Improving Internet Video Streaming Performance by Parallel TCP-based Request-Response Streams

Robert Kuschnig, Ingo Kofler, Hermann Hellwagner
Institute of Information Technology (ITEC), Klagenfurt University, Austria
Email: firstname.lastname@itec.uni-klu.ac.at

***Abstract*—TCP-based video streaming encounters difficulties in unreliable networks with unanticipated packet loss. In combination with high round trip times, the effective throughput deteriorates rapidly and TCP connection resets or stalls may occur. In this paper, we propose a client-driven video transmission scheme which utilizes multiple HTTP/TCP streams. The scheme is largely insensitive to unanticipated packet loss and thereby reduces throughput fluctuations. Since it is based on HTTP, the scheme can easily be deployed in existing network infrastructures. It fosters scalability on the server side by shifting complexity from the server to the clients. Certain features of request-response schemes allow maintaining fairness, despite of using multiple HTTP streams. Making use of TCP, the scheme inherently adapts to congested network links.**

***Index Terms*—video streaming, TCP, HTTP, client-driven, fairness, TCP-friendliness**

I. Introduction

Video streaming based on TCP has become popular because of its easy handling and deployment. TCP features in-order delivery and reliable end-to-end transport, which makes additional tools, like error concealment, unnecessary. In low-latency networks, TCP features good throughput performance and low end-to-end delays, which even makes TCP-based interactive services possible. The inherent TCP-behavior assures that only a fair share of the network bandwidth is consumed by a video stream. In congested networks, though, throughput variations may occur, which can disrupt the video consumption. Furthermore, TCP streaming encounters difficulties in unreliable networks with unanticipated packet loss, because TCP streaming inherits all basic problems of TCP, which are mostly related to packet loss. The effects are highly variable throughput, throughput limitations, and TCP connection timeouts or stalls.

The typical deployment scenario for TCP video streaming is the Internet. Video portals like YouTube are streaming the content throughout the whole world. The delivery of YouTube is almost centralized [7], which leads to round trip times (RTT) of up to 200 ms and more. This makes retransmissions, which may occur due to congestion or packet corruption, challenging. In our use case (see Figure 1), we focus on asymmetric access networks which are assumed to form the bottleneck links. Random packet losses, which may be introduced by changing network conditions, are limited due to link layer retransmissions in the access network. However, these retransmission may result in increased delay and jitter values.

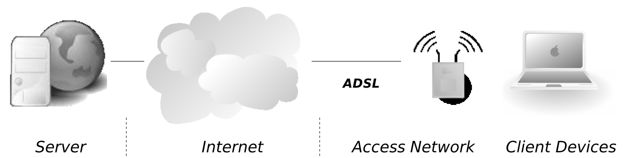

Fig. 1. Use case

In our work we review this inherent problem of TCP regarding packet loss and propose a new client-driven transport mechanism based on HTTP [3]. By utilizing multiple HTTP streams, we are able to reduce throughput fluctuations in error-prone networks. We can maintain the TCP-friendliness of our approach, although using multiple HTTP streams. The usage of HTTP introduces some overhead, but enables advantages in terms of deployment and re-use of existing HTTP software. The proposed transmission scheme avoids complexity, by placing the streaming logic only at the client. To support the in-time delivery of the video data, the system assigns priorities to certain parts of the media stream.

In the next section the characteristics of TCP-based streaming under packet loss are investigated. TCP-based request-response schemes are analyzed in Section III. Section IV presents our client-driven approach, which shows how request-response schemes may be used for video streaming. In Section V an evaluation w.r.t. robustness and TCP-friendliness of our approach will be given. Section VI concludes the paper.

II. Limits of TCP-based Streaming

The usage of TCP in video streaming is popular due to its reliability, adaptability to fluctuating network conditions and easy deployment. Because reliable transmission in TCP is based on acknowledgments and retransmissions, the throughput rate r of TCP is limited by the maximum segment size (MSS) and the round trip time (RTT). Assuming that after the successful transmission of $1/p$ packets one packet is lost, the *upper bound for the TCP throughput* r_{max} is [6]:

$$r = \frac{data\ per\ cycle}{time\ per\ cycle} < \left(\frac{MSS}{RTT}\right)\frac{1}{\sqrt{p}} = r_{max} \qquad (1)$$

According to Equation 1, it is obvious that the maximum throughput of a single TCP connection is limited by the packet loss p for a given RTT. An additional characteristic of TCP is induced by the additive-increase/multiplicative-decrease (AIMD) algorithm used for congestion control. The

MD step reduces the TCP window (and therefore the throughput) in case of packet loss drastically, before the AI step tries to increase the window once again. This leads to a notable variation of the throughput in networks with large RTTs. The streaming performance of TCP under different network conditions was evaluated in [1]. It was shown that TCP streaming offers good performance when the achievable TCP throughput is twice the media bit rate. This over-provisioning is reasonable for Internet streaming at low bit rates, but for emerging services, like high definition video streaming, it may not be feasible to supply twice the bit rate. Although the bandwidth of last-mile links has increased (a significant number of hosts have downlink bandwidths greater than 4 Mbps [4]), the high bit rates needed for high definition video prevent the use of this kind of over-provisioning.

Significant research on enhancing TCP performance was done. New TCP implementations like CUBIC [9] address the TCP performance problem on network links with high bandwidth-delay-products. These implementations achieve better link utilization, but tend to be potentially unfair to TCP Reno [2]. In addition they all share a similar performance problem with unanticipated packet loss, which is based on the MD step in the congestion control. There have also been investigations concerning the use of multiple TCP connections to stabilize TCP throughput [8], [10]. In general TCP connections share the available bandwidth in a fair manner. Aggregating multiple TCP streams for a single use is potentially unfair to concurrent single connections. While being able to stabilize and enhance the throughput, the solution in [10] exhibits this unfairness. The approach described in [8] tries to provide TCP-friendliness to concurrent single TCP connections, by adjusting the rate of each TCP stream such that the aggregated rate corresponds to the rate of a concurrent single TCP connection. The described approaches using multiple TCP streams try to enhance the TCP streaming performance, but at the cost of high deployment effort. Our work avoids this complexity, by placing the streaming logic only at the client. In short, no new TCP implementation is needed and no additional feedback loop between the server and the client. Additionally, the described approaches need a rather complex feedback loop between the client and the server, which has to coordinate the transmission in case of connection aborts. Our client-driven approach responds faster to changing network conditions and enables easy recovery from connection stalls or aborts, because the control loop is at the client.

Another way to transport data via TCP is the use of short lived TCP connections, like in Web browsing. While introducing additional overhead in comparison to a continuous TCP connection, such request-response schemes may also exhibit favorable characteristics, which will be discussed in the following section.

III. TCP-based Request-Response Schemes

TCP-based request-response protocols (e.g., like HTTP) inherit all basic features of TCP. In general, the client requests data from the server. After receiving the request, the server creates a response and sends the data to the client. For large blocks of data (or infinite-source data), the response behaves like a classical TCP connection, but in case of smaller responses a different behavior can be observed. Because small responses are short-lived, they may experience unfairness from infinite-source TCP connections [5]. In addition, TCP features no throughput fairness between connections with different RTTs [5].

Considering these two facts, we can create a system to steer TCP-friendliness in request-response schemes. By inserting temporal gaps between the requests (inter-request gap t_{gap}), we can emulate a TCP connection with increased RTT. As a result, we can aggregate multiple submissive request-response streams, which exhibit the same TCP-friendliness as a single TCP connection.

In the following, we introduce a simple model which allows us to define an upper bound for the throughput of the request-response (rr) streams. We assume that the data (of size $l_d \approx n_d * MSS$) is transferred within a single RTT. In addition, n_c concurrent, but not-interfering request-response streams are used for the transmission of the data. The upper bound for the throughput without packet loss r_{rr} in absence of congestion can now be defined as:

$$\frac{data\ per\ cycle}{time\ per\ cycle} < n_c \left(\frac{l_d}{RTT + t_{gap}} \right) = r_{rr} \tag{2}$$

By using n_c request-response streams and n_d packets per data block, a total of $n_p = n_c * n_d$ packets are transmitted. Assuming the same packet loss as described before, we can define the *maximum number of lost packets* n_l as:

$$n_l = \left\lceil \frac{n_p}{1 + 1/p} \right\rceil \tag{3}$$

Using the simplified assumption that the packet loss is equally distributed over all request-response streams and each connection does not experience more than one lost packet per transported data block ($n_l \leq n_c$), the *upper bound for the throughput under packet loss* r_{rrmax} of the request-response streams would be:

$$r_{rrmax} = (n_c - n_l) \left(\frac{l_d}{RTT + t_{gap}} \right) + n_l \left(\frac{l_d}{2 * RTT + t_{gap}} \right) \tag{4}$$

The request-response streams with a single packet loss have to retransmit the lost packet and need in the best case an additional RTT for the transmission of the data. Equation 4 shows that even under packet loss we are able to tune the throughput with the help of n_c and l_d. In addition, t_{gap} can be used to adjust the TCP-friendliness, but has obviously a negative impact on the throughput. Thus, a trade-off has to be found between these parameters. The idea is to be more error resilient than a single TCP connection, while stabilizing and enhancing the overall throughput, but also to be fair to other connections in case of congestion.

This simplification is usable for the estimation of the achievable bandwidth for small n_c, and to explore which parameters do influence the throughput of the request-response streams.

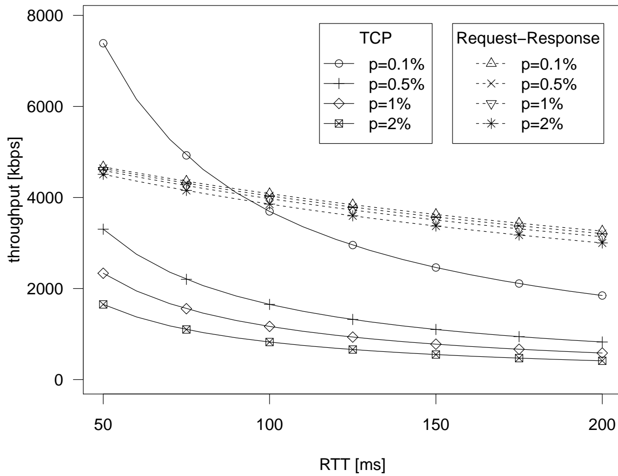

Fig. 2. Theoretical upper bounds for throughput of a single TCP connection (TCP) r_{max} and request-response streams r_{rrmax} ($MSS = 1460\ bytes$, $l_d = 20480\ bytes$, $n_c = 10$, $t_{gap} = 350\ ms$) under packet loss p.

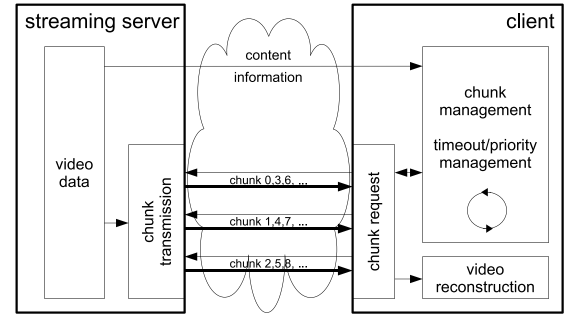

Fig. 3. Request-response-based client-driven streaming system

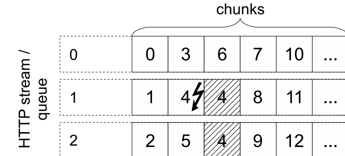

Fig. 4. Three HTTP streams/queues on the client with priority management

For larger n_c, the concurrency may introduce additional queuing delay in the routers and scheduling effort on the server and the client, which may result in much smaller throughput values. In Figure 2 the upper bounds for a single TCP connection and the request-response streams (RR) according to Equations 1 and 4, respectively, are shown. It can be seen, that the performance of the single TCP connection highly depends on the packet loss and the RTT. The variation of the throughput for RR w.r.t. RTT and packet loss is significantly reduced. The next section presents a streaming system based on these findings. An evaluation of this system in Section V validates our theoretical assumptions on throughput stability and TCP-friendliness.

IV. Request-Response-based Client-driven Streaming

We propose a new transmission scheme for video data which is beneficial in case of bad network conditions. The use of multiple request-response streams allows us to reduce the quality fluctuations by stabilizing the transmission rate. Therefore the scheme needs less over-provisioning than a single TCP connection. As will be shown in Section V, a streaming system based on request-response streams is able to provide TCP-friendliness, although using multiple HTTP streams. The basic architecture of our *request-response-based client-driven streaming system* is shown in Figure 3. The system is based on HTTP such that it allows easy deployment, enables reuse of existing infrastructure (HTTP server, client, encryption, etc.) and enables application-layer multicast structures through HTTP proxies. Persistent connections as defined in HTTP/1.1 [3] are used to minimize the effort for the establishment of TCP connections.

Before streaming, the video is split into *chunks* of size l_d and n_c HTTP streams are used to fetch them. Therefore, for each HTTP stream, a queue with video chunks is created on the client (see Figure 4). The number of parallel HTTP requests n_c and the chunk size l_d are fixed at start-up time, based on available bandwidth. A trade-off between parallel HTTP requests, chunk size and TCP-friendliness (t_{gap}) (see Equation 4) has to be found and is addressed in Section V-A. After the initial setup, the chunks are retrieved via HTTP according to their order within the queue. The streaming client coordinates the in-order transmission of the chunks and attempts to maximize the in-order throughput. Our approach uses a simple strategy for fetching the chunks, which calculates the timeout for establishing the transfer timeout for the chunks (in the following denoted as timeout) dynamically. In addition to the *timeout management*, *priority management* is applied to the requests, which handles the transmission of late video chunks. In the following, the two basic mechanisms of our streaming system are described in detail.

A. Timeout Management

Because we only investigate small chunk sizes, we can consider the transfer of a chunk to be stalled if the chunk is not retrieved within 1000 ms to 3000 ms. This estimation is based on the assumption that the RTT is 200 ms at maximum, which means that a chunk has to be retrieved at least after 15 “attempts” ($15 * 200\ ms \approx 3000\ ms$). The lower bound of the timeout is set to 1000 ms, to enable retransmissions on congested links with little packet loss and delay. The timeout is initialized as 2000 ms. The *transfer duration* of each chunk is monitored at the client. The timeout is calculated from a moving average over the last 20 transfer durations plus a tolerance of 30%. Also expired transfers are considered (with a penalty) in the moving average, in order to supply the timeout management with early feedback on stalled transfers.

B. Priority Management

Timeliness is most important for video streaming because late video frames reduce the quality of experience. Therefore

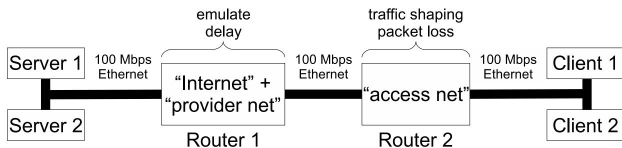

Fig. 5. Test setup

priority management is applied to the multiple request-response streams in order to prioritize the transfer of chunks required in the near future. The video chunks are numbered in order of appearance within the video stream. As depicted in Figure 4, each HTTP stream fetches the chunks in the same order as located in the queue, i.e., in priority order. If the transfer of a chunk stalls and reaches a timeout (as chunk 4 in Figure 4), it will be fetched by two HTTP streams again, to increase the probability of successful transmission. Although sounding greedy, queuing a chunk twice does not change the number of used HTTP streams. As a result it also does not change the TCP-friendliness. The back-off in case of congestion is done implicitly by the HTTP streams, since they are based on TCP.

The proposed transmission scheme uses HTTP and features low complexity on the server (basic HTTP service), resulting in high scalability at the cost of increased client complexity. It stabilizes the in-order throughput by utilizing multiple HTTP streams. In addition, the system assigns priorities to certain parts of the video, which supports the in-time delivery of the video data.

V. Evaluation

Our evaluation on *request-response-based client-driven streaming* is divided into two parts. First, the robustness w.r.t. packet loss and delay is investigated in Section V-A. The metric for comparing TCP Reno with our client-driven approach is the average throughput. Second, an evaluation concerning the TCP-friendliness of the client-driven streaming will be given in Section V-B.

The test setup consists of two servers, two routers and two clients each running Ubuntu Linux, kernel 2.6.27, as illustrated in Figure 5. The routers are using Netem[1] for network emulation. Router1 emulates the symmetric end-to-end delay of the Internet and the provider network, with a normal distributed delay with 10% standard deviation, which is applied to all packets. The access network to the clients is emulated by Router2, which limits the up- and downstream bandwidth (BW), while allowing a maximum queuing delay of 400 ms. In addition, the packets are dropped in a random fashion to emulate packet loss. On all computers, the TCP implementation Reno is used. The Apache[2] HTTP Server is employed for serving the video chunks. Our prototype software uses Python and the HTTP library libcurl[3].

[1] http://www.linuxfoundation.org/en/Net:Netem
[2] http://httpd.apache.org
[3] http://curl.haxx.se

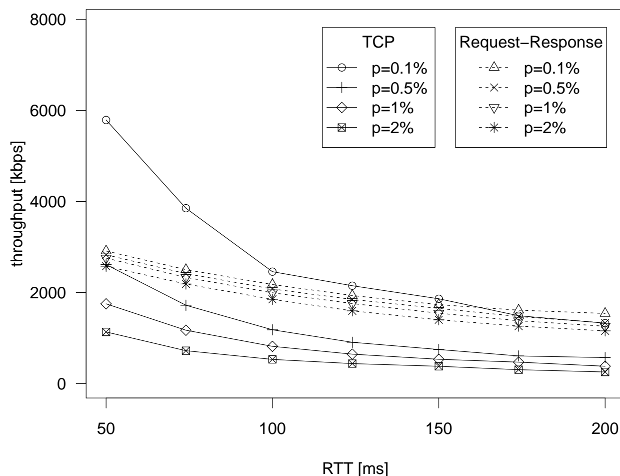

Fig. 6. Measured throughput of a single TCP connection (TCP) and request-response streams ($BW = 8192\ kbps$, $MSS = 1460\ bytes$, $l_d = 20480\ bytes$, $n_c = 10$, $t_{gap} = 350\ ms$) under packet loss p.

For our investigation of transmission overhead, we analyzed the packets sent for an HTTP request and a single TCP transfer. The packet trace was created using wireshark[4]. Using a standard HTTP client (libcurl) and server (Apache), the HTTP request without the URL is approx. 200 bytes long and the HTTP response approx. 300 bytes. Apart from establishing the TCP connection there is an absolute overhead of approx. 600 bytes. The relative overhead for chunk sizes of 10240, 20480 and 40960 bytes is 5.8%, 2.9% and 1.5%, respectively. To keep the overhead of establishing the TCP connections minimal, persistent connections as defined in HTTP/1.1 are used.

A. Streaming Performance

In this section, the performance of request-response-based client-driven streaming and TCP streaming using a single TCP connection is evaluated under different network conditions, to show the robustness with respect to packet loss and network delay. In addition, the results should clarify if the models for the upper bound of the throughput as presented in Sections II and III are valid.

For the evaluation Server1 and Client1 are used. The network bandwidth (BW) was set to 8192 kbps, which should not limit the maximum throughput. The test video streams are emulated by an infinite data source, to assure the correct measurement of the maximum achievable throughput. Each test run lasts for 500 seconds and every 10 seconds the in-order throughput is measured. In Figure 6 the average throughput for a single TCP connection and the request-response streams are shown. In networks with small RTTs, the request-response and the single TCP approach are able to cope with packet loss via retransmissions. With increasing delay and packet loss, the single TCP approach deteriorates rapidly, while the impact on the request-response system is limited. The direct comparison with the model (see Figure 2) reveals a good correlation to the measured results. Although lower, the values for the maximum

[4] http://www.wireshark.org

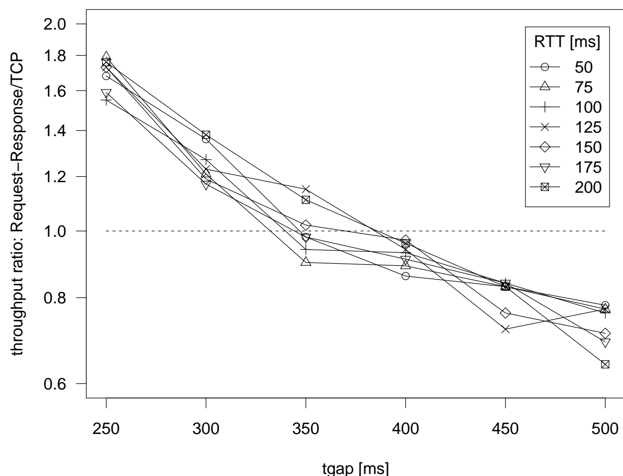

Fig. 7. TCP-friendliness of request-response streams ($BW = 2048\ kbps$, $MSS = 1460\ bytes$, $l_d = 20480\ bytes$, $n_c = 10$).

throughput are shaped equally and behave according to our assumptions. In our model, the upper bound for request-response systems is defined in a very optimistic manner, because the model assumes that the whole data block is transmitted within a single RTT and no server processing is considered. In reality, multiple RTTs may be needed, depending on the used configuration ($l_d = 20480\ bytes$, $n_c = 10$) and the network conditions. The handling of the request on the server also increases the effective RTT. The priority management of our streaming system may also decrease the achieved in-order throughput, but enhances the stability of the throughput.

The results show that our approach has a notable advantage over the single TCP approach in networks with unanticipated packet loss. It can stabilize the in-order throughput regarding RTT and packet loss. In the next section, we will show how to enable TCP-friendliness for our request-response-based client-driven streaming system.

B. TCP Friendliness

For the investigation of TCP-friendliness of our approach and the impact of t_{gap} on it, we use the *throughput ratio* metric [2] as an indicator of how the transmission system behaves towards a concurrent TCP Reno connection. A throughput ratio greater than one indicates that the request-response streams are unfair to the concurrent TCP Reno connection, while values lower than one indicate a submissive behavior. The system is regarded to be TCP-friendly if it allows a concurrent TCP Reno stream to gain its fair bandwidth share ($ratio = 1$), while the two streams compete for their bandwidth share on a congested network link.

The test setup was configured in such a manner that congestion on the access network link would occur. In parallel to the client-driven system (running on Server1 and Client1), an HTTP download from Server2 to Client2 is started and competes for its network share. Again, each test run lasts for 500 seconds and every 10 seconds the in-order throughput is measured. The throughput of the request-response-based client-driven streaming system is about 2048 kbps, assuming $l_d = 20480\ bytes$, $n_c = 10$ and $t_{gap} = 350\ ms$ (see Figure 6). So the network bandwidth BW is reduced to 2048 $kbps$ and the packet loss set to 0 %, to assure congestion on the network link. After this, the throughput ratio is measured for different RTTs, while t_{gap} is varied around 350 ms.

Figure 7 shows the throughput ratio for the different RTTs. It clearly points out that, with the help of temporal gaps (t_{gap}) between the requests, the TCP-friendliness of our approach can be steered. Around $t_{gap} = 350\ ms$ the throughput ratio is ≈ 1, which means that for the configuration $l_d = 20480\ bytes$ and $n_c = 10$, a temporal gap of $t_{gap} = 350\ ms$ would lead to a TCP-friendly request-response stream.

VI. CONCLUSION

We introduced a client-driven streaming system based on the request-response scheme of HTTP. It features good performance, even on network links with high delay and jitter, because it is not as prone to network errors and bandwidth fluctuations as a single TCP connection. Our client-driven approach uses multiple HTTP streams to mitigate throughput variations, but still adapts to congested network links and does not lead to starvation of a concurrent TCP connection. The TCP-friendliness of our approach is steerable by temporal gaps between the single requests. The achievable throughput of the system is influenced by the selection of the parameter for the temporal gap. Therefore a careful selection of this parameter is required. Future work will cover a detailed investigation of the selection of this parameter, considering a given chunk size and the number of concurrent HTTP streams.

VII. ACKNOWLEDGMENT

This work was supported by the Austrian Science Fund (FWF) under project “Adaptive Streaming of Secure Scalable Wavelet-based Video (P19159)” and by the EC in the context of the P2P-Next project (FP7-ICT-216217).

REFERENCES

[1] B. Wang, J. Kurose, P. Shenoy and D. F. Towsley. Multimedia streaming via TCP: An analytic performance study. *ACM TOMCCAP*, 4(2), 2008.
[2] D. Miras, M. Bateman and S. Bhatti. Fairness of High-Speed TCP Stacks. In *Proc. AINA*, 2008.
[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999.
[4] M. Dischinger, A. Haeberlen, K. P. Gummadi and S. Saroiu. Characterizing Residential Broadband Networks. In *Proc. IMC*, 2007.
[5] M. Hassan and R. Jain, editor. *High Performance TCP/IP Networking: Concepts, Issues, and Solutions*. Pearson Prentice Hall, 2004.
[6] M. Mathis, J. Semke, J. Mahdavi. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3), 1997.
[7] M. Saxena, U. Sharan, S. Fahmy. Analyzing Video Services in Web 2.0: A Global Perspective. In *Proc. NOSSDAV*, May 2008.
[8] R. P. Karrer, J. Park, J. Kim. TCP-ROME: Performance and fairness in parallel downloads for web and real time multimedia streaming applications. Technical report, Deutsche Telekom Labs, September 2006.
[9] S. Ha, I. Rhee, L. Xu. CUBIC: a new TCP-friendly high-speed TCP variant. *ACM SIGOPS Operating Systems Review*, 42(5), 2008.
[10] S. Tullimas, T. Nguyen, R. Edgecomb, S. Cheung. Multimedia streaming using multiple TCP connections. *ACM TOMCCAP*, 4(2), 2008.