, the video segments are then downloaded from traditional, potentially different, web servers. In order to adapt to available bandwidth, each video segment is stored in multiple quality levels, producing different, independent files with different bitrates. This differs from SVC, which has one base layer and multiple enhancement layers. The segment download component can therefore, based on download speed and buffer occupancy, change the video quality at the 2-second boundaries in the streaming session as the resource availability oscillates.

When a user watches a video, data is requested from a server using plain HTTP GET requests. The client that displays the video is responsible for requesting data, buffering video segments and adjusting the quality according to experienced throughput. Based on the number of buffered video segments, the requested quality is increased or decreased.

## 3.2 Multilink support

In order to allow the utilization of multiple access networks for streaming video to the client, we made several modifications to the original DAVVI platform [4], enhancing the system with an extended version of our multilink HTTP download and pipelining mechanisms [5, 6]. Support for using multiple interfaces was implemented to allow the client to initiate and utilize multiple access networks simultaneously. Establishing connections over specific interfaces is achieved by binding the communication socket to the desired interface, which is supported by most major operating systems. The only requirement is that the routing table of the client machine is configured properly. Packets must be routed through the correct interfaces, and the machine must be aware of the default interface and how to reach other machines in the connected networks.

### 3.2.1 Partial segments

As illustrated in Figure 1, video segments of a given quality are logically divided into partial segments. This is necessary in case a low latency is desired, to increase the granularity of the scheduler, and to allow the download of video segments over multiple interfaces concurrently. HTTP/1.1 supports *range retrieval requests* that enable the client to request a specific part of a file. Such techniques are frequently used in parallel access schemes, such as the one we proposed in [5]. For example, if the first 50 kB of a 100 kB large file are requested, only bytes $0 - 49\,999$ are sent by the server.

### 3.2.2 Request pipelining

Dividing a complete segment into partial segments introduces two challenges. First, smaller segments imply that more requests are made to the server. Provided that the server is fast enough to handle all the requests, this intro-

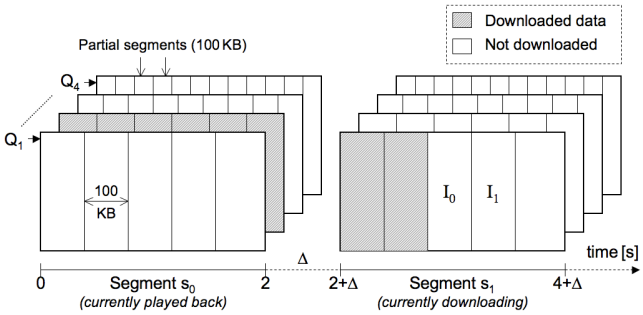


**Figure 1: In this example, two interfaces $I_0$ and $I_1$ have finished downloading segment $s_0$ of quality $Q_2$. As the throughput has dropped, they currently collaborate on downloading a lower-quality segment.**

duces additional delays caused by the client waiting for responses. This challenge is alleviated by using HTTP pipelining [6]. The scheduler tries to ensure that no interface becomes idle by initially requesting two partial segments. At runtime, the scheduler then always requests a new partial segment for each one received, ensuring that the server always has a request to process.

The second challenge relates to the size of the partial segments. If they are chosen too small, an entire segment might be sent from the server before the next data request is received, causing a drop in performance due to idle network interfaces. Based on earlier work [6], the partial segment size was kept constant at 100 kB throughout this paper. This size allows sufficient flexibility for the scheduler, and is large enough to take advantage of HTTP pipelining.

## 3.3 Request scheduler

In order to effectively utilize the available client interfaces, to avoid video deadline misses (a video segment is received after its desired playout time) and to provide the best possible user experience, a request scheduler was developed. The functionality of the scheduler is to estimate the aggregated throughput for choosing the desired video quality level, requesting segments over the available interfaces, and to manage the arrival and playback of video data. HTTP requests are treated in order by the server, so by registering which video segment is requested over which interface, the player is able to play back the video in order. The request scheduler is the most important part of our approach. Without a good scheduler, adding multiple interfaces can reduce performance and the quality of the user experience. For example, the quality adjustments might be too optimistic, causing a high number of deadline misses and playback interrupts.

We implemented and tested the request scheduler using the DAVVI player, based on the FIFO-like request approach presented in [5] and inspired by the "replica servers" scenario analyzed in [10]. Both these approaches perform well with stable and dynamic links for transferring bulk data. Our scheduler, outlined in Algorithm 1, is designed for adapting video quality according to bandwidth and conforming to the constraints imposed by live video streaming in DAVVI.

The first constraint imposed by live streaming is the number of complete video segments the server has made available. This decides how "live" the video stream will be. In DAVVI, each video segment contains two seconds of content, meaning that every segment adds two seconds of delay

**Algorithm 1** Request Scheduler *[simplified]*

```
quality_level = "low"
request(initial partial segment over each interface)
request("pipelined" partial segment over each interface)
while (more data available from server) do
  data = receive()
  I = interface that received data
  estimate aggregated throughput
  if (data == complete partial segment) then
    if (data == complete segment) then
      queue_for_playback(segment)
      adjust quality_level to throughput
    end if
    request(next partial segment, I, quality_level);
  end if
end while
```

compared to the live event. The number of segments the scheduler must wait for before starting to request data, is currently specified using a command line parameter.

The second constraint is imposed by the deadlines, which influence the quality adjustments. To achieve smooth playback, the next complete video segment has to be received before the current one is played, i.e., within two seconds. The scheduler tries to ensure this by adjusting the requested video quality according to the average throughput. Before a new segment is requested, the scheduler fetches the file size of the different quality levels using the HTTP HEAD command. The quality level of the requested segment is decided based on the number of bytes the client can receive in two seconds (calculated using the average throughput).

# 4. EXPERIMENTAL SETUP

For performing the experiments presented in this paper, a testbed was configured consisting of a client and a video server. Both machines run Linux 2.6.31. To control the bandwidth and latency heterogeneity over the two links, the network emulator *netem* was used with a hierarchical token bucket queuing discipline. In addition, for performance measurements of the scheduler in a real-world environment, the client was connected to public WLAN (IEEE 802.11b) and HSDPA networks. The characteristics of these two networks are summarized in Table 1.

**Table 1: Observed Characteristics of used Links**

| | WLAN | HSDPA |
|---|---|---|
| Maximum achievable throughput | 640 KB/s | 320 KB/s |
| Average experienced throughput | 600 KB/s | 250 KB/s |
| Average RTT for header-only IP packets | 20 ms | 100 ms |
| Average RTT for full-size IP packets | 30 ms | 220 ms |

Although the DAVVI video platform is designed to stream a large number of files, for the purpose of this work we focus on a single movie with the content of a soccer match. The movie has a total play length of about 100 minutes (3127 segments) and is available on the server in 4 different quality levels (see Table 2).

**Table 2: Quality levels of the soccer movie**

| Quality level | Low | Medium | High | Super |
|---|---|---|---|---|
| Minimum bitrate per segment (Kbit/s) | 212 | 431 | 941 | 1580 |
| Average bitrate per segment (Kbit/s) | 600 | 1150 | 2000 | 3000 |
| Maximum bitrate per segment (Kbit/s) | 1186 | 2458 | 4112 | 5949 |

# 5. RESULTS AND DISCUSSION

Both the quality of a video and the user experience are expected to increase when requests are scheduled over multiple interfaces. In this section, we analyze how bandwidth and delay heterogeneity affect performance, both in a completely controlled and a real-world wireless scenario.

## 5.1 Static links

The emulation testbed introduced in Section 4 provides a static, fully controllable network environment, where it is possible to isolate and adjust different parameters, such as the effect of various degrees of bandwidth and latency heterogeneity on the video quality. In addition, we measure the deadline misses, which are of highest importance from a user perspective, as shown in [8]. In general, users are willing to trade quality for smooth playback.

### *5.1.1 Bandwidth heterogeneity*

Figure 2 shows the effect of bandwidth heterogeneity on average video quality. 2 links whose combined bandwidth was kept at 3 Mbit/s were used in this experiment. A buffer size of 2 segments was used, meaning that the client lagged 4 s behind the live stream when the video started playing.

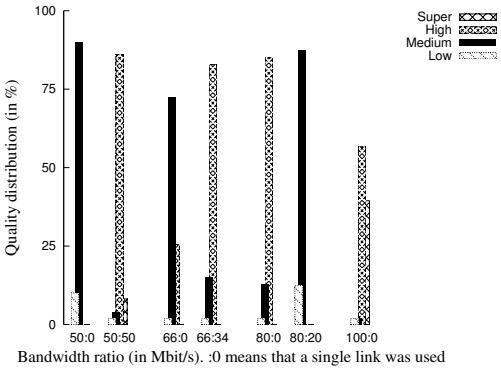


**Figure 2: Video quality distribution when the scheduler was faced with bandwidth heterogeneity in a fully controlled environment (0.1 ms RTT on both links).**

Using only one link shows an expected increase in quality that can be achieved when link bandwidth is increased from 1.5 Mbit/s (50:0) to 3 Mbit/s (100:0). It is less expected that the performance of a pair of links whose combined capacity is 100% of the total decreases quickly with growing bandwidth heterogeneity. At an 80:20 ratio, the achieved quality is even lower than in the 80:0 case.

This is caused by the *last segment problem* [6] that comes into effect when trying to stay too close to the live stream. Since no new segments are available on the server, the fast interface is idle once for every segment, while it waits for the download of the last partial segment by the slow interface.

The last segment problem can be mitigated by increasing the number of buffered segments and thus, startup delay, i.e., by trading liveness against video quality. The gain of increasing the number of buffered segments is shown in Figure 3. This figure shows how a longer startup delay increases the efficiency of throughput aggregation in the case of a high bandwidth ratio (80:20). For example, with a startup delay of 5 video segments, which equals 10 s, the average quality when using two links exceeds that achieved over a single link. This is expected, because the fast interface can receive four segments for every one received by the slow interface.

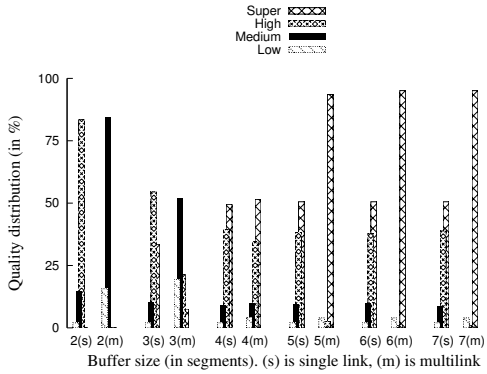


**Figure 3: The number of buffered segments plotted against video quality distribution (bandwidth ratio 80:20).**

Another interesting observation made in Figure 3 is that limiting the buffer size also affects the performance when a single link is used. Even though the connections to the server were kept alive during the whole stream, the cost of having to wait and then request a new segment is high.

Figure 4 shows the deadline misses for the different bandwidth ratios. The RTT was kept constant at 0.1 ms, and the deadline misses occur when the scheduler has to cope with bandwidth heterogeneity. This is another consequence of the last segment problem. The slow interface causes a delay of the complete video segment.

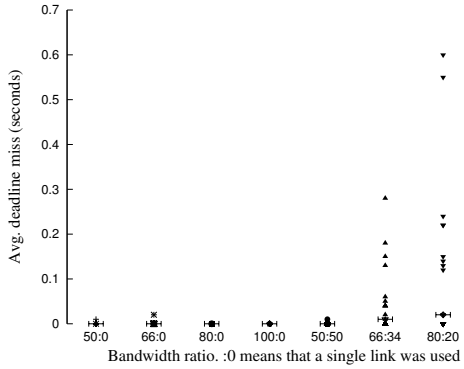


**Figure 4: Deadline misses for 2-segment buffers and various levels of bandwidth heterogeneity.**

### *5.1.2 Latency heterogeneity*

For the results presented in Figure 5, we varied the RTT heterogeneity. One link was kept constant at 10 ms, while the other link was assigned an RTT of $r$ ms, with $r \in \{10, 20, \ldots, 100\}$. The bandwidth of each link was limited to 1.5 Mbit/s, and the buffer size 2 segments.

As depicted in Figure 5, video quality is not significantly affected by RTT heterogeneity. Earlier experiments [6] have already shown that latency heterogeneity can be compensated by pipelining adequately small segments and introducing a startup latency.

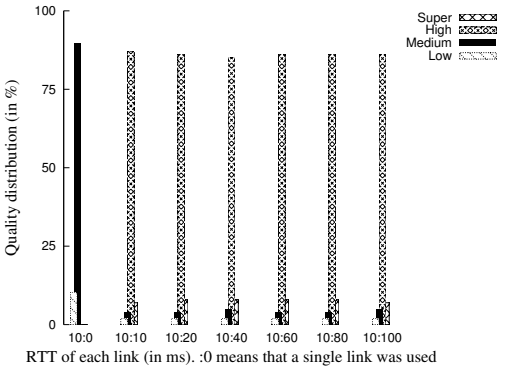


**Figure 5: Video quality distribution when the scheduler was faced with RTT heterogeneity in a fully controlled environment.**

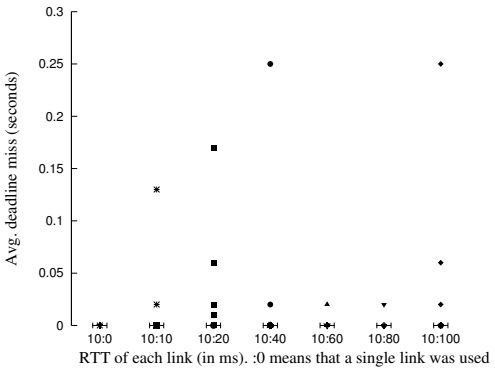


**Figure 6: Deadline misses when the scheduler is faced with RTT heterogeneity in an emulated environment.**

However, not taking the RTT into account can have an effect on deadline misses, as seen in Figure 6. The buffer size limits the frequency of segment requests, and because a complete segment is not requested before the previous is finished, it takes one RTT before the first bytes are received. Packet losses or an overestimation of bandwidth by the scheduler can lead to deadline misses. However, the observed lateness is not significant compared to the complete segment length. The average lateness for all bandwidth ratios is close to 0 s, and the maximum observed lateness is less than 0.3 s.

Buffering is one way of avoiding deadline misses. By adding more buffers and sacrificing the liveness, the client has enough stored data to compensate for deadline misses (similar tests performed with a buffer size of one and three segments confirm this). A buffer size of one caused a significant increase in the number of deadline misses, and they were more severe. When the buffer was increased to three segments, all deadline misses were eliminated.

## 5.2 Dynamic links

Dynamic links impose different challenges than static links. Their behaviour requires that the scheduler adapts to changes in the network, often rapidly. We test our scheduler in two different scenarios. In the first, we emulate dynamic behaviour to control all parameters and analyze how the scheduler performs. In the second, we test our scheduler using public WLAN and HSDPA networks to get an impression of its performance in the real world.

### 5.2.1 Emulated dynamics

To emulate dynamics, we created a script that emulates network behaviour observed in real world networks. The sum of the bandwidths is constant at 3 Mbit/s, but at a random interval of $t$ seconds, $t \in [2, 10]$, the bandwidth $bw$ Mbit/s, $bw \in [0.5, 2.5]$, is updated. The RTT of link 1 is normally distributed between 0 ms and 20 ms, the RTT of link 2 is uniformly distribution between 20 ms and 80 ms. The worst case heterogeneity is 5:1 (2.5 Mbit/s and 0.5 Mbit/s), meaning that a buffer size of six segments was used, according to the discussion in 5.1.1.

Figure 7 shows the average achieved throughput for every requested segment (over 40 runs) when single links were used, and when they were combined. When both links were used at the same time, the throughput exceeded the average bitrate-requirement for "High" quality most of the time. When single links were used, the achieved throughput stayed between "Medium" and "High". With single links, 35 % of the segments had "Medium" and 20 % "High" quality, 26 % "Super" and 18 % "Low". In combination, 95 % of the segments were in "Super" quality.

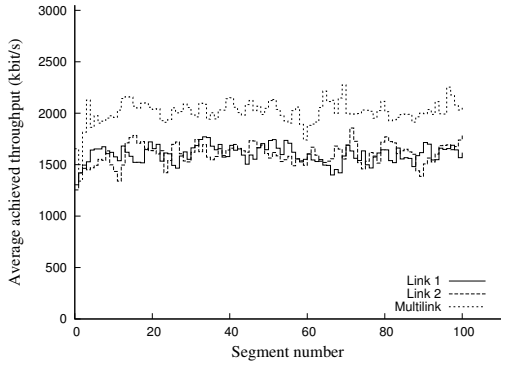


**Figure 7: Average achieved throughput of the scheduler with emulated dynamic network behaviour.**

Figure 8 shows that deadline misses occur although the buffer size is big enough to support the maximum heterogeneity. This is caused by the implementation of the player. Even though it waits for a given number of ready video segments, it starts playing video as soon as the first segment is completely received to maximize liveness. When the bandwidth fluctuates, the following segments may arrive too late, causing deadline misses.

One way to solve this would be to wait until the buffer is completely filled before the player starts playing back the video, reducing liveness even further. Another way is to create a more adaptive scheduler. The scheduler could monitor the links and adjust the quality accordingly and on-the-fly. In addition, provided that enough bandwidth is available, the scheduler could adjust the quality accordingly and ensure that there is always buffered data ready.

### 5.2.2 Real-world networks

To get an impression of the scheduler's performance over real wireless networks, we experimented with the DAVVI player using WLAN and HSDPA interfaces with character-

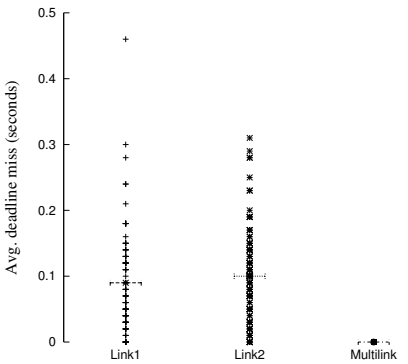


**Figure 8: Deadline misses of the scheduler with emulated dynamics.**

istics as summarized in Table 1. The observed worst case heterogeneity was around 5:1, so a buffer size of six was used.

Figure 9 shows the average achieved throughput (also over 40 runs) for every requested segment. The scheduler improves the performance and thereby the video quality significantly. With the fastest of the two interfaces, WLAN, 45 % of the segments were in "Super" quality, compared to 91 % when both links were used. The worst observed deadline miss over both links was only 0.3 s.

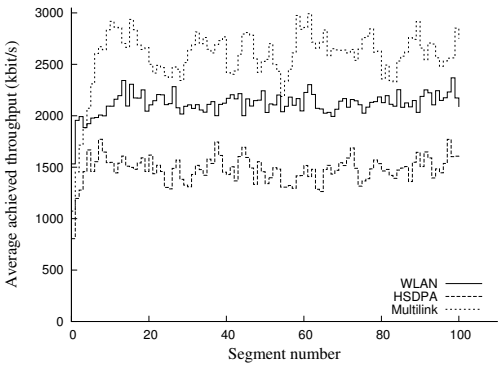


**Figure 9: Average achieved throughput of the scheduler in real-world wireless networks.**

# 6. CONCLUSION

In this paper, we introduced and analyzed a pull-based scheduler for streaming real-time video over an aggregate of heterogeneous network interfaces. Using the file segmentation approach in DAVVI, which is also used in popular commercial systems [1,7,13], the proposed scheduler adapts the video quality according to the combined throughput of all the available interfaces at the client.

Experiments conducted in a controlled emulation testbed showed that the scheduler achieves close to perfect throughput aggregation for high degrees of latency heterogeneity. For links with heterogeneous bandwidths, there exists a complex tradeoff between their bandwidth disparity, the achievable video liveness, and the efficiency of throughput aggregation. For two stable interfaces of equal bandwidth, the scheduler achieves close to perfect bandwidth aggregation, while maintaining a quasi-live video stream, i.e., never more than 4 seconds late (after a segment is available at the web server, thus excluding coding and server uploading delay). If the bandwidth heterogeneity is higher, which is the case for a combination of WLAN and HSDPA, about 10 seconds of delay have to be accepted to obtain the full potential of aggregated throughput and best possible video quality.

In its current form, the request scheduler requires all network interfaces to be properly configured before the video player can utilize them simultaneously. In a future version, we will implement a more dynamic and robust scheme that automatically updates the routing table when the connectivity of the interfaces changes.

# 7. REFERENCES


[1] Apple Inc. Mac OS X Server – QuickTime Streaming and Broadcasting Administration, 2007.

[2] Biersack, E., and Geyer, W. Synchronized delivery and playout of distributed stored multimedia streams. *Multimedia Syst. J. 7*, 1 (1999), 70–90.

[3] Griwodz, C. *Wide-area True Video-on-Demand by a Decentralized Cache-based Distribution Infrastructure.* PhD thesis, Darmstadt University of Technology, 2000.

[4] Johansen, D., Johansen, H., Aarflot, T., Hurley, J., Kvalnes, A., Gurrin, C., Zav, S., Olstad, B., Aaberg, E., Endestad, T., Riiser, H., Griwodz, C., and Halvorsen, P. DAVVI: A prototype for the next generation multimedia entertainment platform. In *Proc. ACM MM* (2009), pp. 989–990.

[5] Kaspar, D., Evensen, K., Engelstad, P., Hansen, A., Halvorsen, P., and Griwodz, C. Enhancing video-on-demand playout over multiple heterogeneous access networks. In *Proc. IEEE CCNC* (2010).

[6] Kaspar, D., Evensen, K., Engelstad, P., and Hansen, A. F. Using HTTP pipelining to improve progressive download over multiple heterogeneous interfaces. In *Proc. IEEE ICC* (2010).

[7] Move Networks. Internet television: Challenges and opportunities. Tech. rep., Move Networks, Inc., November 2008.

[8] Ni, P., Eichhorn, A., Griwodz, C., and Halvorsen, P. Fine-grained scalable streaming from coarse-grained videos. In *Proc. ACM NOSSDAV* (2009), pp. 103–108.

[9] Nielsen, H. F., Gettys, J., Baird-Smith, A., Prud'hommeaux, E., Lie, H. W., and Lilley, C. Network performance effects of HTTP/1.1, CSS1, and PNG. In *Proc. ACM SIGCOMM* (1997), pp. 155–166.

[10] Rodriguez, P., and Biersack, E. W. Dynamic parallel access to replicated content in the internet. *IEEE/ACM Trans. Netw. 10*, 4 (2002), 455–465.

[11] Wang, B., Wei, W., Guo, Z., and Towsley, D. Multipath live streaming via TCP: Scheme, performance and benefits. *ACM Trans. Multimedia Comput. Commun. Appl. 5*, 3 (2009), 1–23.

[12] Wu, F., Gao, G., and Liu, Y. Glitch-Free Media Streaming. Patent Application (US2008/0022005), January 24 2008.

[13] Zambelli, A. IIS Smooth Streaming technical overview. Tech. rep., Microsoft Corporation, 2009.