Transparent Protocol Translation for Streaming

Håvard Espeland[1], Carl Henrik Lunde[1], Håkon Kvale Stensland[1,2], Carsten Griwodz[1,2], Pål Halvorsen[1,2]
[1]IFI, University of Oslo, Norway [2]Simula Research Laboratory, Norway
{haavares, chlunde, haakonks, griff, paalh}@ifi.uio.no

ABSTRACT

The transport of streaming media data over TCP is hindered by TCP's probing behavior that results in the rapid reduction and slow recovery of the packet rates. On the other side, UDP has been criticized for being unfair against TCP connections, and it is therefore often blocked out in the access networks. In this paper, we try to benefit from a combined approach using a proxy that transparently performs transport protocol translation. We translate HTTP requests by the client transparently into RTSP requests, and translate the corresponding RTP/UDP/AVP stream into the corresponding HTTP response. This enables the server to use UDP on the server side and TCP on the client side. This is beneficial for the server side that scales to a higher load when it doesn't have to deal with TCP. On the client side, streaming over TCP has the advantage that connections can be established from the client side, and data streams are passed through firewalls. Preliminary tests demonstrate that our protocol translation delivers a smoother stream compared to HTTP-streaming where the TCP bandwidth oscillates heavily.

Categories and Subject Descriptors

D.4.4 [**OPERATING SYSTEMS**]: Communications Management—*Network communication*

General Terms

Measurement, Performance

1. INTRODUCTION

Streaming services are today almost everywhere available. Major newspapers and TV stations make on-demand and live audio/video (A/V) content available, video-on-demand services are becoming common and even personal media are frequently streamed using services like pod-casting or uploading to streaming sites such as YouTube.

The discussion about the best protocols for streaming has been going on for years. Initially, streaming services on the Internet used UDP for data transfer because multimedia applications often have demands for bandwidth, reliability and jitter than could not be offered by TCP. Today, this approach is impeded with filters in Internet service providers (ISPs), by firewalls in access networks and on end-systems. ISPs reject UDP because it is not fair against TCP traffic, many firewalls reject UDP because it is connectionless and requires too much processing power and memory to ensure security. It is therefore fairly common to use HTTP-streaming, which delivers streaming media over TCP. The disadvantage is that the end-user can experience playback hiccups and quality reductions because of the probing behavior of TCP, leading to oscillating throughput and slow recovery of the packet rate. A sender that uses UDP would, in contrast to this, be able to maintain a desired constant sending rate. Servers are also expected to scale more easily when sending smooth UDP streams and avoid dealing with TCP-related processing.

To explore the benefits of both TCP and UDP, we experiment with a proxy that performs a transparent protocol translation. This is similar to the use of proxy caching that ISPs employ to reduce their bandwidth, and we do in fact aim at a combined solution. There are, however, too many different sources for adaptive streaming media that end-users can retrieve data from to apply proxy caching for all of them. Instead, we aim at live protocol translation in a TCP-friendly manner that achieves a high perceived quality to end-users. Our prototype proxy is implemented on an Intel IXP2400 network processor and enables the server to use UDP at the server side and TCP at the client side.

We have earlier shown the benefits of combining the use of TFRC in the backbone with the use of TCP in access networks [1]. In the experiments presented in that paper, we used course-grained scalable video (scalable MPEG (SPEG) [4]) which makes it possible to adapt to variations in the packet rate. To follow up on this idea, we describe in this paper our IXP2400 implementation of a dynamic transport protocol translator. Preliminary tests comparing HTTP video streaming from a web-server and RTSP/RTP-streaming from the komssys video server show that, in case of some loss, our solution using a UDP server and a proxy later translating to TCP delivers a smoother stream at play out rate while the TCP stream oscillates heavily.

2. RELATED WORK

Proxy servers have been used for improved delivery of

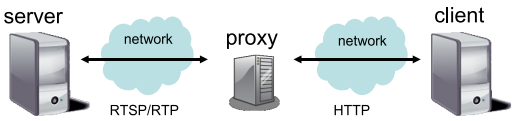


Figure 1: System overview

streaming media in numerous earlier works. Their tasks include caching, multicast, filtering, transcoding, traffic shaping and prioritizing. In this paper, we want to draw attention to issues that occur when a proxy is used to translate transport protocols in such a way that TCP-friendly transports mechanisms can be used in backbone networks and TCP can be used in access networks to deliver streaming video through firewalls. Krasic et al. argue that the most natural choice for TCP-friendly traffic is using TCP itself [3]. While we agree in principle, their priority progress streaming approach requires a large amount of buffering to hide TCP throughput variations. In particular, this smoothing buffer is required to hide the rate-halving and recovery time in TCP's normal approach of probing for bandwidth which grows proportionally with the round-trip time. To avoid this large buffering requirement at the proxy, we would prefer an approach that maintains a more stable packet rate at the original sender. The survey of [7] shows that TFRC is a reasonably good representative of the TCP-friendly mechanisms for unicast communication. Therefore, we have chosen this mechanism for the following investigation.

With respect to the protocol translation that we describe here, we do not know of much existing work, but the idea is similar to the multicast-to-unicast translation [6]. We have also seen voice-over-IP proxies translating between UDP and TCP. In these examples, a packet is translated from one type to another to match the various parts of the system, and we here look at how such an operation performs in the media streaming scenario.

3. TRANSLATING PROXY

An overview of our protocol translating proxy is shown in figure 1. The client and server communicates by the proxy, which transparently translates between HTTP and RTSP/RTP. Both peers are unaware of each other.

The steps and phases of a streaming session follows. The client tries to set up a HTTP streaming session, by initiating a TCP connection to the server. All packets are intercepted by the proxy, and modified before passing it on to the streaming server. The proxy also forwards the TCP 3-way handshake between client and server, updating the packet with the server's port. When established, the proxy splits the TCP connection into two separate connections that allow for individual updating of sequence numbers. The client sends a GET request for a video file. The proxy translates this into a SETUP request and sends it to the streaming server using the TCP port of the client as its proposed RTP/UDP port. If the setup is unsuccessful, the proxy will inform the client and close the connections. Otherwise, the server's response contains the confirmed RTP and RTCP ports assigned to a streaming session. The proxy sends a response with an unknown content length to the client and issues a PLAY command to the server. When received, the server starts streaming the video file using RTP/UDP. The UDP packets are translated by the proxy as part of the

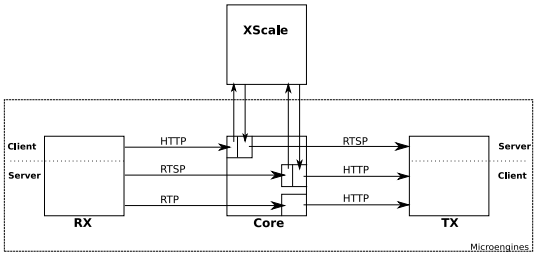


Figure 2: Packet flow on the IXP2400

HTTP response, using the source port and address matching the HTTP connection. Because the RTP and UDP headers combined are longer than a standard TCP header, the proxy can avoid the penalty of moving the video data in memory, thus permitting reuse of the same packet by padding the TCP options field with NOPs. When the connection is closed by the client during or after playback, the proxy issues a TEARDOWN request to the server to avoid flooding the network with excess RTP packets.

4. IMPLEMENTATION

Our prototype is implemented on a programmable network processor using the IXP2400 chipset [5], which is designed to handle a wide range of access, edge and core applications. The basic features include a 600 MHz XScale core running Linux, eight 600 MHz special packet processors called micro-engines (μEngines), several types of memory and different controllers and busses. With respect to the different CPUs, the XScale is typically used for the control plane (slow path) while μEngines perform general packet processing in the data plane (fast path).

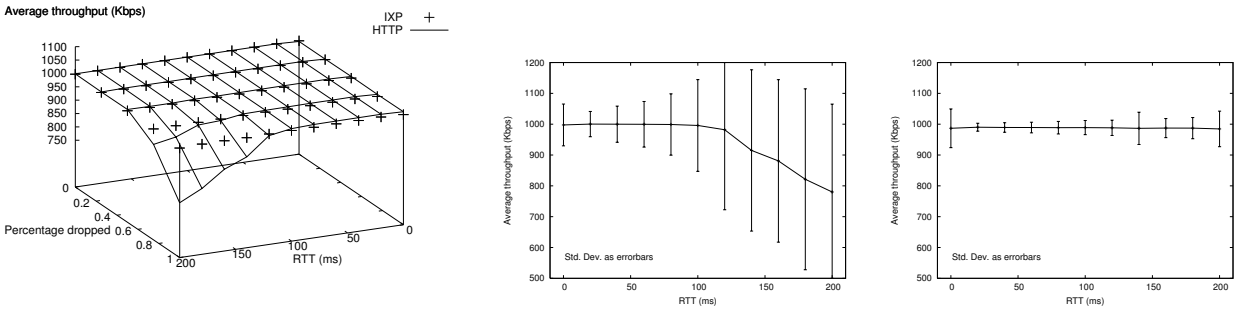
The transport protocol translation operation[1] is shown in figure 2. The protocol translation proxy uses the XScale core and one μEngine application block. In addition, we use two μEngines for the receiving (RX) and the sending (TX) blocks. Incoming packets are classified by the μEngine based on the header. RTSP and HTTP packets are enqueued for processing on the XScale core (control path) while the handling of RTP packets is performed on the μEngine (fast path). TCP acknowledgements with zero payload size are processed on the μEngine for performance reasons.

The main task of the XScale is to set up and maintain streaming sessions, but after the initialization, all video data is processed (translated and forwarded) by the μEngine. The proxy supports a partial TCP/IP implementation, covering only basic features. This is done to save both time and resources on the proxy.

To be fair with competing TCP streams, we implemented congestion control for the client loss experiment. TFRC [2] computation is used to determine the bandwidth available for streaming from the server. TFRC is a specification for best effort flows competing for bandwidth, designed to be reasonable fair to other TCP flows. The outgoing bandwidth is limited by the following formula:

$$X = \frac{s}{R * \sqrt{2 * b * \frac{p}{3}} + (t_{RTO} * 3 * \sqrt{3 * b * \frac{p}{8}} * p * (1 + 32 * p^2))}$$

[1]Our proxy also performs proxying of normal RTSP sessions and transparent load balancing between streaming servers, but this is outside of the scope of this paper. We also have unused resources (μEngines) enabling more functionality.



(a) HTTP and translation results (b) HTTP streaming (c) Protocol translation

Figure 3: Achieved bandwidth varying drop rate and link latency with 1% server-proxy loss

where X is the transmit rate in bytes per second, s is the packet size in bytes, R is the RTT in seconds, b is the number of packets ACKed by a single TCP acknowledgment, p is the loss event rate (0-1.0), and t_{RTO} is the TCP retransmission timeout. The formula is calculated on a μEngine using fixed point arithmetic. Packets arriving at a rate exceeding the TFRC calculated threshold are dropped.

We are aware that this kind of dropping has different effects on the user-perceived quality than sender-side adaptation. We have only made preliminary investigations on the matter and leave it for future work. In that investigation, we will also consider the effect of buffering for at most 1 RTT.

5. EXPERIMENTS AND RESULTS

We investigated the performance of our protocol translation proxy compared to plain HTTP-streaming in two different settings. In the first experiment, we induced unreliable network behavior between the streaming server and the proxy, while in the second experiment, the unreliable network connected proxy and client. We performed several experiments where we examined both the bandwidth and the delay while changing both the link delays (0 - 200 ms) and the packet drop rate (0 - 1 %). We used a web-server and an RTSP video server using RTP streaming, running on a standard Linux machine. Packets belonging to end-to-end HTTP connections made to port 8080 were forwarded by the proxy whereas packets belonging to sessions initiated by connection made to port 80 were translated. The bandwidth was measured on the client by monitoring the packet stream with tcpdump.

5.1 Server-Proxy Losses

The results from the test where we introduced loss and delay between server and proxy are shown in figure 3. Figure 3(a) shows a 3D plot where we look at the latency that we achieved for the different combinations of loss and link delays. Additionally, figures 3(b) and 3(c) show the respective results for the HTTP and protocol translation scenarios when keeping the loss rate constant at 1% (keeping the link delay constant gives similar results). The plots show that our proxy that translates transparently from RTP/UDP to TCP achieves a mostly constant rate for the delivered stream. Sending the HTTP stream from the server, on the other hand, shows large performance drops when the loss rate and the link delay increase. From figures 3(b) and 3(c), we see also that the translation provides a smoother stream whereas the bandwidth oscillates heavily using TCP end-to-end.

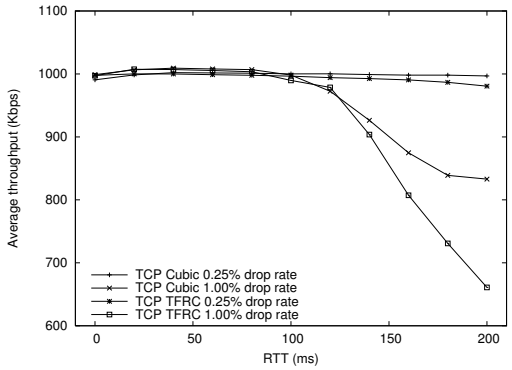


Figure 4: TCP cubic congestion vs. TFRC

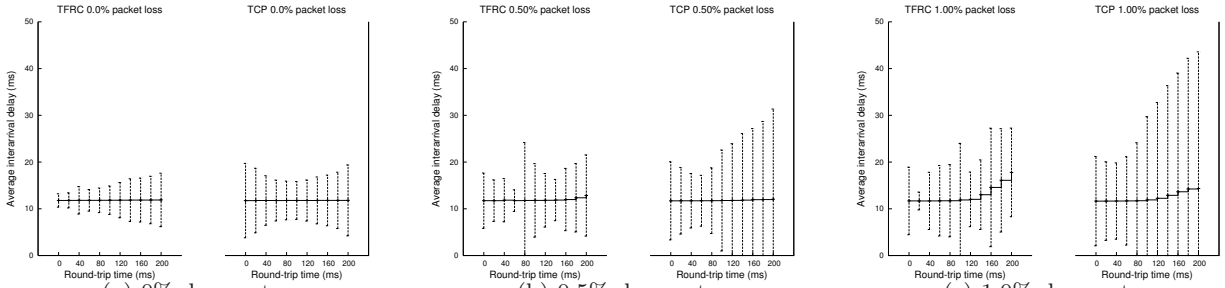
5.2 Proxy-Client Losses

In the second experiment, loss and delay are introduced between the proxy and the client, and the data rate is limited according to TFRC measuring RTT and packet loss during the transfer. Furthermore, packets are not buffered on the network card, meaning that the traffic exceeding the calculated rate of TFRC are dropped and that TCP retransmissions contains only data with zero values.

In figure 4, we first show the average throughput of streaming video from a web-server using cubic TCP congestion control compared with our TCP implementation using TFRC. As expected, the TFRC implementation behaves similar (fair) to normal TCP congestion control with a slightly more pessimistic approach. Moreover, figure 5 is a plot of the received packets' interarrival time. This shows that the delay variation of normal TCP congestion control increases with the drop rate, while TFRC is less affected. Thus, we see again that that our proxy gives a stream without large variations whereas the bandwidth oscillates heavily using TCP throughout the path.

6. DISCUSSION

Even though our proxy seems to give better, more stable bandwidths, there is a trade-off, because instead of retransmitting lost packet (and thus old data if the client does not buffer), the proxy fills the new packet with new updated data from the server. This means that the client in our pro-



(a) 0% drop rate (b) 0.5% drop rate (c) 1.0% drop rate

Figure 5: Average interarrival delay and variation with proxy-client loss

totype does not receive all data, and some artifacts may be displayed. On the other hand, in case of live and interactive streaming scenarios, delays due to retransmission may introduce dropped frames and delayed play out. This can cause video artifacts, depending on the codec used. However, this problem can easily be reduced by adding a limited buffer per stream sufficient for one retransmission on the proxy.

One issue in the context of proxies is where and how it should be implemented. For this study, we have chosen the IXP2400 platform as we earlier have explored the offloading capabilities of such programmable network processors. Using such an architecture, the network processor is suited for many similar operations, and the host computer could manage the caching and persistent storage of highly popular data served from the proxy itself. However, the idea itself could also be implemented as a user-level proxy application or integrated into the kernel of an intermediate node performing packet forwarding.

The main advantage of the scheme proposed in this paper is a lower variation in bandwidth and interarrival times in an unreliable network compared to normal TCP. It also combines some of the benefits of HTTP streaming (firewall traversal, client player support) with the performance of RTP streaming. The price of this is uncontrolled loss of data packets that may impact the perceived video quality more strongly than hiccups.

HTTP streaming may perform well in a scenario where a stored multimedia object is streamed to a high capacity end-system. Here, a large buffer may add a small, but acceptable, delay to conceal losses and oscillating resource availability. However, in the case where the receiver is a small device like a mobile phone or a PDA with a limited amount of resources, or in an interactive scenario like conferencing applications where there is no time to buffer, our protocol translation mechanisms could be very useful.

The server-proxy losses test can be related to a case where the camera on a mobile phone is used for streaming. Mobile devices are usually connected to unreliable networks with high RTT. The proxy-client losses test can be related to a traditional video conference scenario.

In the experiment, we compare a normal web-server streaming video with a RTP server (komssys) to a client by encapsulating the video data in HTTP packets on a IXP network card close to the video server. The former setup runs a simple web-server on Linux, limiting the average bandwidth from user-space to the video's bit rate.

Using RTP/UDP from the server through the backbone to a proxy is also an advantage for the resource utilization. RTP/UDP packets reduce memory usage, CPU usage and overhead in the network compared to TCP. This combined with the possibility of sending a single RTP/UDP stream to the proxy, and make the proxy do separation and adaptation of the stream to each client can reduce the load in the backbone. Therefore the proxy should be placed as close to the clients as possible, e.g. in the ISP's access network, or in a mobile provider's network.

7. CONCLUSION

Both TCP and UDP have their strengths and weaknesses. In this paper, we use a proxy that performs transparent protocol translation to utilize the strengths of both protocols in a streaming scenario. It enables the server to use UDP on the server side and TCP on the client side. The server gains scalability by not having to deal with TCP processing. On the client side, the TCP stream is not discarded and passes through firewalls. The experimental results show that our protocol transparent proxy achieves translation and delivers smoother streaming than HTTP-streaming.

8. REFERENCES

[1] Griwodz, C., Fiksdal, S., and Halvorsen, P. Translating scalable video streams from wide-area to access networks. *Campus Wide Information Systems 21*, 5 (2004), 205–210.

[2] Handley, M., Floyd, S., Padhye, J., and Widmer, J. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448 (Proposed Standard), Jan. 2003.

[3] Krasic, B., and Walpole, J. Priority-progress streaming for quality-adaptive multimedia. In *Proceedings of the ACM Multimedia Doctoral Symposium* (Oct. 2001).

[4] Krasic, C., Walpole, J., and Feng, W.-C. Quality-adaptive media streaming by priority drop. In *Proceedings of the International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (2003), pp. 112–121.

[5] Intel Corporation. Intel IXP2400 network processor datasheet, Feb. 2004.

[6] Parnes, P., Synnes, K., and Schefström, D. Lightweight application level multicast tunneling using mtunnel. *Computer Communication 21*, 515 (1998), 1295–1301.

[7] Widmer, J., Denda, R., and Mauve, M. A survey on TCP-friendly congestion control. *Special Issue of the IEEE Network Magazine "Control of Best Effort Traffic" 15* (Feb. 2001), 28–37.