DYNAMIC ADAPTIVE STREAMING OVER HTTP/2.0

Christopher Mueller, Stefan Lederer, Christian Timmerer, and Hermann Hellwagner
Alpen-Adria-Universität Klagenfurt
Universitätsstraße 65-67
9020 Klagenfurt am Wörthersee, Austria
{firstname.lastname}@itec.aau.at

ABSTRACT

MPEG Dynamic Adaptive Streaming over HTTP (DASH) is a new streaming standard that has been recently ratified as an international standard (IS). In comparison to other streaming systems, e.g., HTTP progressive download, DASH is able to handle varying bandwidth conditions providing smooth streaming. Furthermore, it enables NAT and Firewall traversal, flexible and scalable deployment as well as reduced infrastructure costs due to the reuse of existing Internet infrastructure components, e.g., proxies, caches, and Content Distribution Networks (CDN). Recently, the Hypertext Transfer Protocol Bis (httpbis) working group of the IETF has officially started the development of HTTP 2.0. Initially three major proposals have been submitted to the IETF i.e., Googles' SPDY, Microsofts' HTTP Speed+Mobility and Network-Friendly HTTP Upgrade, but SPDY has been chosen as working draft for HTTP 2.0. In this paper we implemented MPEG-DASH over HTTP 2.0 (i.e., SPDY), demonstrating its potential benefits and drawbacks. Moreover, several experimental evaluations have been performed that compare HTTP 2.0 with HTTP 1.1 and HTTP 1.0 in the context of DASH. In particular, the protocol overhead, the performance for different round trip times, and DASH with HTTP 2.0 in a lab test scenario has been evaluated in detail.

Index Terms – MPEG-DASH, HTTP 2.0, SPDY, Dynamic Adaptive Streaming over HTTP, Evaluation.

1. INTRODUCTION

The Hypertext Transfer Protocol (HTTP) is currently one of most used protocols on the application layer as shown in [1]. In particular, real-time entertainment is one of the major drivers and currently accounting for more than 50% of the Internet traffic in North Americas' fixed access networks, where Netflix alone is accounting for more than 30%. Considering that Netflix – and also others – uses HTTP on the application layer for their multimedia streaming system and that other HTTP traffic, i.e., Web browsing (16.59%), YouTube (9.9%), and Facebook (1.84%), are accounting for more than 28% of the Internet traffic, it implies that HTTP is the major protocol in modern networks (see also [2]).

Although HTTP has been initially designed for best effort file transfer, its flexible architecture does not prevent the advanced use cases of modern networks, i.e., multimedia streaming, rich Internet services, etc. However, there is room for improvements as many services are still using HTTP 1.0 [3] which has been specified in May 1996 or are not utilizing all features of HTTP 1.1 [4], such as persistent TCP connections or pipelining of HTTP requests. For example, already deployed HTTP streaming solutions such as Microsoft Smooth Streaming, Adobe Dynamic Streaming, and Apple HTTP Live Streaming like shown in [5] do not use HTTP 1.1 pipelining which could definitely increase the streaming performance. Unfortunately, there are not many proxies that support HTTP 1.1 pipelining due to the Head-of-Line blocking problem [6] where one pending response could possibly delay a range of other responses. The problem is that the proxy has to send in-order responses, which means that earlier arriving out-of-order responses will be blocked until the first (Head-of-Line) response arrives.

In order to overcome these shortcomings the IETFs' Hypertext Transfer Protocol Bis (httpbis) working group has recently started the development process of HTTP 2.0. Three proposals have been submitted to the IETF [7] where Google's SPDY proposal [8] has been chosen as working draft for HTTP 2.0. Moreover, several major companies, e.g., Facebook, Twitter, Akamai, Mozilla, and obviously Google itself support SPDY already or have announced that they will support it in the near future.

SPDY is already used for all Google Web services when users connect with the Google Chrome or Mozilla Firefox browser, which implies that it will be already heavily tested and definitely further improved due to the results of these tests. SPDY could be used as session layer between HTTP and TCP to multiplex multiple HTTP requests over one single TCP connection while requiring no or little changes from the application layer. The major benefits of SPDY are that it offers multiplexing over one TCP connection, prioritized requests, compressed headers, and the option for the server to push resources to the client.

In this paper, we have combined our open source available MPEG-DASH [9] implementation [10], which could handle HTTP 1.1 persistent connections with pipelining and HTTP 1.0 connections, with SPDY as well as SSL-encrypted SPDY and evaluated that solution under restricted conditions. The paper experimentally evaluates the overhead, bandwidth utilization, and streaming performance of HTTP 1.0, HTTP 1.1, and SPDY. Please note that in this paper HTTP 2.0 is synonymously used with SPDY.

The remainder of this paper is organized as follows. Section 3 describes the SPDY protocol and its behavior in detail. Section 4 describes our test-bed and the integration of SPDY into the MPEG-DASH client that has been used for all our experiments. The evaluation results are detailed in Section 5 and Section 6 concludes the paper including further work.

2. RELATED WORK

So far, we have not found any scientific related work, but there are different reports, which evaluate the performance of the SPDY protocol. [11] compares the performance of HTTP and SPDY

Acknowledgments: This work was supported in part by the EC in the context of the ALICANTE (FP7-ICT-248652), SocialSensor (FP7-ICT-287975), and QUALINET (COST IC 1003) projects and partly performed in the Lakeside Labs research cluster at AAU.

focusing on regular web traffic and page load times. They show that the advanced features of HTTP/1.1, such as pipelining and persistent connection, can bring HTTP close to the performance of SPDY. However, [11] concentrates on regular web traffic as well as small objects, and not on multimedia streaming or adaptive multimedia streaming. [12] shows the performance of SPDY in mobile networks focusing on page load times and regular web browsing traffic. The results show that it can significantly reduce page load times compared to HTTP but it is not clear which features of HTTP have been enabled. It seems that [12] compares SPDY with HTTP/1.0 in mobile networks, where the advantages of SDPY, such as persistent connections enable a significant performance improvement.

3. SPDY PROTOCOL

This section describes the SPDY protocol and its communication scheme in detail. The protocol is based on TCP and maintains a single connection for each session, in comparison to HTTP, which does not mandate persistent connections. During a session multiple streams can be opened between the client and the server in full-duplex mode. Typically, only one SPDY connection between a server and a client exists until the client navigates to another server. The servers should leave connections open as long as possible until a given threshold timeout or when a client initiates a connection close.

SPDY is fully compatible with HTTP and could be integrated as a session layer between HTTP and TCP. The HTTP request will be mapped into a SPDY stream and vice versa for the HTTP response headers. Additionally, it is also possible to send multiple requests in parallel to support HTTP 1.1 pipelining. Therefore, SPDY offers an interface for HTTP which simplifies its integration for already existing HTTP applications. After this handover from HTTP to SPDY the whole communication will be handled on the SPDY framing layer until a response arrives which will be passed to the HTTP layer. The network communication that takes place on the SPDY layer will be further described in the next section.

3.1 Framing

The SPDY network communication is based on frames that are exchanged between the client and the server based on a TCP connection. This framing layer comprises two frame types, i.e., the control frame depicted in Figure 1 and the data frame depicted in Figure 2. Each frame has a common header of exactly 8 bytes, which has been designed to simplify the parsing and minimize the framing overhead. The server or client can easily distinguish between control and data frames from the first bit named control bit, which is depicted in Figure 1 and Figure 2 on the top left corner with a “C”. The control bit is always one for control frames and zero for data frames. The 15 bits version field of the control frame indicates the used SPDY version, which is SPDY version 2 for all of our experiments. The type field denotes the type of the control frame, which could be:

- *SYN_STREAM:* This frame allows the sender to create a stream between the sender and the receiver and it can also be used to send additional metadata or HTTP request headers that can be mapped into the payload section of this frame.
- *SYN_REPLY:* This frame will be sent when the receiver of a *SYN_STREAM* frame accepts the stream creation from the sender. It typically contains also the response metadata or the HTTP response headers mapped into the payload section of this frame.
- *RST_STREAM:* This frame will be used to terminate the session. When the initiator of the stream sends this frame, it indicates that the session should be canceled. When the receiver of a stream sends this frame, it indicates that an error has occurred on the receiver side.
- *SETTINGS:* The settings frame contains a set of id-value pairs that could be used to configure the stream parameters. Generally this frame could be sent at any time either from the server or the client to signal, e.g., the available upload or download bandwidth, the round trip time, the allowed maximum concurrent streams etc.
- *NOOP:* This frame is the no-operation frame. When the server or the client receives this frame, it could simply throw away this frame.
- *PING:* This frame could be sent either from the server or the client to measure the round trip time. The receiver of this frame should send an identical frame to the sender as soon as possible.
- *GOAWAY:* This frame can be sent from the server or the client and tells the receiving endpoint that it should no longer use this connection for further communication. This mechanism enables a proper shutdown of the connection. Obviously a race condition will be introduced between the client and the server, due to this scheme. Hence, it contains a stream identifier, which should identify the last stream that will be accepted by the endpoint that has sent the *GOAWAY* frame. Streams with higher stream identifiers than the identifier of the *GOAWAY* frame will be canceled.
- *HEADERS:* This frame could be optionally sent on a stream at any time to modify already sent headers or to add new header fields.

In comparison to the control frame, the data frame contains a *Stream-ID* field instead of the *Version* and *Type* field of the control frame. The Stream-ID field simply identifies the stream that the data frame belongs to. This is very important because multiple streams can be used in parallel. Therefore, it is needed to separate between data of concurrent streams.

The *Flags* field of the control frame and data frame has 8 bits and is in case of the control frame dependent on the individual frame type and in case of the data frame it could only signal the end of the stream so that no additional round trip will be introduced to signal the end of a stream.

3.2 Streams

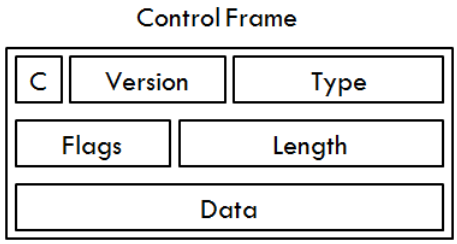


Figure 1. SPDY Control Frame

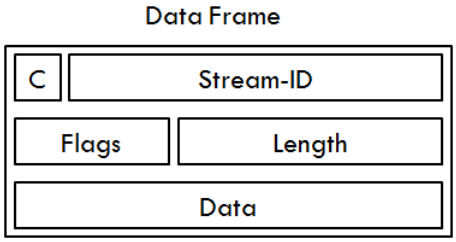


Figure 2. SPDY Data Frame

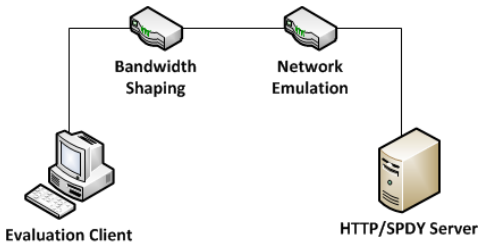


Figure 3. Experimental Setup

SPDY streams are sequences of frames that can be created either by the server or by the client. The streams are bidirectional, i.e., the server and the client can simultaneously send data. When using HTTP over SPDY one stream will be opened for each HTTP request and the stream will not be used for any further requests. However, this does not influence the performance as streams can be created on an established connection without an additional round trip. In general, streams will be created with a *SYN_STREAM* control frame that contains the HTTP headers that will be mapped to *SYN_STREAM* frame name-value pairs in the payload section of this frame.

The Stream-ID of the *SYN_STREAM* depends on the initiator of the stream, i.e., streams opened by the server contain even Stream-IDs and streams opened by the client contain odd Stream-IDs. Subsequent opened streams must follow this scheme, which means that all further client-initiated streams have to contain odd Stream-IDs and vice versa for server streams. Typically streams are bidirectional, but the stream creator could configure the stream in unidirectional mode with a flag in the *SYN_STREAM* frame.

The receiver of a *SYN_STREAM* should immediately respond with a *SYN_REPLY* when it accepts the stream or with a *RST_STREAM* to cancel the stream request. After this stream negotiation, data frames will be exchanged until one frame contains a *FLAG_FIN*.

4. TEST-BED

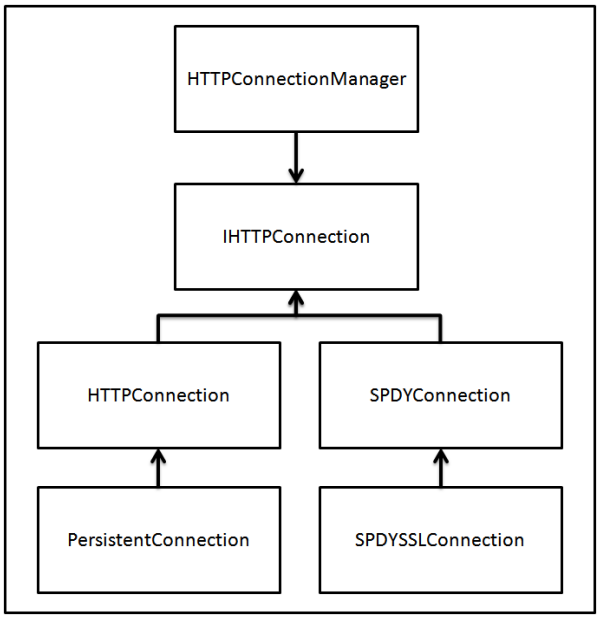


Figure 4. MPEG-DASH Plugin Extension

4.1 Experimental Setup

This section describes our test-bed. We have consistently used the same content for all of our experiments from the dataset of the University Klagenfurt [13]. The content has been encoded with x264 at 14 different bitrates (100, 200, 350, 500, 700, 900, 1100, 1300, 1600, 1900, 2300, 2800, 3400, and 4500 kbps) with a GOP (Group of Pictures) size of 48 to enable a 2 seconds segmentation. Our test network is depicted in Figure 3 and consists of four nodes, i.e., Evaluation Client, Bandwidth Shaping, Network Emulation, and HTTP/SPDY Server, which are all based on Ubuntu Linux 12.04. Moreover, all nodes have similar hardware to provide a homogenous test-bed that enables an objective comparison. The Bandwidth Shaping node depicted in Figure 3 is responsible for the bandwidth restriction in the network which will be configured with Linux Traffic control (tc) and the Hierarchal Token Bucket (htb) system. The Network Emulation node has been used to configure the RTT for our experiments with Linux netem. The HTTP/SPDY server component hosts a common Apache Web server which has been extended for the SPDY experiments with the mod_spdy plugin [14]. Although SSL is mandatory for SPDY, it is possible to disable it within mod_spdy.

4.2 MPEG-DASH SPDY Client

On the client side we have extended the open source available MPEG-DASH VLC plugin [10] with the most complete SPDY library, spdylay [15]. The architecture of the plugin is quite flexible which simplified the integration of SPDY and SSL-encrypted SPDY connections.

Figure 4 shows the simplified architecture of the network part of the MPEG-DASH VLC plugin. We have extended this part with the *SPDYConnection* and the *SPDYSSLConnection*, which are utilizing the spdylay library. As the *HTTPConnectionManager* consistently uses the *IHTTPConnection* interface, no changes outside of this part are needed. Therefore, it is possible to objectively compare the performance of the MPEG-DASH client with different network protocols, i.e., HTTP 1.0, HTTP 1.1, SPDY, and SPDY with SSL encryption while maintaining the same behavior of the adaptation logic, buffer, etc.

5. EXPERIMENTAL EVALUATION

This section provides the evaluation of the SPDY protocol for MPEG-DASH, focusing on different critical variables such as overhead, RTT, and performance. The system has been tested under laboratory test scenarios and all experiments have been consistently performed with the same content and the test-bed as described in Section 4.

5.1 Overhead Analysis

This section theoretically analyses the overhead of the protocols below SPDY, i.e., TCP, IP and Ethernet. Therefore it provides the theoretical lower bound on top of which SPDY will add an additional overhead. Beside that, the entire overhead of SPDY and HTTP-based MPEG-DASH streaming has been practically evaluated with our test-bed and the results will be described in the following section.

SPDY is using TCP on the transport layer and IP on the network layer. This introduces an overhead of 20 bytes for the TCP header [16] (+ additional 12 bytes for the optional header fields) and another 20 bytes for the IP header [17]. As Ethernet [18] is used on the link layer, an additional 14-byte frame header is added to the TCP/IP packets. Ethernet restricts the Maximum

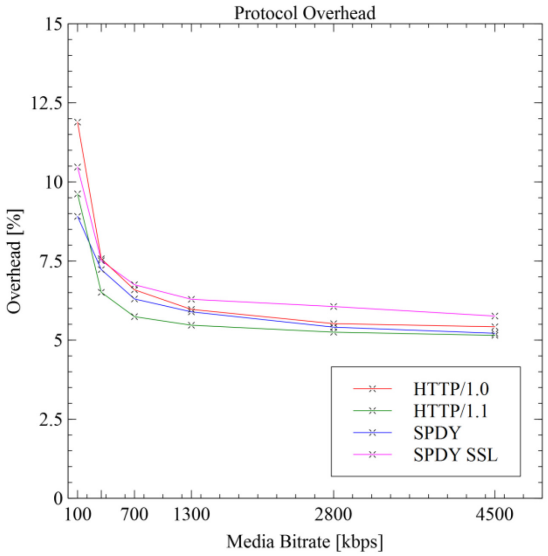


Figure 5. Protocol Overhead

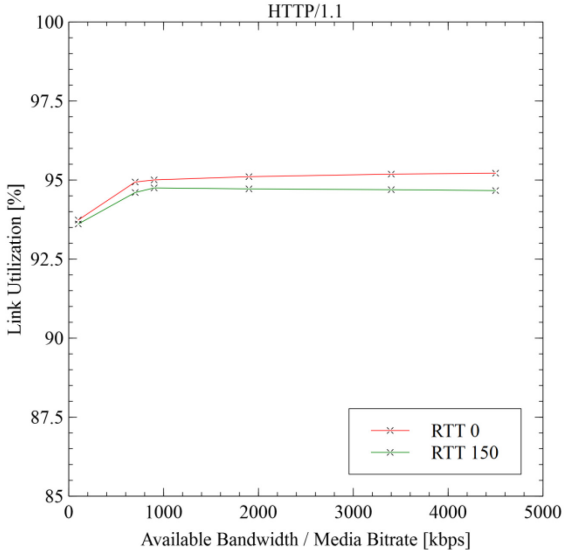


Figure 7. Link Utilization under HTTP 1.1

Transportation Unit (MTU) to 1500 bytes. The lower bound of the TCP/IP protocol overhead can be calculated: considering the resulting maximum payload of the TCP packet of 1448 bytes and the Ethernet frame size of 1514 (incl. Ethernet frame header), this results in an overhead of 4.56% caused by these headers. In addition to this, one has to consider packets needed for TCP connection establishment and ACKs, as well as other Ethernet overhead like check sequence etc.

On top of TCP, SPDY introduces further overhead due to the framing. In the following evaluations, the protocol overhead produced by HTTP as well as SPDY is investigated in practice to give a comparison to the calculated lower bound.

5.2 Overhead Evaluation

The overhead evaluation has been performed with several quality levels (media bitrates), i.e., 100, 350, 700, 1300, 2800, and 4500 kbps as depicted in Figure 5. We have tested all protocols, i.e., HTTP 1.0, HTTP 1.1, SPDY, and SPDY with SSL encryption within our test-bed (Figure 3). For this experiment, the bandwidth shaping component as well as the network emulation component have been disabled, so that no other variables are influencing the experiment. The quality level has been fixed for each individual experiment. Hence, the adaption process does not affect the experiment.

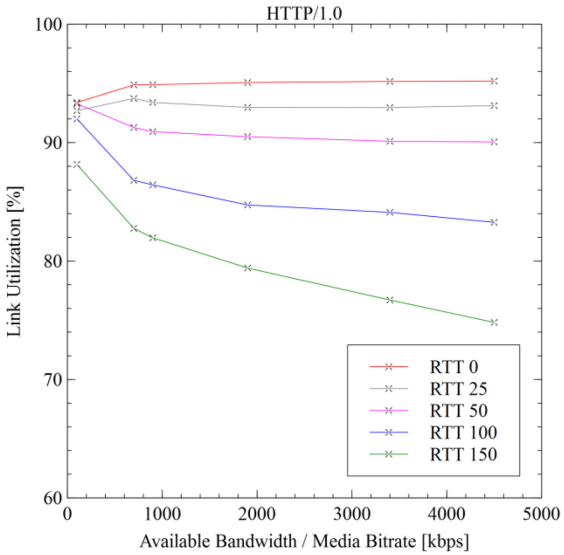


Figure 6. Link Utilization under HTTP 1.0

The overall throughput has been calculated from the network statistics of ifconfig and the effective media throughput (payload) has been measured with the MPEG-DASH VLC plugin. The overhead will then be calculated as follows:

$$\text{overhead} = 1 - \frac{\text{media throughput}}{\text{overall throughput}} \qquad (1)$$

Moreover, each of the individual experiments, e.g., 100 kbps quality level, has been performed 3 times and an average has been calculated that is depicted in Figure 5. The figure shows the overhead for the effective media bitrate as described in Equation (1).

The graph shows that SPDY has a slightly higher overhead compared to HTTP 1.1 due to the framing layer. Only for the 100 kbps media bitrate, i.e., quality level, SPDY is more efficient thanks to the header compression and small payload. For all other quality levels, the header compression is good but it could not compensate the overhead that gets introduced as a consequence of the framing. Nevertheless, the difference is rather small and the current server implementation could only send data packets with a maximum payload of 4096 bytes or less. Moreover, most of the time the payload of the data packets is smaller which further decreases the efficiency. However, this is a tradeoff because when data is available it should be sent as soon as possible, even when it is not enough to efficiently fill a data frame, otherwise a significant delay would be introduced that is much more critical than the overhead incurred.

SPDY with SSL encryption is less efficient than the other protocols, but due to the header compression it could also outperform HTTP 1.0 in the case of the 100 kbps quality level. For all other experiments, its overhead is higher than that of HTTP 1.0. Furthermore, it introduces a computational effort on the server and on the client which could probably negatively influence the packing of the data frames due to the delay that gets introduced.

5.3 Link Utilization Evaluation

Several experiments have been performed to evaluate the performance of HTTP 1.0 and HTTP 1.1 compared to SPDY and SPDY with SSL encryption under restricted bandwidth conditions

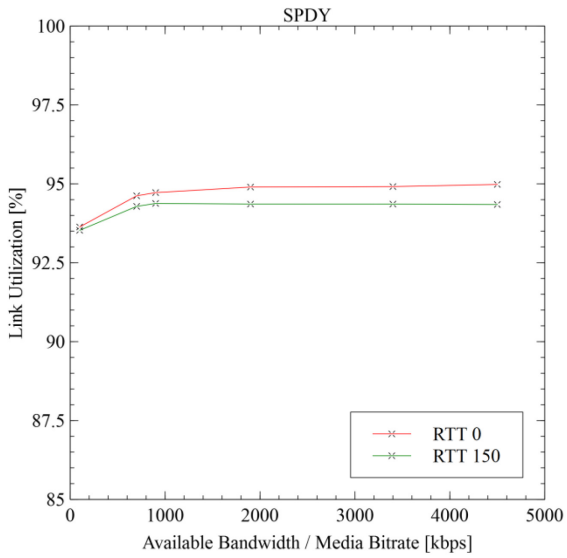


Figure 8. Link Utilization under SPDY

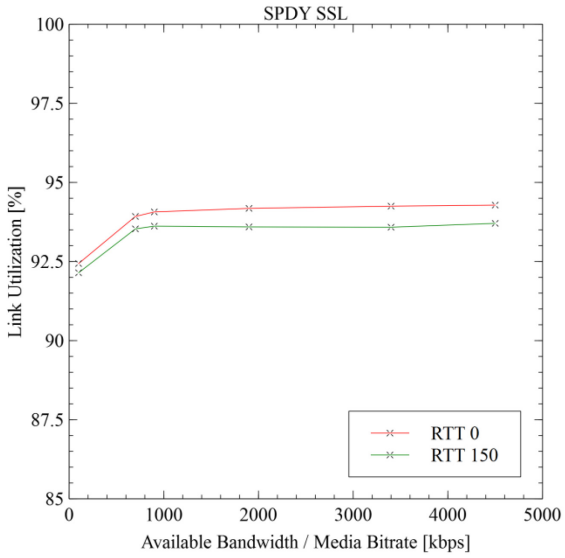


Figure 10. Link Utilization under SPDY SSL

and round trip times (RTT) with different quality levels. We have used 6 different quality levels (100, 700, 900, 1900, 3400, and 4500 kbps) for each RTT experiment. The RTT is ranging from 0 to 150 ms: 0 to 25 ms is typical for local area networks, 50 to 100 ms is typical for fixed line Internet connections, and 150 ms is typical for mobile networks.

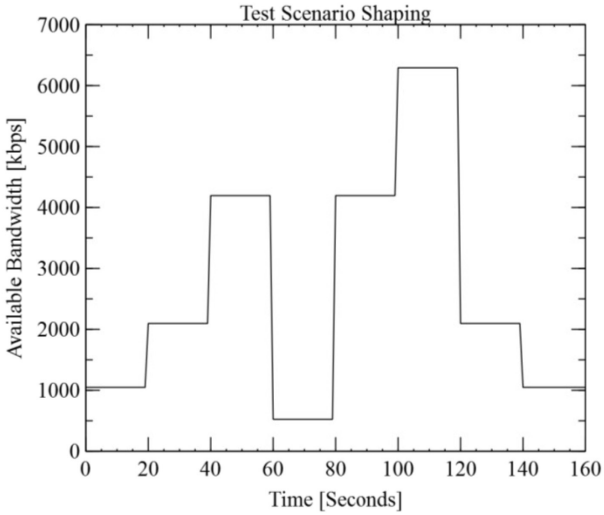
Figure 6 shows the link utilization of HTTP 1.0 under different bandwidth conditions and RTT configurations. The vertical axis shows the link utilization in percent and the horizontal axis shows the restricted bandwidth in kbps. For each experiment, the bandwidth has been restricted to the media bitrate under consideration. The link utilization has been calculated from the effective media throughput, which has been measured with the MPEG-DASH VLC plugin, and the available bandwidth that has been restricted with the Bandwidth Shaping component of our test-bed, as follows:

$$\text{link utilization} = \frac{\text{media throughput}}{\text{available bandwidth}} \qquad (2)$$

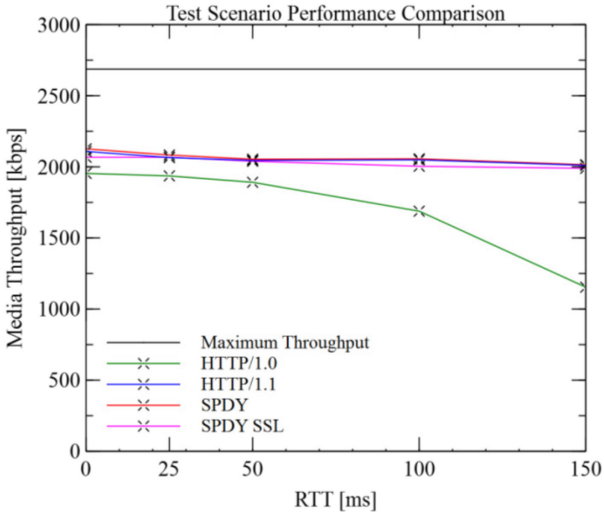
In case of a low RTT, HTTP 1.0 performs more or less well but with a higher RTT, i.e., 100 ms and 150 ms, the link utilization is very low especially for high bandwidths and quality levels, e.g., 4500 kbps quality level with 4500 kbps bandwidth restriction. The reason for that is that HTTP 1.0 opens one TCP connection per segment request. Due to TCP slow start, it is not possible to utilize the maximum available bandwidth even with the increased initial congestion window on the server and the increased TCP receive window on the client of Ubuntu Linux 12.04 which is 10 times of the MSS.

It is obvious that the link utilization would be worse with a low quality level, e.g., 100 kbps, and a high available bandwidth, e.g., 4500 kbps, due to the smaller size of the segments compared to the header overhead of HTTP 1.0, but it is not common that such a low quality level will be used for high bandwidths. Therefore, we have used high quality levels with high bandwidths and low quality levels with low bandwidths.

Figure 7 shows the performance of HTTP 1.1 for RTT = 0 ms and RTT = 150 ms. For the sake of simplicity, we have omitted the intermediate RTTs due to the small differences. The difference of the link utilization between these two RTT levels is rather small



(a)



(b)

Figure 9. Test Scenario

compared to HTTP 1.0 as the implementation takes advantage of the persistent connection and pipelining features of HTTP 1.1. This means that for each experiment only a single TCP connection will be used and TCP slow start is only influencing the performance at the beginning of the session and not at the beginning of every segment.

Furthermore, we have also evaluated SPDY shown in Figure 8 and SPDY with encryption shown in Figure 9. SPDY without encryption performs equally well like HTTP 1.1 because all streams will be multiplexed over a single TCP connection. SPDY with SSL encryption has a lower link utilization compared to SPDY and HTTP 1.1. Nevertheless, it is obviously also robust against high RTTs due to the single persistent TCP connection.

5.4 Laboratory Scenario

We have also evaluated all solutions with the VLC MPEG-DASH plugin with a pre-defined bandwidth trace as depicted in Figure 10(a). The vertical axis describes the available bandwidth in kbps and the horizontal axis describes the time in seconds. Each experiment lasts exactly 160 seconds and the available bandwidth which is available during the experiment ranges from 1 Mbps to 6 Mbps. The content set provides 14 different media qualities ranging from 100 kbps to 4500 kbps which the client could individually choose at segment boundaries. All experiments have been performed with the same adaptation logic that is based on the buffer fill state and the measured throughput of the last segment. The buffer has been restricted for all experiments to 40 seconds. Each solution has been tested several times with RTT = 0, 25, 50, 100, and 150 ms and the average of each experiment is depicted in Figure 10(b). The vertical axis of Figure 10(b) shows the media throughput in kbps. It has been measured with the VLC MPEG-DASH plugin. The average maximum throughput has been calculated from the pre-defined bandwidth trace in Figure 10(a) which could be seen as the maximum achievable throughput. This line depicts the maximal achievable throughput without any overhead and optimal adaptation decisions, which is the upper bound for all transferring mechanisms in this laboratory setup.

Figure 10(b) shows that HTTP 1.1, SPDY, and SPDY with SSL encryption perform equally well and quite stable over all RTTs. As expected, HTTP 1.0 could not achieve the same media throughput especially for high RTTs due to the problems that have already been described in Section 5.3, i.e., one TCP connection per segment and TCP slow start.

6. CONCLUSIONS

In this paper we have described the working draft for HTTP 2.0, i.e., SPDY and its usage for DASH-based media streaming. The structure of the protocol as well as its behavior has been described in detail. Additionally, the protocol has been combined with the MPEG-DASH standard. Moreover, several experimental evaluations have been performed focusing on dynamic multimedia streaming based on SPDY as well as SPDY with SSL encryption. We have evaluated the overhead that gets introduced due to the framing of SPDY and shown that it is not as efficient as HTTP 1.1. Nevertheless, SPDY and SPDY with SSL encryption are very robust against increasing RTT because they are maintaining only one single TCP connection during the whole communication. Furthermore, we have also evaluated all solutions with an abstract test scenario where the SPDY solutions as well as HTTP 1.1 performed equally well. However SPDY implicitly solves the Head-of-Line blocking problem of HTTP 1.0 and due to the lack of proper adoption of HTTP 1.1 on caches it could definitely enhance the streaming performance of future networks.

As seen in our experiments, SPDY achieves very good results with MPEG-DASH when SSL encryption is disabled. Currently, SPDY mandates the SSL encryption but especially for multimedia streaming this is not necessary as the content would be DRM encrypted anyway. Furthermore, the additional SSL encryption would introduce additional computational overhead on both the server and the client without any need.

Our future work will include the evaluation of SPDY based multimedia streaming within real world mobile environments and use cases where multiple clients compete for a bottleneck.

7. REFERENCES

[1] Sandvine, “Global Internet Phenomena Report Fall 2011”, Sandvine Intelligent Broadband Networks, 2011.

[2] L. Popa, A. Ghodsi, and I. Stoica. 2010. “HTTP as the narrow waist of the Future Internet”. In Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX). ACM, New York, NY, USA

[3] T. Berners-Lee, R. Fielding, H. Frystyk, Hypertext Transfer Protocol -- HTTP/1.0, URL: http://www.ietf.org/rfc/rfc1945.txt (last access Dec. 2012).

[4] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee, Hypertext Transfer Protocol -- HTTP/1.1, URL: http://www.ietf.org/rfc/rfc2616.txt (last access: Dec. 2012).

[5] C. Mueller, S. Lederer, C. Timmerer, “An Evaluation of Dynamic Adaptive Streaming over HTTP in Vehicular Environments,” *In Proc. of the 4th Workshop on Mobile Video (MoVid12)*, Feb. 2012.

[6] K. J. Grinnemo, T. Andersson, A. Brunstrom, “Performance Benefits of avoiding head-of-line blocking in SCTP,” *In Proceedings of ICAS/ICNS*, 2005.

[7] HTTP 2.0 Call for Expression of Interest, http://trac.tools.ietf.org/wg/httpbis/trac/wiki/Http2CfI, (last access: Dec. 2012).

[8] M. Belshe, R. Peon, “SPDY Protocol”, http://tools.ietf.org/html/draft-mbelshe-httpbis-spdy-00 (last access: Dec. 2012).

[9] ISO/IEC DIS 23009-1.2, “Information Technology — Dynamic Adaptive Streaming over HTTP (DASH) — Part 1: Media Presentation Description and Segment Formats”. C. Mueller, C. Timmerer, “A VLC Media Player Plugin enabling Dynamic Adaptive Streaming over HTTP,” *In Proceedings of the ACM Multimedia 2011*, Scottsdale, Arizona, Nov. 2011.

[10] C. Mueller, C. Timmerer, “A VLC Media Player Plugin enabling Dynamic Adaptive Streaming over HTTP,” *In Proceedings of the ACM Multimedia 2011*, Scottsdale, Arizona, Nov. 2011.

[11] J. Padhye, F. Nielson, “A comparison of SPDY and HTTP performance”, 2012.

[12] M. Welsh, B. Greenstein, M. Piatek, “SPDY Performance on Mobile Networks”, https://developers.google.com/speed/articles/spdy-for-mobile, (last access: April 2013).

[13] S. Lederer, C. Mueller, C. Timmerer, “Dynamic Adaptive Streaming over HTTP Dataset”, *ACM Multimedia Systems*, Chapel Hill, North Carolina, USA, Feb. 2012.

[14] Apache mod_spdy module, http://code.google.com/p/mod-spdy/, (last access: Dec. 2012).

[15] Spdylay, http://spdylay.sourceforge.net/, (last access: Dec. 2012).

[16] Transmission control Protocol, RFC 793, URL: http://tools.ietf.org/html/rfc793(last access: Dec. 2012).

[17] Internet Protocol, RFC 791, URL: http://tools.ietf.org/html/rfc791(last access: Dec. 2012).

[18] IEEE 802.3-2800 Ethernet, URL: http://standards.ieee.org/about/get/802/802.3.html (last access: Dec. 2012).